VITAM - Architecture

Version 10.0.0-SNAPSHOT

VITAM

sept. 25, 2026

Table des matières

1 Introduction **1**
1.1 Objectif de ce document . . . 1
1.2 Structure du document . . . 1

2 Rappels **2**
2.1 Information concernant les licences . . . 2
2.2 Documents de référence . . . 2
2.2.1 Documents internes . . . 2
2.2.2 Référentiels externes . . . 3
2.3 Glossaire . . . 3

3 Vue d'ensemble **6**
3.1 Drivers du projet . . . 6
3.1.1 Enjeux . . . 6
3.1.2 Contraintes et objectifs . . . 7
3.1.3 Positionnement . . . 7
3.2 Interfaces externes du système . . . 8
3.2.1 Interfaces requises . . . 8
3.2.2 Interfaces métier exposées . . . 8
3.3 Orientations générales . . . 8
3.3.1 Open Source . . . 9
3.3.2 API REST . . . 9
3.3.3 Big Data et Cloud computing . . . 10
3.3.4 Cloud storage . . . 10
3.3.5 PCA/PRA et répartitions des travaux . . . 11
3.3.6 Sécurité des données additionnelle . . . 11
3.3.7 Architecture multi-tenants . . . 12
3.3.8 Solution exploitable . . . 12
3.4 Architecture fonctionnelle . . . 13

4 Architecture applicative **14**
4.1 Architecture applicative . . . 14
4.1.1 Drivers de l'architecture . . . 14
4.1.2 Services . . . 14
4.1.3 Détail des flux d'information métier . . . 17
4.1.4 Données métier . . . 17
4.2 Architecture des données & multisite . . . 17

4.2.1 Inventaire des données . . . 17
4.2.2 Stockage et stratégies . . . 19
4.2.3 Multisite . . . 20
4.2.4 Stratégies & multisite . . . 21
4.2.4.1 Mode standard : exemple d'architecture mono-stratégie . . . 22
4.2.4.2 Mode avancé : exemple d'architecture multi-stratégie orienté Qualité de service . . . 23
4.2.4.3 Mode avancé : exemple d'architecture multi-stratégie orienté Offres objets . . . 24
4.3 Services métiers . . . 26
4.3.1 API externes (ingest-external et access-external) . . . 26
4.3.2 Moteur d'entrée (ingest-internal) . . . 26
4.3.3 Moteur d'accès (access-internal) . . . 27
4.3.4 Gestion des droits & accès (security-internal) . . . 27
4.3.5 Moteur d'exécution (processing) . . . 27
4.3.6 Espace de travail (workspace) . . . 28
4.3.7 Worker (worker) . . . 28
4.3.8 Moteur de données (metadata) . . . 28
4.3.9 Moteur de journalisation (logbook) . . . 28
4.3.10 Gestion des référentiels (functional-administration) . . . 29
4.3.11 Moteur de stockage (storage) . . . 29
4.3.12 Offre de stockage (offer) . . . 29
4.3.13 Interface de démonstration (ihm-demo) . . . 30
4.3.14 Scheduler . . . 30
4.3.15 Connecteur générique . . . 30

5 Architecture technique / exploitation 31
5.1 Principes d'architecture technique . . . 31
5.1.1 Principes communs et environnement des services . . . 31
5.1.1.1 Principes relatifs aux composants délivrés . . . 31
5.1.1.1.1 Nommage . . . 31
5.1.1.1.2 Principes relatifs aux services VITAM . . . 32
5.1.1.1.3 Principes relatifs aux COTS . . . 33
5.1.2 Utilisateurs, dossiers & droits . . . 33
5.1.2.1 Utilisateurs et groupes d'exécution . . . 33
5.1.2.1.1 Groupes . . . 34
5.1.2.1.2 Utilisateurs . . . 34
5.1.2.2 Arborescence de fichiers . . . 34
5.1.2.2.1 Services VITAM . . . 34
5.1.2.2.1.1 Arborescence VITAM . . . 34
5.1.2.2.1.2 Intégration au système . . . 35
5.1.2.2.2 COTS . . . 35
5.1.3 Principes sur les communications inter-services et le clustering . . . 35
5.1.3.1 Clusters applicatifs métier . . . 35
5.1.3.1.1 Appels REST des services métier . . . 36
5.1.3.1.2 Workers . . . 36
5.1.3.2 COTS & clustering . . . 36
5.1.3.3 Annuaire de services (service registry) . . . 36
5.1.4 Packaging . . . 37
5.1.4.1 Principes communs . . . 37
5.1.4.2 Dépôts . . . 37
5.1.4.2.1 AlmaLinux . . . 38
5.1.4.2.2 Debian . . . 38
5.1.4.3 Prise en compte de la configuration dans le packaging . . . 38
5.1.4.3.1 AlmaLinux . . . 38
5.1.4.3.2 Debian . . . 39

5.1.5 Déploiement de la solution . . . 39
5.1.5.1 Principes de déploiement . . . 39
5.1.5.2 Contraintes et vue d'ensemble . . . 39
5.1.5.3 Installation initiale . . . 41
5.1.5.4 Principes de mise à jour à chaud . . . 41
5.1.5.5 Multi-site . . . 41
5.1.5.6 Support de l'élasticité . . . 42
5.1.5.7 Validation du déploiement . . . 42
5.1.6 Suivi de l'état du système . . . 42
5.1.6.1 API de supervision . . . 42
5.1.6.2 Métriques . . . 42
5.1.6.3 Logs . . . 43
5.1.6.3.1 Protocoles : syslog . . . 43
5.1.6.3.2 Types de log . . . 43
5.1.6.3.2.1 Logs applicatifs . . . 44
5.1.6.3.2.2 Logs du garbage collector Java . . . 44
5.1.6.3.2.3 Logs d'accès . . . 44
5.1.6.4 Suivi de l'état de déploiement . . . 44
5.1.6.5 Intégration à un système de monitoring tiers . . . 45
5.1.7 Administration technique . . . 45
5.1.7.1 Démarrage / arrêt des services . . . 45
5.1.7.2 Tâches régulières . . . 45
5.1.8 Gestion des données du système . . . 45
5.1.8.1 Cas des déploiements de petite taille . . . 45
5.1.8.1.1 Dossiers . . . 46
5.1.8.1.2 Sauvegarde . . . 46
5.1.8.2 Restauration . . . 46
5.2 Services techniques fournis par la solution . . . 46
5.2.1 Moteur de déploiement et de configuration . . . 46
5.2.2 Chaîne de traitement de logs et de métriques . . . 47
5.2.3 Service registry . . . 47
5.3 Composants logiciels utilisés . . . 47
5.3.1 Fournis . . . 48
5.3.1.1 COTS . . . 48
5.3.1.2 Bibliothèques structurantes . . . 48
5.3.2 Requis . . . 49
5.4 Architecture technique détaillée . . . 49
5.4.1 Flux métier . . . 50
5.4.2 Flux exploitation . . . 53
5.4.3 Flux techniques . . . 54
5.4.4 Découpage en zones . . . 55
5.5 Stockage des données . . . 56
5.5.1 Stratégies de stockage . . . 57
5.5.2 Offre filesystem . . . 57
5.5.3 Offre Swift . . . 58
5.5.4 Offre S3 . . . 59
5.5.5 Offre Tape-library . . . 64
5.6 Concentration et exploitation des logs applicatifs . . . 65
5.6.1 Besoins . . . 65
5.6.2 Modèle générique . . . 65
5.6.3 Choix des implémentations . . . 66
5.6.3.1 Émetteur de logs . . . 67
5.6.3.2 Agent de transport de log . . . 68
5.6.3.3 Concentration de logs . . . 68

5.6.3.4 Stockage des logs . . . 68
5.6.3.4.1 Gestion des index . . . 69
5.6.3.5 Visualisation des logs . . . 70
5.6.4 Intégration à un système de gestion de logs existants . . . 70
5.6.5 Limites . . . 70
5.7 Métriques applicatives . . . 70
5.7.1 Besoins . . . 70
5.7.2 Modèle générique . . . 71
5.7.3 Choix des implémentations . . . 71
5.7.3.1 Enregistreur de métriques . . . 71
5.7.3.2 Reporters de métriques . . . 71
5.7.3.3 Endpoint des métriques . . . 72
5.7.3.4 Stockage des métriques . . . 72
5.7.4 Limites . . . 72
5.8 Outillage de déploiement . . . 73
5.8.1 Outil . . . 73
5.8.2 Architecture de l'outil . . . 73
5.8.3 Gestion des secrets . . . 74
5.9 Service registry . . . 74
5.9.1 Architecture . . . 74
5.9.2 Résolution DNS . . . 75
5.9.3 Multi-site . . . 75
5.9.4 Packaging . . . 76
5.9.5 Monitoring . . . 76
5.10 Dépendances aux services d'infrastructures . . . 76
5.10.1 Ordonnanceurs techniques / batchs . . . 76
5.10.1.1 Curator . . . 76
5.10.1.2 Sécurisation des journaux d'opérations . . . 76
5.10.1.3 Sécurisation des journaux d'écriture . . . 76
5.10.1.4 Sécurisation des cycles de vie . . . 77
5.10.1.5 Cas de la sauvegarde . . . 77
5.10.2 Socles d'exécution . . . 77
5.10.2.1 Middlewares . . . 77
5.11 Composants déployés . . . 77
5.11.1 Access-external . . . 77
5.11.2 Access-internal . . . 77
5.11.3 Batch-report . . . 78
5.11.4 Collect-external . . . 78
5.11.5 Collect-internal . . . 78
5.11.6 Consul . . . 78
5.11.6.1 Architecture de déploiement . . . 79
5.11.7 Curator . . . 79
5.11.8 Elasticsearch-data . . . 80
5.11.8.1 Architecture de déploiement . . . 80
5.11.8.2 Monitoring . . . 80
5.11.9 Elasticsearch-log . . . 80
5.11.9.1 Architecture de déploiement . . . 81
5.11.10 Functional-administration . . . 81
5.11.11 Grafana . . . 81
5.11.11.1 Architecture de déploiement . . . 81
5.11.11.1.1 Ports utilisés . . . 81
5.11.12 Ingest-external . . . 81
5.11.12.1 Antivirus . . . 82
5.11.13 Ingest-internal . . . 82

5.11.14 Kibana . . . 82
5.11.14.1 Déploiement . . . 83
5.11.15 Logbook . . . 83
5.11.16 Logstash . . . 83
5.11.17 Metadata . . . 83
5.11.18 Metadata Collect . . . 84
5.11.19 Mongodb . . . 84
5.11.19.1 Base mongo-data . . . 84
5.11.19.2 Base mongo-offer . . . 84
5.11.19.3 Architecture de déploiement . . . 85
5.11.19.3.1 Architecture 1 noeud . . . 85
5.11.19.3.2 Architecture distribuée . . . 85
5.11.19.3.3 Ports utilisés . . . 86
5.11.20 Offer . . . 86
5.11.20.1 Types d'offre de stockage . . . 87
5.11.20.1.1 Cas des *containers* objet . . . 87
5.11.21 Processing . . . 87
5.11.22 Prometheus server . . . 87
5.11.22.1 Architecture de déploiement . . . 87
5.11.22.1.1 Ports utilisés . . . 87
5.11.23 Prometheus alertmanager . . . 88
5.11.23.1 Architecture de déploiement . . . 88
5.11.23.1.1 Ports utilisés . . . 88
5.11.24 Prometheus node_exporter . . . 88
5.11.24.1 Architecture de déploiement . . . 88
5.11.24.1.1 Ports utilisés . . . 88
5.11.24.1.2 API exposées . . . 88
5.11.25 Prometheus Elasticsearch Exporter . . . 88
5.11.25.1 Architecture de déploiement . . . 88
5.11.25.1.1 Ports utilisés . . . 89
5.11.25.1.2 API exposées . . . 89
5.11.26 restic . . . 89
5.11.26.1 Architecture de déploiement . . . 89
5.11.27 Scheduler . . . 89
5.11.28 Security-internal . . . 90
5.11.28.1 API d'administration . . . 90
5.11.29 Siegfried . . . 90
5.11.29.1 Mode de fonctionnement dans VITAM . . . 90
5.11.30 Storage . . . 91
5.11.31 Worker . . . 91
5.11.31.1 Particularités . . . 91
5.11.32 Workspace . . . 91
5.11.33 Workspace Collect . . . 92
5.12 Guidelines de déploiement . . . 92
5.13 Eléments de dimensionnement . . . 92
5.13.1 Compute . . . 93
5.13.1.1 « xsmall » : développement local . . . 93
5.13.1.2 « small » : recette simple métier . . . 95
5.13.1.3 « medium » : production pour volumétries moyennes . . . 97
5.13.1.4 « large » : production pour volumétries moyennes avec besoin de résilience . . . 99
5.13.1.5 « xlarge » : production pour fortes volumétries . . . 101
5.13.2 Stockage . . . 103
5.13.3 Réseau : inter-site . . . 105
5.13.4 Scalabilité . . . 105

5.14 Matrice des flux . . . 105
5.14.1 Matrice des flux intra-site . . . 105
5.14.2 Matrice des flux inter-site . . . 106

6 Sécurité **107**
6.1 Principes . . . 107
6.1.1 Principes de cloisonnement . . . 107
6.1.2 Principes de sécurisation des accès externes . . . 107
6.1.3 Principes de sécurisation des communications internes au système . . . 108
6.1.4 Principes de sécurisation des bases de données . . . 109
6.1.4.1 MongoDB . . . 109
6.1.4.2 Elasticsearch . . . 109
6.1.5 Principes de sécurisation des secrets de déploiement . . . 109
6.2 Liste des secrets . . . 110
6.3 Certificats . . . 110
6.4 SELinux . . . 111
6.5 Documentation de sécurité . . . 111

7 Architecture détaillée **112**
7.1 Access . . . 112
7.1.1 Généralités . . . 112
7.1.2 Architecture Technique . . . 112
7.1.2.1 Introduction . . . 112
7.1.2.1.1 Présentation . . . 112
7.1.2.1.2 Itération 4 . . . 112
7.1.2.1.3 Modules - packages . . . 113
7.1.2.2 Access-api . . . 113
7.1.2.2.1 Présentation . . . 113
7.1.2.3 Access-client . . . 113
7.1.2.4 Utilisation . . . 114
7.1.2.5 Le client . . . 114
7.1.2.6 Access-common . . . 114
7.1.2.6.1 Présentation . . . 114
7.1.2.7 Access-core . . . 115
7.1.2.7.1 Présentation . . . 115
7.1.2.7.2 Packages : . . . 115
7.1.2.7.2.1 Récupération d'un objet spécifique . . . 115
7.1.2.8 Access-rest . . . 116
7.1.2.8.1 Présentation . . . 116
7.1.2.8.2 Packages : . . . 116
7.1.2.8.3 fr.gouv.vitam.access.external.rest . . . 116
7.1.2.8.3.1 Rest API . . . 116
7.1.2.9 -AccessApplication.java . . . 116
7.1.2.10 -AccessResourceImpl.java . . . 117
7.1.2.11 -LogbookExternalResourceImpl.java . . . 119
7.1.2.12 -AdminManagementExternalResourceImpl.java . . . 121
7.2 Batch-report . . . 123
7.2.1 Généralités . . . 123
7.2.2 Architecture Technique . . . 123
7.2.2.1 Introduction . . . 123
7.2.2.1.1 Présentation . . . 123
7.2.2.1.2 Découpage du code . . . 123
7.2.2.2 batch-report-client . . . 123
7.2.2.3 batch-report-common . . . 124

7.2.2.4 Acbatch-report-rest . . . 124
7.3 Collect . . . 124
7.3.1 Généralités . . . 124
7.3.2 Architecture Technique . . . 124
7.3.2.1 Introduction . . . 124
7.3.2.1.1 Présentation . . . 124
7.3.2.1.2 Itération 4 . . . 124
7.3.2.1.3 Modules - packages . . . 124
7.3.2.2 collect-client . . . 125
7.3.2.3 Utilisation . . . 125
7.3.2.4 Le client . . . 125
7.3.2.5 collect-rest . . . 126
7.3.2.5.1 Présentation . . . 126
7.3.2.5.2 Packages : . . . 126
7.3.2.5.3 fr.gouv.vitam.collect.resource . . . 126
7.3.2.5.3.1 Rest API . . . 126
7.3.2.6 -TransactionResource.java . . . 126
7.4 Common . . . 128
7.4.1 Architecture Fonctionnelle . . . 128
7.4.1.1 Introduction . . . 128
7.4.1.1.1 But de cette documentation . . . 128
7.4.1.1.2 GUID . . . 128
7.4.1.1.3 ServerIdentity et Logger . . . 128
7.4.1.2 GUID . . . 128
7.4.1.2.1 Présentation de la problématique . . . 128
7.4.1.2.1.1 Qu'est ce qu'une URL pérenne ? . . . 128
7.4.1.2.1.2 Objectifs . . . 129
7.4.1.2.1.3 Préconisation E-ARK . . . 129
7.4.1.2.2 Solutions envisagées . . . 129
7.4.1.2.2.1 ARK . . . 129
7.4.1.2.2.2 Forme d'un ARK . . . 129
7.4.1.2.2.3 Identifiant Vitam . . . 129
7.4.1.2.2.4 Logique de construction . . . 130
7.4.1.2.2.5 Logique d'affichage . . . 130
7.4.1.2.2.6 Capacité de déconstruction . . . 130
7.4.1.3 Graphes . . . 131
7.4.1.3.1 Objectifs . . . 131
7.4.1.4 Vérification des formats : . . . 131
7.4.2 Architecture Technique . . . 131
7.4.2.1 Introduction . . . 131
7.4.2.1.1 But de cette documentation . . . 131
7.4.2.1.2 GUID . . . 131
7.4.2.2 GUID . . . 131
7.4.2.2.1 Identifiant Vitam . . . 131
7.4.2.2.1.1 Forme d'un identifiant Vitam . . . 132
7.4.2.3 Configuration jetty . . . 132
7.4.2.4 Gestion des Handlers : . . . 133
7.4.2.5 Schéma de certificats et d'authentification . . . 134
7.4.2.5.1 Présentation . . . 134
7.4.2.6 Common format identification . . . 134
7.4.2.6.1 Présentation . . . 134
7.4.2.6.2 Sous packages . . . 134
7.4.2.6.2.1 Identification : . . . 134
7.4.2.6.2.2 Exceptions : . . . 134

7.4.2.6.2.3 Model : . . . 135
7.4.2.6.2.4 Siegfried : . . . 135
7.4.2.7 Messages . . . 135
7.4.2.8 Messages Logbook . . . 135
7.4.2.9 Request ID . . . 135
7.4.2.9.1 Filtre client . . . 136
7.4.2.9.2 Sauvegarde dans le thread local . . . 136
7.4.2.9.3 Filtre Serveur . . . 136
7.4.2.9.4 Affichage dans les logs . . . 136
7.4.3 Securite . . . 136
7.4.3.1 Introduction . . . 136
7.4.3.2 Securité de MongoDB . . . 136
7.4.3.2.1 Objectifs . . . 136
7.4.3.3 secret de la plateforme . . . 137
7.4.3.3.1 Objectifs . . . 137
7.5 Functional administration . . . 137
7.5.1 Architecture Fonctionnelle . . . 137
7.5.1.1 Introduction . . . 137
7.5.1.1.1 But de cette documentation . . . 137
7.5.1.2 Gestion de format . . . 137
7.5.1.3 Gestion de règles . . . 137
7.5.1.4 Sauvegarde du référentiel des règles de gestion . . . 138
7.6 IHM demo . . . 138
7.6.1 Architecture fonctionnelle . . . 138
7.6.1.1 Architecture fonctionnelle de l'application Back . . . 138
7.6.1.1.1 But de cette documentation . . . 138
7.6.1.1.2 Fonctionnement général du module . . . 138
7.6.1.1.2.1 Recherche des units : POST /ihm-demo/v1/api/archivesearch/units 138
7.6.1.1.2.2 Affichage du détail d'une archive unit : GET /ihm-demo/v1/api/archivesearch/unit/{id} . . . 139
7.6.1.1.2.3 Modification et enregistrement des détails d'une archive unit : PUT /ihm-demo/v1/api/archiveupdate/units/{id} . . . 139
7.6.1.1.2.4 Remarque importante . . . 140
7.6.1.1.2.5 Reste à faire . . . 140
7.6.1.2 Architecture fonctionnelle de l'application Front . . . 140
7.6.1.2.1 But de cette documentation . . . 140
7.6.1.2.2 Modules AngularJS déclarés . . . 140
7.6.1.2.3 Routage . . . 140
7.6.1.2.4 Factories/Services . . . 141
7.6.1.2.5 Controllers . . . 141
7.6.1.2.6 Components . . . 141
7.6.2 Architecture technique . . . 142
7.6.2.1 Architecture technique de l'application Back . . . 142
7.6.2.1.1 But de cette documentation . . . 142
7.6.2.1.2 Organisation du module ihm-demo . . . 142
7.6.2.1.2.1 **1. Module ihm-demo-web-application** . . . 142
7.6.2.1.2.2 **package fr.gouv.vitam.ihmdemo.appserver** . . . 142
7.6.2.1.2.3 **2. Module ihm-core** . . . 142
7.6.2.1.2.4 **package fr.gouv.vitam.ihmdemo.core** . . . 143
7.6.2.2 Architecture technique de l'application Front . . . 143
7.6.2.2.1 But de cette documentation . . . 143
7.6.2.2.2 Le Framework Front : AngularJS 1.5.3 . . . 143
7.6.2.2.2.1 Les modules AngularJS utilisés : . . . 143
7.6.2.2.2.2 Autres frameworks Front utilisés . . . 143

7.6.2.2.3 Organisation de l'application . . . 143
7.7 IHM recette . . . 144
7.7.1 Architecture technique . . . 144
7.7.1.1 Architecture technique de l'application Back . . . 144
7.7.1.1.1 But de cette documentation . . . 144
7.7.1.1.2 Organisation du module ihm-recette . . . 144
7.7.1.1.2.1 **1. Module ihm-demo-web-application** . . . 144
7.7.1.1.2.2 **package fr.gouv.vitam.ihmdemo.appserver** . . . 145
7.7.1.1.2.3 **package fr.gouv.vitam.ihmdemo.appserver.performance.** . . . 145
7.7.1.1.2.4 **2. Module ihm-recette-web-front** . . . 145
7.7.1.1.2.5 **3. Module ihm-core** . . . 145
7.7.1.2 Architecture technique de l'application Front . . . 145
7.7.1.2.1 But de cette documentation . . . 145
7.7.1.2.2 Le Framework Front : AngularJS 1.5.3 . . . 145
7.7.1.2.2.1 Les modules AngularJS utilisés : . . . 145
7.7.1.2.2.2 Autres frameworks Front utilisés . . . 146
7.7.1.2.3 Organisation de l'application . . . 146
7.8 Ingest . . . 146
7.8.1 Architecture Fonctionnelle . . . 146
7.8.1.1 Généralités . . . 146
7.8.1.2 Généralités . . . 147
7.8.1.3 Téléchargement standard et test à blanc d'un SIP : . . . 147
7.8.1.4 Autres Fonctionnalités : . . . 147
7.8.1.5 Ingest ExternalAntivirus . . . 147
7.8.1.6 Généralités . . . 148
7.8.1.7 Fonctionnalités concernant le workflow . . . 148
7.8.1.8 Les actions : . . . 148
7.8.1.9 Asynchrone : . . . 149
7.8.2 Technique . . . 149
7.8.2.1 Architecture Technique Ingest . . . 149
7.8.2.1.1 Présentation . . . 149
7.8.2.2 ingest-rest . . . 149
7.8.2.2.1 Présentation . . . 149
7.8.2.2.2 `IngestInternalApplication.java` . . . 149
7.9 Security-Internal . . . 150
7.9.1 Architecture Fonctionnelle . . . 150
7.9.1.1 Introduction . . . 150
7.9.1.1.1 But de cette documentation . . . 150
7.9.1.1.2 Security-internal . . . 150
7.9.2 Securite . . . 150
7.9.2.1 Introduction . . . 150
7.10 Logbook . . . 151
7.10.1 Architecture Fonctionnelle . . . 151
7.10.1.1 Généralités . . . 151
7.10.1.1.1 Journal d'opération . . . 151
7.10.1.2 Journal de cycle de vie . . . 151
7.10.1.3 Modèle de données . . . 151
7.10.1.3.1 Description des champs . . . 151
7.10.1.4 Modèle de données . . . 153
7.10.1.4.1 Description des champs . . . 153
7.10.2 Architecture technique . . . 155
7.10.2.1 Introduction . . . 155
7.10.2.1.1 Présentation . . . 155
7.10.2.1.2 Itération 3 et Itération 5 . . . 155

7.10.2.1.2.1 Itérations suivantes / à plus long terme . . . 155
7.10.2.1.3 Modules - packages logbook . . . 155
7.10.2.2 DSL . . . 156
7.10.2.2.1 Analyse . . . 156
7.10.2.2.1.1 Présentation . . . 156
7.10.2.2.1.2 Explication . . . 157
7.10.2.2.1.3 Utilisation . . . 157
7.10.2.2.2 Conclusion . . . 158
7.10.2.3 Rest . . . 158
7.10.2.3.1 Présentation . . . 158
7.10.2.3.2 Services . . . 158
7.10.2.4 Common-client . . . 158
7.10.2.4.1 Présentation . . . 158
7.10.2.4.2 Services . . . 159
7.10.2.5 Common-client . . . 159
7.10.2.5.1 Présentation . . . 159
7.10.2.5.2 Services . . . 159
7.10.2.5.3 Données . . . 159
7.10.2.6 Commons . . . 160
7.10.2.6.1 Présentation . . . 160
7.10.2.6.2 Services . . . 160
7.10.2.7 Operation Client . . . 160
7.10.2.7.1 Présentation . . . 160
7.10.2.7.2 Services . . . 160
7.10.2.8 Opération . . . 160
7.10.2.8.1 Présentation . . . 160
7.10.2.8.2 Services . . . 161
7.10.2.8.3 Rest API . . . 161
7.10.2.9 Lifecycle Client . . . 161
7.10.2.9.1 Présentation . . . 161
7.10.2.9.2 Services . . . 161
7.10.2.10 Lifecycle . . . 161
7.10.2.10.1 Présentation . . . 161
7.10.2.10.2 Services . . . 161
7.10.2.10.3 Rest API . . . 161
7.10.2.11 Administration-client . . . 162
7.10.2.11.1 Présentation . . . 162
7.10.2.11.2 Services . . . 162
7.10.2.12 Administration . . . 162
7.10.2.12.1 Présentation . . . 162
7.10.2.12.2 Services . . . 162
7.10.2.12.3 Rest API . . . 162
7.10.2.13 Sécurisation (traçabilité) . . . 162
7.10.2.13.1 Présentation . . . 162
7.10.2.13.1.1 Vue d'ensemble . . . 163
7.10.2.13.2 Algorithme générique . . . 164
7.10.2.13.2.1 Déroulement . . . 164
7.10.2.13.2.2 Contenu du zip . . . 165
7.10.2.13.2.3 Rôle de LogbookTraceabilityHelper . . . 165
7.10.2.13.3 Périmètre sécurisé . . . 165
7.10.2.13.3.1 Bornes de l'intervalle . . . 166
7.10.2.13.3.2 Continuité de la couverture . . . 166
7.10.2.13.3.3 Réduction du périmètre par le plafond . . . 166
7.10.2.13.3.4 Cloisonnement et limites . . . 166

7.10.2.13.4 Contenu de l'archive sécurisée . . . 167
7.10.2.13.4.1 data.txt . . . 167
7.10.2.13.4.2 Cas du journal des opérations . . . 167
7.10.2.13.4.3 Cas des cycles de vie . . . 167
7.10.2.13.4.4 merkleTree.json . . . 169
7.10.2.13.4.5 token.tsp . . . 169
7.10.2.13.4.6 computing_information.txt . . . 169
7.10.2.13.4.7 additional_information.txt . . . 169
7.10.2.13.5 Arbre de Merkle . . . 169
7.10.2.13.5.1 Construction . . . 169
7.10.2.13.5.2 Propagation d'une altération . . . 170
7.10.2.13.5.3 Sérialisation . . . 170
7.10.2.13.5.4 Vérification . . . 170
7.10.2.13.6 Horodatage et signature . . . 171
7.10.2.13.6.1 Empreinte horodatée et chaînage . . . 171
7.10.2.13.6.2 Génération du jeton . . . 171
7.10.2.13.6.3 Certificats et configuration . . . 172
7.10.2.13.6.4 Vérification du jeton . . . 172
7.10.2.13.6.5 Validation de la confiance . . . 173
7.10.2.13.7 Journal des opérations . . . 173
7.10.2.13.7.1 Déclenchement et sélection . . . 173
7.10.2.13.7.2 Plafonnement . . . 173
7.10.2.13.7.3 Événements produits . . . 174
7.10.2.13.8 LFC Unit et LFC GOT . . . 174
7.10.2.13.8.1 Déclenchement . . . 174
7.10.2.13.8.2 Workflows . . . 174
7.10.2.13.8.3 Données produites . . . 175
7.10.2.13.8.4 Croisement d'empreintes . . . 175
7.10.2.13.8.5 Plafonnement des cycles de vie . . . 175
7.10.2.13.9 SecurisationVersion . . . 176
7.10.2.13.9.1 Définition . . . 176
7.10.2.13.9.2 Stockage . . . 176
7.10.2.13.9.3 Critère de chaînage . . . 176
7.10.2.13.9.4 Usage côté vérification . . . 176
7.10.2.13.9.5 Configuration . . . 177
7.10.2.13.9.6 Portée du champ . . . 177
7.10.2.13.10 Synthèse de la chaîne de preuve . . . 177
7.10.2.13.10.1 Chaîne de dépendances . . . 177
7.10.2.13.10.2 Récapitulatif pas à pas . . . 178
7.10.2.13.11 Vérifications aval . . . 178
7.11 Metadata . . . 179
7.11.1 Architecture Fonctionnelle . . . 179
7.11.1.1 Introduction . . . 179
7.11.1.2 Généralités . . . 179
7.11.2 Architecture technique . . . 179
7.11.2.1 Introduction . . . 179
7.11.2.1.1 Présentation . . . 179
7.11.2.1.2 Itération 4 . . . 179
7.11.2.1.3 Modules - packages . . . 179
7.11.2.2 Opération . . . 180
7.11.2.2.1 Présentation . . . 180
7.11.2.2.2 Services . . . 180
7.11.2.2.3 Rest API . . . 180
7.11.2.3 Metadata-api . . . 181

7.11.2.3.1 Présentation . . . 181
7.11.2.4 Metadata-builder . . . 181
7.11.2.4.1 Présentation . . . 181
7.11.2.5 Operation Client . . . 182
7.11.2.5.1 Présentation . . . 182
7.11.2.6 metadata-core . . . 182
7.11.2.6.1 Présentation . . . 182
7.11.2.6.2 1. Modules et packages . . . 182
7.11.2.6.3 2. Classes . . . 182
7.11.2.6.3.1 2.1 Class DbRequest . . . 182
7.11.2.6.3.2 2.2 Class ElasticsearchAccessMetadata . . . 183
7.11.2.6.3.3 2.3 Class MetaDataImpl . . . 183
7.11.2.6.3.4 2.4 Class UnitNode . . . 184
7.11.2.6.3.5 2.5 Class UnitRuleCompute . . . 184
7.11.2.6.3.6 2.5 Class UnitInheritedRule . . . 184
7.11.2.7 metadata-parser . . . 184
7.11.2.7.1 Présentation . . . 184
7.11.2.8 Métadata . . . 184
7.11.2.8.1 Présentation . . . 184
7.11.2.8.2 Services . . . 185
7.11.2.8.3 Rest API . . . 185
7.11.2.9 Rest . . . 185
7.11.2.9.1 Présentation . . . 185
7.11.2.9.2 Services . . . 185
7.12 Processing . . . 185
7.12.1 architecture-fontionnelle-processing . . . 185
7.12.1.1 Introduction . . . 185
7.12.1.1.1 But de cette documentation . . . 185
7.12.1.1.2 Processing . . . 185
7.12.1.2 Processing Management . . . 186
7.12.1.3 Engine . . . 186
7.12.1.4 Distributor . . . 186
7.12.1.5 Worker . . . 187
7.12.1.6 Process Monitoring . . . 187
7.12.2 Architecture Technique . . . 187
7.12.2.1 Introduction . . . 187
7.12.2.2 DAT : module processing . . . 187
7.12.2.2.1 Module et packages . . . 187
7.12.2.2.2 Modèle . . . 188
7.12.2.2.3 Process Distributor . . . 188
7.12.2.2.4 Parallélisme dans le distributeur . . . 188
7.12.2.3 Rangement des objets . . . 189
7.12.2.3.1 Algorithme . . . 189
7.12.2.4 Vérification de la disponibilité . . . 190
7.12.2.4.1 Algorithme . . . 190
7.12.2.5 Vérifier SEDA . . . 190
7.12.2.5.1 Algorithme . . . 190
7.12.2.6 Métriques spécifiques du composant processing . . . 191
7.12.2.6.1 Besoins . . . 191
7.12.2.6.2 Liste des métriques . . . 191
7.12.2.6.3 Exploitation des métriques . . . 191
7.13 Scheduler . . . 192
7.13.1 Généralités . . . 192
7.13.2 Architecture Technique . . . 192

7.13.2.1 Introduction . . . 192
7.13.2.1.1 Présentation . . . 192
7.13.2.2 Jobs de logbook VITAM . . . 192
7.13.2.2.1 Liste des classes implémentant les jobs . . . 192
7.13.2.2.1.1 TraceabilityLFCJob.java . . . 192
7.13.2.2.1.2 TraceabilityJob.java . . . 192
7.13.2.2.1.3 TraceabilityAuditJob.java . . . 192
7.13.2.2.1.4 ReconstructionOperationJob.java . . . 192
7.13.2.3 Jobs de metadata VITAM . . . 193
7.13.2.3.1 Liste des classes implémentant les jobs . . . 193
7.13.2.3.1.1 AuditDataConsistencyMongoEsJob.java . . . 193
7.13.2.3.1.2 ProcessObsoleteComputedInheritedRulesJob.java . . . 193
7.13.2.3.1.3 PurgeDipJob.java . . . 193
7.13.2.3.1.4 PurgeSipJob.java . . . 193
7.13.2.3.1.5 ReconstructionJob.java . . . 193
7.13.2.3.1.6 StoreGraphJob.java . . . 193
7.13.2.4 Jobs de functional-administration VITAM . . . 194
7.13.2.4.1 Liste des classes implémentant les jobs . . . 194
7.13.2.4.1.1 ReconstructionAccessionRegisterJob.java . . . 194
7.13.2.4.1.2 ReconstructionReferentialJob.java . . . 194
7.13.2.4.1.3 ReferentialCreateSymblolicAccessionRegisterJob.java . . . 194
7.13.2.4.1.4 RuleManagementAuditJob.java . . . 194
7.13.2.5 Jobs de offer VITAM . . . 194
7.13.2.5.1 Liste des classes implémentant les jobs . . . 194
7.13.2.5.1.1 OfferLogCompactionJob.java . . . 195
7.14 Storage . . . 195
7.14.1 Architecture Technique . . . 195
7.14.1.1 Introduction . . . 195
7.14.1.1.1 Présentation . . . 195
7.14.1.1.2 Itération 16 . . . 195
7.14.1.1.3 Modules - packages Storage . . . 195
7.14.1.2 Architecture générale . . . 196
7.14.1.2.1 Schéma général . . . 196
7.14.1.2.2 Workflow du stockage des objets . . . 197
7.14.1.2.3 Itération 6 . . . 197
7.14.1.2.4 Itération 7 . . . 197
7.14.1.2.5 Itération 13 . . . 198
7.14.1.2.6 Itération 14 . . . 198
7.14.1.2.7 Itération 16 . . . 198
7.14.1.2.8 R12 . . . 198
7.14.1.3 Storage Driver . . . 198
7.14.1.3.1 Présentation . . . 198
7.14.1.3.2 Architecture . . . 199
7.14.1.3.3 Pour aller plus loin . . . 199
7.14.1.4 Storage Engine . . . 199
7.14.1.4.1 Présentation . . . 199
7.14.1.4.1.1 Services . . . 200
7.14.1.4.1.2 Rest API . . . 200
7.14.1.4.1.3 URI d'appel . . . 200
7.14.1.4.1.4 Headers . . . 200
7.14.1.4.1.5 Méthodes . . . 200
7.14.1.4.1.6 Distribution . . . 201
7.14.1.4.1.7 DriverManager : SPI . . . 202
7.14.1.4.1.8 Principe . . . 202

7.14.1.4.1.9 Persistance . . . 203
7.14.1.5 Storage Engine Client . . . 203
7.14.1.5.1 Présentation . . . 203
7.14.1.6 Storage Offers . . . 204
7.14.1.6.1 Présentation . . . 204
7.14.1.7 Vitam Offer . . . 204
7.14.1.7.1 Présentation . . . 204
7.14.1.7.2 Driver . . . 204
7.14.1.7.3 Serveur . . . 204
7.14.1.7.3.1 Description . . . 204
7.14.1.7.3.2 REST . . . 205
7.14.1.7.3.3 Description . . . 205
7.14.1.7.3.4 REST API . . . 205
7.14.1.8 Métriques spécifiques du composant storage-engine . . . 207
7.14.1.8.1 Besoins . . . 207
7.14.1.8.2 Liste des métriques . . . 207
7.14.1.8.3 Exploitation des métriques . . . 207
7.15 Worker . . . 208
7.15.1 architecture-fontionnelle-processing . . . 208
7.15.1.1 Introduction . . . 208
7.15.1.1.1 But de cette documentation . . . 208
7.15.1.1.2 Worker . . . 208
7.15.1.2 Worker . . . 208
7.15.1.3 notification-atr-ok . . . 208
7.15.1.4 notification-atr-ko . . . 208
7.15.1.5 Contrôle de la cohérence de SIPs . . . 209
7.15.2 Architecture Technique . . . 209
7.15.2.1 Module Worker . . . 209
7.15.2.2 Worker server . . . 210
7.15.2.2.1 Présentation . . . 210
7.15.2.2.1.1 Services . . . 210
7.15.2.2.1.2 Rest API . . . 210
7.15.2.2.1.3 URI d'appel . . . 210
7.15.2.2.1.4 Headers . . . 210
7.15.2.2.1.5 Méthodes . . . 210
7.15.2.3 Extraire les métadonnées des ArchiveUnit et DataObject . . . 210
7.15.2.3.1 Général . . . 210
7.15.2.3.1.1 Workspace avant extraction : . . . 211
7.15.2.3.1.2 Workspace après extraction : . . . 211
7.15.2.3.2 Algorithme . . . 211
7.15.2.3.2.1 Algorithme d'update pour l'extract SEDA . . . 212
7.15.2.4 notification-atr-ok . . . 212
7.15.2.5 notification-atr-ko . . . 212
7.15.2.6 Plugin Worker . . . 213
7.15.2.6.1 But de cette documentation . . . 213
7.15.2.6.2 Introduction . . . 213
7.15.2.6.3 Appel du plugin . . . 215
7.15.2.6.4 Résultat du plugin . . . 215
7.15.2.6.5 Implémentation . . . 215
7.15.2.6.5.1 Worker . . . 215
7.15.2.6.5.2 PluginPropertiesLoader . . . 216
7.15.2.6.5.3 Intégration . . . 216

8 Annexes **217**

Index 220

CHAPITRE 1

Introduction

1.1 Objectif de ce document

Ce document est le document d'architecture de la solution logicielle *VITAM* ; il vise à donner une vision d'ensemble des problématiques structurantes et de la solution (d'un point de vue applicatif et technique), ainsi que de présenter les choix structurants de principes et de composants et les raisons de ces choix.

Il s'adresse aux personnes suivantes :

- Les architectes applicatifs des projets désirant intégrer *VITAM* ;
- Les architectes techniques des projets désirant intégrer *VITAM*.

1.2 Structure du document

Ce document est séparé en 3 grandes parties :

- L'architecture applicative, principalement à destination des architectes applicatifs ;
- L'architecture technique, avec notamment :
  - En première sous-section, les principes d'architecture technique, principalement à destination des architectes d'infrastructure
  - Dans la suite, les choix d'architecture et de composants techniques, à destination des architectes d'infrastructure et des exploitants ;
- Les principes et règles de sécurité appliqués et applicables à la solution.

CHAPITRE 2

Rappels

2.1 Information concernant les licences

La solution logicielle *VITAM* est publiée sous la licence CeCILL 2.1 [1] ; la documentation associée (comprenant le présent document) est publiée sous Licence Ouverte V2.0 [2].

Les clients externes java de solution *VITAM* sont publiés sous la licence CeCILL-C [3] ; la documentation associée (comprenant le présent document) est publiée sous Licence Ouverte V2.0 [4].

2.2 Documents de référence

2.2.1 Documents internes

TABLEAU 1 – Documents de référence VITAM

Nom	Lien
DAT	http://www.programmevitam.fr/ressources/DocCourante/html/archi
DIN	http://www.programmevitam.fr/ressources/DocCourante/html/installation
DEX	http://www.programmevitam.fr/ressources/DocCourante/html/exploitation
DMV	http://www.programmevitam.fr/ressources/DocCourante/html/migration
Release notes	https://github.com/ProgrammeVitam/vitam/releases/latest

1. https://cecill.info/licences/Licence_CeCILL_V2.1-fr.html
2. https://www.etalab.gouv.fr/wp-content/uploads/2017/04/ETALAB-Licence-Ouverte-v2.0.pdf
3. https://cecill.info/licences/Licence_CeCILL-C_V1-fr.html
4. https://www.etalab.gouv.fr/wp-content/uploads/2017/04/ETALAB-Licence-Ouverte-v2.0.pdf

2.2.2 Référentiels externes

2.3 Glossaire

API *Application Programming Interface*

AU *Archive Unit*, unité archivistique

BDD Base De Données

BDO *Binary DataObject*

CA *Certificate Authority*, autorité de certification

CAS Content Adressable Storage

CCFN Composant Coffre Fort Numérique

CN Common Name

COTS Component Off The shelf ; il s'agit d'un composant « sur étagère », non développé par le projet *VITAM*, mais intégré à partir d'un binaire externe. Par exemple : MongoDB, ElasticSearch.

CRL *Certificate Revocation List* ; liste des identifiants des certificats qui ont été révoqués ou invalidés et qui ne sont donc plus dignes de confiance. Cette norme est spécifiée dans les RFC 5280 et RFC 6818.

CRUD *create, read, update, and delete*, s'applique aux opérations dans une base de données MongoDB

DAT Dossier d'Architecture Technique

DC Data Center

DEX Dossier d'EXploitation

DIN Dossier d'INstallation

DIP *Dissemination Information Package*

DMV Documentation de Montées de Version

DNS *Domain Name System*

DNSSEC *Domain Name System Security Extensions* est un protocole standardisé par l'IETF permettant de résoudre certains problèmes de sécurité liés au protocole DNS. Les spécifications sont publiées dans la RFC 4033 et les suivantes (une version antérieure de DNSSEC n'a eu aucun succès). Définition DNSSEC [5]

DSL *Domain Specific Language*, langage dédié pour le requêtage de VITAM

DUA Durée d'Utilité Administrative

EBIOS Méthode d'évaluation des risques en informatique, permettant d'apprécier les risques Sécurité des systèmes d'information (entités et vulnérabilités, méthodes d'attaques et éléments menaçants, éléments essentiels et besoins de sécurité…), de contribuer à leur traitement en spécifiant les exigences de sécurité à mettre en place, de préparer l'ensemble du dossier de sécurité nécessaire à l'acceptation des risques et de fournir les éléments utiles à la communication relative aux risques. Elle est compatible avec les normes ISO 13335 (GMITS), ISO 15408 (critères communs) et ISO 17799

EAD Description archivistique encodée

ELK Suite logicielle *Elasticsearch Logstash Kibana*

FIP *Floating IP*

GOT Groupe d'Objet Technique

IHM Interface Homme Machine

IP *Internet Protocol*

IsaDG Norme générale et internationale de description archivistique

JRE *Java Runtime Environment* ; il s'agit de la machine virtuelle Java permettant d'y exécuter les programmes compilés pour.

5. https://fr.wikipedia.org/wiki/Domain_Name_System_Security_Extensions

JVM *Java Virtual Machine* ; Cf. *JRE*

LAN *Local Area Network*, réseau informatique local, qui relie des ordinateurs dans une zone limitée

LFC *LiFe Cycle*, cycle de vie

LTS *Long-term support*, support à long terme : version spécifique d'un logiciel dont le support est assuré pour une période de temps plus longue que la normale.

M2M *Machine To Machine*

MitM L'attaque de l'homme du milieu (HDM) ou *man-in-the-middle attack* (MITM) est une attaque qui a pour but d'intercepter les communications entre deux parties, sans que ni l'une ni l'autre ne puisse se douter que le canal de communication entre elles a été compromis. Le canal le plus courant est une connexion à Internet de l'internaute lambda. L'attaquant doit d'abord être capable d'observer et d'intercepter les messages d'une victime à l'autre. L'attaque « homme du milieu » est particulièrement applicable dans la méthode d'échange de clés Diffie-Hellman, quand cet échange est utilisé sans authentification. Avec authentification, Diffie-Hellman est en revanche invulnérable aux écoutes du canal, et est d'ailleurs conçu pour cela. Explication [6]

MoReq *Modular Requirements for Records System*, recueil d'exigences pour l'organisation de l'archivage, élaboré dans le cadre de l'Union européenne.

NoSQL Base de données non-basée sur un paradigme classique des bases relationnelles. Définition NoSQL [7]

NTP *Network Time Protocol*

OAIS *Open Archival Information System*, acronyme anglais pour Systèmes de transfert des informations et données spatiales – Système ouvert d'archivage d'information (SOAI) - Modèle de référence.

OOM Aussi apelé *Out-Of-Memory Killer* ; mécanisme de la dernière chance incorporé au noyau Linux, en cas de dépassement de la capacité mémoire

OS *Operating System*, système d'exploitation

OWASP *Open Web Application Security Project*, communauté en ligne de façon libre et ouverte à tous publiant des recommandations de sécurisation Web et de proposant aux internautes, administrateurs et entreprises des méthodes et outils de référence permettant de contrôler le niveau de sécurisation de ses applications Web

PDMA Perte de Données Maximale Admissible ; il s'agit du pourcentage de données stockées dans le système qu'il est acceptable de perdre lors d'un incident de production.

PKI Une infrastructure à clés publiques (ICP) ou infrastructure de gestion de clés (IGC) ou encore Public Key Infrastructure (PKI), est un ensemble de composants physiques (des ordinateurs, des équipements cryptographiques logiciels ou matériel type HSM ou encore des cartes à puces), de procédures humaines (vérifications, validation) et de logiciels (système et application) en vue de gérer le cycle de vie des certificats numériques ou certificats électroniques. Définition PKI [8]

PCA Plan de Continuité d'Activité

PRA Plan de Reprise d'Activité

REST *REpresentational State Transfer* : type d'architecture d'échanges. Appliqué aux services web, en se basant sur les appels http standard, il permet de fournir des API dites « RESTful » qui présentent un certain nombre d'avantages en termes d'indépendance, d'universalité, de maintenabilité et de gestion de charge. Définition REST [9]

RGAA Référentiel Général d'Accessibilité pour les Administrations

RGI Référentiel Général d'Interopérabilité

RPM *Red Hat Package Manager* ; il s'agit du format de paquets logiciels nativement utilisé par les distributions Linux RedHat/CentOS (entre autres)

SAE Système d'Archivage Électronique

SEDA Standard d'Échange de Données pour l'Archivage

6. https://fr.wikipedia.org/wiki/Attaque_de_l'homme_du_milieu
7. https://fr.wikipedia.org/wiki/NoSQL
8. https://fr.wikipedia.org/wiki/Infrastructure_%C3%A0_cl%C3%A9s_publiques
9. https://fr.wikipedia.org/wiki/Representational_state_transfer

SGBD Système de Gestion de Base de Données

SGBDR Système de Gestion de Base de Données Relationnelle

SIA Système d'Informations Archivistique

SIEM *Security Information and Event Management*

SIP *Submission Information Package*

SSH *Secure SHell*

Swift *OpenStack Object Store project*

TLS *Transport Layer Security*

TNA *The National Archives*, Pronom [10]

TNR Tests de Non-Régression

TSA *Time Stamping Authority*, autorité d'horodatage délivrant les jetons d'horodatage utilisés dans les preuves

TTL *Time To Live*, indique le temps pendant lequel une information doit être conservée, ou le temps pendant lequel une information doit être gardée en cache

UDP *User Datagram Protocol*, protocole de datagramme utilisateur, un des principaux protocoles de télécommunication utilisés par Internet. Il fait partie de la couche transport du modèle OSI

UID *User IDentification*

VITAM Valeurs Immatérielles Transférées aux Archives pour Mémoire

VM *Virtual Machine*

WAF *Web Application Firewall*

WAN *Wide Area Network*, réseau informatique couvrant une grande zone géographique, typiquement à l'échelle d'un pays, d'un continent, ou de la planète entière

10. https://www.nationalarchives.gov.uk/PRONOM/

CHAPITRE 3

Vue d'ensemble

3.1 Drivers du projet

3.1.1 Enjeux

Les enjeux de la solution logicielle *VITAM* se répartissent en 3 grandes catégories :

- Les enjeux liés au respect des processus métier d'archivage ; il s'agit de permettre l'identification, le maintien de la disponibilité et de la sécurité, ainsi que le maintien du contrôle sur les documents confiés à VITAM. Dans le cas particulier de l'archivage définitif, *VITAM* doit permettre l'utilisation des documents à des fins historiques liées à leur réutilisation, et permettre la conservation de documents dont la *DUA* est échue mais ayant vocation à être conservés indéfiniment.
- Les enjeux liés au volume, à la variété et aux besoin de performances des traitements des données gérées par *VITAM* :
  - *VITAM* doit pouvoir gérer la conservation et l'accès de volumes élevés d'archives numériques (> 10^{10} objets, 10 To => 10 Po), tout en garantissant une perte de données nulle (*PDMA* ~ 0) pour les données qui ont été « acceptées » par VITAM après acquittement d'un versement, ainsi que pour l'ensemble des données nécessaires pour assurer la preuve systémique de la plateforme (journaux des opérations, du cycle de vie, du *SAE*) ;
  - *VITAM* doit pouvoir gérer un large éventail de types de données archivées, et ce notamment dans le temps, incluant une forte variété de métadonnées descriptives des archives et une forte variété de type de format des objets numériques ;
  - *VITAM* doit être performant dans ses capacités de gestion des données archivées, et notamment permettre de répondre à des requêtes de recherche simples en quelques secondes, à des recherches complexes archivistiques en quelques minutes et à une demande d'accès à un contenu quelconque en une dizaine de secondes.
- Les enjeux liés à la sécurité, en fournissant un accès sécurisé et contrôlé ainsi qu'en garantissant la traçabilité des actions (gestion notamment de documents devant conserver leur valeur probante). En outre, VITAM doit permettre de garantir une très longue durée d'accès et de conservation (> 50 ans) des archives, et doit notamment pouvoir résister à l'obsolescence informatique.

3.1.2 Contraintes et objectifs

L'accès aux archives numériques doit être facile :

- Adapté : Services Web, Nouveaux média
- Interopérable : *RGI* et respect des standards ou normes d'échange et de communication
- Requêtable : le *SAE* doit fournir un service de recherche, tout comme un *SGBD* : une interface de requêtes des bases qu'il héberge
- Mutualisable :
  - le *SAE* doit pouvoir fournir un plan de classement multiple et une capacité d'accès depuis plusieurs applications
  - le *SAE* doit pouvoir gérer des dizaines de milliards d'entrées et leurs métadonnées associées avec une variabilité des formats des unités d'archives (objets numériques) et des descriptions associées (métadonnées)

L'accès aux archives numériques doit être rapide :

- Le temps d'accès pour une archive unitaire (un document) ou des métadonnées doit être compatible avec les technologies actuelles (Cf. le paragraphe précédent) ;
- Pour les accès à des lots d'archives, les moyens utilisés doivent être appropriés :
  - Via un support physique
  - Via un téléchargement de masse
- Du fait de la sensibilité des données :
  - L'accès doit être sécurisé (Réseau, Protocolaire, Filtrage)
  - L'accès doit être contrôlé (sur la base de contrats et de filtres métiers associés)

3.1.3 Positionnement

La solution logicielle *VITAM* est un *back-office* pouvant s'interfacer à tout *front-office* (utilisateur) devant accéder à des données archivées (pas nécessairement pour de l'archivage définitif). Il disposera cependant des *IHM* d'administration pour l'administration technique et fonctionnelle de la plateforme ainsi que d'une IHM minimale pouvant pallier à l'absence temporaire d'un *front-office*.

La solution logicielle *VITAM* a pour but d'être largement réutilisable, et ce notamment en se basant sur l'usage de standards métiers (ex : *SEDA* pour les versements).

Enfin, le socle logiciel doit pouvoir être utilisable pendant 20 ans (en incluant les évolutions technologiques).

3.2 Interfaces externes du système

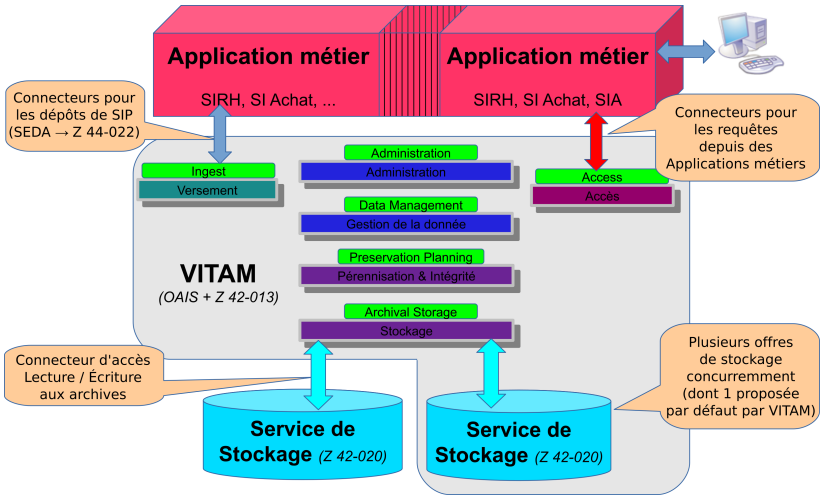


FIG. 1 – Vue de VITAM dans son environnement (vue « boîte noire »)

3.2.1 Interfaces requises

Dans cette version du système, aucune interface externe autre que les services IT standard (*NTP*, *DNS*, dépôts de mise à jours des *OS*, ...) n'est requise par *VITAM*.

3.2.2 Interfaces métier exposées

La solution logicielle *VITAM* expose trois grands groupes d'*API* métier :

- Les API d'*ingest* : elles permettent l'entrée d'une nouvelle archive dans le système ;
- Les API d'accès : elles permettent d'accéder aux données d'archives présentes dans le système (métadonnées et données d'archives, journaux, référentiels) ;
- Les API d'administration fonctionnelles : elles permettent notamment la modification des référentiels métier.

Ces API sont exposées en tant qu'API *REST* (HTTPS) au niveau des composants externes (composants `*-external`), avec un accès protégé par une authentification par certificat.

Voir aussi :

Les points relatifs à la sécurité des interfaces externes exposées sont abordés dans la section *sécurité* (page 107).

3.3 Orientations générales

Avertissement : Ces orientations générales donnent la direction vers laquelle tend la solution logicielle *VITAM* ; il contient donc des références à des fonctionnalités qui ne sont pas forcément présentes dans cette version du système *VITAM*.

3.3.1 Open Source

Les logiciels utilisés et le résultat sont *Open Source* afin de faciliter la réutilisation et d'éviter les contraintes de marchés publics pour la réutilisation au sein des différentes entités publiques.

Le logiciel produit est un logiciel de *Back-office*, supposant qu'il y a donc des *Front-offices* développés par ailleurs.

Le *back-office* se veut être mutualisé entre plusieurs *Front-offices*, pour :

- Permettre la réutilisation des données (objets numériques et métadonnées) dans plusieurs contextes (mémoire de l'entité publique)
- Permettre la réduction des coûts en centralisant les investissements.

Chaque *front-office* aura des conditions particulières d'usage du *back-office*. Ces conditions particulières pourront varier selon :

- La nature des grandes opérations qui pourront être effectuées (versement, accès, gestion, ...)
- La nature des variantes d'opérations qui pourront être effectuées (ajout d'une entrée, modification d'une entrée ou de métadonnées, ...) : en résumé lecture seule, écriture simple (insertion), écriture riche (mise à jour), effacement
- Le domaine d'application de ces opérations (quelles filières, quels objets numériques, quels périmètres)
- Le filtrage sur ces domaines (2 niveaux : habilité / non habilité, utilisables en fonction de règles de gestion : communicabilité, diffusion, données publiques/privées, ...)

Même si *VITAM* est un *back-office*, certaines *IHM* sont prévues pour différentes fonctions :

- *IHM* d'administration : pour les opérations d'administration (métier) à accès restreint. Selon le *front-office* utilisé, cette *IHM* peut ne pas être nécessaire ;
- *IHM* minimale : elle assure un socle minimal d'"*IHM* pour assurer un usage rapide de la solution logicielle *VITAM*. Cette *IHM* est prévue pour être utilisée dans les cas simples, et donc, selon le front-office utilisé, elle peut ne pas être nécessaire ;
- *IHM* de démonstration : elle porte des exemples d'implémentations limitatives tant en fonctionnalité qu'en garantie de fonctionnement. Ces *IHM* ne doivent pas être mises en production mais sont des exemples dont peuvent s'inspirer les concepteurs d'applications *front-offices*.
    - Cette *IHM* porte notamment des codes de démonstration, des cas particulier d'exemples pour de futures implémentations de front-offices, mais uniquement sur un aspect codage (requêtes et réponses) pour illustrer des cas d'usages.

3.3.2 API REST

Pour assurer l'interconnexion entre le *back-office* et les *front-offices*, il est proposé d'utiliser des interfaces HTTPS *REST* (hors protocoles spécifiques additionnels de transferts de fichiers). Ainsi, toutes les fonctionnalités accessibles aux *front-offices* seront offertes via ces *API*. Les *IHM* minimales et de démonstration utiliseront ces *API*. Les *IHM* d'administration pourront utiliser des API spécifiques si nécessaire (mais ce n'est pas une obligation, ces API pouvant elles aussi être exposées in fine).

Une analogie peut être faite entre *VITAM* et une base de données :

- Une base de données peut héberger une ou plusieurs tables communes à de multiples applications clientes ;
- Les applications clientes utilisent des *API* (SQL) pour échanger avec le moteur de la base de données ;
- La base de données dispose d'une *IHM* spécifique d'administration pouvant utiliser les mêmes *API* (SQL) ou des API spécifiques du moteur.

3.3.3 Big Data et Cloud computing

Les contraintes de volumétrie (plusieurs dizaines de milliards d'objets) conduisent à une volumétrie (en nombre) dépassant les capacités des logiciels usuels (type *SGBDR*). Les technologies *NoSQL* ou Cloud computing à forte distribution permettent de pallier ce problème.

Pour chaque objet numérique, les métadonnées associées sont variables ([Nom, Prénom, . . .] pour un dossier RH, [Projet, Domaine, . . .] pour un dossier projet, [Action, Plan comptable, . . .] pour de la comptabilité, . . .). Cette variabilité peut être assumée par des technologies NoSQL dites *schemaless*.

Chaque objet numérique peut être d'un format différent (Word, PDF, JPG, AVI, . . .). La lisibilité dans le temps d'un objet numérique est un enjeu majeur. Si on ne peut plus le lire (le consulter par exemple), il n'est plus compréhensible par l'humain. Les transformations de format pour en assurer la lisibilité sont donc indispensables dans le temps. Du fait de la masse (dizaines de milliards d'objets numériques), cette contrainte impose de gérer une vélocité de grande masse.

Du fait de la prolifération des formats et des usages (usage dit de master pour la conservation, usage dit de diffusion dans une qualité moindre, voire d'autres usages comme une qualité de type vignette ou contenu textuel (TEXTE)), ces formats induisent eux aussi une grande variabilité qui doit être gérée de manière efficiente (vélocité).

De plus, l'accès aux métadonnées ou aux objets numériques doit pouvoir se faire dans des temps acceptables (de la seconde à quelques dizaines de secondes pour certains éléments massifs). Là aussi, la vélocité est donc un point important.

Ces 3 V (Volume, Variété, Vélocité) imposent une vision « Big Data » mais non analytique (*Big Data* de traitements). Il n'est pas prévu par exemple de pouvoir effectuer des traitements de masse sur le contenu des archives et d'en déduire des analyses statistiques ou d'utiliser des mécanismes d'intelligence artificielle. Ainsi le modèle Hadoop ne correspond pas à notre usage.

3.3.4 Cloud storage

La particularité de l'accès aux objets numériques est un accès unitaire à minima (l'accès à un lot se résumant à faire des accès unitaires pour chacun des éléments de ce lot). Ainsi, on accède à un courriel et non uniquement à une boîte aux lettres. De ce fait, chaque objet étant accédé unitairement, la logique retenue pour le stockage est une logique Objet (*CAS*) et non une logique systèmes de fichiers. L'implémentation réelle peut s'appuyer sur une logique de systèmes de fichiers, mais l'interface visible sera bien objet. Le modèle de référence (ce qui ne veut pas dire l'implémentation réelle ni l'interface exacte) s'inspire de la NF Z 42-020 et du modèle *Swift* ou *CEPH*. L'avantage des deux dernières technologies est qu'elles permettent d'envisager un modèle qui peut croître en taille sans avoir à tout changer à chaque fois. Il s'agit donc du modèle *Cloud Storage*.

Note : Les notions de *Cloud computing* ou *Cloud Storage* ne sont pas à prendre au sens hébergement chez Amazon, Google ou Azure, mais au sens des technologies sous-jacentes.

Par contre, il doit être possible de regrouper logiquement des unités en lots (des courriels d'une boîte aux lettres) afin d'en faciliter l'accès. Comme il s'agit de regroupement logique, et que pour une même unité, plusieurs regroupements peuvent être envisagés (un courriel classé dans une boîte, et ce même courriel classé dans un dossier d'affaire), c'est une vision arborescente (dossiers, sous-dossiers, tout comme une arborescence de répertoires contenant des fichiers) disjointe des objets numériques qui est mise en oeuvre. Celle-ci s'inspire du modèle *IsaDG*, *EAD*, *SEDA* mais aussi du modèle *MoReq* 2010. Il a conduit à la notion d" « unités d'archives » (ou Units) structurés dans une arborescence (plan de classement).

Cette façon de distinguer ce qui est porté dans l'arbre de métadonnées (le classement) et dans le stockage (les objets unitaires) permet de faciliter le développement différencié des deux en en réduisant la complexité pour chacun, ce qui permet d'envisager le remplacement plus facilement de telle ou telle partie, et en particulier pour le stockage, d'autoriser d'autres implémentations.

3.3.5 PCA/PRA et répartitions des travaux

La solution logicielle *VITAM* est prévue pour être installée sur un nombre de sites suffisant pour assurer la sécurité des données. Selon les volumétries, le nombre de sites peut être variable :

- 1 site : Ce cas ne peut concerner que des volumes de très petite taille dont la sauvegarde journalière et complète (« Full daily backup ») est permise et réaliste, ainsi que l'acceptation d'un délai de remise en oeuvre de quelques jours. Ceci n'empêche pas la mise en oeuvre de sauvegarde différentielle, mais donne une limite raisonnable d'application du modèle. Une version particulière de Vitam nommée mini-Vitam devrait permettre une telle mise en oeuvre mais avec des fonctionnalités amoindries pour tenir sur un ensemble limité de serveurs ;
- 2 sites : Ce cas peut être acceptable tant qu'un plan de sauvegarde traditionnel des volumétries est applicable (moins d'un To a priori) via, par exemple, un schéma de sauvegarde de type « Full backup » hebdomadaire et « Incremental backup » journalier. Il s'agit de la réplication des architectures usuelles pour les applications informatiques. Le second site est considéré comme le site de Plan de Reprise d'Activité (PRA) ;
- 3 sites : Ce cas devrait être le plus général, car il permet de couvrir les volumes les plus importants (plusieurs centaines de To ou plus) où les moyens de sauvegarde usuels ne fonctionnent plus, tout en assurant la sécurité. En cas de sinistre sur un site, le deuxième site « chaud » permet de redémarrer rapidement le service. En cas d'incident après un sinistre, le 3 ème site assure la sécurité des données, comme une sauvegarde classique le ferait.

3.3.6 Sécurité des données additionnelle

Chaque offre de stockage doit répondre aux enjeux définis dans la norme « NF Z 42-020 » (*CCFN*).

La recommandation en termes de sécurité est d'avoir au moins 3 copies d'une même archive, réparties sur au moins 3 sites pour des raisons de sécurités géographiques (en limitant l'impact de sinistres impliquant la disparition d'un site de production) et sur au moins 2 types de stockage de natures distinctes.

- Le recours à plusieurs offres de stockage permet d'assurer une meilleure résilience : une attaque, une faille de sécurité ou un défaut d'usure sont liés à la technologie utilisée ; varier les technologies tend à diminuer ce risque (comme il est d'usage de le faire par exemple avec les solutions de sécurité) ;
- Plusieurs offres de stockage doivent être supportées simultanément par le logiciel Vitam afin de permettre les migrations dans le temps entre les offres (tous les 5 à 10 ans selon les technologies utilisées) ;
- Des implémentations d'offres de stockages réalisées par exemple par des acteurs privés en dehors du Programme Vitam (constructeur, éditeur, etc.) pourront venir compléter ou remplacer les solutions proposées par le programme Vitam, ceci permettant d'offrir à Vitam une meilleure capacité à résister dans le temps par la multiplicité des choix proposés. Une illustration de l'architecture de stockage est présentée ci-après.
- Plusieurs offres de stockage permettent de servir plusieurs niveaux de services :
  - Par exemple des accès rapides pour les accès aux versions de diffusion des archives, et à l'inverse des accès lents pour les accès aux originaux (masters) potentiellement plus volumineux ; à l'instar de la vidéo en mode HDV pouvant être considérée comme le format « master » mais non diffusable du fait de sa taille – 3 Mo/s environ, soit plus de 11 Go/h – qui serait stockée sur des supports lents, tandis que le format Xvid – 500 Mo/h – serait utilisé pour la diffusion et servi par des supports rapides ;
  - Ces niveaux de services différents permettent aussi de répondre à des exigences de sécurité (résilience par rapport à une autre offre). Il est proposé ainsi la mise en oeuvre de deux niveaux de services majeurs pour offrir un délai complémentaire de réactivité et éviter ainsi des destructions d'archives (suite à un incident, une attaque ou un défaut) :
    - Via une offre dénommée « stockage primaire » (ou secondaire en secours immédiat ou « chaud ») servant aux accès rapides mais pouvant subir des éliminations tout aussi rapides (et donc dangereuses en termes de sécurité) ;
    - Et l'autre dénommée « stockage de sécurité », lent par nature (et même si possible « offline » ou « froid ») dont les propriétés d'accès rendent lentes les opérations d'écriture et d'élimination.

3.3.7 Architecture multi-tenants

La solution logicielle *VITAM* doit pouvoir être instanciée sur une infrastructure mutualisée, avec une administration centralisée et unique des composants, mais en autorisant une séparation virtuelle et sécurisée des informations (archives et métadonnées) pour chaque client (client = « tenant » en anglais) ainsi que la gestion séparée de ces informations par chaque client.

L'objectif est de permettre, par exemple, le regroupement d'acteurs publics au sein d'une même infrastructure pour diminuer les coûts d'infrastructure et d'exploitation, tout en assurant une étanchéité entre ces environnements logiques pour chaque client.

3.3.8 Solution exploitable

Du fait de la complexité des composants à mettre en oeuvre et donc de l'exploitation associée, tant par composant que dans des visions de suivi d'opérations, la solution logicielle *VITAM* doit apporter un maximum d'aides et de facilités aux administrateurs techniques et exploitants, sans forcément se substituer aux outils d'administration propres à chacun des composants.

Ainsi, il est nécessaire de disposer d'un outillage permettant la configuration, l'installation et la mise à jour des composants et des services pour une plate-forme *VITAM*.

