



VITAM - Architecture

Version 0.20.0

VITAM

juil. 21, 2017

Table des matières

1	Introduction	1
1.1	Objectif de ce document	1
1.2	Structure du document	1
2	Rappels	3
2.1	Information concernant les licences	3
2.2	Documents de référence	3
2.2.1	Documents internes	3
2.2.2	Référentiels externes	3
2.3	Glossaire	3
3	Vue d'ensemble	5
3.1	Drivers du projet	5
3.1.1	Enjeux	5
3.1.2	Contraintes et objectifs	5
3.1.3	Positionnement	6
3.2	Interfaces externes du système	6
3.2.1	Interfaces requises	7
3.2.2	Interfaces métier exposées	7
3.3	Orientations générales	7
3.3.1	Open Source	7
3.3.2	API REST	8
3.3.3	Big Data et Cloud computing	8
3.3.4	Cloud storage	9
3.3.5	PCA/PRA et répartitions des travaux	9
3.3.6	Sécurité des données additionnelle	10
3.3.7	Architecture multi-tenants	10
3.3.8	Solution exploitable	10
3.4	Architecture fonctionnelle	12
4	Architecture applicative	13
4.1	Architecture applicative	13
4.1.1	Drivers de l'architecture	13
4.1.2	Services	13
4.1.3	Détail des flux d'information métier	15
4.2	Services métiers	15
4.2.1	Moteur d'exécution (processing)	15
4.2.2	Moteur de stockage (storage)	16

4.2.3	Espace de travail (workspace)	16
4.2.4	Worker (worker)	16
4.2.5	Moteur de données (metadata)	16
4.2.6	Moteur de journalisation (logbook)	17
4.2.7	Gestion des référentiels (functional-administration)	17
4.2.8	Offre de stockage par défaut (storage-offer-default)	17
4.2.9	Moteur d'entrée (ingest-internal)	18
4.2.10	Moteur d'accès (access-internal)	18
4.2.11	API externes (ingest-external et access-external)	18
4.2.12	Interface de démonstration (ihm-demo)	18
5	Architecture technique / exploitation	21
5.1	Principes d'architecture technique	21
5.1.1	Principes communs et environnement des services	21
5.1.1.1	Principes relatifs aux composants délivrés	21
5.1.1.1.1	Nommage	21
5.1.1.1.2	Principes relatifs aux services VITAM	21
5.1.1.1.3	Principes relatifs aux COTS	23
5.1.2	Utilisateurs, dossiers & droits	23
5.1.2.1	Utilisateurs et groupes d'exécution	23
5.1.2.1.1	Groupes	24
5.1.2.1.2	Utilisateurs	24
5.1.2.2	Arborescence de fichiers	24
5.1.2.2.1	Services VITAM	24
5.1.2.2.1.1	Arborescence VITAM	24
5.1.2.2.1.2	Intégration au système	25
5.1.2.2.2	COTS	25
5.1.3	Principes sur les communications inter-services et le clustering	25
5.1.3.1	Clusters applicatifs métier	25
5.1.3.1.1	Appels REST des services métier	26
5.1.3.1.2	Workers	26
5.1.3.2	COTS & clustering	26
5.1.3.3	Annuaire de services (service registry)	26
5.1.4	Packaging	27
5.1.4.1	Principes communs	27
5.1.4.2	Dépôts	27
5.1.4.2.1	CentOS	28
5.1.4.2.2	Debian	28
5.1.4.3	Prise en compte de la configuration dans le packaging	28
5.1.4.3.1	CentOS	28
5.1.4.3.2	Debian	29
5.1.5	Déploiement de la solution	29
5.1.5.1	Principes de déploiement	29
5.1.5.2	Contraintes et vue d'ensemble	29
5.1.5.3	Installation initiale	31
5.1.5.4	Principes de maj à chaud	31
5.1.5.5	Multi-site	31
5.1.5.6	Support de l'élasticité	31
5.1.5.7	Validation du déploiement	32
5.1.6	Suivi de l'état du système	32
5.1.6.1	API de supervision	32
5.1.6.2	Métriques	32
5.1.6.3	Logs	33
5.1.6.3.1	Protocoles : syslog	33

5.1.6.3.2	Types de log	33
5.1.6.3.2.1	Logs applicatifs	33
5.1.6.3.2.2	Logs du garbage collector Java	34
5.1.6.3.2.3	Logs d'accès	34
5.1.6.4	Intégration à un système de monitoring tiers	34
5.1.7	Administration technique	34
5.1.7.1	Démarrage / arrêt des services	34
5.1.7.2	Tâches régulières (“cron”)	35
5.1.8	Gestion des données du système	35
5.1.8.1	Sauvegarde	35
5.1.8.1.1	Vue d'ensemble	35
5.1.8.1.2	Principe d'export MongoDB pour une base répartie	35
5.1.8.1.3	Dossiers	38
5.1.8.2	Restauration	38
5.2	Services techniques fournis par la solution	38
5.2.1	Moteur de déploiement et de configuration	38
5.2.2	Chaîne de traitement de logs et de métriques	39
5.2.3	Service registry	39
5.3	Composants logiciels utilisés	39
5.3.1	Fournies	39
5.3.1.1	COTS	39
5.3.1.2	Bibliothèques structurantes	40
5.3.2	Requises	40
5.4	Architecture technique détaillée	40
5.4.1	Flux métier	41
5.4.2	Flux exploitation	44
5.4.3	Flux techniques	45
5.5	Concentration et exploitation des logs applicatifs	46
5.5.1	Besoins	46
5.5.2	Modèle générique	46
5.5.3	Choix des implémentations	47
5.5.3.1	Emetteur de logs	48
5.5.3.2	Agent de transport de log	49
5.5.3.3	Concentration de logs	49
5.5.3.4	Stockage des logs	49
5.5.3.4.1	Gestion des index	50
5.5.3.5	Visualisation des logs	51
5.5.4	Intégration à un système de gestion de logs existants	51
5.5.5	Limites	51
5.6	Métriques applicatives	51
5.6.1	Besoins	51
5.6.2	Modèle générique	51
5.6.3	Choix des implémentations	52
5.6.3.1	Enregistreur de métriques	52
5.6.3.2	Reporters de métriques	52
5.6.3.3	Stockage des métriques	52
5.6.3.3.1	Gestion des index	53
5.6.3.4	Visualisation des métriques	53
5.6.4	Limites	54
5.7	Outilage de déploiement	54
5.7.1	Outil	54
5.7.2	Architecture de l'outil	54
5.7.3	Gestion des secrets	55
5.8	Service registry	55

5.8.1	Architecture	55
5.8.2	Résolution DNS	56
5.8.3	Packaging	56
5.8.4	Monitoring	57
5.9	Dépendances aux services d'infrastructures	57
5.9.1	Ordonnateurs techniques / batchs	57
5.9.1.1	Curator	57
5.9.1.2	Sécurisation des journaux d'opérations	57
5.9.1.3	Sécurisation des journaux d'écriture	57
5.9.1.4	Cas de la sauvegarde	57
5.9.2	Socles d'exécution	57
5.9.2.1	OS	57
5.9.2.2	Middlewares	58
5.10	Composants déployés	58
5.10.1	Access-external	58
5.10.2	Access-internal	58
5.10.3	Consul	59
5.10.3.1	Architecture de déploiement	59
5.10.4	Curator	59
5.10.5	Elasticsearch-data	60
5.10.5.1	Architecture de déploiement	60
5.10.5.2	Monitoring	60
5.10.6	Elasticsearch-log	61
5.10.6.1	Architecture de déploiement	61
5.10.7	Functional-administration	61
5.10.8	Ingest-external	61
5.10.8.1	Antivirus	62
5.10.9	Ingest-internal	62
5.10.10	Kibana	62
5.10.10.1	Déploiement	63
5.10.11	Logbook	63
5.10.12	Logstash	63
5.10.13	Metadata	63
5.10.14	Mongodb	64
5.10.14.1	Architecture de déploiement	64
5.10.14.1.1	Architecture 1 noeud	64
5.10.14.1.2	Architecture distribuée	64
5.10.14.1.3	Ports utilisés	65
5.10.15	Processing	65
5.10.16	Siegfried	66
5.10.16.1	Mode de fonctionnement dans VITAM	66
5.10.17	Storage	66
5.10.18	Storage-offer	66
5.10.19	Worker	67
5.10.19.1	Particularités	67
5.10.20	Workspace	67
5.11	Guidelines de déploiement	67
5.12	Utilisation des ressources informatiques	68
5.13	Matrice des flux	68
6	Securite	71
6.1	Principes	71
6.1.1	Principes de cloisonnement	71
6.1.2	Principes de sécurisation des accès externes	71

6.1.3	Principes de sécurisation des communications internes au système	72
6.1.4	Principes de sécurisation des bases de données	72
6.1.4.1	MongoDB	72
6.1.4.2	Elasticsearch	73
6.1.5	Principes de sécurisation des secrets de déploiement	73
6.2	Liste des secrets	73
6.3	Certificats	74
7	Architecture détaillée	75
7.1	Access	75
7.1.1	Architecture Fonctionnelle	75
7.1.1.1	Généralités	75
7.1.2	Architecture Technique	75
7.1.2.1	Introduction	75
7.1.2.1.1	Présentation	75
7.1.2.1.2	Itération 4	75
7.1.2.1.3	Modules - packages	76
7.1.2.2	Access-api	76
7.1.2.2.1	Présentation	76
7.1.2.2.3	Access-client	76
7.1.2.4	Utilisation	76
7.1.2.5	Le client	77
7.1.2.6	Access-common	77
7.1.2.6.1	Présentation	77
7.1.2.7	Access-core	77
7.1.2.7.1	Présentation	77
7.1.2.7.2	Packages :	78
7.1.2.7.2.1	Récupération d'un objet spécifique	78
7.1.2.8	Access-rest	78
7.1.2.8.1	Présentation	78
7.1.2.8.2	Packages :	79
7.1.2.8.3	fr.gouv.vitam.access.external.rest	79
7.1.2.8.3.1	Rest API	79
7.1.2.9	-AccessApplication.java	79
7.1.2.10	-AccessResourceImpl.java	79
7.1.2.11	-LogbookExternalResourceImpl.java	82
7.1.2.12	-AdminManagementExternalResourceImpl.java	84
7.1.3	Sécurité	86
7.2	Common	86
7.2.1	Architecture Fonctionnelle	86
7.2.1.1	Introduction	86
7.2.1.1.1	But de cette documentation	86
7.2.1.1.2	GUID	86
7.2.1.1.3	ServerIdentity et Logger	86
7.2.1.2	Introduction	86
7.2.1.2.1	Présentation de la problématique	87
7.2.1.2.1.1	Qu'est ce qu'une URL pérenne ?	87
7.2.1.2.1.2	Objectifs	87
7.2.1.2.1.3	Préconisation E-ARK	87
7.2.1.2.2	Solutions envisagées	87
7.2.1.2.2.1	ARK	87
7.2.1.2.2.2	Forme d'un ARK	87
7.2.1.2.2.3	Identifiant Vitam	88
7.2.1.2.2.4	Logique de construction	88

7.2.1.2.2.5	Logique d'affichage	88
7.2.1.2.2.6	Capacité de déconstruction	88
7.2.1.3	Introduction	89
7.2.1.3.1	Objectifs	89
7.2.1.4	Introduction	89
7.2.1.4.1	Vérification des formats :	89
7.2.2	Architecture Technique	89
7.2.2.1	Introduction	89
7.2.2.1.1	But de cette documentation	89
7.2.2.1.2	GUID	90
7.2.2.2	Introduction	90
7.2.2.2.1	Identifiant Vitam	90
7.2.2.2.1.1	Forme d'un identifiant Vitam	90
7.2.2.3	Configuration jetty	91
7.2.2.4	Gestion des Handlers :	91
7.2.2.5	Schéma de certificats et d'authentification	92
7.2.2.5.1	Présentation	92
7.2.2.6	Common format identification	92
7.2.2.6.1	Présentation	92
7.2.2.6.2	Sous packages	92
7.2.2.6.2.1	Identification :	92
7.2.2.6.2.2	Exceptions :	93
7.2.2.6.2.3	Model :	93
7.2.2.6.2.4	Siegfried :	93
7.2.2.7	Messages	93
7.2.2.8	Messages Logbook	93
7.2.2.9	Request ID	94
7.2.2.9.1	Filtre client	94
7.2.2.9.2	Sauvegarde dans le thread local	94
7.2.2.9.3	Filtre Serveur	94
7.2.2.9.4	Affichage dans les logs	94
7.2.3	Securite	95
7.2.3.1	Introduction	95
7.2.3.2	Securité de MongoDB	95
7.2.3.2.1	Objectifs	95
7.2.3.3	secret de la plateforme	95
7.2.3.3.1	Objectifs	95
7.3	Functional administration	95
7.3.1	Architecture Fonctionnelle	95
7.3.1.1	Introduction	95
7.3.1.1.1	But de cette documentation	95
7.3.1.2	Introduction	96
7.3.1.2.1	Gestion de format	96
7.3.1.3	Introduction	96
7.3.1.3.1	Gestion de règles	96
7.3.2	Architecture Technique	96
7.3.2.1	Introduction	96
7.3.3	Securite	96
7.3.3.1	Introduction	96
7.4	IHM demo	96
7.4.1	Architecture fonctionnelle	96
7.4.1.1	Architecture fonctionnelle de l'application Back	96
7.4.1.1.1	But de cette documentation	96
7.4.1.1.2	Fonctionnement général du module	96

7.4.1.1.2.1	Recherche des units : POST /ihm-demo/v1/api/archivesearch/units	96
7.4.1.1.2.2	Affichage du détail d'une archive unit : GET /ihm-demo/v1/api/archivesearch/unit/{id}	97
7.4.1.1.2.3	Modification et enregistrement des détails d'une archive unit : PUT /ihm-demo/v1/api/archiveupdate/units/{id}	97
7.4.1.1.2.4	Remarque importante	98
7.4.1.1.2.5	Reste à faire	98
7.4.1.2	Architecture fonctionnelle de l'application Front	98
7.4.1.2.1	But de cette documentation	98
7.4.1.2.2	Modules AngularJS déclarés	98
7.4.1.2.3	Routage	99
7.4.1.2.4	Factories/Services	99
7.4.1.2.5	Controllers	99
7.4.1.2.6	Components	100
7.4.2	Architecture technique	100
7.4.2.1	Architecture technique de l'application Back	100
7.4.2.1.1	But de cette documentation	100
7.4.2.1.2	Organisation du module ihm-demo	100
7.4.2.1.2.1	1. Module ihm-demo-web-application	100
7.4.2.1.2.2	package fr.gouv.vitam.ihmdemo.appserver	100
7.4.2.1.2.3	2. Module ihm-core	101
7.4.2.1.2.4	package fr.gouv.vitam.ihmdemo.core	101
7.4.2.2	Architecture technique de l'application Front	101
7.4.2.2.1	But de cette documentation	101
7.4.2.2.2	Le Framework Front : AngularJS 1.5.3	101
7.4.2.2.2.1	Les modules AngularJS utilisés :	101
7.4.2.2.2.2	Autres frameworks Front utilisés	101
7.4.2.2.3	Organisation de l'application	102
7.5	IHM recette	102
7.5.1	Architecture technique	102
7.5.1.1	Architecture technique de l'application Back	102
7.5.1.1.1	But de cette documentation	102
7.5.1.1.2	Organisation du module ihm-recette	103
7.5.1.1.2.1	1. Module ihm-demo-web-application	103
7.5.1.1.2.2	package fr.gouv.vitam.ihmdemo.appserver	103
7.5.1.1.2.3	package fr.gouv.vitam.ihmdemo.appserver.performance	103
7.5.1.1.2.4	2. Module ihm-recette-web-front	103
7.5.1.1.2.5	3. Module ihm-core	103
7.5.1.2	Architecture technique de l'application Front	103
7.5.1.2.1	But de cette documentation	103
7.5.1.2.2	Le Framework Front : AngularJS 1.5.3	104
7.5.1.2.2.1	Les modules AngularJS utilisés :	104
7.5.1.2.2.2	Autres frameworks Front utilisés	104
7.5.1.2.3	Organisation de l'application	104
7.6	Ingest	104
7.6.1	Architecture Fonctionnelle	104
7.6.1.1	Généralités	104
7.6.1.2	Généralités	105
7.6.1.3	Téléchargement standard et test à blanc d'un SIP :	105
7.6.1.4	Autres Fonctionnalités :	105
7.6.1.5	Ingest ExternalAntivirus	106
7.6.1.6	Généralités	106
7.6.1.7	Fonctionnalités concernant le workflow	106
7.6.1.8	Les actions :	106

7.6.1.9	Asynchrone :	107
7.6.2	Technique	107
7.6.2.1	Architecture Technique Ingest	107
7.6.2.1.1	Présentation	107
7.6.2.2	ingest-rest	107
7.6.2.2.1	Présentation	107
7.6.3	Securite	108
7.6.3.1	Introduction	108
7.7	Logbook	108
7.7.1	Architecture Fonctionnelle	108
7.7.1.1	Généralités	108
7.7.1.2	Généralités	109
7.7.1.3	Modèle de données	109
7.7.1.3.1	Description des champs	109
7.7.1.4	Modèle de données	111
7.7.1.4.1	Description des champs	111
7.7.2	Architecture technique	113
7.7.2.1	Introduction	113
7.7.2.1.1	Présentation	113
7.7.2.1.2	Itération 3 et Itération 5	113
7.7.2.1.2.1	Itérations suivantes / à plus long terme	113
7.7.2.1.3	Modules - packages logbook	113
7.7.2.2	DSL	114
7.7.2.2.1	Analyse	114
7.7.2.2.1.1	Présentation	114
7.7.2.2.1.2	Explication	115
7.7.2.2.1.3	Utilisation	115
7.7.2.2.2	Conclusion	116
7.7.2.3	Rest	116
7.7.2.3.1	Présentation	116
7.7.2.3.2	Services	116
7.7.2.4	Common-client	116
7.7.2.4.1	Présentation	116
7.7.2.4.2	Services	117
7.7.2.5	Common-client	117
7.7.2.5.1	Présentation	117
7.7.2.5.2	Services	117
7.7.2.5.3	Données	117
7.7.2.6	Commons	118
7.7.2.6.1	Présentation	118
7.7.2.6.2	Services	118
7.7.2.7	Operation Client	118
7.7.2.7.1	Présentation	118
7.7.2.7.2	Services	118
7.7.2.8	Opération	118
7.7.2.8.1	Présentation	118
7.7.2.8.2	Services	119
7.7.2.8.3	Rest API	119
7.7.2.9	Lifecycle Client	119
7.7.2.9.1	Présentation	119
7.7.2.9.2	Services	119
7.7.2.10	Lifecycle	119
7.7.2.10.1	Présentation	119
7.7.2.10.2	Services	119

7.7.2.10.3	Rest API	119
7.7.2.11	Administration-client	120
7.7.2.11.1	Présentation	120
7.7.2.11.2	Services	120
7.7.2.12	Administration	120
7.7.2.12.1	Présentation	120
7.7.2.12.2	Services	120
7.7.2.12.3	Rest API	120
7.7.3	Securite	120
7.7.3.1	Introduction	120
7.8	Metadata	120
7.8.1	Architecture Fonctionnelle	120
7.8.1.1	Introduction	120
7.8.2	Architecture technique	120
7.8.2.1	Introduction	120
7.8.2.1.1	Présentation	120
7.8.2.1.2	Itération 4	121
7.8.2.1.3	Modules - packages	121
7.8.2.2	Metadata-api	121
7.8.2.2.1	Présentation	121
7.8.2.3	Metadata-builder	122
7.8.2.3.1	Présentation	122
7.8.2.4	Operation Client	123
7.8.2.4.1	Présentation	123
7.8.2.5	metadata-core	123
7.8.2.5.1	Présentation	123
7.8.2.5.2	1. Modules et packages	123
7.8.2.5.3	2. Classes	123
7.8.2.5.3.1	2.1 Class DbRequest	123
7.8.2.5.3.2	2.2 Class ElasticsearchAccessMetadata	124
7.8.2.5.3.3	2.3 Class MetaDataImpl	124
7.8.2.5.3.4	2.4 Class UnitNode	125
7.8.2.5.3.5	2.5 Class UnitRuleCompute	125
7.8.2.5.3.6	2.5 Class UnitInheritedRule	125
7.8.2.6	metadata-parser	125
7.8.2.6.1	Présentation	125
7.8.2.7	Métadata	125
7.8.2.7.1	Présentation	125
7.8.2.7.2	Services	126
7.8.2.7.3	Rest API	126
7.8.2.8	Rest	126
7.8.2.8.1	Présentation	126
7.8.2.8.2	Services	126
7.8.3	Securite	126
7.8.3.1	Introduction	126
7.9	Processing	126
7.9.1	architecture-fonctionnelle-processing	126
7.9.1.1	Introduction	126
7.9.1.1.1	But de cette documentation	126
7.9.1.1.2	Processing	127
7.9.1.2	Processing Management	127
7.9.1.3	Introduction	127
7.9.1.3.1	But de cette documentation	127
7.9.1.4	Introduction	128

7.9.1.4.1	But de cette documentation	128
7.9.1.5	Introduction	128
7.9.1.5.1	But de cette documentation	128
7.9.1.6	Process Monitoring	128
7.9.1.6.1	Explication	128
7.9.2	Architecture Technique	129
7.9.2.1	Introduction	129
7.9.2.2	DAT : module processing	129
7.9.2.2.1	1. Module et packages	129
7.9.2.2.2	2. Modèle	129
7.9.2.2.3	3. Process Distributor	129
7.9.2.2.4	4. Parallélisme dans le distributeur	130
7.9.2.3	Rangement des objets	131
7.9.2.3.1	Algorithme	131
7.9.2.4	Vérification de la disponibilité	131
7.9.2.4.1	Algorithme	131
7.9.2.5	Vérifier SEDA	131
7.9.2.5.1	Algorithme	131
7.9.3	Securite	132
7.9.3.1	Introduction	132
7.10	Storage	132
7.10.1	Architecture Fonctionnelle	132
7.10.1.1	Introduction	132
7.10.2	Architecture Technique	132
7.10.2.1	Introduction	132
7.10.2.1.1	Présentation	132
7.10.2.1.2	Itération 16	133
7.10.2.1.3	Modules - packages Storage	133
7.10.2.2	Architecture générale	134
7.10.2.2.1	Schéma général	134
7.10.2.2.2	Workflow du stockage des objets	134
7.10.2.2.3	Itération 6	135
7.10.2.2.4	Itération 7	135
7.10.2.2.5	Itération 13	135
7.10.2.2.6	Itération 14	136
7.10.2.2.7	Itération 16	136
7.10.2.3	Storage Driver	136
7.10.2.3.1	Présentation	136
7.10.2.3.2	Architecture	136
7.10.2.3.3	Pour aller plus loin	137
7.10.2.4	Storage Engine	137
7.10.2.4.1	Présentation	137
7.10.2.4.1.1	Services	137
7.10.2.4.1.2	Rest API	138
7.10.2.4.1.3	URI d'appel	138
7.10.2.4.1.4	Headers	138
7.10.2.4.1.5	Méthodes	138
7.10.2.4.1.6	Distribution	139
7.10.2.4.1.7	DriverManager : SPI	140
7.10.2.4.1.8	Principe	140
7.10.2.4.1.9	Persistance	141
7.10.2.5	Storage Engine Client	141
7.10.2.5.1	Présentation	141
7.10.2.6	Storage Offers	141

7.10.2.6.1	Présentation	141
7.10.2.7	Vitam Offer	142
7.10.2.7.1	Présentation	142
7.10.2.7.2	Driver	142
7.10.2.7.3	Serveur	142
7.10.2.7.3.1	Description	142
7.10.2.7.3.2	REST	143
7.10.2.7.3.3	Description	143
7.10.2.7.3.4	REST API	143
7.10.3	Securite	146
7.10.3.1	Introduction	146
7.11	Technical administration	146
7.11.1	Architecture Fonctionnelle	146
7.11.1.1	Introduction	146
7.11.2	Architecture Technique	146
7.11.2.1	Introduction	146
7.11.3	Securite	146
7.11.3.1	Introduction	146
7.12	Workspace	146
7.12.1	Architecture Fonctionnelle	146
7.12.1.1	Introduction	146
7.12.2	Architecture Technique	146
7.12.2.1	Introduction	146
7.12.3	Securite	146
7.12.3.1	Introduction	146
8	Annexes	147
Index		153

Introduction

1.1 Objectif de ce document

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

Rappels

2.1 Information concernant les licences

La solution logicielle *VITAM* est publiée sous la license CeCILL 2.1¹ ; la documentation associée (comprenant le présent document) est publiée sous Licence Ouverte V2.0².

2.2 Documents de référence

2.2.1 Documents internes

Tableau 2.1 – Documents de référence VITAM

Nom	Lien
<i>DAT</i>	(à renseigner)
<i>DIN</i>	(à renseigner)
<i>DEX</i>	(à renseigner)
Release notes	(à renseigner)

2.2.2 Référentiels externes

2.3 Glossaire

API Application Programming Interface

BDD Base De Données

COTS Component Off The Shelves ; 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.

DAT Dossier d'Architecture Technique

DEX Dossier d'EXploitation

DIN Dossier d'Installation

1. http://www.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>

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 DNSSEC³](#)

DUA Durée d'Utilité Administrative

IHM Interface Homme Machine

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

JVM Java Virtual Machine ; Cf. [JRE](#)

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. [Explication⁴](#)

NoSQL Base de données non-basée sur un paradigme classique des bases relationnelles. [Définition⁵](#)

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.

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 PKI⁶](#)

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. [Définition⁷](#)

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

SAE Système d'Archivage Électronique

SEDA Standard d'Échange de Données pour l'Archivage

SIA Système d'Informations Archivistique

TNR Tests de Non-Régression

VITAM Valeurs Immatérielles Transférées aux Archives pour Mémoire

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

4. https://fr.wikipedia.org/wiki/Attaque_de_l%27homme_du_milieu

5. <https://fr.wikipedia.org/wiki/NoSQL>

6. https://fr.wikipedia.org/wiki/Infrastructure_%C3%A0_cl%C3%A9s_publiques

7. https://fr.wikipedia.org/wiki/Representational_state_transfer

Vue d'ensemble

3.1 Drivers du projet

3.1.1 Enjeux

Les enjeux de 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 liée 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 besoins 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 \Rightarrow 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, et notamment 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 recherches 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

- Requêteable : le SAE doit fournir un service de recherche, tout comme un SGBD (système de gestion de bases de données) fournit 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

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.

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

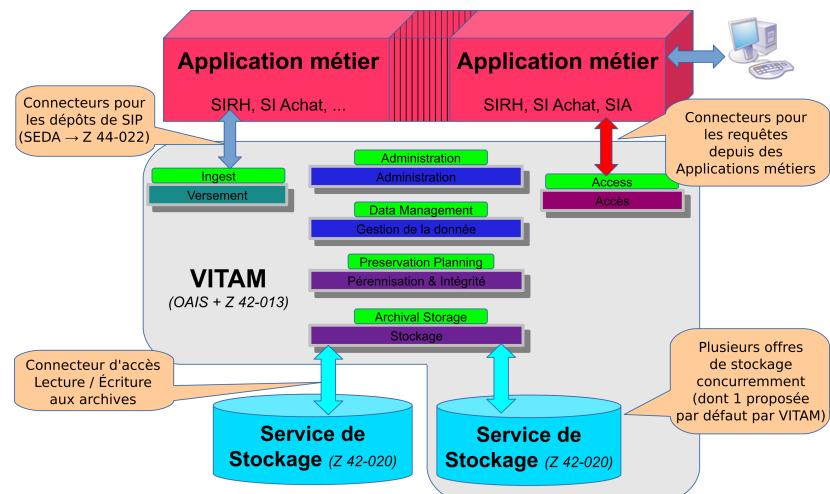


Fig. 3.1 – Vue de VITAM dans son environnement (vue “boîte noire”)

3.2.1 Interfaces requises

Dans cette version du système, aucune interface externe n'est requise par VITAM.

3.2.2 Interfaces métier exposées

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'archive présentes dans le système (métadonnées et données d'archives, journeaux, 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 71)

3.3 Orientations générales

Avertissement : Ces orientations générales donnent la direction vers laquelle tend le système 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 c'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 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 palier à 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 “schema less”.

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 vitesse 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 (vitesse).

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 vitesse est donc un point important.

Ces 3 V (Volume, Variété, Vitesse) 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. Ainsi chaque objet étant accédé unitairement, la logique retenue pour le stockage est une logique Objet (“Content Adressable Storage = 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 elle sera 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 ou Google ou Azur, 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

Le logiciel Vitam est prévu pour être installé 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.

Cette distribution des sites peut aussi permettre d'équilibrer la charge des opérations entre les sites :

- Le site primaire s'occupe de toutes les opérations en écriture (insertion et modification) ainsi qu'un sous-ensemble des accès en lecture seule.
- Le site secondaire s'occupe d'un sous-ensemble des accès en lecture seule (pour permettre un équilibrage de charge entre les deux premiers sites) et peut assurer la reprise des opérations d'écritures en cas de sinistre sur le site primaire (PRA).

- Le site de secours s'occupe uniquement du secours des objets numériques et des métadonnées (PCA).

3.3.6 Sécurité des données additionnelle

Chaque offre de stockage doit répondre aux enjeux définis dans la norme “NF Z 42-020” (Composant Coffre Fort Numérique ou “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 sécurité 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

Le logiciel Vitam doit pouvoir être instancié 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 nécessaire d'avoir de disposer d'un outillage permettant de suivre l'activité du système global :

- Gestion des logs centralisés
- 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 PostgreSQL ou 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 le logiciel Vitam pour ses propres usages (et donner de l'information à l'administrateur technique).

3.4 Architecture fonctionnelle

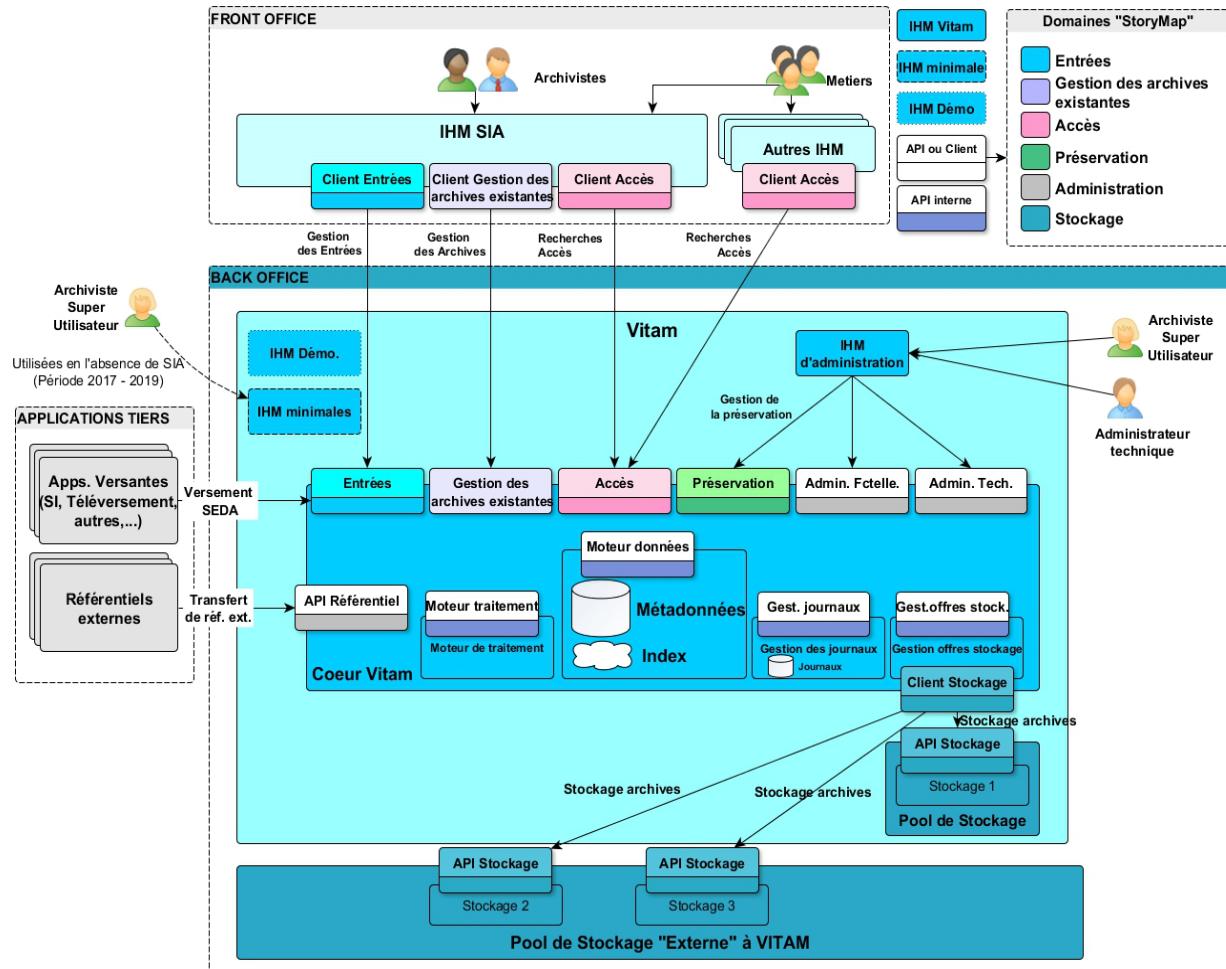


Fig. 3.2 – Architecture fonctionnelle cible de VITAM

VITAM est constitué 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 requêteable 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

Architecture applicative

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

4.1 Architecture applicable

4.1.1 Drivers de l'architecture

Les principes d'implémentation applicative ont pour but de faciliter, voire d'assurer les enjeux auxquels le système VITAM est confronté :

- Modèle Open-Source pour la réutilisation dans la sphère publique ainsi que conserver la maîtrise dans le temps du socle logiciel ;
- Couplage lâche entre les composants ;
- Nécessité de pouvoir disposer des 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

Le système VITAM est découpé 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 :

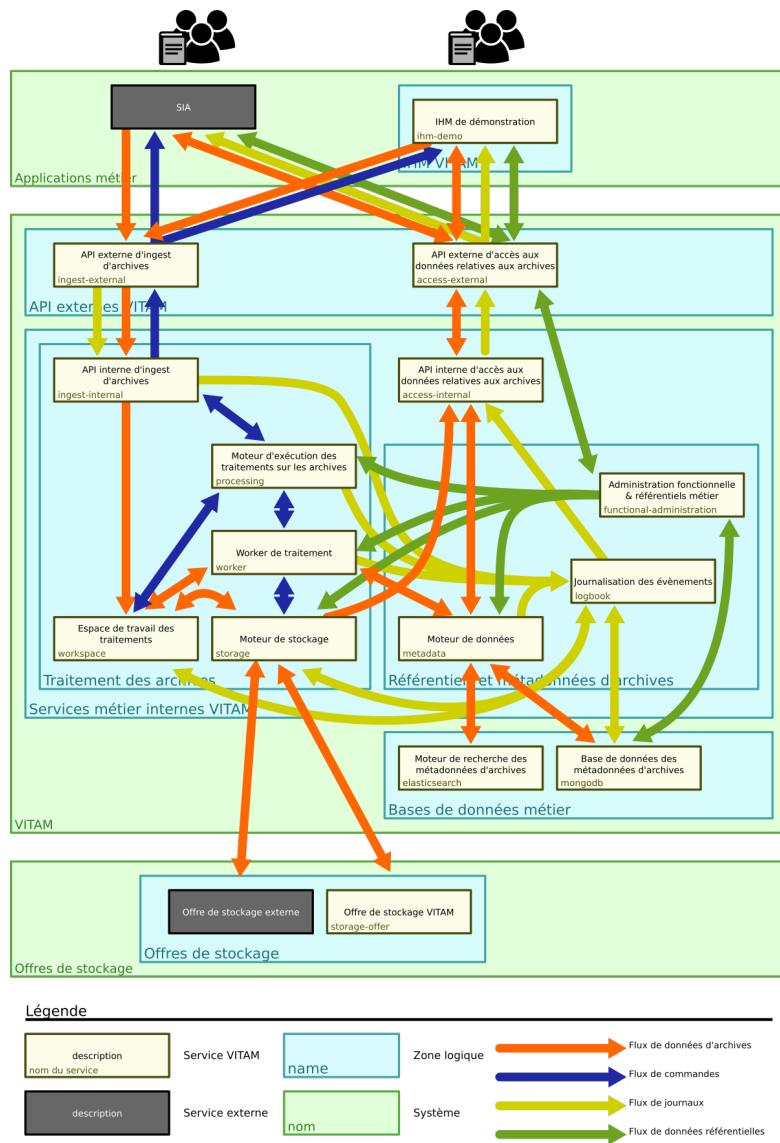


Fig. 4.1 – Architecture applicative et flux d'informations entre composants. Tous les composants jaunes sont fournis dans le cadre de la solution VITAM ; tous sont requis pour le bon fonctionnement de la solution, à l'exception de deux d'entre eux : ihm-demo et storage-offer-default (selon les choix de déploiement)

Chaque service possède un nom propre qui l'identifie de manière unique au sein du système VITAM.

Les services sont organisés en zones logiques :

- Les API externes contiennent les services exposés aux clients (ex : à un SIA) ; tout accès externe au système 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 contenu des archives (données stockées ou métadonnées associées) ;
- Les flux de commande : 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'événements (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, ...)

4.2 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*).