Il est également nécessaire de disposer d'un outillage permettant de suivre l'activité du système global :

- Gestion des logs centralisée
- Suivi des opérations ou d'une opération en cours
- Planification

Il n'est pas obligatoire de substituer des outils d'administration d'un composant lorsqu'ils existent déjà :

- Administration d'une base MongoDB ou d'une base ElasticSearch
- Supervision technique des *VM* et *OS*, du réseau,...

Par contre, certaines informations utiles (soit pour le déroulement d'une opération comme la charge CPU d'un serveur, soit pour une vision globale de l'activité comme la charge CPU ou réseau de la plate-forme) pourraient être captées par la solution logicielle *VITAM* pour ses propres usages (et donner de l'information à l'administrateur technique).

3.4 Architecture fonctionnelle

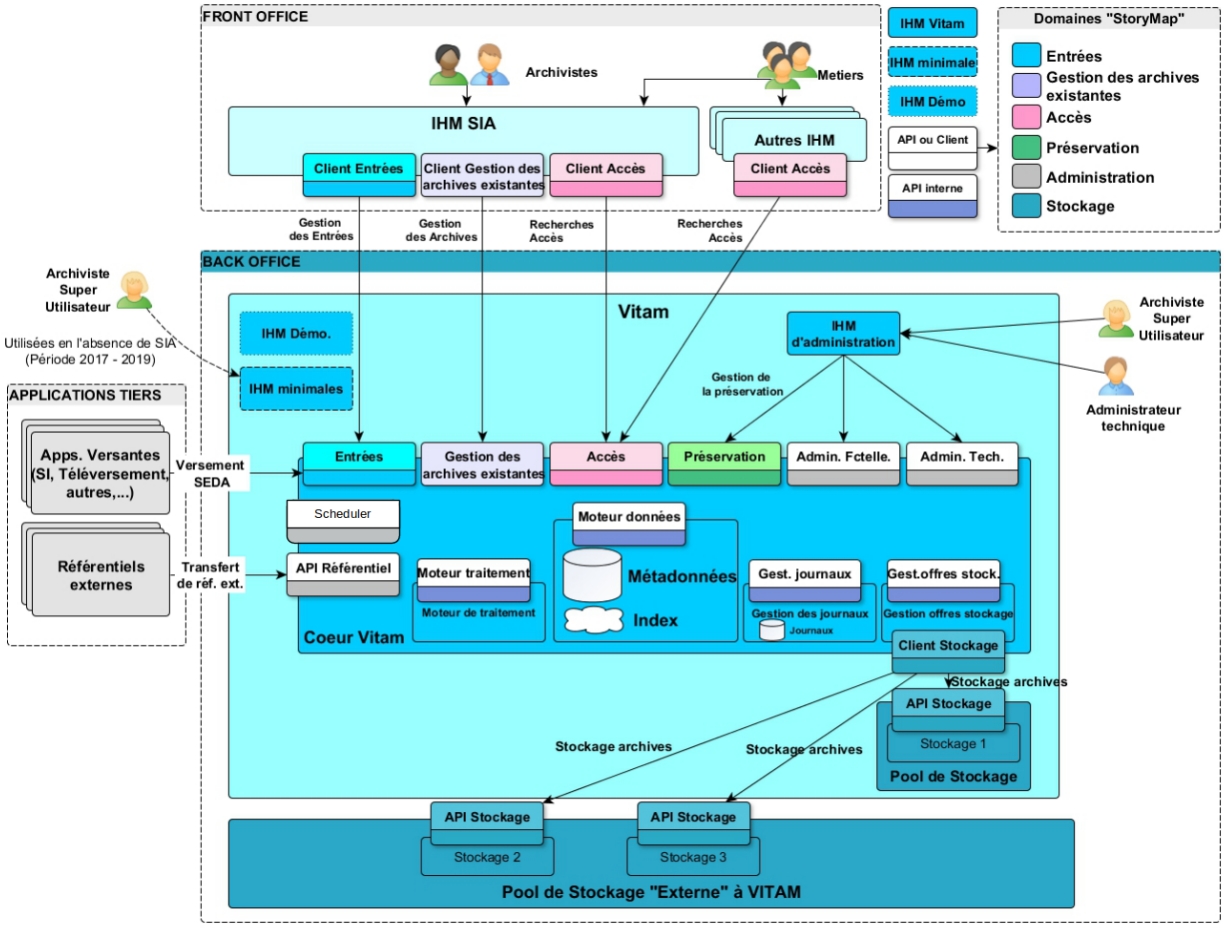


FIG. 2 – Architecture fonctionnelle cible de *VITAM*

La solution logicielle *VITAM* est constituée de différents composants liés aux fonctionnalités attendues :

- *API* externes : exposition des *API REST* (aux *front-offices*, aux applications tierces)
- Moteur d'exécution : gestion de toutes les tâches massives/asynchrones. Exemples de moteurs :
  - *Workflow* de transformation : sert à la transformation des documents dans des formats pérennes (versement) ou pour résister à l'obsolescence des formats stockés (préservation)
  - *Workflow* d'audit
- Moteur de stockage : stockage pérenne des données (méta-données et objets numériques)
- Moteur de données : stockage accessible et requêtable des méta-données
- Journalisation fonctionnelle : traçabilité fonctionnelle (dont à valeur probante)
- *IHM* d'administration : interface d'administration technique et fonctionnelle

Pour l'exploitabilité de la solution, on peut rajouter les composants suivants :

- Moteur de déploiement et de configuration
- Composants d'assistances/*hook* à l'exploitabilité (sauvegarde, supervision, ordonnancement)
- Journalisation technique : concentration des logs techniques

CHAPITRE 4

Architecture applicative

Cette section décrit l'architecture applicative interne de la solution logicielle *VITAM*, i.e. les différents composants la constituant et leurs interactions.

4.1 Architecture applicative

4.1.1 Drivers de l'architecture

Les principes d'implémentation applicative ont pour but de faciliter, voire d'assurer les enjeux auxquels la solution logicielle *VITAM* est confrontée :

- Modèle *Open-Source* pour la réutilisation dans la sphère publique ainsi que pour conserver la maîtrise dans le temps du socle logiciel ;
- Couplage lâche entre les composants ;
- Nécessité de pouvoir disposer de composants de générations différentes rendant un même service ;
- Usage d'*API REST* pour la communication entre composants internes à *VITAM*, ainsi qu'en extrême majorité pour les services exposés à l'extérieur ;
- Exploitabilité de la solution : limiter le coût d'entrée et de maintenance en :
  - Intégrant un outillage favorisant le déploiement et les mises à jour de la plateforme ;
  - Intégrant les éléments nécessaires pour l'exploiter (supervision, sauvegarde, ordonnancement) ;
  - Enfin, à terme, la solution doit pouvoir tirer partie d'une infrastructure élastique et disposant d'offres de services de stockage diverses (externes).

4.1.2 Services

La solution logicielle *VITAM* est découpée en services autonomes interagissant pour permettre de rendre le service global ; ce découpage applicatif suit en grande partie le découpage présenté plus haut dans l'architecture fonctionnelle.

Les schémas suivants présentent l'architecture applicative et les flux d'informations entre composants. Tous les composants qui sont en jaune, sont fournis dans le cadre de la solution logicielle VITAM ; tous sont requis pour le bon

fonctionnement de la solution, à l'exception de deux d'entre eux : `ihm-demo` et `offer` (selon les choix de déploiement). Enfin, chaque service possède un nom propre qui l'identifie de manière unique au sein de la solution logicielle *VITAM*.

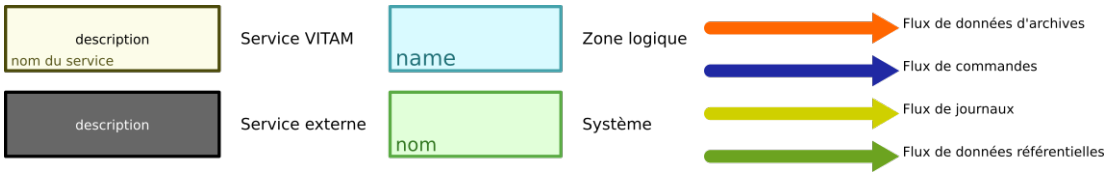


FIG. 1 – Architecture applicative : légende

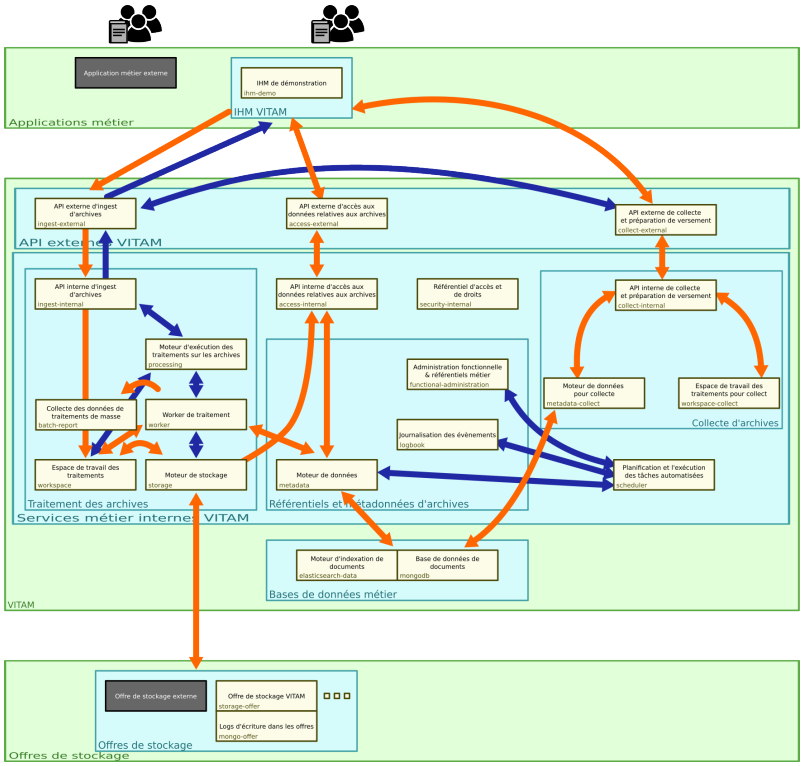


FIG. 2 – Architecture applicative : flux de données d'archives et de commandes

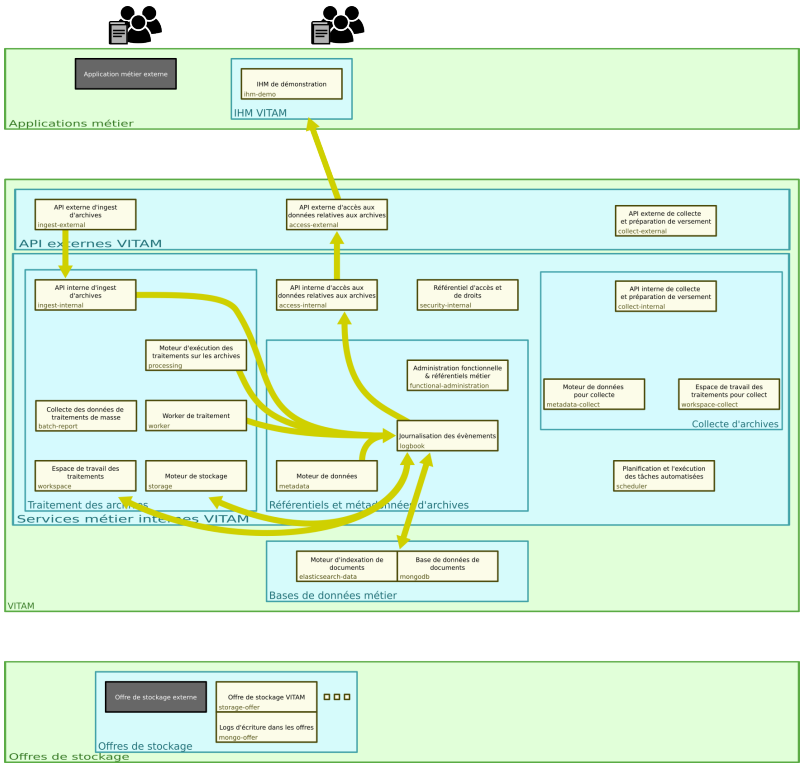


FIG. 3 – Architecture applicative : flux de données de journalisation

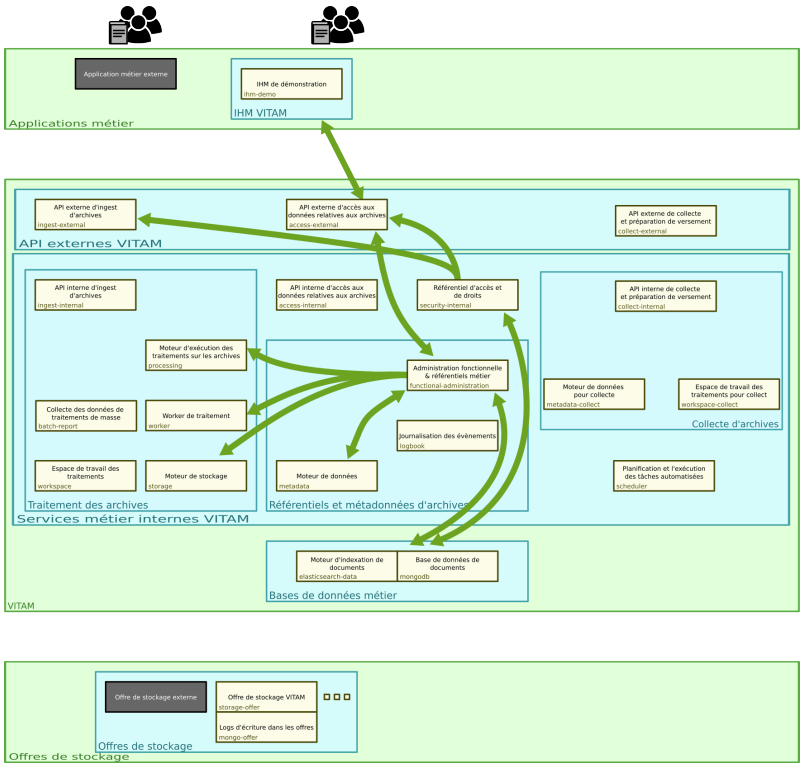


FIG. 4 – Architecture applicative : flux de données de référentiels

Les services sont organisés en zones logiques :

- Les *API* externes contiennent les services exposés aux clients (ex : à un *SIA*) ; tout accès externe à la solution logicielle *VITAM* doit passer par eux. Ils sont responsables notamment de la validation de l'authentification des systèmes externes, de la validation du droit d'accès aux *API* internes et de l'appel des *API* internes (principe d'*API*-Gateway) ;
- Les services métiers internes hébergent la logique métier de gestion des archives ; ils se subdivisent en :
  - Les services de traitement des archives : ils effectuent tous les traitements concernant les archives (unitaires ou de masse) ;
  - Les services de recherche et d'accès aux archives : ils permettent de consulter les métadonnées et le contenu des archives ;
  - Les services de gestion des référentiels et des métadonnées d'archives : ils permettent de travailler sur les métadonnées des archives (au sens large, i.e. comprenant les référentiels et les journaux).
- Les offres de stockage (internes - i.e. fournies par VITAM - ou externes - i.e. fournies par un tiers) stockent les données d'archives gérées par VITAM ; la sélection de l'offre de stockage à utiliser pour une archive donnée est réalisée en amont (dans le moteur de stockage).
- Enfin, les bases de données métiers stockent les données de travail concernant les archives et leurs traitements (notamment : métadonnées d'archives, journaux, référentiels)

Une dernière zone, optionnelle, consiste en une *IHM* de démonstration de la solution. Du point de vue de la solution *VITAM*, elle se comporte comme un application métier externe ; elle accède notamment aux services VITAM via les mêmes *API* qu'une application métier.

4.1.3 Détail des flux d'information métier

On distingue globalement 4 types de flux de données différents :

- Les flux de données d'archives : ils portent les informations métiers associées aux contenus des archives (données stockées ou métadonnées associées) ;
- Les flux de commandes : ils portent les demandes d'exécution de traitement d'archives et l'état de ces exécutions (et comprennent donc notamment les notifications de fin d'exécution de ces traitements) ;
- Les flux de journaux : ils portent les journaux d'évènements (traces probantes des actions réalisées sur les archives) ;
- Les flux de référentiels : ils portent les informations des référentiels hébergés au sein de *VITAM* (référentiels des formats, des contrats, . . .)

4.1.4 Données métier

Le modèle de donnés métier est décrit dans un document dédié [11] .

4.2 Architecture des données & multisite

4.2.1 Inventaire des données

Voir aussi :

Le modèle de données complet est explicité dans la documentation externe dédiée (« Modèle de données »).

Le tableau ci-dessous représente l'inventaire des données gérées par *VITAM*, avec leur localisation et le composant responsable du cycle de vie de la donnée (i.e. règles de création / modification / suppression) :

11. http://www.programmevitam.fr/ressources/DocCourante/html/data-model

TABLEAU 1: Inventaire des données VITAM

Type	Données prises en charge par le module	Composant responsable de la donnée	Persistence locale	Persistence BDD métier	Persistence Workspace	Persistence Storage	Persistence BDD techniques	Persistence inventaire ansible
métier	Sas d'entrée des SIP (check antivirus et format du SIP)	ingest-external	/vitam/tmp					
métier	Fichier de définition d'un référentiel	functional-administration				REF : X		
métier	Référentiels métier : règles de gestion, formats, contrats, contextes, profils de sécurité, . . .	functional-administration		ES + MongoDB		REF : X		
	certificats SIA & personae	security-internal		MongoDB				
métier	Etat des workflows (en cours, en pause, terminés)	processing			X			
	Définition des workflows de traitement	processing	/vitam/conf					REF : X
métier	Données en cours de traitement par un processus	worker	/vitam/tmp		REF : X			
métier	Registre des fonds	functional-administration		ES + MongoDB				
métier	JOP (Journal des opérations)	logbook		ES + MongoDB		REF : X		
métier	JOP sécurisé (Journal des opérations sécurisé)	logbook				REF : X		
métier	JCV Unit / ObjectGroup	logbook		MongoDB		REF : X		
métier	JCV sécurisé	logbook				REF : X		
métier	ArchiveUnit (AU), ObjectGroup (GOT)	metadata		ES + MongoDB		REF : X		
métier	BinaryObject (BDO)	storage				REF : X		
métier	Stratégies de stockage	storage	/vitam/conf					REF : X
métier	Journal des écritures	storage	REF : /vitam/log					
métier	Sécurisation du journal des écritures	storage				REF : X		
technique métier	Log des écritures dans une offre de stockage	offer					MongoDB (offres)	
technique	Logs logiciels	(tous)	/vitam/log				ES (log)	
technique	Métriques applicatives	(tous)					REF : ES (log)	

Suite sur la page suivante

Tableau 1 – suite de la page précédente

Type	Données prises en charge par le module	Composant responsable de la donnée	Persistence locale	Persistence BDD métier	Persistence Workspace	Persistence Storage	Persistence BDD techniques	Persistence inventaire ansible
technique	Données de configuration (incl. certificats)	(tous)	/vitam/conf					REF : X

Quelques remarques :

- Si une donnée est persistée à plusieurs endroits, l'emplacement de référence (i.e. faisant foi en cas de désynchronisation entre les emplacements) est indiqué par le préfixe `REF:`. Les processus de reconstruction ou de remise en cohérence de la solution logicielle s'appuient sur cet emplacement référentiel pour alimenter les autres emplacements de stockage. En particulier, les offres de stockage *VITAM* portent la référence des données concernant les archives hébergées par le système :term : *VITAM* : leur contenu binaire (*BDO*), mais également les métadonnées associées au sens large (*AU*, *GOT*, journaux) et les référentiels métier
- Les données de référence à l'origine du registre des fonds sont les journaux opération (JOP)
- Il existe 2 types de journaux d'écriture :
  - Le premier, au niveau du moteur de stockage, qui permet de s'assurer de la bonne prise en compte des écritures par le système *VITAM*. Il s'agit d'un journal métier, participant à la preuve systémique (il est donc sécurisé comme les journaux d'opération et de cycle de vie des archives) ;
  - Le deuxième, au niveau de l'offre de stockage, qui permet de conserver l'ordre d'écriture des éléments stockés pour permettre leur rejeu lors d'une reconstruction (totale ou partielle). Il s'agit donc d'un journal technique, s'inspirant fortement du concept des `archivelog` des bases de données.

4.2.2 Stockage et stratégies

Voir aussi :

La déscription complète et les usages dans la documentation externe dédiée (« Gestion de multiples stratégies de stockage »).

Le stockage des données est pris en charge par le moteur de stockage. Celui-ci est en charge de la gestion du stockage de type *Persistence Storage* par le biais des offres de stockages. Le moteur de stockage s'appuie sur des stratégies de stockage pour définir la distribution des écritures dans les offres de stockage avec :

- la stratégie de stockage de plateforme *default* (obligatoire)
- une ou plusieurs stratégies additionnelles (optionnel)

La répartition posible des données selon les types de stratégies est alors la suivante :

TABLEAU 2: Inventaire des données selon le type de stratégie VITAM

Type	Données prises en charge par le module	Default strategy	Additionnal strategy
métier	Fichier de définition d'un référentiel	REF : X	
métier	Référentiels métier	REF : X	
métier	JOP (Journal des opérations)	REF : X	
métier	JOP sécurisé (Journal des opérations sécurisé)	REF : X	
métier	JCV Unit / ObjectGroup	REF : X	REF : X
métier	JCV sécurisé	REF : X	
métier	ArchiveUnit (AU), ObjectGroup (GOT)	REF : X	REF : X
métier	BinaryObject (BDO)	REF : X	X

Suite sur la page suivante

Tableau 2 – suite de la page précédente

Type	Données prises en charge par le module	Default strategy	Additionnal strategy
métier	Sécurisation du journal des écritures	REF : X	

Les stratégies additionelles utilisées doivent déclarer au moins une offre dite *référente* pour le stockage des ArchiveUnit (AU), ObjectGroup (GOT) et de leur JCV. Pour le stockage des BinaryObject (BDO) il n'y a aucune règle particulière.

Prudence : L'utilisation en mode standard de *VITAM* est le déploiement mono-stratégie (ie. avec uniquement la stratégie de plateforme *default*). Le déploiement multi-stratégies (ie. avec les stratégies additionnelles) est considéré comme un mode avancé qui ne doit être utilisé que si le besoin a été identifié.

4.2.3 Multisite

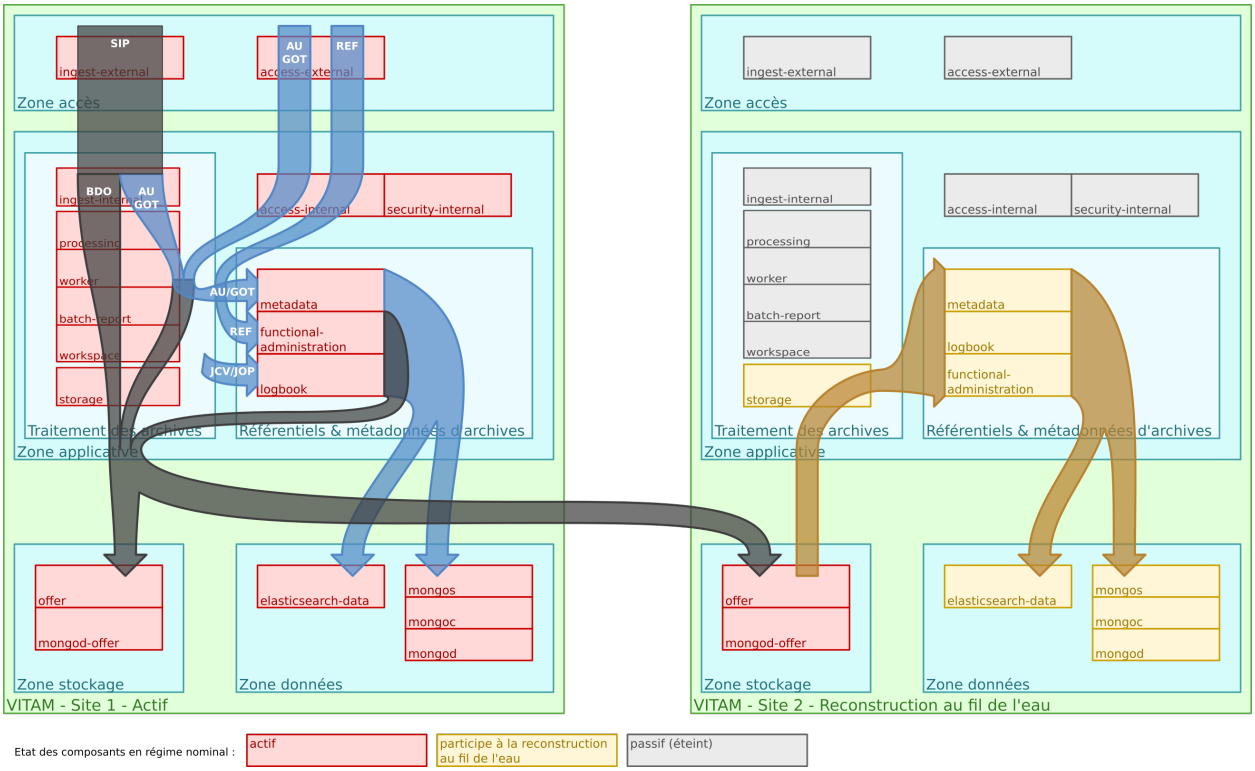


FIG. 5 – Architecture des données d'archives ; fonctionnement multisite.

Le fonctionnement multisite s'appuie fortement sur les capacités de reconstruction de *VITAM* :

- VITAM doit être déployé avec la stratégie de stockage de plateforme *default* comportant une offre de stockage sur chaque site ;
- Le fonctionnement de VITAM sur plusieurs sites fonctionne sur un principe actif / passif :
  - le site principal fonctionne en mode nominal,
  - le site secondaire fonctionne en mode « reconstruction au fil de l'eau » (les tâches planifiées de sécurisation et d'audit sont arrêtées, les composants frontaux et de traitement de données sont arrêtés (en gris dans le schéma précédent), les tâches planifiées de reconstruction au fil de l'eau sont activées)

- Toute donnée liée aux archives est systématiquement écrite dans les offres de stockage (le cas échéant, en même temps que dans les bases de données), donc sur les 2 sites en même temps ;
- Sur le site secondaire, des processus viennent régulièrement récupérer les données écrites en dernier dans l'offre de stockage de ce site (en se basant sur le contenu des logs d'écriture de l'offre) pour alimenter en update le contenu des bases de données « secondaires » :
  - Référentiels : reconstruction régulière et totale
  - *AU*/*GOT*/*BDO*/Journaux/Graphe : reconstruction au fil de l'eau

En cas de perte du site primaire, l'intégralité des données est donc présente dans le stockage sur le site secondaire, et est presque entièrement reconstruite dans les bases de données du même site. Une fois la reconstruction complètement terminée, le site secondaire est donc accessible ; le niveau d'accessibilité dépendra de la stratégie de stockage sur le site secondaire :

- Soit la dégradation du niveau de résilience des offres est acceptée, et la stratégie de stockage devra être modifiée pour limiter les écritures à une seule offre.
- Soit cette stratégie continue à requérir l'écriture sur 2 offres de stockage, et le système ne sera accessible qu'en lecture seule ; seule une recréation de l'offre de stockage sur le site principal permettra le retour à un fonctionnement nominal (Cf. admonition ci-dessous). Ce scénario est délicat à implémenter, et nécessite notamment la mise en place d'un contrat d'accès spécifique permettant de bloquer les accès en modification.

Prudence : En cas de bascule de site (*PRA*), les traitements en cours sur le site 1 sont perdus ; en particulier, les ingests non terminés doivent être renvoyés à *VITAM* et les autres *batchs* en cours doivent être relancés. L'incohérence des données sera réglée dans une version ultérieure du système *VITAM*.

4.2.4 Stratégies & multisite

Le fonctionnement multisite multi-stratégie suit le même principe que le mode mono-stratégie.

Pour respecter les normes de l'architecture multisite ainsi que ces processus associés, des règles supplémentaires spécifiques au mode avancé multi-stratégies doivent être respectées :

- La procédure de reconstruction utilise la notion d'offre dite « référente ». Il s'agit d'un groupe d'offres qui doivent contenir TOUTES les données nécessaires à la reconstruction d'un site Vitam à partir des données des offres de stockage. Il est donc obligatoire d'avoir un groupe d'offres de stockage dites « référente » par site, servant de source pour ces données, en vue de garantir la reconstruction. De plus pour des raisons de performance de la reconstruction les données contenues dans ces offres doivent être disjointes entre les offres.

Note :

Les données nécessaires à la reconstruction des bases de données sont :

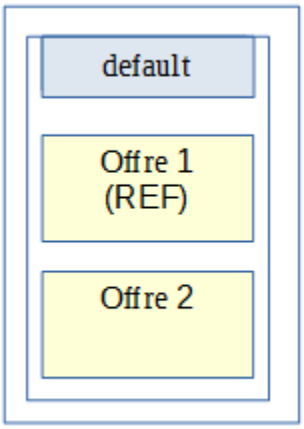
- les métadonnées des unités archivistiques et groupes d'objets techniques ainsi que leur journal de cycle de vie,
- les données relatives aux référentiels,
- les journaux d'opérations.

- La procédure de resynchronisation d'une offre permet de remettre en cohérence le contenu d'une offre à partir d'un autre offre. Pour que ce mécanisme marche il est nécessaire que les offres source et cible de la resynchronisation soient configurées pour être des copies. Les stratégies utilisées doivent être configurées pour contenir qu'une offre aie au moins toujours une autre offre mirroir contenant les même données.

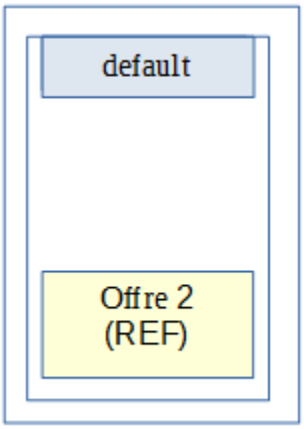
4.2.4.1 Mode standard : exemple d'architecture mono-stratégie

Il s'agit du mode par défaut de la solution logicielle Vitam. Dans ce cas nous avons uniquement la stratégie de plateforme *default* déclarant deux offres de stockage avec deux sites.

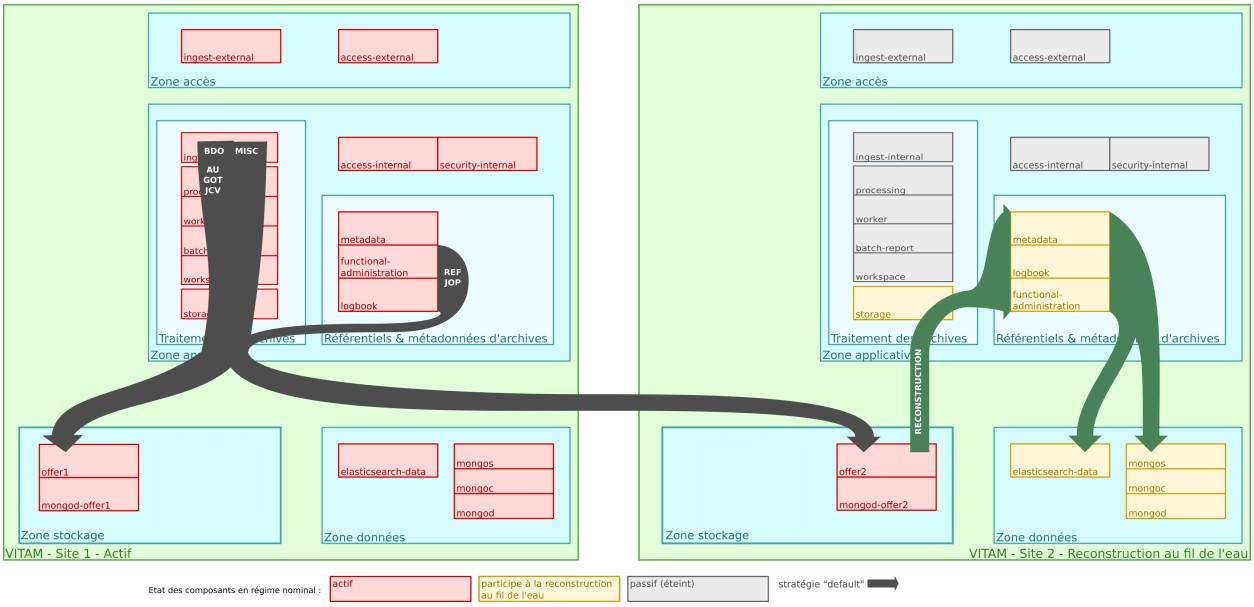
Stratégies du site principal :



Stratégies du site secondaire :



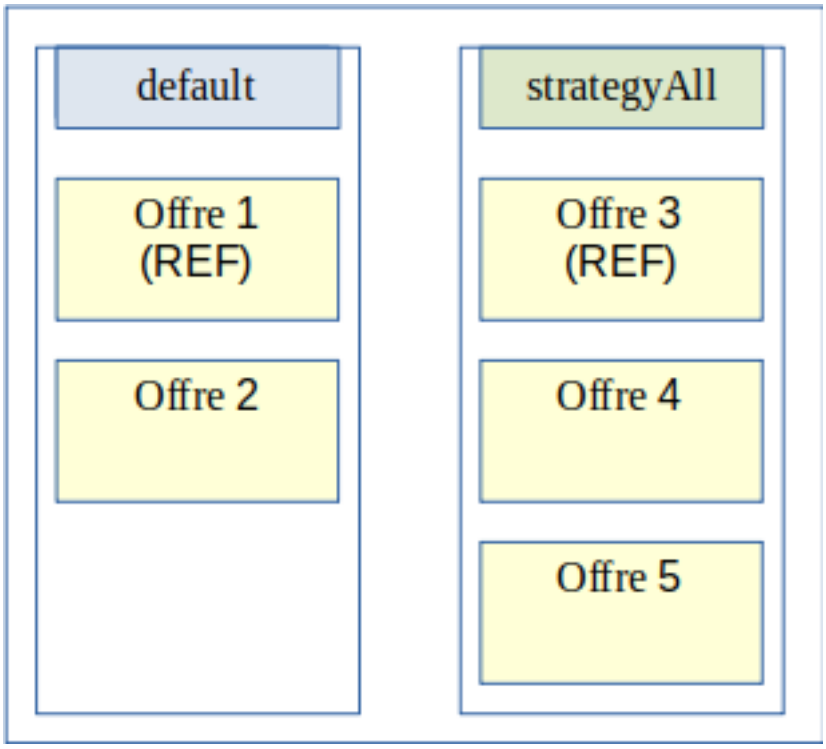
Flux de stockage :



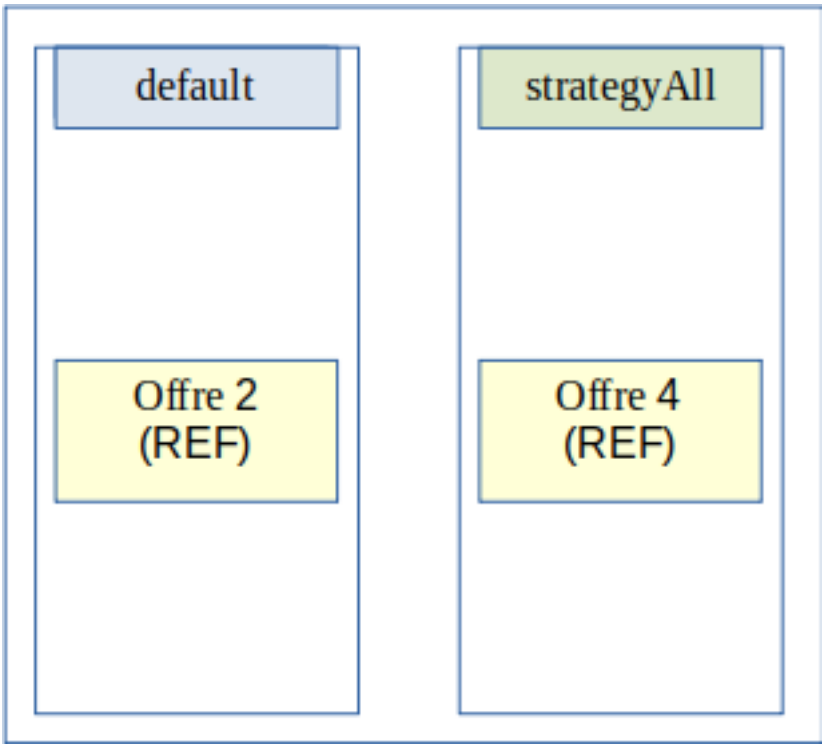
4.2.4.2 Mode avancé : exemple d'architecture multi-stratégie orienté Qualité de service

Le but d'un déploiement orienté **Qualité de service** de la solution logicielle Vitam est de fournir la possibilité de proposer un nombre de copies stockées différemment en fonction des applications utilisatrices de la plateforme *VITAM*.

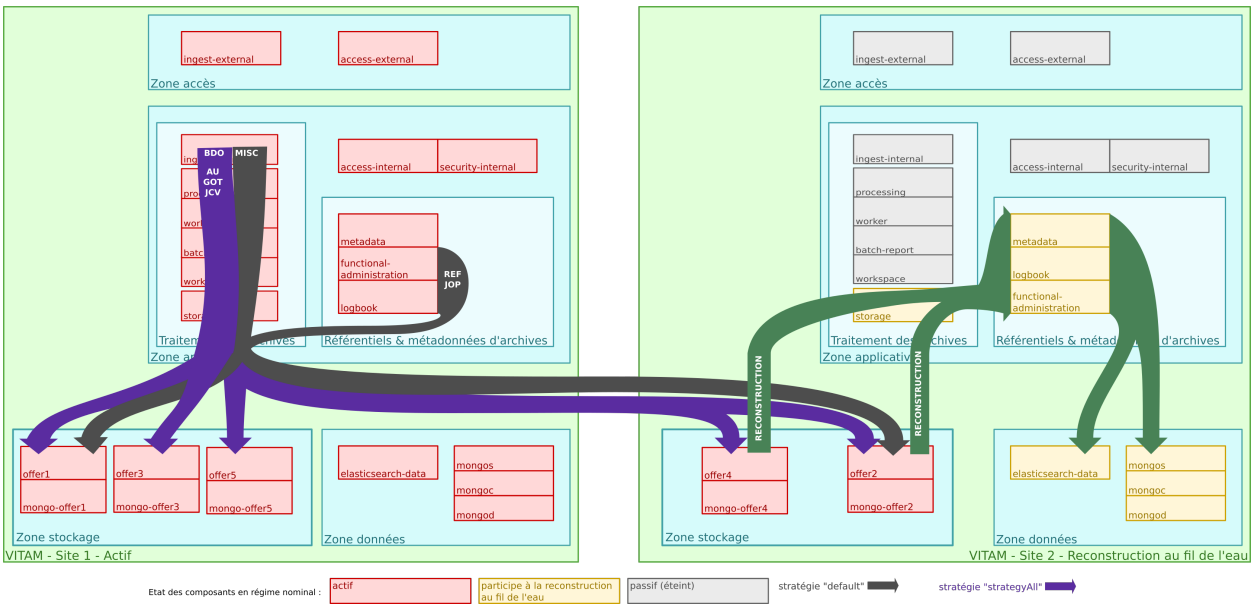
Stratégies du site principal :



Stratégies du site secondaire :



Flux de stockage :

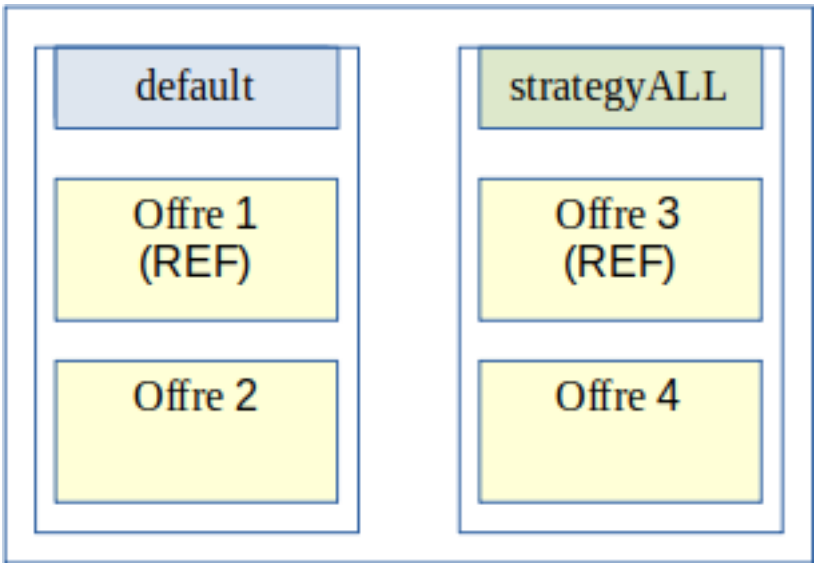


4.2.4.3 Mode avancé : exemple d'architecture multi-stratégie orienté Offres objets

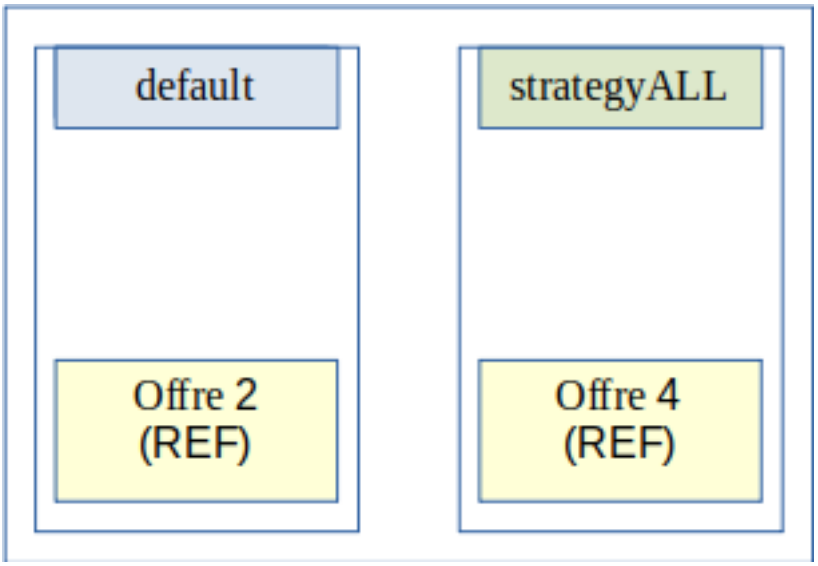
Le but d'un déploiement orienté **Offres objets** de la solution logicielle Vitam est de fournir la possibilité de stocker les objets numériques uniquement sur des offres séparée dites *objets* pour certaines ou toutes les applications utilisatrices de la plateforme *VITAM*. Ce type de déploiement d'offre peut être particulièrement adapté pour stocker des objets techniques uniquement sur des offres dites *froides* (sur bandes magnétiques).

Une offre dite *référente* peut être une offre de type synchrone (offre dite *chaude*) ou asynchrone (offre *froide*).

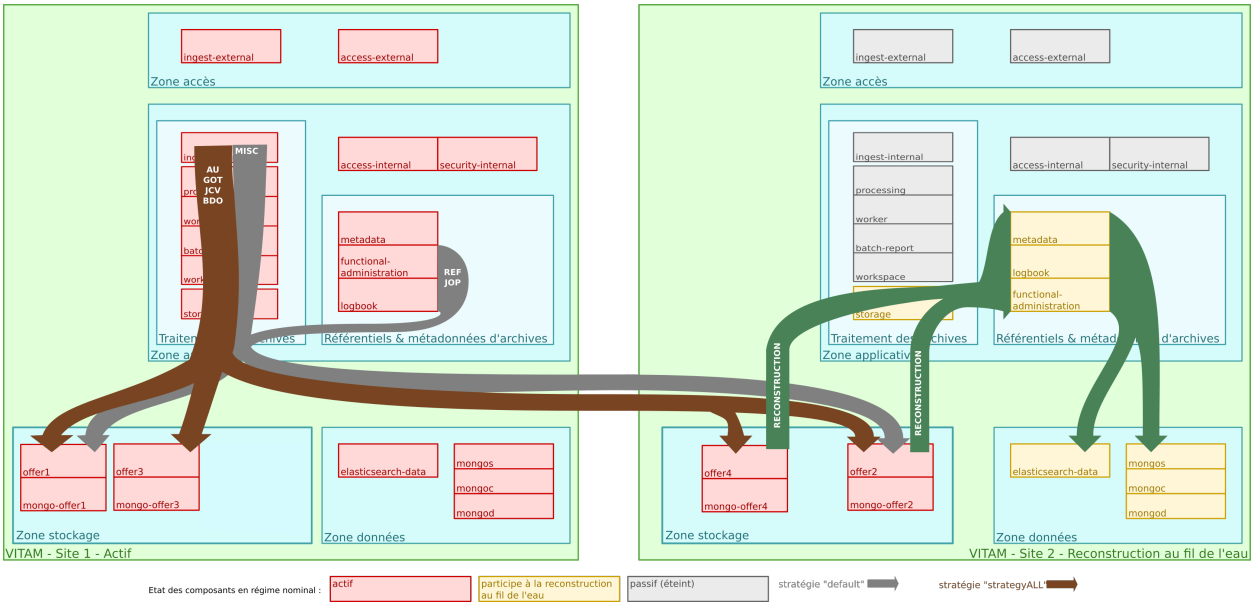
Stratégies du site principal :



Stratégies du site secondaire :



Flux de stockage :



4.3 Services métiers

Les services métiers sont présentés dans les sections suivantes ; pour chaque service, est indiqué son nom commun (en français), ainsi que le nom de service correspondant (en anglais, basé sur les usages *OAIS*).

4.3.1 API externes (ingest-external et access-external)

Rôle :

- Exposer les *API* publiques du système
- Sécuriser l'accès aux *API* de *VITAM*

Contraintes techniques :

- Authentification forte requise de la part des clients
- *WAF*

Données gérées :

- Pour `ingest-external` : *SIP* dans le sas d'entrée (conservés uniquement pendant leur analyse antivirus)

4.3.2 Moteur d'entrée (ingest-internal)

Rôle :

- Permettre l'entrée d'une archive *SEDA* dans le *SAE*

Fonctions :

- Upload HTTP de fichiers au format *SEDA*
- Persistance du *SEDA* dans `workspace`
- Lancement des *workflows* de traitements liés à l'entrée dans `processing`

Données gérées :

- Aucune

4.3.3 Moteur d'accès (access-internal)

Rôle :

- Permettre l'accès aux données du système *VITAM*

Fonction :

- Exposition des fonctions de recherche d'archives offertes par `metadata`
- Exposition des fonctions de parcours de journaux offertes par `logbook`
- Exposition des fonctions d'admnistration métier du système offertes par `functional-administration`

Données gérées :

- Aucune

4.3.4 Gestion des droits & accès (security-internal)

Rôle :

- Gérer le référentiel d'authentification des applications

Fonctions :

- Gestion des certificats d'accès des applications (*SIA*)
- Gestion des certificats personnels
- Gestion des *endpoints* nécessitant le contrôle des certificats personnels

Données gérées :

- Certificats des applications appelant *VITAM* (*SIA*)
- Certificats personnels (pour les *endpoints* nécessitant une authentification personae)

4.3.5 Moteur d'exécution (processing)

Rôle :

- Exécution massive de processus métiers complexes
- Utilisé notamment lors du versement et de la préservation

Fonctions :

- Découpage en micro tâches de processus métier (en fonction d'un référentiel)
- Supervision de l'état d'exécution de chaque « job »
- Reprise sur incident
- Traçabilité de l'ensemble des actions effectuées

Contraintes techniques :

- Grand nombre de tâches
- La durée d'exécution d'un ensemble de tâches peut être longue (ex : une campagne de transformation de document peut durer plusieurs semaines, voire plusieurs mois)
- Possibilité de devoir gérer des objets lourds ; cela implique notamment l'usage de l'espace de travail pour passer des informations entre tâches, et des optimisations (colocalisations ou copies directes) permettant de limiter les contraintes sur le réseau.

Données gérées :

- Etat des *workflows* en cours d'exécution

4.3.6 Espace de travail (workspace)

Rôle :

- Fourniture d'un espace pour l'échange de fichiers (et faire un appel par pointeur lors des appels entre composants) entre les différents composants de *VITAM*

Fonctions :

- Utilisation du moteur de stockage dans un mode minimal (opérations CREATE, READ, DELETE sur 1 seule offre de stockage)

Contraintes techniques :

- Être résilient à une panne simple

Données gérées :

- Données temporaires en cours de traitement

4.3.7 Worker (worker)

Rôle :

- Effectuer les traitements de masse sur les archives & paquets d'archive (*SIP* / . . .)

Fonction :

- Déclenchement des opérations sur requête du moteur d'exécution
- Gestion d'un cache local des éléments traités, en interaction avec l'espace de travail

Données gérées :

- Aucune ; il s'agit d'un composant de traitement pur

4.3.8 Moteur de données (metadata)

Rôle :

- Stocker de manière requêtable et rapide les métadonnées des objets (également stockées mais de manière pérenne dans l'offre de stockage)

Fonctions :

- Fournit une *API* agrégeant et abstrayant une technologie de base de données et un moteur d'indexation

Données gérées :

- Métadonnées et structures des archives : Archive Units, Object Group

4.3.9 Moteur de journalisation (logbook)

Rôle :

- Gérer les journaux métiers à fort besoin d'intégrité et potentiellement à valeur probante : journal du cycle de vie, journal métier (*SAE*/opérations + écritures)

Fonctions :

- Gestion des journaux (ajout, lecture)
- Sécurisation des journaux

Contraintes techniques :

- Besoin fort de fiabilité

Données gérées :

- Journaux de cycle de vie (JCV)
- Journaux d'opérations (JOP)
- Eléments de preuve issus de la sécurisation des journaux précédents

4.3.10 Gestion des référentiels (functional-administration)

Rôle :

- Gérer les réfentiels métier de la plate-forme

Fonctions :

- Gestion des référentiels métier *VITAM*

Données gérées :

- Référentiels techniques et métiers :
  - Formats
  - Règles de gestion
  - Contrats (d'entrée, d'accès)
  - Contextes
  - Profils
  - Arbre de positionnement
  - ...

4.3.11 Moteur de stockage (storage)

Rôle :

- Stockage des données (Métadonnées, Objets Numériques et journaux *SAE* et de l'archive)

Fonctions :

- Utilisation de stratégie de stockage (abstraction par rapport aux offres de stockage sous-jacentes)
- Gestion des différentes offres de stockage

Données gérées :

- Journaux d'écriture
- Sécurisation des journaux d'écriture

4.3.12 Offre de stockage (offer)

Rôle :

- Fournir une offre de stockage permettant la persistance des données

Fonctions :

- Offre de stockage
- Plusieurs modes de persistence sont supportés :
  - sur système de fichiers local
  - sur service de stockage objet compatible protocole Swift
  - sur service de stockage objet compatible protocole S3
  - sur bandes magnétiques (stockage dit *froid*)
- Log des écritures dans l'offre en permettant le rejeu

Données gérées :

- Tout ce qui doit être conservé à long terme (mais uniquement pour la gestion technique de ces données)

4.3.13 Interface de démonstration (ihm-demo)

Rôle :

- Permettre une utilisation basique de *VITAM*, notamment sans *SIA*

Fonctions :

- Représentation des arborescences et des graphes
- Formulaires dynamiques
- Suivi des opérations
- Gestion des référentiels

Contraintes techniques :

- *IHM* intuitive (sans *workflows* métiers), accessible (au sens *RGAA*), *responsive design*
- Compatibilité avec les navigateurs actuels
- Pas d'applets/clients lourds

Données gérées :

- Aucune

4.3.14 Scheduler

Rôle :

- Exécution des tâches périodiquement

Fonctions :

- Planifier une tâche
- Supervision de l'état d'exécution de chaque « job »
- Reprise sur incident

Contraintes techniques :

- Toutes les horloges des instances doivent être synchronisés

Données gérées :

- Aucune

4.3.15 Connecteur générique

Rôle :

- Permettre aux applications tierces de s'intégrer plus facilement dans Vitam via une API constructive d'un *SIP*

Fonctions :

- Exposer des *API* de construction de SIP

Contraintes techniques :

- Référentiels techniques et métiers de VITAM à respecter (Formats, Règles de gestion, Contrats (d'entrée, d'accès), Contextes . . .)

Données gérées :

- Données temporaires qui devraient servir à construire un *SIP* ingérable par VITAM

CHAPITRE 5

Architecture technique / exploitation

5.1 Principes d'architecture technique

Cette section vise à introduire l'environnement dans lequel s'intègrent les composants présentés à la section précédente et qui permet leur exploitation ; elle se concentre principalement sur les contraintes imposées à cet environnement et les choix d'interfaces techniques exposées et consommées avec l'écosystème logiciel d'exploitation.

5.1.1 Principes communs et environnement des services

5.1.1.1 Principes relatifs aux composants délivrés

Prudence : Dans la suite, les composants développés dans le cadre du projet VITAM seront appelés les « services VITAM » ; les composants intégrés, mais non développés, seront appelés les « COTS ».

5.1.1.1.1 Nommage

Dans la suite, on distinguera les identifiants différents suivants :

- ID de service (ou `service_id`) : c'est une chaîne de caractères qui nomme de manière unique un service. Cette chaîne de caractère doit respecter l'expression régulière suivante : `[a-z][a-z-]*`.
- ID de package (ou `package_id`) : il est de la forme `vitam-<service_id>`. C'est le nom du package à déployer.
- ID d'instance (ou `instance_id`) : c'est l'ID d'un service instancié dans un environnement ; ainsi, pour un même service, il peut exister plusieurs instances de manière concurrente dans un environnement donné. Cet ID a la forme suivante : `<service_id>-<instance_number>`, avec `<instance_number>` respectant l'expression régulière suivante : `[0-9]{2}`.

5.1.1.1.2 Principes relatifs aux services VITAM

Les services développés dans le cadre du projet VITAM interagissent avec un ensemble de composants externes dédiés à leur exploitation :

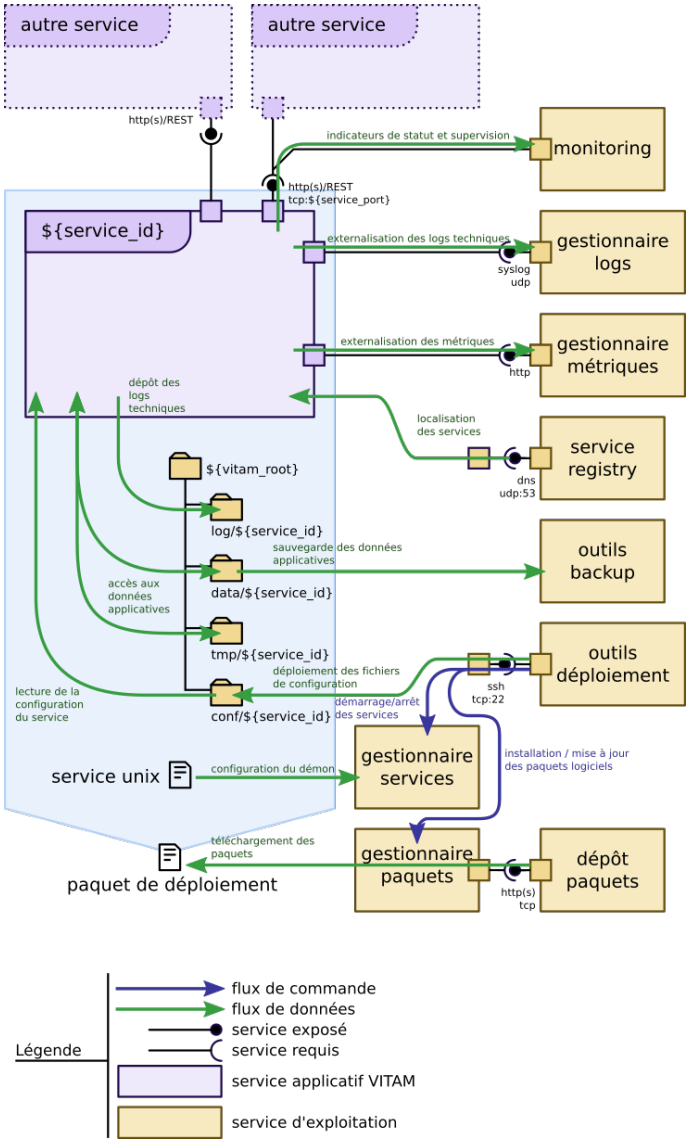


FIG. 1 – Environnement d'un service VITAM

Les interactions entre ces services et leur environnement se séparent essentiellement en 2 grandes familles :

- Les interactions avec des services externes ; on y trouve :
  - L'accès aux packages logiciels pour installation (Cf. *Packaging* (page 37)) ;
  - Le déploiement, permettant l'orchestration du déploiement de la solution (Cf. *Déploiement de la solution* (page 39)) ;
  - L'annuaire de services, permettant à chaque service de localiser les services dont il dépend et d'y accéder de manière indépendante de la topologie de déploiement ; cette section intègre ainsi également les principes de load-balancing et de haute disponibilité (Cf. *Principes sur les communications inter-services et le clustering* (page 35)) ;

    - Le monitoring, avec (Cf. *Suivi de l'état du système* (page 42)) :
        - L'accès offert au système de supervision aux données de monitoring exposées par les services (sur un port d'administration dédié) ;
        - L'export des logs applicatifs vers le sous-système de gestion des logs ;
- Les interactions locales au serveur, notamment avec des fichiers (dont la nomenclature est précisée dans *une section dédiée* (page 33)) :
    - L'installation, avec l'exécution du gestionnaire de paquets de l'OS (Cf. *Packaging* (page 37)) ;
    - La gestion des fichiers de configuration de l'application via l'outil de déploiement (Cf. *Déploiement de la solution* (page 39)) ;
    - Le démarrage / arrêt des services (Cf. *Administration technique* (page 45)) ;
    - La sauvegarde / restauration des données applicatives (Cf. *Gestion des données du système* (page 45)).

5.1.1.1.3 Principes relatifs aux COTS

Note : Les *COTS* correspondent aux éléments intégrés dans VITAM, mais dont le code source n'est pas maîtrisé par VITAM. Ils comprennent notamment les moteurs de base de données (ex : MongoDB, Elasticsearch)

De manière générale, les distributions binaires utilisées sont celles fournies nativement par les distributions linux, ou à défaut les paquets fournis par l'éditeur du logiciel.

Les *COTS* respectent les principes énoncés ci-dessus dans la mesure de leurs possibilités ; les éléments suivants sont notamment respectés :

- Le packaging logiciel : la nature des packages et les outils utilisés pour installer ces logiciels doivent être les mêmes que pour les autres composants VITAM.
- Le déploiement : les outils et principes de déploiement doivent également être identiques à ceux utilisés pour déployer les autres composants VITAM.
- L'arrêt / démarrage des services : ces logiciels doivent utiliser le même gestionnaire de services système que les autres composants VITAM.
- L'export des logs : les logs de ces logiciels doivent être envoyés à la chaîne de gestion de logs suivant les mêmes protocoles que les autres services ; par contre, le format des messages de logs peut être différent.

Voir aussi :

Les principes non respectés par les *COTS* (et qui concernent notamment les problématiques de LB/HA et de monitoring) sont détaillées dans *les sections de documentation associées* (page 77).

5.1.2 Utilisateurs, dossiers & droits

5.1.2.1 Utilisateurs et groupes d'exécution

La segmentation des droits utilisateurs doit permettre de respecter les contraintes suivantes :

- Assurer une séparation des utilisateurs humains du système et des utilisateurs système sous lesquels tournent les process système VITAM ;
- Séparer les droits des rôles d'exploitation différents suivants :
    - Les administrateurs système (OS) ;
    - Les administrateurs techniques des logiciels VITAM ;
    - Les administrateurs des bases de données VITAM.

Les utilisateurs et groupes décrits dans les paragraphes suivants doivent être ajoutés par les scripts d'installation de la solution VITAM. En outre, les règles de sudoer associées aux groupes `vitam*-admin` doivent également être mis en place par les scripts d'installation.

Les sudoers sont paramétrés en mode NOPASSWD, c'est à dire qu'aucun mot de passe n'est demandé à l'utilisateur faisant partie du groupe `vitam*-admin` pour lancer les commandes d'arrêt relance des applicatifs Vitam.

Note : Les fichiers de règles sudoers des groupes `vitam-admin` et `vitamdb-admin` seront systématiquement écrasés à chaque installation des paquets (rpm / deb) déclarant les utilisateurs VITAM. (Un backup de l'ancien fichier sera tout de même effectué).

5.1.2.1.1 Groupes

- `vitam` (GID : 2000) : il s'agit du groupe primaire des utilisateurs de service
- `vitam-admin` (GID : 3000) : il s'agit du groupe d'utilisateurs ayant les droits « sudo » permettant le le lancement des services VITAM
- `vitamdb-admin` (GID : 3001) : il s'agit du groupe d'utilisateurs ayant les droits « sudo » permettant le lancement des services VITAM stockant de la donnée.

5.1.2.1.2 Utilisateurs

- utilisateur de service ; les processus VITAM tournent sous cet utilisateur. Leur login est désactivé.
  - `vitam` (UID : 2000) : pour les services ne stockant pas les données
  - `vitamdb` (UID : 2001) : pour les services stockant des données (Ex : MongoDB et ElasticSearch)

5.1.2.2 Arborescence de fichiers

5.1.2.2.1 Services VITAM

Pour un service d'id `<service_id>`, les fichiers et dossiers impactés par VITAM sont les suivants.

5.1.2.2.1.1 Arborescence VITAM

L'arborescence `/vitam` héberge les fichiers propres aux différents services ; son arborescence interne est normalisée selon le pattern suivant : `/vitam/<folder_type>/<service_id>` où :

- `<service_id>` est l'id du service auquel appartient les fichiers ;
- `<folder-type>` est le type de fichiers contenu par le dossier :
  - `app` : fichiers de ressources (non-jar) requis pour l'application (ex : .war)
  - `bin` : binaires (le cas échéant)
  - `script` : Répertoire des scripts d'exploitation du module (start/stop/status/backup)
  - `conf` : Fichiers de configuration
  - `lib` : Fichiers binaires (ex : jar)
  - `log` : Logs du composant
  - `data` : Données du composant
  - `tmp` : Données temporaires produites par l'application

Les dossiers `/vitam` et `/vitam/<folder_type>` ont les droits suivants :

- Owner : `root`
- Group owner : `root`
- Droits : 0555

A l'intérieur de ces dossiers, les droits par défaut sont les suivants :

- Fichiers standards :
  - Owner : `vitam` (ou `vitamdb`)
  - Group owner : `vitam`
  - Droits : 0640
- Fichiers exécutables et répertoires :
  - Owner : `vitam` (ou `vitamdb`)
  - Group owner : `vitam`
  - Droits : 0750

Prudence : Cette arborescence ne peut contenir de caractère spécial ; les éléments du chemin (notamment le `service_id`) doivent respecter l'expression régulière suivante : `[0-9A-Za-z-_]+`

Le système de déploiement et de gestion de configuration de la solution est responsable de la bonne définition de cette arborescence (tant dans sa structure que dans les droits utilisateurs associés).

5.1.2.2.1.2 Intégration au système

- `/usr/lib/systemd/system/` : répertoire racine des définitions de units systemd de type « service » sur les distributions Linux type RedHat
- `/lib/systemd/system/` : répertoire racine des définitions de units systemd de type « service » sur les distributions Linux type Debian
- `<service_id>.service` : fichier de définition du service systemd associé au service VITAM

5.1.2.2.2 COTS

Les *COTS* utilisent la même nomenclature de répertoires et utilisateurs que les services VITAM, aux exceptions suivantes :

- Les fichiers binaires et bibliothèques utilisent les dossiers de l'installation du paquet natif.

5.1.3 Principes sur les communications inter-services et le clustering

5.1.3.1 Clusters applicatifs métier

Globalement, les principes de haute disponibilité et d'équilibrage de charge peuvent se diviser en 2 grandes catégories :

- Les principes utilisés par le service `worker` ;
- Les principes utilisés par les autres services (dans le cadre des appels REST).

Prudence : Dans cette version de VITAM, 2 composants ne sont pas déployables à plus d'une seule instance : `workspace` et `processing`. Pour le composant offer, il faut faire très attention, il peut être facilement multi-instancié (pour un type d'offre au sens vitam) avec des offres type swift ou s3, mais pas avec des technologies type filesystem standard.

5.1.3.1.1 Appels REST des services métier

Chaque cluster de service possède un nom unique de service (le `service_id`); chaque instance dans ce cluster possède un identifiant d'instance (`instance_id`).

Globalement, les services VITAM suivent les principes suivants lors d'un appel entre deux composants :

1. Le composant amont effectue un appel à l'annuaire de services en indiquant le `service_id` du service qu'il souhaite appeler;
2. L'annuaire de service lui retourne une liste ordonnée d'`instance_id`); c'est de la responsabilité de l'annuaire de service de trier cette liste dans l'ordre préférentiel d'appel (en fonction de l'état des différents services, et avec un algorithme d'équilibrage dont il a la charge);
3. Le composant amont appelle la première instance présente dans la liste. En cas d'échec de cet appel, il recommence depuis le point 1.

Note : Ces principes ont pour but de garantir les deux points suivants :

- Les clients des services doivent être agnostiques de la topologie de déploiement, et notamment du nombre d'instances de chaque service dans chaque cluster; la connaissance de cette topologie est déléguée à l'annuaire de service.
- Le plan de contrôle (choix de l'instance cible d'un appel) doit être décorrélé du plan de données (appel effectif), notamment dans un but de performance du plan de données.

5.1.3.1.2 Workers

Au démarrage, ces workers s'enregistrent auprès du composant processing; ensuite, les tâches sont distribuées par le processing aux différents workers. C'est donc processing qui a à sa charge la gestion de la distribution et de la résilience des workers.

5.1.3.2 COTS & clustering

La gestion de l'équilibrage de charge et de la haute disponibilité doit être intégrée de manière native dans le *COTS* utilisé.

Voir aussi :

Plus de détails seront apportés dans les chapitres spécifiques présent dans *la section* (page 77) décrivant en détail les contraintes techniques des différents services VITAM.

5.1.3.3 Annuaire de services (service registry)

La découverte des services est réalisée via l'utilisation du protocole *DNS*.

Note : Les avantages de l'utilisation de ce protocole sont multiples :

- Simple et éprouvé
- Connu des équipes d'exploitation

Le service DNS configuré lors du déploiement doit pouvoir résoudre les noms DNS associés à la fois aux `service_id` et aux `instance_id`. Tout hôte portant un service VITAM devra utiliser ce service DNS par défaut.

L'installation et configuration du service DNS applicatif est intégré à VITAM.

Voir aussi :

La solution de DNS applicatif intégrée à VITAM est présentée plus en détails dans *la section dédiée à Consul* (page 74).

5.1.4 Packaging

5.1.4.1 Principes communs

Tout package doit respecter les principes suivants :

- Nom des packages : `vitam-<id>` du package
- Version du package : Numéro de « release » du projet Vitam

Les dossiers (ainsi que les droits associés) compris dans les packages doivent respecter les principes dictés dans *la section dédiée* (page 33).

Note : Les limitations associés au format de packaging choisi (packaging natif, rpm / deb selon l'OS cible) sont :

- L'instanciation d'une seule instance d'un même moteur par machine (il n'est ainsi pas possible d'installer 2 moteurs d'exécution sur le même OS) ;
- La redondance de certains contenus dans les packages (ex : les librairies Java sont embarquées dans les packages, et non tirées dans les dépendances de package)

Les fichiers de configuration sont gérés par l'outil de déploiement de manière externe aux packages ; ils ne sont pas inclus dans les packages.

Les composants de la solution logicielle *VITAM* sont tous disponibles sous forme de packages natifs aux distributions supportées (rpm pour AlmaLinux 9 et 10, deb pour Debian 12 (bookworm)) ; ceci inclut notamment :

- L'usage des pré-requis (au sens Require ou Depends) nativement inclus dans la distribution concernée ;
- L'arborescence des répertoires OS de la distribution concernée ;
- L'usage du système de démarrage systemd.