4.2.1 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 workflow en cours d'exécution

4.2.2 Moteur de stockage (storage)

Rôle :

- Stockage des données (Méta Données, 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

4.2.3 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

4.2.4 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 interactions avec l'espace de travail ;

Données gérées :

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

4.2.5 Moteur de données (metadata)

Rôle :

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

Fonctions :

- Fournit une API agrégant une technologie de base de données et un moteur d'indexation
- Fournit un cache des requêtes pour optimisation

Données gérées :

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

4.2.6 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 :

- Appel uniquement à partir de l'application

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

4.2.7 Gestion des référentiels (functional-administration)

Rôle :

- Gérer les référentiels métier de la plate-forme

Fonctions :

- Gestion du référentiel des formats (PRONOM)
- Gestion des règles de gestion des archives

Données gérées :

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

4.2.8 Offre de stockage par défaut (storage-offer-default)

Rôle :

- Fournir une offre de stockage par défaut permettant la persistance des objets sur un système de fichier local

Fonctions :

- Offre de stockage fournie par défaut
- Stockage simple des objets numériques sur un système de fichiers local

Données gérées :

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

4.2.9 Moteur d'entrée (ingest-internal)

Rôle :

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

Fonctions :

- Upload HTTP de fichiers au format SEDA
- Sas de validation antivirus des fichiers entrants
- Persistance du SEDA dans workspace
- Lancement des workflows de traitements liés à l'entrée dans processing

Données gérées :

- Aucune

4.2.10 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.

Données gérées :

- Aucune

4.2.11 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)

4.2.12 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» (gestion des résolutions différentes tout en restant sur des écrans « PC » (15" et +))
- Compatibilité avec les navigateurs actuels
- Pas d'applets/clients lourds

Données gérées :

- Aucune

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 distingue 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 :

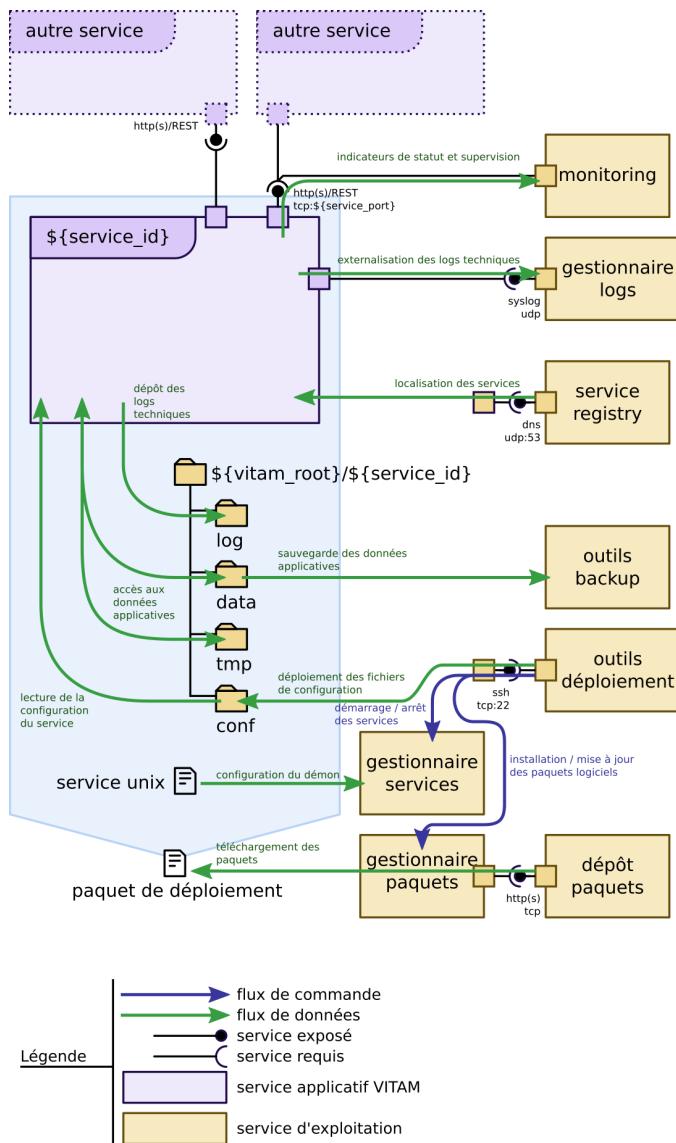


Fig. 5.1 – Environnement d'un service VITAM

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 27)) ;
 - Le déploiement, permettant l'orchestration du déploiement de la solution (Cf. [Déploiement de la solution](#) (page 29)) ;
 - 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 25)) ;
 - Le monitoring, avec (Cf. [Suivi de l'état du système](#) (page 32)) :
 - 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 23)) :
 - L'installation, avec l'exécution du gestionnaire de paquets de l'OS (Cf. [Packaging](#) (page 27)) ;
 - La gestion des fichiers de configuration de l'application via l'outil de déploiement (Cf. [Déploiement de la solution](#) (page 29)) ;
 - Le démarrage / arrêt des services (Cf. [Administration technique](#) (page 34)) ;
 - La sauvegarde / restauration des données applicatives (Cf. [Gestion des données du système](#) (page 35)).

5.1.1.3 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 58).

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 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”
- <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

Note : Dans le cadre de cette version de VITAM, les stratégies et outillage d’équilibrage de charge sont utilisés pour un seul service applicatif (logbook) ainsi que pour les COTS du système (MongoDB, Elasticsearch et Consul (serveurs)) ; ils seront affinés et complétés entre cette version et la version finale.

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).

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 id d'instance (`instance_id`).

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

1. Le composant amont effectue un appel à l'annuaire de service 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 58) 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
 - Connus 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 55).

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 23).

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 VITAM sont tous disponibles sous forme de packages natif aux distributions supportées (rpm pour CentOS 7, deb pour Debian 8 (Jessie)) ; 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 (SRPM pour CentOS, par exemple) ne seront pas fournis, les sources étant livrées sous un autre format.

5.1.4.2 Dépôts

Note : La typologique 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 2 dépôts internes différents :

- vitam-produit : ce dépôt héberge les packages des logiciels développés dans le cadre de VITAM ; ces packages sont maintenus par VITAM, et les licenses d'utilisations associées sont celles de VITAM.
- vitam-externe : 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 licenses d'utilisation restent celles des packages originaux.

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

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 CentOS

VITAM s'appuie sur les dépôts suivants :

- Centos 7 (Base, Extras) : il s'agit des dépôts standard de la distribution
- EPEL 7 (Extra Packages for Enterprise Linux) : il s'agit d'un dépôt maintenu par Fedora et fournissant un ensemble de packages complétant ceux de RHEL/Centos

5.1.4.2.2 Debian

VITAM s'appuie sur les dépôts suivants :

- Debian Jessie (dépôts main dans jessie, jessie-updates et security) : il s'agit des dépôts standard de la distribution
- jessie-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 CentOS

Conformément aux usages RPM de Centos/RHEL, 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 .rpmsave ou .rpmpnew, il ne sera pas utilisé la directive %config.

Prudence : A 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 résultat de commandes telles que `rpm -q1`.

5.1.4.3.2 Debian

Tout comme pour CentOS, 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'outil 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'outil de déploiement, se reporter à la section *sur l'outil de déploiement* (page 54)

Prudence : Dans cette version de VITAM, le déploiement ne supporte que des serveurs cibles comportant une seule IP, ce qui signifie notamment que le déploiement n'est pas compatible avec un réseau d'administration séparé du réseau d'exploitation. Cette limitation sera levée dans une prochaine version.

5.1.5.2 Contraintes et vue d'ensemble

Les zones logiques présentées dans *la section sur l'architecture applicative VITAM* (page 15) correspondent également aux zones de sécurité préconisées pour le déploiement de VITAM :

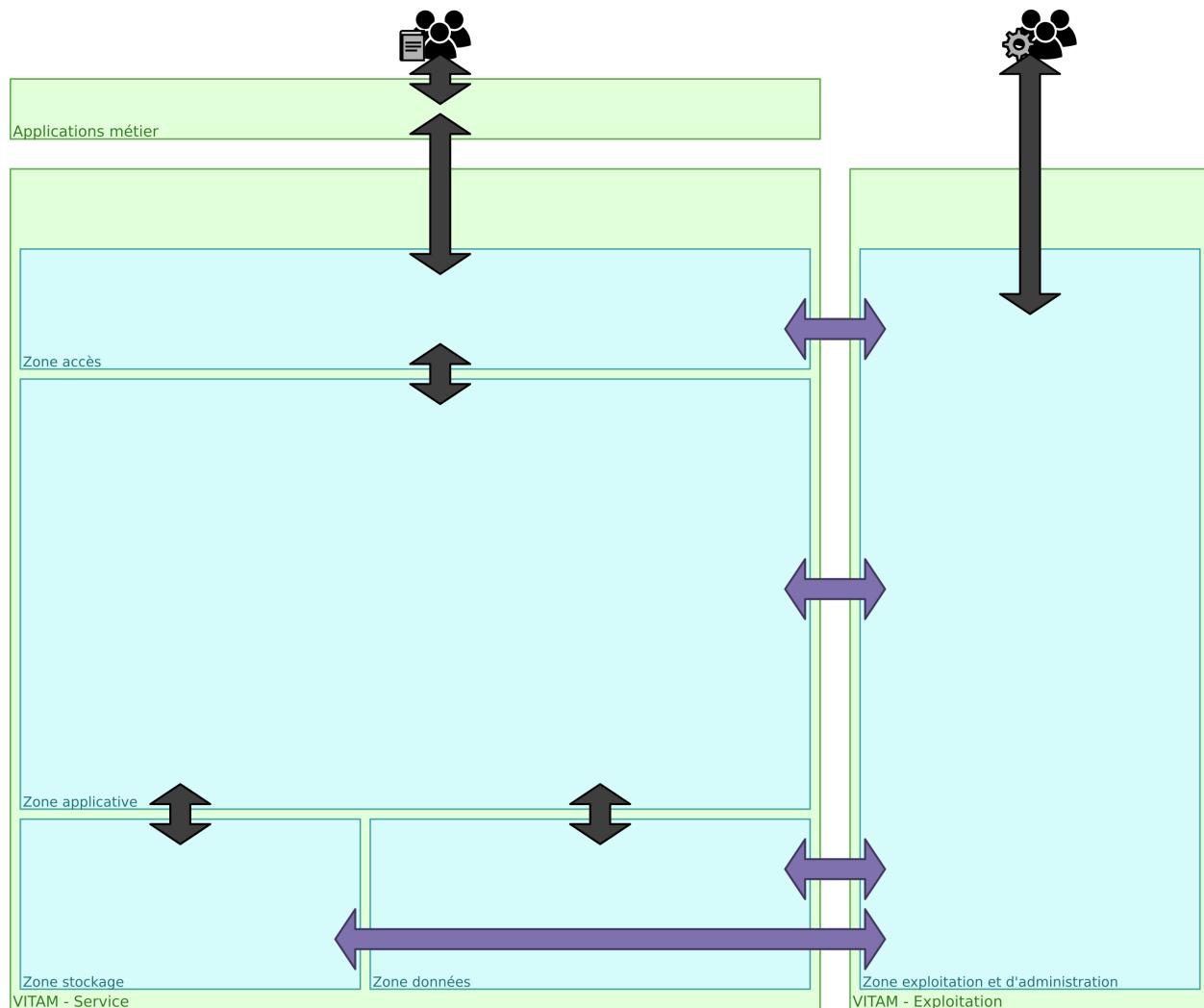


Fig. 5.2 – Déploiement VITAM : zones & principes de communication ; les utilisateurs métier archivistes sont présentés à gauche, et les exploitants technique à droite.

Voir aussi :

Ce découpage est repris dans [la présentation de l'architecture technique détaillée](#) (page 40).

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 supporté) ; 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 67).

Indice : A terme, VITAM supportera 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 transverse 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 29).

Voir aussi :

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

5.1.5.4 Principes de maj à chaud

La mise à jour à chaud n'est pas supportée dans cette version de la solution VITAM.

5.1.5.5 Multi-site

Le déploiement de VITAM sur plusieurs sites physiquement distincts n'est pas supporté dans cette version de la solution VITAM.

5.1.5.6 Support de l'élasticité

Un déploiement de VITAM sur une infrastructure élastique 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 ressources.
- /admin/v1/version : statut renvoyant les informations relatives à la version.

Voir aussi :

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

Note : A terme, ces API de supervision pourront être exposées sur un réseau d'administration disjoint du réseau de service.

5.1.6.2 Métriques

Chaque composant VITAM doit permettre l'envoi d'un certain nombre de métriques soit dans les logs de l'application, soit dans une base de données Elasticsearch ; ces métriques sont de 3 types différents :

- Les métriques relatives aux statistiques d'accès des interfaces REST :
 - Fréquence d'appel sur les dernières 1, 5 et 15 minutes ;
 - Nombre de résultats selon le code HTTP renvoyé ;
 - Avec un sampling des temps de réponses biaisé 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
 - Etat 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 51).

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](#)⁸ (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 :

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'évènements 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 23))). 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).

8. <http://logback.qos.ch/manual/appenders.html#SyslogAppender>

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 46).

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.

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 23))) : 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 23))). 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 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

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 (démarrage / status / arrêt).

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 services 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 (“cron”)

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`⁹.

5.1.8 Gestion des données du système

Prudence : Les principes de sauvegarde / restauration présentés dans cette section ne sont pas les principes cibles ; 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 sont en cours de définition et d'implémentation ; ils consistent majoritairement en la pérennisation des données à ne pas perdre dans les offres de stockage, et seront décrits plus en détails dans une future version de ce document.

5.1.8.1 Sauvegarde

5.1.8.1.1 Vue d'ensemble

Prudence : La sauvegarde des données du système se fait à froid dans cette version du système ; par conséquent, une plage d'indisponibilité complète du service est à prévoir (par la suite, cette plage d'indisponibilité sera considérée comme étant la nuit entre 20h00 et 08h00, cette durée pouvant être adaptée en fonction de la volumétrie des sauvegardes à réaliser).

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

- Une phase initiale de suppression des possibilités d'écritures externes dans les bases de données (arrêt des composants frontaux) ;
- 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 ;
- Une phase d'export des bases de données sous forme de fichiers ;
- Une phase de sauvegarde des fichiers (avec a minima les fichiers d'archives et fichiers d'export des bases de données) ;
- Une phase de redémarrage ordonné des services VITAM.

Voir aussi :

Ce processus est décrit plus en détails dans le DEX

5.1.8.1.2 Principe d'export MongoDB pour une base répartie

La procédure recommandée pour sauvegarder un point temporel d'une base de donnée répartie est complexe et nécessite de nombreux points de synchronisation. Dans le cadre de la sauvegarde à froid, il est préférable de sauvegarder les données directement depuis le serveur principal d'un lot de réPLICATION.

En outre, la méthode de sauvegarde à utiliser doit être adaptée à la volumétrie des bases de données.

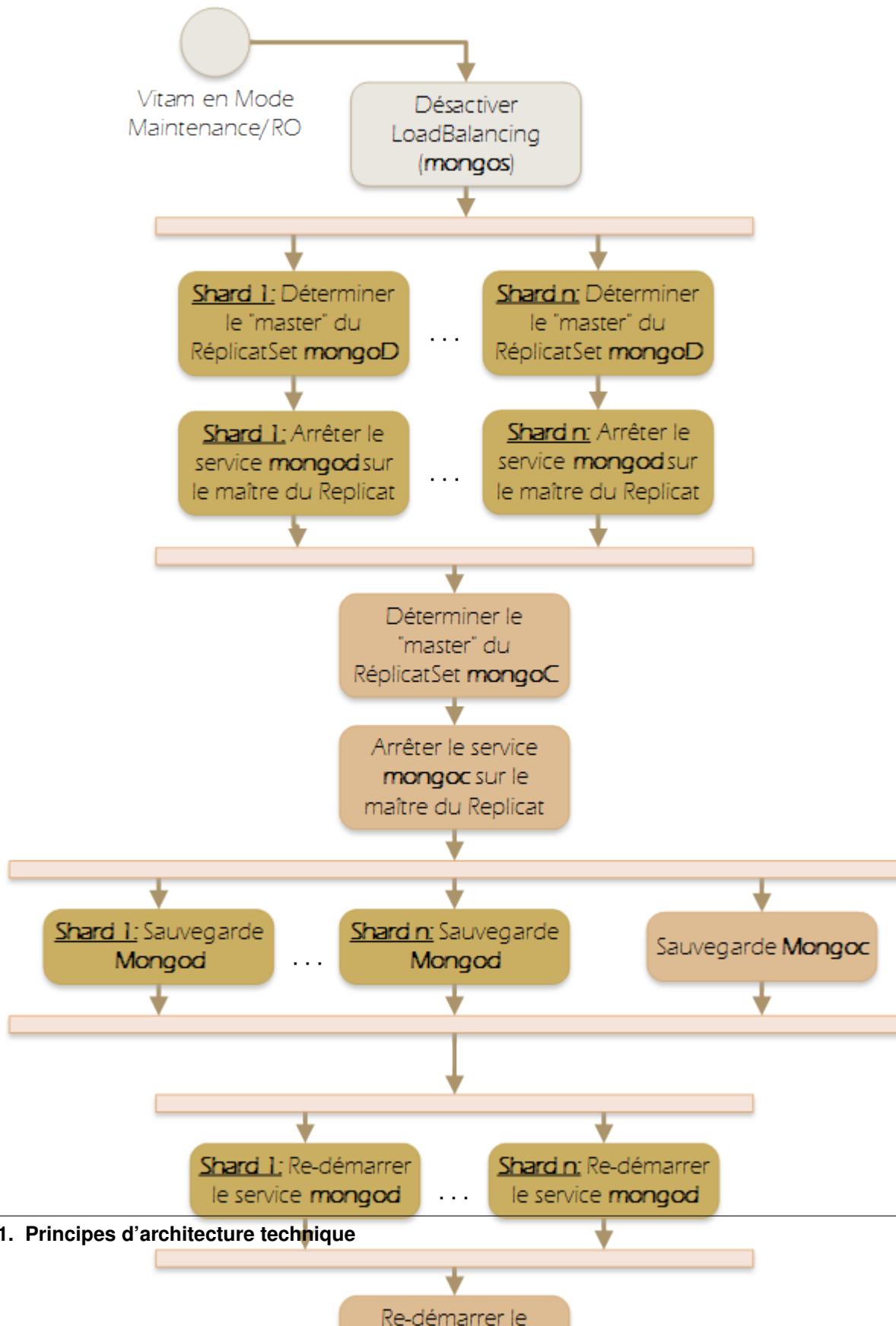
Dans les grandes lignes, il faut :

1. Arrêter les services d'entrée **mongos**.
2. Arrêter les services de données **mongod**.

9. <https://www.freedesktop.org/software/systemd/man/systemd.timer.html>

3. Arrêter les services de configuration **mongoc**.
4. Sauvegarder les bases de données **mongod** et **mongoc**
5. Relancer les services **mongod**
6. Relancer les services **mongoc**
7. Relancer les services d'entrée **mongos**

Procédure de sauvegarde MongoDB pour Vitam



Voir aussi :

Ce processus est décrit plus en détails dans [*la description des besoins en ordonnancement*](#) (page 57)

5.1.8.1.3 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.

5.1.8.2 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.
 - Arrêter les services subsistant.
2. Restaurer les dossiers précédemment sauvegardés à leur emplacement respectif.
3. Démarrer les services suivant la procédure de démarrage VITAM.

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 :

- Etat 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 releases notes de chaque version de VITAM.

5.3.1 Fournies

5.3.1.1 COTS

- [MongoDB](#)¹⁰ : base de données orientée documents
- [Elasticsearch](#)¹¹ (+ plugins) : base d'indexation
- [Cerebro](#)¹² : IHM d'administration d'Elasticsearch
- [MongoClient](#)¹³ : IHM d'administration de MongoDB
- [Curator](#)¹⁴ : maintenance des index d'elasticsearch
- [Logstash](#)¹⁵ (+ plugins) : agrégation et traitement des logs

10. <https://www.mongodb.com/fr>

11. <https://www.elastic.co/products/elasticsearch>

12. <https://github.com/lmenezes/cerebro>

13. <https://www.mongoclient.com/>

14. <https://www.elastic.co/guide/en/elasticsearch/client/curator/current/index.html>

15. <https://www.elastic.co/fr/products/logstash>

- [Kibana](#)¹⁶ : dashboards et recherche des logs techniques
- [Consul](#)¹⁷ : annuaire de services
- [Siegfried](#)¹⁸ : identification des formats de fichiers

5.3.1.2 Bibliothèques structurantes

- [Jetty](#)¹⁹ : moteur de servlet

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

5.3.2 Requises

- Java ([JRE](#)) 8

Voir aussi :

Pour chaque version du système VITAM :

- les composants fournis ou installés par dépendance sont précisés dans la documentation d'installation ([DIN](#)) ;
- la liste des bibliothèques opensources incluses est précisée 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 abordées plus en détail dans les paragraphes dédiés aux différents composants.

16. <https://www.elastic.co/fr/products/kibana>

17. <https://www.consul.io/>

18. <http://www.itforarchivists.com/siegfried>

19. <https://eclipse.org/jetty/>

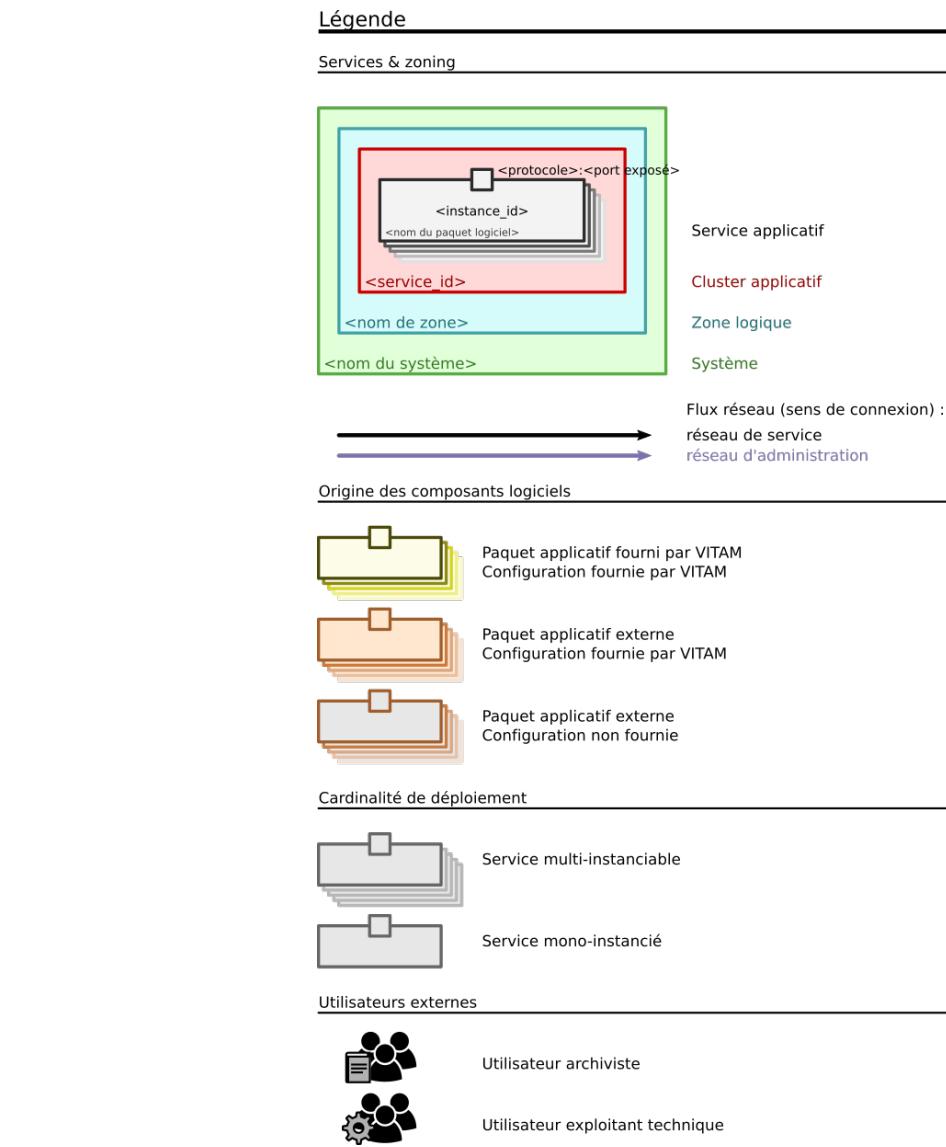


Fig. 5.4 – Architecture technique : légende

5.4.1 Flux métier

Les flux réseaux “métier” sont divisés en 3 schémas pour plus de clarté ; tout d’abord, les flux généraux :

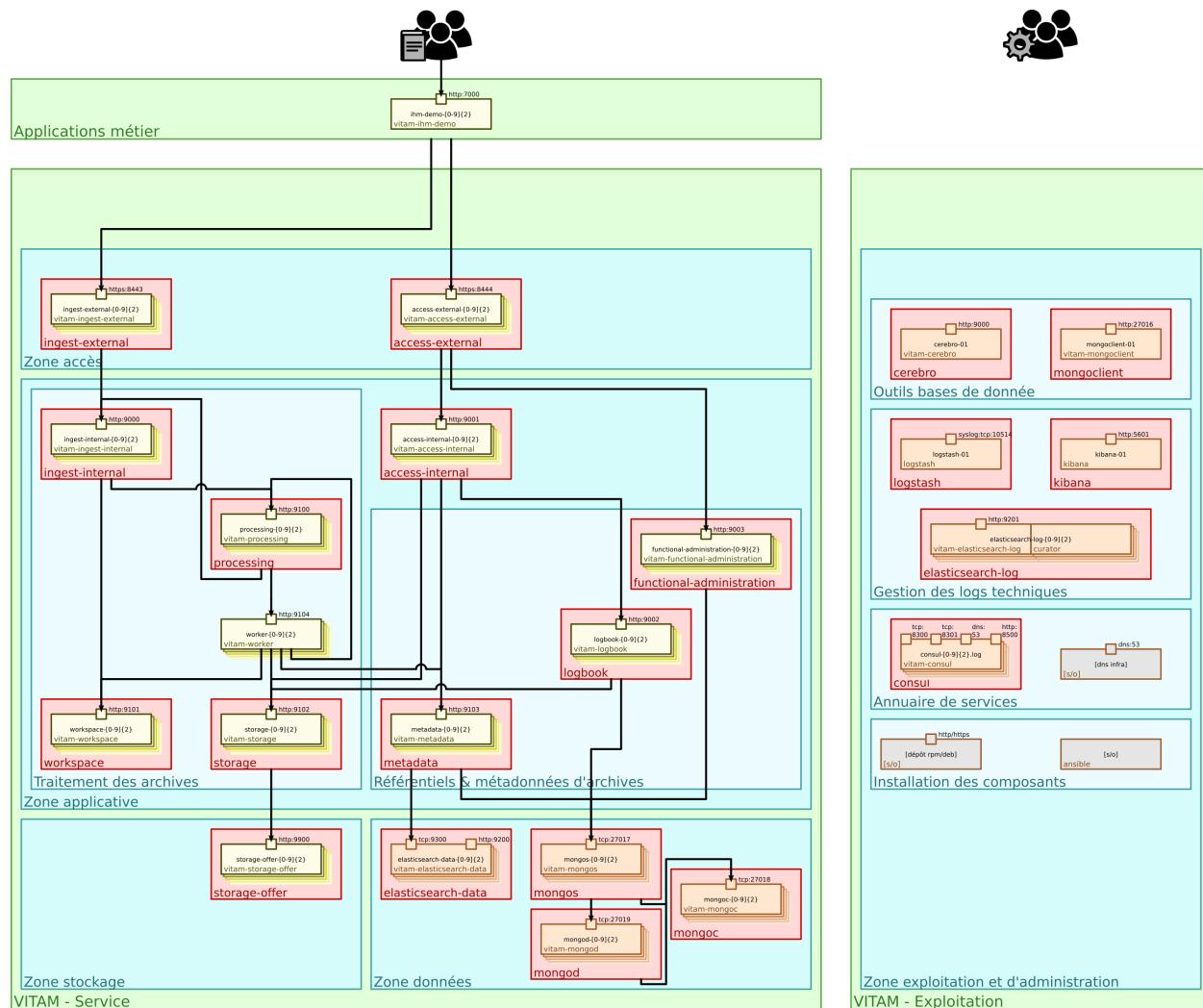


Fig. 5.5 – Architecture technique : flux (1/5 : flux métiers généraux)

Ensuite, les flux dédiés au dépôt des journaux dans le composant “logbook” :

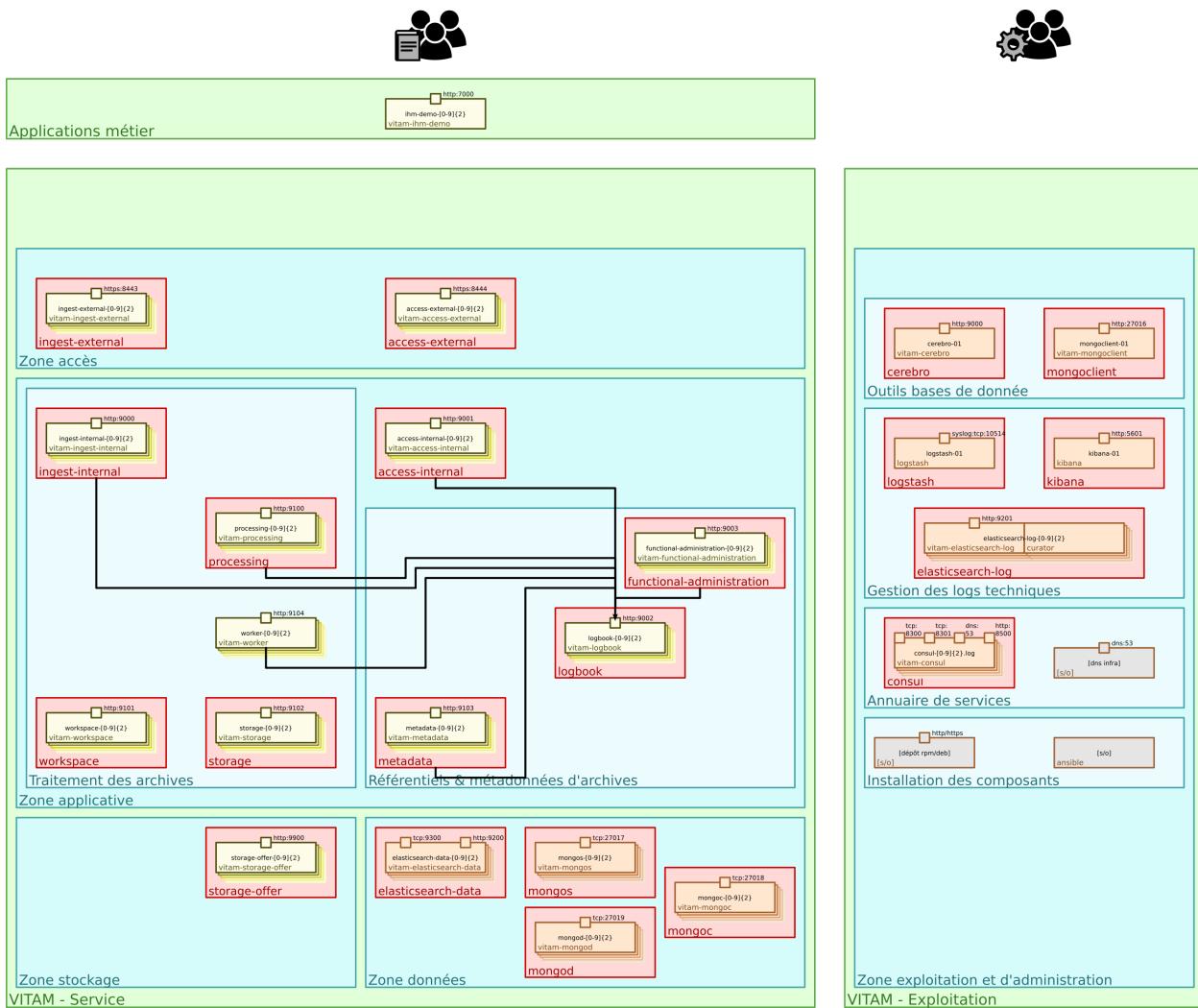


Fig. 5.6 – Architecture technique : flux (2/5 : flux métiers de dépôt des journaux)

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

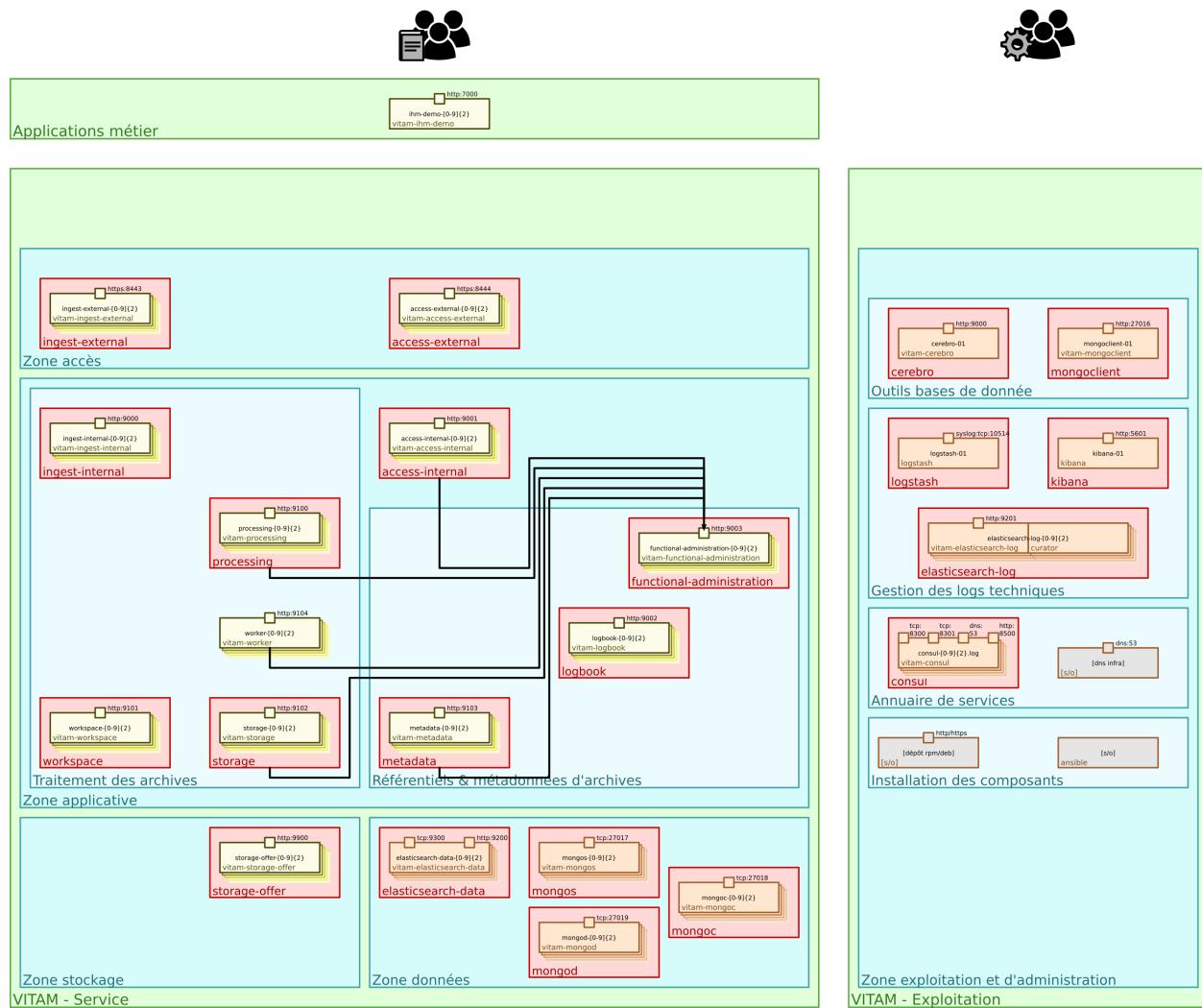


Fig. 5.7 – Architecture technique : flux (3/5 : flux métiers de lecture des référentiels métier)

5.4.2 Flux exploitation

Les flux réseau “exploitation” correspondent aux flux des utilisateurs exploitants vers les outils d’exploitation fournis par Vitam :