Note : Seuls les paquets binaires cibles sont disponibles ; les paquets sources (SRPMS par exemple) ne seront pas fournis, les sources étant disponibles dans le dépôt git public.

5.1.4.2 Dépôts

Note : La typologie des dépôts présentée ci-dessous a notamment pour but de permettre l'installation de VITAM sur des environnements présentant un accès restreint à Internet (i.e. limité aux miroirs des dépôts d'update standard des distributions linux). Par conséquent, aucun dépôt externe autre que les dépôts natifs des distributions Linux n'est requis.

L'installation de VITAM s'appuie sur 2 dépôts internes différents :

- `vitam-product` : ce dépôt héberge les packages des logiciels développés dans le cadre de VITAM ; ces packages sont maintenus par VITAM, et les licences d'utilisations associées sont celles de VITAM.
- `vitam-external` : ce dépôt héberge les packages des logiciels requis par l'installation de VITAM mais non présents dans les dépôts natifs de la distribution. Ces packages sont fournis par VITAM, mais sont redistribués sans modifications de la part de VITAM. Ils ne sont en particulier pas maintenus par VITAM, et les licences d'utilisation restent celles des packages originaux.

Le contenu de ces dépôts est présent dans la distribution de la solution VITAM.

Note : Un dépôt supplémentaire, pour les *griffins*, permet de déployer également les briques logicielles relatives à la préservation.

Note : La création, configuration et initialisation des dépôts internes à partir des packages livrés est un pré-requis à l'installation de VITAM ; ces tâches ne sont pas incluses dans l'installation de la solution logicielle afin de pouvoir respecter la manière d'héberger des dépôts natifs de distributions Linux qui varie grandement selon les différents gestionnaires d'infrastructure.

Astuce : En outre, le programme VITAM mettra à disposition un miroir externe accessible sur Internet comportant les paquets à jour pour les dépôts internes mentionnés ci-dessus.

En plus des paquets logiciels livrés, l'installation de la solution VITAM requiert des dépôts nativement disponibles dans les distributions cibles.

5.1.4.2.1 AlmaLinux

VITAM s'appuie sur les dépôts suivants :

- AlmaLinux 9 (BaseOS, AppStream, Extras) et EPEL 9
- AlmaLinux 10 (BaseOS, AppStream, Extras, CRB) et EPEL 10

EPEL (Extra Packages for Enterprise Linux) est un dépôt maintenu par Fedora et fournit un ensemble de packages complétant ceux de RHEL/AlmaLinux.

5.1.4.2.2 Debian

VITAM s'appuie sur les dépôts suivants :

- Debian bookworm (dépôts main dans bookworm, bookworm-updates et security) : il s'agit des dépôts standard de la distribution
- bookworm-backports : il s'agit d'un backport de paquets plus récents non disponibles au moment de la publication de la version Debian

5.1.4.3 Prise en compte de la configuration dans le packaging

5.1.4.3.1 AlmaLinux

Conformément aux usages RPM de RedHat, les packages ne contiennent pas dans les pré/post action d'arrêt/démarrage/redémarrage de services.

Note : La configuration de démarrage des services et leur démarrage (a minima initial) est de la responsabilité de l'outillage de déploiement.

Contrairement aux usages de RPM, les fichiers de configuration ne seront pas gérés dans RPM. En effet, les fichiers de configuration seront instanciés par l'outil de déploiement. Pour éviter la génération de fichier .rpmnew ou .rpmsave, il ne sera pas utilisé la directive %config.

Prudence : À ce jour, les fichiers de configuration ne sont pas listés dans les fichiers de configuration des fichiers RPM ; par conséquent, ils n'apparaissent pas dans le résultats de commandes telles que `rpm -ql`.

5.1.4.3.2 Debian

Tout comme pour AlmaLinux, les paquets Debian n'intègrent pas les fichiers de configuration, et ne sont donc pas connus de dpkg ; en outre, ils ne s'intègrent pas dans debconf.

5.1.5 Déploiement de la solution

5.1.5.1 Principes de déploiement

Les principes généraux de déploiement sont les suivants :

- Les packages d'installation (rpm / deb) sont identiques pour tous les environnements ; seule leur configuration change.
- La configuration des services est externalisée et gérée par l'outillage de déploiement.
- Le déploiement est décrit intégralement dans un fichier de définition du déploiement. En dehors des pré-requis, le déploiement initial est automatisé en totalité (sauf exception).
- Les services sont configurés par défaut pour permettre leur colocalisation (dans le sens de la colocalisation de deux instances de deux moteurs différents) (ex : dossiers d'installation / de fonctionnement différents, ports d'écoute différents, ...).

Le déploiement s'effectue à partir d'un point central ; les commandes passées sur chaque serveur à partir de ce point central utilisent le protocole SSH.

Voir aussi :

Pour plus d'informations sur l'outillage de déploiement, se reporter à la section *sur l'outillage de déploiement* (page 73)

5.1.5.2 Contraintes et vue d'ensemble

Les zones logiques présentées dans *la section sur l'architecture applicative VITAM* (page 26) correspondent également aux zones de sécurité préconisées pour le déploiement de VITAM :

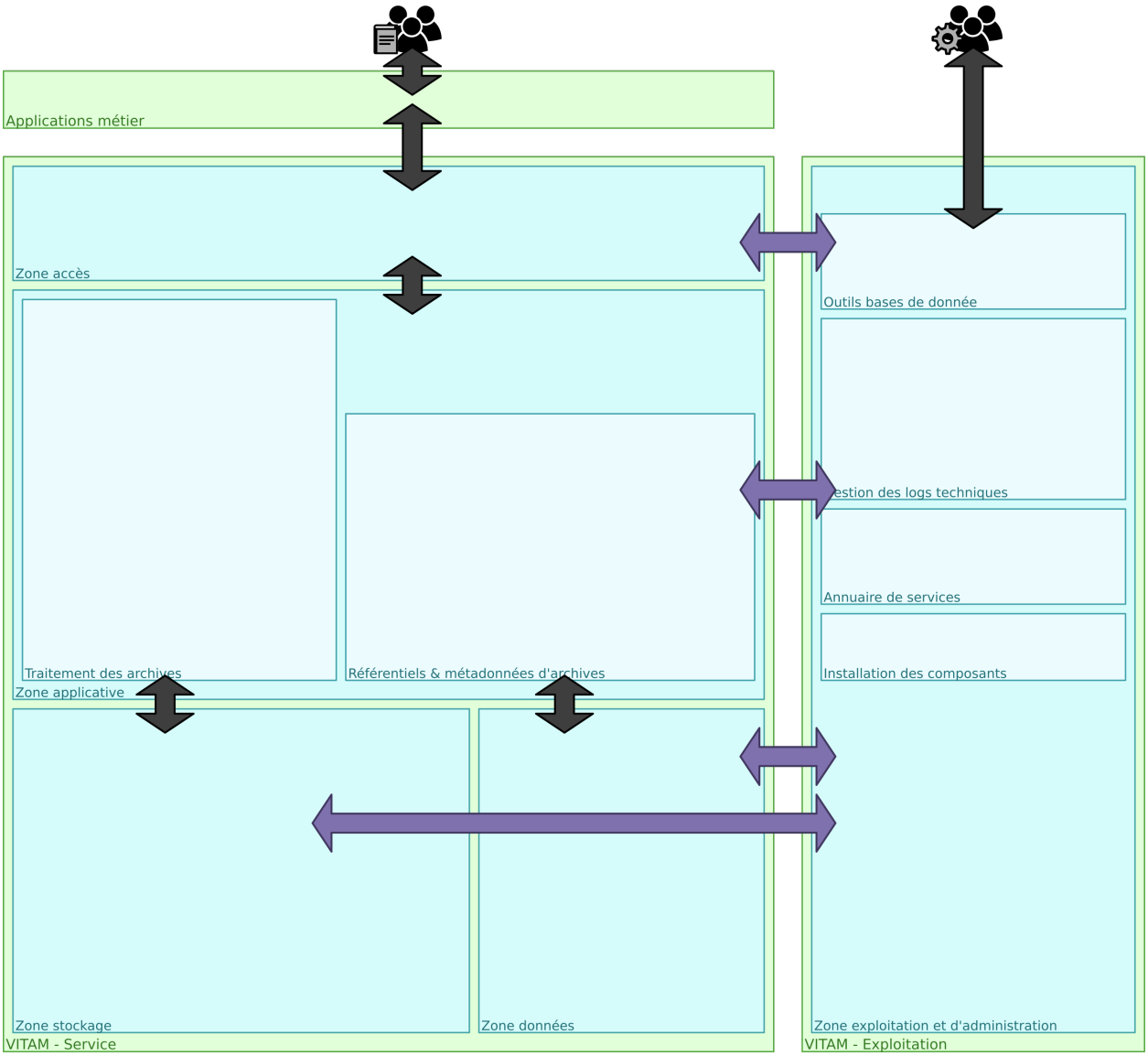


FIG. 2 – Déploiement VITAM : zones & principes de communication ; les utilisateurs métier archivistes sont présentés à gauche, et les exploitants technique à droite.

Voir aussi :

Ce découpage est repris dans *la présentation de l'architecture technique détaillée* (page 49).

Chaque zone héberge des clusters de services ; un cluster doit être présent en entier dans une zone, et ne peut par conséquent pas être réparti dans deux zones différentes. Chaque noeud d'un cluster applicatif doit être installé sur un hôte (OS) distinct (la colocalisation de deux instances d'un même service n'étant pas supportée) ; dans le cas de l'utilisation d'un système de virtualisation d'OS (type hyperviseur), il est recommandé de placer deux noeuds d'un même cluster applicatif sur deux serveurs physiques différents.

Le découpage en zones suit un découpage classique de système « n-tiers » ; par conséquent, il est prévu pour respecter les contraintes de flux inter-zones suivants :

- les systèmes externes utilisateurs de VITAM peuvent uniquement communiquer avec les services de la zone accès ;
- les services déployés dans la zone accès peuvent communiquer avec les services de la zone applicative ;

- les services déployés dans la zone applicative peuvent communiquer avec les services des zones stockage et données ;
- les services déployés dans les zones accès, applicative, stockage et données peuvent communiquer avec les services déployés dans la zone exploitation et administration ;
- les exploitants techniques peuvent accéder aux services déployés dans la zone exploitation et administration.

Voir aussi :

Un complément plus fin sur la problématique de colocalisation de composants est disponible dans *l'architecture technique détaillée* (page 92).

Prudence : Globalement, les connexions réseau associées aux flux métier se font dans le sens externe vers interne. Une exception à cette règle existe cependant au niveau des flux elasticsearch ; en effet, le fonctionnement des clients natifs elasticsearch (utilisés notamment dans metadata et functional-administration) impose des flux bidirectionnels entre les membres du cluster elasticsearch et ses clients natifs, donc entre la zone applicative et la zone données.

VITAM supporte l'installation sur une infrastructure dont les serveurs possèdent 2 interfaces réseau disjointes pour le réseau de service (sur lequel transitent les flux métier, représentés en noir dans les schémas) et le réseau d'administration/exploitation (sur lequel transitent les flux transverses à toutes les zones et la zone d'administration/exploitation, représentés en violet dans les schémas).

5.1.5.3 Installation initiale

Le processus de déploiement a les responsabilités suivantes :

- Effectuer une mise en conformité des OS des serveurs cibles pour certains pré-requis à l'installation de VITAM, notamment :
  - les utilisateurs, groupes et dossiers propres à VITAM ;
  - certains services système utilisés par VITAM (ex : rsyslog).
- Déployer, installer et configurer les composants logiciels VITAM ;
- Déployer certaines configurations de tuning système (ex : `sysctl.conf`, `limits.conf`).

Note : La portée des modifications appliquées au système sera décrite de manière plus précise dans la documentation d'installation livrée avec chaque version.

La portée de la configuration applicative est décrite dans le schéma présenté au paragraphe *Contraintes et vue d'ensemble* (page 39).

Voir aussi :

Plus de détails sur l'installation sont disponibles dans le *DIN*.

5.1.5.4 Principes de mise à jour à chaud

La mise à jour à chaud depuis une version précédente du système VITAM n'est pas supportée dans cette version de la solution VITAM.

5.1.5.5 Multi-site

Les principes de déploiement de VITAM sur plusieurs sites sont décrits dans la section *Architecture des données & multisite* (page 17).

5.1.5.6 Support de l'élasticité

Un déploiement de VITAM sur une infrastructure élastique (ex : AWS Auto Scaling, Azure AutoScaling, GCE managed instance groups , Openstack Heat AutoScalingGroup, ...) n'est pas supporté dans cette version de la solution VITAM.

5.1.5.7 Validation du déploiement

La validation du déploiement peut être réalisée à partir d'un ensemble de tests techniques et métier fournis par VITAM et permettant de valider le bon fonctionnement du système. A terme, ces tests seront exécutables même sur des environnements de production, dans un tenant dédié pour ne pas impacter les autres utilisateurs du système.

En particulier, les autotests des composants permettent d'avoir une première validation technique d'un déploiement.

5.1.6 Suivi de l'état du système

5.1.6.1 API de supervision

Chaque composant VITAM doit exposer en interne de la plate-forme, sur un port dédié, les API REST suivantes :

- `/admin/v1/status` : statut simple, renvoyant un statut de fonctionnement incluant des informations techniques sur l'état actuel du composant. Un exemple d'utilisation typique est l'intégration à un outil de supervision ou à un élément actif tiers (ex : load-balancer, ...) . L'appel doit être peu coûteux.
- `/admin/v1/autotest` : autotest du composant, lançant un test de présence des différentes ressources requises par le composant et renvoyant un statut d'état de ces resources.
- `/admin/v1/version` : statut renvoyant les informations relatives à la version.
- `/admin/v1/metrics` : Expose les métriques applicatives prometheus (techniques et métier).

Chaque VM de l'environnement VITAM doit installer prometheus node exporter. Ce dernier expose des métriques liées au matériel et au noyau du système via l'API suivante : * `/metrics` : Expose les métriques liées au matériel et au noyau.

Voir aussi :

D'autres interfaces de statut dédiées aux applications métier sont disponibles sur les composants externes (zone accès) ; elles sont décrites dans la documentation d'API de VITAM.

Ces API sont exposées sur un réseau d'administration qui peut être différent du réseau de service.

Voir aussi :

D'autres API d'administration sont disponibles selon les composants ; se reporter au paragraphe idoine dans *la liste des services* (page 77)

5.1.6.2 Métriques

Chaque composant VITAM doit permettre l'envoi ou l'exposition d'un certain nombre de métriques soit dans les logs de l'application, soit dans une base de données Elasticsearch, soit via des API au format prometheus ; ces métriques sont de 3 types différents :

- Les métriques relatives aux statistiques d'accès des interfaces REST :
  - Fréquence d'appel sur les dernière 1, 5 et 15 minutes ;
  - Nombre de résultats selon le code HTTP renvoyé ;
  - Avec un sampling des temps de réponses basé sur les 5 dernières minutes :
    - Le minimum

    - Le maximum
    - La moyenne
    - L'écart type
    - Le 95 ème percentile
- Les métriques relatives à l'usage de la JVM :
    - Consommation mémoire des différentes zones mémoire interne de la JVM
    - État des threads utilisés
    - Statistiques d'appels du/des ramasse-miette(s)
- Les métriques métier, relatives à des cas d'utilisation métier (archivistes) du système.

Note : VITAM propose un sous-système dédié à la collecte et exploitation des métriques qui s'appuie sur les composants également utilisés pour la gestion centralisée des logs ; il est décrit plus en détails dans *la section dédiée* (page 70).

Note : Chaque service applicatif VITAM dispose d'une documentation sur les différentes métriques exposées.

5.1.6.3 Logs

5.1.6.3.1 Protocoles : syslog

Les protocoles d'émission de logs (entre un émetteur de logs et l'agent syslog local) possibles sont :

- Le format syslog unix (écriture dans `/dev/log`), privilégié pour les messages émis par les scripts shell (protocole par défaut de la commande logger) .
- Le format syslog udp (sans garantie d'acheminement, vers l'adresse `localhost`), privilégié pour les messages émis par les applications.

Dans les deux cas, et en se basant sur la RFC 5424, les paramètres imposés sur les messages syslog sont les suivants :

- Facility : `local0` (id 21) ; Vitam n'utilise pas les facilités « système » mais seulement les facilités local0 à local3.
- Message Severity : dans le cas des applications Java, le mapping de sévérité suit le mapping imposé par l'appender logback SyslogAppender [12] (`DEBUG` 7, `INFO` 6, `WARN` 4 et `ERROR` 3).
- Le positionnement du champ `APP-NAME` correspondant à l'application ; pour les applications VITAM, ce champ doit être égal à l'id du composant vitam (devant respecter le pattern `vitam-.*`). Pour les scripts, il doit être égal au nom du script (comportement par défaut pour un logger unix).

Note : A noter que l'instance de l'application n'est pas mise dans le champ `APP-NAME` car du fait des principes de packaging, il ne peut y avoir qu'une seule instance d'application par hôte et le tuple (`HOSTNAME`, `APPNAME`) identifie bien l'application.

5.1.6.3.2 Types de log

Les logs se divisent en plusieurs catégories :

12. http://logback.qos.ch/manual/appenders.html#SyslogAppender

5.1.6.3.2.1 Logs applicatifs

Les logs applicatifs couvrent les logs produits par le code des applications ; ils permettent de suivre un certain nombre d'événements techniques et métiers remontés par les applications.

Leur format est imposé par VITAM (se reporter au *DEX* pour le format exact des logs).

Par défaut, ces logs sont déposés de deux manières différentes :

- des fichiers de logs (dans le répertoire de log dédié pour chaque composant (Cf. la *section dédiée* (page 33))). Ils sont configurés pour rouler quotidiennement, avec une taille globale maximale ; le pattern des fichiers est `<service_id>.%d.log` (`%d` étant remplacé par `yyyy-MM-dd`).
- le service syslog local, en utilisant le protocole syslog UDP (port 514 ; format défini dans la RFC3164).

Note : VITAM propose un sous-système dédié à la collecte et exploitation des logs qui s'appuie sur ce service syslog local pour l'acquisition des logs ; il est décrit plus en détails dans *la section dédiée* (page 65).

La corrélation des logs afférents à la même requête métier mais distribuée au sein des différents composants du système est réalisée grâce au positionnement d'un identifiant de requête au niveau des briques externes. Cet identifiant se retrouve dans tous les logs applicatifs, et est propagé entre les composants via l'usage du header HTTP `X-REQUEST-ID`.

Enfin, ces logs applicatifs transportent également les alertes émises par les composants VITAM, et notamment les alertes de sécurité.

5.1.6.3.2.2 Logs du garbage collector Java

Ces logs permettent de faire une analyse fine du fonctionnement interne de la JVM à travers les informations d'exécution des différents garbage collectors.

Leur format est imposé par l'implémentation de la *JVM*.

Ils sont déposés dans des fichiers (dans le répertoire de log dédié pour chaque composant (Cf. la *section dédiée* (page 33))) : `gc/gc.log` pour le fichier courant, `gc.log.<n>` pour les fichiers roulés (avec `<n>` le numéro du fichier, sur base 0). Le roulement est basé sur une limite de taille unitaire des fichiers, avec un nombre maximal de fichiers.

5.1.6.3.2.3 Logs d'accès

Les logs d'accès sont placés sur tous les services métiers VITAM ; ils permettent de tracer de manière fine (avec une granularité à la requête) les appels de ces services.

Leur format est imposé par VITAM (se reporter au *DEX* pour le format exact des logs).

Ces logs sont déposés dans des fichiers (dans le répertoire de log dédié pour chaque composant (Cf. la *section dédiée* (page 33))). Ils sont configurés pour rouler quotidiennement, avec une taille globale maximale ; le pattern des fichiers est `accesslog-<service_id>.%d.log` (`%d` étant remplacé par `yyyy-MM-dd`).

5.1.6.4 Suivi de l'état de déploiement

Le suivi de l'état de déploiement se fait au travers de l'outil de déploiement utilisé.

5.1.6.5 Intégration à un système de monitoring tiers

L'intégration à un système de monitoring tiers est possible via les points d'extension suivants :

- Les API REST de monitoring des composants Java
- L'utilisation des composants standards de monitoring des COTS utilisés

5.1.7 Administration technique

5.1.7.1 Démarrage / arrêt des services

Les services VITAM s'intègrent à systemd pour la gestion de leur cycle de vie (start / status / stop).

Le nom d'un service VITAM dans le gestionnaire de service de l'OS est par défaut son `package_id`.

Note : Le principe de lancement est de permettre le lancement des commandes de démarrage et d'arrêt des serices via sudo pour tous les utilisateurs membres du groupe `vitam-admin` (ou `vitamdb-admin`). La configuration sudoers des groupes `vitam-admin` et `vitamdb-admin` est fournie par VITAM. Les fichiers sudoers des groupes `vitam-admin` et `vitamdb-admin` seront systématiquement écrasés à chaque nouvelle installation (avec sauvegarde du fichier précédent dans le même répertoire). Le fichier sudoers est configuré en mode NOPASSWD, c'est à dire que le mot de passe de l'utilisateur ne sera pas demandé lors de l'utilisation des sudoers vitam.

5.1.7.2 Tâches régulières

Les tâches techniques devant être lancées à intervalles réguliers et ne nécessitant pas de coordination entre plusieurs serveurs sont implémentées de préférence à l'aide de units timer systemd [13].

5.1.8 Gestion des données du système

Dans VITAM, les principes de sauvegarde / restauration utilisés de manière classique ne peuvent être appliqués ; en effet, ils ne peuvent pas convenir dans le cadre de déploiements gérant une quantité massive d'archives. Par conséquent, des principes de sauvegarde et restauration applicatives particuliers sont mis en place dans le cadre de la solution, principalement basés sur l'utilisation des offres de stockage afin d'assurer la pérennisation des données.

VITAM doit être reconstructible à partir de 2 éléments qui sont :

- Les données persistées dans les offres de stockage ;
- La configuration & les données (notamment les secrets) de déploiement. Ces secrets incluent notamment les certificats (et notamment les certificats clients des applications externes et des personae).

Voir aussi :

La reconstruction et la vision applicative des données est abordée à la section *Architecture des données & multisite* (page 17).

5.1.8.1 Cas des déploiements de petite taille

Dans le cas de déploiement de petite taille, il est possible d'effectuer une sauvegarde classique du système, à froid.

13. https://www.freedesktop.org/software/systemd/man/systemd.timer.html

5.1.8.1.1 Dossiers

Les dossiers suivants sont éligibles aux processus de sauvegarde (dans l'ordre décroissant de criticité) :

- `/vitam/data` : ce répertoire contenant les données des applications, sa sauvegarde est vitale.
- `/vitam/tmp` : la sauvegarde de ce répertoire est optionnelle ; elle peut permettre de diminuer le temps de reprise après incident.
- `/vitam/conf` : la sauvegarde de la configuration est normalement peu utile, car tous les fichiers de configuration sont gérés par ansible, donc facilement réinstanciables.

Les autres répertoires sont intégralement fournis par les packages d'installation ; leur sauvegarde n'est donc pas indispensable.

5.1.8.1.2 Sauvegarde

La sauvegarde s'effectue pendant la nuit, avec globalement 5 phases :

1. Une phase initiale de suppression des possibilités d'écritures externes dans les bases de données (arrêt des composants frontaux) ;
2. Une phase d'attente de la fin des processus internes en cours (workflow d'entrée et sécurisation des journaux) ; cette phase se clôt par l'arrêt ordonné de tous les services VITAM ;
3. Une phase d'export des bases de données sous forme de fichiers ;
4. Une phase de sauvegarde des fichiers (avec a minima les fichiers d'archives et fichiers d'export des bases de données) ;
5. Une phase de redémarrage ordonné des services VITAM.

5.1.8.2 Restauration

La procédure de restauration s'appuie sur le postulat que le système VITAM est dans un état incohérent au début de celle-ci.

1. S'assurer que tous les services VITAM sont arrêtés.
2. Restaurer les dossiers précédemment sauvegardés à leur emplacement respectif.
3. Démarrer les services suivant la procédure de démarrage VITAM.

Note : La sauvegarde / restauration des bases MongoDB et Elasticsearch (data et logs) n'est pas indispensable. A contrario, la sauvegarde / restauration des bases MongoDB des offres, et notamment les offres froides (stockage sur bande) est fortement recommandée ; des détails sont fournis dans le *DEX*.

5.2 Services techniques fournis par la solution

5.2.1 Moteur de déploiement et de configuration

Rôle :

- Faciliter et centraliser la configuration, le déploiement et la mise à jour de *VITAM*

Fonctions :

- Gestion des binaires d'installations (version, intégrité)
- Gestion des éléments de configuration spécifiques à chaque plate-forme (y compris les secrets)

- Pilotage de l'installation des services sur les éléments d'infrastructure (*VM*/containers) de manière cohérente

Données gérées :

- Configuration technique du système *VITAM*
- Certificats x509 : le moteur de déploiement et de configuration doit posséder la référence des certificats techniques déployés sur la plate-forme (car il doit entre autres assurer la cohérence de ces certificats entre les différentes instances des composants *VITAM* déployés)

5.2.2 Chaîne de traitement de logs et de métriques

Rôle :

- Agréger, mettre en forme et exploiter les logs techniques du système
- Agréger, mettre en forme et exploiter les métriques techniques du système

Fonctions :

- Récupérer les éléments de logs provenant des composants du système
- Structurer les logs techniques
- Stocker les logs et métriques techniques
- Présenter des dashboards d'analyse et de recherche des logs et métriques techniques

Données gérées :

- Logs techniques
- Métriques techniques

5.2.3 Service registry

Rôle :

- Identifier la localisation et l'état (disponible / indisponible) des services *VITAM*

Fonctions :

- Maintenir une vision cohérente de l'état des services
- Fournir des interfaces de requêtage de la localisation des services

Données gérées :

- État et localisation des composants en cours d'exécution

5.3 Composants logiciels utilisés

Voir aussi :

La liste des dépendances logicielles exactes est décrite dans les release-notes de chaque version de *VITAM*.

5.3.1 Fournis

5.3.1.1 COTS

- MongoDB [14] : base de données orientée documents
- Elasticsearch [15] (+ plugins) : base d'indexation
- Cerebro [16] : IHM d'administration d'Elasticsearch
- Curator [17] : maintenance des index d'Elasticsearch
- Logstash [18] (+ plugins) : agrégation et traitement des logs
- Kibana [19] : dashboards et recherche des logs techniques et métier
- Consul [20] : annuaire de services
- Siegfried [21] : identification des formats de fichiers
- Prometheus node exporter [22] : Exposition des métriques liées au matériel et au noyau du système
- Prometheus Elasticsearch Exporter [23] : Exposition des métriques liées à Elasticsearch

Dans les extras, les outils supplémentaires suivants sont également fournis, sans garantie de bon fonctionnement :

- Head [24] : interface alternative pour les index d'Elasticsearch
- mongo-express [25] : interface d'accès au contenu de la base MongoDB
- Prometheus server [26] : Supervision
- Prometheus alertmanager [27] : Envoi des alertes
- Grafana [28] : Visualisation des données elasticsearch et prometheus
- restic [29] : Outil de sauvegarde

5.3.1.2 Bibliothèques structurantes

- Jetty [30] : moteur de servlet

Note : Jetty est utilisé en mode « embedded », et n'est par conséquent pas remplaçable par un autre moteur de servlet.

14. https://www.mongodb.com/fr
15. https://www.elastic.co/products/elasticsearch
16. https://github.com/lmenezes/cerebro
17. https://www.elastic.co/guide/en/elasticsearch/client/curator/current/index.html
18. https://www.elastic.co/fr/products/logstash
19. https://www.elastic.co/fr/products/kibana
20. https://www.consul.io/
21. http://www.itforarchivists.com/siegfried
22. https://prometheus.io/docs/guides/node-exporter/
23. https://github.com/prometheus-community/elasticsearch_exporter
24. https://github.com/mobz/elasticsearch-head
25. https://github.com/mongo-express
26. https://prometheus.io/
27. https://prometheus.io/docs/alerting/latest/alertmanager/
28. https://grafana.com/
29. https://restic.net/
30. https://eclipse.org/jetty/

5.3.2 Requis

- Java (*JRE*) 11

Voir aussi :

Pour chaque version du système *VITAM* * les composant fournis ou installés par dépendance sont précisés dans la documentation d'installation (*DIN*) ; * la liste des bibliothèques et *COTS* opensources inclus (ainsi que leur version) sont précisés dans les release-notes.

5.4 Architecture technique détaillée

L'architecture technique s'appuie sur la vision des flux d'information entre les composants d'une part, et le diagramme de déploiement applicatif d'autre part. Les schémas suivants décrivent les connexions réseau établies entre les différents clusters de composants (connexion tcp ou udp) ; on différenciera les flux de service (accès aux services, communication entre les services métier d'un même système) des flux d'administration (accès entre les services métier *VITAM* et les services d'infrastructure et d'exploitation).

Les détails sur les communications intra-cluster sont abordés plus en détail dans les paragraphes dédiés aux différents composants.

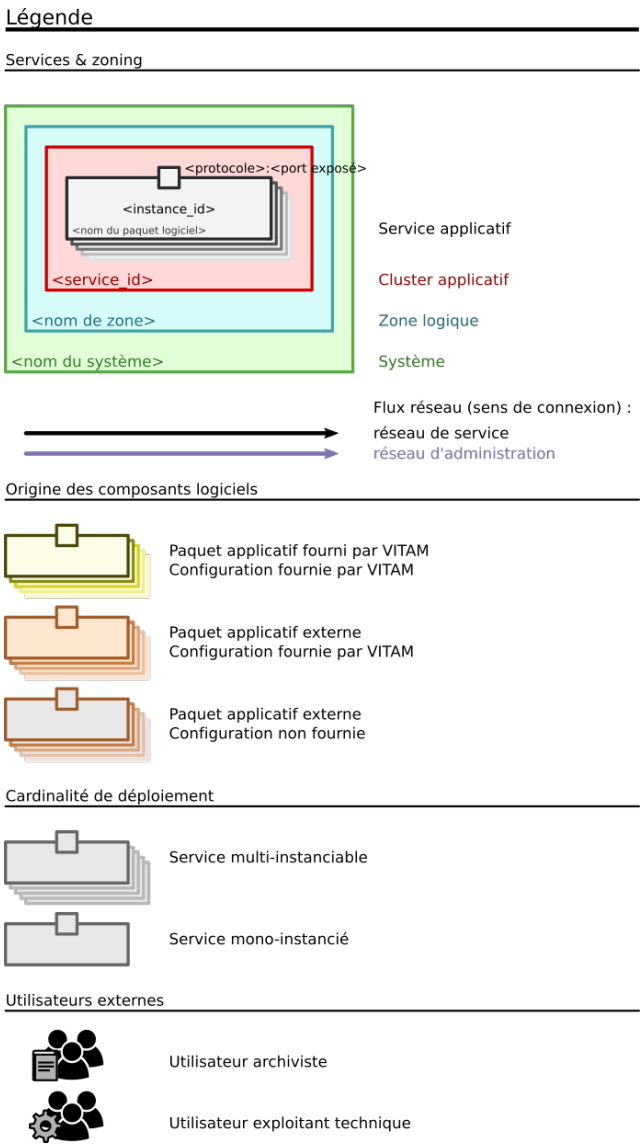


FIG. 3 – Architecture technique : légende

5.4.1 Flux métier

Les flux réseaux « métier » sont divisés en 3 schémas pour plus de clarté ; tout d'abord, les flux généraux :

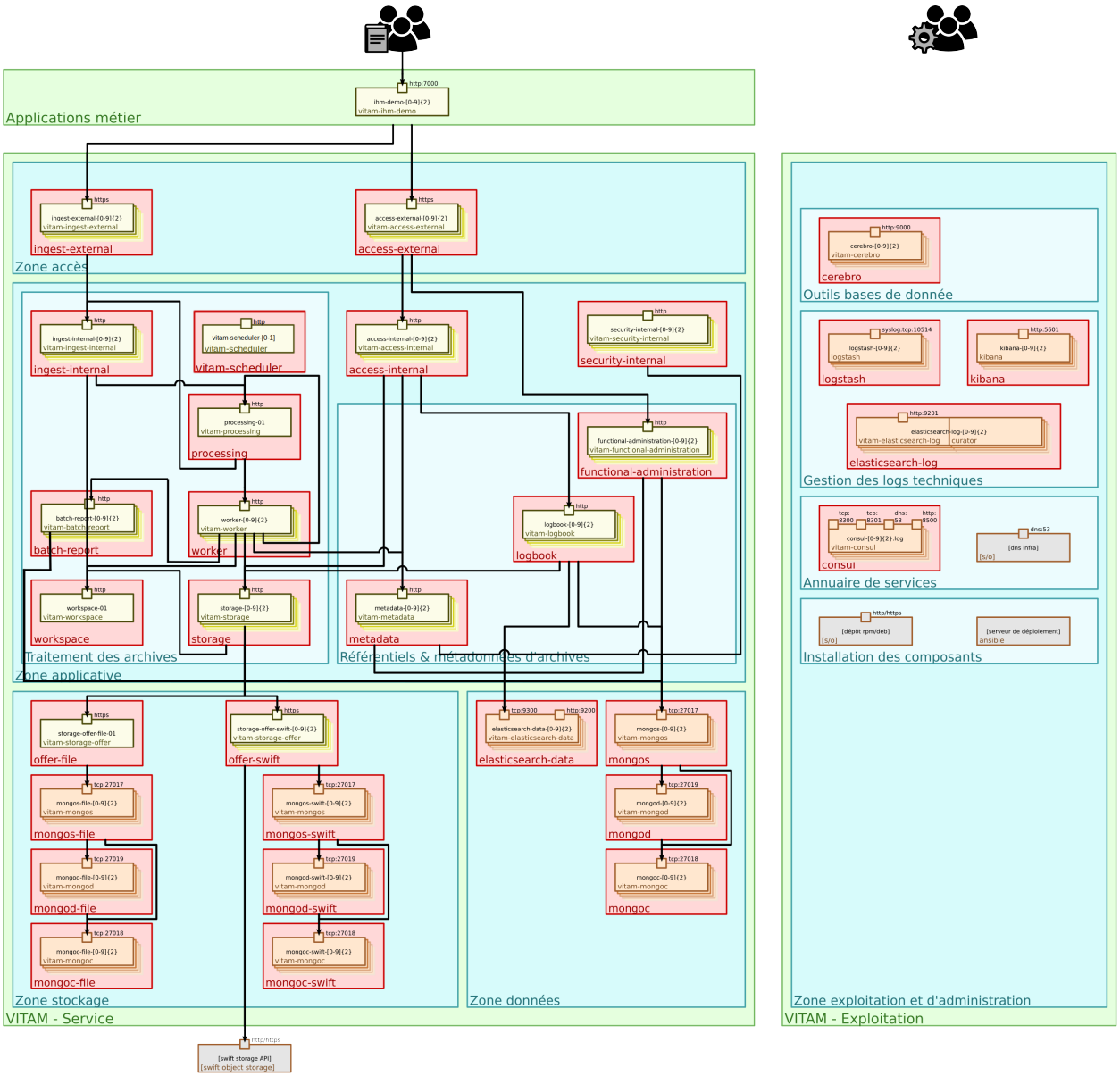


FIG. 4 – Architecture technique : flux (1/5 : flux métiers généraux)

Ensuite, les flux dédiés au dépôt des journaux dans le composant « logbook » :

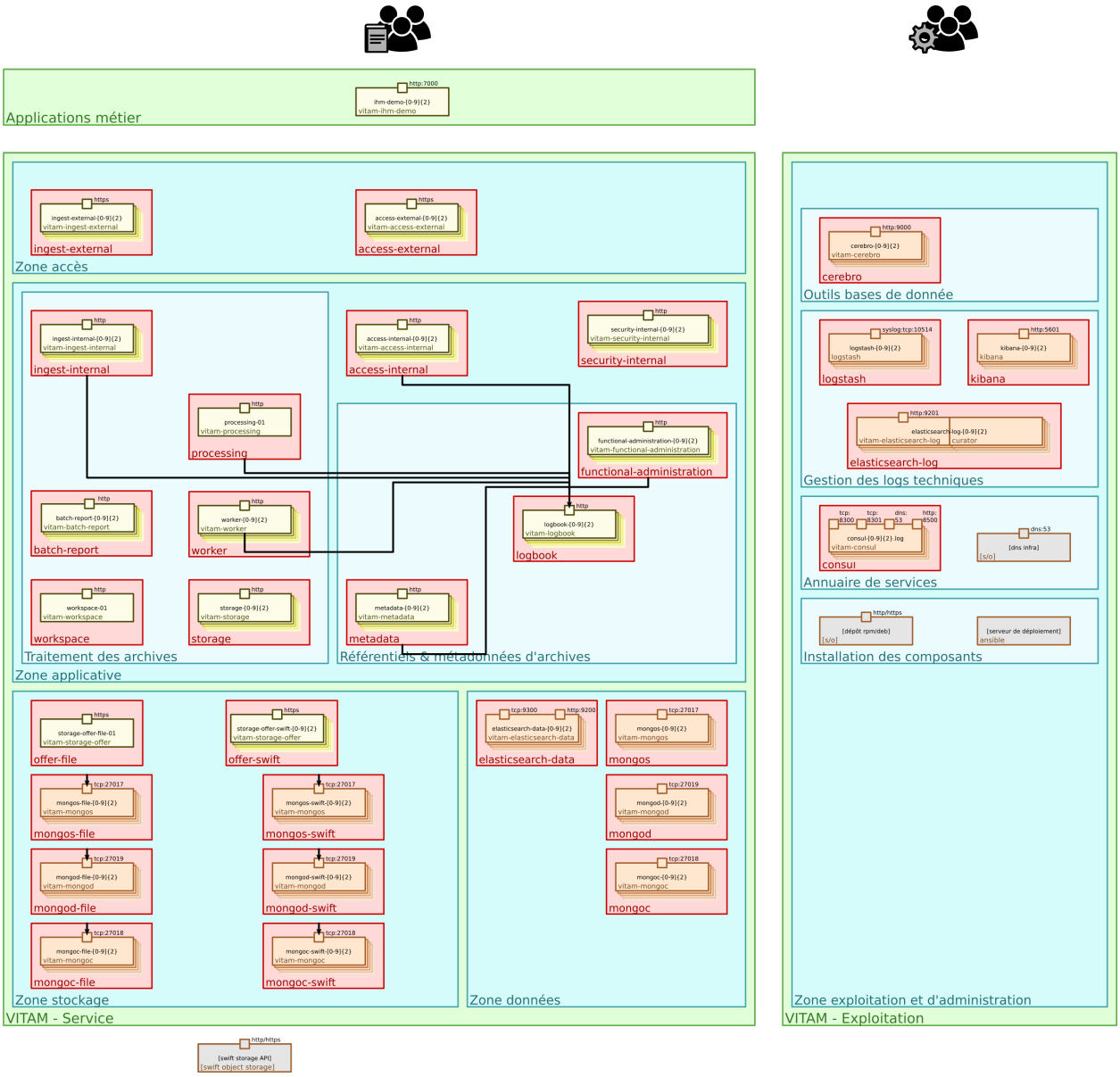


FIG. 5 – Architecture technique : flux (2/5 : flux métiers de dépôt des journaux)

Enfin, les flux dédiés à la lecture des référentiels en interne de *VITAM* :

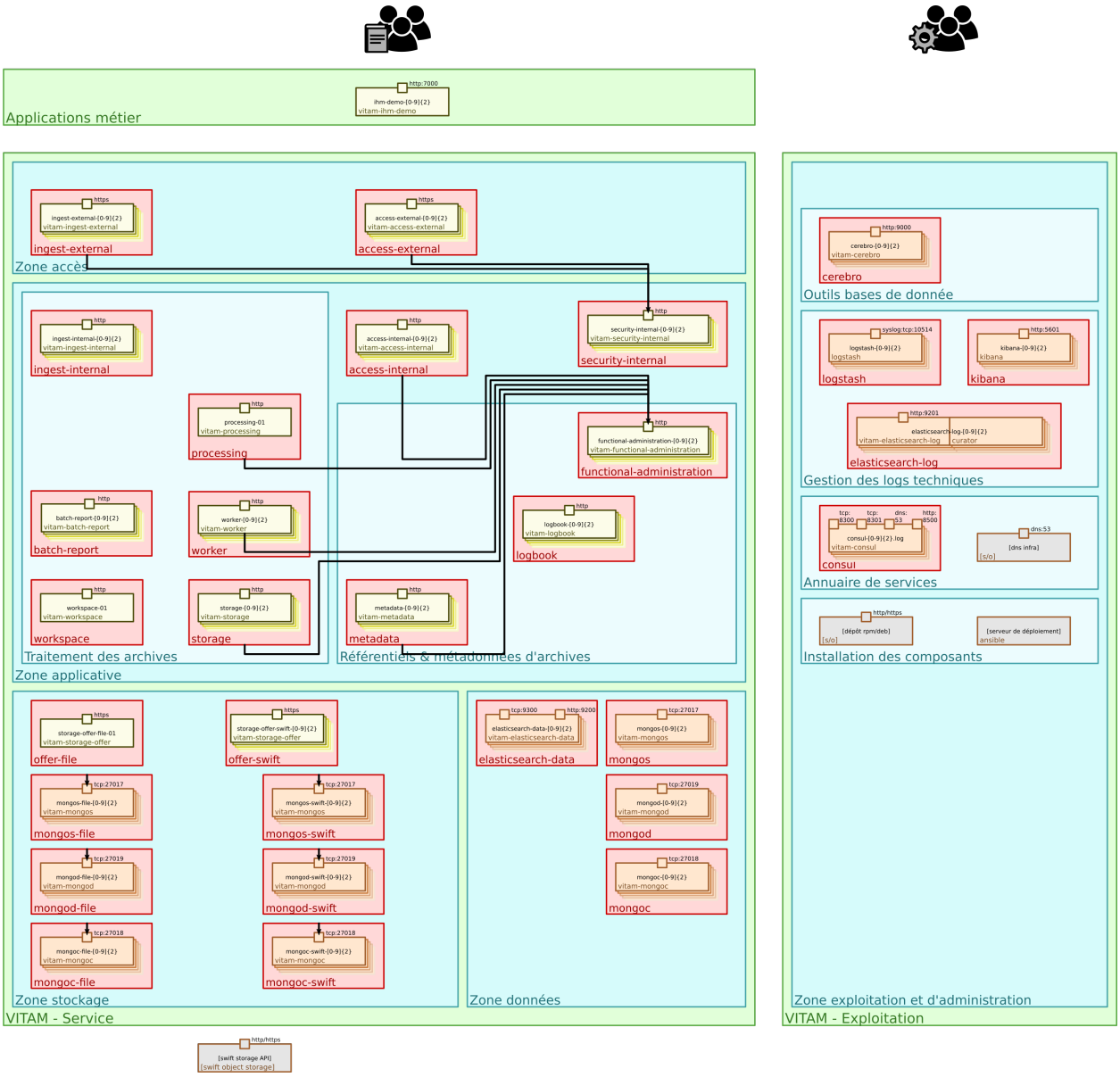


FIG. 6 – Architecture technique : flux (3/5 : flux métiers de lecture des référentiels métier)

5.4.2 Flux exploitation

Les flux réseau « exploitation » correspondent aux flux des utilisateurs exploitants vers les outils d'exploitation fournis par *VITAM* :

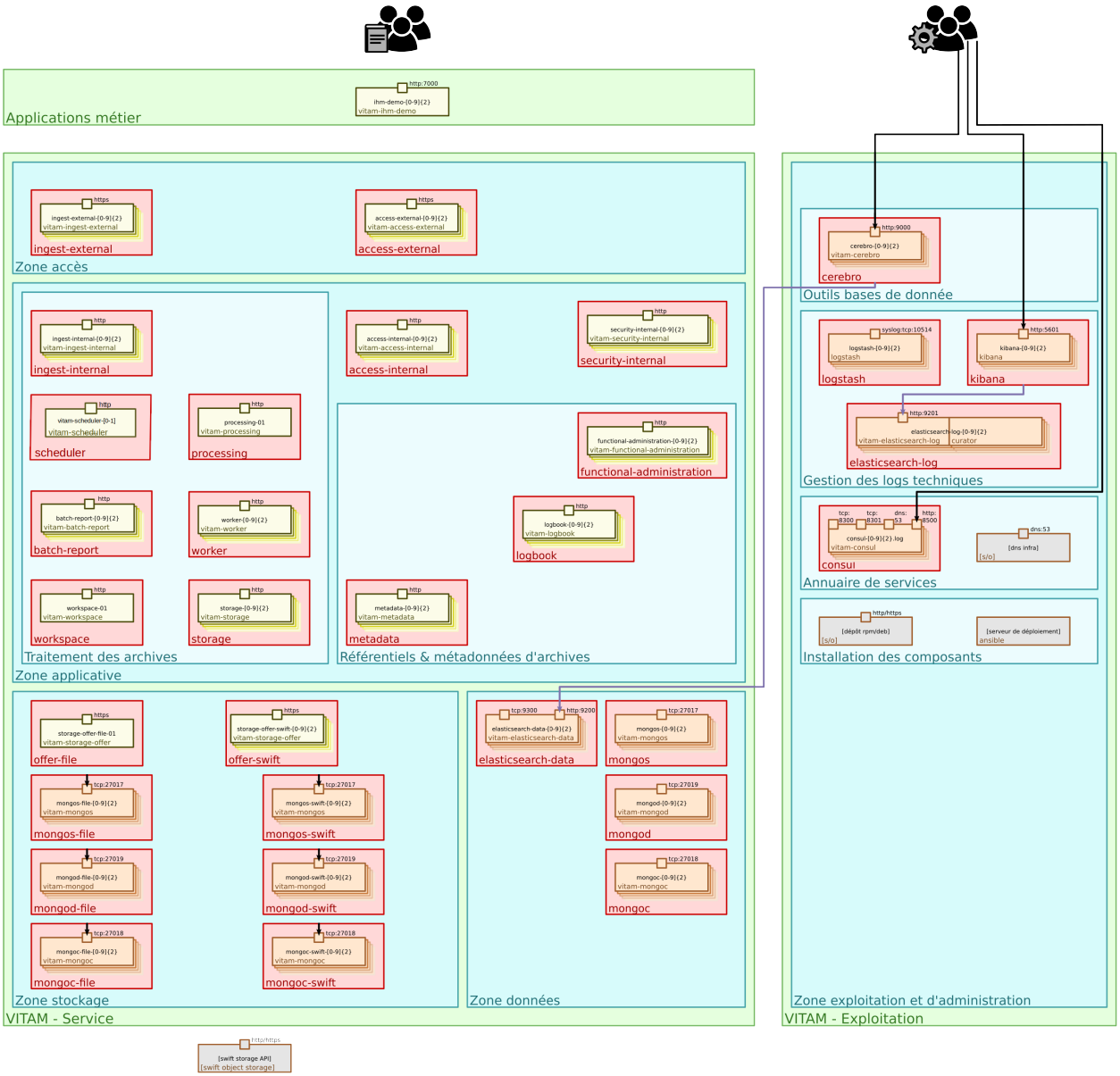


FIG. 7 – Architecture technique : flux (4/5 : flux des outils d'exploitation)

5.4.3 Flux techniques

A l'inverse des flux métier qui relient les composants instanciés de manière indépendante de leur topologie de déploiement (et notamment de leur colocalisation possible), les flux réseaux techniques sont centrés sur la communication entre des composants techniques d'exploitation liés à un hôte (*OS*) et des composants d'administration ; par conséquent, le schéma ci-dessous se répète pour tout serveur hébergeant un ou plusieurs composant(s) *VITAM* :

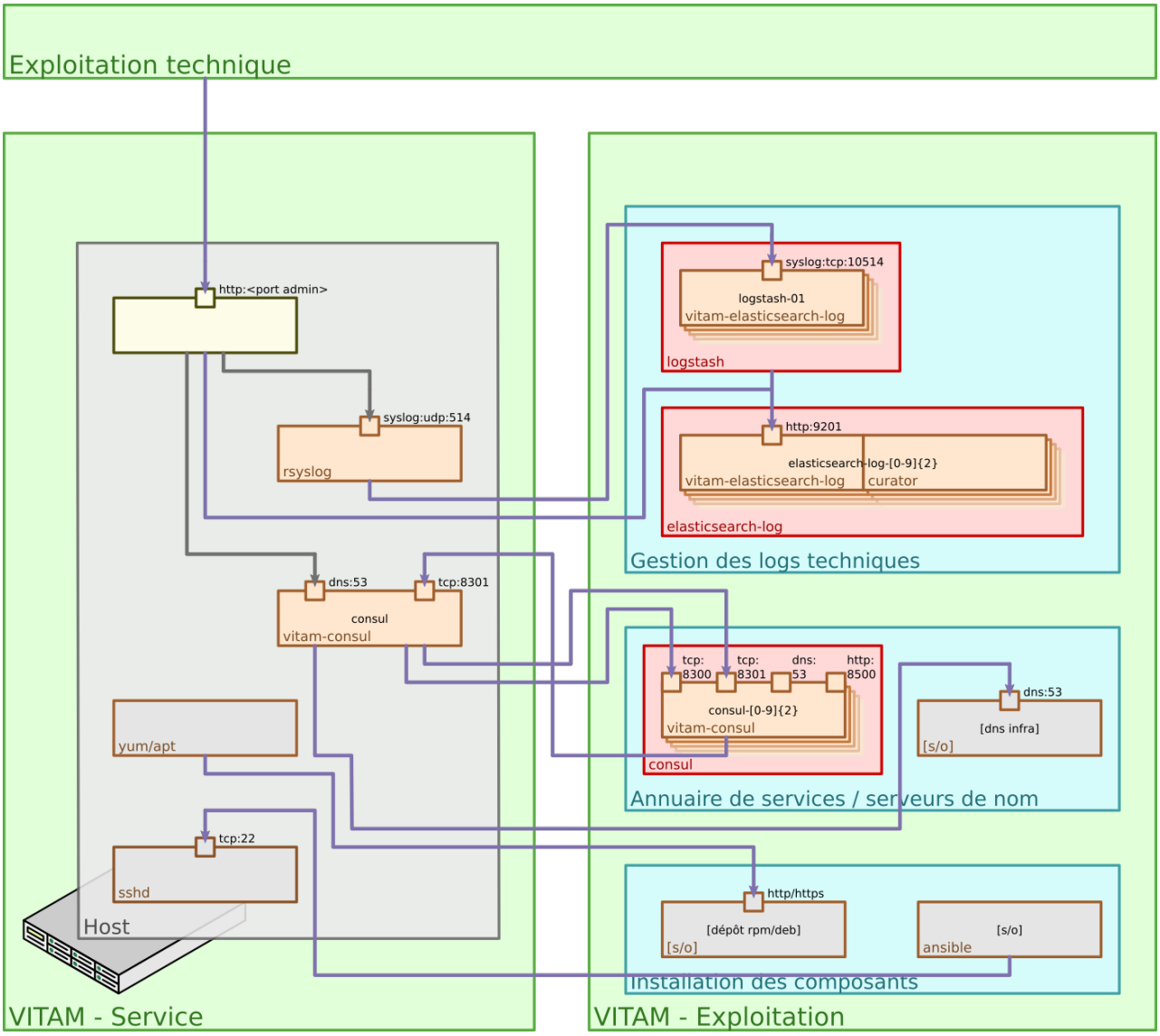


FIG. 8 – Architecture technique : flux (5/5 : flux du socle technique). Seul le port exposant les services d'administration/exploitation est représenté sur le composant *VITAM* présenté dans cette figure.

5.4.4 Découpage en zones

Le schéma ci-dessous reprend les composants applicatifs, en les regroupant par zones.

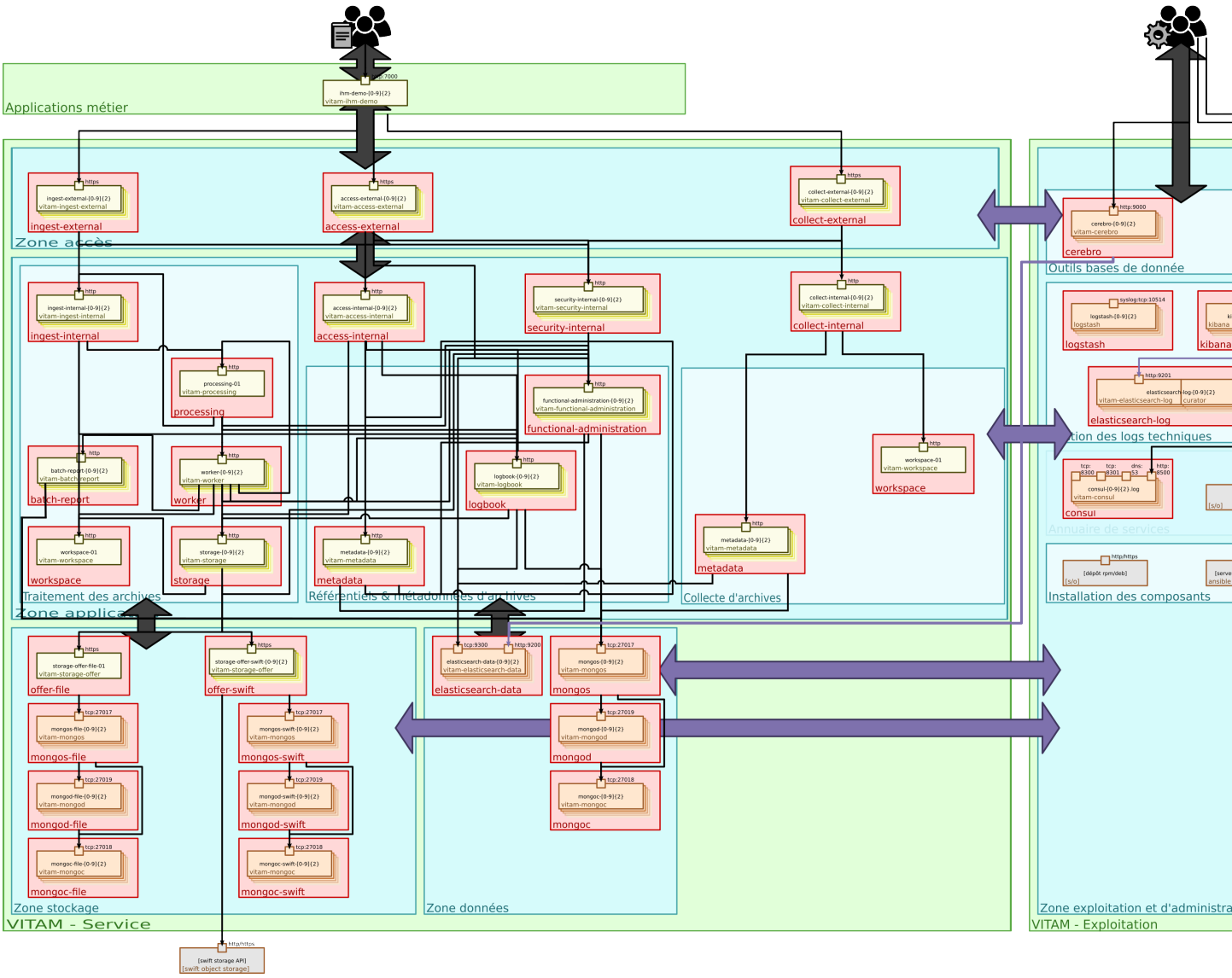


FIG. 9 – Architecture technique : délimitation par zones

5.5 Stockage des données

Voir aussi :

Cette section s'appuie fortement sur *la description de l'architecture des données* (page 17), en particulier en ce qui concerne les données d'archive.

Les offres de stockage *VITAM* portent la référence des données concernant les archives hébergées par le système *VITAM* : leur contenu binaire (*BDO*), mais également les métadonnées associées au sens large (*AU*, *GOT*, journaux).

VITAM possède trois implémentations possibles d'offres de stockage « classiques » : l'implémentation basée sur un système de fichiers, l'implémentation Swift et l'implémentation S3. En complément, une implémentation *Tape-library*, aussi désignée Offre froide, permet l'usage de robotique de cartouches magnétiques.

VITAM peut stocker les données dans plusieurs offres de stockage en parallèle afin de se parer contre la perte de données. Les deux types d'offres peuvent être utilisées seuls ou ensemble sur des offres de stockage différentes : ainsi,

on peut configurer *VITAM* pour déposer les données dans 2 offres de stockage filesystem disjointes, ou dans une offre de stockage filesystem et une offre *Swift*, ou encore dans une offre de stockage filesystem et 2 offres *Swift* différentes (se basant sur 2 *clusters Swift* distincts) ; tout dépend des contraintes de non-perte de données, de scalabilité et de résilience à la panne qui sont abordés dans la description des types d'offres ci-dessous. C'est la configuration des stratégies qui permet de définir les offres sur lesquelles vont être stockées les données.

Note : Un mécanisme de resynchronisation d'une offre de stockage avec une autre, migrant de fait les données entre offres, est disponible en mode complet (la procédure est décrite dans le *DEX*).

Important : Dans le but d'assurer au maximum la pérennité des données conservées dans le système *VITAM*, il est très fortement conseillé de stocker les données dans au moins 2 technologies de stockage différentes (ex : 2 stockages objets de constructeurs et technologies différentes, un stockage objet et un stockage bloc, 2 stockages bloc de constructeurs et technologies différentes, ...)

5.5.1 Stratégies de stockage

VITAM permet de définir plusieurs stratégies de stockage sur une plateforme VITAM.

Plusieurs règles :

- une stratégie de plateforme VITAM est obligatoire
- la stratégie de plateforme VITAM est utilisée par défaut
- la définition d'une stratégie sur une donnée est immutable : on ne peut changer la stratégie d'un objet stocké

5.5.2 Offre filesystem

L'offre filesystem permet de stocker les données sur un système de fichiers accessible localement par le composant `offer`.

Points positifs :

- facile à mettre en place
- facile à exploiter
- facile à sauvegarder

Points négatifs :

- pour une offre de stockage, seule une seule instance du service offer peut être active à un instant donné, ce qui implique que cette offre :
  - n'est pas scalable par multi-instanciation (i.e. horizontalement) ;
  - ne possède pas de solution de haute disponibilité portée par la solution logicielle.

Par conséquent, elle est particulièrement adaptée pour les déploiements de test ou de petite taille (ordre de grandeur : < 10 To), mais est à déconseiller pour les déploiements sur des volumétries importantes.

Prudence : L'offre filesystem nécessite un système de fichiers acceptant les attributs étendus (ex : XFS) ; en particulier, il n'est donc pas possible d'héberger les données sur un montage NFS (NFS ne supportant pas les attributs étendus).

5.5.3 Offre Swift

L'offre Swift permet de stocker les données sur un stockage objet implémentant l'*API Swift*.

Points positifs :

- scalable : offer se comporte dans ce scénario comme une passerelle vers l'*API Swift* ; il est donc multi-instanciable au sein d'une offre de stockage.
- consomme une *API* normalisée : elle est donc compatible avec un grand nombre d'implémentations différentes de *Swift*.

Points négatifs :

- nécessite la mise en place et l'exploitation d'un stockage objet, ce qui est potentiellement plus complexe et moins courant que la mise à disposition d'un simple stockage bloc ou fichier.

Par conséquent, elle est particulièrement adaptée pour les déploiements en production de forte volumétrie.

Note : Dans cette version de la solution logicielle *VITAM*, l'implémentation *Swift* n'est en théorie pas obligée de permettre l'*upload* de fichiers de taille non connue par avance (mode *chunk encoding*) ; cependant, aucun test pertinent n'a pu être effectué faute d'implémentation disponible. Merci de remonter à l'équipe support tout bug associé à ce comportement.

Avertissement : Seule l'*API* d'authentification *keystone* v3 est aujourd'hui officiellement supportées par la solution logicielle *VITAM*. Les *API* d'authentification *keystone* v1 et v2 sont dépréciées.

Note : L'activation de la configuration de headers spécifiques Vitam lors des appels vers le stockage Swift est gérée par la variable enableCustomHeaders (boolean). Si activé, il faut éditer la variable customHeaders avec une liste de clé/valeur pour définir la liste des headers à envoyer au niveau des appels HTTP de l'offre vers le serveur de stockage Swift.

Note : *VITAM* utilise par défaut des connections HTTP persistantes (keep-alive) vers le serveur de stockage Swift si ce dernier les supporte. Si l'infrastructure réseau ou des Reverse-Proxy intermédiaires sont susceptibles de couper occasionnellement les connexions réseau, ceci peut causer des erreurs d'accès au serveur Swift distant. Il peut alors être pertinent de désactiver la persistance des connexions HTTP via le paramètre swiftDisableKeepAlive (peut avoir des impacts sur les performances d'accès à l'offre).

Note : Par tenant *VITAM* utilisé, jusqu'à 17 *containers* sont créés.

La liste des *containers* est :

```
"units"
"objects"
"objectgroups"
"logbooks"
"reports"
"manifests"
"profiles"
"storagelog"
"storageaccesslog"
```

(suite sur la page suivante)

(suite de la page précédente)

```
"storagetraceability"
"rules"
"dip"
"agencies"
"backup"
"backupoperations"
"unitgraph"
"objectgroupgraph"
"distributionreports"
"accessionregistersdetail"
"accessionregisterssymbolic"
"tmp"
"archivaltransferreply"
```

5.5.4 Offre S3

L'offre S3 permet de stocker les données sur un stockage objet implémentant l'*API* S3.

Les points positifs et négatifs sont les mêmes que pour l'offre *Swift*.

L'offre S3 utilise le client java S3 du SDK Amazon V1. De ce fait la compatibilité du stockage avec l'*API* S3 sera limitée à sa compatibilité avec le client logiciel sélectionné. Pour que *VITAM* soit compatible avec l'*API* S3 les noms de conteneurs sont transformés pour obtenir des noms de *bucket* valides :

- remplacement de tous les caractères non alphanumériques par des "."
- suppression des "." au début et à la fin
- passage de tous les caractères en minuscule

Note : Dans le cas où l'implémentation S3 sous-jacente supporte le versioning des buckets ou la suppression logique des fichiers (aussi connue sous les appellations « soft delete » ou « delete markers »), ces fonctionnalités doivent alors être désactivées. En effet, l'activation des ces fonctionnalités cause une démultiplication de l'espace de stockage et la non suppression des données éliminées.

Note : Dans cette version de la solution logicielle *VITAM*, l'implémentation S3 fournie par *VITAM* nécessite la taille du fichier pour l'envoyer dans le stockage S3.

Note : Par tenant VITAM utilisé, jusqu'à 17 *containers* (et donc **buckets**) sont créés.

La liste des *containers* est :

```
"units"
"objects"
"objectgroups"
"logbooks"
"reports"
"manifests"
"profiles"
"storagelog"
"storageaccesslog"
"storagetraceability"
```

(suite sur la page suivante)

(suite de la page précédente)

```
"rules"
"dip"
"agencies"
"backup"
"backupoperations"
"unitgraph"
"objectgroupgraph"
"distributionreports"
"accessionregistersdetail"
"accessionregisterssymbolic"
"tmp"
"archivaltransferreply"
```

Avertissement : Par défaut, le fournisseur Amazon définit une limite à 100 *buckets* ; il convient de revoir ce paramétrage dans le cas où la solution logicielle *VITAM* doit gérer plus de **5** *tenants*.

La liste des API S3 utilisées par *VITAM* est :

Buckets

- API : GET Bucket acl

Exemple :

```
[REQUEST (objectAPIHandlers).GetBucketACLHandler-fm] [160087199.728184] [2020-09-23␣
↪14:39:57 +0000]
GET /5.dbxuwfvoes/?acl
Host: 127.0.0.1:9999
X-Amz-Content-Sha256: e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
X-Amz-Date: 20200923T143957Z
Content-Length: 0
Connection: Keep-Alive
Authorization: AWS4-HMAC-SHA256 Credential=MKU4HW1K9HSST78MDY3T/20200923//s3/aws4_
↪request, SignedHeaders=amz-sdk-invocation-id;amz-sdk-retry;content-type;host;user-
↪agent;x-amz-content-sha256;x-amz-date,␣
↪Signature=640458e964fd2caa14b0ffb4808ee99fdf5d232808a4a8e397c121ea77b00a74
User-Agent: aws-sdk-java/1.11.720 Linux/4.15.0-117-generic OpenJDK_64-Bit_Server_VM/
↪11.0.8+10-post-Ubuntu-0ubuntu118.04.1 java/11.0.8 vendor/Ubuntu
Amz-Sdk-Retry: 0/0/500
Content-Type: application/octet-stream
Amz-Sdk-Invocation-Id: 8db9bb54-6dcf-c5d8-579a-c3dbd293f165
---
```

- API : PUT Bucket

Exemple :

```
[REQUEST (objectAPIHandlers).PutBucketHandler-fm] [160087200.043971] [2020-09-23␣
↪14:40:00 +0000]
PUT /5.dbxuwfvoes/
Host: 127.0.0.1:9999
Amz-Sdk-Invocation-Id: 9d77b44d-8d52-7065-1872-ac9975a63425
Amz-Sdk-Retry: 0/0/500
Authorization: AWS4-HMAC-SHA256 Credential=MKU4HW1K9HSST78MDY3T/20200923//s3/aws4_
↪request, SignedHeaders=amz-sdk-invocation-id;amz-sdk-retry;content-type;host;user-
↪agent;x-amz-content-sha256;x-amz-date,␣
↪Signature=e03a2feaaff4538887a06ca53a7c4c9609effc6dbd34d9191ec461f3b5e43918
```

(suite sur la page suivante)

(suite de la page précédente)

```
Connection: Keep-Alive
Content-Type: application/octet-stream
User-Agent: aws-sdk-java/1.11.720 Linux/4.15.0-117-generic OpenJDK_64-Bit_Server_VM/
↪11.0.8+10-post-Ubuntu-0ubuntu118.04.1 java/11.0.8 vendor/Ubuntu
X-Amz-Content-Sha256: e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
X-Amz-Date: 20200923T144000Z
Content-Length: 0
---
```

- API : GET Bucket (List Objects)Version 2

Exemple 1 :

```
[REQUEST (objectAPIHandlers).ListObjectsV2Handler-fm] [160087204.792921] [2020-09-23␣
↪14:40:47 +0000]
GET /7.hvybpdxfsm/?list-type=2&max-keys=1000&fetch-owner=false
Host: 127.0.0.1:9999
Content-Type: application/octet-stream
Amz-Sdk-Invocation-Id: df03b558-11b5-1ce6-a69c-f1086718ee0b
Amz-Sdk-Retry: 0/0/500
User-Agent: aws-sdk-java/1.11.720 Linux/4.15.0-117-generic OpenJDK_64-Bit_Server_VM/
↪11.0.8+10-post-Ubuntu-0ubuntu118.04.1 java/11.0.8 vendor/Ubuntu
X-Amz-Content-Sha256: e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
X-Amz-Date: 20200923T144047Z
Content-Length: 0
Connection: Keep-Alive
Authorization: AWS4-HMAC-SHA256 Credential=MKU4HW1K9HSST78MDY3T/20200923//s3/aws4_
↪request, SignedHeaders=amz-sdk-invocation-id;amz-sdk-retry;content-type;host;user-
↪agent;x-amz-content-sha256;x-amz-date,␣
↪Signature=3331501246aee47854132b6d9deec6dce68468e6c70363d2901434f7753673c3
```

Exemple 2 :

```
[REQUEST (objectAPIHandlers).ListObjectsV2Handler-fm] [160087204.796455] [2020-09-23␣
↪14:40:47 +0000]
GET /7.hvybpdxfsm/?list-type=2&continuation-
↪token=aeaaaaaaaabe6it7abvtgaluxnunfjqaaaaq188&max-keys=1000&fetch-owner=false
Host: 127.0.0.1:9999
Amz-Sdk-Retry: 0/0/500
Authorization: AWS4-HMAC-SHA256 Credential=MKU4HW1K9HSST78MDY3T/20200923//s3/aws4_
↪request, SignedHeaders=amz-sdk-invocation-id;amz-sdk-retry;content-type;host;user-
↪agent;x-amz-content-sha256;x-amz-date,␣
↪Signature=c482925c51dcaa4fa8ab65ade329fc1e968726edb2c2fe0313345aeced2ea1e6
Content-Type: application/octet-stream
User-Agent: aws-sdk-java/1.11.720 Linux/4.15.0-117-generic OpenJDK_64-Bit_Server_VM/
↪11.0.8+10-post-Ubuntu-0ubuntu118.04.1 java/11.0.8 vendor/Ubuntu
Content-Length: 0
Amz-Sdk-Invocation-Id: 8f314f3e-fde9-b341-6bc1-8c0b3a830ef3
X-Amz-Content-Sha256: e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
X-Amz-Date: 20200923T144047Z
Connection: Keep-Alive
```

Objects

- API : HEAD Object

Exemple :

```
[REQUEST (objectAPIHandlers).HeadObjectHandler-fm] [160087199.737232] [2020-09-23␣
↪14:39:57 +0000]
HEAD /5.dbxuwfvoes/aeaaaaaaaabe6it7abu44aluxnubyjiaaaaq
Host: 127.0.0.1:9999
Content-Type: application/octet-stream
User-Agent: aws-sdk-java/1.11.720 Linux/4.15.0-117-generic OpenJDK_64-Bit_Server_VM/
↪11.0.8+10-post-Ubuntu-0ubuntu118.04.1 java/11.0.8 vendor/Ubuntu
X-Amz-Date: 20200923T143957Z
Connection: Keep-Alive
X-Amz-Content-Sha256: e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
Amz-Sdk-Invocation-Id: 3b849433-402b-b1fe-da13-d6d7ed267ff5
Amz-Sdk-Retry: 0/0/500
Authorization: AWS4-HMAC-SHA256 Credential=MKU4HW1K9HSST78MDY3T/20200923//s3/aws4_
↪request, SignedHeaders=amz-sdk-invocation-id;amz-sdk-retry;content-type;host;user-
↪agent;x-amz-content-sha256;x-amz-date,␣
↪Signature=4a465630ac0523e27bb3dfaa6a573ee3a3516cc884ff037b11f947b309095041
---
```

- API : Delete Object

Exemple :

```
[REQUEST (objectAPIHandlers).DeleteObjectHandler-fm] [160087199.737773] [2020-09-23␣
↪14:39:57 +0000]
DELETE /5.dbxuwfvoes/aeaaaaaaaabe6it7abu44aluxnubyjiaaaaq
Host: 127.0.0.1:9999
Amz-Sdk-Invocation-Id: 7920dbea-9ac8-8928-8f48-b0a587738f24
X-Amz-Content-Sha256: e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
X-Amz-Date: 20200923T143957Z
Connection: Keep-Alive
Amz-Sdk-Retry: 0/0/500
Authorization: AWS4-HMAC-SHA256 Credential=MKU4HW1K9HSST78MDY3T/20200923//s3/aws4_
↪request, SignedHeaders=amz-sdk-invocation-id;amz-sdk-retry;content-type;host;user-
↪agent;x-amz-content-sha256;x-amz-date,␣
↪Signature=1b5243ccfc2f43518d70a0a71fdc96971b8b16ab4ff2721648ad38b77ea48048
Content-Type: application/octet-stream
User-Agent: aws-sdk-java/1.11.720 Linux/4.15.0-117-generic OpenJDK_64-Bit_Server_VM/
↪11.0.8+10-post-Ubuntu-0ubuntu118.04.1 java/11.0.8 vendor/Ubuntu
---
```

- API : GET Object

Exemple :

```
[REQUEST (objectAPIHandlers).GetObjectHandler-fm] [160087200.042941] [2020-09-23␣
↪14:40:00 +0000]
GET /5.dbxuwfvoes/aeaaaaaaaabe6it7abu44aluxnubyjiaaaaq
Host: 127.0.0.1:9999
X-Amz-Date: 20200923T144000Z
Content-Length: 0
Amz-Sdk-Invocation-Id: e146fb12-0283-af0d-83ef-fbcf01ad589d
Amz-Sdk-Retry: 0/0/500
Authorization: AWS4-HMAC-SHA256 Credential=MKU4HW1K9HSST78MDY3T/20200923//s3/aws4_
↪request, SignedHeaders=amz-sdk-invocation-id;amz-sdk-retry;content-type;host;user-
↪agent;x-amz-content-sha256;x-amz-date,␣
↪Signature=687b0e4098bced5f3fb99bf9a7bb1a994532f65bb66a101889892fa1a55ed925
Content-Type: application/octet-stream
```

(suite sur la page suivante)

(suite de la page précédente)

```
User-Agent: aws-sdk-java/1.11.720 Linux/4.15.0-117-generic OpenJDK_64-Bit_Server_VM/
↪11.0.8+10-post-Ubuntu-0ubuntu118.04.1 java/11.0.8 vendor/Ubuntu
X-Amz-Content-Sha256: e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
Connection: Keep-Alive

<BODY>
---
```

- API : PUT Object

Exemple :

```
[REQUEST (objectAPIHandlers).PutObjectHandler-fm] [160087199.739414] [2020-09-23␣
↪14:39:57 +0000]
PUT /5.dbxuwfvoes/aeaaaaaaaabe6it7abu44aluxnubyjiaaaaq
Host: 127.0.0.1:9999
Amz-Sdk-Retry: 0/0/500
X-Amz-Content-Sha256: STREAMING-AWS4-HMAC-SHA256-PAYLOAD
X-Amz-Date: 20200923T143957Z
Content-Length: 7081
Connection: Keep-Alive
Expect: 100-continue
Amz-Sdk-Invocation-Id: f7ce7617-0563-19ea-1187-33811dc527ac
Authorization: AWS4-HMAC-SHA256 Credential=MKU4HW1K9HSST78MDY3T/20200923//s3/aws4_
↪request, SignedHeaders=amz-sdk-invocation-id;amz-sdk-retry;content-length;content-
↪type;host;user-agent;x-amz-content-sha256;x-amz-date;x-amz-decoded-content-length,␣
↪Signature=02c10a4598ae8804e416c69249c6e214a100e145b9ec02f3ff3925a73f6b085f
Content-Type: application/octet-stream
User-Agent: aws-sdk-java/1.11.720 Linux/4.15.0-117-generic OpenJDK_64-Bit_Server_VM/
↪11.0.8+10-post-Ubuntu-0ubuntu118.04.1 java/11.0.8 vendor/Ubuntu
X-Amz-Decoded-Content-Length: 6906

<BODY>
---
```

- API : PUT Object - Copy

Exemple :

```
[REQUEST (objectAPIHandlers).CopyObjectHandler-fm] [160087200.056875] [2020-09-23␣
↪14:40:00 +0000]
PUT /5.dbxuwfvoes/aeaaaaaaaabe6it7abu44aluxnubyjiaaaaq
Host: 127.0.0.1:9999
X-Amz-Meta-Digest:␣
↪9ba9ef903b46798c83d46bcbd42805eb69ad1b6a8b72e929f87d72f5263a05ade47d8e2f860aece8b9e3acb948364fedf75
X-Amz-Date: 20200923T144000Z
Content-Length: 0
Content-Type: application/octet-stream
Amz-Sdk-Invocation-Id: a3de1f75-d08d-b7d6-ad23-7c16e23c4e4f
User-Agent: aws-sdk-java/1.11.720 Linux/4.15.0-117-generic OpenJDK_64-Bit_Server_VM/
↪11.0.8+10-post-Ubuntu-0ubuntu118.04.1 java/11.0.8 vendor/Ubuntu
X-Amz-Content-Sha256: e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
X-Amz-Copy-Source: /5.dbxuwfvoes/aeaaaaaaaabe6it7abu44aluxnubyjiaaaaq
X-Amz-Meta-Digest-Type: SHA-512
X-Amz-Metadata-Directive: REPLACE
Connection: Keep-Alive
```

(suite sur la page suivante)

(suite de la page précédente)

```
Amz-Sdk-Retry: 0/0/500
Authorization: AWS4-HMAC-SHA256 Credential=MKU4HW1K9HSST78MDY3T/20200923//s3/aws4_
↪request, SignedHeaders=amz-sdk-invocation-id;amz-sdk-retry;content-length;content-
↪type;host;user-agent;x-amz-content-sha256;x-amz-copy-source;x-amz-date;x-amz-meta-
↪digest;x-amz-meta-digest-type;x-amz-metadata-directive,␣
↪Signature=6a366171c7f83a9fed12402e9984aaab8148bb7f438cbf43465ee7ed8ae7b7dc
---
```

5.5.5 Offre Tape-library

L'offre *Tape-library*, aussi désignée **Offre Froide**, permet de stocker les données sur des librairies de cartouches magnétiques.

Elle s'appuie sur des commandes linux standard pour manipuler les éléments robotiques. Elle est donc à priori compatible avec tous les matériels compatibles Linux.

Points positifs :

- Froide : à contrario des offres disques déjà utilisables dans *VITAM* (FS ou Objet), l'accès aux données sur les cartouches n'est pas immédiat. Il nécessite le montage des cartouches dans des lecteurs, qui sont en nombre limités. En cas de corruption des données des offres disques (par exemple, attaque de type *ransomware*), sa répercussion vers les données archivées sur cartouches serait très lente. C'est une garantie de sécurité supplémentaire.
- Peu onéreuse pour le stockage des grands volumes : comparée à un stockage disque, un stockage bande est moins onéreux : 1 To de stockage sur LTO revient à 10€ HT.
- Durabilité des bandes : 15-30 ans contre 5 ans pour les HDD/SSD, avec 20000 cycles de chargements/déchargements.
- Possibilité de blocage physique des réécritures (WORM), de compression et de chiffrement natifs des données.
- Externalisable : les cartouches peuvent être extraites de la librairie une fois les données inscrites, et stockées dans un local sécurisé tiers.

Note : Dans sa version actuelle, *VITAM* ne prend pas en charge les opérations d'externalisation. Ce process ne peut être réalisé que manuellement.

Points négatifs :

- N'est pas multi-instantiable : Pour une offre de stockage, seule une seule instance du service offer peut être active à un instant donné (i.e. ne peut être déployée en haute disponibilité)
- Nécessite une infrastructure lourde (hardware, hébergement…), coûteuse à installer et compliquée à exploiter.
- Nécessite un espace de stockage disque local `/vitam/data/offer` conséquent (espace « tampon » pour écriture rapide + espace « cache » pour relecture des données).

L'offre Tape-library utilise les commandes standard `mt`, `mtx` et `dd` pour manipuler les lecteurs de bandes et la librairie. Ces outils sont installés sur le serveur où est déployée l'offre froide. Cette même machine doit également avoir accès à la librairie soit par attachement direct, soit par le biais d'un accès distant (ex : iscsi)

Note : L'usage des commandes `mt` et `mtx` nécessite des droits spécifiques. L'utilisateur vitam est automatiquement ajouté au groupe unix « tape » sur le serveur où est déployée l'offre froide.

Note : Le dossier `/vitam/data/offer/` doit correspondre à une seule partition de système de fichiers (i.e. tout le contenu du dossier `/vitam/data/offer` doit appartenir au même point de montage). Le système de fichiers

doit supporter les opérations de atomiques (type atomic rename / move) et la création de liens symboliques (ex. XFS, EXT4…)

Voir aussi :

Les principes de fonctionnement de l'offre froide sont décrits dans la documentation externe dédiée (« Archivage sur Offre Froide »).

5.6 Concentration et exploitation des logs applicatifs

5.6.1 Besoins

Contrairement aux journaux applicatifs, les logs techniques générés par les applications ne participent pas à la valeur probante et à la preuve systémique du *SAE*. Il n'y a donc pas de besoin métier sur la non perte de logs. Cependant, étant donné la présence notable des alertes de sécurité, un effort est fait pour réduire au maximum les risques de perte de logs.

5.6.2 Modèle générique

On peut noter les composants suivants :

- Émetteur du log : il s'agit de l'application qui est à l'origine du log
- Agent de transport du log : il s'agit d'un composant recevant tous les logs associés à un serveur/*VM* (mais pas container)
- Concentrateur du log : il s'agit de la cible de réception du log .
- Stockage des logs : il s'agit du composant stockant les logs (de manière plus ou moins requêtable)
- Visualisation des logs : il s'agit du composant (souvent *IHM*) qui permet la recherche et la visualisation des logs

Les échanges doivent se faire selon des protocoles données :

- Protocole d'émission du log (entre émetteur et agent de transport)
- Protocole de transport du log (entre agent de transport et concentrateur)

L'architecture générique peut être vue de la manière suivante :

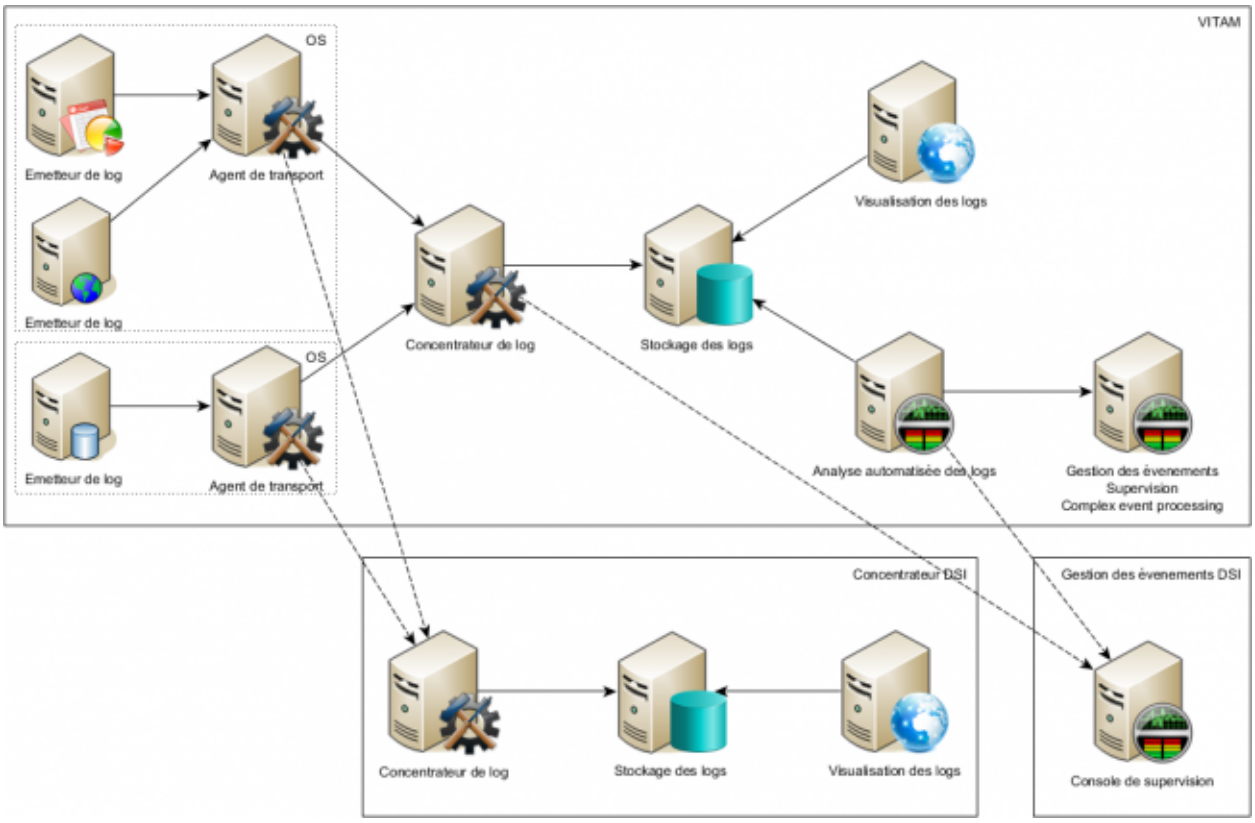


FIG. 10 – Architecture générique d'un système de gestion de logs.

VITAM n'implémente qu'une sous partie de cette architecture générique (la centralisation / stockage / visualisation), mais permet l'intégration d'un composant externe de gestion de logs.

5.6.3 Choix des implémentations

De manière générale, l'implémentation s'appuie fortement sur une architecture `syslog`.

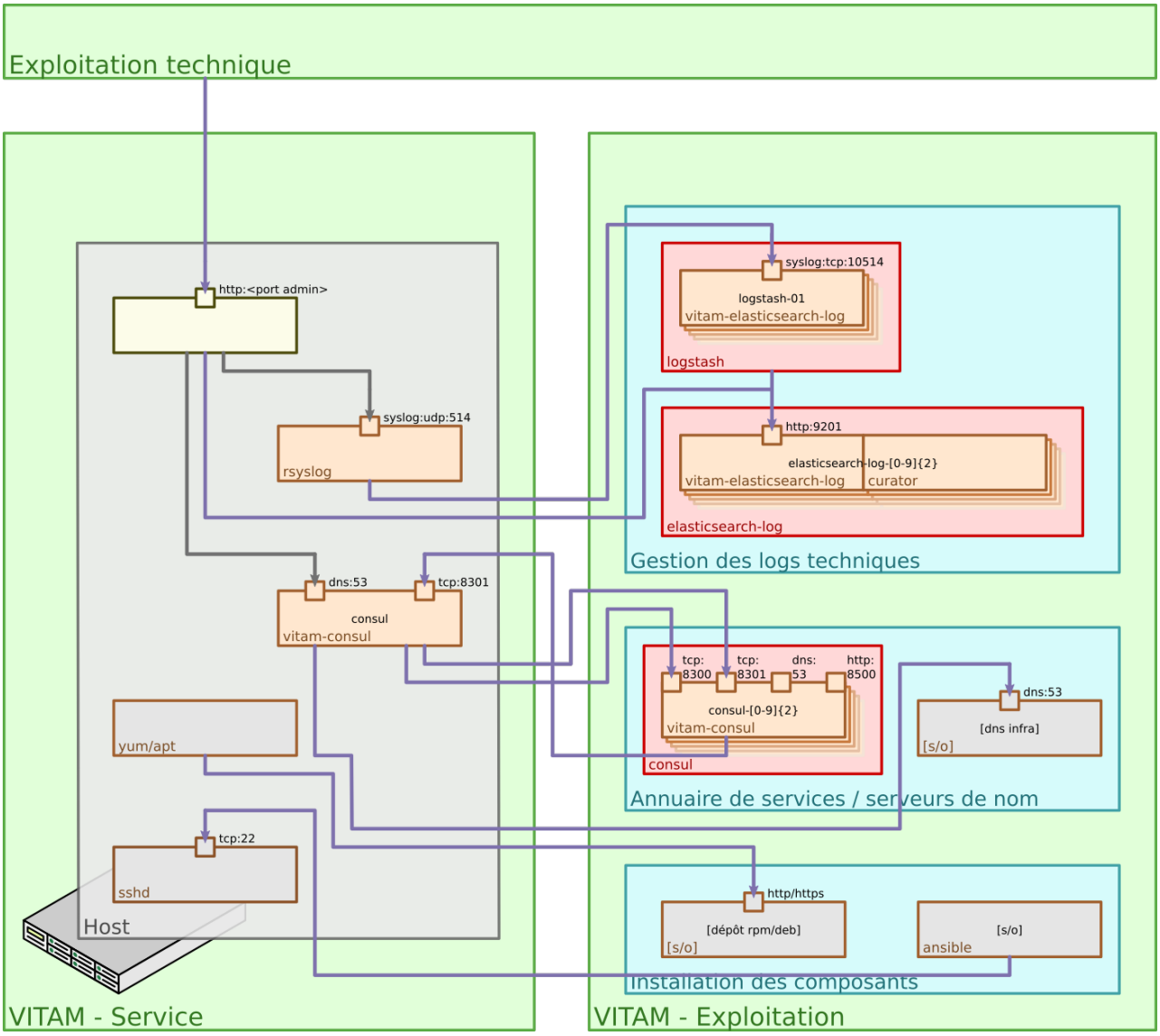


FIG. 11 – Architecture du sous-système de centralisation des logs

Cette implémentation vise à éviter au maximum les pertes de logs ; cela implique notamment l'utilisation de *buffers* stockant temporairement les logs en cas de déconnexion réseau, et l'utilisation de protocole non reliables (ex : *UDP*) uniquement sur des liens réseaux locaux à une instance (ex : boucle locale).

5.6.3.1 Émetteur de logs

Dans le système *VITAM*, l'émetteur des logs peut être :

- Pour les composants logiciels Java *VITAM* : l'appender logback SyslogAppender [31] ;
- Pour les script unix : la commande `logger`.

Un émetteur de logs a les responsabilités suivantes :

- Le formatage du message selon le format de log préconisé pour l'application ;

31. http://logback.qos.ch/manual/appenders.html#SyslogAppender

- L'envoi des logs à l'agent de transport de logs selon le protocole défini dans *la section présentant les principes de log* (page 42).

5.6.3.2 Agent de transport de log

L'agent de transport de log est `rsyslog`. Il est installé localement sur chaque serveur hébergeant des composants logiciels du système *VITAM*.

Il a les responsabilités suivantes :

- L'acquisition des logs au format syslog *UDP* (sur le port par défaut 514) et syslog unix (`/dev/log`) ;
- Le buffering des logs (utilisation d'une action queue rsyslog de type « Disk-Assisted Memory Queue »[32]) ;
- La transmission des logs au concentrateur.

Note : Rationale : il s'agit de l'agent syslog par défaut sur les distributions supportées par *VITAM*, et il présente une consommation mémoire limitée (notamment par rapport à d'autres solutions en Java ou Ruby).

Le protocole de transport du log (entre agent de transport et concentrateur) doit être conforme au format syslog tcp (RFC 3195, basé sur la RFC 3164).

Note : Ce format est privilégié car il est un bon compromis entre fiabilité (sécurité d'acheminement de TCP) et exploitabilité . Il n'y a en effet pas de contraintes imposant des protocoles plus "reliable" comme RLTP ou RELP.

En se basant sur la RFC 5424, les paramètres imposés sur les messages syslog sont identiques aux paramètres décrits dans *la section présentant les principes de log* (page 42).

5.6.3.3 Concentration de logs

Le concentrateur de logs est `logstash`. Il est instancié de manière unique ou en *cluster*, et a les responsabilités suivantes :

- Acquisition des logs au format syslog TCP (RFC 3164) ;
- Parsing des logs pour en extraire la structure ;
- Dépôt des logs dans le stockage de logs.

5.6.3.4 Stockage des logs

Le stockage des logs se fait dans le moteur d'indexation ElasticSearch, dans un cluster dédié au stockage des logs (pour séparer les données de logs et les données métier d'archives).

La configuration de ce cluster dépend de la taille du déploiement *VITAM* envisagé. *Des dimensionnements indicatifs sont disponibles dans une section dédiée* (page 92). Le paramétrage par défaut des shards et replicas est le suivant :

- Nombre nominal de shards primaires par index : 4 ;
- Nombre nominal de replicas : 1 ;

Note : Les abaques proposées correspondent à un compromis en terme d'usage des ressources VS résilience du système. Ces paramètres peuvent être changés si un besoin plus fort de résilience était identifié. Dans ce cas, on peut augmenter le nombre de noeuds ainsi que le nombre de replicas, en veillant à ce que le nombre de shards primaires ne

32. http://www.rsyslog.com/doc/v8-stable/concepts/queues.html

soit jamais inférieur au nombre de noeuds du cluster, et que le nombre de replicas ne soit jamais supérieur au nombre de noeuds du cluster - 1.

Prudence : Une modification du nombre de shards primaires d'un index est une opération coûteuse à réaliser sur un cluster en cours de fonctionnement et qui doit dans la mesure du possible être évitée (indisponibilité du cluster et/ou risque de corruption et de perte de données en cas de problème au cours de l'opération) ; le bon dimensionnement de cette valeur doit être réalisé dès l'installation du cluster.

- Index : chaque index stockant des données de logs correspond à 1 jour de logs (déterminé à partir du timestamp du log). Les index définis sont les suivants :
  - `logstash-vitam-YYYY.MM.dd` pour les messages concernant les composants de la solution *VITAM*, avec un type de données par format de logs, i.e. :
    - type `logback` pour les logs issus des applications Java ;
    - type `scripts` pour logs issus des scripts ;
    - type `mongo` pour les logs de mongodb ;
    - type `elastic` pour les logs d'elasticsearch (cluster métier).
  - `logstash-logs-YYYY.MM.dd` pour les logs issus du sous-système de logs, avec un type de données par format de logs, i.e. :
    - type `elastic` pour les logs d'elasticsearch (cluster de logs) ;
    - type `logstash` pour les logs de logstash (`WARN` ou plus) ;
    - type `kibana` pour les logs issus de Kibana.
    - type `curator` pour les logs issus de Curator.
  - `logstash-failure-YYYY.MM.dd` (1 par jour ; le jour correspond au jour de l'horodatage des messages), pour les messages correspondant à un échec de parsing.
  - `.kibana` pour le stockage des paramètres (et notamment des dashboards) Kibana.

Prudence : Dans le cadre de cette version de la solution *VITAM*, cette réflexion n'intègre pas la problématique des traces associées aux actions utilisateur (par exemple : accès au système, lancement d'une opération sur les archives, consultations d'archives, échec d'authentification, refus d'accès, ...) ; cette problématique est encore en cours d'étude, notamment pour en définir les besoins en terme de criticité (et notamment la non-perte d'information, leur degré de confidentialité et d'intégrité), et sera potentiellement prise en compte par un autre sous-système.

5.6.3.4.1 Gestion des index

La création des templates d'index et des index doit être réalisée par l'application à l'origine de l'écriture dans Elasticsearch (kibana pour l'index `.kibana`, logstash pour les autres index). La gestion des index est réalisée par l'application Curator [33]. Le paramétrage est réalisable par l'exploitant (cf. *DIN*). Les valeurs suivantes sont recommandées :

- Durée de maintien des index « online » : 30 jours ; cela signifie qu'au bout de 30 jours, les index seront fermés, et n'apparaîtront donc plus dans l'*IHM* de suivi des logs. Cependant, ils sont conservés, et pourront donc être réouverts en cas de besoin.
- Durée de conservation des index : 365 jours ; au bout de cette durée, les index seront supprimés.

33. https://www.elastic.co/guide/en/elasticsearch/client/curator/4.0/index.html

5.6.3.5 Visualisation des logs

La visualisation des logs se fait par le composant Kibana. Il est instancié de manière unique et persiste sa configuration dans ElasticSearch (dans l'index `.kibana`).

Aucun mécanisme d'authentification n'est mis en place pour sécuriser l'accès à Kibana.

Indication : La version opensource de Kibana, utilisée dans *VITAM*, ne supporte pas nativement l'authentification des clients ; d'autres solutions peuvent être mises en place (ex : l'utilisation du composant Security [34]), sous réserve d'une étude de compatibilité de la solution choisie.

5.6.4 Intégration à un système de gestion de logs existants

L'intégration à un autre système de logs (pour y dupliquer les logs) est possible ; deux points d'ancrage sont envisageables :

- au niveau de logback ; ce point d'extension ne permet que d'obtenir les logs en provenance des applicatifs métier (java) ; ce point d'extension est par conséquent déconseillé ;
- au niveau de rsyslog ; ce point d'extension permet d'agir sur les logs provenant de tous les composants déployés (y compris les bases de données et d'autres composants d'infrastructure déployés dans le cadre de *VITAM*). C'est le point d'extension conseillé en cas d'intégration avec un système de gestion de logs externe.

Astuce : Les règles de grok fournies avec le composant logstash (disponibles dans le répertoire de configuration de composant) sont un bon point de départ pour intégrer le format des différents logs dans un système de gestion de logs tiers.

5.6.5 Limites

La solution implémentée dans *VITAM* possède les limites connues suivantes :

- Cette solution réutilise les principes de centralisation de logs basés sur les systèmes syslog ; par conséquent, elle en hérite certaines de leurs limites, et notamment l'absence de sécurité dans les protocoles syslog (udp ou tcp) (absence d'authentification, de vérification d'intégrité ou de confidentialité des informations).
- Aucune brique d'alerting n'est intégrée dans cette version de la solution logicielle *VITAM*.

Astuce : Il est à noter que les logs ne sont pas complètement perdus en cas de perte du système de centralisation des logs ; en effet, ils sont dans tous les cas déposés dans des fichiers locaux aux noeuds.

5.7 Métriques applicatives

5.7.1 Besoins

À des fins de monitoring des composants logiciels Java *VITAM* et de l'utilisation des ressources système par ceux-ci, *VITAM* intègre un reporting et une gestion de métriques applicatives.

34. https://www.elastic.co/products/x-pack/security

5.7.2 Modèle générique

On peut noter les composants suivants :

- Enregistreur de métriques : il s'agit de la librairie en charge de l'enregistrement d'une métrique.
- Reporters de métriques : il s'agit de librairies en charge de collecter les métriques enregistrées et d'en faire un reporting.
- Endpoint des métriques : Il s'agit d'une API depuis laquelle des outils de monitoring peuvent récupérer des métriques en mode *pull* au format prometheus
- Stockage des métriques : il s'agit du composant stockant les métriques (de manière plus ou moins requêtable).
- Visualisation des métriques : il s'agit du composant (souvent :term :*IHM*) qui permet la recherche et la visualisation des métriques.

5.7.3 Choix des implémentations

5.7.3.1 Enregistreur de métriques

Dans le système *VITAM*, l'enregistrement de métriques s'effectue uniquement dans les composants logiciels Java à l'aide du librairie Prometheus metrics [35].

Les plugins suivants sont utilisés pour leur métriques respectives :

- Prometheus simpleclient_common et simpleclient [36] pour développer de nouvelles métriques.
- Prometheus simpleclient_hotspot [37] pour les métriques JVM.
- Prometheus simpleclient_dropwizard [38] pour envelopper les métriques Dropwizard.

L'enregistreur de métriques possède un registre interne qui peut stocker différentes métriques :

- Prometheus : **Gauges**, **Counter**, **Summary** ou **Histograms**. Ces métriques seront exposée via une API.

A la différence des Counter dans Dropwizard, ceux de prometheus ne se décrémente pas, il faut privilégier une Gauge dans ce das de figure

Les métriques RESTEasy sont automatiquement générées par l'application *VITAM*. Elles représentent un jeu de 3 métriques, **Meter**, **Timer** et **ExceptionMeter** pour chaque *end-point* des ressources de l'application.

Les métriques JVM sont aussi uniques par application. Elles représentent plusieurs types de métriques sur la consommation de ressources système.

Note : Une description fonctionnelle des métriques est disponible dans le manuel utilisateur dropwizard metrics [39]. Veuillez vous référer au document d'architecture de chaque composant *VITAM* qui doit documenter ses propres métriques, si des métriques spécifiques sont ajoutées.

5.7.3.2 Reporters de métriques

Dans le système *VITAM*, un ou plusieurs reporters de métriques peuvent être utilisés. A ce jour, il existe deux reporters différents :

- Un reporter logback ;

35. https://prometheus.io/docs/instrumenting/clientlibs/
36. https://github.com/prometheus/client_java/tree/master/simpleclient
37. https://github.com/prometheus/client_java/tree/master/simpleclient_hotspot
38. https://github.com/prometheus/client_java/tree/master/simpleclient_dropwizard
39. https://metrics.dropwizard.io/4.1.2/manual/core.html

- Un reporter ElasticSearch issue de la librairie metrics elasticsearch reporter[40].

Les reporters sont utilisés dans les composants logiciels Java. Ils sont en charge de récupérer les valeurs de toutes les métriques enregistrées et de les transmettre sur différents canaux, ici soit un logger logback ou une base de données ElasticSearch.

5.7.3.3 Endpoint des métriques

À partir de la release R14, le système *VITAM* permet d'exposer des métriques au format prometheus. Ces métriques sont exposées via une API dédiée. Un serveur prometheus, ou une autre solution de monitoring compatible avec le format prometheus, peut récupérer les métriques en mode *pull* depuis cette API. Un avantage du mode "pull" est de donner l'information sur l'état de la disponibilité du service en question. Ce mode est aussi plus économe en ressource qu'un mode "push" depuis le service lui-même vers l'outil de monitoring.

5.7.3.4 Stockage des métriques

Si un reporter de métriques ElasticSearch est utilisé, celles-ci seront stockées dans le moteur d'indexation ElasticSearch, dans un cluster dédié au stockage des logs/métriques (pour séparer les données de logs/métriques et les données métier d'archives). La description de ce cluster commun logs/métriques, incluant la gestion des index et la visualisation, se trouve *dans la section précédente* (page 65).

- Index : chaque index stockant des données de métriques correspond à 1 jour de métriques (déterminé à partir du timestamp de la métrique). Les index définis sont les suivants :
  - `metrics-vitam-rest-YYYY.MM.dd` pour les métriques de RESTEasy, avec un champ *name* automatiquement généré sous la forme :

    uri :http_method :consumed_types :produced_types :metric_type Exemple : **_offer_v1__bulk_objects__type_ :PUT :application_octet_stream :application_json :meter_total _offer_v1__bulk_objects__type_ :PUT :application_octet_stream :application_json :timer**
  - `metrics-vitam-jvm-YYYY.MM.dd` pour les métriques JVM.
  - `metrics-vitam-business-YYYY.MM.dd` pour les métriques métier.
  - `.kibana` pour le stockage des paramètres (et notamment des dashboards) Kibana.

À partir de la release R14 de la solution *Vitam* expose ses métriques au format prometheus. Il est possible de configurer un serveur prometheus pour récupérer ces métriques et un Grafana pour les visualiser. Ces deux outils sont largement utilisés, à ce jour, dans la communauté open source.

Note : Veuillez vous référer à la documentation d'exploitation pour savoir comment fonctionne l'intégration et la configuration du serveur prometheus dans *Vitam*

5.7.4 Limites

La solution implémentée dans *Vitam* possède les limites connues suivantes :

- Du fait que la librairie Dropwizard Metrics fait une agrégation des métriques et que le système de visualisation Kibana fonctionne lui aussi à l'aide d'agrégations, les résultats visualisés sont corrects dans la limite d'une certaine précision (certaines données deviennent non-représentatives de la réalité).

40. https://github.com/ProgrammeVitam/elasticsearch-metrics-reporter-java

5.8 Outillage de déploiement

5.8.1 Outil

L'outil de déploiement utilisé sur *VITAM* est ansible. Cette solution de déploiement a les caractéristiques suivantes :

- Agent-less : la propagation des ordres de déploiement utilise SSH et nécessite sur les serveurs un interpréteur Python 2.6+. (Cf. la documentation officielle [41] pour la liste exhaustive des dépendances requises).
- Centralisation des actions : l'intégralité des actions d'administration technique et d'exploitation de la plate-forme est réalisée par cet outil de déploiement (sauf exception mentionnée le cas échéant dans le *DEX*).
- Méthode d'authentification : l'authentification est faite par un utilisateur habilité à se connecter à SSH et devant pouvoir avoir les élévations de privilèges nécessaires pour faire les actions (via su ou sudo) :
  - Le choix de la méthode d'authentification (mot de passe, clé publique sans passphrase ou clé publique avec passphrase) peut être choisi en fonction des contraintes d'hébergement. Cependant, certaines méthodes limiteront l'automatisation du déploiement.
  - La mise en place de cet utilisateur est un pré-requis à la mise en oeuvre de Vitam.

Indication : Sur AlmaLinux et Debian, l'interpréteur Python et les packages python requis pour l'exécution d'ansible sur les noeuds gérés sont inclus dans les packages logiciels du système, et généralement déjà installés dans les systèmes de base.

L'outil de déploiement prend en entrée :

- La topologie de l'environnement (quel composant est installé sur quel serveur)
- L'ensemble des paramètres de l'environnement

Ces 2 entrées sont définies par l'utilisateur sous la forme de fichiers ansible (fichier d'inventaire et de variables).

Prudence : L'utilisation d'ansible nécessite les droits *root* sur l'environnement cible (soit en tant qu'utilisateur *root*, soit en *sudoer*) par l'utilisateur linux faisant le déploiement. Le *DIN* contiendra les informations requises pour prendre en compte cet utilisateur.

Avertissement : L'utilisation d'une méthode de déploiement autre n'est pas supportée par le projet *VITAM*.

5.8.2 Architecture de l'outil

On dispose de 3 types de playbooks principaux :

- 1 playbook de déploiement (`ansible-vitam`) qui est le coeur du déploiement ;
- 1 playbook de déploiement (`ansible-vitam-extra`) qui contient des éléments potentiellement utiles, mais non nécessaires au fonctionnement du système ;
- N playbooks d'exploitation (`ansible-vitam-exploitation`) pour l'automatisation des actes d'exploitation (décrits dans le *DEX*).

On dispose de 2 types de rôles :

- rôle « helper » qui est appelé par les autres rôles et qui n'est pas contenu dans les playbooks ;
- rôle « service » : 1 rôle par service déployé.

L'ensemble des fichiers de configuration (devant être instanciés) sera géré par l'outil de déploiement (via le langage de templating Jinja2).

41. https://docs.ansible.com/ansible/intro_installation.html

5.8.3 Gestion des secrets

Pour les variables ayant une criticité (au sens de la sécurité - par exemple : les mots de passe de connexion aux bases de données), le déploiement *VITAM* est compatible avec l'utilisation du module Ansible Vault : celui-ci permet de chiffrer de manière symétrique les variables sensibles.

Avertissement : Cette fonctionnalité nécessite d'entrer la passphrase du fichier chiffré et donc est difficilement compatible avec une automatisation forte.

Les certificats (notamment *CA* et certificats serveur) devront être fournis au préalable et être placés dans les répertoires d'installation mentionnés dans le *DIN*.

Les composants nécessitant un certificat sont :

- ceux exposés à l'extérieur du système, à savoir les frontaux (i.e. faisant partie de la zone Accès) et storage ;
- ceux qui réalisent un horodatage sécurisé, à savoir logbook, worker et storage.

Pour chacun de ces certificats, l'intégralité des certificats des *CA* de la chaîne de certification devra également être fournie, ainsi que l'URL des *CRL* associées.

Avertissement : Les systèmes front-office en interface avec la solution *Vitam* doivent également mettre à disposition leurs certificats et chaînes de certification système, ainsi que les certificats individuels en cas d'utilisation des Personae.

Voir aussi :

La liste des secrets nécessaires au bon fonctionnement de *VITAM* est décrite dans la *section dédiée* (page 107).

5.9 Service registry

Le *service registry* est le composant permettant à chaque service de localiser les services dont il dépend ; par conséquent, son bon fonctionnement est particulièrement critique pour le bon fonctionnement de la solution logicielle *VITAM*. L'outil de *service registry* utilisé par *VITAM* est Consul [42].

5.9.1 Architecture

Un déploiement Consul est composé de 2 types de noeuds différents :

- Les noeuds serveurs : ils persistent l'état des données stockées dans Consul ; les données sont répliquées entre eux, et eux seuls participent à l'élection du maître (ils forment un cluster Raft). Un quorum de ces noeuds doit toujours être déclaré ; dans le cas contraire, on entre dans un cas de désastre de cluster (Cf. la documentation sur l"« outage recovery » [43]) ; le nombre de serveurs doit être impair, avec un minimum conseillé de 3 noeuds (pour des problématiques de maintien de quorum).
- Les noeuds client : ils exposent les *API* d'accès aux structures de données Consul, et réalisent les *healthchecks* des services dont ils ont la définition. Ils communiquent avec les serveurs.

Un noeud Consul est également appelé un agent.

Note : Les noeuds serveurs sont en fait des noeuds clients réifiés, i.e. ils ont également les capacités des clients.

42. https://www.consul.io
43. https://www.consul.io/docs/guides/outage.html

Dans le cadre de *VITAM*, le déploiement des noeuds Consul doit correspondre aux principes suivants :

- Un cluster de serveurs Consul sur un nombre impair de noeuds dédiés, chacun d'entre eux étant configuré pour exposer l'*IHM* de suivi ;
- 1 client par serveur hébergeant un service *VITAM*.

Indication : Préconisation : Le fonctionnement de Consul via trois noeuds *master* au minimum nous prémunit de la perte d'un de ces noeuds sans perturbation du service. Un seul noeud Consul est vivement déconseillé.

5.9.2 Résolution DNS

Les résolutions de noms de service se font via l'*API DNS* de Consul ; un *resolver* externe doit être configuré pour les requêtes externes.

Chaque client agit comme serveur *DNS* local ; il écoute sur le port udp 53 (sur la boucle locale - `127.0.0.1`), et est configuré comme serveur *DNS* de l'*OS* (typiquement dans le fichier `/etc/resolv.conf`).

Prudence : Cela rend Consul incompatible avec d'autres implémentations de serveur *DNS* qui seraient lancées sur l'*OS*, et en particulier les caches *DNS* installés par défaut dans certaines distributions linux (ex : dnsmasq). En outre, il faut prendre garde à l'écrasement de la configuration du `resolv.conf`, qui doit garder `127.0.0.1` comme premier serveur *DNS*.

Note : Pour pouvoir écouter sur le port 53, Consul nécessite la capacité `CAP_NET_BIND_SERVICE` (Cf. la section suivante).

Lorsque le système fait une requête *DNS*, cette dernière arrive à l'agent Consul local et la séquence suivante est exécutée :

- Si le nom à résoudre appartient au domaine réservé pour Consul (par défaut `consul`), il est résolu en tant que nom de service ou de noeud (Cf. la documentation officielle concernant l'interface DNS [44]) ;
- Dans le cas contraire, la requête est transmise aux serveurs *DNS* configurés dans la liste des recursors [45]).

Note : Consul a pour l'instant été configuré en mode `allow_stale = false` (cf. la directive de configuration [46]), ce qui signifie que chaque requête DNS se traduit par un appel RPC au noeud *leader* des serveurs Consul. Cela permet d'assurer la consistance des réponses DNS, mais peut potentiellement poser des problèmes de performance sur des larges déploiements. Il est possible de changer ce comportement (clés de configuration `allow_stale` et `max_stale` - qui permettent de préciser la durée maximum pendant laquelle le noeud répond aux requêtes DNS sans interroger le *leader*), et également de changer le *TTL* des réponses DNS (qui est par défaut gardé à 0).

5.9.3 Multi-site

En multi-site, la solution logicielle *VITAM* exploite les *datacenters* consul. Un *datacenter* consul est créé par site.

Chaque site doit posséder au moins 3 serveurs consul, qui ne supervisent que les services dans le datacenter auquel ils sont rattachés.

44. https://www.consul.io/docs/agent/dns.html
45. https://www.consul.io/docs/agent/options.html#recursors
46. https://www.consul.io/docs/agent/options.html#allow_stale

5.9.4 Packaging

La solution logicielle *VITAM* intègre des packages *OS* (rpm & deb) dédiés pour Consul ; ces packages permettent essentiellement :

- De configurer Consul en tant que service systemd ;
- De permettre le lancement de Consul sous l'utilisateur `vitam` ;
- Enfin, ils intègrent une directive `setcap` de post-install pour attribuer la capacité `CAP_NET_BIND_SERVICE` au binaire `/vitam/bin/consul/consul` afin de permettre à ce dernier d'exposer une interface *DNS* sur le port 53 sans pour autant nécessiter les droits *root*.

5.9.5 Monitoring

Chaque instance de service doit être déclarée dans Consul ; cette déclaration se fait en déposant un fichier de configuration dans le répertoire de configuration de Consul. Ce fichier contient notamment l'identifiant du service ainsi que son port d'écoute, ainsi qu'une liste de *healthchecks* qui permettent à Consul de connaître l'état du service. Pour les services *VITAM*, ces *healthchecks* s'appuient sur les *API* de supervision qui ont été décrites dans *la section dédiée* (page 42).

Consul permet d'exposer une *IHM* Web permettant d'accéder à la topologie des services déployés (i.e. quel service sur quel noeud) et à leur état instantané.

5.10 Dépendances aux services d'infrastructures

5.10.1 Ordonnanceurs techniques / batchs

L'ordonnancement technique se fait par le biais de timers systemd, dont la liste est donnée dans le *DAT*.

5.10.1.1 Curator

Curator permet d'effectuer des opérations périodiques de maintenance sur les index elasticsearch. Les jobs Curator sont initiés automatiquement au déploiement de *VITAM* et sont lancés via un timer systemd [47] sur chaque serveur.

Voir aussi :

Plus de détails sont disponibles dans *la présentation de curator* (page 79).

5.10.1.2 Sécurisation des journaux d'opérations

Job de sécurisation du logbook : lancé toutes les nuits peu après minuit sur une des machines (la dernière dans la liste de déploiement) hébergeant le composant `vitam-logbook`.

5.10.1.3 Sécurisation des journaux d'écriture

La sécurisation des journaux d'écriture est un processus local à chaque serveur hébergeant une instance du moteur de stockage.

47. https://www.freedesktop.org/software/systemd/man/systemd.timer.html

5.10.1.4 Sécurisation des cycles de vie

Job de sécurisation des cycles de vie des *Unit* et *objectGroup* : lancé sur une des machines (la dernière dans la liste de déploiement) hébergeant le composant `vitam-logbook`.

5.10.1.5 Cas de la sauvegarde

Se référer au *DEX*.

5.10.2 Socles d'exécution

5.10.2.1 Middlewares

- Java : JRE 21 ; les versions suivantes ont été testées :
  - OpenJDK 21, dans la version présente dans les dépôts officiels au moment de la parution de cette release de *Vitam* (AlmaLinux 9, AlmaLinux 10 et Debian 12)

5.11 Composants déployés

Cette section vise à décrire les particularités des différents composants déployés dans le cadre d'une solution *VITAM* ; chaque service est nommé suivant son `service_id`.

Les estimations de consommation de ressources sont données pour un système équilibré en `ingest`, `audit` et `access` ; elles sont à adapter pour chaque composant en fonction des cas d'utilisation des systèmes (ex : archivage définitif VS archivage courant) (Cf. *les guidelines de dimensionnement* (page 92)).

5.11.1 Access-external

Type : Composant VITAM Java
Données stockées :
- Cache d'authentification M2M (mémoire) ;
- Certificats x509 d'authentification clients

Typologie de consommation de ressources :
- CPU : faible
- Mémoire : faible
- Réseau : généralement faible, sauf dans le cas de sortie massive d'archives (sortant)
- Disque : faible (logs)

5.11.2 Access-internal

Type : Composant VITAM Java
Données stockées :
- Aucune

Typologie de consommation de ressources :
- CPU : faible
- Mémoire : faible
- Réseau : généralement faible, sauf dans le cas de sortie massive d'archives (sortant)
- Disque : faible (logs)

5.11.3 Batch-report

Type : Composant VITAM Java

Typologie de consommation de ressources :

- CPU : faible
- Mémoire : faible
- Réseau : généralement faible, sauf dans le cas de sortie massive d'archives (sortant)
- Disque : faible (logs)

5.11.4 Collect-external

Type : Composant VITAM Java

Données stockées :

- Cache d'authentification M2M (mémoire) ;
- Certificats x509 d'authentification clients

Typologie de consommation de ressources :

- CPU : faible
- Mémoire : faible
- Réseau : généralement faible, sauf dans le cas d'entrées massive d'archives (entrant)
- Disque : faible (logs)

5.11.5 Collect-internal

Type : Composant VITAM Java

Données stockées :

- Aucune

Typologie de consommation de ressources :

- CPU : faible
- Mémoire : faible
- Réseau : généralement faible, sauf dans le cas d'entrées massive d'archives (entrant)
- Disque : faible (logs)

5.11.6 Consul

Type : COTS

Données stockées :

- État du cluster et localisation des services

Typologie de consommation de ressources :

- **Serveurs :**
  - CPU : faible
  - Mémoire : faible
  - Réseau : faible
  - Disque : très faible

- **Agents :**
  - CPU : faible
  - Mémoire : faible
  - Réseau : faible
  - Disque : très faible

Prudence : Consul est un service critique d'infrastructure ! Un dysfonctionnement de ce service peut rapidement entraîner une panne générale du système.

5.11.6.1 Architecture de déploiement

L'architecture de déploiement conseillée correspond aux principes présentés dans *la section d'introduction à Consul* (page 74) :

- 2n + 1 noeuds pour les serveurs ; chaque noeud serveur doit répondre aux requêtes RPC des agents et expose l'IHM de suivi de l'état du cluster consul. Un déploiement typique comporte 3 noeuds serveur. Les données sont répliquées sur tous les serveurs.
- 1 noeud agent par serveur hébergeant des services VITAM ; chaque noeud agent agit comme serveur DNS local.

Les ports utilisés par Consul sont les suivants :

- `tcp:8300` : Port RPC ; il permet aux agents d'exécuter des requêtes vers les serveurs.
- `tcp:8301` : Port de « gossip » ; il permet la découverte automatique des agents entre eux, et la propagation des événements du cluster vers tous les noeuds.
- `tcp:8400` : Port RPC local ; il est utilisé par la console consul locale (CLI).
- `tcp:8500` : Port HTTP ; il est notamment utilisé par les noeuds serveur pour servir l'interface de monitoring et d'administration.
- `udp:53` & `tcp:53` : Port d'écoute DNS

5.11.7 Curator

Curator permet de gérer les index d'Elasticsearch des logs techniques et d'en assurer la maintenance (fermeture des index non utilisés, suppression des index obsolètes, ...)

Curator est colocalisé avec les noeuds Elasticsearch de log. Il est lancé sur chaque noeud par un timer systemd avec le flag `--master-only`. Ce mode de fonctionnement permet d'avoir un service Curator possédant le même degré de résilience que le cluster Elasticsearch dont il assure la maintenance des index.

Voir aussi :

Plus d'information est disponible dans la documentation officielle [48].

Type : COTS

Données stockées :

- Aucune

Typologie de consommation de ressources :

- CPU : très faible
- Mémoire : très faible
- Réseau : très faible
- Disque : très faible

48. https://www.elastic.co/guide/en/elasticsearch/client/curator/3.5/master-only.html

5.11.8 Elasticsearch-data

Cluster d'indexation dédié aux données métier.

Type : COTS

Données stockées :

- Index de recherche des données d'archive

Typologie de consommation de ressources :

- CPU : forte
- Mémoire : forte
- Réseau : forte
- Disque : forte

5.11.8.1 Architecture de déploiement

Dans le paramétrage par défaut du déploiement, tous les noeuds sont considérés comme des noeuds « master » et « data » ; par conséquent, le nombre de noeuds du cluster doit être impair (i.e. 2n + 1 noeuds, n > 0).

2 types de clients sont utilisés dans VITAM :

- les clients « transports » : ils sont utilisés par les composants développés dans le cadre de la solution logicielle (notamment les composants metadata, functional-administration, logbook). Ils sont considérés par le cluster elasticsearch comme membres du cluster, de type « client » ;
- les clients « http » : ils sont utilisés par les composants d'administration (cerebro, curator).

5.11.8.2 Monitoring

Le monitoring d'elasticsearch est possible :

- soit à partir des API http (notamment les "cat APIs" [49], les API de gestion des index [50] ou les API de gestion du cluster [51]) ;
- soit en utilisant le composant Cerebro (Cf. la page officielle [52]) installé dans le cadre de la solution logicielle VITAM.

5.11.9 Elasticsearch-log

Cluster dédié à la centralisation des logs applicatifs.

Type : COTS

Données stockées :

- Logs techniques des composants déployés dans le cadre de VITAM (services java, bases de données, composants de support (logstash, curator))

Typologie de consommation de ressources :

- CPU : moyenne
- Mémoire : forte
- Réseau : forte
- Disque : forte

49. https://www.elastic.co/guide/en/elasticsearch/reference/5.6/cat.html
50. https://www.elastic.co/guide/en/elasticsearch/reference/5.6/indices.html
51. https://www.elastic.co/guide/en/elasticsearch/reference/5.6/cluster.html
52. https://github.com/lmenezes/cerebro

5.11.9.1 Architecture de déploiement

Voir aussi :

Se reporter à *Elasticsearch-data* (page 80) pour les informations générales concernant elasticsearch.

Dans le paramétrage par défaut du déploiement, tous les noeuds sont considérés comme des noeuds « master » et « data » ; par conséquent, le nombre de noeuds du cluster doit être impair (i.e. 2n + 1 noeuds, n > 0).

5.11.10 Functional-administration

Type : Composant VITAM Java

Données stockées :

- Fichiers temporaires : fichiers de chargement des référentiels

Typologie de consommation de ressources :

- CPU : faible
- Mémoire : faible
- Réseau : faible
- Disque : moyenne (utilisation du répertoire temporaire pour des chargements de fichiers de référentiel)

5.11.11 Grafana

Visualisation des données ElasticSearch et Prometheus.

Type : COTS EXTRA

Données stockées :

- Dashboards

5.11.11.1 Architecture de déploiement

Pour plus d'informations, veuillez vous référer à la documentation officielle : https://grafana.com/docs/

5.11.11.1.1 Ports utilisés

Le port utilisé par le serveur Grafana est le suivant :

- `tcp:3000` : Port d'écoute modifiable via la variable `grafana.http_port` dans le fichier `environments/group_vars/all/advanced/cots_vars.yml`.

5.11.12 Ingest-external

Type : Composant VITAM Java

Données stockées :

- Cache d'authentification M2M (mémoire) ;
- Certificats x509 d'authentification clients ;
- Fichiers SEDA (sas de validation de conformité et sanity checks)

Typologie de consommation de ressources :

- CPU : faible

- Mémoire : faible
- Réseau : généralement faible, sauf dans le cas d'entrées massive d'archives (entrant)
- Disque : important (stockage temporaire des fichiers SEDA entrants)

Voir aussi :

Ce composant fait également appel *au composant Siegfried* (page 90) pour l'identification des formats de fichier.

5.11.12.1 Antivirus

Lors de l'entrée d'un fichier SEDA, ce dernier est soumis à un scan antivirus. L'antivirus utilisé est configurable ; la configuration du service `ingest-external` (effectuée dans le fichier `ingest-external.conf`) permet de définir un exécutable (ou script shell) qui est lancé pour réaliser l'analyse antivirale. Cet exécutable doit respecter le contrat suivant :

- Sémantique des codes de retour
  - 0 : Analyse terminée - aucun virus trouvé
  - 1 : Analyse terminée - virus trouvé et corrigé
  - 2 : Analyse terminée - virus trouvé mais non corrigé
  - 3 : Analyse en échec
- Arguments
  - Argument 1 : chemin absolu du fichier à analyser
- Streams de sortie
  - stdout :
    - Si l'analyse se termine : nom des virus trouvés, un par ligne
    - Si l'analyse échoue : raison de l'échec
  - stderr :
    - Messages de log de l'antivirus

5.11.13 Ingest-internal

Type : Composant VITAM Java
Données stockées : Aucune
Typologie de consommation de ressources :

- CPU : faible
- Mémoire : faible
- Réseau : généralement faible, sauf dans le cas d'entrées massive d'archives (entrant)
- Disque : faible (logs)

5.11.14 Kibana

Kibana est une application web permettant de faire des recherches et de construire des dashboards à partir des données des logs, techniques ou métier.
Type : COTS
Données stockées : Aucune
Typologie de consommation de ressources :

- CPU : très faible
- Mémoire : très faible
- Réseau : faible
- Disque : très faible

5.11.14.1 Déploiement

Kibana (à partir de sa version 4) se présente sous la forme d'un serveur web qui a deux fonctions :

- servir les ressources nécessaires à l'application web qui s'exécute dans le navigateur internet client ;
- agir comme proxy pour les requêtes émises par le navigateur internet à destination de la base d'index de logs (elasticsearch-log ou elasticsearch-data).

Ainsi, aucun accès direct entre un navigateur client et les serveurs elasticsearch n'est requis pour la visualisation des données des logs.

5.11.15 Logbook

Type : Composant VITAM Java

Données stockées :

- Certificat d'horodatage

Typologie de consommation de ressources :

- CPU : moyenne
- Mémoire : moyenne
- Réseau : moyenne
- Disque : faible (logs)

5.11.16 Logstash

Type : COTS

Données stockées : Aucune

Typologie de consommation de ressources :

- CPU : moyenne
- Mémoire : moyenne
- Réseau : forte
- Disque : faible (logs)

5.11.17 Metadata

Type : Composant VITAM Java

Données stockées : Aucune

Typologie de consommation de ressources :

- CPU : moyenne
- Mémoire : moyenne
- Réseau : moyenne
- Disque : faible (logs)

5.11.18 Metadata Collect

Type : Composant VITAM Java

Données stockées : Aucune

Typologie de consommation de ressources :

- CPU : moyenne
- Mémoire : moyenne
- Réseau : moyenne
- Disque : faible (logs)

5.11.19 Mongodb

5.11.19.1 Base mongo-data

Base de données dédiée de type documents, utilisée pour le stockage des données métier (un cluster mongodb par site).

Type : COTS

Données stockées :

- Données d'archives
- Journaux métier
- Référentiels métier

Typologie de consommation de ressources :

- CPU : moyenne
- Mémoire : forte
- Réseau : forte
- Disque : forte

5.11.19.2 Base mongo-offer

Base de données dédiée à chaque offre de stockage utilisée pour la persistence des données techniques des offres (un cluster mongodb par offre de stockage).

Type : COTS

Données stockées :

- Offset d'écriture dans les offres
- Catalogue technique pour les offres froides : Listing des bandes magnétiques, des objets archivés sur bandes…

Typologie de consommation de ressources :

- CPU : moyenne
- Mémoire : forte (en particulier pour une offre froide)
- Réseau : forte (en particulier pour une offre froide)
- Disque : forte (en particulier pour une offre froide)

5.11.19.3 Architecture de déploiement

5.11.19.3.1 Architecture 1 noeud

L'architecture à 1 noeud est uniquement constituée d'un noeud mongod ; elle n'est pas supportée par VITAM.

5.11.19.3.2 Architecture distribuée

Une architecture MongoDB distribuée utilise les notions suivantes :

- **Sharding**
  - Mongodb utilise le sharding pour scaler la base de données (scalabilité horizontale)
  - Le sharding distribue les données à travers les n partitions physiques (shards) dont le cluster est composé
  - Bien choisir la clé de sharding est primordial pour une répartition égale des documents insérés dans les différents shards
  - Chaque shard est composé d'un Replica Set
- **Replica Set (RS)**
  - Les Replica Sets assurent la haute disponibilité de Mongodb
  - Un Replica Set est (règles Mongodb de production) composé de `2 x n + 1` noeuds, avec `n >= 1` (1 noeud primaire, les autres étant des noeuds secondaires) ; le noeud primaire est choisi de manière arbitraire par MongoDB dans la liste des noeuds du Replica Set
  - L'écriture se fait obligatoirement sur le noeud primaire
- **Replica Set de config**
  - Un Replica Set est dédié pour le stockage de la configuration du cluster
  - Comme tous les autres Replica Sets, il est recommandé de le peupler de `2 x n + 1` noeuds, avec `n >= 1`
- **Routeur de requêtes**
  - Le routeur mongos permet de rediriger une requête sur le ou les shards requis, en fonction de la clé de sharding ; il agit comme coordinateur de requête

Une architecture MongoDB distribuée comprend 3 types de noeuds différents :

- mongod : stockent les données des Replica Sets métier ;
- mongos : routent les requêtes ;
- mongoc : stockent les données d'état et de configuration du cluster (ces noeuds utilisent en fait un moteur mongod, mais pour un Replica Set particulier : le Replica Set de configuration).

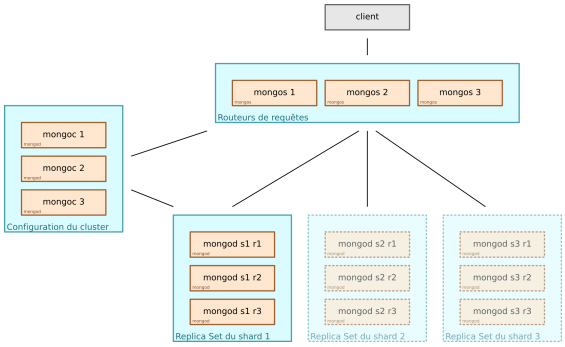


FIG. 12 – Déploiement d'un cluster Mongo DB avec sharding.

L'architecture proposée dans le cadre de VITAM consiste à séparer les noeuds liés au routage des requêtes et de gestion du cluster d'une part (donc de colocaliser mongos et mongoc), avec les noeuds de stockage des données (mongod) d'autre part.

Ainsi, avec n shards et r noeuds par Replica Set (cluster), on obtient le déploiement suivant :

- au moins 3 serveurs config / service, chacun hébergeant :
  - 1 noeud mongos (service)
  - 1 noeud mongoc (Replica Set de configuration)
- n x r serveurs, chacun hébergeant :
  - 1 noeud mongod

Note : Une typologie de cluster complète (mongos, mongoc, mongod) est systématiquement déployée dans le cadre de la solution logicielle VITAM, cela afin de permettre une extension ultérieure du cluster par le rajout d'un nouveau shard et le rééquilibrage du cluster, et ce même si un seul shard est instancié au démarrage.

Prudence : Une colocalisation des composants (mongos, mongoc & mongod) reste possible dans un déploiement avec un seul shard ; cette colocalisation n'est plus possible dans le cas d'un déploiement multi-shardé et il sera nécessaire d'avoir des noeuds dédiées (mongos & mongoc) et des noeuds dédiés pour chacun des shards (mongod) selon les recommandations précédentes.

5.11.19.3.3 Ports utilisés

Les ports utilisés par mongodb sont les suivants :

- `tcp:27017` : Port de communication pour les noeuds mongos
- `tcp:27018` : Port d'écoute des noeuds du Replica Set de config (mongoc)
- `tcp:27019` : Port d'écoute des noeuds du/des Replica Set(s) de données (mongod)

5.11.20 Offer

Type : Composant VITAM Java

Données stockées :

- Cache d'authentification M2M (mémoire) ;
- Certificats x509 d'authentification clients

Données gérées :

- Données d'archives

Typologie de consommation de ressources :

- CPU : moyenne
- Mémoire : forte (principalement pour le cache I/O)
- Réseau : forte
- Disque : forte (notamment pour les offres de système de fichiers et les offres froides)

5.11.20.1 Types d'offre de stockage

Par le biais du composant `offer`, la solution VITAM permet d'utiliser les types d'offre de stockage décrits dans *Stockage des données* (page 56).

Dans les offres Système de fichiers, les données sont stockées selon une arborescence à 4 niveaux de profondeur qui est déterminée par le hash du nom de fichier. Cette arborescence est dans une structure de fichiers composée par un numéro de tenant et un élément identifiant (ex. : `0_objectGroup/`), elle-même dans l'emplacement `/vitam/data/offer/container/`.

Dans les offres froides, un stockage local dans `/vitam/data/offer` est nécessaire. Il est utilisé comme espace « tampon » pour l'écriture rapide et comme espace « cache » pour la relecture des données.

5.11.20.1.1 Cas des *containers* objet

Dans le cas d'utilisation de stockage objet, il faut prévoir, par tenant VITAM, la création de 17 *containers*. Se référer au *DEX* pour plus d'informations.

5.11.21 Processing

Type : Composant VITAM Java

Données stockées : Aucune

Typologie de consommation de ressources :

- CPU : faible
- Mémoire : moyenne
- Réseau : faible
- Disque : faible (logs)

5.11.22 Prometheus server

Supervision des événements et des alertes

Type : COTS EXTRA

Données stockées :

- Métriques techniques
- Métriques métier

5.11.22.1 Architecture de déploiement

Veuillez vous référer à la documentation officielle.

5.11.22.1.1 Ports utilisés

Le port utilisé par le serveur prometheus est le suivant :

- `tcp:9090` : Port d'écoute modifiable via la variable `prometheus.server.port` dans le fichier `environments/group_vars/all/advanced/cots_vars.yml`.

5.11.23 Prometheus alertmanager

Envoi des alertes.

Type : COTS EXTRA

5.11.23.1 Architecture de déploiement

Veuillez vous référer à la documentation officielle.

5.11.23.1.1 Ports utilisés

Les ports utilisés par Prometheus alertmanager sont les suivants :

- `tcp:9093` : Port d'API modifiable via la variable `prometheus.alertmanager.api_port` dans le fichier `environments/group_vars/all/advanced/cots_vars.yml`
- `tcp:9094` : Port de cluster modifiable via la variable `prometheus.alertmanager.cluster_port` dans le fichier `environments/group_vars/all/advanced/cots_vars.yml`

5.11.24 Prometheus node_exporter

Exposition des métriques liées au matériel et au noyau du système.

Type : COTS

5.11.24.1 Architecture de déploiement

Ce composant est à installer sur chacune des VMs ou matériels à superviser.

5.11.24.1.1 Ports utilisés

Le port utilisé par Prometheus node_exporter est le suivant :

- `tcp:9101` : Port d'écoute modifiable via la variable `prometheus.node_exporter.port` dans le fichier `environments/group_vars/all/advanced/cots_vars.yml`

5.11.24.1.2 API exposées

Prometheus node_exporter expose l'API suivant :

- `/metrics` : API sur laquelle les métriques sont exposées.

5.11.25 Prometheus Elasticsearch Exporter

Exposition des métriques liées à Elasticsearch.

Type : COTS

5.11.25.1 Architecture de déploiement

Ce composant est à installer sur chacune des VMs où des services Elasticsearch sont à superviser.

5.11.25.1.1 Ports utilisés

Le port utilisé par Prometheus Elasticsearch Exporter est le suivant :

- `tcp:9114` : Port d'écoute modifiable via la variable `prometheus.node_exporter.port` dans le fichier `environments/group_vars/all/advanced/cots_vars.yml`

5.11.25.1.2 API exposées

Prometheus Elasticsearch Exporter expose l'API suivant :

- `/metrics` : API sur laquelle les métriques sont exposées.

5.11.26 restic

Outil de sauvegarde opensource. Utilisé pour le backup des bases mongo-offer.

Supporte les différents providers suivants : *filesystem*, *filesystem-hash*, *openstack-swift-v3* et *amazon-s3-v2*.

Note : La sauvegarde des bases pour les offres froides est détaillée dans le *DEX*.

Type : COTS EXTRA (beta)

Données stockées :

- Sauvegarde des bases mongodb.

5.11.26.1 Architecture de déploiement

restic sera déployé sur l'ensemble des machines renseignées dans le groupe `[hosts_offer]` de votre fichier d'inventaire qui possèdent le paramètre `restic_enabled=true`. Attention à ne mettre ce paramètre que pour l'une des machines pour chacun des `offer_conf`.

Afin de permettre aux machines du groupe `[hosts_offer]` de faire les backups, elles doivent pouvoir accéder aux bases de données afin de pouvoir effectuer la commande *mongodump*. Ainsi, les flux réseaux doivent permettre d'atteindre les bases configurées (`{{ restic.backup.host }}`).

Pour des raisons de sécurité, restic est déployé sur ces machines afin d'éviter de diffuser les paramètres d'accès aux offres de stockage (notamment dans le cas des offres objet).

De plus, en cas de défaillance critique du système Vitam, tant que les offres de stockage sont intactes, il sera plus facile de ressortir les snapshots des bases de données associées aux données de l'offre sous-jacente.

5.11.27 Scheduler

Type : Composant VITAM Java

Données stockées : Aucune

Typologie de consommation de ressources :

- CPU : faible
- Mémoire : moyenne
- Réseau : faible
- Disque : faible (logs)

5.11.28 Security-internal

Type : Composant VITAM Java

Données stockées : Aucune

Typologie de consommation de ressources :

- CPU : faible
- Mémoire : faible
- Réseau : faible
- Disque : faible (logs)

5.11.28.1 API d'administration

Ce composant possède des *API REST* d'administration permettant de réaliser l'ajout et la consultation des profils de sécurité dans la base des contextes (endpoint : `http://<ip_admin>:<port_admin>/v1/admin/securityprofiles`)

5.11.29 Siegfried

Type : Composant binaire d'identification de format de fichiers

Données stockées :

- Aucune

Typologie de consommation de ressources :

- CPU : faible
- Mémoire : faible
- Réseau : très faible, et sur localhost uniquement
- Disque : faible (logs)

> **Avertissement :** Dans cette version du système VITAM, le référentiel des formats utilisé par Siegfried ne peut pas être mis à jour facilement, et aucune validation automatique de cohérence avec le référentiel des formats chargé dans VITAM n'est effectuée.

5.11.29.1 Mode de fonctionnement dans VITAM

Dans VITAM, Siegfried est utilisé dans son mode serveur accédant à des fichiers locaux ; dans ce cadre, le serveur Siegfried est uniquement bindé sur `localhost`, et donc uniquement accessible à des processus locaux à ce serveur.