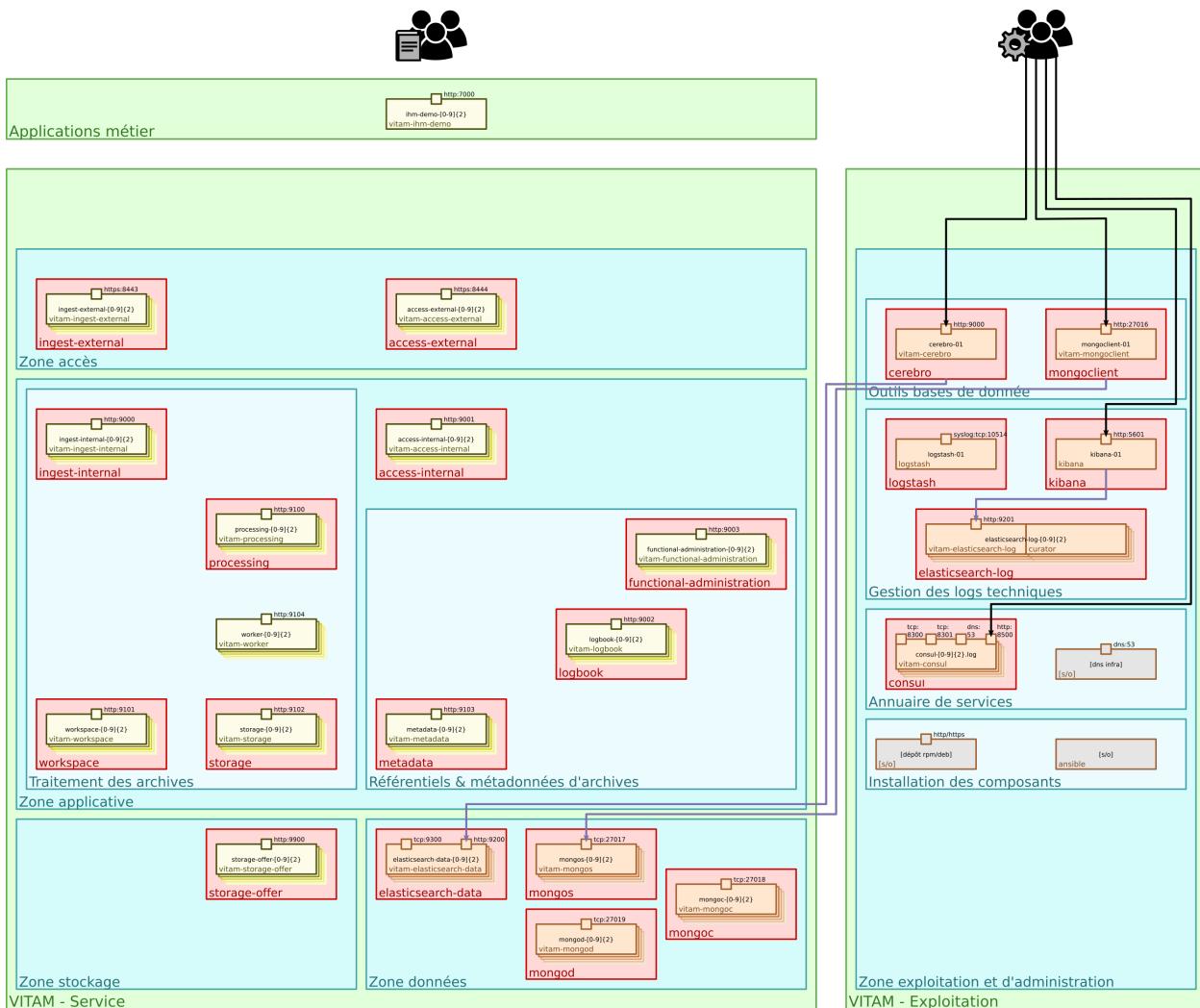


Fig. 5.8 – Architecture technique : flux (4/5 : flux des outils d'exploitation)

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 :

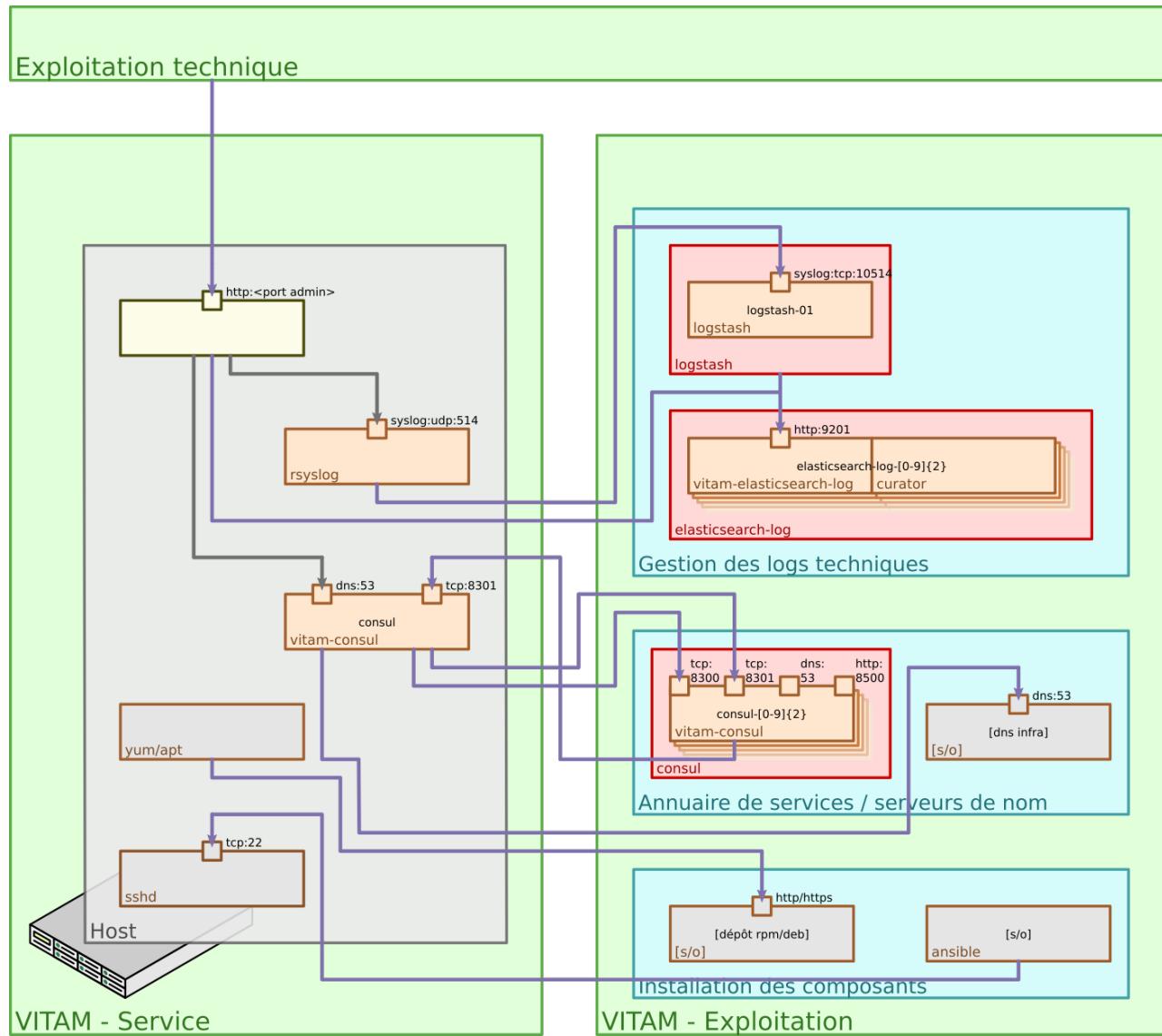


Fig. 5.9 – Architecture technique : flux (5/5 : flux du socle technique). Seul le port exposant les services d’administration/exploitation est représenté sur le composant VITAM présenté dans cette figure.

5.5 Concentration et exploitation des logs applicatifs

5.5.1 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.

5.5.2 Modèle générique

On peut noter les composants suivants :

- Emetteur 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 requétable)
 - 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 :

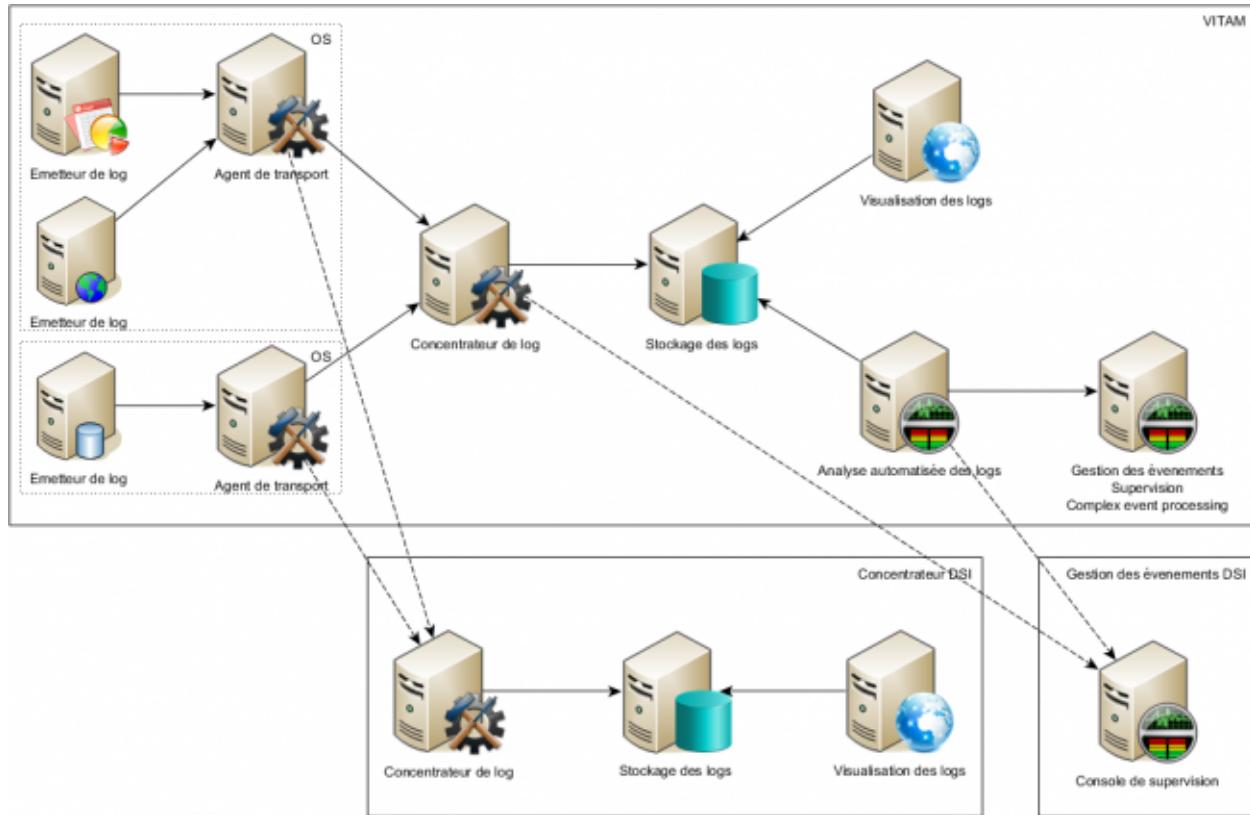


Fig. 5.10 – Architecture générique d'un système de gestion de logs.

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.

5.5.3 Choix des implémentations

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

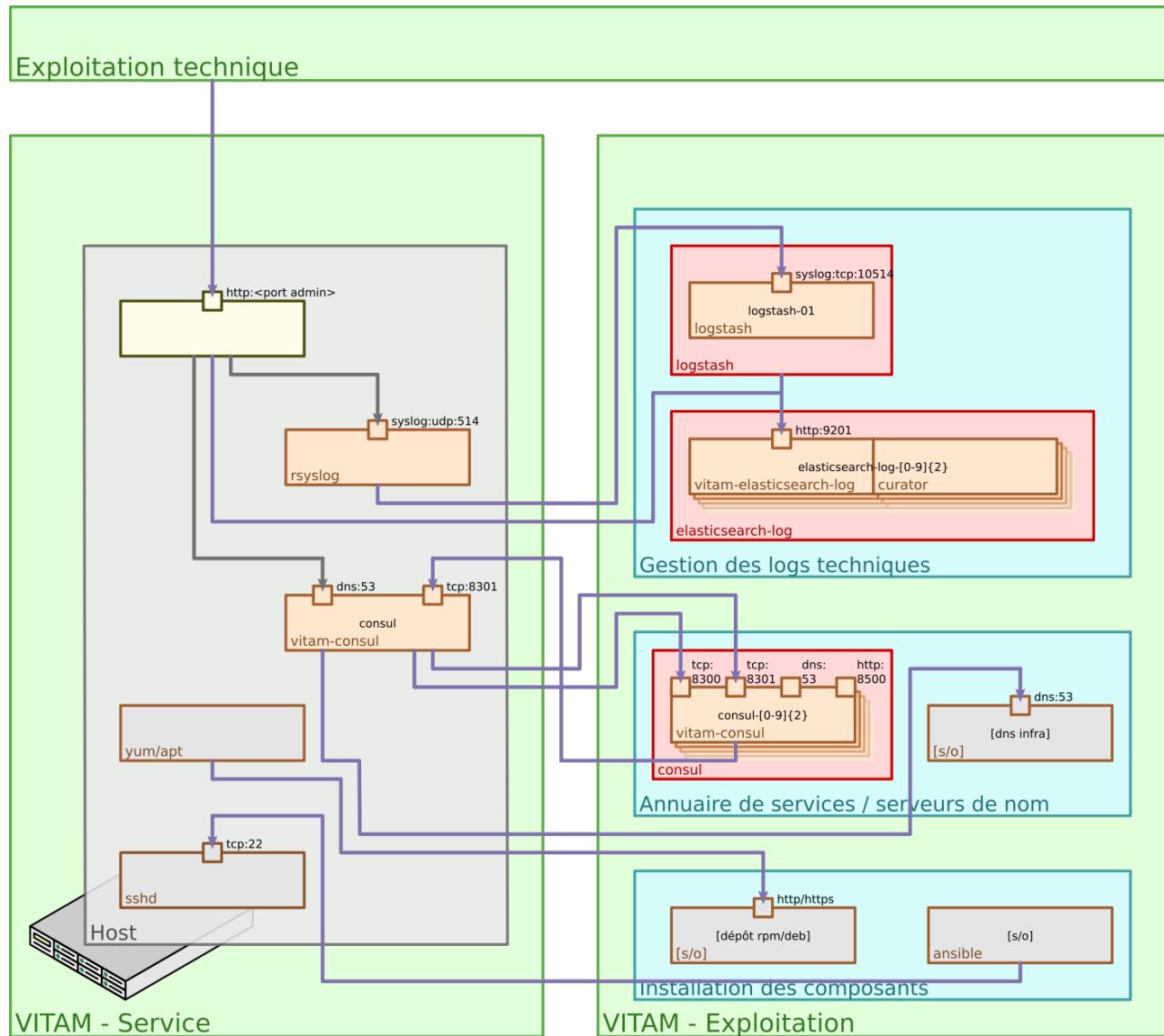


Fig. 5.11 – Architecture du sous-système de centralisation des logs

5.5.3.1 Emetteur de logs

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

- Pour les composants logiciels Java VITAM : via l'appender logback SyslogAppender²⁰ ;
- Pour les script unix : via la commande `logger`.

Un émetteur de logs a les responsabilités suivantes :

- Le formatting du message selon le format de log préconisé pour l'application ;
- L'envoi des logs à l'agent de transport de logs selon le protocole défini dans *la section présentant les principes de log* (page 32)

Note : Dans cette version de la solution VITAM, l'envoi des stacktraces Java à l'agent de transport de log est désactivé.

20. <http://logback.qos.ch/manual/appenders.html#SyslogAppender>

5.5.3.2 Agent de transport de log

L'agent de transport de log est `rsyslog`. Il est installé localement à 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"²¹) ;
- 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 32).

5.5.3.3 Concentration de logs

Le concentrateur de logs est `logstash`. Il est instancié de manière unique, 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

5.5.3.4 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).

Ce cluster est configuré de la manière suivante :

- Taille du cluster (pour les déploiements VITAM de taille importante, ce nombre pourra être amené à évoluer (Cf. les abaques *fournies plus loin* (page 68))) :
 - Nombre nominal de noeuds : 2 ;
 - Nombre nominal de shards primaires par index : 4 ;
 - Nombre nominal de replica : 1 ;

Note : Ces paramètres ne permettent pas de se préparer contre la perte d'un noeud elastic search, et 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 replica, en veillant à ce que le nombre de shards primaires ne soit jamais inférieur au nombre de noeuds du cluster, et que le nombre de replica ne soit jamais supérieur au nombre de noeuds du cluster - 1.

21. <http://www.rsyslog.com/doc/v8-stable/concepts/queues.html>

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.

5.5.3.4.1 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 Curator²². Par défaut, l'outil est livré avec la configuration suivante :

- 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.

22. <https://www.elastic.co/guide/en/elasticsearch/client/curator/4.0/index.html>

5.5.3.5 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.

Indice : 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 Security²³), sous réserve d'une étude de compatibilité de la solution choisie.

5.5.4 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 possibles :

- 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.

5.5.5 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).

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.

5.6 Métriques applicatives

5.6.1 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.

5.6.2 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.
- Stockage des métriques : il s'agit du composant stockant les métriques (de manière plus ou moins requêtéable).

23. <https://www.elastic.co/products/x-pack/security>

- Visualisation des métriques : il s'agit du composant (souvent IHM) qui permet la recherche et la visualisation des métriques.

5.6.3 Choix des implémentations

5.6.3.1 Enregistreur de métriques

Dans le système VITAM, l'enregistrement de métriques s'effectue uniquement dans les composants logiciels Java à l'aide de la librairie [Dropwizard metrics](#)²⁴.

Les plugins suivants sont utilisés pour leur métriques respectives :

- [Dropwizard Jersey integration](#)²⁵ pour les métriques Jersey.
- [Dropwizard JVM integration](#)²⁶ pour les métriques JVM.

L'enregistreur de métriques possède un registre interne qui peut stocker différentes métriques : **Gauges**, **Timer**, **Meter**, **Counter** ou **Histograms**. Ces métriques seront collectées dans le temps par le/les reporter(s) de métriques.

Les métriques Jersey sont automatiquement générées par 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](#)²⁷.

5.6.3.2 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
- Un reporter ElasticSearch issue de la librairie [metrics.elasticsearch reporter](#)²⁸.

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.

5.6.3.3 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).

Ce cluster est configuré de la manière suivante :

- Taille du cluster (pour les déploiements VITAM de taille importante, ce nombre pourra être amené à évoluer (Cf. les abaques *fournies plus loin* (page 68))) :
 - Nombre nominal de noeuds : 2 ;
 - Nombre nominal de shards primaires par index : 4 ;

24. <http://metrics.dropwizard.io/3.1.0/>

25. <http://metrics.dropwizard.io/3.1.0/manual/jersey/#instrumenting-jersey-2-x>

26. <http://metrics.dropwizard.io/3.1.0/manual/jvm/>

27. <http://metrics.dropwizard.io/3.1.0/manual/core/>

28. <https://github.com/elastic/elasticsearch-metrics-reporter-java>

- Nombre nominal de replica : 1 ;

Note : Ces paramètres ne permettent pas de se parer contre la perte d'un noeud elasticsearch, et correspondent à un compromis en terme d'usage des resources 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 replica, en veillant à ce que le nombre de shards primaires ne soit jamais inférieur au nombre de noeuds du cluster, et que le nombre de replica 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 métriques correspond à 1 jour de métriques (déterminé à partir du timestamp de la métrique). Les index définis sont les suivants :
 - metrics-vitam-jersey-YYYY.MM.dd pour les métriques de Jersey, avec un champ *name* automatiquement généré sous la forme :
uri :http_method :consumed_types :produced_types :metric_type
 - 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.

5.6.3.3.1 Gestion des index

La gestion des index est réalisée par l'application [Curator](#)²⁹. Par défaut, l'outil est livré avec la configuration suivante :

- Durée de maintien des index “online” : 7 jours ; cela signifie qu'au bout de 7 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 : 30 jours ; au bout de cette durée, les index seront supprimés.

5.6.3.4 Visualisation des métriques

La visualisation des métriques 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.

Indice : 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](#)³⁰), sous réserve d'une étude de compatibilité de la solution choisie.

29. <https://www.elastic.co/guide/en/elasticsearch/client/curator/4.0/index.html>

30. <https://www.elastic.co/products/x-pack/security>

5.6.4 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).
- Il n'existe à ce jour que 3 types de métriques, **Meter**, **Timer** et **ExceptionMeter** supportés par le plugin Jersey Dropwizard Metrics.

5.7 Outilage de déploiement

5.7.1 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 officielle³¹ pour la liste exhaustive des dépendances requises).
- 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

Indice : Sur Centos 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.

5.7.2 Architecture de l'outil

On dispose de 2 types de playbooks :

- 1 playbook de déploiement (`vitam-ansible`) qui est le cœur du déploiement

³¹. https://docs.ansible.com/ansible/intro_installation.html

- 1 playbook de déploiement (`vitam-ansible-extra`) qui contient des éléments potentiellement utiles, mais non nécessaires au fonctionnement du système.

On dispose de 2 types de rôles :

- rôle “helper” qui est appelé par les autres rôles et qui n'est pas contenu dans les playbook ;
- 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 Jinja 2).

5.7.3 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](#).

A ce jour, seuls les composants frontaux (i.e. faisant partie de la zone Accès) nécessitent un certificat. Pour tout certificat, l'intégralité des certificats des CA de la chaîne de certification devra également être fournie, ainsi que l'URL des CRL associées.

Voir aussi :

La liste des secrets nécessaires au bon fonctionnement de VITAM est décrite dans la [section dédiée](#) (page 71).

5.8 Service registry

5.8.1 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”³²](#)) ; 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.

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 ;

³². <https://www.consul.io/docs/guides/outage.html>

- 1 client par serveur hébergeant un service VITAM.

Note : 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 très déconseillé.

5.8.2 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 cache 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 DNS³³);
- Dans le cas contraire, la requête est transmise aux serveurs DNS configurés dans la liste des recursors³⁴.

Note : Consul a pour l'instant été configuré en mode allow_stale = false (cf. la directive de configuration³⁵), 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).

5.8.3 Packaging

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/vitam-consul/bin/consul afin de permettre à ce dernier d'exposer une interface DNS sur le port 53 sans pour autant nécessiter les droits root.

33. <https://www.consul.io/docs/agent/dns.html>

34. <https://www.consul.io/docs/agent/options.html#recursors>

35. https://www.consul.io/docs/agent/options.html#allow_stale

5.8.4 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 32).

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.

5.9 Dépendances aux services d'infrastructures

5.9.1 Ordonnanceurs techniques / batchs

5.9.1.1 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`³⁶ sur chaque serveur.

Voir aussi :

Plus de détails sont disponibles dans *la présentation de curator* (page 59)

5.9.1.2 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.

Prudence : Dans cette release, ce job est le seul à être lancé par le crontab système ; il sera migré en timer `systemd` prochainement.

5.9.1.3 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 ;

5.9.1.4 Cas de la sauvegarde

Se référer au *DAT* (dans *la section dédiée* (page 35)) et *DEX*.

5.9.2 Socles d'exécution

5.9.2.1 OS

Seules deux distributions Linux sont supportées à ce jour :

- CentOS 7

36. <https://www.freedesktop.org/software/systemd/man/systemd.timer.html>

- Debian 8 (jessie)

SELinux doit être configuré en mode permissive ou disabled.

5.9.2.2 Middlewares

- Java : JRE 8 ; les versions suivantes ont été testées :
 - OpenJDK 1.8.0, dans la version présente dans les dépôts officiels au moment de la parution cette release de Vitam (Centos et Debian en 1.8.0_131)

5.10 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 pour chaque composant sont à adapter en fonction des cas d'utilisation des systèmes (ex : archivage définitif VS archivage courant).

5.10.1 Access-external

Type : Composant VITAM Java

Données stockées :

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

Typologie de consommation de resources :

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

5.10.2 Access-internal

Type : Composant VITAM Java

Données stockées :

- Aucune

Typologie de consommation de resources :

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

5.10.3 Consul

Type : COTS

Données stockées :

- Etat 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.

5.10.3.1 Architecture de déploiement

L'architecture de déploiement conseillée correspond aux principes présentés dans *la section d'introduction à Consul* (page 55) :

- 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 évènements 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

5.10.4 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 officielle³⁷.

Type : COTS

Données stockées :

- Aucune

Typologie de consommation de resources :

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

5.10.5 Elasticsearch-data

Cluster d'indexation dédié aux données métier

Type : COTS

Données stockées :

- Index de recherche des données d'archive

Typologie de consommation de resources :

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

5.10.5.1 Architecture de déploiement

Dans le déploiement actuel, 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, $n > 0$).

5.10.5.2 Monitoring

Le monitoring d'elasticsearch est possible :

- soit à partir des API http (notamment les ‘cat APIs’³⁸, les API de gestion des index³⁹ ou les API de gestion du cluster⁴⁰) ;
- soit en utilisant le composant Cerebro (Cf. la page officielle⁴¹) installé dans le cadre de la solution logicielle VITAM.

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

38. <https://www.elastic.co/guide/en/elasticsearch/reference/2.4/cat.html>

39. <https://www.elastic.co/guide/en/elasticsearch/reference/2.4/indices.html>

40. <https://www.elastic.co/guide/en/elasticsearch/reference/2.4/cluster.html>

41. <https://github.com/lmenezes/cerebro>

5.10.6 Elasticsearch-log

Cluster dédié aux données métier

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

5.10.6.1 Architecture de déploiement

Voir aussi :

Se reporter à *Elasticsearch-data* (page 60) pour les informations générales concernant elasticsearch.

Dans le déploiement actuel, 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, $n > 0$).

5.10.7 Functional-administration

Type : Composant VITAM Java

Données stockées :

- Fichiers temporaires : fichiers de chargement des référentiels

Typologie de consommation de ressources :

- CPU : moyenne
- Mémoire : forte
- Réseau : forte
- Disque : moyen : utilisation du répertoire temporaire pour des chargements de fichiers de référentiel

5.10.8 Ingest-external

Type : Composant VITAM Java

Données stockées :

- 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 66) pour l’identification des formats de fichier.

5.10.8.1 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

5.10.9 Ingest-internal

Type : Composant VITAM Java

Données stockées : Aucune

Typologie de consommation de resources :

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

5.10.10 Kibana

Kibana est une application web permettant de faire des recherches et de construire des dashboards à partir des données des logs techniques.

Type : COTS

Données stockées : Aucune

Typologie de consommation de resources :

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

5.10.10.1 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).

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

5.10.11 Logbook

Type : Composant VITAM Java

Données stockées : Aucune

Typologie de consommation de ressources :

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

5.10.12 Logstash

Type : COTS

Données stockées : Aucune

Typologie de consommation de ressources :

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

5.10.13 Metadata

Type : Composant VITAM Java

Données stockées : Aucune

Typologie de consommation de ressources :

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

5.10.14 Mongodb

Base de données dédiée aux données métier.

Type : COTS

Données stockées :

- Données d'archives
- Journaux métier
- Référentiels métier

Typologie de consommation de ressources :

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

5.10.14.1 Architecture de déploiement

5.10.14.1.1 Architecture 1 noeud

- 1 serveur mongodb :
 - 1 noeud mongod

5.10.14.1.2 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 composé d'un noeud primaire et de deux noeuds secondaires. (Règles Mongodb de production)
 - 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 d'au moins 3 noeuds
- **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).

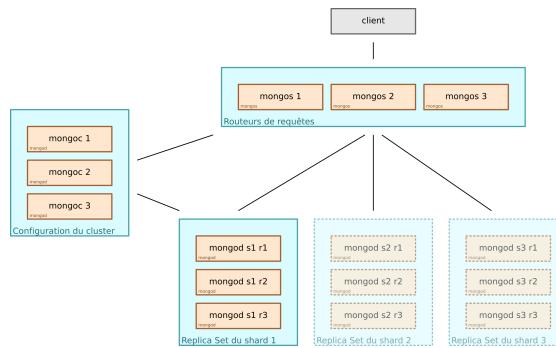


Fig. 5.12 – Déploiement d'un cluster Mongo DB avec sharding.

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 :

- 3 serveurs config / service, chacun hébergeant :
 - 1 noeud mongos (service)
 - 1 noeud mongoc (Replica Set de configuration)
- $n \times r$ serveurs, chacun hébergeant :
 - 1 noeud mongod

Note : Dans le cadre de cette version du système VITAM, seul un shard sera configuré, mais de telle manière à pouvoir instancier d'autres shards sans modification conséquente du déploiement et de la configuration.

5.10.14.1.3 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 Replica Set de données (mongod)

5.10.15 Processing

Type : Composant VITAM Java

Données stockées : Aucune

Typologie de consommation de ressources :

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

5.10.16 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.

5.10.16.1 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.

5.10.17 Storage

Type : Composant VITAM Java

Données stockées : Aucune

Typologie de consommation de ressources :

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

5.10.18 Storage-offer

Type : Composant VITAM Java

Données gérées :

- Données d'archives

Typologie de consommation de ressources :

- CPU : moyen
- Mémoire : fort (principalement pour le cache I/O)
- Réseau : fort
- Disque : fort (stockage pérenne des données d'archive)

5.10.19 Worker

Type : Composant VITAM Java

Données gérées :

- Aucune

Typologie de consommation de resources :

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

Voir aussi :

Ce composant fait également appel *au composant Siegfried* (page 66) pour l'identification des formats de fichier.

5.10.19.1 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.

5.10.20 Workspace

Type : Composant VITAM Java

Données stockées : Aucune

Typologie de consommation de resources :

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

5.11 Guidelines de déploiement

Les principes de zoning associés à l'architecture du système VITAM ont été présentés *lors de la description des principes de déploiement* (page 29) ; 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 ;

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

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

Note : Ces principes de colocation sont les préconisations initiales relatives à cette version du système VITAM ; ils seront revus suite aux campagnes de tests de performance en cours.

5.12 Utilisation des resources informatiques

Prudence : Les abaques de dimensionnement sont en cours de définition ; les éléments présentés ci-dessous correspondent aux cibles de dimensionnement en cours de test ou prévus pour les tests de performance. Les indications de volumétrie qui y sont présentées sont purement indicatives, et relatives au système VITAM dans son état actuel, à savoir au tout début de la phase de tuning et d'optimisations des performances. En outre, ils se basent sur des tests sur systèmes virtualisés (avec une surallocation CPU de 4), et comprennent une réservation de ressources pour chaque OS.

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

Configurations indicatives pour le développement et/ou la recette d'applications métier :

- Configuration “xsmall” : 1 VM (4 vCPU, 16 Go RAM) : adapté à un poste de développement ; ce déploiement ne comprend pas les composants d'exploitation de la solution VITAM ;
- Configuration “small” : 3 VM (4 vCPU, 16 Go RAM) : adapté à un environnement de recette simple d'application métier utilisant VITAM ;

Configurations indicatives pour la production :

- Configuration “medium” : déploiement à 11 VM (36 vCPU, 72 Go RAM) : déploiement simple pour des volumétries moyennes ; seules les bases de données et les offres de stockage sont multi-instanciées ;
- Configuration “large” : déploiement à 24 VM (72 vCPU, 150 Go RAM) : déploiement résilient pour des volumétries plus importantes ; ce déploiement comprend au moins deux instances pour tous les composants le supportant ;
- Configuration “xlarge” : déploiement à 45 VM (160 vCPU, 360 Go RAM) : déploiement pour de fortes volumétries (ordre de grandeur des capacités d'ingest : > 50 To / an, > 100.10^6 objets / an) ;

5.13 Matrice des flux

Voir aussi :

La matrice complète des flux s'appuie sur les schémas présentés dans [la description de l'architecture technique](#) (page 40), en y ajoutant notamment les flux internes au cluster.

Tableau 5.1 – Matrice des flux inter-zones

Zone source	Zone cible	Proto- cole	Port cible	Description
Externe	Accès	https	8443	Accès à ingest-external
Externe	Accès	https	8444	Accès à access-external
Accès	Applicative	http	9000	Accès à ingest-internal
Accès	Applicative	http	9001	Accès à access-internal
Accès	Applicative	http	9003	Accès à functional-administration
Applicative	Stockage	http	9900	Accès à storage-offer
Applicative	Données	tcp	9300	Accès à elasticsearch-data
Applicative	Données	tcp	27017	Accès à mongodb
Accès / Applicative / Stockage / Données	Administration	sys-log/tcp	10514	Envoi des logs au concentrateur
Accès / Applicative / Stockage / Données	Administration	tcp	8300	Appels RPC consul
Accès / Applicative / Stockage / Données	Administration	tcp	8301	Gossip Consul
Accès / Applicative / Stockage / Données	Administration	http	9201	Envoi des métriques à Elasticsearch
Administration	Accès / Applicative / Stockage / Données	tcp	8301	Gossip Consul
Administration	Accès / Applicative / Stockage / Données	ssh	22	Accès ssh pour déploiement
Accès / Applicative / Stockage / Données	Administration	http(s)	n/a	Accès aux dépôts rpm
Exploitation technique	Administration	http	5601	Accès utilisateur à Kibana
Exploitation technique	Administration	http	9201	Accès utilisateur à l'administration elasticsearch-log
Exploitation technique	Administration	http	8500	Accès utilisateur à l'administration consul
Exploitation technique	Données	http	9200	Accès utilisateur à l'administration elasticsearch-data
Accès / Applicative / Stockage / Données	Administration	dns/udp	53	Accès aux serveurs DNS externes
Exploitation technique	Accès / Applicative / Stockage / Données	http	divers	Accès aux API de monitoring des composants

Securite

6.1 Principes

Les principes de sécurité de 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.

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 29).

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 : SSLv2, SSLv3
 - Ciphers exclus : .*NULL.* , .*RC4.* , .*MD5.* , .*DES.* , .*DSS.*

Note : Dans cette version du système VITAM, les ciphers recommandés sont : TLS_ECDHE.* , TLS_DHE_RSA.*

- 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 est bien présent dans le référentiel d'authentification des certificats valides (qui est un fichier keystore contenant la liste des certificats valides).

Note : Dans cette version du système VITAM, la liste des certificats reconnus est stockée dans un keystore Java.

- 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](#)⁴² 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`) permettant de valider l'inocuité 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 du système.

Dans chaque requête, les deux headers suivants sont positionnés :

- `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 ;
- Vérification que timestamp envoyé est distant de l'heure actuelle sur le serveur requêté de moins de 10 secondes ($| \text{Timestamp} - \text{temps local} | < 10 \text{ s}$)
- Validation du hash transmis via la réalisation du même calcul sur le serveur cible et de la comparaison des résultats.

En cas d'échec d'une de ces validations, 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 performance.

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.

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

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 ; ceci est dû à une limitation de la version OpenSource d'Elasticsearch, et pourra être réévalué dans les futures versions du système VITAM.

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 mot de passe et token utilisés par ansible doivent être stockés dans des fichiers d’inventaire chiffrés par ansible-vault ;
- Les clé privées des certificats doivent être protégées par des mot 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), ainsi que les CA (finales et intermédiaires) et CRL associées
 - Ces certificats seront déployés dans des [keystores java](#)⁴³ en tant qu’élément de configuration de ces services (Cf. le [DIN](#) pour plus d’information)
- Certificats x509 client pour les clients du SAE (ex : les applications métier, le service ihm-demo), ainsi que les CA (finales et intermédiaires) et CRL associées
 - Ces certificats seront déployés dans des [keystores java](#)⁴⁴ en tant qu’élément de configuration de ces services (Cf. le [DIN](#) pour plus d’information)

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.

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

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

6.3 Certificats

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

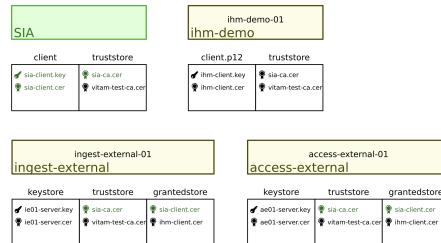


Fig. 6.1 – Vue d'ensemble des magasins de certificats déployés dans un système VITAM

Architecture détaillée

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

7.1 Access

7.1.1 Architecture Fonctionnelle

7.1.1.1 Généralités

Le rôle d'accès est de :

- Rechercher les Unités d'archives via des mots-clés.
- Afficher la liste des résultats par rapport au 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 implémentant les API publiques.
- vitam-access-rest : module pour les api REST.

//TODO si IHM

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

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 contrôleur 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 update-
Query, String unitId); selectObjectById(String selectObjectQuery, String objectId); getObjectAsInput-
Stream(String selectObjectQuery, String objectGroupId, String usage, int version);
```

Paramètre de la fonction : String ds, String Identification //TODO (Itérations futures : ajouter méthode modification des métadonnées ?)

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 publique du module access-api

7.1.2.7.2 Packages :

fr.gouv.vitam.access.core

Classes utilisées

AccessModuleImpl

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

```
public JsonNode selectUnit(String selectRequest) {
    ...
    // Récupération du client metadata
    metaDataClientFactory = new MetaDataClientFactory();
    metaDataClient = metaDataClientFactory.create(accessConfiguration.
        getUrlMetaData());
    ...
    // appel du client metadata
    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 contrôle 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 contrôleur 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/object
<https://vitam/access-external/v1/acquisition-registers>
https://vitam/access-external/v1/acquisition-registers/document_id
https://vitam/access-external/v1/acquisition-registers/document_id/acquisition-register-detail
<https://vitam/access-external/v1/operations>
https://vitam/access-external/v1/operations/operation_id
https://vitam/access-external/v1/unitlifecycles/lifecycle_id
https://vitam/access-external/v1/objectgrouplifecycles/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'arrêter le serveur. Dans le fichier de configuration, le paramètre jettyConfig est à paramétriser avec le nom du fichier de configuration de jetty.

7.1.2.10 -AccessResourceImpl.java

classe contrôleur 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}/object")  
@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}/object")  
@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 contrôleur 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("/operations/{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'opération 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("/unitlifecycles/{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("/objectgrouplifecycles/{id_lc}")
@Produces(MediaType.APPLICATION_JSON)
public Response getObjectGroupLifeCycle(@PathParam("id_lc") String_
    ↴objectGroupLifeCycleId)
...
```

7.1.2.12 -AdminManagementExternalResourceImpl.java

classe contrôleur 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 renseigné et égale à GET alors, c'est un find, donc redirection vers la méthode findDocuments ci-dessus. Sinon, c'est créer. Cette méthode est utilisée pour créer des profils 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 permet 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 permet de récupérer 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) {
    ...
}
```

7.1.3 Sécurité

7.2 Common

7.2.1 Architecture Fonctionnelle

7.2.1.1 Introduction

7.2.1.1.1 But de cette documentation

L'objectif de cette documentation est d'expliquer l'architecture fonctionnelle de ce module.

7.2.1.1.2 GUID

Cf chapitre dédié

7.2.1.1.3 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 (a priori ELK).

L'ensemble des logs seront centralisés mais tous n'iront pas dans la partie “analytique” des logs.

7.2.1.2 Introduction

Le sujet porte notamment sur les **GUID**.

7.2.1.2.1 Présentation de la problématique

7.2.1.2.1.1 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.