L'utilisation typique de Siegfried par un composant est donc la suivante :

- Appel du serveur siegfried sur localhost ; cet appel contient uniquement une demande de traitement, et contient le chemin d'un fichier local à analyser ;
- Siegfried réalise l'analyse du fichier local ;
- Siegfried répond à la requête en indiquant le format du fichier analysé.

5.11.30 Storage

Type : Composant VITAM Java

Données stockées : Aucune

Typologie de consommation de ressources :

- CPU : forte
- Mémoire : moyenne
- Réseau : forte
- Disque : moyenne (logs & traces d'écriture)

5.11.31 Worker

Type : Composant VITAM Java

Données stockées :

- Certificat d'horodatage

Typologie de consommation de ressources :

- CPU : forte
- Mémoire : forte
- Réseau : forte (entrant et sortant)
- Disque : moyenne (logs + fichiers temporaires de travail)

Voir aussi :

Ce composant fait également appel *au composant Siegfried* (page 90) pour l'identification des formats de fichier.

5.11.31.1 Particularités

Les workers utilisent des outils externes pouvant avoir des pré-requis importants sur les OS utilisés ; pour réduire l'impact sur les systèmes, ces outils pourront être à terme packagés dans des conteneurs Docker. Cependant, aucun conteneur Docker n'est fourni ni supporté dans cette version de la solution VITAM.

5.11.32 Workspace

Type : Composant VITAM Java

Données stockées : Aucune

Typologie de consommation de ressources :

- CPU : moyenne
- Mémoire : forte (notamment pour le cache d'I/O système)
- Réseau : forte
- Disque : forte (zone d'échange des données de travail entre tous les composants)

5.11.33 Workspace Collect

Type : Composant VITAM Java

Données stockées : Aucune

Typologie de consommation de ressources :

- CPU : moyenne
- Mémoire : forte (notamment pour le cache d'I/O système)
- Réseau : forte
- Disque : forte (zone d'échange des données de travail entre tous les composants)

5.12 Guidelines de déploiement

Les principes de *zoning* associés à l'architecture de la solution logicielle *VITAM* ont été présentés *lors de la description des principes de déploiement* (page 39) ; cette section a pour but de compléter ces principes par des recommandations concernant la colocalisation des composants.

De manière générale, pour des raisons de sécurité, il est déconseillé de colocaliser des composants appartenant à des zones différentes. Il est par contre possible de colocaliser des composants appartenant à des sous-zones différentes dans la zone des services internes ; ainsi, les colocalisations des composants suivants sont relativement pertinentes :

- ingest-external, access-external et administration-external, hors contraintes particulières de sécurité ;
- ingest-internal et access-internal ;
- elasticsearch-data et mongod ;
- mongos et mongoc ;
- logstash, elasticsearch-log (mono-instance), kibana (pour les déploiements de tests) ; elasticsearch-log et consul (serveur) (pour des déploiements de taille moyenne)
- workspace et storage ;
- prometheus-server, prometheus-alertmanager, grafana-server

Prudence : Il est recommandé de ne pas colocaliser les composants restants :

- offer étant dans une zone logique particulière ;
- worker ayant une consommation de ressources système potentiellement importante.

5.13 Eléments de dimensionnement

Prudence : Les abaques de dimensionnement sont étroitement liés à la nature de l'infrastructure sous-jacente et à l'usage qui est fait de la solution logicielle *VITAM*. Par conséquent, les indications de volumétrie qui sont présentées dans la suite de ce document sont purement indicatives et relatives au système *VITAM* dans sa version actuelle, installé sur les environnements de tests de la solution logicielle (qui sont opérés en environnement complètement virtualisé).

Prudence : Réaliser un benchmarking sur un environnement de test de charge (avec des jeux de données et une infrastructure similaires à celle de production) est fortement recommandé pour valider le bon dimensionnement de la plateforme.

Note : Sauf mention contraire, les enveloppes de ressources ci-dessous comprennent notamment les composants associés à l'exploitation de la solution logicielle *VITAM* et fournis dans le cadre de la solution (traitement et stockage des logs et des métriques, gestion des bases de données, …)

Important : Les configurations de référence ci-dessous sont données pour un seul site primaire comportant une seule offre de stockage. *VITAM* préconise très fortement un déploiement comportant *a minima* 2 offres de stockage. En fonction des contraintes de disponibilité du système, il sera donc nécessaire :

- Soit d'ajouter une autre offre de stockage (i.e. les composants offer et mongo*-offer) dans le cas d'un déploiement mono-site ;
- Soit d'ajouter un site secondaire comportant sa propre offre de stockage.

Important : Un monitoring régulier de la plateforme (via prometheus par exemple) est nécessaire. En plus de permettre la vérification du bon fonctionnement de la solution, il permet également de valider son dimensionnement, et de le réadapter si besoin.

Important : Les composants `workspace` et `processing` sont mono-instantiables uniquement. De même, pour les composants `offer` de type « Système de fichier » ou « Bandes magnétiques » (offre froide), une seule instance peut être déployée par offre. Les composants `offer` de type « S3 » ou « Swift » sont multi-instantiables.

5.13.1 Compute

5.13.1.1 « xsmall » : développement local

Adapté à un poste de développement ; ce déploiement ne comprend pas les composants d'exploitation de la solution *VITAM*. La chaîne de traitement de logs n'est pas déployée, et le même cluster mongodb est utilisé pour l'offre de stockage et les métadonnées.

Ce déploiement n'est pas adapté pour un fonctionnement en production.

TABLEAU 1 – Dimensionnement XSmall

Zone	Composants	# instances	vCPU / instance	RAM / instance
• zone_access • zone_external • zone_applicative • zone_data • zone_storage • zone_admin	• hosts_ihm_demo • hosts_ihm_recette • hosts_cerebro • hosts_ingest_external • hosts_access_external • hosts_ingest_internal • hosts_access_internal • hosts_storage • hosts_workspace • hosts_processing • hosts_worker • hosts_functional_administration • hosts_logbook • hosts_security_internal • hosts_batch_report • hosts_scheduler • hosts_metadata • hosts_mongoc_data • hosts_mongos_data • hosts_mongod_data • hosts_elasticsearch_data • hosts_offer (file) • hosts_consul_server	1	4	16 Go
TOTAL GLOBAL ENV	vitam	1	4	16 Go

Note : Pour ce type d'environnement, il est recommandé de définir un paramètre `elasticsearch_memory` (pour les composants `elasticsearch-log` et `elasticsearch-data`) avec une taille faible et compatible avec les ressources disponibles, afin de ne pas rencontrer de phénomènes de *OOM*.

Voir aussi :

Se reporter au *DIN* pour plus d'informations.

5.13.1.2 « small » : recette simple métier

Adapté à un environnement de recette simple d'application métier utilisant *VITAM*.

Ce déploiement n'est pas adapté pour un fonctionnement en production.

TABLEAU 2 – Dimensionnement Small

Zone	Composants	# instances	vCPU / instance	RAM / instance
• zone_external • zone_applicative	• hosts_ingest_external • hosts_access_external • hosts_ingest_internal • hosts_access_internal • hosts_storage • hosts_workspace • hosts_processing • hosts_worker	1	6	12 Go
• zone_access • zone_applicative • zone_data	• hosts_ihm_demo • hosts_ihm_recette • hosts_functional_administration • hosts_security_internal • hosts_logbook • hosts_batch_report • hosts_scheduler • hosts_metadata • hosts_mongoc_data • hosts_mongos_data • hosts_mongod_data • hosts_elasticsearch_data	1	6	12 Go
• zone_storage • zone_admin	• hosts_offer (file) • hosts_mongoc_offer	1	6	12 Go
	• hosts_mongos_offer • hosts_mongod_offer			

S'agissant d'un environnement de recette, l'utilisation de 2 offres de stockages ou de 2 sites est possible, mais non préconisée (il s'agit d'un environnement de recette métier, et non technique).

Note : Pour ce type d'environnement, il est recommandé de définir un paramètre `elasticsearch_memory` (pour les composants `elasticsearch-log` et `elasticsearch-data`) avec une taille faible et compatible avec les ressources disponibles, afin de ne pas rencontrer de phénomènes de *OOM*.

Voir aussi :

Se reporter au *DIN* pour plus d'informations.

5.13.1.3 « medium » : production pour volumétries moyennes

Adapté à un déploiement simple pour des volumétries moyennes (quelques To / an) ; seuls le worker et les composants stockant des données sont multi-instanciés (i.e. les bases de données et les offres de stockage). L'offre de stockage proposée est une offre de stockage « file », plus simple à exploiter et compatible avec une volumétrie moyenne.

Sur les 3 serveurs mongod et mongoc pour l'offre de stockage, l'un d'eux est déployé en tant qu'arbitre (participe au quorum du replica set, mais ne stocke pas de données).

TABLEAU 3 – Dimensionnement Medium

Zone	Composants	# instances	vCPU / instance	RAM / instance
zone_access	• hosts_ihm_demo	1	1	2 Go
zone_external	• hosts_ingest_external • hosts_access_external	1	1	2 Go
zone_applicative	• hosts_ingest_internal • hosts_access_internal • hosts_functional_administration • hosts_logbook • hosts_security_internal • hosts_metadata	1	4	4 Go
zone_applicative	• hosts_storage • hosts_workspace • hosts_batch_report • hosts_scheduler • hosts_processing	1	4	4 Go
zone_applicative	• hosts_worker	2	4	4 Go
zone_storage	• hosts_offer (file) • hosts_mongoc_offer (arbitre) • hosts_mongod_offer (arbitre)	1	4	4 Go
zone_storage	• hosts_mongoc_offer • hosts_mongos_offer	2	2	4 Go

Comme précisé précédemment, ce dimensionnement ne contient qu'une seule offre de stockage ; il devra être complété de préférence par un deuxième site (avec le même dimensionnement), ou bien par une offre de stockage supplémentaire sur le site principal (en doublant les ressources allouées à la zone storage).

5.13.1.4 « large » : production pour volumétries moyennes avec besoin de résilience

Adapté à un déploiement résilient pour des volumétries plus importantes (10 à 20 To / an) ; ce déploiement comprend au moins deux instances pour tous les composants le supportant, et passe à une offre de stockage objet Swift ou S3 (pour une meilleure scalabilité de l'offre).

TABLEAU 4 – Dimensionnement Large

Zone	Composants	# instances	vCPU / instance	RAM / instance
zone_access	• hosts_ihm_demo	1	2	4 Go
zone_external	• hosts_ingest_external • hosts_access_external	2	1	4 Go
zone_applicative	• hosts_ingest_internal • hosts_access_internal • hosts_functional_administration	2	1	4 Go
zone_applicative	• hosts_logbook • hosts_batch_report • hosts_security_internal • hosts_metadata • hosts_storage	2	4	4 Go
zone_applicative	• hosts_worker	3	4	4 Go
zone_applicative	• hosts_workspace • hosts_batch_report • hosts_scheduler • hosts_processing	1	4	6 Go
zone_storage	• hosts_offer (swift/s3) • hosts_mongoc_offer • hosts_mongos_offer • hosts_mongod_offer	3	4	8 Go
zone_data	•	3	4	6 Go

Comme précisé précédemment, ce dimensionnement ne contient qu'une seule offre de stockage ; il devra être complété de préférence par un deuxième site (avec le même dimensionnement), ou bien par une offre de stockage supplémentaire sur le site principal (en doublant les ressources allouées à la zone storage).

Note : Le composant `batch-report` est multi-instanciable et peut donc être colocalisé avec les composants mono-instanciables suivants : `workspace` et `processing`. L'alternative est de colocaliser avec la zone applicative comprenant `logbook`, `security-internal`, `metadata` et `storage`.

5.13.1.5 « xlarge » : production pour fortes volumétries

Adapté à un déploiement pour de fortes volumétries (ordre de grandeur des capacités d'ingest : > 50 To / an, > 100.10^6 objets / an). Ce déploiement implique la multi-instanciation de tous les composants le supportant et l'usage d'un stockage objet Swift ou S3.

TABLEAU 5 – Dimensionnement XLarge

Zone	Composants	# instances	vCPU / instance	RAM / instance
zone_access	• hosts_ihm_demo	1	2	4 Go
zone_external	• hosts_ingest_external • hosts_access_external	2	2	4 Go
zone_applicative	• hosts_ingest_internal • hosts_access_internal	2	2	4 Go
zone_applicative	• hosts_logbook • hosts_security_internal	3	4	4 Go
zone_applicative	• hosts_metadata • hosts_functional_administration	3	4	4 Go
zone_applicative	• hosts_processing	1	2	4 Go
zone_applicative	• hosts_worker	10	4	6 Go
zone_applicative	• hosts_workspace	1	8	8 Go
zone_applicative	• hosts_batch_report	2	8	4 Go
zone_applicative	• hosts_scheduler	2	8	4 Go
zone_applicative	• hosts_storage	4	4	4 Go
zone_storage	• hosts_offer (swift/s3)	2	4	4 Go
zone_storage	• hosts_mongoc_offer	3	2	4 Go

Comme précisé précédemment, ce dimensionnement ne contient qu'une seule offre de stockage ; il devra être complété de préférence par un deuxième site (avec le même dimensionnement), ou bien par une offre de stockage supplémentaire sur le site principal (en doublant les ressources allouées à la zone storage).

5.13.2 Stockage

Plus que tout autre, le calcul du dimensionnement du stockage dépend étroitement de la nature des archives qui doivent être conservées dans la solution logicielle.

Les drivers principaux de dimensionnement des différents emplacements de stockage sont les suivants :

- Répertoire « tmp » du composant `ingest-external` : ce répertoire doit pouvoir stocker les *SIP* en cours d'analyse antivirus avant leur dépôt dans workspace ; sa taille dépend donc de la taille maximale des *SIP* présents en entrée et du nombre d'ingest initiés en parallèle.
- Répertoire « data » du composant `workspace` : ce répertoire doit pouvoir stocker les données en cours de traitement (contenu décompressé des *SIP* en cours d'ingest, des objets binaires en cours de préservation, ainsi que les exports de données *DIP* en cours...) ; sa taille dépend donc de la taille maximale des *SIP* présents en entrée et du nombre d'ingest et de préservation simultanés (en attente ou en cours de traitement) ainsi que du volume et de la durée de rétention des *DIP* (par défault 7 jours, paramétrables dans la configuration du module `metadata`).
- Répertoire « tmp » du composant `worker` : ce répertoire doit pouvoir stocker les objets binaires en cours de traitement par le worker ; il s'agit généralement du produit `"capacité du worker" x "taille maximale d'un objet binaire"`.
- Répertoire « data » du composant `elasticsearch-data` : ce cluster stocke les métadonnées associées aux archives (*GOT* et *AU*) ainsi que les journaux d'opération. Pour ces éléments :
  - La taille et la quantité des *AU* et des *GOT* dépend des données entrées dans *VITAM* (facteur métier) ;
  - Le nombre d'opérations dépend de l'usage du système (et notamment de la granularité des *SIP* en entrée). En ordre de grandeur, le journal d'une opération d'ingest a une taille brute de 50 Ko ; le journal d'une opération d'update, 5 Ko (d'après des mesures effectuées sur des environnements de tests de la solution logicielle) ;
  - Au niveau global du cluster, le rapport entre la donnée brute (entrée dans elasticsearch) et la donnée persistée est le produit `"facteur de réplication" x 2` (le facteur 2 provient du champ `_source` qui contient le document original conservé par elasticsearch à côté des index) ;
  - La taille unitaire d'un répertoire « data » sur une instance se calcule ensuite en fonction du nombre de noeuds disponibles dans le cluster (l'hypothèse d'une répartition uniforme peut être retenue).
- Répertoire « data » du composant `mongod-data` : ce cluster stocke les métadonnées associées aux archives (*GOT*, *AU* et *LFC* associé) ainsi que les journaux d'opération. Pour ces éléments :
  - La taille et la quantité des AU et des GOT dépend du métier ;
  - Les *LFC* associés à une *AU* sont estimés à un peu moins de 5 Ko (d'après des mesures effectuées sur des environnements de tests de la solution logicielle) ;
  - Le nombre d'opérations dépend de l'usage du système (et notamment de la granularité des *SIP* en entrée). En ordre de grandeur, le journal d'une opération d'ingest a une taille moyenne brute de 50 Ko ; le journal d'une opération d'update ou audit, 5 Ko (d'après des mesures effectuées sur des environnements de tests de la solution logicielle) ;
  - Au niveau global du cluster, le rapport entre la donnée brute (entrée dans MongoDB) et la donnée persistée est le produit `"facteur de réplication" x "facteur d'expansion"`. Le facteur d'expansion dépend de la base de données impactée, et il est fonction du taux d'indexation et de sa capacité de compression. D'après des mesures effectuées sur des environnements de tests de la solution logicielle, ce facteur prend les valeurs suivantes :
    - 1,2 pour la base de données des métadonnées d'archive (*AU* & *GOT*)
    - 0,4 pour les journaux d'opération

    - La taille unitaire d'un répertoire « data » sur une instance se calcule ensuite en fonction du nombre de noeuds disponibles dans le cluster (l'hypothèse d'une répartition uniforme peut être retenue, MongoDB opérant un rééquilibrage progressif des shards).
- Répertoire « log » du composant storage : chaque écriture vers le stockage implique la création d'une entrée dans le journal des écritures du composant storage. Ainsi :
    - La taille de ce répertoire dépend du nombre d'éléments écrits, et notamment : *AU*, *GOT*, *BDO*, journaux d'opérations ;
    - Pour les journaux d'opération : chaque journal implique au moins deux écritures à cause de sa sécurisation ;
    - Chaque entrée du journal des écritures a une taille moyenne de 500 octets (d'après des mesures effectuées sur des environnements de tests de la solution logicielle).
- Espace de stockage du composant `offer` pour le stockage pérenne des données conservées dans *VITAM*, qui comprend notamment :
    - les *AU*, *GOT* et *BDO* ;
    - les journaux d'opération, de cycle de vie, d'écritures et d'accès ;
    - les autres données techniques persistées par Vitam (sécurisations des journaux, ATRs, rapports d'opérations…)

  Cette capacité doit être allouée selon le type de l'offre de cible :
    - Système de fichiers : Répertoire « data » du composant `offer` ;
    - Object store S3 ou Swift : Capacité de stockage dans l'object store « S3 » ou « Swift » cible ;
    - Archivage sur bandes magnétiques : Capacité de stockage sur bibliothèque de bandes
- Répertoire « tmp » du composant `offer` : ce répertoire doit pouvoir stocker les rapports liés à l'audit comparatif des offres ; sa taille dépends du nombre de fichiers présents dans les conteneurs à comparer. Pour un conteneur contenant plus de 1 million de fichiers, prévoir environs 300 Mo d'espace disque.
- Répertoire « data » du composant `mongod-offer` : chaque écriture dans une offre de stockage implique la journalisation de cette écriture dans l'archivelog d'écriture. Le nombre d'entrées est le nombre de données écrites via storage (cf. point précédent) ; la taille unitaire d'une entrée dans ce log est 260 octets (d'après des mesures effectuées sur des environnements de tests de la solution logicielle).
- Répertoire « data » du composant `elasticsearch-log` : ce *cluster* stocke les logs techniques issus de l'application. Il est assez difficile de donner un dimensionnement analytique réaliste de ce composant (trop d'éléments entrant en jeu). Pour donner un ordre de grandeur purement indicatif, pour un système en ingest pur (i.e. sans accès), il a été observé une moyenne de 20 Ko de log brut par triplet (*AU*, *GOT*, *BDO*) entré dans le système.

Note : Pour le stockage sur bandes magnétiques (offre froide), il est à noter que :

- L'offre froide de Vitam ne supporte actuellement PAS l'élimination physique des données (seule une élimination logique est réalisée). De ce fait, la capacité de stockage allouée sur bandes & cache disque doit contenir **toutes les versions des données écrites** (écritures + MAJ), et sur la **totalité de la durée de vie** de la solution déployée.
- En plus du stockage sur bande, le répertoire « data » du composant `offer` est également utilisé :
    - Comme espace « tampon » pour le stockage temporaire des objets à archiver. Il doit disposer de suffisamment d'espace pour contenir les données en cours d'archivage à l'instant T, le temps qu'elles soient écrites sur bande sur bande.
    - Comme espace « cache » pour le stockage des données à lire. Il doit disposer suffisamment d'espace pour contenir sur la **durée de vie** de la solution déployée :
        - l'ensemble des métadonnées et leur journaux de cycle de vie (environs 10 Ko par *AU* ou *GOT*, pour chaque version écrite)
        - l'ensemble des données techniques (journaux de sécurisation, ATRs, rapports…)
        - suffisamment d'espace pour la mise à disposition des binaires en cours de lecture (binaires en cours d'export de données DIP ou de transfert, en cours de préservation…)

  Pour une plateforme large de production, prévoir *à minima* 10 To de stockage disque pour le répertoire « data » du composant `offer`.

5.13.3 Réseau : inter-site

Un lien réseau *IP* doit exister entre les deux sites et respecter les flux décrits dans la matrice de flux externes (se reporter à *Matrice des flux* (page 105)).

Le routage niveau 3 est permis sur ce lien, par translation d'adresse, mais pas par translation de port (i.e. chaque serveur devant être exposé sur le site 2 au site 1 peut exposer une adresse *IP WAN* visible depuis le site 1 différente de son adresse *IP LAN* locale).

Concernant ce lien intersite, les éléments permettant son dimensionnement sont les suivants :

- La latence est peu critique (elle joue principalement sur la performance des batchs, et pas des accès utilisateurs ; l'optimisation des performances se fera dans ce cas par l'augmentation des pools de threads de storage et l'augmentation de la capacité des workers) ;
- Par contre, un débit adapté est requis ; dans cette version de *VITAM*, ce dernier peut se calculer à partir de la somme des débits d'ingest des *AU* + *GOT* + *BDO* + journaux.

5.13.4 Scalabilité

De manière générale, la consommation en ressources (CPU/RAM/réseau/stockage) de *VITAM* dépend de 3 grands cas d'utilisation :

- La quantité d'archives versées (*ingest*) : supporter plus d'ingest nécessite de renforcer les ressources disponibles pour les composants actifs lors d'un ingest : ingest-external, ingest-internal, processing, worker, workspace, logbook, metadata, storage, offer, elasticsearch-data, mongodb ;
- La quantité d'archives gérées (audit & pérennisation) : dans cette version de *VITAM*, les fonctions liées à ces deux domaines sont limitées ; par conséquent, la quantité de données gérées a uniquement une influence sur les dépôts de données : storage, offer, elasticsearch-data, mongodb ;
- La quantité d'archives consultées (*access*) : supporter plus de requêtes concurrentes nécessite de renforcer les ressources disponibles pour les composants actifs lors d'une consultation : access-external, access-internal, logbook, metadata, storage, offer, elasticsearch-data, mongodb.

5.14 Matrice des flux

Voir aussi :

La matrice complète des flux s'appuie sur les schémas présentés dans *la description de l'architecture technique* (page 49).

Les matrices de flux ne contiennent que les flux dans le sens de l'établissement des connexions. Les flux retours correspondant à des connexions établies doivent par conséquent également être autorisés.

En cas de problème avec les ports utilisés par Consul, il est recommandé de se reporter à la documentation officielle [53] ; en particulier, il pourra être requis d'ouvrir les ports *UDP* associés aux *Gossip* Consul dans certains cas.

5.14.1 Matrice des flux intra-site

Cette matrice des flux décrit les flux inter-zones pour la configuration par défaut des ports d'écoute des différents composants.

53. https://www.consul.io/docs/agent/options.html#ports

TABLEAU 6: Matrice des flu

Zone source	Zone cible	Protocole
Externe	Accès	https
Externe	Accès	https
Externe	Accès	https
Accès	Applicative	http
Accès	Applicative	http
Accès	Applicative	http
Accès	Applicative	http
Accès	Applicative	http
Applicative	Stockage	http(s)
Applicative	Données	tcp
Applicative	Données	http
Applicative	Données	tcp
Accès / Applicative / Stockage / Données	Administration	syslog/tcp
Accès / Applicative / Stockage / Données	Administration	tcp
Accès / Applicative / Stockage / Données / Administration	Accès / Applicative / Stockage / Données / Administration	tcp/udp
Accès / Applicative / Stockage / Données	Administration	http
Administration	Accès / Applicative / Stockage / Données	ssh
Administration	Accès / Applicative / Stockage / Données	http
Administration	Données	http
Administration	Données	http
Accès / Applicative / Stockage / Données	Administration	http(s)
Exploitation technique	Administration	http
Exploitation technique	Administration	http
Exploitation technique	Administration	http
Exploitation technique	Administration	http
Exploitation technique	Administration	http
Exploitation technique	Administration	http
Exploitation technique	Administration	ssh
Accès / Applicative / Stockage / Données	Administration	dns/udp
Administration	Accès / Applicative / Stockage	http

5.14.2 Matrice des flux inter-site

TABLEAU 7 – Matrice des flux inter-sites

Site source	Zone source	Site cible	Zone cible	Protocole	Port cible	Interface	Description
Site 1	Administration	Site 2	Administration	tcp	8300	default = admin (cf. consul.network)	Appels RPC consul
Site 2	Administration	Site 1	Administration	tcp	8300	default = admin (cf. consul.network)	Appels RPC consul
Site 1	Administration	Site 2	Administration	tcp/udp	8302	default = admin (cf. consul.network)	Gossip WAN Consul
Site 2	Administration	Site 1	Administration	tcp/udp	8302	default = admin (cf. consul.network)	Gossip WAN Consul
Site 1	Applicative	Site 2	Stockage	http(s)	9900	service	Accès à offer
Site 2	Applicative	Site 1	Stockage	http(s)	9900	service	Accès à offer (selon stratégie site 2)

CHAPITRE 6

Sécurité

6.1 Principes

Les principes de sécurité de la solution logicielle *VITAM* suivent les directives suivantes :

- Authentification et autorisation systématique des systèmes clients de *VITAM* basées sur une authentification *TLS* mutuelle utilisant des certificats (pour les composants de la couche accès) ;
- Validation systématique des entrées du système :
  - Détection et suppression de codes malveillants dans les archives déposées dans *VITAM* ;
  - Robustesse contre les failles du *Top Ten OWASP* pour toutes les interfaces *REST* ;
- Validation périodique des listes de *CRL* pour toutes les *CA trustées* par *VITAM* (non implémentée dans cette version de VITAM, cf. ci-dessous).

6.1.1 Principes de cloisonnement

Les principes de cloisonnement en zones, et notamment les implications en termes de communication entre ces zones ont été décrits dans *la section dédiée aux principes de déploiement* (page 39).

Avertissement : Le principe de cloisonnement des flux ne peut être mené que par une équipe d'infrastructure. L'implémentation du filtrage des flux inter-zones doit etre effectuée lors du déploiement de la solution *VITAM*, conformément à la matrice de flux, en annexe du document. Il est aussi indispensable de ne pas donner un accès internet aux machines dans les zones applicative, stockage, et donnée.

6.1.2 Principes de sécurisation des accès externes

Les services logiciels en contact direct avec les clients du *SAE* (i.e. les services `*-external`) implémentent les mesures de sécurité suivantes :

- Chiffrement du transport des données entre les applications externes et *VITAM* via HTTPS ; par défaut, la configuration suivante est appliquée :

- Protocoles exclus par défaut : `SSLv2`, `SSLv3`, `TLSv1.0`, `TLSv1.1`
- Ciphers exclus par défaut : `.*NULL.*`, `.*RC4.*`, `.*MD5.*`, `.*DES.*`, `.*DSS.*`, `TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA`, `TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA`, `TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA`, `TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA`, `TLS_DHE_RSA_WITH_AES_256_CBC_SHA`, `TLS_DHE_RSA_WITH_AES_128_CBC_SHA`

Note : Les ciphers recommandés sont : `TLS_ECDHE.*`, `TLS_DHE_RSA.*`

Fichier déployé :

```
# Use Bouncy Castle Provider when it is available
security.provider.9=org.bouncycastle.jce.provider.BouncyCastleProvider

# Override the default list of Centos 7 that disable Elliptic Curved Based Algorithms
jdk.tls.disabledAlgorithms="SSLv3, TLSv1, TLSv1.1, RC4, MD5withRSA, DH keySize < 768,
↪RSA keySize < 2048"
```

- Authentification par certificat x509 requise des applications externes (authentification *M2M*) basée sur une liste blanche de certificats valides :
  - Lors d'une connexion, la vérification synchrone confirme que le certificat proposé n'est pas expiré (*not before*, *not after*) et qu'il est validé par une Autorité de Certification connue (liste des *CA* portée par un fichier *truststore*) ;
  - Avant de valider tout appel d'*API*, l'applicatif vérifie que le certificat proposé est bien présent dans le référentiel d'authentification des certificats valides (un des référentiels métier portés par la base des méta-données).

Prudence : La révocation des certificats se fait par leur suppression dans les différents magasins et référentiels. Se reporter au *DEX* pour plus d'informations.

- Filtrage exhaustif des données et requêtes entrant dans le système basé sur :
  - Un *WAF* applicatif permettant le filtrage d'entrées pouvant être une menace pour le système (intégration de la bibliothèque ESAPI [54] dans les composants `*-external` protégeant notamment contre les attaques de type XSS) ;
  - Support de l'utilisation d'un ou plusieurs antivirus (configurables et extensibles) dans le composant d'entrée (`ingest-external`) permettant de valider l'innocuité des données entrantes.

Note : Dans cette version du système, le paramétrage de l'antivirus est supporté lors de l'installation, mais pas le paramétrage d'ESAPI (notamment les filtres appliqués).

6.1.3 Principes de sécurisation des communications internes au système

Le secret de plateforme permet de se protéger contre des erreurs de manipulation et de configuration en séparant les environnements de manière logique (secret partagé par l'ensemble de la plateforme mais différent entre plateformes). Ce secret (chaîne de caractères) est positionné dans la configuration des composants lors de l'installation de la solution logicielle *VITAM*.

Dans chaque requête, les deux headers suivants sont positionnés :

54. https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API

- `X-Request-Timestamp` : il contient le timestamp de la requête sous forme epoch (secondes depuis 1970)
- `X-Platform-ID` : il contient la valeur suivante : `SHA256("<methode>;<URL>;<Valeur du header X-Request-Timestamp>;<Secret partagé de plateforme>")`

Du côté du composant cible de la requête, le contrôle est alors le suivant :

- Existence des deux *headers* précédents ; Dans le cas contraire, la requête est refusée.
- Vérification que *timestamp* envoyé est distant de l'heure actuelle sur le serveur requêté de moins de x secondes (`| Timestamp - temps local | < x s`). Si la différence de temps est supérieure au seuil acceptable (10s par défaut), alors des erreurs sont tracées dans les logs et des alertes sont remontées dans le dashboard Kibana « Alertes de sécurité ». Au delà d'un seuil critique (60s par défaut), la requête est refusée.
- Validation du hash transmis via la réalisation du même calcul sur le serveur cible et de la comparaison des résultats ; En cas d'échec de validation, la requête est refusée.

Note : Les *headers* et le *body* de la requête ne sont pas inclus dans le calcul du `X-Platform-ID` pour des raisons de performances.

6.1.4 Principes de sécurisation des bases de données

Les bases de données sont sécurisées via un cloisonnement physique et/ou logique des différentes bases de données qui les constituent.

6.1.4.1 MongoDB

Dans le cas de MongoDB, le cloisonnement est logique. Chaque service hébergeant des données dans MongoDB se voit attribuer une base et un utilisateur dédié. Cet utilisateur a uniquement les droits de lecture / écriture dans les collections de cette base de données, mais ne peut notamment pas modifier la structure des collections de sa base de données ni accéder aux collections d'une autre base de données.

Un utilisateur technique *root* est également créé pour les besoins de l'installation et de la configuration de MongoDB.

Chaque base de données ne doit être accédée que par les instances d'un seul service (ex : le service logbook est le seul à accéder à la base de données logbook).

Enfin, l'accès anonyme à MongoDB est désactivé, et les utilisateurs sont authentifiés par le couple utilisateur / mot de passe.

6.1.4.2 Elasticsearch

Dans le cas d'Elasticsearch, le cloisonnement est principalement physique, dans le sens où le *cluster* hébergeant les données métier est disjoint du *cluster* hébergeant les données techniques.

Prudence : L'accès au cluster Elasticsearch est anonyme sans authentification requise ;

6.1.5 Principes de sécurisation des secrets de déploiement

Les secrets de l'intégralité de la solution *VITAM* déployée sont tous présents sur le serveur de déploiement ; par conséquent, ils doivent y être stockés de manière sécurisée, avec les principes suivants :

- Les mots de passe et *tokens* utilisés par ansible doivent être stockés dans des fichiers d'inventaire chiffrés par ansible-vault ;

- Les clés privées des certificats doivent être protégées par des mots de passe complexes ; ces derniers doivent suivre la règle précédente.

6.2 Liste des secrets

Les secrets nécessaires au bon déploiement de *VITAM* sont les suivants :

- Certificat ou mot de passe de connexion *SSH* à un compte *sudoer* sur les serveurs cibles (pour le déploiement) ;
- Certificats x509 serveur (comprenant la clé privée) pour les modules de la zone d'accès (services `*-external`) et pour le module `storage`, ainsi que les *CA* (finales et intermédiaires) ;
- Certificats x509 client d'horodatage, pour les modules appliquant l'horodatage sécurisé, ainsi que les *CA* (finales et intermédiaires) ;
- Certificats x509 client pour les clients du *SAE* (ex. : les applications métier, le service `ihm-demo`), ainsi que les *CA* (finales et intermédiaires) ;
- *CA* (finales et intermédiaires) éventuels des offres de stockage utilisées (ex. : CA d'une offre de stockage objet swift ou s3).

Note : Ces certificats x509 seront déployés dans des keystores java[55] en tant qu'éléments de configuration de ces services (se rapporter au *DIN* pour plus d'informations).

Les secrets définis lors de l'installation de *VITAM* sont les suivants :

- Mots de passe des keystores ;
- Mots de passe des administrateurs fonctionnels de l'application *VITAM* ;
- Mots de passe d'administration de base de données MongoDB ;
- Mots de passe des comptes d'accès aux bases de données MongoDB.

Le détail de l'usage des certificats pour le déploiement est donné dans le *DIN*.

6.3 Certificats

Les magasins de certificats utilisés par le système *VITAM* sont les suivants :

55. https://docs.oracle.com/cd/E19509-01/820-3503/ggffo/index.html

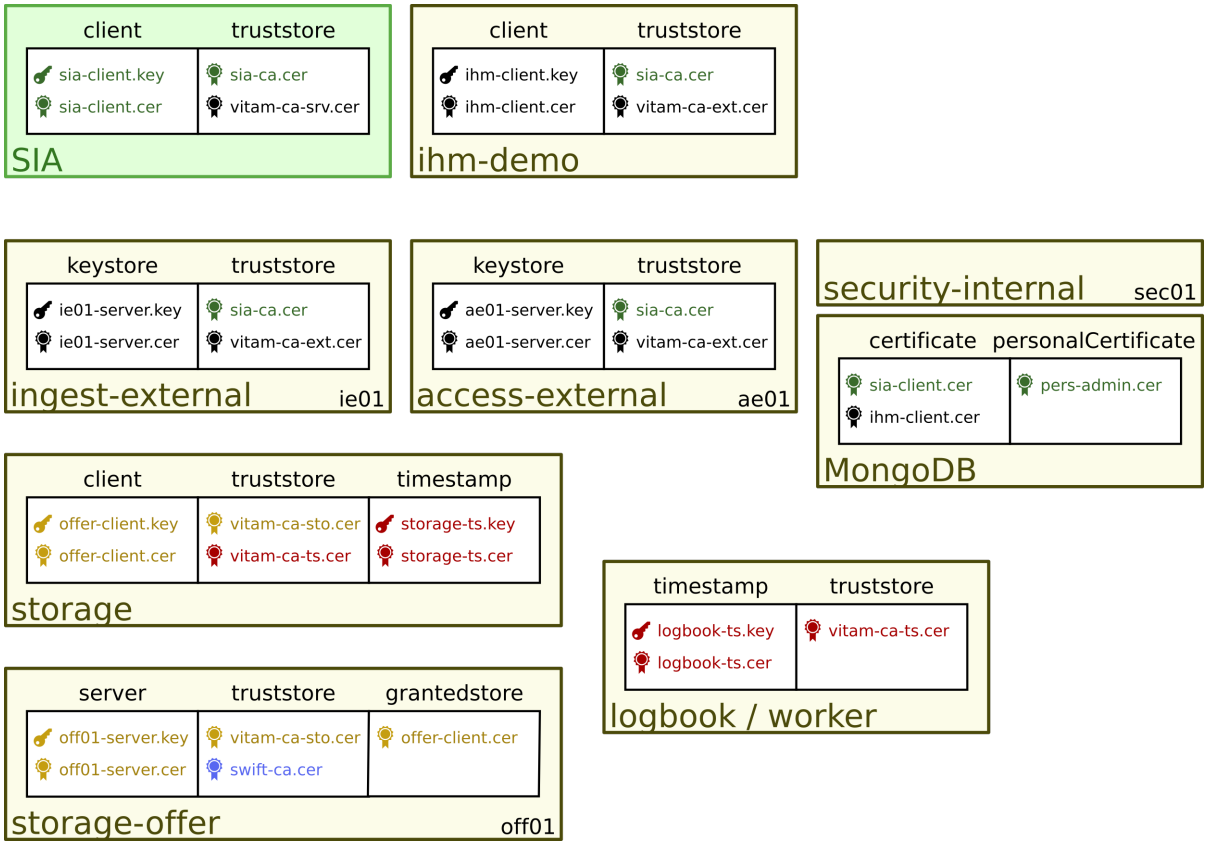


FIG. 1 – Vue d'ensemble des magasins de certificats déployés dans un système *VITAM* ; chaque couleur correspond à une chaîne de certification potentiellement disjointe des autres.

6.4 SELinux

Vitam peut fonctionner avec SELinux en mode `enforcing` Le support de SELinux a été pensé pour faire fonctionner tous les services utilisés par Vitam dans des domaines confinés (mode de sécurité le plus élevé). Le fonctionnement des services Vitam dans le domaine non confiné n'a jamais été testé.

6.5 Documentation de sécurité

Le développement de la solution logicielle *VITAM* suit une méthodologie d'analyse des risques de sécurité basée sur l'adaptation de la méthode *EBIOS* aux projets agiles. Le document d'analyse de risque associé ainsi que le dossier d'homologation de la solution logicielle ne sont pas publiquement disponibles pour des raisons de sécurité ; en revanche, ils peuvent être communiqués sur simple demande auprès de la direction du programme *VITAM*.

CHAPITRE 7

Architecture détaillée

Les sections qui suivent donnent une description plus fine de l'architecture interne des services VITAM.

7.1 Access

7.1.1 Généralités

Le rôle d'*access* est de :

- Rechercher les Unités d'archives via des mots-clés.
- Afficher la liste des résultats par rapport aux critères de recherche renseignés.
- Consulter les détails d'une Unité d'archive.
- Modifier les métadonnées d'une Unité d'archive.

7.1.2 Architecture Technique

7.1.2.1 Introduction

7.1.2.1.1 Présentation

Parent package : **fr.gouv.vitam**
Package proposition : **fr.gouv.vitam.access**

7.1.2.1.2 Itération 4

5 sous-modules pour le module access. Dans access (parent).

- vitam-access-common : Classes contenant les exceptions, les objets réponses.

- vitam-access-api : Interfaces pour les api publiques.
- vitam-access-client : Classes communes pour les clients.
- vitam-access-core : Classes impléméntant les API publiques.
- vitam-access-rest : module pour les api REST.

7.1.2.1.3 Modules - packages

access

/access-common
    fr.gouv.vitam.access.common.exception
    fr.gouv.vitam.access.common.model
    fr.gouv.vitam.access.config
    fr.gouv.vitam.common.model
/access-api
    fr.gouv.vitam.access.api
    fr.gouv.vitam.api.exception
/access-client
    fr.gouv.vitam.access.client
/access-core
    fr.gouv.vitam.core
        /access-rest
    fr.gouv.vitam.access.config
    fr.gouv.vitam.access.model
    fr.gouv.vitam.acces.rest

7.1.2.2 Access-api

7.1.2.2.1 Présentation

- Package parent :
  fr.gouv.vitam.access
- Proposition de package
  fr.gouv.vitam.access.api
  fr.gouv.vitam.access.exception

term *API REST* appelées par le client access interne.

Dans le package *fr.gouv.vitam.access.core* l'interface utilisée :
    AccessModule pour les méthodes implementées par le module (access-core)
Dans le package *fr.gouv.vitam.access.rest* l'interface utilisée :
    AccessResource pour les méthodes implémentées par le controlleur REST (access-rest)

7.1.2.3 Access-client

Ce module est utilisé par le module ihm-demo(package fr.gouv.vitam.ihmdemo.core).

7.1.2.4 Utilisation

- La factory : Afin de récupérer le client-access , une factory a été mise en place.

```
// Récupération du client
 final AccessClient client = AccessClientFactory.getInstance().
↪getAccessOperationClient();
```

- Le Mock Si les paramètres de productions sont introuvables, le client passe en mode Mock par défaut. Il est possible de récupérer directement le mock :

```
// Changer la configuration du Factory client
AccessClientFactory.setConfiguration(AccessClientType.MOCK);
// Récupération explicite du client mock
final AccessClient client = AccessClientFactory.getInstance().
↪getAccessOperationClient();
```

- Pour instancier son client en mode Production :

```
// Changer la configuration du Factory
 AccessClientFactory.setConfiguration(AccessClientType.PRODUCTION);
// Récupération explicite du client
AccessClient client = AccessClientFactory.getInstance().getAccessOperationClient();
```

7.1.2.5 Le client

Le client propose actuellement plusieurs méthodes :

> selectUnits(String dslQuery) ; selectUnitbyId(String sqlQuery, String id) ; updateUnitbyId(String updateQuery, String unitId) ; selectObjectbyId(String selectObjectQuery, String objectId) ; getObjectAsInputStream(String selectObjectQuery, String objectGroupId, String usage, int version) ;

Paramètre de la fonction : String ds, String Identification

Le client récupère une réponse au format Json ou au format InputStream.

7.1.2.6 Access-common

7.1.2.6.1 Présentation

Package parent : **fr.gouv.vitam.access**
Proposition de package : **fr.gouv.vitam.access.common**

Module utilisé pour les objets communs :

- modeles reponse
- exceptions
- params
- configuration
- autres…

7.1.2.7 Access-core

7.1.2.7.1 Présentation

Ce module permet d'implémenter les *API* publiques du module access-api

7.1.2.7.2 Packages :

fr.gouv.vitam.access.core

Classes utilisées

AccessModuleImpl

Classe qui dialogue avec le module métadata. Elle transmet au métadata client d'une requête dsl.

```
public JsonNode selectUnit(String selectRequest){
  ...
   // Récupération du client métadata
    metaDataClientFactory = new MetaDataClientFactory();
    metaDataClient = metaDataClientFactory.create(accessConfiguration.
↪getUrlMetaData());
  ...
// appel du client métadata
  try {
          jsonNode = metaDataClient.selectUnits(
          accessModuleBean != null ? accessModuleBean.getRequestDsl() : "");

  }
  ...
 }
```

7.1.2.7.2.1 Récupération d'un objet spécifique

Il faut utiliser la méthode getOneObjectFromObjectGroup() pour récupérer un objet binaire.

Exemple :

```
try {
  InputStream objectData = getOneObjectFromObjectGroup("idObjectGroup",␣
↪queryAsJsonNode, "BinaryMaster", 0, "0");
} catch (MetaDataNotFoundException exc) {
  // Handle objectGroup not found
} catch (StorageNotFoundException exc) {
  // Object with given qualifier and version was not found in storage offer
} catch (InvalidParseOperationException exc) {
  // Handle badly formatted json query
} catch (AccessExecutionException exc) {
  // Technical exception that should not happen. The message give details on the error
}
```

7.1.2.8 Access-rest

7.1.2.8.1 Présentation

API REST appelées par le client access interne. Il y a un controle des paramètres (SanityChecker.checkJsonAll) transmis avec ESAPI.

7.1.2.8.2 Packages :

fr.gouv.vitam.access.external.config : contient les paramètres de configurations du service web d'application.

fr.gouv.vitam.access.external.model : classes métiers, classes implémentant le pattern DTO. . . .

fr.gouv.vitam.access.external.rest : classes de lancement du serveur d'application et du controlleur REST.

7.1.2.8.3 fr.gouv.vitam.access.external.rest

7.1.2.8.3.1 Rest API

https://vitam/access-external/v1/units
https://vitam/access-external/v1/units/unit_id
https://vitam/access-external/v1/objects
https://vitam/access-external/v1/units/unit_id/objects
https://vitam/access-external/v1/accessionregisters
https://vitam/access-external/v1/accessionregisters/document_id
https://vitam/access-external/v1/accessionregisters/document_id/accessionregisterdetail
https://vitam/access-external/v1/logbookoperations
https://vitam/access-external/v1/logbookoperations/operation_id
https://vitam/access-external/v1/logbookunitlifecycles/lifecycle_id
https://vitam/access-external/v1/logbookobjectslifecycles/lifecycle_id
https://vitam/admin-external/v1/collection_id
https://vitam/admin-external/v1/collection_id/document_id

7.1.2.9 -AccessApplication.java

classe de démarrage du serveur d'application.

```
// démarrage
public static void main(String[] args) {
    try {
        startApplication(args);
        server.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Dans le startApplication on effectue le start de VitamServer. Le *join* permet de lancer les tests unitaires et d'arreter le serveur. Dans le fichier de configuration, le paramètre `jettyConfig` est à paramétrer avec le nom du fichier de configuration de jetty.

7.1.2.10 -AccessResourceImpl.java

classe controlleur REST La classe contient actuellement 9 méthodes :

1. getUnits()

   NB : the post X-Http-Method-Override header

```
@POST
@Path("/units")
public Response getUnits(String requestDsl,
@HeaderParam("X-Http-Method-Override") String xhttpOverride) {
...
try {
if (xhttpOverride != null && "GET".equalsIgnoreCase(xhttpOverride)) {
    queryJson = JsonHandler.getFromString(requestDsl);
    result = accessModule.selectUnit(queryJson.toString());
} else {
    throw new AccessExecutionException("There is no 'X-Http-Method-Override:GET' as a␣
↪header");
}
....
```

2. createOrSelectUnits()

   Récupère la liste des units avec la filtre

   NB : La méthode HTTP GET n'est pas compatible, on utilisera une méthode HTTP POST dont l'entête contiendra « X-HTTP-Method-GET »

   méthode createOrSelectUnits() va appeler méthode getUnits()

```
    @POST
@Path("/units")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response createOrSelectUnits(JsonNode queryJson,
    @HeaderParam(GlobalDataRest.X_HTTP_METHOD_OVERRIDE) String xhttpOverride)
 ...
```

3. getUnitById()

   récupère un unit avec son id NB : the post X-Http-Method-Override header

```
@POST
@Path("/units/{id_unit}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response getUnitById(String queryDsl,
    @HeaderParam(GlobalDataRest.X_HTTP_METHOD_OVERRIDE) String xhttpOverride,
    @PathParam("id_unit") String id_unit) {
...
```

4. createOrSelectUnitById()

Note : La méthode HTTP GET n'est pas compatible, on utilisera une méthode HTTP POST dont l'entête contiendra « X-HTTP-Method-GET »

méthode createOrSelectUnitById() va appeler méthode getUnitById()

```
      @POST
@Path("/units/{idu}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response createOrSelectUnitById(JsonNode queryJson,
    @HeaderParam(GlobalDataRest.X_HTTP_METHOD_OVERRIDE) String xhttpOverride,
    @PathParam("idu") String idUnit) {
 ...
```

5. updateUnitById()

   mise à jour d'un unit par son id avec une requête json

```
@PUT
  @Path("/units/{id_unit}")
  @Consumes(MediaType.APPLICATION_JSON)
  @Produces(MediaType.APPLICATION_JSON)
  public Response updateUnitById(String queryDsl,
                                 @PathParam("id_unit") String id_unit) {
  ...
```

6. getObjectGroup()

   récupérer une groupe d'objet avec la filtre

Note : the post X-Http-Method-Override header

```
    @GET
@Path("/objects/{ido}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response getObjectGroup(@PathParam("ido") String idObjectGroup, JsonNode␣
↪queryJson)
 ...
```

7. getObjectGroupPost()

Note : La méthode HTTP GET n'est pas compatible, on utilisera une méthode HTTP POST dont l'entête contiendra « X-HTTP-Method-GET »

méthode getObjectGroupPost() va appeler méthode getObjectGroup()

```
    @POST
@Path("/objects/{ido}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response getObjectGroupPost(@Context HttpHeaders headers,
    @PathParam("ido") String idObjectGroup, JsonNode queryJson)
 ...
```

8. getObject()

   récupérer le group d'objet par un unit

Note : the post X-Http-Method-Override header

```
    @GET
@Path("/units/{ido}/objects")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_OCTET_STREAM)
public void getObject(@Context HttpHeaders headers, @PathParam("ido") String␣
↪idObjectGroup,
    JsonNode query, @Suspended final AsyncResponse asyncResponse) {
 ...
```

9. getObjectPost()

Note : La méthode HTTP GET n'est pas compatible, on utilisera une méthode HTTP POST dont l'entête contiendra « X-HTTP-Method-GET »

méthode getObjectPost() va appeler méthode getObject()

```
    @POST
@Path("/units/{ido}/objects")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_OCTET_STREAM)
public void getObjectPost(@Context HttpHeaders headers, @PathParam("ido") String␣
↪idObjectGroup,
    JsonNode query, @Suspended final AsyncResponse asyncResponse) {
 ...
```

7.1.2.11 -LogbookExternalResourceImpl.java

classe controlleur REST

la classe contient actuellement 6 méthodes :

1. getOperationById()

   récupère l'opération avec son id NB : the post X-Http-Method-Override header

```
    @GET
@Path("/logbookoperations/{id_op}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response getOperationById(@PathParam("id_op") String operationId) {
 ...
```

2. selectOperationByPost()

Note : La méthode HTTP GET n'est pas compatible, on utilisera une méthode HTTP POST dont l'entête contiendra « X-HTTP-Method-GET »

méthode selectOperationByPost() va appeler méthode getOperationById()

```
    @POST
@Path("/operations/{id_op}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response selectOperationByPost(@PathParam("id_op") String operationId,
    @HeaderParam("X-HTTP-Method-Override") String xhttpOverride)
 ...
```

3. selectOperation()

   récupérer tous les journaux de l'opéraion NB : the post X-Http-Method-Override header

```
    @GET
@Path("/operations")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response selectOperation(JsonNode query)
 ...
```

4. selectOperationWithPostOverride()

Note : La méthode HTTP GET n'est pas compatible, on utilisera une méthode HTTP POST dont l'entête contiendra « X-HTTP-Method-GET »

méthode selectOperationWithPostOverride() va appeler méthode selectOperation()

```
    @POST
@Path("/operations")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response selectOperationWithPostOverride(JsonNode query,
    @HeaderParam("X-HTTP-Method-Override") String xhttpOverride)
 ...
```

5. getUnitLifeCycle()

   récupère le journal sur le cycle de vie d'un unit avec son id

```
    @GET
@Path("/logbookunitlifecycles/{id_lc}")
@Produces(MediaType.APPLICATION_JSON)
public Response getUnitLifeCycle(@PathParam("id_lc") String unitLifeCycleId)
 ...
```

6. getObjectGroupLifeCycle()

   récupère le journal sur le cycle de vie d'un groupe d'objet avec son id

```
    @GET
@Path("/logbookobjectslifecycles/{id_lc}")
@Produces(MediaType.APPLICATION_JSON)
public Response getObjectGroupLifeCycle(@PathParam("id_lc") String␣
↪objectGroupLifeCycleId)
 ...
```

7.1.2.12 -AdminManagementExternalResourceImpl.java

classe controlleur REST

la classe contient actuellement 10 méthodes :

1. checkDocument()

   vérifier le format ou la règle

```
    @Path("/{collection}")
@PUT
@Consumes(MediaType.APPLICATION_OCTET_STREAM)
@Produces(MediaType.APPLICATION_JSON)
public Response checkDocument(@PathParam("collection") String collection, InputStream␣
↪document) {
 ...
```

2. importDocument()

   Importer le fichier du format ou de la règle

```
    @Path("/{collection}")
@POST
@Consumes(MediaType.APPLICATION_OCTET_STREAM)
@Produces(MediaType.APPLICATION_JSON)
public Response importDocument(@PathParam("collection") String collection,␣
↪InputStream document) {
 ...
```

3. importProfileFile()

   Importer un fichier au format xsd ou rng et l'attacher à un profile métadata déjà existant.

```
    @Path("/{collection}/{id}")
@PUT
@Consumes(MediaType.APPLICATION_OCTET_STREAM)
@Produces(MediaType.APPLICATION_JSON)
public Response importProfileFile(@Context UriInfo uriInfo, @PathParam("collection")␣
↪String collection, @PathParam("id") String profileMetadataId,
    InputStream profileFile) {
 ...
```

4. downloadProfileFileOrTraceabilityFile()

   Télécharger un fichier d'un profile métadata existant au format xsd ou rng Ou télécharger un fichier d'opération de traçabilité

```
    @GET
@Path("/{collection}/{id}")
@Produces(MediaType.APPLICATION_OCTET_STREAM)
public void downloadProfileFileOrTraceabilityFile(@PathParam("collection") String␣
↪collection, @PathParam("id") String profileMetadataId,
    @Suspended final AsyncResponse asyncResponse) {
 ...
```

5. findDocuments()

Récupérer le format, la règle, le contrat (entrée ou accès), le profile.

```
    @Path("/{collection}")
@GET
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response findDocuments(@PathParam("collection") String collection, JsonNode␣
↪select) {
 ...
```

6. createOrfindDocuments()

   Si la valeur de xhttpOverride est rensigné et égale à GET alors, c'est un find, donc redirection vers la méthode findDocuments ci-dessus. Sinon, c'est créate. Cette méthode est utilisé pour créer des profiles au format json. On peut noter que dans ce cas de figure, ça ressemble à la méthode importDocument, sauf que le Consumes qui change.

```
    @Path("/{collection}")
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response createOrfindDocuments(@PathParam("collection") String collection,␣
↪JsonNode select, @HeaderParam(GlobalDataRest.X_HTTP_METHOD_OVERRIDE) String␣
↪xhttpOverride) {
 ...
```

7. findDocumentByID()

   En utilisant la méthode POST avec un paramètre xhttpOverride, ce méthode permets de récupérer avec un id en entrée, le format, la règle, les contrats (accès, entrée), les profiles.

```
    @Path("/{collection}/{id_document}")
@POST
@Produces(MediaType.APPLICATION_JSON)
public Response findDocumentByID(@PathParam("collection") String collection,␣
↪@PathParam("id_document") String documentId, @HeaderParam(GlobalDataRest.X_HTTP_
↪METHOD_OVERRIDE) String xhttpOverride) {
 ...
```

8. findDocumentByID()

   En utilisant la méthode GET, ce méthode permets derécupérer avec un id en entrée, le format, la règle, les contrats (accès, entrée), les profiles.

```
    @Path("/{collection}/{id_document}")
@GET
@Produces(MediaType.APPLICATION_JSON)
public Response findDocumentByID(@PathParam("collection") String collection,
    @PathParam("id_document") String documentId) {
 ...
```

9. updateAccessContract()

   Mise à jour du contrat d'accès

```
@PUT
  @Path("/accesscontract")
  @Consumes(MediaType.APPLICATION_JSON)
  @Produces(MediaType.APPLICATION_JSON)
   public Response updateAccessContract(JsonNode queryDsl) {
   ...
```

10. updateIngestContract()

    Mise à jour du contrat d'entrée

```
@PUT
  @Path("/contract")
  @Consumes(MediaType.APPLICATION_JSON)
  @Produces(MediaType.APPLICATION_JSON)
   public Response updateIngestContract(JsonNode queryDsl) {
   ...
```

7.2 Batch-report

7.2.1 Généralités

Le rôle de *batch report* est de fournir une *API* permettant de contruire des rapports.

7.2.2 Architecture Technique

7.2.2.1 Introduction

7.2.2.1.1 Présentation

Parent package : **fr.gouv.vitam**
Package proposition : **fr.gouv.vitam.batch.report**

7.2.2.1.2 Découpage du code

Il y a 3 sous modules :

- batch-report-client, dont le package est **fr.gouv.vitam.batch.report.client**, contient le client REST exposé vers l'exterieur permettant d'appeler l'API de ce module.
- batch-report-common, dont les packages sont **fr.gouv.vitam.batch.report.model** et **fr.gouv.vitam.batch.report.exception**, contiennent les modèles et les exceptions liés à ce module.
- batch-report-rest, dont le package est **fr.gouv.vitam.batch.report.rest** contient l'API REST.

7.2.2.2 batch-report-client

Ce module contient le client permettant d'appeler l'API.

7.2.2.3 batch-report-common

Ce module contient l'ensemble de modèles et exception nécessaire au *batch report*.

7.2.2.4 Acbatch-report-rest

Ce module contient les ressources exposant l'"*API* du *batch report*.

7.3 Collect

7.3.1 Généralités

Le rôle de *collect* est de :

- Intialiser des transacrions.
- Ajouter des Unités d'archives à la transaction.
- Ajouter un object group a une unité d'archive.
- Ajouter un binaire a un object group.
- Fermer la transaction.
- Produire un SIP et l'envoyer a Vitam (Ingest).

7.3.2 Architecture Technique

7.3.2.1 Introduction

7.3.2.1.1 Présentation

Parent package : **fr.gouv.vitam**
Package proposition : **fr.gouv.vitam.collect**

7.3.2.1.2 Itération 4

2 sous-modules pour le module collect. Dans collect (parent).

- vitam-collect-client : Classes communes pour les clients.
- vitam-collect-rest : module pour les api REST.

7.3.2.1.3 Modules - packages

collect

/collect-client
    fr.gouv.vitam.collect.external.client
        /collect-rest

fr.gouv.vitam.collect.internal.dto
fr.gouv.vitam.collect.internal.exception
fr.gouv.vitam.collect.internal.helpers
fr.gouv.vitam.collect.internal.model
fr.gouv.vitam.collect.internal.repository
fr.gouv.vitam.collect.internal.resource
fr.gouv.vitam.collect.internal.server
fr.gouv.vitam.collect.internal.service

7.3.2.2 collect-client

Ce module est utilisé par les test d'integration(package fr.gouv.vitam.collect).

7.3.2.3 Utilisation

- La factory : Afin de récupérer le client-collect , une factory a été mise en place.

```
// Récupération du client
 final CollectClient collectClient = CollectClientFactory.getInstance().getClient();
```

- Le Mock Si les paramètres de productions sont introuvables, le client passe en mode Mock par défaut. Il est possible de récupérer directement le mock :
- Pour instancier son client en mode Production :

```
// Changer la configuration du Factory
 CollectClientFactory.changeMode(COLLECT_CLIENT_CONF);
```

7.3.2.4 Le client

Le client propose actuellement plusieurs méthodes :

initProject(VitamContext vitamContext, ProjectDto projectDto) updateProject(VitamContext vitamContext, ProjectDto projectDto) getProjectById(VitamContext vitamContext, String projectId) getTransactionByProjectId(VitamContext vitamContext, String projectId) getTransactionById(VitamContext vitamContext, String transactionId) getProjects(VitamContext vitamContext) deleteTransactionById(VitamContext vitamContext, String transactionId) deleteProjectById(VitamContext vitamContext, String projectId) getUnitById(VitamContext vitamContext, String unitId) getUnitsByTransaction(VitamContext vitamContext, String transactionId, JsonNode query) getObjectById(VitamContext vitamContext, String gotId) initTransaction(VitamContext vitamContext, TransactionDto transactionDto, String projectId) uploadArchiveUnit(VitamContext vitamContext, JsonNode unitJsonNode, String transactionId) addObjectGroup(VitamContext vitamContext, String unitId, Integer version, JsonNode objectJsonNode, String usage) addBinary(VitamContext vitamContext, String unitId, Integer version, InputStream inputStreamUploaded, String usage) closeTransaction(VitamContext vitamContext, String transactionId) ingest(VitamContext vitamContext, String transactionId) abortTransaction(VitamContext vitamContext, String transactionId) reopenTransaction(VitamContext vitamContext, String transactionId) uploadProjectZip(VitamContext vitamContext, String transactionId, InputStream inputStreamUploaded) getObjectStreamByUnitId(VitamContext vitamContext, String unitId, String usage, int version) searchProject(VitamContext vitamContext, CriteriaProjectDto criteria) updateTransaction(VitamContext vitamContext, TransactionDto transactionDto) updateUnits(VitamContext vitamContext, String transactionId, InputStream is) selectUnitsWithInheritedRules(VitamContext vitamContext, JsonNode selectQuery)

7.3.2.5 collect-rest

7.3.2.5.1 Présentation

A faire

7.3.2.5.2 Packages :

fr.gouv.vitam.collect.dto : contient les objets entrants et sortants du service web d'application.

fr.gouv.vitam.collect.exception : classes d'exceptions gérées par le module collecte.

fr.gouv.vitam.collect.helpers : classes gérant la structure des objets en interne du module collecte.

fr.gouv.vitam.collect.model : classes interne du module collecte.

fr.gouv.vitam.collect.repository : classes pour mapper les objets de la base de données.

fr.gouv.vitam.collect.resource : classe du controlleur REST qui contient les endpoints.

fr.gouv.vitam.collect.server : classes de lancement du serveur d'application.

fr.gouv.vitam.collect.service : classes fonctionnelles.

7.3.2.5.3 fr.gouv.vitam.collect.resource

7.3.2.5.3.1 Rest API

https://vitam/collect/v1/transactions
https://vitam/collect/v1/units
https://vitam/collect/v1/objects/{usage}/{version}
https://vitam/collect/v1/objects/{usage}/{version}/binary
https://vitam/collect/v1/transactions/close
https://vitam/collect/v1/transactions/send
https://vitam/collect/v1/projects/{projectId}/binary
https://vitam/collect/v1/projects/{projectId}/units

7.3.2.6 -TransactionResource.java

classe controlleur REST La classe contient actuellement 6 méthodes :

1. initTransaction()

```
 @Path("/transactions")
  @POST
  @Consumes(APPLICATION_JSON)
  @Produces(MediaType.APPLICATION_JSON)
  @Secured(permission =  TRANSACTION_CREATE, description = "Créer une transaction")
  public Response initTransaction(TransactionDto transactionDto) {

....
```

2. uploadArchiveUnit()

Ajouter une unité archivistique

```
    @Path("/transactions/{transactionId}/units")
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@Secured(permission =  TRANSACTION_UNIT_CREATE, description = "Créer une␣
↪unité archivistique")
public Response uploadArchiveUnit(@PathParam("transactionId") String␣
↪transactionId, JsonNode unitJsonNode) {

 ...
```

3. uploadObjectGroup()

   ajouter un object group a une unité archivistique NB : the post X-Http-Method-Override header

```
@Path("/units/{unitId}/objects/{usage}/{version}")
  @POST
  @Consumes(MediaType.APPLICATION_JSON)
  @Produces(MediaType.APPLICATION_JSON)
  @Secured(permission =  TRANSACTION_OBJECT_UPSERT, description = "ajouter ou␣
↪modifier un objet group")
  public Response uploadObjectGroup(@PathParam("unitId") String unitId,
                                    @PathParam("usage") String usageString,
                                    @PathParam("version") Integer version,
                                    ObjectDto objectDto) {
...
```

4. upload()

   méthode pour uploader un binaire

```
    @Path("/units/{unitId}/objects/{usage}/{version}/binary")
 @POST
 @Consumes(MediaType.APPLICATION_OCTET_STREAM)
 @Produces(MediaType.APPLICATION_JSON)
 @Secured(permission =  TRANSACTION_BINARY_UPSERT, description = "ajouter ou modifier␣
↪un binaire")
 public Response upload(@PathParam("unitId") String unitId,
                        @PathParam("usage") String usageString,
                        @PathParam("version") Integer version,
                        InputStream uploadedInputStream) throws CollectException {
...
```

5. closeTransaction()

   fermeture de la transaction

```
@Path("/transactions/{transactionId}/close")
  @POST
  @Consumes(APPLICATION_JSON)
  @Produces(MediaType.APPLICATION_JSON)
  @Secured(permission =  TRANSACTION_CLOSE, description = "Fermer une transaction")
  public Response closeTransaction(@PathParam("transactionId") String transactionId) {
  ...
```

6. generateAndSendSip()

   génerer un SIP et l'envoyer a Vitam

```
    @Path("/transactions/{transactionId}/send")
@POST
@Consumes(APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@Secured(permission =  TRANSACTION_SEND, description = "Envoyer une transaction")
public Response generateAndSendSip(@PathParam("transactionId") String transactionId) {
 ...
```

7.4 Common

7.4.1 Architecture Fonctionnelle

7.4.1.1 Introduction

7.4.1.1.1 But de cette documentation

L'objectif de cette documentation est d'expliquer l'architecture fonctionnelle de ce module.

7.4.1.1.2 GUID

Cf chapitre dédié

7.4.1.1.3 ServerIdentity et Logger

Ces 2 packages sont liés car ServerIdentity fournit des informations utiles au Logger.

Le Logger enverra un certain nombre d'information vers le log centralisé, via un filtre issu de VitamLoggerHelper.

Cette centralisation permettra notamment d'avoir des informations analysées par l'outil d'administration (par défaut, *ELK*).

L'ensemble des logs seront centralisés mais tous n'iront pas dans la partie « analytique » des logs.

7.4.1.2 GUID

Le sujet porte notamment sur les **GUID**.

7.4.1.2.1 Présentation de la problématique

7.4.1.2.1.1 Qu'est ce qu'une URL pérenne ?

- Les URL pérennes sont des adresses internet particulières qui permettent de citer un document numérique, tout en ayant la garantie que ce lien hypertexte ne risque pas de changer.
  - Il existe différents systèmes permettant de créer des URL pérennes. Cela conduit à la gestion d'identifiants pérennes.

7.4.1.2.1.2 Objectifs

- L'objectif de la mise en place de ces URL est de faciliter la « citabilité » et le référencement de documents numériques, donc l'accès (ou mieux encore l'accessibilité, la présence…)
  - Permet d'ajouter un document dans ses favoris, de le citer sur un site Web, dans un mail, sur un blog ou sur les réseaux sociaux (et autres forums), simplement en utilisant l'adresse avec la garantie que l'accès sera préservé dans le temps
- La mise en œuvre d'URL avec identifiants pérennes permet :
  - d'afficher l'identifiant pérenne dans la barre d'URL lors de la consultation d'un document numérisé ;
  - de conserver dans l'URL le nom de domaine du contexte de visualisation (différents services peuvent exposer le même objet numériques avec des visualisations différentes)
  - d'appeler chaque service de visualisation (pagination, table des matières, etc.) dans l'URL à l'aide d'un paramètre simple, nommé « qualifieur » ;
  - d'obtenir plus facilement qu'auparavant l'URL d'une page précise au sein d'un document numérisé.

7.4.1.2.1.3 Préconisation E-ARK

- Requirement 2.4 : It SHOULD be possible to identify any Information Package globally uniquely
  - « Globally » par opposition à « repository » qui vaut pour le SAE en charge à un instant *t*

7.4.1.2.2 Solutions envisagées

Identifiants au format ARK et au format « Vitam »

7.4.1.2.2.1 ARK

Source : http://tools.ietf.org/id/draft-kunze-ark-15.txt

7.4.1.2.2.2 Forme d'un ARK

[http://NMAH/{]}ark:/NAAN/Name{[}Qualifier]

- [http://NMAH/]
  - Non obligatoire : Indique le lien Web complet, y compris l'URL d'accès.
- ark :/NAAN/Name
  - NAAN indique la référence du contexte (BNF par exemple) via un identifiant attribué
  - Name indique la référence unique de l'objet dans le contexte NAAN
- [Qualifier]
  - Permet de préciser le « type » de ce qu'on veut accéder (métadonnées, original, …)

7.4.1.2.2.3 Identifiant Vitam

La logique est d'utiliser des GUID (Global Unique Identifier) pour chacun des éléments dans Vitam (Unit, Groupe d'objet, Objets mais aussi Journaux, Logs, Services, …).

7.4.1.2.2.4 Logique de construction

- Version fixe
- Type d'objet fourni en paramètre
- **Domaine métier / Tenant (NAAN) fourni en paramètre et lié au tenant ou à un numéro 0 (interdit sinon) pour le test uniqu**
    - La valeur 1 serait sans doute pour toute la plateforme (information transverse à tous les tenants).
- Identifiant plate-forme fixe par fichier de propriété ou dynamique pour les ccas non Vitam (offres de stockage)
- Processus calculé à l'instanciation de la classe
- Temps UTC dynamique
- Compteur discriminant en fonction du temps UTC (seule zone de calcul en mode « synchronized » pour assurer l'unicité au sein d'une JVM)
- 4 bits de fin à zéro

7.4.1.2.2.5 Logique d'affichage

- Vision ARK : ark :/Domaine sur 9 chiffres/reste des informations avec la même logique que la vision Vitam
- Vision Vitam : dans l'ordre et représenté en forme Base 32
    1. Domaine
    2. Version
    3. Type d'objet
    4. Plate-forme
    5. Processus
    6. Temps UTC
    7. Compteur
    8. Non utilisé

7.4.1.2.2.6 Capacité de déconstruction

Il faudra déterminer ce qui pourrait être reconstruit depuis un identifiant Vitam de ce qui ne devrait pas, mais a priori toutes les informations seraient re-constructibles.

1. **Domaine**
    - L'intérêt est de pouvoir déterminer rapidement si un identifiant concerne un Tenant en particulier.
2. Version
    - L'intérêt est de pouvoir déchiffrer très vite sur quelle s'appuie l'identifiant et donc l'extraction des éléments suivants
3. **Type d'objet**
    - Utile dans le cadre d'un service « WhoAmI » calculé sans appel à la base
4. Plate-forme
    - Utile pour la traçabilité des opérations
5. Processus
    - Utile pour la traçabilité des opérations
6. **Temps UTC**
    - Utile pour la détermination a posteriori de l'adéquation du temps « officiel » avec le temps de création de l'ID
7. Compteur
    - A priori sans intérêt particulier (a pour objet uniquement d'éviter les collisions)

7.4.1.3 Graphes

Vitam traite des arbres des archive units et des groupes d'objet qui peuvent être présenter par des graphes précisement par Graphe orienté acyclique(D.A.G).

7.4.1.3.1 Objectifs

Pour vérifier la structure des arbres dans le bordereau SEDA (on n'a pas des cycles dans les arbres des units), et pour cela il faudrait créér des Graphes orientés. un autre problème qui s'impose pour un fichier seda complexe, l'ordre de l'indexation : il faut toujours indexer les parents avant les fils afin qu'ils puissent hériter toutes les informations des parents lors de l'indexation.

7.4.1.4 Vérification des formats :

Cette vérification de format devra intervenir à différents endroits du processing, et pour différents types de workflow. A l'heure actuelle, pour le processus d'Ingest, nous avons :

- vérification du format du SIP intégré dans l'upload (zip, tar, tar.gz…)
- vérification des objets techniques contenus dans le SIP.

Il apparait clairement, qu'une mise en commun de cet outil doit être effectué. C'est pourquoi le module common-format-identification a été ajouté dans la partie commune. De cette manière un outil de vérification des formats pourra être utilisé dans n'importe couche Vitam, si besoin.

Pour le moment, l'outil choisi pour effectuer cette vérification de format est Siegfried.

7.4.2 Architecture Technique

7.4.2.1 Introduction

7.4.2.1.1 But de cette documentation

L'objectif de cette documentation est d'expliquer l'architecture fonctionnelle de ce module.

7.4.2.1.2 GUID

Cf chapitre dédié

7.4.2.2 GUID

Le sujet porte notamment sur les **GUID**.

7.4.2.2.1 Identifiant Vitam

La logique est d'utiliser des GUID (Global Unique Identifier) pour chacun des éléments dans Vitam (Unit, Groupe d'objet, Objets mais aussi Journaux, Logs, Services, …).

Le GUID s'appuie sur l'objet ServerIdentity que chaque Service (JVM) doit instancié correctement.

7.4.2.2.1.1 Forme d'un identifiant Vitam

- Identifiant en base 32 (pour des raisons de lisibilité et d'éviter des erreurs de transcription)
- Longueur de 36 caractères base 32 représentant 22 octets natifs
- L'identifiant ne doit pas être trop long car il coûte en mémoire et sur disque
  - pour 10 milliards d'objets, on peut estimer qu'un octet coûte 100 Go sur disques et 1 Mo en mémoire par serveur
- La composition de l'identifiant serait a priori la suivante : **22 octets soit 168 bits**
  - Une version de l'algorithme d'identifiant entre 0 et 255 (**8 bits**)
  - Un identifiant de type d'objets entre 0 et 255 (Unit, Groupe d'objets, Objet, Entrée, Transfert, Journal, ...) (**8 bits**)
  - Un domaine métier = tenant entre 0 et 2^30-1 permettant une distribution par tenant, correspondant au NAAN de ARK (**30 bits**)
    - ARK impose une longueur de 5 ou 9 caractères en numérique uniquement
    - Compte tenu que la liste ARK dépasse déjà 95 000, il faudrait peut-être anticiper la taille à 9 chiffres
  - Un identifiant de plateforme entre 0 et 2^31-1 permettant une distribution par instance Vitam (**31 bits**)
    - Cet identifiant serait en 2 parties : partie fixe par plate-forme (1 par site ou 1 pour 3 sites), partie variable par instance de host (VM)
    - La partie plate-forme devrait permettre 2^20-1 items, soit 20 bits
    - La partie par instance de host devrait permettre 2^11-1 items, soit 11 bits
    - Cet identifiant est assimilable à une adresse MAC mais dont la garantie n'est pas suffisamment fiable en virtuel (assignation dynamique de MAC address)
    - Cet identifiant de 31 bits pourrait aussi être utilisé dans d'autres cas que Vitam pur, comme dans une offre de stockage pour gérer la distribution
      - Par exemple : Distribution sur les Cas Container sur 20 bits et distribution d'un Cas Storage dans un Cas Container sur 11 bits
  - Un identifiant de processus attribuant l'Id (0 à 2^22-1) (**22 bits**)
  - Le temps UTC exprimé en millisecondes entre 0 et 2^48-1 (8 925 années après 1970) (**48 bits**)
  - Un compteur discriminant de milliseconde entre 0 et 2^24-1 (**24 bits**)

Avertissement : Risque de collisions autour de 2^17 ~ 100K GUID générés par millisecondes, donc avec la progression des puissances de calculs sur 20 ans (Loi de Moore approchée : *2 tous les 3 ans) = 2^7+17 = 2^24

Note : Certains bits ne sont pas utilisés (5) pour de futurs usages.

7.4.2.3 Configuration jetty

Le besoin est de pouvoir fournir la capacité de configurer de manière programmatique Jetty. On peut penser aux besoins suivants :

1. Choisir le port de connection
2. Choisir le connecteur HTTP/HTTPS que l'on désire utiliser
   - **Paramètres communs aux connecteurs HTTP et HTTPS**
     - Taille des pools de thread (min,max nombre de threads)

    - Taille du backlog (nombre de connections en attente d'un thread disponible)
    - Différents timeout
- **Paramétres spécifiques à la couche TLS**
    - Paramétres liés aux keystore (emplacement, mot de passe keystore, mot de passe des clés privées)
    - Paramétres liés aux trustore (idem keystore)
    - Paramétres liés à TLS (protocoles autorisés, ciphers autorisés, options TLS)

7.4.2.4 Gestion des Handlers :

Pour la gestion de ces différents paramètres, on utilise le système de configuration en "Inversion of Control" de Jetty. Un exemple de configuration est disponible à l'adresse suivante : https://gist.github.com/gustavosoares/1438086

Cette solution présente les avantages suivants : une gestion relativement souple de la configuration (la prise en compte du binding d'un paramètre ne nécessite pas de coder le binding) un exploitant qui connaît déjà Jetty sera en terrain connu de configuration

Parmi les choix à faire, il faut décider si on limite la configuration par fichier xml à la configuration "serveur d'application" ou si on pousse à la configuration des servlet .

L'important est d'utiliser la classe XMLConfiguration (package maven jetty-xml) dont la javadoc est disponible à l'adresse : http://download.eclipse.org/jetty/stable-9/apidocs/org/eclipse/jetty/xml/XmlConfiguration.html Pour la mise en oeuvre de ce composant, voici le pseudo code :

```
URL jettyConfigFileURL = PropertiesUtils.findFile(<fichier>).toURI().toURL();
Server jettyServer = (Server) new XmlConfiguration(jettyConfigFileURL).configure();
<Ajout RequestHandler>
<Ajout ContextHandler>
jettyServer.start();
...
```

avec fichier qui est défini de la manière suivante :

- Si Le fichier passé en 1er argument du module (ex : access.conf) contient une variable nommé « jettyConfig » alors le serveur cherche dans son répertoire un configuration un fichier du nom de la valeur de jettyConfig .
- Si la variable jettyConfig n'existe ou s'il n'existe pas de fichier correspond à la valeur de la variable "jettyConfig" , le serveur cherche un fichier « jetty-vitam.xml » dans le répertoire de configuration
- Si les 2 premiers cas échoue, le serveur s'arrête en erreur

A noter : pour les tests Unitaires, comme il n'y a pas de besoins de tuning particulier (pour l'instant) et qu'il y a un besoin d'avoir le port variable, on conserve la méthode actuelle pour démarrer les serveur (la méthode actuelle est de faire un (new Server (port) de la classe org.eclipse.jetty.server.Server).

Les modules concernées sont :

- access-rest
- ihm-demo-web-application
- ingest-external-rest
- ingest-internal-rest
- metadata-rest

TODO : * functional-administration-rest * logbook-rest * processing-management * storage-engine-server * storage-offer-default * workspace-rest

7.4.2.5 Schéma de certificats et d'authentification

7.4.2.5.1 Présentation

Pour sécuriser les échanges, les services externes (ingest-external et access-external) seront exposés en HTTPS avec une authentification TLS mutuelle (authentification des clients par certificats x509). Pour permettre la consultation des URLs de status sans disposer de certificat (par exemple, pour la supervision), au niveau TLS, l'usage d'un certificat client sera proposé mais non obligatoire (WANT et non NEED clientCertificate) Si un certificat est présenté, - Jetty fait la poignée de main TLS et refuse si le certificat n'est pas « valide » à ses yeux. Un certificat valide est un certificat signé par une autorité présente dans la liste des autorités de confiance du serveur (truststore), qui n'est pas expiré (champs Not Before, Not After), qui, s'il implémente les extensions x509 keyUsage et extendedKeyUsage, dispose des bons droits pour être un certificat client. Si le client présente un certificat client invalide, jetty ferme la session TCP - Shiro vérifie si le certificat présenté et bien autorisé par Vitam. Dans l'implémentation actuelle (itération 8), cela

1. Configuration serveur jetty : le serveur sera lancé avec 2 magasins de clé suivants - keystore.jks : contient le certificat le la clé privé du serveur - truststore.jks : contient la chaînes des CAs qui génère ce certificats de clients & serveurs 2.Configuration de Shiro - granted_certs.jks : list de certificats du client qui sont autorisés à faire des requêtes vers le serveur - truststore.jks : contient la chaînes des CAs qui génère ce certificats de clients & serveurs 3. Configuration client : le client qui doit présenter sa clé privé & le certificat (format certificat PEM ou PKCS12 contenant clé privé ou publique) pour l'authentification lors de la requête.

7.4.2.6 Common format identification

7.4.2.6.1 Présentation

Le fonctionnement de cette brique est la suivante. Un outil d'identification est installé sur un environnement à déterminer. Ce service offre une API Rest permettant d'obtenir :

- un status
- l'analyse d'un format en fonction du Path vers le fichier à analyser.

Package parent : **fr.gouv.vitam.common.format.identification**

7.4.2.6.2 Sous packages

7.4.2.6.2.1 Identification :

Package : **fr.gouv.vitam.common.format.identification**

Ce package contient une factory, une interface de client, ainsi qu'un client mocké. Il contient également une enum précisant les différents clients disponibles (pour l'instant au nombre de 2 : siegfried + mock).

7.4.2.6.2.2 Exceptions :

Package : **fr.gouv.vitam.common.format.identification.exception**

Exceptions retournées par la vérification de formats. Sont au nombre de 5 :

- FileFormatNotFoundException : exception levée en cas de non résolution d'un format de fichier.

- FormatIdentifierBadRequestException : exception levée si la requete soumise à l'outil n'est pas correcte.
- FormatIdentifierFactoryException : exception levée dans le cadre de la factory.
- FormatIdentifierNotFoundException : exception levée si l'outil ne peut pas être interrogé.
- FormatIdentifierTechnicalException : exception levée en cas d'erreur technique générique.

7.4.2.6.2.3 Model :

Package : **fr.gouv.vitam.common.format.identification.model**

Ce package contient une classe de configuration ainsi que 2 POJO de réponses pour des appels au service.

7.4.2.6.2.4 Siegfried :

Package : **fr.gouv.vitam.common.format.identification.siegfried**

Ce package contient les différences classes pour l'utilisation d'un client Siegfried. Une factory, un mock ainsi qu'un client REST.

7.4.2.7 Messages

La classe **fr.gouv.vitam.common.i18n.Messages** permets de récupérer des messages internationalisé par l'utilisation d'un *ResourceBundle*.

Elle utiliser des fichiers de resources properties dans le format suivant : *messages_fr.properties* où :

- *messages* est le nom du bundle
- *fr* est la locale

Aujourd'hui, seule la locale « fr » est gérée et les fichiers doivent être créés dans le dossier src/main/resources du module common-public.

Cette classe peut être utilisée en définissant un service qui utilise la classe *Messages* avec un fichier custom.

7.4.2.8 Messages Logbook

Ce service permet de centraliser les messages des logbooks.

- **Nom du bundle** : vitam-logbook-messages
- **Service** : fr.gouv.vitam.common.i18n.VitamLogbookMessages.java

Ce service offre des méthodes permettant de récupérer des messages de logbook opération et cycle de vie. Il offre également la possibilité de récupérer toutes les clés et messages du fichier. Cette méthode ne doit être que ponctuellement pour des raisons de performance (elle est destinée à l'ihm-demo).

7.4.2.9 Request ID

Le **Request ID** est un identifiant métier de corrélation qui doit être positionné par l'appelant.

Il permet de suivre un traitement à travers tous les services qui y participent.

Cet identifiant est transporté par le header HTTP « **X-REQUEST-ID** ».

7.4.2.9.1 Filtre client

Classe : **fr.gouv.vitam.common.client2.RequestIdClientFilter**

Récupère le **Request ID** depuis le **VitamSession** et le positionne dans le Header « **X-REQUEST-ID** ».

Ce filtre est référencé dans *fr.gouv.vitam.common.client2.AbstractCommonClient.AbstractCommonClient(VitamClientFactoryInterface<*

7.4.2.9.2 Sauvegarde dans le thread local

Package : **fr.gouv.vitam.common.thread**

Le **Request ID** est sauvegardé dans l'objet **VitamSession** qui est positionné dans le **VitamThreadFactory.VitamThread** qui étend le thread local.

Le **VitamThreadPoolExecutor** gère la recopie du **VitamSession** d'un thread père vers un thread fils.

Le **VitamThreadPoolExecutor.VitamRunnable** encapsule le **VitamThreadFactory.VitamThread**.

VitamThreadUtils permet de récupérer le **VitamSession**. Si l'état du thread ne le permet pas, une **VitamThreadAccessException** est levée.

7.4.2.9.3 Filtre Serveur

Classe : **fr.gouv.vitam.common.server2.RequestIdContainerFilter**

Extrait le **Request ID** depuis le Header « **X-REQUEST-ID** » et le positionne dans le **VitamSession**.

Ce filtre est référencé dans *fr.gouv.vitam.common.server2.application.AbstractVitamApplication.buildApplicationHandler()*

Si le request ID présent dans la session n'était pas nul, on trace un warning.

7.4.2.9.4 Affichage dans les logs

Pour afficher le request ID dans les logs, le mécanisme MDC de Logback est utilisé : http://logback.qos.ch/manual/mdc.html

Dans le **VitamSession**, lorsque qu'on fait un **setRequestId**, cela positionne la valeur au niveau du MDC :

MDC.put(GlobalDataRest.X_REQUEST_ID, newRequestId);

Dans la configuration de Logback, on rajoute **%X{X-REQUEST-ID}** dans le pattern de log. Par exemple :

```
<pattern>%d{ISO8601} [%thread] [**%X{X-REQUEST-ID}**] %-5level %logger - %replace(
↪%caller{1..2}){'Caller\+1    at |\n',''} : %msg %rootException{5}%n</pattern>
```

7.4.3 Securite

7.4.3.1 Introduction

7.4.3.2 Securité de MongoDB

7.4.3.2.1 Objectifs

L'objectif est de sécuriser l'accèss à la base de donnée MongoDB. MongoDB exige que tous les clients de s'authentifier afin de déterminer leur accès.

Pour contrôler l'accèss à Mongo, vous avez besoin de créer une base de donnée pour chaque module (ex. MetaData, Logbook et Functional-administration) et ajouter les comptes applicatifs aux bases de données.

- Pour functional-administration : db-functional-administration ; user : user-functional-administration
- Pour logbook : db-logbook ; user : user-logbook
- Pour metadata : db-metadata ; user : user-metadata

Lorsque vous ajoutez un compte applicatif, vous créez l'utilisateur avec son mot de passe dans une base de données spécifique. Cette base de données est la base de données d'authentification pour l'utilisateur.

7.4.3.3 secret de la plateforme

7.4.3.3.1 Objectifs

Un secret de plateforme est utilisé afin de protéger contre des erreurs de configuration entre différentes plateformes

Si le secret de plateforme n'est pas transmis ou s'il est faux (non reconnu), la requête doit être refusée. Si le secret de plateforme est transmis et reconnu, la requête s'exécute normalement.

7.5 Functional administration

7.5.1 Architecture Fonctionnelle

7.5.1.1 Introduction

7.5.1.1.1 But de cette documentation

Ce document fournit une vision globale sur le module functional-administration.

Le module functional-administration propose un service de gestion sur les aspect différents de la plate-forme VITAM. Pour l'instant, deux fonctionnalités de gestion prévues supportées

- gestion de format
- gestion de règles
- gestion des contrats d'accès
- gestion des contracts d'entrée
- gestion des profiles
- gestion des contextes
- gestion des profiles de sécurité

7.5.1.2 Gestion de format

7.5.1.3 Gestion de règles

- L'application VITAM permet d'importer un référentiel pour les règles de gestion.
- Un référentiel peut être importé plusieurs fois.
- Si une règles de gestions est lié à une archive unit alors cette règles de gestion ne peux pas être supprimer.
- Si une règles de gestions présente dans le référentiel a été modifiée alors la version de la règles est égale à la version du référentiel.

7.5.1.4 Sauvegarde du référentiel des règles de gestion

- Le référentiel importé est stocké dans l'espace de stockage avec sa version.

7.6 IHM demo

7.6.1 Architecture fonctionnelle

7.6.1.1 Architecture fonctionnelle de l'application Back

7.6.1.1.1 But de cette documentation

On présente dans ce document l'architecture fonctionnelle de l'application Back IHM de VITAM.

7.6.1.1.2 Fonctionnement général du module

L'application IHM-DEMO est une application web dont la partie Front est une application Single Page développée avec le framework AngularJS 1.5.3 et côté serveur on utilise un serveur Jetty intégré qui gère les appels à ses services REST. Dans ce document, on s'intéresse à l'application côté serveur. On détaille dans la suite le fonctionnement par service REST.

> **Prudence :** La solution logicielle *VITAM* étant avant tout un *back office*, si vous possédez une *IHM* raccordée à VITAM, il n'est pas recommandé d'installer ce composant en environnement de production.

7.6.1.1.2.1 Recherche des units : POST /ihm-demo/v1/api/archivesearch/units

L'application Front construit en amont un objet Json passé dans le corps de la requête HTTP qui décrit les critères de recherche, les colonnes à afficher et le tri par défaut. Ci-dessous, la structure de l'objet reçu :

{ Title = *titleCriteria*
projection_transactdate = « TransactedDate »
projection_id = « #id »
projection_title = « Title »
orderby = « TransactedDate »
}

- L'entrée *Title* définit la chaîne de caractères saisie par l'utilisateur et utilisée pour la **recherche exacte** sur les titres des archive units.
- Pour faire la distinction entre les champs utilisés dans la partie *query* de la requête DSL et les colonnes sélectionnées (*projection*), le préfixe *projection_* doit être ajouté à toutes les colonnes à afficher. Le résultat affiché inclut les colonnes *TransactedDate*, *id* et *Title*.
- L'entrée *orderby* définit la colonne sur laquelle le tri par défaut sera fait côté serveur.
- Il faudrait noter ici le caractère # ajouté à la sélection du champ *_id*. En fait, afin de permettre la sélection des champs protégés tels que *_id* il faut remplacer le caractère _ par le caractère #.

Cet objet est convertit en Map<String, String> qui sera utilisée pour construire la requête DSL de sélection. On passe la main maintenant à la classe utilitaire *DslQueryHelper* qui construit à partir de la Map reçue la requête DSL de sélection. Une instance de la classe *fr.gouv.vitam.builder.request.construct.Select* est créée et alimentée pour obtenir à la fin la structure suivante :

{

$query :[{« $and » :[{« $eq » :{« title » : »titleCriteria »}}]}],

$filter :{« $orderby » :{« TransactedDate » :1}},

$projection :{« $fields » :{« Title » :1, « #id » :1, « TransactedDate » : 1}}

}

La classe *UserInterfaceTransactionManager* appelle le client Access qui prend en charge l'appel de MetaData et la récupération du résultat de recherche. Ci-dessous la structure du résultat retourné à l'application Front :

{ $hint : { total :x },

$context : {},

$result : [*tableau des archive units trouvées*]

}

7.6.1.1.2.2 Affichage du détail d'une archive unit : GET /ihm-demo/v1/api/archivesearch/unit/{id}

Le processus d'affichage des détails d'une archive unit est déclenché suite à une sélection faite sur un résultat de recherche. L'id de l'unité sélectionnée est passé en tant que paramètre dans l'URL.

Pour indiquer qu'il s'agit d'une sélection par id (c'est à dire une archive unit spécifique), la Map utilisée pour la construction de la requête DSL va contenir seulement une entrée : **(SELECT_BY_ID, id)**. De ce fait, la requête DSL de sélection introduit l'entrée root égale à l'id de l'unit sélectionnée. Donc, on aura la structure suivante :

{« $roots » :[id], »$query » :[], »$filter » :{}, »$projection » :{}}

De même, on appelle le client Access pour passer la requête au moteur MetaData qui retourne la structure de résultat de recherche mais on aura dans le bloc $result un tableau contenant un seul objet qui est l'archive unit sélectionnée.

{ $hint : { total :1 },

$context : {},

$result : [{*détails de l'archive unit sélectionnée (toutes les colonnes)*}]

}

7.6.1.1.2.3 Modification et enregistrement des détails d'une archive unit : PUT /ihm-demo/v1/api/archiveupdate/units/{id}

Au niveau du formulaire d'une archive unit, l'utilisateur peut modifier toutes les données affichées mises à part l'id et les données de management. L'application Front passe seulement les champs qui ont été modifiés pour la sauvegarde sous la forme d'un tableau d'objet Json. Dans la suite la structure retournée :

[{« fieldId » : »XXXXXXXX », »newFieldValue » : »VVVVVVVVV »}, {« fieldId » : »YYYYYYYYY », »newFieldValue » : »VVVVVVVVV »}, ...]

On convertit cette structure en Map<String, String> et on ajoute une entrée (SELECT_BY_ID, id) pour intégrer le bloc *root* à la construction de la requête DSL de l'update. on construit cette fois-ci une instance de la classe *fr.gouv.vitam.builder.request.construct.Update* et on ajoute des Actions de type *fr.gouv.vitam.builder.request.construct.action.SetAction*.

Voici un exemple de la requête obtenue :

{« $roots » :[*id*], »$query » :[], »$filter » :{}, »$action » :[{« $set » :{« date » : »09/09/2015 »}},{« $set » :{« title » : »Archive2 »}}]}

De nouveau, on passe la requête DSL à Access qui retourne à son tour le résultat de l'opération d'update avec la même structure des requêtes de sélection mais sans résultat car l'application Front relance la récupération de l'archive unit à la réception de la réponse.

7.6.1.1.2.4 Remarque importante

Pour le moment, on ne gère pas la mise à jour des champs de type tableau qui va faire appel à un autre type d'action.

7.6.1.1.2.5 Reste à faire

Dans la suite les services REST qui sont en cours de traitement :

- Recherche sur les opérations logbook
- Affichage du détail d'une opération logbook
- Téléchargement d'un SIP
- Recherche sur le référentiel des formats
- Affichage du détail d'un format
- Validation d'un référentiel à télécharger
- Téléchargement d'un référentiel de formats
- Suppression d'un format

7.6.1.2 Architecture fonctionnelle de l'application Front

7.6.1.2.1 But de cette documentation

Cette documentation présente l'architecture fonctionnelle de l'application Front du programme VITAM.

7.6.1.2.2 Modules AngularJS déclarés

Afin d'assurer la modularité et la séparation des différentes fonctionnalités de l'application Front, on a opté pour créer des modules AngularJS par fonctionnalité. Dans la suite les modules spécifiques créés :

- **ihm.demo** : Module principal
- **core** : Module qui regroupe les factories, services (fonctionnalités partagées entre les controllers)
- **archiveSearch** : Module de recherche d'archives
- **archive.unit** : Module du formulaire d'une archive

7.6.1.2.3 Routage

Les routes sont déclarées dans le fichier */modules/parent/app.config.js* :

- **/archiveSearch** : Recherche sur les Archive Units
- **/importPronoun** : Import du référentiel PRONOUN
- **/uploadSIP** : Import de SIP
- **/archiveunit/ :archiveId** : Affichage des détails d'une archive unit
- **/admin/logbookOperations** : Journal des opérations
- **/admin/formats** : Recherche sur le référentiel PRONOUN
- **/admin/logbookLifecycle** : en cours de construction
- **/admin/managementrules** : en cours de construction

7.6.1.2.4 Factories/Services

On a eu recours à des factories et des services pour assurer certaines fonctionnalités qui nécessitent un passage de données entre les controllers définis.

- **ihm-demo-factory.js** [trois factories ont été déclarées dans ce fichier :]
    1. **ihmDemoFactory** [définit les appels http aux services REST suivants :]
        - POST /ihm-demo/v1/api/archivesearch/units
        - GET /ihm-demo/v1/api/archivesearch/unit/id
        - PUT /ihm-demo/v1/api/archiveupdate/units/id
        - GET modules/config/archive-details.json
    2. ihmDemoCLient : crée un client RESTAngular configurable
    3. idOperationService : recherche l'id d'une opération logbook dans une liste de résultat
- **ihm-demo-service.js** [un seul service a été déclaré dans ce fichier :]
    - **archiveDetailsService** [définit la fonction *findArchiveUnitDetails* qui assure la récupération et l'affichage des détails d'une archive unit et qui prend en paramètre :]
        1. archiveUnitId : id de l'archive unit à afficher
        2. displayFormCallBack : fonction callback qui gère l'affichage du détail à la réception du retour de l'appel REST
        3. failureCallback : fonction callback qui gère l'echec de l'appel REST de récupération des détails d'une archive unit

7.6.1.2.5 Controllers

- **import-pronoun-controller.js** : le controller « MyController » déclaré dans ce fichier assure la création d'une instance FileUploader (composant qui gère l'import de fichier) et la définition de ses évènements *onSuccessItem* et *onErrorItem*.
- **upload-sip-controller.js**
- **archive-unit.controller.js** : définit le controller *ArchiveUnitController* qui assure l'affichage récursif des détails d'une Archive Unit et la sauvegarde des données modifiées dans le formulaire.
- **archive-search.controller.js** : définit le controller *ArchiveUnitSearchController* qui prend en charge la recherche par titre (mot exact) sur les archive units et aussi le lancement de l'affichage du formulaire d'une archive unit sélectionnée.
- **main.controller.js** : définit le controller *mainViewController* rattaché à la page principale index.html qui gère l'affichage du menu principal. Ce menu ne doit pas être affiché pour l'écran du formulaire d'une archive unit
- **file-format-controller.js**
- **logbook-controller.js**

7.6.1.2.6 Components

Les components ont été introduits à partir de la version 1.5.3 d'AngularJS pour apporter une solution plus simple pour développer des directives. Pour plus d'information, référez-vous à ce lien Component AngularJS [56].

- **archive-unit.component.js**
- **archive-search.component.js**
- **fileformat-component.js**
- **logbook-component.js**

56. https://docs.angularjs.org/guide/component

7.6.2 Architecture technique

7.6.2.1 Architecture technique de l'application Back

7.6.2.1.1 But de cette documentation

Cette documentation décrit l'architecture technique de la partie Back de l'application IHM de VITAM.

7.6.2.1.2 Organisation du module ihm-demo

L'application IHM de VITAM est assurée par le module ihm-demo composé de deux sous-modules :

7.6.2.1.2.1 1. Module ihm-demo-web-application

Ce module encapsule à la fois le serveur d'application et l'application Front (sous le répertoire main/resources/webapp). Vous pouvez vous référer à la documentation de l'application Front pour plus de détails.

7.6.2.1.2.2 package fr.gouv.vitam.ihmdemo.appserver

- ServerApplication : cette classe configure et lance le serveur d'application Jetty.
- **WebApplicationConfig** [cette classe définit les paramètres de configuration du serveur d'application]
    - **Paramètres de configuration du serveur IHM :**
        - port : port du serveur
        - serverHost : adresse du serveur
        - baseUrl : URL de base
- **WebApplicationResource** [cette classe définit les services REST assurés par l'application IHM :]
    - POST /ihm-demo/v1/api/archivesearch/units
    - GET /ihm-demo/v1/api/archivesearch/unit/{id}
    - POST /ihm-demo/v1/api/logbook/operations
    - POST /ihm-demo/v1/api/logbook/operations/{idOperation}
    - GET /ihm-demo/v1/api/status
    - POST /ihm-demo/v1/api/ingest/upload
    - PUT /ihm-demo/v1/api/archiveupdate/units/{id}
    - POST /ihm-demo/v1/api/admin/formats
    - POST /ihm-demo/v1/api/admin/formats/{idFormat}
    - POST /ihm-demo/v1/api/format/check
    - POST /ihm-demo/v1/api/format/upload
    - DELETE /ihm-demo/v1/api/format/delete

7.6.2.1.2.3 2. Module ihm-core

Ce module gère la couche fonctionnelle de l'IHM ainsi que l'interaction avec les autres modules de VITAM.

7.6.2.1.2.4 package fr.gouv.vitam.ihmdemo.core

- DslQueryHelper : cette classe fournit les méthodes de construction des requêtes DSL requises par les services de l'application IHM telles que les requêtes de sélection et de mise à jour.
- UiConstants (Enumeration) : définit les constantes partagées
- UserInterfaceTransactionManager : cette classe assure l'appel des autres modules VITAM ; en l'occurrence elle gère l'appel au module Access.

7.6.2.2 Architecture technique de l'application Front

7.6.2.2.1 But de cette documentation

Cette documentation présente la structure technique de l'application Front Single Page développée avec AngularJS 1.

7.6.2.2.2 Le Framework Front : AngularJS 1.5.3

7.6.2.2.2.1 Les modules AngularJS utilisés :

- angular-animate
- angular-resource
- angular-route

7.6.2.2.2.2 Autres frameworks Front utilisés

- bootstrap (3.3.x) : Responsive feature + CSS + Composants graphiques (bouton + label + zone de saisie)
- jquery (2.2.x)
- angular-material (1.1.0) : Les alertes affichées (de confirmation, d'erreur et d'information) et l'écran de détails d'une opération logbook
- angular-file-upload (2.3.4) : Composant pour l'import des fichiers (SIP, référentiels)
- restangular (1.5.2) : Client REST
- v-accordion (1.6.0) : Composant de regroupement (effet accordion) utilisé dans l'écran du formulaire d'une archive
- bootstrap-material-design-icons : Les îcones utilisées dans le menu et les boutons

7.6.2.2.3 Organisation de l'application

/webapp	**/archives** : les fichiers json utilisés pour tester l'affichage du formulaire d'une archive unit côté Front **/bower_components** : librairies et dépendances (téléchargées en exécutant npm install ou bower install) **/css** : feuilles de styles **/images** : images affichées dans l'application	
	/js	**/controller** : controllers AngularJS

/modules	**/archive-unit** : module qui gére l'écran du formulaire d'une Archive Unit **/archive-unit-search** : module qui gére l'écran de recherche des Archive Units **/config** : contient les éventuels fichiers utilisés pour customiser l'affichage. Actuellement, le fichier de traduction d'un premier lot de labels de meta données a été ajouté. **/core** : contient les factories et les services **/file-format** : module qui gére l'écran d'import du référentiel PRONOUN **/logbook** : module qui gére l'écran de recherche d'opérations Logbook **/parent** : répertoire qui contient les fichiers app.module.js et app.config.js qui définissent respectivement les modules Angular et les routes déclarés dans l'application. /views : templates HTML bower.json : dépendances gérées par bower index.html : page principale package.json : fichier de configuration nodejs

7.7 IHM recette

7.7.1 Architecture technique

7.7.1.1 Architecture technique de l'application Back

7.7.1.1.1 But de cette documentation

Cette documentation décrit l'architecture technique de la partie Back de l'application IHM de VITAM.

7.7.1.1.2 Organisation du module ihm-recette

L'application IHM de recette de VITAM est assurée par le module ihm-recette composé de trois sous-modules :

7.7.1.1.2.1 1. Module ihm-demo-web-application

Ce module encapsule à la fois le serveur d'application.

7.7.1.1.2.2 package fr.gouv.vitam.ihmdemo.appserver

- ServerApplication : cette classe configure et lance le serveur d'application Jetty.
- **WebApplicationConfig** [cette classe définit les paramètres de configuration du serveur d'application]
    - **Paramètres de configuration du serveur IHM :**
        - port : port du serveur
        - serverHost : adresse du serveur
        - baseUrl : URL de base

7.7.1.1.2.3 package fr.gouv.vitam.ihmdemo.appserver.performance.

- **PerformanceResource** [cette classe définit les services REST assurés par l'application IHM :]
    - POST /ihm-recette/v1/api/performance : permet de lancer un test de performance
    - HEAD /ihm-recette/v1/api/performance : permet de connaître l'état du test (en cours ou fini)
    - GET /ihm-recette/v1/api/performance/reports : liste les rapports de tests
    - GET /ihm-recette/v1/api/performance/reports/{fileName} : télécharge un rapport de test
    - GET /ihm-recette/v1/api/performance/sips : liste les fichiers pouvant servir de pour le test de performance

7.7.1.1.2.4 2. Module ihm-recette-web-front

Ce module contient la partie front de l'IHM de recette. Il s'agit d'une application classique angular 1.5.3 dont les dépendances de build sont gérés par le fichier *package.json* et les dépendances applicatives par le fichier *bower.json*.

7.7.1.1.2.5 3. Module ihm-core

Ce module gère la couche fonctionnelle de l'IHM ainsi que l'interaction avec les autres modules de VITAM.

7.7.1.2 Architecture technique de l'application Front

7.7.1.2.1 But de cette documentation

Cette documentation présente la structure technique de l'application Front Single Page développée avec AngularJS 1.

7.7.1.2.2 Le Framework Front : AngularJS 1.5.3

7.7.1.2.2.1 Les modules AngularJS utilisés :

- angular-animate
- angular-resource
- angular-route

7.7.1.2.2.2 Autres frameworks Front utilisés

- bootstrap (3.3.x) : Responsive feature + CSS + Composants graphiques (bouton + label + zone de saisie)
- jquery (2.2.x)
- angular-material (1.1.0) : Les alertes affichées (de confirmation, d'erreur et d'information) et l'écran de détails d'une opération logbook
- angular-file-upload (2.3.4) : Composant pour l'import des fichiers (SIP, référentiels)
- restangular (1.5.2) : Client REST
- v-accordion (1.6.0) : Composant de regroupement (effet accordion) utilisé dans l'écran du formulaire d'une archive
- bootstrap-material-design-icons : Les îcones utilisées dans le menu et les boutons

7.7.1.2.3 Organisation de l'application

/webapp	**/archives** : les fichiers json utilisés pour tester l'affichage du formulaire d'une archive unit côté Front **/css** : feuilles de styles **/images** : images affichées dans l'application		
		/js	**/controller** : controllers AngularJS
		/modules	**/parent** : répertoire qui contient les fichiers app.module.js et app.config.js qui définissent respectivement les modules Angular et les routes déclarés dans l'application. /views : templates HTML bower.json : dépendances gérées par bower index.html : page principale package.json : fichier de configuration nodejs

7.8 Ingest

7.8.1 Architecture Fonctionnelle

7.8.1.1 Généralités

Le rôle de l'ingest-internal est de réaliser un upload d'un SIP comme un InputStream, transféré de l'ingest-interne, qui viens d'une application externe via l'ingest-externe et de transférer les objets du serveur de stockage à ingest-externe. La procédure de upload d'un SIP est le suivant :

- appeller le service journalisation logbook pour créer des log
- Pousser le document le SIP dans le workspace.
- Appeller le service processing pour :
  - Lancer un workflow de production en mode continu ou étape par étape (*).
  - Lancer un workflow pour faire un test blanc en mode antinue ou etape par etape.Dans ce cas, on n'aura pas des unités archnivistiques et des groupes d'objet indexés, et on n'aura pas des objets stockés dans les offres de stockage(*).

- **Relancer un processus workflow en pause :**
    - En Mode étape par étape pour éxcuter l'etape suivante.
    - En Mode Continu pour exécuter toutes les etapes.
- Mettre en pause un processus workflow en cours d'exécution.
- Annuler un processus workflow en cours d'exécution ou en pause.

(*) : L'ingest interne est capable de déteminer l'identifiant du workflow qui sera exécuté par le moteur workflow (processEngine) grâce à l'identifant du contexte.

A titre d'exemple : le contextid c'est la combinaison mode d'exécution : production ou test à blanc, utilisateur connecté et contrat.

7.8.1.2 Généralités

Le rôle de l'ingest-external est de réaliser un upload d'un SIP provenant d'une application externe à vitam et de télécharger les fichiers sauvegardé au serveur après l'opération ingest (accusé de réception et seda).

7.8.1.3 Téléchargement standard et test à blanc d'un SIP :

La procédure de upload d'un SIP via ingest-externe est le suivant :

- **sauvegarder le fichier SIP temporaire dans le système**
    - préparer logbook opération (START)
    - scan le fichier SIP sauvegardé temporaire pour détecter des virus
    - préparer logbook opération (FIN)
    - si le fichier n'est pas infecté : appel client ingest-internal pour continuer le procédure de test à blanc (sans stockage des objets, sans indexations) ou le de dépôt en utilisant ingest-internal pour un dépôt dans la base VITAM.

7.8.1.4 Autres Fonctionnalités :

On peut également :

- **Relancer un processus workflow (production / test blanc) en pause :**
    - En Mode étape par étape pour éxécuter l'etape suivante.
    - En Mode continu pour exécuter toutes les etapes.
- Mettre en pause un processus workflow en cours d'exécution.
- Annuler un processus workflow en cours d'exécution ou en pause.

7.8.1.5 Ingest ExternalAntivirus

L'antivirus est intégré dans le processus de upload d'un SIP pour déterter un fichier infecté. L'antivirus rejettera les fichiers vérolés (qu'il pourrait corriger ou pas) afin d'éviter des problèmes d'authenticité au moment du contrôle.

Le critère d'acceptance - Étant donné : un SIP contenant un ou plusieurs fichiers infectés. Lorsque le SAE réalise : l'étape de check sanitaire

- Si des fichiers vérolés sont détectés et que l'antivirus peut les corriger, le workflow s'arrête à cette étape

eventType : « Contrôle sanitaire SIP » outcome avec statut « KO », outcomeDetailMessage : « Échec du contrôle sanitaire du SIP : présence de fichiers infectés » (fichier éventuellement corrigeable par l'antivirus). » objectIdentifierIncome : <nomDuSIP.extension>

- Si des fichiers vérolés sont détectés sans aucune correction de l'antivirus, alors le worflow s'arrête à cette étape.

eventType : « Contrôle sanitaire SIP » outcome avec statut « KO », outcomeDetailMessage : « Échec du contrôle sanitaire du SIP : présence de fichiers infectés ». objectIdentifierIncome : <nomDuSIP.extension>

7.8.1.6 Généralités

En plus de réaliser des upload de SIP, l'ingest-external expose aussi des méthodes pour gérer un process de traitement avec un workflow. Pour rappel, ingest-external fait appel à ingest-internal pour continuer l'exécution des méthodes demandées.

7.8.1.7 Fonctionnalités concernant le workflow

L'ingest external expose les méthodes suivantes pour gérer ces process :

- initVitamProcess : Initialiser un process avec un workflow.
- initWorkFlow : Initialiser un process avec un workflow. Cette méthode est dépréciée en faveur de initVitamProcess.
- updateOperationActionProcess : exécuter une action sur un process (next, resume, pause, cancel)
- updateVitamProcess : Cette méthode est dépréciée en faveur de updateOperationActionProcess
- executeOperationProcess : Elle expose les même fonctionnalités que updateOperationActionProcess en utilisant la méthode http POST.
- cancelOperationProcessExecution : Annuler l'exécution d'un process.
- getOperationProcessStatus : Retourne le statut d'un process
- getOperationProcessExecutionDetails : Retourne le détail d'un process
- listOperationsDetails : Lister tous les process qui sont en état RUNNING ou PAUSE.
- wait(int tenantId, String processId, ProcessState state, int nbTry, long timeWait, TimeUnit timeUnit) : Permet de bien gérer le pooling côté serveur. En effet, cette méthode fait appel à getOperationProcessStatus nbTry fois et esapce les appels avec un temps de timeWait. La réponse au client est retournée dans les cas suivants :

  > nbTry est atteint (nombre de rappel) > le state du process est COMPLETED > Le state du process est PAUSE et le statut est supérieur à STARTED

7.8.1.8 Les actions :

Les actions possible pour un workflow sont : INIT, NEXT, RESUME, PAUSE, CANCEL Dans le cas des méthodes initVitamProcess et cancelOperationProcessExecution les actions sont par défaut INIT et CANCEL respectivement.

Pour les autres méthodes : Les actions doivent êtres (NEXT, RESUME ou PAUSE)

- INIT : Initialiser un process avec un workflow et mettre son état à PAUSE (en attente d'une action)
- NEXT : Exécuter la première étape d'un process et mettre en état PAUSE. Si c'est la dernière étape alors mettre en état COMPLETED.
- RESUME : Exécuter tous les états d'un process et mettre en état COMPLETED
- PAUSE : Mettre le process en état PAUSE dès que possible. Si c'est la dernière étape en cours d'exécution alors mettre en état COMPLETED
- CANCEL : Mettre le process en état COMPLETED dès que possible.

7.8.1.9 Asynchrone :

L'exécution d'un process est complètement asynchrone. Donc pour avoir l'état final d'un process et son statut final il faut faire du pooling. La méthode wait est là pour vous aider. Attention : Au retour de la réponse d'une action sur le process ne vaut pas dire que l'exécution est terminé, il faut donc attendre la fin de l'exécution en appelant la méthode getOperationProcessStatus ou wait. Il faut faire attention aussi pour les tests d'intergrations et les tests de non régression.

7.8.2 Technique

7.8.2.1 Architecture Technique Ingest

7.8.2.1.1 Présentation

Cette section présente en bref l'architecture en général du module ingest. Le module ingest se compose de deux sous modules : ingest-external et ingest-internal.

Le premier rôle de l'ingest-internal est de réaliser un upload d'un SIP en prenant des données (le SIP le logbook) qui viennent de ingest-external après un scan virus sur le SIP. Son deuxième rôle est de transférer les objets sauvegardés dans le serveur de stockage comme Inputstream au ingest-external

Le premier rôle de l'ingest-external est de réaliser un upload d'un SIP provenant d'une application externe de vitam en se connectant au service. Le service ingest-external réalise un scan virus sur le SIP envoyé, préparé le logbook sur cette opération. Si le SIP n'est pas infecté, ingest-externe va appelé le service ingest-internal via son client avec des données de parametres (logbook & SIP) pour continuer le service. Son deuxième rôle est de télécharger les objets sauvegardés dans le serveur de stockage.

7.8.2.2 ingest-rest

7.8.2.2.1 Présentation

- Proposition de package : **fr.gouv.vitam.ingest.upload.rest**

Module utilisant le service REST avec Jersey pour charger SIP et faire l'appel des autres modules (workspace, processing et logbook, etc ...).

La logique technique actuelle est la suivante :

1. lancement du serveur d'application en appelant le fichier ingest-rest.properties (voir le document d'exploitation).
2. Créer une méthode upload du fichier sip
    1. Appel du journal pour la création des opérations (suivi du SIP).
    2. Push SIP dans le workspace.
    3. Appel du processing (journalisation des opération).
    4. Fermeture de la page des opérations.

7.8.2.2.2 `IngestInternalApplication.java`

classe de démarrage du serveur d'application de l'ingest interne.

```
// démarrage
public static void main(String[] args) {
    try {
        final VitamServer vitamServer = startApplication(args);
        vitamServer.run();
    } catch (final VitamApplicationServerException exc) {
        LOGGER.error(exc);
        throw new IllegalStateException("Cannot start the Ingest Internal ␣
↪Application Server", exc);
    }
}
```

Dans le `startApplication`, on effectue le start de `VitamServer`. Le *join* est effectué dans `run`. Le startApplication permet d'être lancé par les tests unitaires. Il peut être configuré avec un port d'écoute par les tests.

Dans le fichier de configuration, le paramètre `jettyConfig` est à paramétrer avec le nom du fichier de configuration de jetty.

7.9 Security-Internal

7.9.1 Architecture Fonctionnelle

7.9.1.1 Introduction

7.9.1.1.1 But de cette documentation

L'objectif de cette documentation est d'expliquer l'architecture fonctionnelle de ce module.

7.9.1.1.2 Security-internal

Le rôle de security-internal est de gérer les certificats applicatifs ainsi que les certificats personnels :

- Les certificats applicatifs sont associés aux SIA, et sont utilisés pour valider le certificat TLS d'appel à Vitam.
- Les certificats personnels sont utilisés dans l'authentification personae pour les endpoints dits « sensibles ».

7.9.2 Securite

7.9.2.1 Introduction

Les certificats applicatifs (SIA) et personnels sont stockés en base dans la collection MongoDB Identity dans les collections Certificate et PersonalCertificate respectivement. Ils sont indexés en base via leur hash (SHA256 du certificat encodé au format DER).

Le contrôle du certificat applicatif permet de vérifier si le certificat d'authentification TLS du SIA utilisé pour appeler Vitam est bien autorisé. Il permet également de récupérer l'identifiant du contexte associé.

Le contrôle d'accès sur les certificats personnels sont fait uniquement si le endpoint externe cible requière une authentification forte. La liste des endpoints nécessitant une authentification personnelle ou non est défini dans la configuration du module security-internal.

En cas d'échec de vérification du certificat personnel pour un endpoint nécessitant une authentification forte, la tentative d'accès est journalisée dans le journal des opérations.

7.10 Logbook

7.10.1 Architecture Fonctionnelle

7.10.1.1 Généralités

7.10.1.1.1 Journal d'opération

Le rôle du journal d'opération est de conserver une trace des opérations réalisées au sein du système lors de traitements sur des lots d'archives.

Chaque opération est tracée sous la forme de 2 enregistrements (début et fin).

Évènements tracés par exemple :

- Démarrage de Ingest avec affectation d'un eventIdentifierProcess = GUID (OperationId) (création)
  - A partir d'ici tous seront en mode **update**
- Stockage du lot d'archives dans l'espace de travail
- Démarrage d'un workflow
- Démarrage d'une étape de workflow
- Fin d'une étape de workflow
- Fin d'un workflow
- Fin du Stockage du lot
- Fin de Ingest

7.10.1.2 Journal de cycle de vie

Le rôle du journal de cycle de vie est de tracer toutes les événements qui impactent l'archive dès sa prise en charge dans le système et doit être conservé autant que l'archive.

- dès la réception de l'entrée, on trace les opérations effectuées sur les ArchiveUnit et les ObjectGroup qui sont dans le SIP
- les journaux du cycle de vie sont « committés » une fois le stockage des objets OK et l'indexation des MD OK, avant notification au service versant

7.10.1.3 Modèle de données

Afin d'assurer le suivi des opérations effectuées sur les archives, un ensemble d'informations sont conservées.

7.10.1.3.1 Description des champs

Les noms des champs sont basés sur les distinctions faites par PREMIS V3 entre :

- objet / agent / évènement
- type / identifiant

Les champs seront tous au même niveau dans le journal ==> pas de notion de bloc comme dans PREMIS, même si on préserve la capacité à générer un schéma PREMIS (et les blocs qui le compose).

Référence : http://www.loc/gov/standard/premis/v3/premis-3-0-final.pdf

Ci-après la liste des champs stockés dans le journal des opérations associées à leur correspondance métier :

Champ	Description	Oblig.	Provenance	Exemple métier	Commentaire
eventIdentifier	Identifiant de l'opération	**Oui**	interne (vitam)		Unique pour chaque ligne
eventType	Type d'opération	**Oui**	interne	CheckSEDAManifestExists	Information identifiant l'étape/action concernée (format : etape_action)
eventDateTime	Date de l'opération	**Oui**	calculé par le journal		
eventIdentifierProcess	Identifiant du processus	**Oui**	interne		GUID
eventTypeProcess	Type de processus	**Oui**	interne	Ingest	
outcome	Résultat	**Oui**	interne	Started, OK, Fatal, Warning	Il s'agit du status de l'opération. Par exemple lorsqu'une opération est lancée, le status est 'Started'.
outcomeDetail	Code correspondant à l'erreur	*Non*	interne	404_XXX	Constitué d'un code d'erreur http et d'un sous code d'erreur vitam plus précis.
outcomeDetailMessage	Informations détaillant la nature de l'erreur ou le message informatif de succès	**Oui**	interne		2 fonctions : Contient le message d'erreur détaillant le problème OU contient le contenu du champ SEDA 'comment' extrait. Dans ce dernier cas, la valeur n'est renseignée qu'une seule fois pour ne pas dupliquer l'information sur les lignes correspondant aux sous-opérations associées au même lot.
agentIdentifier	Agent réalisant l'action	**Oui**	calculé par le journal		Nom du serveur vitam exécutant l'action : calculé par le journal
agentIdentifierApplication	Nom de l'application s'authentifiant à Vitam pour lancer l'opération	**Non**	externe		Identifiant de l'application externe qui appelle Vitam pour effectuer une opération
agentIdentifierApplicationSession	Identifiant donnée par l'application utilisatrice à la session utilisée pour lancer l'opération	**Non**	externe		X-ApplicationId. l'application externe est responsable de la gestion de cet identifiant. Il correspond à un identifiant pour une session donnée côté application externe.

7.10.1.4 Modèle de données

Afin d'assurer le suivi des opérations du journal du cycle de vie effectuées sur les archives, un ensemble d'informations sont conservées.

7.10.1.4.1 Description des champs

Les noms des champs sont basés sur les distinctions faites par PREMIS V3 entre :

- objet / agent / évènement
- type / identifiant

Les champs seront tous au même niveau dans le journal du cycle de vie ==> pas de notion de bloc comme dans PREMIS, même si on préserve la capacité à générer un schéma PREMIS (et les blocs qui le compose).

Référence : http://www.loc/gov/standard/premis/v3/premis-3-0-final.pdf

Ci-après la liste des champs stockés dans le journal des opérations du journal du cycle de vie associées à leur correspondance métier :

Champ	Description	Oblig.	Provenance	Exemple métier	Commentaire
eventIdentifier	Identifiant de l'opération	**Oui**	interne (vitam)		Unique pour chaque ligne
eventType	Type d'opération	**Oui**	interne	CheckSEDAManifestExists	Information identifiant l'étape/action concernée (format : etape_action)
eventDateTime	Date de l'opération	**Oui**	calculé par le journal		
eventIdentifierProcess	Identifiant du processus	**Oui**	interne		GUID
eventTypeProcess	Type de processus	**Oui**	interne	Ingest	
outcome	Résultat	**Oui**	interne	Started, OK, Fatal, Warning	Il s'agit du status de l'opération. Par exemple lorsqu'une opération est lancée, le status est 'Started'.
outcomeDetail	Code correspondant à l'erreur	*Non*	interne	404_XXX	Constitué d'un code d'erreur http et d'un sous code d'erreur vitam plus précis.
outcomeDetailMessage	Informations détaillant la nature de l'erreur ou le message informatif de succès	**Oui**	interne		2 fonctions : Contient le message d'erreur détaillant le problème OU contient le contenu du champ SEDA 'comment' extrait. Dans ce dernier cas, la valeur n'est renseignée qu'une seule fois pour ne pas dupliquer l'information sur les lignes correspondant aux sous-opérations associées au même lot.
agentIdentifier	Agent réalisant l'action	**Oui**	calculé par le journal		Nom du serveur vitam exécutant l'action : calculé par le journal
agentIdentifierApplication	Nom de l'application s'authentifiant à Vitam pour lancer l'opération	**Non**	externe		Identifiant de l'application externe qui appelle Vitam pour effectuer une opération
agentIdentifierApplicationId	Identifiant donnée par l'application utilisatrice à la session utilisée pour lancer l'opération	**Non**	externe		X-ApplicationId. l'application externe est responsable de la gestion de cet identifiant. Il correspond à un identifiant pour une session donnée côté application externe.

7.10.2 Architecture technique

7.10.2.1 Introduction

7.10.2.1.1 Présentation

Parent package : **fr.gouv.vitam**
Package proposition : **fr.gouv.vitam.logbook**

7.10.2.1.2 Itération 3 et Itération 5

4 sous-modules pour le Logbook Engine. Dans logbook (parent).

- vitam-logbook-common : Classes et exception communes aux différents modules
- vitam-logbook-common-client : Classes communes pour les clients
- vitam-logbook-operations : module lié aux opérations
- vitam-logbook-operations-client : module client pour les opérations

7.10.2.1.2.1 Itérations suivantes / à plus long terme

- vitam-logbook-lifecycles : module des cycles de vie logs
- vitam-logbook-lifecycles-client : module client pour les cycles de vie
- vitam-logbook-administration : module pour l'administration du moteur de journalisation (sera détaillé plus en détail)
- vitam-logbook-administration-client : module client pour l'administration du moteur de journalisation (sera détaillé plus en détail)

7.10.2.1.3 Modules - packages logbook

logbook
/logbook-common
fr.gouv.vitam.logbook.common.client
fr.gouv.vitam.logbook.common.exception
fr.gouv.vitam.logbook.common.model
fr.gouv.vitam.logbook.common.parameters

/logbook-common-client
fr.gouv.vitam.logbook.common.client.singlerequest

/logbook-common-server
fr.gouv.vitam.logbook.common.server.database.collections.request
fr.gouv.vitam.logbook.common.server.exception

/logbook-operations
fr.gouv.vitam.logbook.operations.api
fr.gouv.vitam.logbook.operations.core

/logbook-operations-client

/logbook-lifecycles
fr.gouv.vitam.logbook.lifecycle.api
fr.gouv.vitam.logbook.lifecycle.core

/logbook-lifecycles-client

/logbook-administration

/logbook-administration-client

/logbook-rest

7.10.2.2 DSL

7.10.2.2.1 Analyse

7.10.2.2.1.1 Présentation

L'objet de cette analyse est de chercher quel pourrait être le langage de requête pour le journal.
A noter : les requêtes doivent disposer de quelques critères libres.

Plusieurs implémentations en ligne de mire possibles :

- **Requêtes équivalentes à l'usage dans des collections REST classiques en URL, avec la contrainte VITAM (cela doit être da**
    - Exemple Projection Google : ?fields=url,object(content, attachments/url)
    - Exemple de recherche classique : ?name=napoli&type=chinese,japanese&zipcode=75*&sort=rating,name&desc=rating 9
- **Requêtes dans le body permettant d'être un peu plus riche et notamment dans la composition :**
    - Des classes « Expression » permettent de gérer les différents cas de recherche : AND/OR, Property=value, opérateurs autres (IN, NE, GT, GTE…), NOT…

Un exemple de classes « Expression » :

- **Interface Expression ;**
    - **Interface AExpression ;**
        - **Abstract LogicalExpression ;**
            - Class AndExpression ;

        - Class OrExpression ;
    - Class PropertyExpression ;
- **Interface BExpression ;**
    - **Abstract OperatorExpression ;**
        - Class EqualExpression ;
        - Class GreaterThanExpression ;
        - ...
    - Class NotExpression ;

7.10.2.2.1.2 Explication

Une interface **Expression**.
2 interfaces **AExpression** et **BExpression**.

Une classe abstract **LogicalExpression** permettant de gérer les expressions logiques AND et OR.

```
LogicalExpression{
  Operator ope;//(ENUM)
  AExpression exp;
}
```

Les 2 classes implémentées sont *AndExpression* et *OrExpression*. La classe **PropertyExpression** permet de gérer les requetes sur les champs à proprement parler.

```
PropertyExpression{
  String propertyName;
  BExpression exp;
}
```

Une classe abstract **OperatorExpression** permettant de gérer les opérateurs IN, NE, GT, GTE...

```
OperatorExpression{
  Operator ope; //(ENUM)
  Scalar|Array value; //(Int, String...)
}
```

Les classes implémentées sont entre autres *InExpression*, *GteExpression*.... La classe **NotExpression** permet de gérer les expressions NOT.

```
NotExpression{
  Operator ope; //(ENUM -> NOT)
  BExpression exp;
}
```

7.10.2.2.1.3 Utilisation

Classe Query pour y intégrer une expression

```
Query{
  Expression exp;
}
```

Classe SearchQuery pour y intégrer une liste de Query

```
SearchQuery{
  List<Query> queries;
}
```

7.10.2.2.2 Conclusion

Il apparait clairement que - même s'il est compliqué - le DSL Vitam existant est très proche de l'analyse effectuée. Il pourra donc être utilisé pour la recherche dans le logbook, en adaptant les classes Query et Request (ou en adaptant les Helpers associés).

La réutilisation du même DSL va aussi dans le sens de la simplification du point de vue de l'utilisateur des API par l'uniformisation des DSL utilisés.

La recommandation de l'étude porte donc sur la réutilisation du DSL Vitam destiné aux Units et ObjectGroups pour les Journaux.

Néanmoins, il y aura quelques différences (pas de **roots**, ni de **depth**).

Additions IT18 : A ce jour, une implémentation du DSL en mode mono-query a été développée : la classe **DBRequestSingle**. Pour le moment il n'y a pas encore de mutualisation entre le DBRequestSingle et les requêtes Logbook car celles-ci sont encore trop spécifiques.

7.10.2.3 Rest

7.10.2.3.1 Présentation

Package Parent : **fr.gouv.vitam.logbook**
Proposition de package : **fr.gouv.vitam.logbook.rest**

Module hébergeant le support REST et le jar de lancement du service.

7.10.2.3.2 Services

7.10.2.4 Common-client

7.10.2.4.1 Présentation

Package parent : **fr.gouv.vitam.logbook**

Proposition de package : **fr.gouv.vitam.logbook.common.client**

Module utilisé pour les objets communs client/server :

- utils
- DTO

7.10.2.4.2 Services

7.10.2.5 Common-client

7.10.2.5.1 Présentation

Package parent : **fr.gouv.vitam.logbook**
Proposition de package : **fr.gouv.vitam.logbook.common.server**

Ce module est utilisé par les modules server operations et lifecycles et utilise :

- metadata-core
- logbook-common

7.10.2.5.2 Services

La logique technique actuelle est la suivante :

- Chaque journal est une collection dans MongoDB
- Chaque entrée dans la collection est la somme des événements d'une opération / cycle de vie
  - Opération ingest x contient l'ensemble des étapes de cette opération
  - Cycle de vie d'une archive x contient l'ensemble des événements associés à cette archive

Ceci facilite les recherches sur la base de l'entrée primaire (la première) mais n'interdit pas la recherche sur les entrées secondaires qui sont dans le tableau « **events** ».

Plus tard, ces journaux seront aussi écrits dans des fichiers.

- Opérations
  - Un par jour
  - Chaque event (unitaire et non globalisé) devra être écrit au fur et à mesure, c'est à dire en respectant les dates d'events (dans l'ordre acquité par le Moteur de journalisation)
- LifeCycles
  - Un fichier unique, les events dans l'ordre chronologique (qui correspond à l'odre deans events)

7.10.2.5.3 Données

Les données sont stockées dans 3 types de stockage :

- une base maître (MongoDB) qui contient toutes les données de type journal
- un base index (EslasticSearch) qui contient uniquement les journaux de type operation
- les offres de stockage qui contienent des fichiers sécurisés des journaux de type opération

La gestion de la base MongoDB se fait par le service d'accès *LogbookMongoDbAccessImpl* (implémentation de *LogbookDbAccess*). La gestion de la base ElasticSearch se fait par le service d'accès *LogbookElasticsearchAccess* (implémentation de *ElasticsearchAccess*).

En cas d'ajout / mise à jour /suppresssion les données sont d'abord gérées dans MongoDB puis la modification est répercutée (si nécessaire) dans ElasticSearch.

Pour le cas de la recherche, la requête de recherche est d'abord envoyée dans ElasticSearch pour récupérer une liste d'identifiants (List<ID>) qui sont ensuite envoyés en remplacement de la Query originale dans MongoDb pour récupérer le détail des données. La traduction d'une requête DSL vers une requête MongoDb se fait à l'aide des objets de traduction présent dans le package du module common-database-private : **fr.gouv.vitam.common.database.translators.mongodb**. La traduction d'une requête DSL et/ou MongoDb vers une requête Elasticsearch se fait à l'aide des objets de traduction présent dans le package du module common-database-private : **fr.gouv.vitam.common.database.translators.elasticsearch**.

7.10.2.6 Commons

7.10.2.6.1 Présentation

Package parent : **fr.gouv.vitam.logbook**
Proposition de package : **fr.gouv.vitam.logbook.common**

Module utilisé pour les objets communs :

- classes utils
- exceptions
- autres…

7.10.2.6.2 Services

7.10.2.7 Operation Client

7.10.2.7.1 Présentation

Parent package : **fr.gouv.vitam.logbook**
Package proposition : **fr.gouv.vitam.logbook.operations.client**

Module pour le client des logs opération.

7.10.2.7.2 Services

7.10.2.8 Opération

7.10.2.8.1 Présentation

Parent package : **fr.gouv.vitam.logbook**
Package proposition : **fr.gouv.vitam.logbook.operations**

Module pour le module opération : api / rest.

7.10.2.8.2 Services

7.10.2.8.3 Rest API

http://server/logbook/v1
POST /operations/id_op -> **POST nouvelle opération**
PUT /operations/id_op -> **Append sur une opération existante (ajout d'un item)**
GET /operations -> **retourne une liste d'opérations sous forme : id + autres infos de la dernière ligne de chaque opération ([{ id_op : id, last_line_infos }, . . .])**
GET /operations/id_op -> **accès aux évenements d'une opération**
GET /status -> **statut du logbook**

7.10.2.9 Lifecycle Client

7.10.2.9.1 Présentation

Parent package : **fr.gouv.vitam.logbook**
Package proposition : **fr.gouv.vitam.logbook.lifecycle**

Module client pour les logs lifecycle.

7.10.2.9.2 Services

7.10.2.10 Lifecycle

7.10.2.10.1 Présentation

Parent package : **fr.gouv.vitam.logbook**
Package proposition : **fr.gouv.vitam.logbook.lifecycle.client**

Module pour les logs lifecycle : api / rest.

7.10.2.10.2 Services

7.10.2.10.3 Rest API

http://server/app/v1
POST /operations/{id_op}/lifecycles/ -> **POST un lifecyle sur une opération**
PUT /operations/{id_op}/lifecycles/{id_li} -> **Append sur un lifecycle existant**
GET /lifecycle -> **Administration du lifecycle**

7.10.2.11 Administration-client

7.10.2.11.1 Présentation

Package Parent : **fr.gouv.vitam.logbook**
Proposition de package : **fr.gouv.vitam.logbook.administration.client**

7.10.2.11.2 Services

7.10.2.12 Administration

7.10.2.12.1 Présentation

Package parent : **fr.gouv.vitam.logbook**
Proposition de Package : **fr.gouv.vitam.logbook.administration**

7.10.2.12.2 Services

7.10.2.12.3 Rest API

http://server/app/v1

Administration
GET /status -> statut du logbook

7.10.2.13 Sécurisation (traçabilité)

7.10.2.13.1 Présentation

Package parent : **fr.gouv.vitam.logbook**
Packages concernés : **fr.gouv.vitam.logbook.common.traceability**, **fr.gouv.vitam.logbook.administration.core**, **fr.gouv.vitam.worker.core.plugin.lfc_traceability**

La sécurisation fige périodiquement le contenu des journaux dans un fichier horodaté, stocké sur les offres. Elle permet de démontrer a posteriori qu'aucune entrée n'a été modifiée ni supprimée. Trois périmètres sont sécurisés indépendamment :

- le journal des opérations (`TraceabilityType.OPERATION`);
- le journal du cycle de vie des unités archivistiques (`TraceabilityType.UNIT_LIFECYCLE`);
- le journal du cycle de vie des groupes d'objets (`TraceabilityType.OBJECTGROUP_LIFECYCLE`).

Le journal des écritures de stockage (`TraceabilityType.STORAGE`) suit le même algorithme générique, mais il est porté par le module *storage-engine-server* (**StorageTraceabilityAdministration**, **LogbookStorageTraceability-Helper**). Il n'est pas décrit ici : se référer à la documentation du module *storage*.

Modules impliqués :

- **common-private** : briques cryptographiques transverses, arbre de Merkle (`security.merkletree`) et production du jeton d'horodatage (`timestamp`) ;
- **logbook-common** : algorithme générique (**TraceabilityService**), contrat **LogbookTraceabilityHelper**, écriture du zip (**TraceabilityFile**), modèle **TraceabilityEvent** ;
- **logbook-administration** : sécurisation du journal des opérations (**LogbookAdministration**, **LogbookOperationTraceabilityHelper**) ;
- **logbook-administration-lfc** : déclenchement de la sécurisation des cycles de vie (**LogbookLFCAdministration**, **LfcTraceabilityType**) ;
- **worker-core** (`plugin.lfc_traceability`) : plugins PREPARE / BUILD / GENERATE / FINALIZE et croisement d'empreintes (**DigestValidator**) ;
- **processing-management** : définition des workflows `DefaultUnitLifecycleTraceability.json` et `DefaultObjectGroupLifecycleTraceability.json`;
- **scheduler-server** : jobs Quartz **TraceabilityJob** et **TraceabilityLFCJob**.

La documentation métier (finalité, valeur probante, exploitation fonctionnelle) n'est pas reprise ici : se référer à la documentation *vitam-doc*.

7.10.2.13.1.1 Vue d'ensemble

Deux déclencheurs et deux chemins d'exécution, mais un seul algorithme et un seul format d'archive :

```
                        scheduler-server
                               |
              +----------------+-----------------+
              |                                  |
       TraceabilityJob                   TraceabilityLFCJob
              |                                  |
              v                                  v
    LogbookAdministration            LogbookLFCAdministration
 (dans le serveur logbook,                       |
      sans workflow)               workflow LOGBOOK_*_LFC_TRACEABILITY
              |                    PREPARE > BUILD > GENERATE > FINALIZE
              |                       (distribué sur les workers)
              +----------------+-----------------+
                               v
                        TraceabilityService
                               |
                     LogbookTraceabilityHelper
                      (spécifique au périmètre)
                               |
          +--------------------+--------------------+
          v                    v                    v
      sélection          MerkleTreeAlgo       TimeStampService
(_lastPersistedDate)            |            + TimestampGenerator
          |                     |                    |
          v                     v                    v
      data.txt           merkleTree.json          token.tsp
                               |
                               v
             archive zip sur les offres + TraceabilityEvent
                       (evDetData de l'opération)
```

Le journal des opérations est sécurisé directement dans le serveur logbook ; les cycles de vie passent par un workflow, leur volumétrie et les accès aux métadonnées et aux offres imposant de distribuer le travail. À partir de **TraceabilityService**, le traitement est strictement le même.

7.10.2.13.2 Algorithme générique

L'algorithme est unique, porté par **TraceabilityService**. Il est paramétré par une implémentation de **LogbookTraceabilityHelper** propre à chaque périmètre. Le point d'entrée est `secureData(strategyId)`.

7.10.2.13.2.1 Déroulement

Vue synthétique de l'enchaînement, du point d'entrée jusqu'à la journalisation :