7.2.1.2.1.2 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.

7.2.1.2.1.3 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*

7.2.1.2.2 Solutions envisagées

Identifiants au format ARK et au format “Vitam”

7.2.1.2.2.1 ARK

Source : <http://tools.ietf.org/id/draft-kunze-ark-15.txt>

7.2.1.2.2.2 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, ...)

7.2.1.2.2.3 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, ...).

7.2.1.2.2.4 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 unique**
 - 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

7.2.1.2.2.5 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

7.2.1.2.2.6 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)

7.2.1.3 Introduction

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).

7.2.1.3.1 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 créer 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.

7.2.1.4 Introduction

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).

7.2.1.4.1 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 apparaît 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.

7.2.2 Architecture Technique

7.2.2.1 Introduction

7.2.2.1.1 But de cette documentation

L'objectif de cette documentation est d'expliquer l'architecture fonctionnelle de ce module.

7.2.2.1.2 GUID

Cf chapitre dédié

7.2.2.2 Introduction

Le sujet porte notamment sur les **GUID**.

7.2.2.2.1 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.

7.2.2.2.1.1 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 plate-forme 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**)
 - risque de collisions autour de $2^{17} \sim 100K$ GUID générés par millisecondes, donc avec la progression des puissances de calculs sur 20 ans (Loi de Moore approchée : *2 tous les 3 ans) = $2^7+17 = 2^{24}$

Certains bits ne sont pas utiliser (5) pour de futures usages.

7.2.2.3 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

- **Paramètres spécifiques à la couche TLS**

- Paramètres liés aux keystore (emplacement, mot de passe keystore, mot de passe des clés privées)
- Paramètres liés aux trustore (idem keystore)
- Paramètres liés à TLS (protocoles autorisés, ciphers autorisés, options TLS)

7.2.2.4 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

7.2.2.5 Schéma de certificats et d'authentification

7.2.2.5.1 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 privée 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 privée & le certificat (format certificat PEM ou PKCS12 contenant clé privée ou publique) pour l'authentification lors de la requête.

7.2.2.6 Common format identification

7.2.2.6.1 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

7.2.2.6.2 Sous packages

7.2.2.6.2.1 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).

7.2.2.6.2.2 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 requête 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.

7.2.2.6.2.3 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.

7.2.2.6.2.4 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.

7.2.2.7 Messages

La classe **fr.gouv.vitam.common.i18n.Messages** permet de récupérer des messages internationalisés par l'utilisation d'un *ResourceBundle*.

Elle utilise des fichiers de ressources 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.

7.2.2.8 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).

7.2.2.9 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**”.

7.2.2.9.1 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*<

7.2.2.9.2 Sauvegarde dans le thread local

Package : fr.gouv.vitam.common.thread

Le **Request ID** est sauvégarde 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.

7.2.2.9.3 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.

7.2.2.9.4 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>
```

7.2.3 Sécurité

7.2.3.1 Introduction

7.2.3.2 Sécurité de MongoDB

7.2.3.2.1 Objectifs

L'objectif est de sécuriser l'accès à la base de donnée MongoDB. MongoDB exige que tous les clients se s'authentifier afin de déterminer leur accès.

Pour contrôler l'accès 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.

7.2.3.3 secret de la plateforme

7.2.3.3.1 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.

7.3 Functional administration

7.3.1 Architecture Fonctionnelle

7.3.1.1 Introduction

7.3.1.1.1 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 aspects différents de la plate-forme VITAM. Pour l'instant, deux fonctionnalités de gestion prévues sont supportées

- gestion de format
- gestion de règles
- gestion des contrats d'accès
- gestion des contracts d'entrée
- gestion des profiles
- gestion des contextes

7.3.1.2 Introduction

7.3.1.2.1 Gestion de format

7.3.1.3 Introduction

7.3.1.3.1 Gestion de règles

7.3.2 Architecture Technique

7.3.2.1 Introduction

7.3.3 Securite

7.3.3.1 Introduction

7.4 IHM demo

7.4.1 Architecture fonctionnelle

7.4.1.1 Architecture fonctionnelle de l'application Back

7.4.1.1.1 But de cette documentation

On présente dans ce document l'architecture fonctionnelle de l'application Back IHM de VITAM.

7.4.1.1.2 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.

7.4.1.1.2.1 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]
}
```

7.4.1.1.2.2 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 :

```
{"$roots": [{"id": "id"}], "$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)}]
}
```

7.4.1.1.2.3 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": "XXXXXXXXXX", "newValue": "VVVVVVVVVV"}, {"fieldId": "YYYYYYYYYY", "newValue": "VVVVVVVVVV"}, ...]
```

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.

7.4.1.1.2.4 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.

7.4.1.1.2.5 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

7.4.1.2 Architecture fonctionnelle de l'application Front

7.4.1.2.1 But de cette documentation

Cette documentation présente l'architecture fonctionnelle de l'application Front du programme VITAM.

7.4.1.2.2 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

7.4.1.2.3 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

7.4.1.2.4 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'échec de l'appel REST de récupération des détails d'une archive unit

7.4.1.2.5 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**

7.4.1.2.6 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](#)⁴⁵.

- [archive-unit.component.js](#)
- [archive-search.component.js](#)
- [fileformat-component.js](#)
- [logbook-component.js](#)

7.4.2 Architecture technique

7.4.2.1 Architecture technique de l'application Back

7.4.2.1.1 But de cette documentation

Cette documentation décrit l'architecture technique de la partie Back de l'application IHM de VITAM.

7.4.2.1.2 Organisation du module ihm-demo

L'application IHM de VITAM est assurée par le module ihm-demo composé de deux sous-modules :

7.4.2.1.2.1 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.

7.4.2.1.2.2 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
 - staticContent : emplacement des fichiers statiques
 - 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

⁴⁵. <https://docs.angularjs.org/guide/component>

- PUT /ihm-demo/v1/api/archiveupdate/units/{id}
- POST /ihm-demo/v1/api/admin/formats
- POST /ihm-demo/v1/api/admin/format/{idFormat}
- POST /ihm-demo/v1/api/format/check
- POST /ihm-demo/v1/api/format/upload
- DELETE /ihm-demo/v1/api/format/delete

7.4.2.1.2.3 2. Module ihm-core

Ce module gère la couche fonctionnelle de l'IHM ainsi que l'interaction avec les autres modules de VITAM.

7.4.2.1.2.4 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.

7.4.2.2 Architecture technique de l'application Front

7.4.2.2.1 But de cette documentation

Cette documentation présente la structure technique de l'application Front Single Page développée avec AngularJS 1.

7.4.2.2.2 Le Framework Front : AngularJS 1.5.3

7.4.2.2.2.1 Les modules AngularJS utilisés :

- angular-animate
- angular-resource
- angular-route

7.4.2.2.2.2 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 icônes utilisées dans le menu et les boutons

7.4.2.2.3 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
	/modules
	/controller : controllers AngularJS /archive-unit : module qui gère l'écran du formulaire d'une Archive Unit /archive-unit-search : module qui gère 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 gère l'écran d'import du référentiel PRONOUN /logbook : module qui gère 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

7.5 IHM recette

7.5.1 Architecture technique

7.5.1.1 Architecture technique de l'application Back

7.5.1.1.1 But de cette documentation

Cette documentation décrit l'architecture technique de la partie Back de l'application IHM de VITAM.

7.5.1.1.2 Organisation du module ihm-recette

L'application IHM de recette de VITAM est assurée par le module ihm-recette composé de trois sous-modules :

7.5.1.1.2.1 1. Module ihm-demo-web-application

Ce module encapsule à la fois le serveur d'application.

7.5.1.1.2.2 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
 - staticContent : emplacement des fichiers statiques

7.5.1.1.2.3 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

7.5.1.1.2.4 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*.

7.5.1.1.2.5 3. Module ihm-core

Ce module gère la couche fonctionnelle de l'IHM ainsi que l'interaction avec les autres modules de VITAM.

7.5.1.2 Architecture technique de l'application Front

7.5.1.2.1 But de cette documentation

Cette documentation présente la structure technique de l'application Front Single Page développée avec AngularJS 1.

7.5.1.2.2 Le Framework Front : AngularJS 1.5.3

7.5.1.2.2.1 Les modules AngularJS utilisés :

- angular-animate
- angular-resource
- angular-route

7.5.1.2.2.2 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 icônes utilisées dans le menu et les boutons

7.5.1.2.3 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

7.6 Ingest

7.6.1 Architecture Fonctionnelle

7.6.1.1 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 :

- appeler le service journalisation logbook pour créer des log
- Pousser le document le SIP dans le workspace.
- Appeler 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 antinu ou etape par etape.Dans ce cas, on n'aura pas des unités archivistiques 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 exécuter l'étape suivante.
 - En Mode Continu pour exécuter toutes les étapes.
- 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 déterminer l'identifiant du workflow qui sera exécuté par le moteur workflow (processEngine) grâce à l'identifiant du contexte.

A titre d'exemple : le contextid c'est la combinaison mode d'exécution : production ou test à blanc, utilisateur connecté et contrat.

7.6.1.2 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).

7.6.1.3 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.

7.6.1.4 Autres Fonctionnalités :

On peut également :

- **Relancer un processus workflow (production / test blanc) en pause :**
 - En Mode étape par étape pour exécuter l'étape suivante.
 - En Mode continu pour exécuter toutes les étapes.
- Mettre en pause un processus workflow en cours d'exécution.
- Annuler un processus workflow en cours d'exécution ou en pause.

7.6.1.5 Ingest ExternalAntivirus

L'antivirus est intégré dans le processus de upload d'un SIP pour déterter un fichier infecté. L'antivirus rejettera les fichiers vétrolés (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 vétrolés 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 corrigable par l'antivirus). objectIdentifierIncome : <nomDuSIP.extension>
- Si des fichiers vétrolés 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>

7.6.1.6 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.

7.6.1.7 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 espaces 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

7.6.1.8 Les actions :

Les actions possibles 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 être (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.

7.6.1.9 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.

7.6.2 Technique

7.6.2.1 Architecture Technique Ingest

7.6.2.1.1 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 paramètres (logbook & SIP) pour continuer le service. Son deuxième rôle est de télécharger les objets sauvegardés dans le serveur de stockage.

7.6.2.2 ingest-rest

7.6.2.2.1 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.

- 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 paramétrier avec le nom du fichier de configuration de jetty.

7.6.3 Sécurité

7.6.3.1 Introduction

7.7 Logbook

7.7.1 Architecture Fonctionnelle

7.7.1.1 Généralités

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

// TODO compléter la liste

7.7.1.2 Généralités

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

7.7.1.3 Modèle de données

Afin d'assurer le suivi des opérations effectuées sur les archives, un ensemble d'informations sont conservées.

7.7.1.3.1 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	ObligProve-nance	Exem-ple métier	Commentaire
event Identifier	Identifiant de l'opération	Oui	in-terne (vi-tam)	Unique pour chaque ligne
event Type	Type d'opération	Oui	in-terne	CheckSEDADefinition identifiant l'étape/action concernée (format : etape_action)
event DateTime	Date de l'opération	Oui	cal-culé par le journal	
event Identifier Process	Identifiant du processus	Oui	in-terne	GUID
event Type Process	Type de processus	Oui	in-terne	Ingest
outcome	Résultat	Oui	in-terne	Started, OK, Fatal, Warning Il s'agit du status de l'opération. Par exemple lorsqu'une opération est lancée, le status est 'Started'.
outcome Detail	Code correspondant à l'erreur	Non	in-terne	404_XXX Constitué d'un code d'erreur http et d'un sous code d'erreur vitam plus précis.
outcome Detail Message	Informations détaillant la nature de l'erreur ou le message informatif de succès	Oui	in-terne	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.
agent Identifier	Agent réalisant l'action	Oui	cal-culé par le journal	Nom du serveur vitam exécutant l'action : calculé par le journal
agent Identifier Application	Nom de l'application s'authentifiant à Vitam pour lancer l'opération	Non	ex-terne	Identifiant de l'application externe qui appelle Vitam pour effectuer une opération
agent Identifier Application Session	Identifiant donnée par l'application utilisatrice à la session utilisée pour lancer l'opération	Non	ex-terne	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.
event Identifier Request	Identifiant de la requête déclenchant l'opération	Oui	in-terne	X-RequestId généré par Vitam. 1 requestId est créé pour chaque nouvelle requête http venant de l'extérieur
agent Identifier Submis-	Identifiant du service versant	Non	ex-terne	Correspond au SubmissionAgency du SEDA.
agent Identifier Originating	Identifiant du service producteur	Non	ex-terne	Correspond au OriginatingAgency du SEDA.

7.7.1.4 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.

7.7.1.4.1 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	ObligProve-nance	Exem-ple métier	Commentaire
event Identifier	Identifiant de l'opération	Oui	in-terne (vi-tam)	Unique pour chaque ligne
event Type	Type d'opération	Oui	in-terne	CheckSEDADefinition identifiant l'étape/action concernée (format : etape_action)
event DateTime	Date de l'opération	Oui	cal-culé par le journal	
event Identifier Process	Identifiant du processus	Oui	in-terne	GUID
event Type Process	Type de processus	Oui	in-terne	Ingest
outcome	Résultat	Oui	in-terne	Started, OK, Fatal, Warning Il s'agit du status de l'opération. Par exemple lorsqu'une opération est lancée, le status est 'Started'.
outcome Detail	Code correspondant à l'erreur	Non	in-terne	404_XXX Constitué d'un code d'erreur http et d'un sous code d'erreur vitam plus précis.
outcome Detail Message	Informations détaillant la nature de l'erreur ou le message informatif de succès	Oui	in-terne	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.
agent Identifier	Agent réalisant l'action	Oui	cal-culé par le journal	Nom du serveur vitam exécutant l'action : calculé par le journal
agent Identifier Application	Nom de l'application s'authentifiant à Vitam pour lancer l'opération	Non	ex-terne	Identifiant de l'application externe qui appelle Vitam pour effectuer une opération
agent Identifier Application Session	Identifiant donnée par l'application utilisatrice à la session utilisée pour lancer l'opération	Non	ex-terne	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.
event Identifier Request	Identifiant de la requête déclenchant l'opération	Oui	in-terne	X-RequestId généré par Vitam. 1 requestId est créé pour chaque nouvelle requête http venant de l'extérieur
agent Identifier Submis-	Identifiant du service versant	Non	ex-terne	Correspond au SubmissionAgency du SEDA.
agent Identifier Originating	Identifiant du service producteur	Non	ex-terne	Correspond au OriginatingAgency du SEDA.

7.7.2 Architecture technique

7.7.2.1 Introduction

7.7.2.1.1 Présentation

Parent package : fr.gouv.vitam

Package proposition : fr.gouv.vitam.logbook

7.7.2.1.2 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

7.7.2.1.2.1 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)

7.7.2.1.3 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

7.7.2.2 DSL

7.7.2.2.1 Analyse

7.7.2.2.1.1 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 dans une URL)**

- 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

- **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 ;

7.7.2.2.1.2 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 requêtes 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;
}
```

7.7.2.2.1.3 Utilisation

Classe Query pour y intégrer une expression

```
Query{  
    Expression exp;  
}
```

Classe SearchQuery pour y intégrer une liste de Query

```
SearchQuery{  
    List<Query> queries;  
}
```

7.7.2.2.2 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.

7.7.2.3 Rest

7.7.2.3.1 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.

7.7.2.3.2 Services

7.7.2.4 Common-client

7.7.2.4.1 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

7.7.2.4.2 Services

7.7.2.5 Common-client

7.7.2.5.1 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

7.7.2.5.2 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 acquitté par le Moteur de journalisation)
- LifeCycles
 - Un fichier unique, les events dans l'ordre chronologique (qui correspond à l'ordre de events)

7.7.2.5.3 Données

Les données sont stockées dans 3 types de stockage :

- une base maître (MongoDB) qui contient toutes les données de type journal
- une base index (ElasticSearch) qui contient uniquement les journaux de type operation
- les offres de stockage qui contiennent 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 / suppression 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**.

7.7.2.6 Commons

7.7.2.6.1 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...

7.7.2.6.2 Services

7.7.2.7 Operation Client

7.7.2.7.1 Présentation

Parent package : **fr.gouv.vitam.logbook**

Package proposition : **fr.gouv.vitam.logbook.operations.client**

Module pour le client des logs opération.

7.7.2.7.2 Services

7.7.2.8 Opération

7.7.2.8.1 Présentation

Parent package : **fr.gouv.vitam.logbook**

Package proposition : **fr.gouv.vitam.logbook.operations**

Module pour le module opération : api / rest.

7.7.2.8.2 Services

7.7.2.8.3 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 événements d'une opération

GET /status -> statut du logbook

7.7.2.9 Lifecycle Client

7.7.2.9.1 Présentation

Parent package : fr.gouv.vitam.logbook

Package proposition : fr.gouv.vitam.logbook.lifecycle

Module client pour les logs lifecycle.

7.7.2.9.2 Services

7.7.2.10 Lifecycle

7.7.2.10.1 Présentation

Parent package : fr.gouv.vitam.logbook

Package proposition : fr.gouv.vitam.logbook.lifecycle.client

Module pour les logs lifecycle : api / rest.