```
secureData(strategyId)
    |
    +-- helper.startTraceability()
    |
    +-- création du TraceabilityFile (zip temporaire, sans compression)
    |
    +-- helper.saveDataInZip(merkleAlgo, traceabilityFile)
    |       |
    |       +-- sélection des entrées (par _lastPersistedDate)
    |       +-- sérialisation CanonicalJsonFormatter
    |       +-- écriture d'une ligne dans data.txt
    |       +-- merkleAlgo.addLeaf(...)        <-- feuille ajoutée au vol
    |
    +-- merkleAlgo.generateMerkle()            --> merkleTree.json
    |       (si aucune entrée : suppression du zip, helper.saveEmpty(), arrêt)
    |
    +-- timeStampService.getDigestFrom(racine, jetons précédent / M-1 / A-1)
    |
    +-- TimestampGenerator (tampon RFC 3161)   --> token.tsp
    |
    +-- écriture de computing_information.txt et additional_information.txt
    |
    +-- helper.storeAndDeleteZip(...)
    |
    +-- helper.createLogbookOperationEvent(...) --> TraceabilityEvent en evDetData
```

Le détail de chaque étape :

1. `helper.startTraceability()` : ouverture ou rattachement de l'opération de traçabilité dans le journal des opérations.
2. Création du zip temporaire dans le dossier de travail, nommé `<tenant>_<zipName>_<yyyyMMdd_HHmmss>.zip`, où `zipName` est fourni par le helper et la date est la date de fin de traçabilité. **TraceabilityFile** écrit l'archive sans compression (`setLevel(0)`), le fichier étant destiné à être re-parcouru ligne à ligne lors des vérifications.
3. `helper.saveDataInZip(merkleAlgo, traceabilityFile)` : écriture de `data.txt`, une ligne par entrée sécurisée sérialisée en JSON canonique (**CanonicalJsonFormatter**). Chaque ligne est simultanément ajoutée comme feuille de l'arbre de Merkle.
4. `merkleAlgo.generateMerkle()` : construction de l'arbre, sérialisé dans `merkleTree.json`. Si l'arbre est `null`, c'est-à-dire s'il n'y a aucune entrée à sécuriser, le zip est supprimé, `helper.saveEmpty(tenantId)` est appelé et le traitement s'arrête.
5. Horodatage : l'empreinte à horodater est calculée par **TimeStampService** sur la concaténation de la racine de Merkle et des tampons des traçabilités précédente, M-1 et A-1, les valeurs nulles étant ignorées. Le tampon RFC 3161 est produit par **TimestampGenerator** puis écrit en base 64 dans `token.tsp`. Les valeurs d'entrée du calcul sont conservées en clair dans `computing_information.txt`.
6. `additional_information.txt` : `numberOfElements`, `startDate`, `endDate` et `securisationVersion`.

7. `helper.storeAndDeleteZip(...)` : dépôt du zip, directement sur les offres pour le journal des opérations, via le workspace pour les cycles de vie, puis suppression du fichier temporaire.
8. `helper.createLogbookOperationEvent(...)` : journalisation de l'événement final, portant le **TraceabilityEvent** en `evDetData`.

7.10.2.13.2.2 Contenu du zip

```
data.txt                      # une ligne JSON canonique par entrée sécurisée
merkleTree.json               # arbre de Merkle sérialisé
token.tsp                     # tampon RFC 3161 (base 64)
computing_information.txt     # racine courante + tampons J-1 / M-1 / A-1
additional_information.txt    # nombre d'entrées, bornes, SecurisationVersion
```

Le détail de chaque fichier est donné dans *Contenu de l'archive sécurisée* (page 167).

Le **TraceabilityEvent** enregistré en `evDetData` reprend ces informations sous forme exploitable : `LogType`, `StartDate` et `EndDate`, `Hash` (racine de Merkle), `TimeStampToken`, les identifiants et dates des traçabilités précédentes (`PreviousLogbookOperationId`, `MinusOneMonthLogbookOperationId`, `MinusOneYearLogbookOperationId`, ...), `NumberOfElements`, `FileName`, `Size`, `DigestAlgorithm`, `MaxEntriesReached`, `SecurisationVersion` et `Statistics`.

7.10.2.13.2.3 Rôle de LogbookTraceabilityHelper

L'interface concentre tout ce qui dépend du périmètre sécurisé :

- sélection et écriture des données : `saveDataInZip`, `getDataSize`;
- bornes temporelles : `getTraceabilityStartDate`, `getTraceabilityEndDate`;
- chaînage : `getPrevious*OperationId`, `getPrevious*StartDate`, `getPrevious*TimestampToken`, pour les trois profondeurs (précédente, M-1, A-1);
- journalisation : `startTraceability`, `createLogbookOperationEvent`, `saveEmpty`, `getStepName`, `getTimestampStepName`;
- stockage : `storeAndDeleteZip`, `getZipName`;
- métadonnées de l'événement : `getTraceabilityType`, `getMaxEntriesReached`, `getTraceabilityStatistics`, `getSecurisationVersion`.

La méthode par défaut `decorateZipName(zipName, operationId)` construit le nom de base de l'archive. Le nom historique est conservé pour la version de sécurisation par défaut; il est suffixé par `_<version>_<GUID>` pour toute autre version. La date étant ajoutée ensuite par l'appelant, l'analyse de date de **StorageFileNameHelper** n'est pas impactée.

7.10.2.13.3 Périmètre sécurisé

Le périmètre sécurisé est l'ensemble des entrées prises en compte par une opération de traçabilité. Il n'est pas défini par une liste d'identifiants, mais par un intervalle temporel évalué sur la date de dernière persistance de chaque entrée (`_lastPersistedDate`), et non sur sa date d'événement (`evDateTime`).

`_lastPersistedDate` est la date d'écriture effective en base, mise à jour à chaque modification du document. Une entrée ne peut donc pas apparaître dans un intervalle déjà traité après le passage de la traçabilité, ce que la date d'événement ne garantirait pas : une opération peut être journalisée avec un `evDateTime` antérieur à son écriture.

7.10.2.13.3.1 Bornes de l'intervalle

Borne inférieure Date de fin (`EndDate`) de la dernière traçabilité OK du même périmètre et de la même version de sécurisation, ce qui assure la continuité du chaînage. Pour la première traçabilité, la borne est l'époque (`LogbookTraceabilityHelper.INITIAL_START_DATE`).

Borne supérieure `now` diminué du délai de temporisation, 300 s par défaut. Les documents du journal sont enrichis au fil des événements. Sécuriser une entrée trop récente reviendrait à figer un document encore en cours d'écriture, dont l'empreinte évoluerait ensuite. La temporisation limite la sécurisation aux documents stabilisés.

Les deux bornes sont incluses (>= et <=). Une entrée dont la date de persistance tombe exactement sur la borne peut donc être sécurisée par deux opérations consécutives. Ce recouvrement est volontaire, un doublon étant préférable à un trou dans la couverture.

7.10.2.13.3.2 Continuité de la couverture

La borne supérieure d'une opération devient la borne inférieure de la suivante. Bout à bout, les périmètres couvrent la totalité du journal sans discontinuité.

Une traçabilité vide, sans aucune entrée dans l'intervalle, ne produit ni archive ni empreinte, et ne peut donc pas servir de borne. La recherche de la borne inférieure des cycles de vie porte pour cette raison sur la dernière traçabilité disposant d'un zip (paramètre `traceabilityWithZipOnly`), et non sur la dernière traçabilité OK.

7.10.2.13.3.3 Réduction du périmètre par le plafond

Le nombre d'entrées retenues par opération est plafonné : `operationTraceabilityMaxEntries` pour le journal des opérations, `lifecycleTraceabilityMaxEntries` pour les cycles de vie, 100 000 par défaut dans les deux cas. Lorsque le plafond est atteint, la borne supérieure est ramenée au `_lastPersistedDate` de la dernière entrée traitée et `MaxEntriesReached` est positionné dans le `TraceabilityEvent`. La traçabilité suivante repart de cette borne.

Le plafond est une limite souple. Une fois atteint, la sélection continue de renvoyer les entrées dont la date de persistance est identique à celle de la dernière entrée retenue, dans la limite d'une réserve de 10 000 entrées (`OVERFLOW_LIMIT`). Un groupe d'entrées partageant la même date de persistance n'est ainsi jamais coupé en deux, la borne de reprise étant cette même date.

La sélection est réalisée par lecture directe des collections MongoDB (`selectRawByLastPersistenceDateInterval`), sans passer par l'index de recherche ni par le DSL, dont la limite de résultats par défaut est de 10 000 documents.

7.10.2.13.3.4 Cloisonnement et limites

La sélection est filtrée sur le tenant. Il y a donc un périmètre, un chaînage et une version de sécurisation par tenant et par périmètre : journal des opérations, cycles de vie des unités, cycles de vie des groupes d'objets.

Le périmètre ne couvre pas :

- les objets binaires, qui ne sont jamais intégrés à l'archive ; seules leurs empreintes le sont, via la traçabilité des cycles de vie de groupes d'objets ;
- les métadonnées pour elles-mêmes ; elles sont couvertes indirectement, par l'empreinte portée dans la ligne du cycle de vie correspondant ;
- le journal des écritures de stockage, qui relève d'un périmètre distinct porté par le module *storage*.

7.10.2.13.4 Contenu de l'archive sécurisée

L'archive est un zip non compressé de cinq fichiers. Les fichiers `traceabilityEvent.json` et `traceabilityStats.json`, présents dans les workflows de cycle de vie, n'en font pas partie : ce sont des fichiers de travail du workspace, dont le contenu est repris dans l'"`evDetData` de l'opération.

7.10.2.13.4.1 data.txt

`data.txt` est le seul fichier porteur de données ; les quatre autres portent la preuve qui s'y rapporte. Il contient une ligne par entrée sécurisée, séparée par `\n`, chaque ligne étant sérialisée par **CanonicalJsonFormatter**.

La sérialisation canonique rend l'empreinte reproductible : clés d'objet triées, aucune indentation ni espacement, échappement restreint à un jeu fixe de caractères, caractères non imprimables en `\uXXXX`, entiers écrits tels quels. Cet encodage ne doit pas évoluer, sous peine de rendre non reproductibles, donc invérifiables, les empreintes des archives antérieures.

L'ordre des lignes est celui de la sélection, par date de persistance croissante. Il est signifiant, chaque ligne étant une feuille de l'arbre de Merkle dans cet ordre.

Le contenu d'une ligne dépend du périmètre.

7.10.2.13.4.2 Cas du journal des opérations

La ligne est le document logbook brut et complet, tel qu'il est stocké en base : aucun champ n'est retiré, aucune valeur n'est transformée. On y trouve :

- les identifiants et le contexte d'appel : `_id`, `evId`, `evParentId`, `evIdProc` (identifiant du processus), `evIdReq` (identifiant de requête) ;
- le typage : `evType`, `evTypeProc` ;
- la date de l'événement : `evDateTime` ;
- le résultat : `outcome`, `outDetail`, `outMessg` ;
- les agents : `agId`, `agIdApp`, `agIdPers`, `agIdExt`, `evIdAppSession` ;
- les objets concernés : `obId`, `obIdReq`, `obIdIn` ;
- les données de détail : `evDetData`, `rightsStatementIdentifier` ;
- le tableau `events`, contenant tous les sous-événements de l'opération, chacun portant les mêmes champs ;
- les champs techniques : `_tenant`, `_v` et `_lastPersistedDate`.

`_lastPersistedDate` étant le champ qui délimite le périmètre, son inclusion dans l'empreinte interdit de déplacer une entrée d'un périmètre à l'autre après sécurisation.

Pour la liste exhaustive des champs et leur correspondance métier (PREMIS), se référer au *Modèle de données* de l'architecture fonctionnelle du logbook.

7.10.2.13.4.3 Cas des cycles de vie

La ligne n'est pas le document de cycle de vie, mais un condensé structuré (**LifeCycleTraceabilitySecureFileObject**). La volumétrie des cycles de vie interdit de les recopier intégralement, et le périmètre doit également couvrir la métadonnée associée ainsi que, pour les groupes d'objets, les objets binaires.

Champ	Contenu
lEvtIdProc	identifiant de l'opération (evIdProc) portée par le cycle de vie
lEvTypeProc	type de processus du dernier événement du cycle de vie
lEvDTime	date du dernier événement du cycle de vie
lfcId	identifiant du cycle de vie, égal à celui de l'unité ou du groupe d'objets
mdType	UNIT ou OBJECTGROUP
version	version (_v) du document de cycle de vie
ltEvtOutcome	outcome du dernier événement du cycle de vie
hLFC	empreinte du document de cycle de vie complet
hLFCEvts	empreinte du seul tableau events du cycle de vie
hMetadata	empreinte de la métadonnée, après retrait des champs calculés
hGlobalFStorage	empreinte de la métadonnée retenue après croisement avec les offres
hGlobalDetails	strategyId et offerIds interrogées ; en cas d'incohérence seulement, dbHash et offerHash (empreinte par offre)
hOGDocsStorage	groupes d'objets uniquement : par objet binaire, id, hObject et hDetails (même structure que hGlobalDetails). Les objets physiques sont ignorés, n'ayant pas de contenu stocké
up	identifiants des parents directs (_up) de l'unité
idOG	unités uniquement : identifiant du groupe d'objets rattaché (_og)

```
{
  "lEvtIdProc": "aeeaaaaaa...", "lEvTypeProc": "INGEST", "lEvDTime": "2026-05-
↪12T09:41:07.213",
  "lfcId": "aeaqaaaaa...", "mdType": "UNIT", "version": 3, "ltEvtOutcome": "OK",
  "hLFC": "...", "hLFCEvts": "...", "hMetadata": "...",
  "hGlobalFStorage": "...",
  "hGlobalDetails": { "strategyId": "default", "offerIds": ["offer-fs-1"] },
  "up": ["aeaqaaaaa..."], "idOG": "aebaaaaaa..."
}
```

hLFC couvre le document de cycle de vie entier, hLFCEvts uniquement ses événements. La présence des deux empreintes permet, lors d'un contrôle, de distinguer une altération du corps du cycle de vie d'une altération de sa liste d'événements.

Les champs calculés sont retirés de la métadonnée avant calcul de hMetadata. Il s'agit des champs recalculables par la plateforme, dont la valeur peut changer sans que l'archive ait été modifiée : champs de graphe (_us, _sps, _graph, _glpd, _min, _max, _uds, _computedInheritedRules, _validComputedInheritedRules, _vups) et champs temporaires ou techniques (_elimination, _av, _acd, _aud, _opts, _batchId, _uploadPath, _errors, _ogInfo) pour les unités ; _sps, _us, _glpd, _av, _acd, _aud, _batchId et _errors pour les groupes d'objets.

Note : Pourquoi retirer ces champs ? Ils sont recalculés par la plateforme et peuvent donc changer sans qu'aucune donnée métier n'ait été modifiée. S'ils étaient inclus dans hMetadata, un simple recalcul de graphe suffirait à faire diverger l'empreinte de la valeur scellée, et donc à faire échouer une vérification sur une archive pourtant intacte.

hGlobalFStorage vaut l'empreinte calculée en base lorsque toutes les offres de la stratégie renvoient la même valeur, et la chaîne littérale INVALID_HASH dès qu'une offre diverge, une empreinte absente étant traitée comme divergente. Une incohérence constatée lors de la sécurisation est ainsi scellée dans l'archive, et pas seulement signalée dans les journaux.

7.10.2.13.4.4 merkleTree.json

Arbre de Merkle complet des lignes de `data.txt`, sérialisé en JSON. Voir *Arbre de Merkle* (page 169) pour sa construction et son format.

7.10.2.13.4.5 token.tsp

Jeton d'horodatage RFC 3161 de la traçabilité courante, encodé en base 64. Il contient la date d'horodatage, l'empreinte horodatée et la chaîne de certificats du signataire. Voir *Horodatage et signature* (page 171).

7.10.2.13.4.6 computing_information.txt

Fichier de propriétés (`clé=valeur`) donnant les quatre valeurs en entrée du calcul de l'empreinte horodatée. Il permet de refaire ce calcul sans accès à la base :

```
currentHash=<racine de Merkle de la traçabilité courante, base 64>
previousTimestampToken=<jeton de la traçabilité précédente, base 64>
previousTimestampTokenMinusOneMonth=<jeton de la traçabilité de M-1, base 64>
previousTimestampTokenMinusOneYear=<jeton de la traçabilité de A-1, base 64>
```

Les trois dernières valeurs sont sérialisées à `null` lorsqu'il n'existe pas de traçabilité correspondante, cas des premières sécurisations d'une chaîne.

7.10.2.13.4.7 additional_information.txt

Fichier de propriétés décrivant le périmètre couvert par l'archive :

```
numberOfElements=<nombre de lignes de data.txt>
startDate=<borne inférieure du périmètre>
endDate=<borne supérieure du périmètre>
securisationVersion=<version de sécurisation, ex. V1>
```

Ces informations sont redondantes avec le `TraceabilityEvent` journalisé. L'archive reste ainsi exploitable indépendamment de la base.

7.10.2.13.5 Arbre de Merkle

Package : **fr.gouv.vitam.common.security.merkletree**

L'arbre de Merkle est un arbre binaire d'empreintes qui condense l'ensemble des entrées sécurisées en une valeur unique, sa racine, tout en permettant de localiser les entrées responsables d'une divergence. C'est cette racine, et non les données, qui est horodatée : la preuve porte sur un condensé de taille constante, quelle que soit la volumétrie sécurisée.

7.10.2.13.5.1 Construction

MerkleTreeAlgo est alimenté en flux, pendant l'écriture de `data.txt`. Chaque ligne, une entrée sérialisée en JSON canonique, est ajoutée par `addLeaf(byte[])`, qui stocke non pas la donnée mais son empreinte. L'algorithme

de hachage est celui de la plateforme (`VitamConfiguration.getDefaultDigestType()`, `SHA512` par défaut).

`generateMerkle()` construit ensuite l'arbre :

1. Complétion (`addPadding`) : le nombre de feuilles est complété jusqu'à la puissance de deux immédiatement supérieure, avec la feuille vide `MerkleTree.EMPTY_LEAF`, de racine de longueur nulle. Un nombre de feuilles déjà égal à une puissance de deux n'est pas modifié.
2. Remontée par paires : à chaque niveau, les feuilles sont consommées deux par deux et l'empreinte du nœud parent est calculée sur la concaténation des racines gauche puis droite. L'opération est répétée jusqu'à ne conserver qu'un seul nœud, dont la racine est la racine de l'arbre.

```
            racine = H(H12 || H34)
              /                   \
    H12 = H(h1 || h2)      H34 = H(h3 || h4)
     /           \            /           \
h1=H(l1)      h2=H(l2)    h3=H(l3)     h4=H(l4)    # l1..l4 : lignes de data.txt
```

L'ordre des feuilles est celui des lignes de `data.txt`. Toute modification, suppression, insertion ou permutation de ligne change la racine.

7.10.2.13.5.2 Propagation d'une altération

L'intérêt de la structure tient à cette propagation : une modification, même d'un seul caractère d'une seule ligne, remonte mécaniquement jusqu'à la racine. En reprenant l'arbre ci-dessus, si la ligne `l2` est altérée en `l2'` :

1. `h2' = H(l2') != h2` : l'empreinte de la feuille change ;
2. `H12' = H(h1 || h2') != H12` : l'empreinte du nœud parent change ;
3. `racine' = H(H12' || H34) != racine` : la racine change.

Le contrôle `CHECK_MERKLE_TREE` recalcule la racine à partir du `data.txt` de l'archive et constate l'écart avec `merkleTree.json` et avec le `Hash` journalisé : l'action est `KO` avec l'erreur `INCORRECT_MERKLE_TREE`. Comme l'arbre entier est conservé, la comparaison branche par branche localise la ou les lignes en cause.

7.10.2.13.5.3 Sérialisation

L'arbre est sérialisé en entier dans `merkleTree.json`, et non seulement sa racine, ce qui permet à un contrôle ultérieur de descendre dans l'arbre pour isoler la branche fautive. Les nœuds fils portent les noms courts `l` et `r` afin de limiter la taille du fichier, la racine étant sérialisée en base 64 :

```
{
  "Root": "eqA0h...=",
  "l": { "Root": "...", "l": { "Root": "..." }, "r": { "Root": "..." } },
  "r": { "Root": "...", "l": { "Root": "..." }, "r": { "Root": "..." } }
}
```

La racine est également reprise, en base 64, dans `computing_information.txt` (`currentHash`) et dans le `TraceabilityEvent` (`Hash`). Elle est donc présente à la fois dans l'archive sécurisée et dans le journal indexé, ce qui rend les deux comparables.

7.10.2.13.5.4 Vérification

VerifyMerkleTreeActionHandler (action `CHECK_MERKLE_TREE` du workflow de contrôle) recalcule l'arbre à partir du `data.txt` extrait de l'archive, avec l'algorithme d'empreinte déclaré dans le `TraceabilityEvent`

(`DigestAlgorithm`) et non avec celui configuré au moment du contrôle, de façon à ce que les anciennes archives restent vérifiables. Le découpage en feuilles se fait sur le séparateur de ligne ; un reliquat sans saut de ligne final est traité comme une dernière feuille.

La racine recalculée est confrontée à deux références, chacune donnant lieu à une sous-action :

- `COMPARE_MERKLE_HASH_WITH_SAVED_HASH` : comparaison avec la racine de `merkleTree.json`, donc avec l'archive elle-même ;
- `COMPARE_MERKLE_HASH_WITH_INDEXED_HASH` : comparaison avec le champ `Hash` du `TraceabilityEvent` journalisé.

Un écart sur l'une des deux comparaisons place l'action en `KO`, avec l'erreur `INCORRECT_MERKLE_TREE` dans le rapport, et pose un drapeau d'erreur qui court-circuite les plugins suivants pour cette archive.

7.10.2.13.6 Horodatage et signature

Packages : **fr.gouv.vitam.common.timestamp**, **fr.gouv.vitam.logbook.common.traceability**

L'horodatage produit un jeton RFC 3161 signé, attestant qu'une empreinte donnée existait à une date donnée. VITAM ne s'appuie pas sur une autorité d'horodatage externe : il tient le rôle de TSA avec un couple clé privée et certificat, stocké dans un keystore PKCS#12 local.

7.10.2.13.6.1 Empreinte horodatée et chaînage

L'empreinte soumise à l'horodatage n'est pas la seule racine de Merkle. **TimeStampService** calcule un condensé sur la concaténation de quatre valeurs, dans cet ordre :

```
byte[] hashDigest = timeStampService.getDigestFrom(
    merkleRootHash,          // racine de l'arbre de la traçabilité courante
    previousTimestampToken,  // jeton de la traçabilité précédente
    previousMonthToken,      // jeton de la traçabilité de M-1
    previousYearToken);      // jeton de la traçabilité de A-1
```

Soit :

```
empreinte horodatée = H( racine de Merkle || jeton J-1 || jeton M-1 || jeton A-1 )
```

Les valeurs nulles sont ignorées, ce qui couvre le cas des premières traçabilités d'une chaîne. L'intégration des jetons antérieurs dans l'empreinte fait que chaque jeton scelle la chaîne qui le précède : une traçabilité passée ne peut pas être réécrite sans invalider les jetons postérieurs. Les trois profondeurs (précédente, M-1, A-1) rendent le chaînage robuste à la perte d'un maillon intermédiaire.

7.10.2.13.6.2 Génération du jeton

TimestampGenerator construit la requête d'horodatage et délègue la signature à une implémentation de l'interface **TimeStampSignature** :

- la requête porte l'OID de l'algorithme d'empreinte (`VitamConfiguration.getDefaultTimestampDigestType()`, `SHA512` par défaut) et l'empreinte à horodater ;
- `setCertReq(true)` demande l'inclusion du certificat signataire dans le jeton, nécessaire à une vérification autonome ultérieure ;
- le *nonce* n'est pas utilisé ; sa mise en place impliquerait d'adapter la vérification (**VerifyTimeStampActionHandler**) ;

- la réponse est validée contre la requête (`TimeStampResponse.validate`) avant retour du jeton encodé.

TimeStampSignatureWithKeystore est l'implémentation utilisée en production :

- chargement d'un keystore PKCS#12, qui doit contenir un seul alias ; dans le cas contraire l'initialisation échoue, la clé de signature à employer étant ambiguë ;
- récupération de la clé privée et de la chaîne de certificats associée ;
- l'algorithme de signature est déduit du couple algorithme d'empreinte et algorithme de clé, sous la forme `<digest>With<algorithmeDeClé>`, par exemple `SHA512WithRSA` ;
- la chaîne de certificats complète est ajoutée au jeton, la politique d'horodatage est portée par l'OID `1.1` et la date d'horodatage (`genTime`) est l'heure courante du serveur.

Le jeton est écrit en base 64 dans `token.tsp` et repris dans le `TraceabilityEvent` (champ `TimeStampToken`). L'événement `OP_SECURISATION_TIMESTAMP`, ou son équivalent pour les cycles de vie, porte le statut de cette étape. Un échec d'horodatage est `FATAL` et interrompt la sécurisation.

7.10.2.13.6.3 Certificats et configuration

La signature est réalisée là où la traçabilité est calculée, donc dans deux composants :

- le logbook, pour le journal des opérations : `p12LogbookFile` et `p12LogbookPassword` dans `logbook.conf` ;
- le worker, pour les cycles de vie (plugin `GENERATE_*_LFC_TRACEABILITY`) : les mêmes clés, lues dans `verify-timestamp.conf`.

Les certificats de timestamping sont produits par la PKI de déploiement à partir de `environments/certs/timestamping/`, qui porte des CA racine et intermédiaire dédiées et les certificats sous `vitam/`. Ils sont convertis en keystores `keystores/timestamping/keystore_<usage>.p12` par `generate_stores.sh`. Le fichier déployé auprès des composants est `keystore_secure-logbook.p12`, son mot de passe étant porté par le *vault* Ansible.

7.10.2.13.6.4 Vérification du jeton

VerifyTimeStampActionHandler (action `VERIFY_TIMESTAMP`) procède en trois temps, chacun matérialisé par une sous-action et un code d'erreur dans le rapport :

`COMPARE_TOKEN_TIMESTAMP` Le jeton contenu dans `token.tsp` est comparé à celui journalisé dans l'"`evDetData` de l'opération. Une divergence signale une incohérence entre l'archive et le journal (`INEQUAL_TIMESTAMP`).

`VALIDATE_TOKEN_TIMESTAMP` Validation cryptographique du jeton :

- extraction du certificat signataire embarqué dans le jeton, sélectionné par l'identifiant du signataire ; son absence donne `MISSING_SIGNING_CERTIFICATE` ;
- vérification de la signature du jeton avec ce certificat ;
- contrôle de l'attribut signé `SigningCertificate` ou `SigningCertificateV2` : l'empreinte du certificat qu'il référence doit correspondre au certificat effectivement utilisé, ce qui interdit la substitution d'un certificat dans le jeton ;
- lorsqu'un truststore est configuré, validation de la confiance accordée au certificat.

`VERIFY_TOKEN_TIMESTAMP` Vérification que le jeton horodate les données de l'archive : la racine de `merkleTree.json` doit être égale au `currentHash` de `computing_information.txt`, puis l'empreinte est recalculée sur les quatre valeurs de chaînage lues dans `computing_information.txt` et comparée au `messageImprintDigest` extrait du jeton. Tout écart donne `INVALID_TIMESTAMP`.

7.10.2.13.6.5 Validation de la confiance

TimestampCertificateValidator évalue le certificat signataire vis-à-vis d'un truststore fourni par l'exploitant. La date de référence est la date de génération du jeton (`genTime`) et non la date du contrôle : un certificat expiré depuis la sécurisation reste valide pour les traçabilités qu'il a signées. Deux contrôles sont enchaînés :

1. validité temporelle du certificat à la date `genTime` ; un certificat non encore valide ou déjà expiré à cette date donne `CERTIFICATE_NOT_VALID_AT_SIGNATURE_DATE` ;
2. confiance : soit le certificat signataire est lui-même présent dans le truststore (confiance directe, cas d'un certificat feuille non auto-signé), soit sa chaîne est validée par PKIX contre les ancres du truststore, avec la révocation désactivée et `genTime` comme date de validation. Un échec donne `UNSUPPORTED_CERTIFICATE`.

Le truststore est optionnel et désactivé par défaut. Il est déclaré par `timestampTrustStoreFile` et `timestampTrustStorePassword` dans `verify-timestamp.conf`, généré par `generate_stores.sh` à partir des certificats déposés dans `environments/certs/timestamping/trusted/`, et activé par la variable Ansible `vitam.worker.timestamp_truststore_enabled`. En l'absence de truststore, la signature du jeton est vérifiée sans ancrage de confiance : un jeton signé par un certificat arbitraire passe le contrôle. Lorsqu'un truststore est configuré, tout échec de chargement est bloquant au démarrage du worker, afin de ne pas dégrader le contrôle sans que ce soit visible.

7.10.2.13.7 Journal des opérations

La sécurisation du journal des opérations est exécutée directement dans le serveur logbook, sans workflow : le job **TraceabilityJob** appelle l'API du logbook, qui invoque **LogbookAdministration**.

7.10.2.13.7.1 Déclenchement et sélection

`generateSecureLogbook(tenantId)` génère le GUID de l'opération, le pose comme identifiant de requête de la session, résout la version de sécurisation du tenant, puis instancie **LogbookOperationTraceabilityHelper**. `generateSecureLogbooks(tenants)` parallélise le traitement sur les tenants, avec un pool dimensionné par `operationTraceabilityThreadPoolSize`, et lève une alerte pour chaque tenant en échec.

`initialize()` détermine les bornes :

- `startDate` = date de fin (`EndDate`) de la dernière traçabilité OK de la même version, ou l'époque s'il n'en existe aucune ;
- `endDate` = `now` diminué du délai de temporisation (`operationTraceabilityTemporizationDelay`, 300 s par défaut).

`isTraceabilityOperationRequired()` décide de l'exécution. La traçabilité est requise si la dernière est plus ancienne que le délai de renouvellement maximal (`operationTraceabilityMaxRenewalDelay`, 690 minutes par défaut), ou s'il existe de nouvelles opérations éligibles dans l'intervalle. Sinon la sécurisation est ignorée et aucune opération n'est produite.

Les entrées sont sélectionnées par date de persistance (`selectOperationsByLastPersistenceDateInterval`) et sécurisées en tant qu'opérations complètes : le document logbook entier, avec l'ensemble de ses événements, est écrit dans `data.txt`.

7.10.2.13.7.2 Plafonnement

La sélection est bornée par `operationTraceabilityMaxEntries`, 100 000 par défaut. Le mécanisme de réduction de la borne haute et de reprise est décrit dans *Réduction du périmètre par le plafond* (page 166). À la différence des cycles de vie, aucun enchaînement automatique n'est déclenché : le reliquat est traité à l'exécution suivante du job.

7.10.2.13.7.3 Événements produits

- `STP_OP_SECURISATION` : événement maître, `STARTED` à l'ouverture, `OK` en fin de traitement, `WARNING` lorsqu'il n'y avait rien à sécuriser ;
- `OP_SECURISATION_TIMESTAMP` : horodatage ;
- `OP_SECURISATION_STORAGE` : dépôt du zip. Le fichier transite par un conteneur workspace nommé d'après l'identifiant de requête, puis est copié en `DataCategory.LOGBOOK` sur la stratégie de stockage par défaut, avant suppression du conteneur.

7.10.2.13.8 LFC Unit et LFC GOT

À la différence du journal des opérations, la sécurisation des cycles de vie passe par un workflow. La volumétrie et les accès aux métadonnées et aux offres nécessitent la distribution du travail sur les workers.

7.10.2.13.8.1 Déclenchement

Le job **TraceabilityLFCJob** (`@DisallowConcurrentExecution`, paramétré par un item `UNIT` ou `OBJECTGROUP`) traite tous les tenants en parallèle et appelle, pour chacun, `traceabilityLfcUnit()` ou `traceabilityLfcObjectGroup()` sur le client logbook, ce qui aboutit à `LogbookLFCAdministration.generateSecureLogbookLFC(guid, lfcTraceabilityType)`.

Cette méthode ne démarre pas si un processus du même workflow est encore en cours. Elle évalue ensuite `isNewTraceabilityRequired` à partir de la dernière traçabilité de la version courante et de la dernière traçabilité disposant d'un zip, une traçabilité vide n'ayant pas de `FileName` et ne pouvant donc pas servir de borne. La traçabilité est requise si :

- c'est la première traçabilité de ce type ; ou
- la dernière est plus ancienne que `lifecycleTraceabilityMaxRenewalDelay` ; ou
- la dernière avait atteint son plafond (`MaxEntriesReached`) ; ou
- de nouveaux cycles de vie existent depuis la borne, à la temporisation près (`lifecycleTraceabilityTemporizationDelay`).

Le workflow est alors démarré : création du conteneur workspace, événement `STARTED`, `initVitamProcess` puis `RESUME`. Les délais de temporisation et le plafond d'entrées sont transmis en paramètres d'exécution.

Le job attend la fin de l'opération en interrogeant le logbook (`checkLifecycleTraceabilityWorkflowStatus`), avec une temporisation doublée à chaque itération, de 1 s à 60 s au maximum. L'attente est abandonnée au bout de 30 minutes, ou dès que le processus est en `PAUSE`. Si le statut retourné indique `maxEntriesReached`, le job relance un tour complet, jusqu'à épuisement du reliquat.

7.10.2.13.8.2 Workflows

Les workflows `LOGBOOK_UNIT_LFC_TRACEABILITY` et `LOGBOOK_OBJECTGROUP_LFC_TRACEABILITY` comportent deux étapes, toutes deux en distribution `REF` :

```
STP_UNIT_LFC_TRACEABILITY (resp. STP_OG_LFC_TRACEABILITY)
  CHECK_CONCURRENT_WORKFLOW_LOCK   # verrou sur l'identifiant de workflow
  PREPARE_UNIT_LFC_TRACEABILITY    # -> lastOperation.json, traceabilityInformation.
↪json,
                                   #    lfcWithMetadata.jsonl
  BUILD_UNIT_LFC_TRACEABILITY      # -> traceabilityData.jsonl, traceabilityStats.json
```

(suite sur la page suivante)

(suite de la page précédente)

```
  GENERATE_UNIT_LFC_TRACEABILITY   # -> traceabilityEvent.json, traceabilityFile.zip

STP_UNIT_LFC_TRACEABILITY_FINALIZATION (non annulable)
  FINALIZE_UNIT_LFC_TRACEABILITY   # dépôt du zip en DataCategory.LOGBOOK
```

`CHECK_CONCURRENT_WORKFLOW_LOCK` garantit qu'une seule sécurisation d'un même type de cycle de vie s'exécute à un instant donné. `PREPARE` sélectionne les cycles de vie à sécuriser et les apparie avec leurs métadonnées. `BUILD` calcule les empreintes et produit une ligne par entrée. `GENERATE` exécute l'algorithme générique, **TraceabilityService** avec **LogbookUnitLifeCycleTraceabilityHelper** ou **LogbookObjectGroupLifeCycleTraceabilityHelper**, et dépose le zip et l'événement dans le workspace. `FINALIZE`, non annulable, transfère le zip du workspace vers les offres et clôt l'opération. Les fichiers intermédiaires transitent en `MEMORY`, sauf `lastOperation.json`, `traceabilityEvent.json` et `traceabilityFile.zip`, portés par le workspace.

7.10.2.13.8.3 Données produites

`PREPARE` produit `lfcWithMetadata.jsonl`, où chaque ligne apparie un cycle de vie brut et sa métadonnée (**LfcMetadataPair**), `traceabilityInformation.json`, qui porte le périmètre retenu (`startDate`, `endDate`, `nbEntries`, `maxEntriesReached`), et `lastOperation.json`, copie de la dernière traçabilité utilisée comme borne.

`BUILD` transforme cet appariement en `traceabilityData.jsonl`, dont chaque ligne est un **LifeCycleTraceabilitySecureFileObject**. C'est ce fichier qui devient le `data.txt` de l'archive. Le détail de ses champs est donné dans *Cas des cycles de vie* (page 167).

7.10.2.13.8.4 Croisement d'empreintes

DigestValidator compare, pour chaque métadonnée et chaque objet, l'empreinte recalculée en base, par sérialisation canonique de la métadonnée accompagnée de son cycle de vie, aux empreintes retournées par chacune des offres de la stratégie :

- toutes les empreintes concordent : comptabilisé `OK` ;
- une partie des offres diverge ou ne renvoie pas d'empreinte : `WARNING`, une alerte est émise et le détail des empreintes (`dbHash`, `offerHashes`) est conservé dans la ligne sécurisée, où il n'est écrit qu'en cas d'incohérence ;
- toutes les offres divergent de la base : l'étape est `KO` et la liste des identifiants fautifs est reportée en `evDetData`.

Le bilan est agrégé dans `traceabilityStats.json` (**TraceabilityStatistics**), puis repris dans le `TraceabilityEvent`.

7.10.2.13.8.5 Plafonnement des cycles de vie

La sécurisation est bornée par `lifecycleTraceabilityMaxEntries`, 100 000 par défaut, transmis au workflow en paramètre d'exécution et appliqué dès l'étape `PREPARE`. Le dépassement remonte jusqu'au job via `MaxEntriesReached`, qui enchaîne alors les exécutions tant que du reliquat subsiste.

7.10.2.13.9 SecurisationVersion

7.10.2.13.9.1 Définition

La version de sécurisation est un marqueur porté par chaque opération de traçabilité, défini par tenant et par périmètre. **VitamConfiguration** conserve trois tables, pour le journal des opérations, les cycles de vie d'unités et les cycles de vie de groupes d'objets, et retourne `DEFAULT_TRACEABILITY_VERSION` (`V1`) pour tout tenant non configuré :

```
String version = VitamConfiguration.getLogbookOperationTraceabilityVersion(tenantId);
String unitVersion = VitamConfiguration.getLfcUnitTraceabilityVersion(tenantId);
String gotVersion = VitamConfiguration.getLfcGotTraceabilityVersion(tenantId);
```

La configuration est portée par **TenantTraceabilityVersionConfiguration**, qui valide le format attendu (`V1`, `V2`, ...).

7.10.2.13.9.2 Stockage

La valeur est écrite dans le journal et dans l'archive sécurisée :

- `evDetData.SecurisationVersion` de l'événement de traçabilité (constante `TraceabilityEvent.SECURISATION_VERSION`), y compris pour les traçabilités vides, où l'"`evDetData` ne contient que ce champ ;
- `securisationVersion=<version>` dans `additional_information.txt`.

Le champ est déclaré dans le *mapping* Elasticsearch du logbook et dans la description DSL de la collection `LogbookOperation`, ce qui le rend interrogeable.

7.10.2.13.9.3 Critère de chaînage

La recherche de la traçabilité précédente est filtrée sur la version. Un changement de version ouvre donc une nouvelle chaîne au lieu de se raccrocher à la précédente. Sont concernés `findLastTraceabilityOperationOK(version)`, `findLastTraceabilityOperationOKBeforeDate(date, version)`, utilisés pour la borne de départ et pour les tampons M-1 et A-1, `findLastLifecycleTraceabilityOperation(eventType, version, traceabilityWithZipOnly)` et la requête équivalente de **LogbookLifeCycleTraceabilityHelper**.

Une clause de compatibilité est prévue pour `V1` dans `findLastLifecycleTraceabilityOperation` : le filtre accepte alors aussi les opérations dépourvues du champ `events.evDetData.SecurisationVersion`, les anciennes traçabilités vides, sans zip, ne le portant pas. Le même principe est appliqué côté audit de chaîne de traçabilité (**LogbookInternalResourceImpl**), où `V1` et l'absence du champ sont traités comme équivalents.

7.10.2.13.9.4 Usage côté vérification

Les traitements de vérification résolvent la version attendue du tenant et ne considèrent que les traçabilités correspondantes :

- **EvidenceService** : recherche de la traçabilité couvrant une date donnée et détermination de la dernière traçabilité de cycle de vie ;
- **ProbativeCreateReportEntry** : recherche de la traçabilité de référence et de la plus proche traçabilité antérieure ;
- **TraceabilityLinkedCheckPreparePlugin** : les opérations dont la version diffère de celle attendue sont écartées de la distribution et signalées comme ignorées.

7.10.2.13.9.5 Configuration

Le déploiement Ansible génère la section `traceabilityVersionConfiguration` de `vitam.conf` pour chaque tenant, avec un défaut à `V1` :

```
traceabilityVersionConfiguration:
  - tenant: 0
    logbookOperation: V1
    lfcUnit: V1
    lfcGot: V1
```

Les valeurs sont surchargeables par tenant via `extendedConfiguration.custom[<tenant>].traceabilityVersion.*`, et globalement via `extendedConfiguration.default.traceabilityVersion.*`.

7.10.2.13.9.6 Portée du champ

Aucune branche de l'algorithme de sécurisation ne dépend de la valeur de la version : quelle qu'elle soit, les mêmes données sont sélectionnées, le même arbre de Merkle est construit et le même horodatage est appliqué. La version intervient comme critère de filtrage et de chaînage décrit ci-dessus, et comme discriminant dans le nom de l'archive (`decorateZipName`, qui préserve le nom historique pour `V1` et rend le nom unique par opération au-delà, ce qui évite un conflit sur l'offre). Le champ est donc conservatoire : il permet une évolution ultérieure du format ou de l'algorithme sans rompre les chaînes existantes.

7.10.2.13.10 Synthèse de la chaîne de preuve

7.10.2.13.10.1 Chaîne de dépendances

Chaque maillon est vérifiable indépendamment, et chacun dépend du précédent : c'est cet enchaînement qui rend l'archive opposable.

```
data.txt (une ligne par entrée, ordre significatif)
    |
    v
CanonicalJsonFormatter --> empreinte par ligne --> MerkleTreeAlgo
                                                        |
                                                        v
                                                racine de Merkle
                                                        |
         +----------------------------------------------+
         v                                              v
  merkleTree.json                        computing_information.txt
  (archive)                              (currentHash + jetons J-1 / M-1 / A-1)
                                                        |
                                                        v
                                       TimeStampService.getDigestFrom(...)
                                                        |
                                                        v
                                 TimestampGenerator (keystore PKCS#12)
                                                        |
                                                        v
                                                   token.tsp
                                                        |
```

(suite sur la page suivante)

(suite de la page précédente)

```
                                                    v
                                TraceabilityEvent (evDetData, indexé)
```

La vérification remonte cette chaîne en sens inverse : recalcul des empreintes de lignes, de l'arbre et de la racine, comparaison à `merkleTree.json` et au `Hash` journalisé (*Vérification* (page 170)), puis recalcul de l'empreinte horodatée et validation du jeton (*Vérification du jeton* (page 172)).

7.10.2.13.10.2 Récapitulatif pas à pas

1. Résoudre la version de sécurisation du tenant et déterminer les bornes `startDate` et `endDate` du périmètre.
2. Sélectionner les entrées stabilisées par `_lastPersistedDate`, dans la limite du plafond et de sa réserve de débordement.
3. Cycles de vie uniquement : apparier les cycles de vie et leurs métadonnées (`PREPARE`), puis croiser les empreintes de la base avec celles de chaque offre via **DigestValidator** (`BUILD`).
4. Sérialiser chaque entrée avec **CanonicalJsonFormatter**.
5. Écrire chaque ligne dans `data.txt` et l'ajouter comme feuille de **MerkleTreeAlgo**.
6. Appeler `generateMerkle()` : complétion des feuilles puis remontée par paires jusqu'à la racine, sérialisée dans `merkleTree.json`.
7. Récupérer les jetons des traçabilités précédente, M-1 et A-1 de la même version, pour le chaînage.
8. Calculer l'empreinte à horodater via **TimeStampService** et conserver ses quatre valeurs d'entrée dans `computing_information.txt`.
9. Produire le jeton RFC 3161 avec **TimestampGenerator** et le keystore PKCS#12, puis l'écrire en base 64 dans `token.tsp`.
10. Écrire `additional_information.txt` (nombre d'entrées, bornes, version de sécurisation).
11. Déposer l'archive en `DataCategory.LOGBOOK` sur les offres, directement pour le journal des opérations, via le workspace et l'étape `FINALIZE` pour les cycles de vie, puis supprimer le fichier temporaire.
12. Journaliser le **TraceabilityEvent** en `evDetData` de l'opération, ce qui la clôt et rend l'archive retrouvable et comparable depuis le journal indexé.

7.10.2.13.11 Vérifications aval

Les contrôles portant sur l'arbre de Merkle et sur le jeton d'horodatage sont décrits plus haut (*Vérification* (page 170) et *Vérification du jeton* (page 172)). Les traitements qui les orchestrent ne sont pas détaillés ici :

- le contrôle de cohérence de la chaîne de traçabilité (workflow `LINKED_CHECK_SECURISATION`, `LinkedCheckTraceability.json`), qui enchaîne récupération et extraction du zip, contrôle des empreintes, `CHECK_MERKLE_TREE` et `VERIFY_TIMESTAMP` ;
- l'audit de cohérence (**EvidenceService**) ;
- le relevé de valeur probante (**ProbativeCreateReportEntry**).

Se référer à la documentation de ces traitements et à *vitam-doc* pour leur description fonctionnelle.

7.11 Metadata

7.11.1 Architecture Fonctionnelle

7.11.1.1 Introduction

7.11.1.2 Généralités

Le rôle de métadata est de :

- Stocker de manière requêtable et raipde les métadonnes des objects.

pour faciliter la gestion des demandes d'accès à la base de données. Le méta data permet de structurer et de formaliser les réquête types ayant pour but de mieux gérer les demandes d'accès aux tables de la base de données. Chaque informations issues de la description du méta data doit être signifier et expliciter son rôle et son impact dans la requête. En clair : Pour chaque champs description du méta data il faut répondre :

1. A quoi ca sert ce champ ?
2. Comment cette information vit dans le cycle de vie de la données du réquête ?
3. Quelle est sa valeur ajoutée face à la demande du client(valuer à considérer pour le client final, pour l'administrateur du système, pour le gestionnaire etc.... Comment cette information vit et et circule dans le workflow du SI).

7.11.2 Architecture technique

7.11.2.1 Introduction

7.11.2.1.1 Présentation

Parent package : **fr.gouv.vitam**
Package proposition : **fr.gouv.vitam.metadata**

7.11.2.1.2 Itération 4

6 sous-modules pour le metadata. Dans metadata (parent).

- vitam-metadata-api : Classes et exception, model communes aux différents modules
- vitam-metadata-builder : module pour creer les objets des réquetes select, update,insert etc..
- vitam-metadata-client : module client pour metadata (units, groupe d'objets ...)
- vitam-metadata-core :
- vitam-metadata-parser : module client pour parser les réqutes Jsons.
- vitam-metadata-rest :

7.11.2.1.3 Modules - packages

metadata
    /metadata-api
        fr.gouv.vitam.api fr.gouv.vitam.api.config fr.gouv.vitam.api.exception fr.gouv.vitam.api.model

/metadata-builder
fr.gouv.vitam.builder.request
fr.gouv.vitam.builder.request.construct
fr.gouv.vitam.builder.request.construct.action
fr.gouv.vitam.builder.request.construct.configuration
fr.gouv.vitam.builder.request.construct.query
fr.gouv.vitam.builder.request.exception

/metadata-client
fr.gouv.vitam.client

/metadata-core
fr.gouv.vitam.core.database.collections
fr.gouv.vitam.core.database.configuration
fr.gouv.vitam.core.utils
/metadata-parser
fr.gouv.vitam.parser.request.construct.query
fr.gouv.vitam.parser.request.parser.action
fr.gouv.vitam.parser.request.parser fr.gouv.vitam.parser.request.parser.query
/metadata-rest
fr.gouv.vitam.metadata.rest

7.11.2.2 Opération

7.11.2.2.1 Présentation

Parent package : **fr.gouv.vitam.api**
Package proposition : **fr.gouv.vitam.metadata.rest**

Module pour le module opération : api / rest.

7.11.2.2.2 Services

7.11.2.2.3 Rest API

URL : http://server/metadata/v1

POST /units -> **POST nouvelle unit et selection d'une liste des units avec une réquête**

GET /status -> **statut du metadata**

7.11.2.3 Metadata-api

7.11.2.3.1 Présentation

*Parent package : * **fr.gouv.vitam.metadata**
Package proposition : **fr.gouv.vitam.metadata.api**

- Le package fr.gouv.vitam.api permet d'interagir avec le moteur de données à travers la description du métadata

pour les opérations : insertUnit, insertObjectGroup, selectUnitsByQuery, selectUnitsById Le format utilisé pour la description du metadonnees : Json.

- Le package fr.gouv.vitam.api.config permet de configurer la connexion de la base de données (Mongo DB)

  en utilisant les paramètres : host database server IP address, le port database server port, le nom de la BDD, le nom de la collection.

- Le parkage fr.gouv.vitam.api.exception gère les exceptions issues des opérations des demandes d'acess à travers de métadata.

  les exceptions gerées sont :

```
MetaDataAlreadyExistException(String message)
MetaDataAlreadyExistException(Throwable cause)
MetaDataAlreadyExistException(String message, Throwable cause)
```

- Le parkage fr.gouv.vitam.api.model permet de la gestion d'interrogation de la base de donnees.

7.11.2.4 Metadata-builder

7.11.2.4.1 Présentation

Parent package : **fr.gouv.vitam.metadata**
Package proposition : **fr.gouv.vitam.builder**

- Le package fr.gouv.vitam.builder.request.construct pour construire dynamiquement d'une requête meta data.

Les opérations proposées sont :

- Delete
- Insert
- Select
- Update

et propose un helper pour la construction de la requete.

- Le package fr.gouv.vitam.builder.request.construct.action pour naviguer, modifier dynamiquement d'une requête en fonction des besoins(Add, pull, push, add).
- Le package fr.gouv.vitam.builder.request.construct.configuration permet de configuer d'une requête dynamique de metadonnes.
- Le package fr.gouv.vitam.builder.request.construct.query permet de regrouper d'un ensemble de requete dynamiquement.
- Le package fr.gouv.vitam.builder.request.exception permet de gérer, détecter les exceptions lors l'utilisation d'une requête dynamique.

7.11.2.5 Operation Client

7.11.2.5.1 Présentation

Parent package : **fr.gouv.vitam.metadata**
Package proposition : **fr.gouv.vitam.metadata.client**

- Le package fr.gouv.vitam.client permet d'adresser et de localiser la requete client.

7.11.2.6 metadata-core

7.11.2.6.1 Présentation

Parent package : **fr.gouv.vitam.metadata**

Package proposition : **fr.gouv.vitam.metadata.core**

Ce package implémente les différentes opérations sur le module métadata (insertUnit, insertObjectGroup, selectUnitsByQuery, selectUnitsById)

7.11.2.6.2 1. Modules et packages

—fr.gouv.vitam.metadata.core.collections : contenant des classes pour gérer les requetes MongoDb

—fr.gouv.vitam.metadata.core.utils

—fr.gouv.vitam.metadata.core

—fr.gouv.vitam.metadata.core.database.configuration

7.11.2.6.3 2. Classes

Dans cette section, nous présentons quelques classes principales dans les modules/packages abordés ci-dessus.

7.11.2.6.3.1 2.1 Class DbRequest

La classe qui permet de gérer les requetes de metadata : la Méthode execRequest(final RequestParserMultiple requestParser, final Result defaultStartSet) permet de parser le query et définir le type d'objet(Unit ou Object Group) afin de gérer et exécuter la requete . Les différents traitements sont l'ajout, l'update et la suppression.

Pour l'update :

- La Méthode lastUpdateFilterProjection(UpdateToMongodb requestToMongodb, Result last)
  Permet de finaliser la requete avec la dernière list de mise à jour en testant sur le type d'objet Unit ou Object group et ajout d'index qui correspond au champ mise à jour.
- La Méthode indexFieldsUpdated(Result last)
  Permet de mettre à jour les indexes liées aux champs modifiés de Units. Fait appel à une méthode qui permet de mettre à jour un ensemble d'entrées dans l'index ElasticSearch en se basant sur un Curseur de résultat.
- La méthode indexFieldsOGUpdated(Result last)
  Permet de mettre à jour les indexes liées aux champs modifiés de Object Group. fait appel à une méthode qui permet de mettre à jour un ensemble d'entrées dans l'index ElasticSearch en se basant sur un Curseur de résultat.

Pour l'insert :

- La Méthode lastInsertFilterProjection(UpdateToMongodb requestToMongodb, Result last)
  Permet de finaliser la requete et ajout d'index qui correspond au champ mise à jour.
- La Méthode insertBulk(InsertToMongodb requestToMongodb, Result result)
  Permet d'insérer les indexes. Fait appel à une méthode qui permet d'insérer un ensemble d'entrées dans l'index ElasticSearch en se basant sur une requête résultat.

Pour le delete :

- La Méthode lastDeleteFilterProjection(UpdateToMongodb requestToMongodb, Result last)
  Permet de finaliser la requete et supprimer d'index en se basant sur la requete.
- La Méthode removeOGIndexFields(Result last)
  Permet de supprimer les indexes des object group existants dans le résultat .
- La Méthode removeUnitIndexFields(Result last)
  Permet de supprimer les indexes des units existants dans le résultat.

7.11.2.6.3.2 2.2 Class ElasticsearchAccessMetadata

- La Méthode updateBulkUnitsEntriesIndexes(MongoCursor<Unit>)
  permet de mettre à jour un ensemble d'entrées dans l'index ElasticSearch en se basant sur un Curseur de résultat.
- La Méthode updateBulkOGEntriesIndexes(MongoCursor<ObjectGroup>)
  permet de mettre à jour un ensemble d'entrées dans l'index ElasticSearch de Object Group en se basant sur un Curseur de résultat.
- La Méthode insertBulkUnitsEntriesIndexes(MongoCursor<Unit> cursor)
  permet d'insérer un ensemble d'entrées dans l'index ElasticSearch de Units en se basant sur un Curseur de résultat.
- La Méthode updateBulkOGEntriesIndexes(MongoCursor<ObjectGroup> cursor)
  permet de mettre à jour un ensemble d'entrées dans l'index ElasticSearch de Object Group en se basant sur un Curseur de résultat.
- La Méthode deleteBulkOGEntriesIndexes(MongoCursor<ObjectGroup> cursor)
  permet de supprimer un ensemble d'entrées dans l'index ElasticSearch de Object Group en se basant sur un Curseur de résultat.
- La Méthode deleteBulkUnitsEntriesIndexes(MongoCursor<Unit> cursor)
  permet de supprimer un ensemble d'entrées dans l'index ElasticSearch de Unit en se basant sur un Curseur de résultat.

7.11.2.6.3.3 2.3 Class MetaDataImpl

- La Méthode insertUnit(JsonNode insertRequest)

permet de rechercher un ensemble d'entrée dans la collection Unit en se basant sur la requête DSL.

- La Méthode insertObjectGroup(JsonNode objectGroupRequest)

permet de mettre à jour un ensemble d'entrée dans l'index ElasticSearch de Object Group en se basant sur un curseur de résultat.

- La Méthode selectUnitsByQuery(JsonNode selectQuery)

permet de rechercher un ensemble d'entrée dans la collection Unit en se basant sur la requête DSL.

- La Méthode selectUnitsById(JsonNode selectQuery, String unitId)

permet de rechercher un ensemble d'entrée dans la collection Unit en se basant sur la requête DSL et Id d'un Unit.

- La Méthode selectObjectGroupById(JsonNode selectQuery, String objectGroupId)

permet de rechercher un ensemble d'entrée dans la collection ObjectGroup en se basant sur la requête DSL et Id d'un Unit.

- La Méthode selectMetadataObject(JsonNode selectQuery, String unitOrObjectGroupId,List<BuilderToken.FILTERARGS> filters)

permet de rechercher un ensemble d'entrée dans les collections Unit et ObjectGroup en se basant sur la requête DSL, Id et le filtre.

7.11.2.6.3.4 2.4 Class UnitNode

- La Méthode buildAncestors(Map<String, UnitSimplified> parentMap, Map<String, UnitNode> allUnitNode, Set<String> rootList)
  permet de construire un graphe DAG pour les objets dans Vitam.

7.11.2.6.3.5 2.5 Class UnitRuleCompute

- La Méthode computeRule()
  permet de calculer les règles de gestion héritées dans un graphe. Chaque node va calculer un UnitInheritedRule grâce à celui de son parent avec ses propres règles de gestions puis concatener les règles (s'il a plusieurs parents).

7.11.2.6.3.6 2.5 Class UnitInheritedRule

- La Méthode createNewInheritedRule(ObjectNode unitManagement, String unitId)
  permet de calculer les règles de gestion héritées en utilisant le règle du parent avec ses propres règles de gestion.
- La Méthode concatRule(UnitInheritedRule parentRule)
  permet de concaténer les règles de gestion héritées de plusieurs parents.

7.11.2.7 metadata-parser

7.11.2.7.1 Présentation

Parent package : **fr.gouv.vitam.metadata**
Package proposition : **fr.gouv.vitam.metadata.parser**

Ce parquet permet de valider la conformité de la requete metadata dynamique.

7.11.2.8 Métadata

7.11.2.8.1 Présentation

Parent package : **fr.gouv.vitam.api**
Package proposition : **fr.gouv.vitam.metadata.rest**

Ce paquet permet de valider les différents paquets. Module hébergeant le support REST et le jar de lancement du service.

7.11.2.8.2 Services

7.11.2.8.3 Rest API

URL Path : http://server/metadata/v1

POST /units : **POST nouvelle unit et sélection d'une liste des units avec une réquête**

GET /units : **GET sélectionne une liste des units avec une requête**

GET /status : **statut du server rest metadata (available/unavailable)**

POST /objectgroups : **Insérer une nouvelle object groups avec une requête DSL**

GET /objectgroups/{id_og} : **avoir un object groups par id avec une requête DSL**

GET /units/{id_unit} : **POST nouvelle unit et sélection d'une liste des units avec une requête**

PUT /units/{id_unit} : **mettre à jour une unit par identifiant**

7.11.2.9 Rest

7.11.2.9.1 Présentation

Package Parent : **fr.gouv.vitam.metadata**
Proposition de package : **fr.gouv.vitam.metadata.rest**

Module hébergeant le support REST et le jar de lancement du service.

7.11.2.9.2 Services

7.12 Processing

7.12.1 architecture-fontionnelle-processing

7.12.1.1 Introduction

7.12.1.1.1 But de cette documentation

L'objectif de cette documentation est d'expliquer l'architecture fonctionnelle de ce module.

7.12.1.1.2 Processing

Mot-clé

- workflow : une processus de traitement des opérations
- paramètre d'exécution : en ensemble des données founi précisé les paramètres d'exécution

Le module processing de VITAM fournit des services qui permet réaliser une chaine des opérations quand il y a une requête depuis le côté client via le service ingest. Il va procéder via plusieurs étapes qui correspondent à des modules de traitement suivant :

- process management
- process engine
- process distributor
- process worker

7.12.1.2 Processing Management

Ce module est pour le but d'organiser l'exécution d'un process de traitement avec les workflows fournis et un ensemble de paramètres passés par le service d'appel (Ingest : traitement des saisies d'archives).

Lors de l'initialisation du processus avec un workflow donné, ProcessManagement crée une instance de ProcessEngine et de StateMachine qui sont fortement liée au ProcessWorkflow avec une cardinalité un-à-un.

Pour chaque ProcessWorkflow, une et une seule machine à état (StateMachine) est rattachée. Une instance d'une machine à état ne peut gérer qu'un seul PocessWorkflow. Pour une instance d'une machine à état, une et une seul instance de ProcessEngine est crée. Un ProcessEngine ne peut être rattaché qu'une et une seule machine à état

Un ProcessManagement peut avoir zéro ou plusieurs ProcessWorkflow

ProcessManagement (0..n)

(1..1) ProcessWorkflow (1..1)

(1..1) StateMachine (1..1) (1..1) ProcessEngine

7.12.1.3 Engine

Ce service permet d'exécuter une étape d'un processus. Il est complètement piloté par une machine à état.

Il peut faire ce qui suit :

- Exécuter une étape d'un processus
  - Initialiser le logbook,
  - Appeler le distributeur pour exécuter l'étape (unzip d'un document, indexer d'un document, sauvegarde d'un document . . .)
  - Finaliser le logbook concernant l'étape.
  - notifier la machine à état sur le résultat de l'exécution de l'étape.

7.12.1.4 Distributor

Le but de ce module est d'attribuer des taches pour chaque ressources disponibles. Le workflow se compose de plusieurs actions à faire et il sera traité par un des workers de traitement disponibles dans la liste.

Le distributor, en plus de lancer les workflow, offre désormais la possibilité aux Workers de s'abonner, se désabonner. Lors d'un abonnement, le Worker est ajouté à une liste de workers (regroupés par famille de worker). Pour un désabonnement, il est supprimé. Pour le moment, les workers ajoutés ne pourront être appelés, cela sera codé dans une autre itération. Un worker par défaut sera ajouté, et utilisé dans cette itération.

Désormais, l'appel du worker se fera via un appel Rest. Le code du Worker est déplacé dans un module à part : Worker.

7.12.1.5 Worker

L'objectif de cette documentation est d'expliquer l'architecture fonctionnelle de ce module.

Ce module lui-même traite une tache/opération précise dans l'ensemble des opérations de workflow. Le worker se compose de plusieurs ActionHandler qui permet de traiter une tâche précis.

Le worker est désormais appelé via du rest. Un client est fourni et permet l'utilisation de l'API Rest mise en place. Un module Worker séparé est mis en place.

7.12.1.6 Process Monitoring

Le but de ce module est de pouvoir monitorer les différentes étapes des différents Workflows.

Une interface a été déterminée et permet les opérations suivantes :

- initOrderedWorkflow : permet l'initialisation d'un Workflow. Le workflow est rattaché à un process, et est composé de steps. La méthode retourne une liste ordonnée de steps avec un id unique.
- updateStepStatus : permet de mettre à jour le statut d'un step. (STARTED, OK, KO, WARNING, FATAL, PAUSED)
- updateStep : permet de mettre à jour les champs elementToProcess et elementProcessed.
- getWorkflowStatus : permet de récupérer les information de workflow par rapport à un process donné.

L'implémentation choisie permet d'enregistrer toutes les informations de workflow dans une HashMap, tout ceci via un singleton. La liste des workflow étant enregistrée dans une ConcurrentHashMap, permettant de gérer les nombreux appels concurrents. Lors de l'initOrderedWorkflow, l'id unique pour chaque step est généré de cette manière :

- {CONTAINER_NAME}_{WORKFLOW_ID}_{QUANTIEME_DU_STEP}_{STEP_NAME}

7.12.2 Architecture Technique

7.12.2.1 Introduction

7.12.2.2 DAT : module processing

Ce document présente l'ensemble de manuel développement concernant le développment du module metadata qui représente le story #70, qui contient :

- modules & parkages
- classes de métiers

7.12.2.2.1 Module et packages

Les principaux modules sont :

- processing-common : contient les méthodes commons : les modèles, les exceptions, SedaUtil, . . .
- processing-distributor : appelle un worker de processus et distribue le workflow. Offre la possibilité au worker de s'enregistrer, se désabonner.
- processing-distributor-client : client de module processing-distributor
- processing-engine : appelle un distributeur de processus
- processing-engine-client : client de module processing-engine
- processing-management : gestion de workflow
- processing-management-client : client de module processing-management

7.12.2.2.2 Modèle

Un modèle a été mis en place pour permettre la remontée et l'agrégation des status des différents item du worflow.

Un état du worflow utilise l'objet **ItemStatus** qui contient : * itemId : l'identifiant de l'item de processus résponsable du status (identifiant de step, handler, transaction, etc) * statusMeter : une liste de nombre de code status (nombre de OK, KO, WARNING, etc) * globalStatus : un status global * une liste de données remontée par l'item du processus (comme messageIdentifier)

Les statuts du processus de workflow utilisent un objet composite **CompositeItemStatus** qui est un **ItemStatus** et contient une Map de statut de workflow de ses sous-items.

Un workflow est défini par un fichier json contenant les steps ainsi que toutes les actions qui doivent être exécutées par les steps. Chaque Step et Action doivent être identifiés par un ID unique qui est également utilisé pour récupérer les messages.

La combinaison d'un état du processus (PAUSE, RUNNING, COMPLETED) et de son statut qui peut être (OK, WARNING, KO, FATAL) nous donne un vue global sur le processus. Le processus peut être en état COMPLETED avec tous les statut possible. Il faut être en état RUNNING ou PAUSE avec le statut de l'exécution des dernières étapes.

7.12.2.2.3 Process Distributor

Le distributor, en plus de lancer les workflow, offre désormais la possibilité aux Workers de s'abonner, se désabonner. Lors d'un abonnement, le Worker est ajouté à une liste de workers (regroupés par famille de worker). Pour un désabonnement, il est supprimé. Pour le moment, les workers ajoutés ne pourront être appelés, cela sera codé dans une autre itération.

A l'heure actuelle voici les méthodes REST proposées :

POST /processing/v1/worker_family/{id_family}/workers/{id_worker}

> -> permet d'enregistrer un nouveau worker pour la famille donnée. -> Une query json est passé en paramètre et correspond à la configuration du worker.

DELETE /processing/v1/worker_family/{id_family}/workers/{id_worker}

> -> permet de désinscrire un worker pour la famille donnée, selon son id.

Dans les itérations suivantes les autres méthodes suivantes seront implémentées :

- liste des familles de worker
- ajouter/mettre à jour/effacer une famille de worker
- statut d'une fammile de worker
- liste des workers d'une famille
- effacer les workers d'une famille
- statut d'un worker
- mise à jour d'un worker

7.12.2.2.4 Parallélisme dans le distributeur

Les parallélismes suivants sont mis en oeuvre dans le distributeur

- Parallélisme dans l'exécution des steps entre plusieurs workflows : celui-ci est géré de manière naturelle sous la forme de plusieurs requêtes (actuellement Java, demain en HTTP) entre le moteur du processing (process-engine) et le distributeur.

- Parallélisme dans l'exécution d'un step pour une distribution de type list vers un même worker. Les principes sont les suivants

  -> Worker : chaque worker associé à un WorkerConfiguration pré-défini. Chaque worker appartient à une famille correspondant à ses fonctions. et il possède aussi une capacité pour gérer plusieurs threads en parallèle, précisé par le paramètre capacity de WorkerCongiguration, et ces paramètres seront initialisés lors du lancement du Worker.

  -> Enregistrement/déenregistrement d'un worker : Le principe est un découplage asynchrone basé sur plusieurs queues de messages bloquantes (BlockingQueue en java) Il y a plusieurs famille de worker et chaque famille lié à une queue de messages bloquantes. Pour l'enregistrement du worker, nous faisons aussi un contrôle pour s'assurer que le worker ne peut s'enregistrer qu'à une famille lui appartenant. Au moment de l'enregistrement, si la queue de la famille n'existe pas encore, elle sera créée.

  -> Opérarations :

  - Lors de l'enregistrement d'un worker (voir section ci-dessus), un thread (cf WorkerManager) est crée et se met en écoute sur la blocking queue (Consommateur) correspondante de la famille.

  Une fois une tâche consommée, s'il a une capacité suffisante (fournie par le worker lors de l'enregistrement), ce thread (WorkerThreadManager) va créer un thread (WorkerThread) pour gérer l'envoi de la demande au Worker ainsi que la gestion de la callback vers le producteur.

  - Lors de distribution d'un step d'un workflow,
  - le distributeur pousse les tâches dans la blockingQueue (Producteur) et garde en mémoire les tâches qui sont en cours
  - La queue n'est qu'un élement de découplage et a donc une taile réduite : le thread de distribution est donc bloqué soit lors de son insertion dans la queue soit en attente que toutes les tâches soient terminées
  - Une callback est exécutée par le consommateur en fin de traitement pour supprimer la tâche terminée des tâches en cours

Le parallélisme entre plusieurs workers sera mis en oeuvre en V1

7.12.2.3 Rangement des objets

7.12.2.3.1 Algorithme

1. Mise à jour du journal de cycle de vie du groupe d'objet
2. Récupération des informations d'objet technique :

1. Récupération du groupe d'objet dans le workspace
2. Parsing du SEDA pour identifier les chemins dans le workspace des objets technique contenus dans le groupe d'objets (à terme il faudra éviter de refaire un parsing SEDA)

3. Pour chaque objet technique :

1. Mise à jour du journal de cycle de vie du groupe d'objet avec le stockage de l'objet
2. Stockage de l'objet
3. Commit du journal de cycle de vie du groupe d'objet avec le stockage de l'objet

4. Commit du journal de cycle de vie du groupe d'objet

7.12.2.4 Vérification de la disponibilité

7.12.2.4.1 Algorithme

1. Calcul de la taille totale des Objets + manifeste SEDA :

1. Récupération du manifeste SEDA depuis le workspace.
2. Parsing du manifeste pour calculer la taille totale des objets techniques contenus.
3. Récupération depuis le Workspace, des informations sur le fichier manifeste SEDA dont sa taille.
4. Calcul de la taille total (manifeste SEDA + objets techniques à stocker).

2. Comparaison capacité stockage VS taille totale
    1. Appel au moteur de stockage pour récupérer un Json contenant les informations de capacité pour un couple tenant/stratégie de stockage donné.
    2. Comparaison entre capacité retournée par le moteur de stockage et taille totale calculé précédemment
        1. Si capacité supérieure Alors Inscription dans logbook operation d'un OK
        2. Si capacité inférieure Alors Inscription dans logbook operation d'un KO, fin du process : « Disponibilité de l'offre de stockage insuffisante »
        3. Si un problème est rencontré (Offres non dispos, Server down, etc…) Alors Inscription dans logbook operation d'un KO, fin du process : « Offre de stockage non disponible »

7.12.2.5 Vérifier SEDA

7.12.2.5.1 Algorithme

1. Vérifier la validation du seda (SedaUtils->checkSedaValidation)
    1. Vérifier l'existance de manifest.xml (SedaUtils->checkExistenceManifest)
    2. Valider manifest.xml en utilisant XSD (ValidationXsdUtils->checkWithXSD et getSchema)
2. **Vérifier le nombre de BinaryDataObject (CheckObjectsNumberActionHandler)**
        - Si le nombre de BinaryDataObject dans manifest.xml n'est pas égal à le nombre dans workspace
        - Lister toutes les objets numériques non référencés (CheckObjectsNumberActionHandler->foundUnreferencedDigitalObject)
3. Récupérer toutes les informations des BinaryDataObject (SedaUtils->getBinaryObjectInfo)
    - En parcourant manifest.xml, récupère les informations des BinaryDataObject
    - En type map(ID de BinaryDataObject, BinaryObjectInfo)
    - BinaryObjectInfo inclut id, uri, version, empreint, type d'empreint …
4. Vérifier les versions de BinaryDataObject
    1. Créer la liste de version de manifest.xml (SedaUtils->manifestVersionList)
    2. **Comparer la liste avec le fichier version.conf (SedaUtils->compareVersionList)**
        - S'il y a la version invalide, stocker dans une liste de version invalide.
        - Si la liste de version invalide n'est pas vide, handler retourne la réponse avec statut « Warning ».
    3. Journalisation de l'action CheckVersion
5. Vérifier les empreintes de BinaryDataObject
    1. Récupération d'empreinte du GUID/objects/SIP/content/<uri_correspondent> (WorkspaceClient->computeObjectDigest)
    2. Créer la liste d'empreinte de manifest.xml
    3. **Comparer les empreintes (SedaUtils->compareDigestMessage)**
        - S'il y a la version invalide, stocker dans une liste de version invalide.
        - Si la liste de version invalide n'est pas vide, handler retourne la réponse avec status « Warning ».
    4. Journalisation de l'action CheckConformity

7.12.2.6 Métriques spécifiques du composant processing

7.12.2.6.1 Besoins

A des fins de monitoring du composant processing un certain nombre de métriques sont intégrées.

- La possibilité d'avoir une vue instantanée des opérations gérées par le composant processing
- La capacité de détecter des opérations dans un état et un statut particulier
- La possibilité de pouvoir filter ces métriques par type d'opération, par état et par statut
- Calculer la durée d'exécution des steps, des tâches
- Tracer le cycle de vie des tâches crées par le distributeur, création, attente d'entrée dans une queue, temps passé dans la queue et durée d'exécution par un worker.
- Le nombre de worker abonnés au distributeur

Un outil de monitoring, à ce jour, prometheus, permet de faire des requêtes sur ces métriques et surtout de lancer des alertes dans les cas suspects nécessitant une intervention rapide.

7.12.2.6.2 Liste des métriques

- `vitam_processing_workflow_operation_total` : Récupère un snapshot de l'ensemble des opérations visible par le composant processing
- `vitam_processing_worker_task_in_queue_total` : Total des tâches dans la queue en attendre d'exécution
- `vitam_processing_worker_current_task_total` : Total des tâches crées par le distributeur et qui sont pas encore terminées. C'est la somme des tâches en attente d'entrer dans la queue + Tâches dans la queue + Tâches en cours d'exécution pour les workers.
- `vitam_processing_worker_registered_total` : Total des worker enregistré dans le distributeur
- `vitam_processing_worker_task_execution_duration_seconds` : C'est une métrique de type Histogram, elle calcule la durée d'exécution d'une tâche du point de vu Distributeur/Worker
- `vitam_processing_worker_task_idle_duration_in_queue_seconds` : C'est une métrique de type Histogram, elle calcule la durée d'attente d'exécution d'une tâche depuis sa création jusqu'a sa prise en charge par un worker.
- `vitam_processing_workflow_step_execution_duration_seconds` : C'est une métrique de type Histogram, elle calcule la durée d'exécution d'une step du point de vu ProcessEngine

7.12.2.6.3 Exploitation des métriques

L'exploitation de ces métriques à des fins de visualisation ou d'alerting est de la responsabilité d'un collecteur externe de métriques. A ce jour, le serveur prometheus avec une bonne configuration permet d'exploiter ces métriques.

Note :

- Veuillez vous référer au manuel de développement pour avoir plus d'information et de détails sur chacune de ces métriques
- Veuillez vous référer à la documentation d'exploitation pour savoir comment exploiter ces métriques, exemple d'utilisation, alerting, et visualisation

---

7.13 Scheduler

7.13.1 Généralités

Le rôle de *scheduler* est de lancer des jobs via des crons qui seront configurable dans le fichier quartz.properties. Il permet aussi de faire la supervision via les *API*.

7.13.2 Architecture Technique

7.13.2.1 Introduction

7.13.2.1.1 Présentation

7.13.2.2 Jobs de logbook VITAM

7.13.2.2.1 Liste des classes implémentant les jobs

TraceabilityLFCJob
TraceabilityJob
TraceabilityAuditJob
ReconstructionOperationJob

7.13.2.2.1.1 TraceabilityLFCJob.java

Ce job permet de faire la sécurisation du journal du cycle de vie des unités archivistiques et des groupes d'objets uniquement sur le site primaire :

- Période d'exécution par défaut pour les unités : 0 35 * * * ?
- Période d'exécution par défaut pour les groupes objets : 0 15 * * * ?

7.13.2.2.1.2 TraceabilityJob.java

Ce job permet de faire la sécurisation du journal des opérations uniquement sur le site primaire :

- Période d'exécution par défaut : 0 05 * * * ?

7.13.2.2.1.3 TraceabilityAuditJob.java

Ce job permet de faire le contrôle de la validité de la sécurisation des journaux uniquement sur le site primaire :

- Période d'exécution par défaut : 0 55 00 * * ?

7.13.2.2.1.4 ReconstructionOperationJob.java

Ce job permet de faire la reconstruction des données portées par le composant logbook uniquement sur le site secondaire (primary_site = false) :

- Période d'exécution par défaut : 0 0/5 * * * ?

7.13.2.3 Jobs de metadata VITAM

7.13.2.3.1 Liste des classes implémentant les jobs

AuditDataConsistencyMongoEsJob
ProcessObsoleteComputedInheritedRulesJob
PurgeDipJob
PurgeSipJob
ReconstructionJob
StoreGraphJob

7.13.2.3.1.1 AuditDataConsistencyMongoEsJob.java

Ce job permet de faire l'audit sur la cohérence de données MongoDB et Elasticsearch * Période d'exécution par défaut : 0 0 0 1 JAN ? 2020

7.13.2.3.1.2 ProcessObsoleteComputedInheritedRulesJob.java

Ce job permet de faire le recalcule des computedInheritedRules pour les units dont les computedInheritedRules sont marquées comme obsolètes. * Période d'exécution par défaut : 0 30 2 * * ?

7.13.2.3.1.3 PurgeDipJob.java

Ce job permet de faire le nettoyage des exports DIPs expirés. * Période d'exécution par défaut : 0 0 * * * ?

7.13.2.3.1.4 PurgeSipJob.java

Ce job permet de faire le nettoyage des exports transfers expirés. * Période d'exécution par défaut : 0 25 2 * * ?

7.13.2.3.1.5 ReconstructionJob.java

Ce job permet de faire la reconstruction des données portées par le composant metadata. * Période d'exécution par défaut : 0 0/5 * * * ?

7.13.2.3.1.6 StoreGraphJob.java

Ce job permet de faire le Log shipping des données graphes portées par le composant metadata. * Période d'exécution par défaut : 0 10/30 * * * ?

7.13.2.4 Jobs de functional-administration VITAM

7.13.2.4.1 Liste des classes implémentant les jobs

ReconstructionAccessionRegisterJob
ReconstructionReferentialJob
ReferentialCreateSymblolicAccessionRegisterJob
RuleManagementAuditJob

7.13.2.4.1.1 ReconstructionAccessionRegisterJob.java

Ce job permet de faire la reconstruction des données concernant AccessionRegisterSymbolic et AccessionRegisterDetail portées par le composant functional-administration uniquement sur le site secondaire :

- Périodicité d'exécution par défaut pour les unités : 0 0/5 * * * ?

7.13.2.4.1.2 ReconstructionReferentialJob.java

Ce job permet de faire la reconstruction des données portées par le composant functional-administration uniquement sur le site secondaire :

- Périodicité d'exécution par défaut : 0 0/5 * * * ?

7.13.2.4.1.3 ReferentialCreateSymblolicAccessionRegisterJob.java

Ce job permet de déclencher une commande qui va calculer le registre des fonds symbolique et les ajoute dans les bases de données uniquement sur le site primaire :

- Périodicité d'exécution par défaut : 0 50 0 * * ?

7.13.2.4.1.4 RuleManagementAuditJob.java

Ce job permet de faire la validation de la cohérence des règles de gestion entre les offres de stockage et les bases de données uniquement sur le site primaire :

- Périodicité d'exécution par défaut : 0 40 * * * ?

7.13.2.5 Jobs de offer VITAM

7.13.2.5.1 Liste des classes implémentant les jobs

OfferLogCompactionJob

7.13.2.5.1.1 OfferLogCompactionJob.java

Ce job permet de faire la compaction technique des journaux des offres de stockage. * Période d'exécution par défaut : 0 40 * * * ?

7.14 Storage

7.14.1 Architecture Technique

7.14.1.1 Introduction

7.14.1.1.1 Présentation

Parent package : **fr.gouv.vitam**
Package proposition : **fr.gouv.vitam.storage**

7.14.1.1.2 Itération 16

4 sous-modules dans Storage (parent).

- storage-driver-api : module décrivant l'interface du driver
- storage-engine : module embarquant la partie core du storage (client et server)
- cas-manager : module embarquant l'offre Vitam (module vitam-offer) ainsi que l'implémentation du driver pour cette offre (cas-manager-drivers) et de son mock pour les tests
- cas-container : module embraquant les implémentations spécifiques de l'offre de stockage, actuellement que l'implémntation swift.

7.14.1.1.3 Modules - packages Storage

storage
    /storage-driver-api
        fr.gouv.vitam.storage.driver

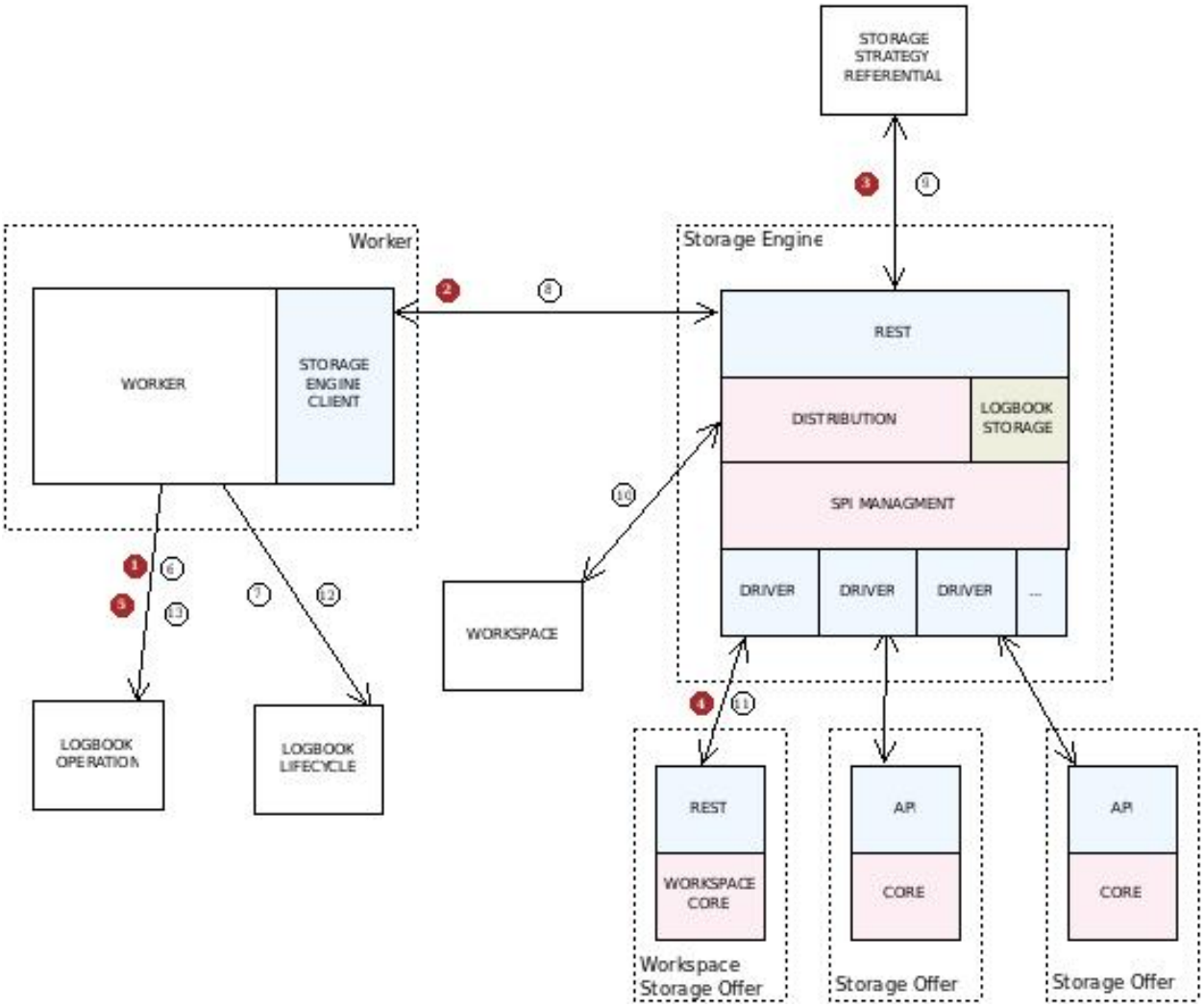
/storage-engine
    /storage-engine-client
        fr.gouv.vitam.storage.engine.client
    /storage-engine-server
        fr.gouv.vitam.storage.engine.server.spi
        fr.gouv.vitam.storage.engine.server.logbook
        fr.gouv.vitam.storage.engine.server.rest
        fr.gouv.vitam.storage.engine.server.distribution
    /storage-engine-common
        fr.gouv.vitam.storage.engine.common

/cas-manager
- /cas-manager-driver
  - /mock-driver
    - fr.gouv.vitam.driver.fake
  - /vitam-driver
    - fr.gouv.vitam.storage.offers.workspace.driver

/cas-container
- /cas-container-filesystem
  - fr.gouv.vitam.cas.container.filesystem
- /cas-container-swift
  - fr.gouv.vitam.cas.container.swift
- /cas-container-utils
  - fr.gouv.vitam.cas.container.utils

7.14.1.2 Architecture générale

7.14.1.2.1 Schéma général



7.14.1.2.2 Workflow du stockage des objets

Le stockage des *objets binaires* contenus dans un *groupe d'objet technique* se fait selon les étapes suivantes :

- Au moment de l'étape de workflow « CheckStorage » (lors de l'étape « Contrôle global entrée (SIP) ») :
  - étape 1 : le **worker** ajoute dans le **journal des opérations** le début de l'opération de vérification de la disponibilité et capacité des offres associées à la *stratégie de stockage* (STARTED)
  - étape 2 : le **worker** appelle le **moteur de stockage** pour faire la vérification de la disponibilité et capacité des offres associées à la *stratégie de stokage*
  - étape 3 : le **moteur de stockage** appelle le **référentiel des stratégies de stockage** pour récupérer le détail de la *statégie de stockage*
  - étape 4 : le **moteur de stockage** appelle les différentes **offres de stockage** définies par la stratégie de stockage pour vérifier leur disponibilité et capacité à travers leur driver correspondant
  - étape 5 : le **worker** ajoute dans le **journal des opérations** le résultat de l'opération de vérification de la disponibilité et capacité pour la stratégie (OK/...)
- Au moment de l'étape de workflow « StoreObjects » (lors de l'étape « Rangement des objets ») :
  - étape 6 : le **worker** ajoute dans le **journal des opérations** le début de l'opération de stockage du *groupe d'objet technique* (STARTED)
  - pour chaque objet binaire du *groupe d'objet technique* :
    - étape 7 : le **worker** met à jour le **journal du cycle de vie** de l'objet (STARTED)
    - étape 8 : le **worker** appelle le **moteur de stockage** pour envoyer l'objet dont l'identifiant est donné en suivant la *stratégie de stockage* donnée
    - étape 9 : le **moteur de stockage** appelle le **référentiel des stratégies de stockage** pour récupérer les détail de la *stratégie de stockage*
    - étape 10 : le **moteur de stockage** récupère l'objet binaire dans le **workspace**
    - étape 11 : le **moteur de stockage** envoi l'objet binaire dans les **offres de stockage** définies par la *stratégie de stockage* à travers leur driver correspondant
    - étape 12 : le **worker** met à jour le **journal du cycle de vie** de l'objet (OK/...)
  - étape 13 : le **worker** ajoute dans le **journal des opérations** la fin de l'opération de stockage du *groupe d'objet technique* (OK/...)

7.14.1.2.3 Itération 6

Limites :

- le **référentiel des stratégies de stockage** n'est pas encore implémenté, de ce fait la *stratégie de stockage* est définie de manière statique
- seule l'"**offre de stockage** utilisant une partie du module workspace est disponible
- la vérification de la disponibilité n'est pas encore implémenté.

7.14.1.2.4 Itération 7

Implémentation de la disponibilité / capacité.

Limites :

- Une seule offre, ainsi la logique est simplifiée au niveau du distributeur qui ne gère alors pas le multi-offres
- La gestion des erreurs est très basique, il serait certainement intéressant de gérer ces erreurs plus finement

7.14.1.2.5 Itération 13

Mise en place du multi-offres.

La stratégie prend maintenant en compte le nombre de copie et les offres qui sont déclarées. Une limite est qu'il faut autant d'offre que de copie.

Dans cette version, le moteur de stockage est séquentiel, il récupère l'objet sur le workspace et l'envoi à la première offre, puis il récupère à nouveau l'objet sur le workspace et l'envoi à l'offre suivante et ainsi de suite.

7.14.1.2.6 Itération 14

Implémentation multi-thread

Dans cette version la distribution du moteur de stockage se charge d'envoyer l'objet issu du workspace en parallèle aux différentes offres. L'objet est récupéré sur le workspace et est « copié » n fois, n étant le nombre de copie à faire. Chacune de ces copies est envoyée à une offre au travers de threads.

L'objet n'est pas tout à fait copié. Il passe au travers d'un **tee** qui crée autant de buffers que de copies. Chacun des buffers est rempli, puis lu en parallèle. Dès que tous les buffers sont vidés, ils sont tous réalimenté jusqu'à ce qu'il n'y ait plus rien à transmettre. Cela signifie que le tee est bloquant. Si un buffer n'est pas vidé les autres attendent potentiellement indéfinement s'il n'y a pas de timeout.

Il n'y a pas de vrai pool de threads dans cette version.

7.14.1.2.7 Itération 16

- Revue de l'architecture golbale du stockage. Mise en place du CAS MANAGER et du CAS CONTAINER.
- Refactoring des éléments communs entre les offres et le workspace. Mise en place d'une implémentation workspace spécifique de stockage en mode filesystem

7.14.1.2.8 R12

Travaux pour la gestion du multi-stratégie dans VITAM : modifications du module storage-engine

- Refactoring pour ajouter le header *X-Strategy-Id* sur les point d'API ne les déclarant pas (débuté en R11)
- Ajout de la configuration multi-stratégie et de sa gestion dans le module

Hors du module storage-engine, possibilité d'utiliser d'autre stratégies que celle de plateforme VITAM d'identifiant *default*.

7.14.1.3 Storage Driver

7.14.1.3.1 Présentation

Parent package : **fr.gouv.vitam.storage**
Package proposition : **fr.gouv.vitam.storage.driver**

Ce module définit l'API « Driver » que doivent implémenter les fournisseurs d'offres de stockage. Un driver peut être assimilé à un module « client » qui permet de dialoguer avec une offre de stockage distante et qui satisfait un contrat de service défini dans l'interface.

7.14.1.3.2 Architecture

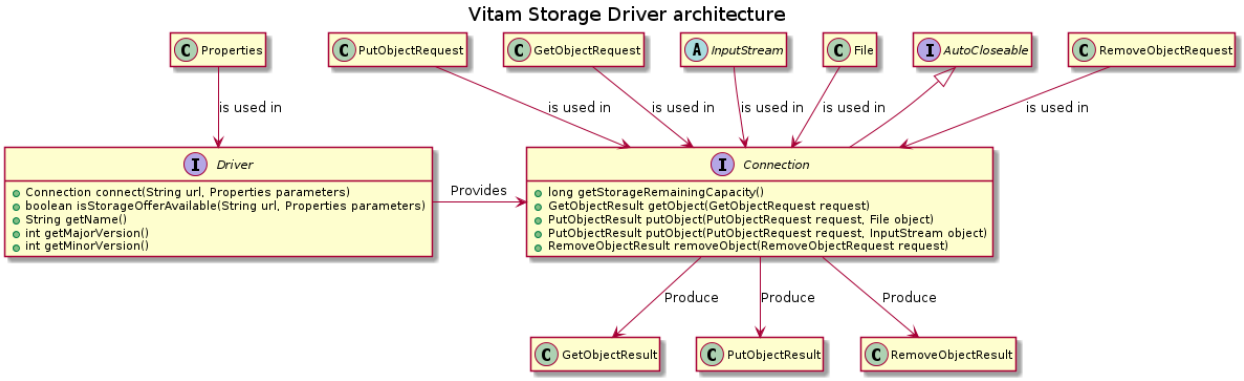
Pour permettre au moteur d'exécution de dialoguer avec un service d'offre de stockage, deux interfaces doivent être implémentées par le fournisseur d'offre :

- **Driver** Objet technique responsable d'établir une connexion avec le service de stockage en fonction des paramètres qui lui sont fournis. C'est aussi lui qui est responsable de déterminer si le service est disponible ou non.
- **Connection** L'établissement d'une connexion au service distant via le driver produit un object **Connection** qui est lié à un contexte d'exécution précis.

  En effet, les paramètres initiaux utilisés pour l'établissement de la connexion, et donc l'instanciation d'un objet **Connection**, ne peuvent être modifiés sur l'objet **Connection**.

  Par exemple, l'URL de base du service ne peut être modifiée. S'il y a besoin d'une connexion vers un autre serveur, ou en tant qu'utilisateur différent, il faut créer une nouvelle connexion en réutilisant le driver.

Le diagramme suivant décrit les relations entre ces 2 interfaces et les objets connexes utilisés dans le cadre de requêtes :



7.14.1.3.3 Pour aller plus loin

Certaines notions seront implémentées plus tard telles que :

- **Thread pool :** Mettre en place un mécanisme de limitation du nombre de thread concurents utilisant des drivers.
- **Connection pool :** Bien que proche du premier point, celui-ci est à mettre en place au niveau Driver dirctement. En effet, le principe est de permettre la configuration (et donc la limitation) du nombre de connexions concurrentes faites par un même Driver pour une offre donnée.
- **Extension des services :** L'interface driver (et l'interface **Connection** associée) a pour vocation de définir les services minimums que doit assurer l'offre de stockage distante. L'interface driver pourra donc évoluer pour augmenter la finesse ou le nombre de services que l'offre doit assurer pour être compatible avec Vitam.

7.14.1.4 Storage Engine

7.14.1.4.1 Présentation

Parent package : **fr.gouv.vitam.storage**
Package proposition : **fr.gouv.vitam.storage.engine**

Module embarquant la partie core du storage (client et server).

7.14.1.4.1.1 Services

De manière générale, pour le Storage, les méthodes utilisées sont les suivantes :

- GET : pour l'équivalent du « Select ».
- POST : **sans** X-Http-Method-Override : GET dans le Header, pour faire un insert.
- POST : **avec** X-Http-Method-Override : GET dans le Header, pour faire un select (avec Body).
- PUT : pour les mises à jour de Units et ObjectGroups.
- DELETE : pour effacer des métadonnées, des objects, des units, des journaux ou bien des containers.
- HEAD : pour les tests d'existence.

7.14.1.4.1.2 Rest API

7.14.1.4.1.3 URI d'appel

http://server/storage/v1

7.14.1.4.1.4 Headers

Plusieurs informations sont nécessaires dans la partie header :

- X-Strategy-Id : Stratégie pour Offres de stockage et Copies (conservation).
- X-Tenant-Id (obligatoire pour toute requête) : id du tenant. Cette information sera utilisée dans toutes les requêtes pour déterminer sur quel tenant se baser.
- X-Request-Id : l'identifiant unique de la requête.
- Accept : Permet de spécifier si un résultat doit contenir uniquement des métadonnées ("application/json"), un DIP complet (un ZIP contenant les métadonnées et les objets) ou seulement des Objects avec un contenu binaire ("application/octet-stream").
- X-ObjectGroup-Id : Id de l'ObjectGroup
- X-Units : Ids des Units parents
- X-Caller-Id : Id du service demandeur

7.14.1.4.1.5 Méthodes

HEAD / -> **Permet d'accéder aux informations d'un container.**
POST / -> avec header X-Http-Method-Override : GET **Permet d'accéder aux informations d'un container.**
POST / -> **Permet de créer un nouveau container (nouveau tenant).**
DELETE / -> **Permet d'effacer un Container (si vide).**
HEAD / -> **Permet de tester l'existence du Container + retourne état et capacité occupée + restante**

GET /objects -> **Liste du contenu binaire pour ce tenant.**
POST /objects -> avec header X-Http-Method-Override : GET **Liste du contenu binaire pour ce tenant.**
GET /objects/{id_object} -> **Permet de lire un Object.**
POST /objects/{id_object} -> avec header X-Http-Method-Override : GET **Permet de lire un Object.**
POST /objects/{id_object} -> **Permet de créer un nouveau Object.**
DELETE /objects/{id_object} -> **Permet de détruire un Object.**

HEAD /objects/{id_object} -> **Permet d'obtenir des informations sur un Object.**

GET /logbooks -> **Liste du contenu d'une collection.**
POST /logbooks -> avec header X-Http-Method-Override : GET **Liste du contenu d'une collection.**
GET /logbooks/{id_logbook} -> **Permet de lire un Journal.**
POST /logbooks/{id_logbook} -> avec header X-Http-Method-Override : GET **Permet de lire un Journal.**
POST /logbooks/{id_logbook} -> **Permet de créer un nouveau Journal.**
DELETE /logbooks/{id_logbook} -> **Permet de détruire un Journal.**
HEAD /logbooks/{id_logbook} -> **Permet d'obtenir des informations sur un Journal.**

GET /units -> **Liste du contenu d'une collection.**
POST /units -> avec header X-Http-Method-Override : GET **Liste du contenu d'une collection.**
GET /units/{id_md} -> **Permet de lire un Unit Metadata.**
POST /units/{id_md} -> avec header X-Http-Method-Override : GET **Permet de lire un Unit Metadata.**
POST /units/{id_md} -> **Permet de créer un nouveau Unit Metadata.**
PUT /units/{id_md} -> **Permet de mettre à jour un Unit Metadata (404 si non pré-existant).**
DELETE /units/{id_md} -> **Permet de détruire un Unit Metadata.**
HEAD /units/{id_md} -> **Permet d'obtenir des informations sur un Unit Metadata.**

GET /objectgroups -> **Liste du contenu d'une collection.**
POST /objectgroups -> avec header X-Http-Method-Override : GET **Liste du contenu d'une collection.**
GET /objectgroups/{id_md} -> **Permet de lire un ObjectGroup Metadata.**
POST /objectgroups/{id_md} -> avec header X-Http-Method-Override : GET **Permet de lire un ObjectGroup Metadata.**
POST /objectgroups/{id_md} -> **Permet de créer un nouveau ObjectGroup Metadata.**
PUT /objectgroups/{id_md} -> **Permet de mettre à jour un ObjectGroup Metadata (404 si non pré-existant).**
DELETE /objectgroups/{id_md} -> **Permet de détruire un ObjectGroup Metadata.**
HEAD /objectgroups/{id_md} -> **Permet d'obtenir des informations sur un ObjectGroup Metadata.**

GET /status -> **statut du storage**

7.14.1.4.1.6 Distribution

Le distributeur (module distribution) est en charge de décider selon la stratégie de stockage dans quelles offres doit être stocké un objet binaire.

Avant tout, le moteur de stockage récupère le binaire sur le workspace et le démultplie via un tee autant de fois que de copies à réaliser. Pour chaque offre de stockage contenue dans la stratégie le distributeur demande au SPI DriverManager le driver associé. Le distributeur instancie alors pour chaque offre un nouveau thread qui va se charger du transfert vers chacune des offres. Dans chaque thread le driver associé à l'offre est utilisé pour le transfert.

Les thread font un retour OK ou KO. Pour chaque offre en KO, une nouvelle tentative de transfert est faite, jusqu'à trois tentatives. Si encore une offre est en KO après trois tentatives (retry), les binaires déposés sur les offres OK sont supprimés (rollback).

Le distributeur gère la mise à jour du journal des écritures du storage liée à l'opération de stockage d'un objet binaire dans une offre. Toutes les tentatives y sont répertoriées pour chaque offre.

D'un point de vue séquentiel :

- Lors d'un appel de type POST /objects/{id_object} pour stocker un nouvel objet, le service est appelé :

1. Il vérifie les paramètres d'entrée (nullité et cohérence simple)
2. Il récupère la stratégie associée à l'ID fourni
3. Regarde uniquement la partie « offres chaudes »
4. Récupère le fichier sur le workspace
5. Pour chaque offre chaude :
    1. Récupération du Driver associé s'il existe (sinon remontée d'une exception technique)
    2. Instancie un thread et dans ce trhead : 1. Récupération des paramètres de l'offre : url du service, paramètres additionels 2. Tentative de connection à l'offre et d'upload de l'objet 3. Comparaison du digest hash renvoyé par l'offre avec le digest calculé à la volée lors de l'envoi du stream à l'offre 4. Retour vers le distributeur du résultat (OK ou KO)
    3. Stockage du résultat de l'upload dans une map temporaire contenant le résultat de l'upload sur chaque offre
6. Pour chaque offre KO, un nouvelle tentative est faite (jusqu'à trois)
7. Si tout est OK, génération d'une réponse sérialisable, en mode "succès" si **tous** les drivers ont correctement stocker l'objet.
   Si une offre au moins est KO, suppression des binaires sur les offres en succès et renvoie une exception

7.14.1.4.1.7 DriverManager : SPI

Service permettant d'ajouter ou de supprimer des drivers d'offre.
Le driver (son interface) est défini dans *Storage Driver* (page 198).

Les différents drivers sont chargés via le ServiceLoader de la JDK puis leurs instances sont stockées dans une liste. Cela permet ensuite de configurer les offres sur les différentes instances de driver en passant par une MAP dont la clef est l'identifant de l'offre, la valeur est le driver instancié dans la liste (une référence à ce driver donc, retrouvé par son nom (getName())).

Le distributeur va alors demander au DriverManager le driver correspondant à l'offre définie dans la stratégie afin de réaliser les opérations de stockage.

7.14.1.4.1.8 Principe

Le driver à ajouter doit implémenter l'interface définie. Dans son jar, il faut donc retrouver l'implémentation du driver ainsi que le fichier permettant au ServiceLoader de fonctionner. Ce fichier **DOIT** se trouver dans les resources, sous META-INF/services (principe du ServiceLoader de la JDK). Son nom est l'interface implémentée par le driver précédé de son package.

Exemple :

```
samples/fr.gouv.vitam.storage.driver.Driver
```

Où VitamDriver est l'interface implémentée.

Son contenu est le nom de la classe qui implémente l'interface (qui est le nom du fichier) précédé de son package.

Exemple :

```
mon.package.ou.se.trouve.mon.driver.VitameDriverImpl
```

Où VitamDriverImpl est l'implémentation du driver.

Voici le fichier : `fr.gouv.vitam.storage.driver.Driver`

Le jar sera déposé via une interface graphique dans un répertoire défini dans le fichier de configuration driver-location.conf avec la clef **driverLocation**. Actuellement il faut le déposer manuellement.
Le paramétrage des offres se fera également via une interface graphique.

Cependant, il faut pouvoir redémarrer Vitam sans perdre l'association driver / offre ou démarrer Vitam avec des drivers et des offres par défaut. Pour se faire, il faut persister la configuration.

7.14.1.4.1.9 Persistance

On s'appuie sur une interface offrant différentes méthodes afin de récupérer les offres à partir d'un nom de driver, persister la configuration... Cela permet demain de changer la stratégie de persistance sans avoir à modifier le code du SPI.

```
public interface DriverMapper {
    List<String> getOffersFor(String driverName) throws StorageException;
    void addOfferTo(String offerId, String driverName) throws StorageException;
    void addOffersTo(List<String> offersIdsToAdd, String driverName) throws␣
↪StorageException;
    void removeOfferTo(String offerId, String driverName) throws StorageException;
    void removeOffersTo(List<String> offersIdsToRemove, String driverName) throws␣
↪StorageException;
}
```

Dans un premier temps, l'implémentation du mapper se fera en passant par un fichier. Dans son implémentation actuelle, le *DriverMapper* a besoin d'un fichier de configuration, `driver-mapping.conf`. Ici, il permet de définir l'emplacement où seront enregistrés les fichiers permettant la persistance via la clef **driverMappingPath**. Une autre clef est nécessaire afin de définir le délimiteur dans ce fichier via la clef **delimiter**, le principe étant de mettre en place un fichier par driver comme un fichier CSV, les offres étant séparées par ce délimiteur.

7.14.1.5 Storage Engine Client

7.14.1.5.1 Présentation

Parent package : **fr.gouv.vitam.storage.engine**
Package proposition : **fr.gouv.vitam.storage.engine.client**

Sous-module du storage engine embarquant le storage engine client.

Ce module permet la discussion entre le worker et le moteur de stockage.

7.14.1.6 Storage Offers

7.14.1.6.1 Présentation

Parent package : **fr.gouv.vitam.storage**
Package proposition : **fr.gouv.vitam.storage.offers**

Module embarquant les différentes offres de stockage Vitam ainsi que leur drivers associés.

Actuellement, ce module embarque :

- une seule offre de stockage, appelée vitam-offer, qui supporte plusieurs types de persistance : système de fichiers, Swift, S3 ou sur bandes magnétiques.
- un seul driver (utilisé par storage-offer-default) appelé vitam-driver. Il permet d'utiliser l'offre par défaut qu'elle soit en mode swift, s3, système de fichiers ou sur bandes magnétiques.
- Il est possible, grace à plusieurs instance de l'offre par défaut d'avoir un stockage multi offres. Il existe une

limite, au sein de la meme JVM, il n'est possible de n'avoir qu'une seul offre d'un seul type.

7.14.1.7 Vitam Offer

7.14.1.7.1 Présentation

Parent package : **fr.gouv.vitam.storage.offers**
Package proposition : **fr.gouv.vitam.storage.offers.workspace**

Module embarquant l'offre de stockage Vitam utilisant une partie du workspace. Utilisation du terme worskpace dans les packages car le terme default est réservé.

L'offre de stockage workspace est séparé en deux parties :

- le serveur de l'offre de stockage par défaut
- l'implémentation du driver associé à l'offre de stockage par défaut

Dans l'offre, tout les objets binaires sont stockés dans des conteneur définis par : {type}_{tenant}. Un objet binaire est lui définit par son identifiant ET son conteneur.

7.14.1.7.2 Driver

Objet technique responsable d'établir une connexion avec le service de stockage en fonction des paramètres qui lui sont fournis. C'est aussi lui qui est responsable de déterminer si le service est disponible ou non. La méthode connect, permet de récupérer un objet Connection afin de pouvoir effectuer des actions sur l'offre de stockage.

7.14.1.7.3 Serveur

7.14.1.7.3.1 Description

Les fonctionnalités sont :

- récupérer la capacité et disponibilité de l'offre
- envoyer un objet

- récupérer un objet
- tester l'existence d'un objet
- récupérer l'empreinte d'un objet
- compter le nombre d'objets d'un conteneur
- contrôler un objet pour valider son transfert
- supprimer un objet

7.14.1.7.3.2 REST

7.14.1.7.3.3 Description

L'API REST, trois header spécifiques sont définis :

- X-Tenant-Id : l'identifiant du Tenant
- X-Type : permets de préciser le résultat attendu pour la recupération de l'objet
  - DATA : l'objet en lui-même (valeur par défaut)
  - DIGEST : empreinte de l'objet

Les réponses en erreur définies par l'API Vitam sont respéctées (400, 401, 404, etc)

7.14.1.7.3.4 REST API

HEAD /

- description : recupération des informations de l'offre
- response :
  - code : 200
  - contenu : information sur l'offre (capacité, disponibilité, ...)

GET /count/{type}/

- description : compter le nombre d'objet d'un conteneur de l'offre
- headers :
  - X-Tenant-Id : id du tenant
- path :
  - {type} : le type permettant d'identifier un conteneur (unit/report/logbook/etc, se basant sur une enum)
- response :
  - code : 200
  - contenu : le nombre d'objets binaires (hors répertoires)

GET /objects/{id}

- description : recupération sur l'offre d'un objet ou de son empreinte
- headers :
  - X-Type : DATA / DIGEST
  - X-Tenant-Id : id du tenant
- path :
  - {id} : path de l'objet
- response :
  - code : 200

    - contenu : data ou empreinte de l'objet

GET /objects/{type}/{id :.+}/check

- description : vérification d'un objet
- headers :
    - X-Type : DATA / DIGEST
    - X-Tenant-Id : id du tenant
- path :
    - {id} : path de l'objet
    - {type} : le type permettant d'identifier un conteneur (unit/report/logbook/etc, se basant sur une enum)
- response :
    - code : 200
    - contenu : un boolean indiquant si le digest de l'objet correspond ou non

PUT /objects/{type}/{id}

- description : écriture d'un objet sur l'offre
- headers :
    - X-Tenant-Id : id du tenant
    - Vitam-Content-Length : Taille de l'objet
    - X-digest-algorithm : Algorithme de hash utilisé pour vérifier l'empreinte de l'objet
- path :
    - {id} : id de l'objet
    - {type} : le type (unit/objectgroup/logbook/etc, se basant sur une enum)
- body :
    - flux : data ou digest
- response :
    - code : 201
    - contenu : un json avec le digest de l'objet et sa taille.

HEAD /objects/{id}

- description : existance de l'objet sur l'offre
- headers :
    - X-Tenant-Id : id du tenant
- path :
    - {id} : id de l'objet
- response :
    - code : 204

DELETE /objects/{type}/{id}

- description : suppression d'un objet de l'offre
- headers :
    - X-Tenant-Id : id du tenant
    - X-Type : DATA / DIGEST
- path :
    - {id} : id de l'objet
    - {type} : le type permettant d'identifier un conteneur (unit/report/logbook/etc, se basant sur une enum)
- response :

    - code : 200
    - contenu : l'id de l'objet supprimé + le statut

GET /status

- description : état du serveur
- reponse :
    - code : 200
    - contenu : statut

7.14.1.8 Métriques spécifiques du composant storage-engine

7.14.1.8.1 Besoins

A des fins de monitoring du composant storage-engine pour estimer les données qui traverse ce composant depuis et vers les offres de stockage, un certain nombre de métriques sont intégrées.

- La de calculer le taux moyen des données téléversées dans les offres de stockage
- La de calculer le taux moyen des données téléchargées depuis les offres de stockage

Il est important d'avoir des filtres sur les critères suivant :

- tenant
- strategy de stockage
- la carégorie de la donnée
- l'identifiant de l'offre de stockage.
- optionnellement, l'origin de la demande et aussi le numéro d'essai pour mesurer s'il y a des erreurs.

Un outil de monitoring, à ce jour, prometheus, permet de faire des requêtes sur ces métriques et surtout de lancer des alertes dans les cas suspects nécessitant une intervention rapide.

7.14.1.8.2 Liste des métriques

- `vitam_storage_download_size_bytes` : Données en octets téléchargées par le composant *vitam-storage-engine* depuis les offres de stockages.
- `vitam_storage_upload_size_bytes` : Données en octets téléversées par le composant *vitam-storage-engine* vers les offres de stockages.

7.14.1.8.3 Exploitation des métriques

L'exploitation de ces métriques à des fins de visualisation ou d'alerting est de la responsabilité d'un collecteur externe de métriques. A ce jour, le serveur prometheus avec une bonne configuration permet d'exploiter ces métriques.

Note :

- Veuillez vous référer au manuel de développement pour avoir plus d'information et de détails sur chacune de ces métriques
- Veuillez vous référer à la documentation d'exploitation pour savoir comment exploiter ces métriques, exemple d'utilisation, alerting, et visualisation

---

7.15 Worker

7.15.1 architecture-fontionnelle-processing

7.15.1.1 Introduction

7.15.1.1.1 But de cette documentation

L'objectif de cette documentation est d'expliquer l'architecture fonctionnelle de ce module.

7.15.1.1.2 Worker

Le module Worker de VITAM fournit des services qui permettent de réaliser une chaine des opérations quand il y a une requête depuis le côté client via le service ingest.

7.15.1.2 Worker

Ce module traite une étape précise dans l'ensemble des opérations de workflow. Pour chaque étape, une liste d'actions est lancée via des ActionHandler.

Le worker offre une API Rest permettant (via un client spécifique) de lancer les différentes méthodes désirées (pour l'instant submitStep est la seule méthode disponible).

7.15.1.3 notification-atr-ok

Cette section présente le processus pour notifier le résultat d'un téléchargement d'un document SIP.

Lorsque le SIP passe toutes les étapes du workflow d'entrée avec succès, Vitam lui envoie une notification au service versant dans ce cas. La procédure se compose deux étapes : la génération d'une notification et le téléchargement de la notification

- Génération et stockage de la notification

A partir de SIP versant, nous devrons générer une réponse en format XML en utilisant les informations précisées dans le SEDA et l'information sur le workflow exécuté. La réponse doit être validé par le schéma XSD pré-défini pour le format de la réponse de notification. Cette notification sera sauvegardé dans l'espace de stockage.

- Téléchargement de la notification

Dans l'interface de téléchargement du SIP, lors d'un UPLOAD succès, nous trouvons le status OK de UPLOAD SIP et en bas nous trouvons aussi un lien pour télécharger la notification générée dans l'étape précédente.

7.15.1.4 notification-atr-ko

Cette section présente le processus pour notifier le résultat négatif d'un téléchargement d'un document SIP.

Lorsque le SIP provoque une erreur au niveau du workflow d'entrée, une étape finale est exécutée. Elle a pour but la génération d'une notification d'erreur au service versant.

La procédure se compose deux étapes : la génération d'une notification et le téléchargement de la notification

- Génération et stockage de la notification :

A partir du SIP soumis par le service versant, nous devrons générer une réponse en format XML en utilisant les informations précisées dans le SEDA et l'information sur le workflow exécuté. La réponse doit être validé par le schéma XSD pré-défini pour le format de la réponse de notification. Cette notification sera sauvegardé dans l'espace de stockage.

- Téléchargement de la notification

Dans l'interface de téléchargement du SIP, lors d'un UPLOAD en erreur, l'icone KO sera affiché, et le xml sera téléchargé automatiquement.

7.15.1.5 Contrôle de la cohérence de SIPs

Pour un SIP versant, il y a certaine contrôle pour valider avant de le mettre dans le VITAM. Une parmis des contrôles est celle de la cohérence de SIP concernant les ArchiveUnit et Object Group. Pour ce contrôle : tous les SIP versés dans Vitam devront avoir tous leurs groupes d'objets référencés dans au moins une archiveUnit. La même contrainte s'applique pour les objets sans groupe d'objets. Le fait d'avoir des objets ou groupes d'objets sans archiveUnits est l'équivalent d'un vrac archivistique, ce que l'équipe souhaite éviter pour vitam.

- Critères d'acceptance : des critères suivantes sont appliqués pour valider cette contrôle.

CA1 : mise ne place de la nouvelle action de contrôle Etant donné le versement d'un SIP, lorsque le SIP passe par le contrôle d'entrée globale, alors le contrôle d'entrée procède à une nouvelle tâche qui est la « vérification concernant la cohérence entre objet/groupe d'objet et archiveUnit ». Cette tâche vérifie que dans le manifeste, CHAQUE objets sans groupe d'objets et CHAQUE groupe d'objets sont référencés par AU MOINS une archiveUnit

CA2 : SIP avec références valide Etant donné le versement d'un SIP dont chaque objets sans groupe d'objets ET chaque groupes d'objets sont référencés par au moins une archiveUnit. Lorsque le SIP a terminé la tâche de vérification concernant la cohérence entre objet/groupe d'objet et archiveUnit alors le workflow d'entrée continue ; et une ligne de status OK est ajouté dans le journal des opérations EVT_CHECK_MANIFEST_01_OK

CA3 : SIP avec références invalides - action non bloquante Etant donné le versement un SIP possédant au moins un objet sans groupe d'objet et/ou au moins un groupe d'objet qui n'est pas référencé par au moins une archiveUnit. Lorsqu'on le contrôle passe par la tâche de vérification concernant la cohérence entre objet/groupe d'objet et archiveUnit alors le workflow d'entrée continue

CA4 : SIP avec références invalides - bloquage du processus à la fin du contrôle d'entrée Lorsque le SAE a rencontré au moins un warning lors de la tâche de vérification concernant la cohérence entre objet/groupe d'objet et archiveUnit alors je peux constater sur l'IHM de suivi des opérations d'entrée que le statut de l'opération d'entrée passe à « erreur ». Le workflow d'entrée s'arrête et une ligne de status KO est ajoutée dans le journal des opérations : EVT_CHECK_MANIFEST_01_KO

7.15.2 Architecture Technique

7.15.2.1 Module Worker

Ce document présente le module Worker

Voici les sous-modules et package associés :

- **worker-common** [contient les méthodes commons : les modèles, les exceptions] — fr.gouv.vitam.worker.common
- **worker-client** [module client] — fr.gouv.vitam.worker.client
- **worker-core** [module core permettant l'exécution des différentes actions] — fr.gouv.vitam.worker.core
- **worker-server** [serveur REST du worker pour pouvoir effectuer des opérations sur des étapes] — fr.gouv.vitam.worker.server — fr.gouv.vitam.worker.server.rest

7.15.2.2 Worker server

7.15.2.2.1 Présentation

Parent package : **fr.gouv.vitam.worker**
Package proposition : **fr.gouv.vitam.worker.server**

Module embarquant la partie server du worker.

7.15.2.2.1.1 Services

De manière générale, pour le Worker, les méthodes utilisées sont les suivantes :

- GET : pour récupérer des infos sur une liste d'étapes, ou sur une étape particulière.
- POST : pour démarrer le lancement d'une étape.
- PUT : pour les mises à jour d'étapes.

7.15.2.2.1.2 Rest API

7.15.2.2.1.3 URI d'appel

http://server/worker/v1

7.15.2.2.1.4 Headers

Plusieurs informations sont nécessaires dans la partie header :

- X-Request-Id : l'identifiant unique de la requête.

7.15.2.2.1.5 Méthodes

GET /tasks -> **Liste les étapes en cours.**
POST /tasks -> **Permet de soumettre une étape.**
GET /tasks/{id_async} -> **Permet de récupérer le statut d'une étape.**
PUT /tasks/{id_async} -> **Permet d'intéragir avec une étape.**

GET /status -> **statut du worker**

7.15.2.3 Extraire les métadonnées des ArchiveUnit et DataObject

7.15.2.3.1 Général

L'extraction du bordereau SEDA tranforme le fichier manifest.xml en plusieurs fichiers contenant les informations du manifest, les définitions des Archives Units et des Groupes d'Objets Techniques, ainsi que la structure des objets. Dans des étapes utlérieures, les fichiers OG et Unit extraits sont indexés en base lors de l'indexation.

7.15.2.3.1.1 Workspace avant extraction :

containerId/SIP containerId/SIP/Content/ containerId/SIP/manifest.xml

7.15.2.3.1.2 Workspace après extraction :

containerId/SIP containerId/SIP/Content/ containerId/SIP/manifest.xml containerId/Units containerId/Units/AU_GUID.json containerId/Uits/... containerId/ObjectGroup containerId/ObjectGroup/GOT_GUID.json containerId/ObjectGroup/... containerId/Maps/ containerId/Maps/ARCHIVE_ID_TO_GUID_MAP.json containerId/Maps/DATA_OBJECT_TO_OBJECT_GROUP_ID_MAP.json containerId/Maps/DATA_OBJECT_ID_TO_DATA_OBJECT_DETAIL_MAP.json containerId/Maps/DATA_OBJECT_ID_TO_GUID_MAP.json containerId/Maps/OBJECT_GROUP_ID_TO_GUID_MAP.json containerId/Maps/ARCHIVE_ID_TO_ARCHIVE_DETAIL_MAP.jsonl containerId/Maps/DO_GUID_TO_DO_MAP.json containerId/UnitsLevel/ containerId/UnitsLevel/ingestLevelStack.json containerId/ATR/ containerId/ATR/globalSEDAParameters.json

7.15.2.3.2 Algorithme

1. Récupération du GUID/objects/SIP/manifest.xml
   - Voir RAML WEB/Internal_Workspace.html#containers__cid__objects__id_object__get
2. Extraction SEDA
   1. Extraction des DataObject (Physical et Binary) dans workspace depuis manifest.xml (GUID/DataObject/GUID) (SedaUtils->extractSEDA)
   - En lisant le fichier XML, extraire les DataObject depuis xml (SedaUtils->writeDataObjectInLocal et extractArchiveUnitToLocalFile)
   - **Mettre en place des MAP utiles**
     - Liaisons DataObject -> object (MAP<idDo, path>)
     - Liaisons DataObject <-> ObjectGroup (MAP<idDo, idOg> et MAP<idOg, List<idDo>>)
     - Liaisons Unit -> Unit (MAP<idUFils, List<idUPere>>)
     - Liaisons Unit <-> ObjectGroup (MAP<idx, idy> avec x et y à décider)
   2. Construction des ObjectGroup depuis les DataObject (SedaUtils->saveObjectGroupsToWorkspace)
   - A partir le map ObjectGroup -> DataObject : construire l'objet ObjectGroup en Json
   - Sauvegarde ces ObjectGroups dans workspace
   3. Sauvegarde des ArchiveUnit dans workspace depuis manifest.xml (GUID/Units/GUID) (SedaUtils->writeArchiveUnitToWorkspace)
3. Journalisation de fin de l'action extraction SEDA (fait par le Distributeur)
4. Indexation
   1. Lors de l'indexation des OGs, le worker va chercher les ObjectGroups dans workspace puis le worker se charge d'indexer dans metadata (SedaUtils->indexObjectGroup)
   2. Lors de l'indexation des Units, le worker va chercher dans le workspace, et le handler se charge de nettoyer et préparer les Units puis indexer dans metadata (SedaUtils->indexArchiveUnit)
5. Journalisation de fin de l'action d'indexation (fait par le Distributeur)

7.15.2.3.2.1 Algorithme d'update pour l'extract SEDA

Après la création de l'Archives Unit temporaire extraite du manifest.xml si une balise *<SystemId>EXISTING_GUID</SystemId>* a été rencontrée les traitement suivant sont fait : * l'Archive Unit existant est récupéré en base à partir du EXISTING_GUID fourni dans le fichier, si il n'est pas trouvé l'extraction est arrêtée * un nouveau fichier d'archive temporaire *EXISTING_GUID.json* est créé à partir du fichier extrait (*GUID.json*) en changeant modifiant d'id l'objet *« ArchiveUnit » :{ « _id » : »GUID », ...}* * l'ancien fichier *GUID.json* est supprimé * le nouveau guid *EXISTING_GUID* remplace l'ancien *GUID* dans la données temporaires d'extraction (correspondance des Id VITAM/SEDA, liste des GUID de unit extrait) et ajouté dans la liste des GUID existants * préparation du lifecycle de l'archive unit spécifique à la mise à jour (*message à définir*)

Lors de la finalisation de l'extraction des units, si le unit est déclaré comme pré-existant on ajoute : * on ajoute une valeur *« existing » :true* dans l'objet *_work : {...}* pour indiquer aux prochaines étapes que l'archive unit manipulé est une mise à jour

7.15.2.4 notification-atr-ok

Cette section présente les modules & services pour traiter le processus de notification du téléchargement d'un document SIP.

1. Génération et stockage de la notification : worker/worker-core - Le schéma de validation de la réponse XSD : le schéma de validation du fichier XML de la réponse de notification est src/main/resources/seda-2.0-main.xsd. - La génération du fichier XML de la réponse de notification est faite par l'XML Stream pour les éléments en dehors de ReplyOutcome et par des POJO JAXB pour les éléments à itérer (ArchiveUnit, BinaryDataObject, PhysicalDataObject). Les modèles Object Element POJO de ces deux éléments se trouvent dans fr.gouv.vitam.worker.model (DataObjectTypeRoot.java et ArchiveUnitReplyTypeRoot) - Handlers : le handler ExtractSedaActionHandler est modifé pour extraire des information nécessaire depuis le SIP pour générer la réponse de notification, à savoir : le map des BDOs et sa version, le json contenant des informations hors Archive Unit et Binary/Physical Data Object de SEDA.

le handler TransferNotifcationActionHandler est ajouté pour l'opération de création de la réponse de notification : création de fichier XML à partir des données générées dans le workflow, validation du fichier, effectuer la sauvergade de la réponse dans le workspace.

storage/storage-engine : - crééer la collection (report) pour sauvegarder des réponses de notification. - fournir le serive de sauvegarder de la réponse comme un document de workspace. Ce service est défini sur différens niveaux à savoir API/Rest/Client

2. Téléchargement de la notification

   storage/storage-engine ingest/ingest-internal ingest/ingest-external ihm-demo/ihm-demo-web-application
   Tous ces 4 services sont mis à jour pour récupérer la réponse de notification sauvegardée dans le storage.

7.15.2.5 notification-atr-ko

Cette section présente les modules & services pour traiter le processus de notification en erreur de l'upload d'un document SIP.

1. Génération et stockage de la notification :

worker/worker-core : - Le schéma de validation de la réponse XSD : les schémas différents de validation du fichier XML de la réponse de notification est : src/main/resources/seda-2.0-main.xsd. - La génération du fichier XML de la réponse de notification est faite par l'XML Stream. - Workflow : Le workflow a été modifié, un behaviour « Finally » a été ajouté. A l'image du Finally java, il permet d'exécuter une étape quoi qu'il se passe dans le process d'ingest d'un SIP. Ceci permet la génération d'une notification KO. Le workflow a été adapté pour que l'on puisse également générer une notification OK dans le cadre d'un succès. - Handlers : le handler TransferNotifcationActionHandler a été modifié pour pouvoir répondre au besoin de génération d'un XML KO : création de fichier XML à partir des données

générées dans le workflow (logbook opération, logbook lifecycle unit et object group), effectuer la sauvergade de la réponse dans le workspace.

storage/storage-engine : - utilisation de l'API REST pour pouvoir sauvegarder la réponse.

2. Téléchargement de la notification
   storage/storage-engine ingest/ingest-internal ingest/ingest-external ihm-demo/ihm-demo-web-application

Tous ces 4 services sont mis à jour pour récupérer la réponse de notification sauvegardée dans le storage.

7.15.2.6 Plugin Worker

7.15.2.6.1 But de cette documentation

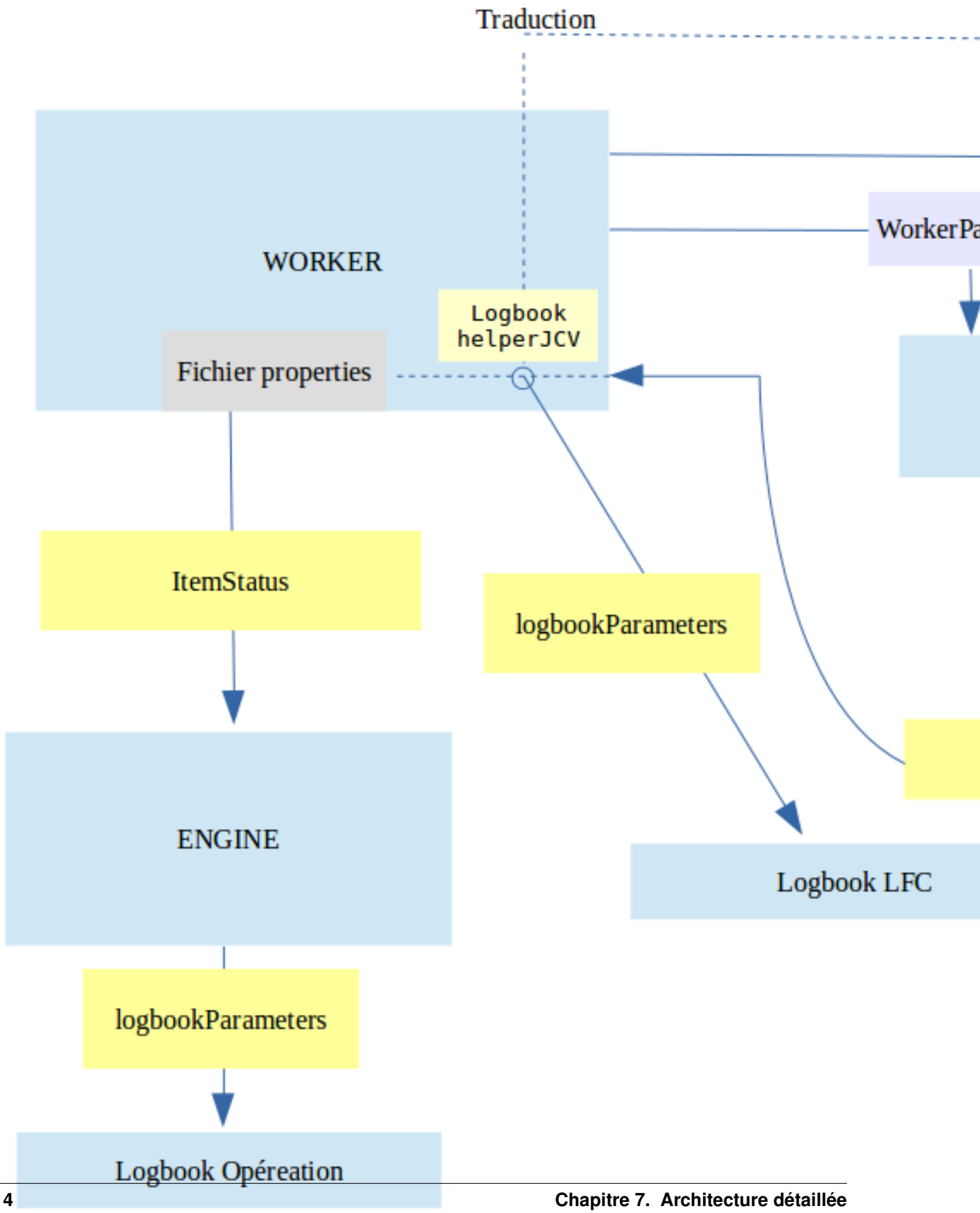
L'objectif de cette documentation est d'expliquer l'architecture technique des plugins de worker.

7.15.2.6.2 Introduction

Le plugin worker est une classe java qui réaslise des actions dans le workflow comme les Handler. Dans le workflow, si l'action traite une action qui a besoin un enregistrement JCV, le plugin sera remplacé le Handler.

Le plugin prends en entrée une interface HandlerIO pour charger les fichier de vitam, les paramètres du worker En sortie, il retourne les status des traitements et chaque status contient un code de traitement (définit dans le fichier properties du plugin, si ce n'est pas définit on utilise le code par défaut)

D'une façon synthétique, le plugin worker est décrit de cette façon :


Traduction
WORKER
Logbook
helperJCV
Fichier properties
WorkerPa
ItemStatus
logbookParameters
ENGINE
Logbook LFC
logbookParameters
Logbook Opéreation

7.15.2.6.3 Appel du plugin

Au démarrage du service, le serveur *worker* charge tous les *plugins* et leurs fichier de propriétés. Les référentiels de *plugin* sont déclarés dans un fichier de configuration :

```
{
   "NOM_DE_PLUGIN_1": {
     "className": "package.plugin.class_1",
     "propertyFile": "le_fichier_de_properties_1"
  },
  "NOM_DE_PLUGIN_2": {
    "className": "package.plugin.class_2",
    "propertyFile": "le_fichier_de_properties_2"
  }
}
```

Au démarrage de chaque *worker*, la liste des plugins va être analysé. Puis le serveur va tenter d'instancier chaque plugin de la liste. Si un des plugins ne se lance pas pour une raison quelconque (nom de classe incorrect, impossible d'instancier la classe, ...), alors le serveur ne démarrera pas.

Les plugins ne sont pas pour l'instant thread safe dans Vitam, ce qui signifie que un plugin est réinstancié pour chaque appel au serveur worker.

7.15.2.6.4 Résultat du plugin

Après ses traitements, Plugin doit retourner au Worker un ItemStatus. Quand le Worker reçoit le résultat :

- Il doit le traduire en utilisant par défaut le fichier de properties VITAM (vitam-logbook-messages_fr.properties)

si les clés ne sont pas définies dans ce fichier, alors il va chercher la valeur du label dans le fichier properties du *plugin* puis envoie à *Engine* pour écrire dans les journaux des opération.

- Construire et écrire LogbookLifeCycle.

7.15.2.6.5 Implémentation

7.15.2.6.5.1 Worker

- **getActionHandler** : pour chaque action, le worker vérifie si l'action est dans la liste des plugins, il va le charger, si non on utilise les handlers prédéfinis dans Vitam
- **writeLogbookLifeCycle** : traduire le code d'action d'un ItemStatus du Plugin en LogbookLifeCycleParameters puis en fonction du type d'élément dans la distribution (Unit ou ObjectGroup), il écrit dans la base de données correspondante

Exemple : Le plugin CHECK_DIGEST fait un traitement CALC_CHECK qui donne un status OK.

Le résultat retourné du plugin contiendra :

```
{
 "globalStatus" : OK ,
 "itemsStatus" : [ {"CALC_CHECK" :  { "globalStatus" : OK  }}]
}
```

Alors le worker va écrire ces événements ci-dessous dans *LFC*.

```
{
 {
   "evType" : "LFC.CHECK_DIGEST ",
   "outcome" : "OK",
   "outDetail" : "LFC.CHECK_DIGEST..OK",
 },
 {
   "evType" : "LFC.CHECK_DIGEST.CALC_CHECK ",
   "outcome" : "OK",
   "outDetail" : "LFC.CHECK_DIGEST.CALC_CHECK.OK",
 }
}
```

L'écriture des journaux des opérations garde son implémentation.

7.15.2.6.5.2 PluginPropertiesLoader

c'est un service pour charger les définitions du code dans le fichier de properties du plugin

7.15.2.6.5.3 Intégration

Cela définit comment Worker appelle les plugins.

`java -cp "/vitam/lib/${unix.name}/*" fr.gouv.vitam.worker.server.rest.WorkerApplication` au lieu de `java -jar "/vitam/lib/${unix.name}/${project.build.finalName}.jar"`

Donc les JAR du plugin doit être placé dans `/vitam/lib/worker/`.

CHAPITRE 8

Annexes

Table des figures

1 Vue de VITAM dans son environnement (vue « boîte noire ») 8
2 Architecture fonctionnelle cible de *VITAM* 13

1 Architecture applicative : légende 15
2 Architecture applicative : flux de données d'archives et de commandes 15
3 Architecture applicative : flux de données de journalisation 16
4 Architecture applicative : flux de données de référentiels 16
5 Architecture des données d'archives ; fonctionnement multisite. 20

1 Environnement d'un service VITAM 32
2 Déploiement VITAM : zones & principes de communication ; les utilisateurs métier archivistes sont présentés à gauche, et les exploitants technique à droite. 40
3 Architecture technique : légende 50
4 Architecture technique : flux (1/5 : flux métiers généraux) 51
5 Architecture technique : flux (2/5 : flux métiers de dépôt des journaux) 52
6 Architecture technique : flux (3/5 : flux métiers de lecture des référentiels métier) 53
7 Architecture technique : flux (4/5 : flux des outils d'exploitation) 54
8 Architecture technique : flux (5/5 : flux du socle technique). Seul le port exposant les services d'administration/exploitation est représenté sur le composant *VITAM* présenté dans cette figure. 55
9 Architecture technique : délimitation par zones 56
10 Architecture générique d'un système de gestion de logs. 66
11 Architecture du sous-système de centralisation des logs 67
12 Déploiement d'un cluster Mongo DB avec sharding. 85

1 Vue d'ensemble des magasins de certificats déployés dans un système *VITAM* ; chaque couleur correspond à une chaîne de certification potentiellement disjointe des autres. . . 111

Liste des tableaux

1 Documents de référence VITAM 2

1 Inventaire des données VITAM 18
2 Inventaire des données selon le type de stratégie VITAM 19

1 Dimensionnement XSmall 94
2 Dimensionnement Small 96
3 Dimensionnement Medium 98
4 Dimensionnement Large 100
5 Dimensionnement XLarge 102
6 Matrice des flux inter-zones 106
7 Matrice des flux inter-sites 106

Index

A
API, 3
AU, 3

B
BDD, 3
BDO, 3

C
CA, 3
CAS, 3
CCFN, 3
CN, 3
COTS, 3
CRL, 3
CRUD, 3

D
DAT, 3
DC, 3
DEX, 3
DIN, 3
DIP, 3
DMV, 3
DNS, 3
DNSSEC, 3
DSL, 3
DUA, 3

E
EAD, 3
EBIOS, 3
ELK, 3

F
FIP, 3

G
GOT, 3

I
IHM, 3
IP, 3
IsaDG, 3

J
JRE, 3
JVM, 4

L
LAN, 4
LFC, 4
LTS, 4

M
M2M, 4
MitM, 4
MoReq, 4

N
NoSQL, 4
NTP, 4

O
OAIS, 4
OOM, 4
OS, 4
OWASP, 4

P
PCA, 4
PDMA, 4
PKI, 4
PRA, 4

R
REST, 4
RGAA, 4
RGI, 4

RPM, 4

S

SAE, 4
SEDA, 4
SGBD, 5
SGBDR, 5
SIA, 5
SIEM, 5
SIP, 5
SSH, 5
Swift, 5

T

TLS, 5
TNA, 5
TNR, 5
TSA, 5
TTL, 5

U

UDP, 5
UID, 5

V

VITAM, 5
VM, 5

W

WAF, 5
WAN, 5