7.7.2.10.2 Services

7.7.2.10.3 Rest API

`http://server/app/v1`

POST /operations/{id_op}/lifecycles/ -> POST un lifecycle sur une opération

PUT /operations/{id_op}/lifecycles/{id_li} -> Append sur un lifecycle existant

GET /lifecycle -> Administration du lifecycle

7.7.2.11 Administration-client

7.7.2.11.1 Présentation

Package Parent : fr.gouv.vitam.logbook

Proposition de package : fr.gouv.vitam.logbook.administration.client

7.7.2.11.2 Services

7.7.2.12 Administration

7.7.2.12.1 Présentation

Package parent : fr.gouv.vitam.logbook

Proposition de Package : fr.gouv.vitam.logbook.administration

7.7.2.12.2 Services

7.7.2.12.3 Rest API

`http://server/app/v1`

Administration

GET /status -> statut du logbook

7.7.3 Sécurité

7.7.3.1 Introduction

7.8 Metadata

7.8.1 Architecture Fonctionnelle

7.8.1.1 Introduction

7.8.2 Architecture technique

7.8.2.1 Introduction

7.8.2.1.1 Présentation

Parent package : fr.gouv.vitam

Package proposition : fr.gouv.vitam.metadata

7.8.2.1.2 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 créer les objets des requêtes 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 requêtes Jsons.
- vitam-metadata-rest :

7.8.2.1.3 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
```

7.8.2.2 Metadata-api

7.8.2.2.1 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 package fr.gouv.vitam.api.exception gère les exceptions issues des opérations des demandes d'accès à travers de métadata.

les exceptions gérées sont :

```
MetaDataAlreadyExistException(String message)
MetaDataAlreadyExistException(Throwable cause)
MetaDataAlreadyExistException(String message, Throwable cause)
```

- Le package fr.gouv.vitam.api.model permet de la gestion d'interrogation de la base de données.

7.8.2.3 Metadata-builder

7.8.2.3.1 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 requête.

- 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 configurer d'une requête dynamique de metadonnees.
- Le package fr.gouv.vitam.builder.request.construct.query permet de regrouper d'un ensemble de requête dynamiquement.
- Le package fr.gouv.vitam.builder.request.exception permet de gérer, détecter les exceptions lors l'utilisation d'une requête dynamique.

7.8.2.4 Operation Client

7.8.2.4.1 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 requête client.

7.8.2.5 metadata-core

7.8.2.5.1 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)

7.8.2.5.2 1. Modules et packages

```
—fr.gouv.vitam.metadata.core.collections : contenant des classes pour gérer les requêtes MongoDb
—fr.gouv.vitam.metadata.core.utils
—fr.gouv.vitam.metadata.core
—fr.gouv.vitam.metadata.core.database.configuration
```

7.8.2.5.3 2. Classes

Dans cette section, nous présentons quelques classes principales dans les modules/packages abordés ci-dessus.

7.8.2.5.3.1 2.1 Class DbRequest

La classe qui permet de gérer les requêtes 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 requête . 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 requête avec la dernière liste 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 indexées 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 indexées 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 requête 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 requête et supprimer d'index en se basant sur la requête.
- 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.

7.8.2.5.3.2 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.

7.8.2.5.3.3 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 unitOrObject-
GroupId, 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.

7.8.2.5.3.4 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.

7.8.2.5.3.5 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).

7.8.2.5.3.6 2.5 Class UnitInheritedRule

- La Méthode `createNewInheritedRule(ObjectNode unitManagement, String unitId)`
permet de calculer les règles de gestion héritées en utilisant la 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.

7.8.2.6 metadata-parser

7.8.2.6.1 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.

7.8.2.7 Métadata

7.8.2.7.1 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.

7.8.2.7.2 Services

7.8.2.7.3 Rest API

URL Path : <http://server/metadata/v1>

POST /units -> ****POST nouvelle unit et sélection d'une liste des units avec une requête ****

GET /units -> ****GET sélectionne une liste des units avec une requête ****

GET /status -> **statut du server rest metadata (available/unavailable)**

POST /objectgroups -> **POST : 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

7.8.2.8 Rest

7.8.2.8.1 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.

7.8.2.8.2 Services

7.8.3 Securite

7.8.3.1 Introduction

7.9 Processing

7.9.1 architecture-fonctionnelle-processing

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 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 chaîne 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

7.9.1.2 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 ProcessWorkflow. Pour une instance d'une machine à état, une et une seul instance de ProcessEngine est créée. 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

7.9.1.3 Introduction

7.9.1.3.1 But de cette documentation

L'objectif de cette documentation est d'expliquer l'architecture fonctionnelle de ce module.

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.

7.9.1.4 Introduction

7.9.1.4.1 But de cette documentation

L'objectif de cette documentation est d'expliquer l'architecture fonctionnelle de ce module.

Le but de ce module est d'attribuer des tâches 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.

7.9.1.5 Introduction

7.9.1.5.1 But de cette documentation

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.

7.9.1.6 Process Monitoring

7.9.1.6.1 Explication

L'objectif de cette documentation est d'expliquer l'architecture fonctionnelle de ce module.

Le but de ce module est de pouvoir monitorer les différentes étapes des différents Workflow.

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 informations 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}

7.9.2 Architecture Technique

7.9.2.1 Introduction

7.9.2.2 DAT : module processing

Ce document présente l'ensemble de manuel développement concernant le développement du module metadata qui représente le story #70, qui contient :

- modules & parkages
- classes de métiers

7.9.2.2.1 1. Module et packages

Les principaux modules sont :

|— processing-common : contient les méthodes communs : 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

7.9.2.2.2 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 responsable 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 une 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.

7.9.2.2.3 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 famille de worker
- liste des workers d'une famille
- effacer les workers d'une famille
- statut d'un worker
- mise à jour d'un worker

7.9.2.2.4 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 WorkerConfiguration, 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.
 - > Opérations :
 - Lors de l'enregistrement d'un worker (voir section ci-dessus), un thread (cf WorkerManager) est créé 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 élément de découplage et a donc une taille 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

7.9.2.3 Rangement des objets

7.9.2.3.1 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

7.9.2.4 Vérification de la disponibilité

7.9.2.4.1 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
 - (a) Appel au moteur de stockage pour récupérer un Json contenant les informations de capacité pour un couple tenant/stratégie de stockage donné.
 - (b) Comparaison entre capacité renournée par le moteur de stockage et taille totale calculé précédemment
 - i. Si capacité supérieure Alors Inscription dans logbook operation d'un OK
 - ii. Si capacité inférieure Alors Inscription dans logbook operation d'un KO, fin du process : "Disponibilité de l'offre de stockage insuffisante"
 - iii. 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"

7.9.2.5 Vérifier SEDA

7.9.2.5.1 Algorithme

1. Vérifier la validation du seda (SedaUtils->checkSedaValidation)
 - (a) Vérifier l'existance de manifest.xml (SedaUtils->checkExistenceManifest)
 - (b) 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
 - (a) Créer la liste de version de manifest.xml (SedaUtils->manifestVersionList)
 - (b) 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".
 - (c) Journalisation de l'action CheckVersion
 5. Vérifier les empreintes de BinaryDataObject
 - (a) Récupération d'empreinte du GUID/objects/SIP/content/<uri_correspondent> (WorkspaceClient->computeObjectDigest)
 - (b) Créer la liste d'empreinte de manifest.xml
 - (c) 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".
 - (d) Journalisation de l'action CheckConformity

7.9.3 Sécurité

7.9.3.1 Introduction

7.10 Storage

7.10.1 Architecture Fonctionnelle

7.10.1.1 Introduction

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.storage

7.10.2.1.2 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 embrasant les implémentations spécifiques de l'offre de stockage, actuellement que l'implémntation swift.

7.10.2.1.3 Modules - packages Storage

```
storage
  /storage-driver-api
    fr.gouv.vitam.storage.driver
```

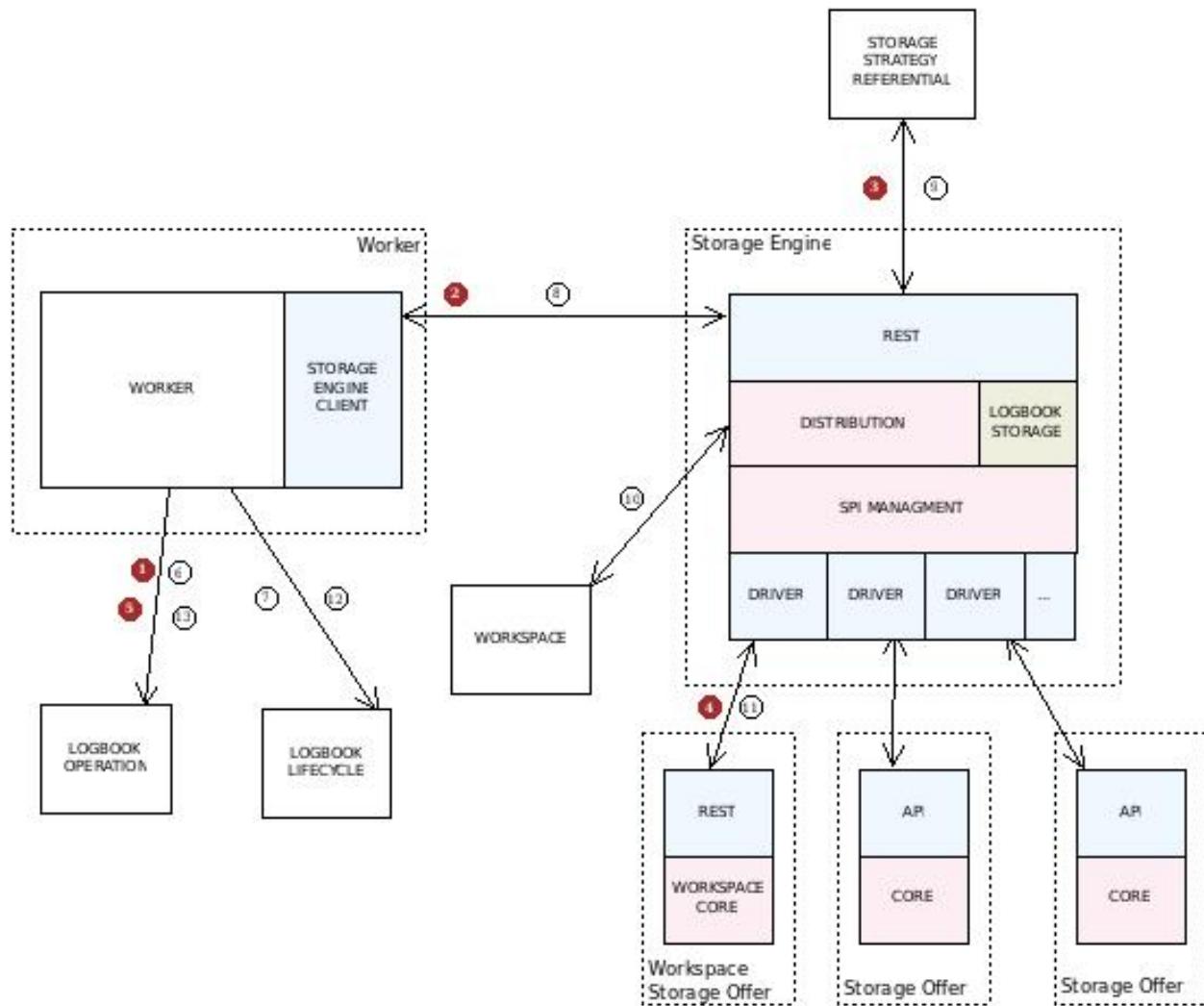
```
/storage-engine
  /storage-engine-client
    fr.gouv.vitam.storage.engine.client
  /storage-engine-server
    fr.gouv.vitam.storage.engine.server.spi
    fr.gouv.vitam.storage.engine.server.logbook
    fr.gouv.vitam.storage.engine.server.rest
    fr.gouv.vitam.storage.engine.server.distribution
  /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
```

7.10.2.2 Architecture générale

7.10.2.2.1 Schéma général



7.10.2.2.2 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 stockage*
 - étape 3 : le **moteur de stockage** appelle le **référentiel des stratégies de stockage** pour récupérer le détail de la *stratégie 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/...)

7.10.2.2.3 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.

7.10.2.2.4 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

7.10.2.2.5 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.

7.10.2.2.6 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 indéfiniment s'il n'y a pas de timeout.

Il n'y a pas de vrai pool de threads dans cette version.

7.10.2.2.7 Itération 16

- Revue de l'architecture globale 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

7.10.2.3 Storage Driver

7.10.2.3.1 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.

7.10.2.3.2 Architecture

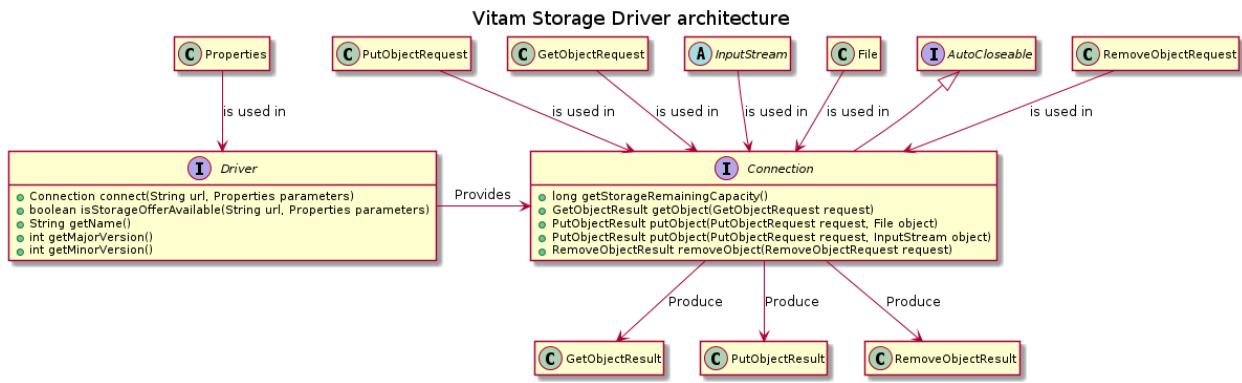
Pour permettre au moteur d'exécution de dialoguer avec un service d'offre de stockage, deux interfaces doivent être implémentées par le fournisseur d'offre :

- **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 :



7.10.2.3.3 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 concurrents utilisant des drivers.
- **Connection pool** : Bien que proche du premier point, celui-ci est à mettre en place au niveau Driver directement. 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.

7.10.2.4 Storage Engine

7.10.2.4.1 Présentation

Parent package : fr.gouv.vitam.storage

Package proposition : fr.gouv.vitam.storage.engine

Module embarquant la partie core du storage (client et server).

7.10.2.4.1.1 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.

7.10.2.4.1.2 Rest API

7.10.2.4.1.3 URI d'appel

http://server/storage/v1

7.10.2.4.1.4 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

7.10.2.4.1.5 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**

7.10.2.4.1.6 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 démultiplie 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 :
 - (a) Récupération du Driver associé s'il existe (sinon remontée d'une exception technique)

- (b) Instancie un thread et dans ce trhead : 1. Récupération des paramètres de l'offre : url du service, paramètres additionnels 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)
 - (c) 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

7.10.2.4.1.7 DriverManager : SPI

Service permettant d'ajouter ou de supprimer des drivers d'offre.

Le driver (son interface) est défini dans storage-driver.

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'identifiant 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.

7.10.2.4.1.8 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.

7.10.2.4.1.9 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.

7.10.2.5 Storage Engine Client

7.10.2.5.1 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.

7.10.2.6 Storage Offers

7.10.2.6.1 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. Cependant, elle permet d'être de deux types différents : système de fichier ou swift - un seul driver (utilisée par storage-offer-default) appelé vitam-driver. Il permet d'utiliser l'offre par défaut qu'elle soit en mode swift ou en mode system de fichier - Il est possible, grâce à plusieurs instance de l'offre par défaut d'avoir un stockage multi offres. Il existe une limite, au sein de la même JVM, il n'est possible de n'avoir qu'une seul offre d'un seul type.

7.10.2.7 Vitam Offer

7.10.2.7.1 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 workspace 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 défini par son identifiant ET son conteneur.

7.10.2.7.2 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.

7.10.2.7.3 Serveur

7.10.2.7.3.1 Description

Les fonctionnalités sont :

- récupérer la capacité et disponibilité de l'offre
- envoyer un objet en mode chunk
- 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

7.10.2.7.3.2 REST

7.10.2.7.3.3 Description

L'API REST, trois header spécifiques sont définis :

- X-Tenant-Id : l'identifiant du Tenant
- X-Command : utilisée pour l'envoi d'un objet par fragments (chunk)
 - INIT : création de l'objet donc l'offre de stockage, sans données envoyées
 - WRITE : envoi d'un fragment de data
 - END : indique que l'objet est fini d'être créé
- X-Type : permet de préciser le résultat attendu pour la récupération 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 respectées (400, 401, 404, etc)

7.10.2.7.3.4 REST API

HEAD /

- description : récupération des informations de l'offre
- response :
 - code : 200
 - contenu : information sur l'offre (capacité, disponibilité, ...)

GET /{type}/count

- 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 : récupération 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

POST /objects

- description : création d'un nouvel objet vide sur l'offre
- headers :
 - X-Command : INIT
 - X-Tenant-Id : id du tenant
- body :
 - GUID
 - ObjectInit contenant la taille (taille finale), le type (unit/objectgroup/logbook/etc, se basant sur une enum), le digest-type (type de digest) ainsi qu'un identifiant vide à l'envoi qui sera rempli pour l'offre. Il s'agit de l'identifiant de l'objet sur l'offre. Dans l'implémentation par défaut, c'est le GUID.
- response :
 - code : 201
 - contenu : l'objectInit envoyé avec l'identifiant de l'objet créé

PUT /objects/{id}

- description : écriture et finalisation d'objet de l'offre
- headers :
 - X-Command : WRITE / END
 - X-Tenant-Id : id du tenant
- path :
 - {id} : id de l'objet
- body :
 - flux : data ou digest
- response :
 - code : 201
 - contenu : un json avec une clef unique, digest, le digest du fichier complet sur l'offre pour le END, le digest du morceau envoyé pour le WRITE

HEAD /objects/{id}

- description : existence 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

7.10.3 Securite

7.10.3.1 Introduction

7.11 Technical administration

7.11.1 Architecture Fonctionnelle

7.11.1.1 Introduction

7.11.2 Architecture Technique

7.11.2.1 Introduction

7.11.3 Securite

7.11.3.1 Introduction

7.12 Workspace

7.12.1 Architecture Fonctionnelle

7.12.1.1 Introduction

7.12.2 Architecture Technique

7.12.2.1 Introduction

7.12.3 Securite

7.12.3.1 Introduction

Annexes

Table des figures

3.1	Vue de VITAM dans son environnement (vue “boîte noire”)	6
3.2	Architecture fonctionnelle cible de VITAM	12
4.1	Architecture applicative et flux d’informations entre composants. Tous les composants jaunes sont fournis dans le cadre de la solution VITAM ; tous sont requis pour le bon fonctionnement de la solution, à l’exception de deux d’entre eux : ihm-demo et storage-offer-default (selon les choix de déploiement)	14
5.1	Environnement d’un service VITAM	22
5.2	Déploiement VITAM : zones & principes de communication ; les utilisateurs métier archivistes sont présentés à gauche, et les exploitants technique à droite.	30
5.3	Vue d’ensemble du processus d’export MongoDB pour base répartie	37
5.4	Architecture technique : légende	41
5.5	Architecture technique : flux (1/5 : flux métiers généraux)	42
5.6	Architecture technique : flux (2/5 : flux métiers de dépôt des journaux)	43
5.7	Architecture technique : flux (3/5 : flux métiers de lecture des référentiels métier)	44
5.8	Architecture technique : flux (4/5 : flux des outils d’exploitation)	45
5.9	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.	46
5.10	Architecture générique d’un système de gestion de logs.	47
5.11	Architecture du sous-système de centralisation des logs	48
5.12	Déploiement d’un cluster Mongo DB avec sharding	65
6.1	Vue d’ensemble des magasins de certificats déployés dans un système VITAM	74

Liste des tableaux

2.1 Documents de référence VITAM	3
5.1 Matrice des flux inter-zones	69

A

API, [3](#)

B

BDD, [3](#)

C

COTS, [3](#)

D

DAT, [3](#)

DEX, [3](#)

DIN, [3](#)

DNSSEC, [4](#)

DUA, [4](#)

I

IHM, [4](#)

J

JRE, [4](#)

JVM, [4](#)

M

MitM, [4](#)

N

NoSQL, [4](#)

O

OAIS, [4](#)

P

PDMA, [4](#)

PKI, [4](#)

R

REST, [4](#)

RPM, [4](#)

S

SAE, [4](#)

SEDA, [4](#)

SIA, [4](#)

T

TNR, [4](#)

V

VITAM, [4](#)