

Guide d'Architecture Technique

Version du	26/03/2020
Auteurs	DSI pôle SIM unité Architecture
Mot clés	Architecture Socle Technique API SOA
Référence	DSI/SIM/ARCHI/SOCLE/guide_archi_technique

Historique des versions :

Version	Date	Auteur(s)	Modifications
1.0	10/11/2018	H.Chéhadé, A.Jolly,	1 ^{ère} version réalisée avec CapFi et le groupe de travail SIM-Nouveau socle applicatif
2.0	26/03/2020	Thomas Broussard	Mise à jour du document
2.1	27/04/2020	Thomas Broussard	Mise à jour du document
2.2	28/04/2021	Lionel Corlobé	Mise à jour du document, ajout Birt
2.3	18/05/2021	Lionel Corlobé	Mise à jour du document § Birt
2.4	10/01/2022	André-Pierre Abellard	Mise à jour du document, ajout de WSO2 et Keycloak-js (mode SSO)
2.4.1	19/09/2022	Boi Chanh HUYNH	Ajout de l'Annexe : Utilisation particulière d'une API

Sommaire

1.	Préambule	4
1.1.	Objectif du document	4
1.2.	Lexique	4
2.	Présentation Générale	6
2.1.	Contexte et enjeux	6
2.2.	Synthèse du choix de l'architecture	7
2.2.1.	L'architecture comme outil au service du métier	7
2.2.2.	Les critères retenus	7
2.3.	Exigences et Contraintes	10
2.3.1.	Temps de réponse	10
2.3.2.	Niveau de robustesse	10
2.3.3.	Disponibilité	10
2.3.4.	Volumétrie	11
2.3.5.	Sécurité	11
3.	Architecture Applicative	12
3.1.	Présentation générale de l'architecture	12
3.2.	Focus sur l'architecture orientée API	12
3.2.1.	Définition d'une API	12
3.2.2.	Impacts de ce changement de modèle	13
4.	Architecture orientée API et API Management	14
4.1.	Gestion des développements	15
4.2.	Gestion des déploiements	17
4.3.	Conception des APIs	18
4.4.	Sécurisation des échanges	18
4.5.	Passage en Orienté API : Impact organisationnel	19
5.	Architecture orientée API : Trajectoire d'implémentation et bonnes pratiques	20
5.1.	Abstraction du modèle historique	20
5.2.	Exploitation des APIs dans l'application existante	21
5.3.	Refonte de l'application cliente	21
5.4.	Raffinement des services	21
6.	Les règles d'implémentation des API Rest	22
6.1.	Maturité de l'approche orientée API basée sur REST	22
6.1.1.	Niveau 0 : Marais de POX (Plain Old XML)	22
6.1.2.	Niveau 1 : Ressources	22
6.1.3.	Niveau 2 : verbes HTTP	22
6.1.4.	Niveau 3 : contrôles hypermédia	23
6.2.	KISS	23
6.3.	Modèle métier et granularité des ressources	24
6.4.	Veiller à l'aspect Stateless de l'API	26
6.5.	Aspect non transactionnel	26
6.6.	Utilisation de la grammaire REST	27
6.6.1.	Verbes de la grammaire REST	27
6.6.2.	Documentation des ressources et de leur utilisation	28
6.6.3.	Filtre et Pagination	29
7.	Architecture logicielle des composants	30
7.1.	Back-end	30
7.2.	Frontend	31
7.3.	Batches	31
7.4.	Serveur d'édition Birt	33
8.	Mutualisation des composants	35
8.1.	Editique	35

8.2.	Upload (Stockage des Documents).....	35
8.3.	Utilisation d'un ESB	35
9.	Sécurité des applications.....	36
9.1.	OWASP	36
9.2.	OAuth2	36
9.3.	Implémentation d'un Single Sign On (SSO) respectant le protocole OAuth2.....	38
9.4.	Sécurisation des Appels aux services REST par WSO2 (API Manager)	40
9.5.	WAF.....	43
9.6.	Intégration avec le portail SIPG.....	43
9.6.1.	Utilisation d'OPENID Connect (OIDC)	43
9.6.2.	Fourniture d'une application Blanche.....	44
10.	Infrastructure de production	46
11.	Annexe : Utilisation particulière d'une API.....	48

1. Préambule

1.1. Objectif du document

Dans le cadre de l'évolution du socle applicatif de ses futures applications, l'agence de la Biomédecine a préalablement choisi de renouveler sa solution d'architecture et son socle technologique.

L'objectif de ce document est de fournir un repère qui regroupe un ensemble d'informations concernant les concepts-clés de l'architecture applicative API, les aspects de l'architecture technique, afin d'accompagner l'agence de la biomédecine et ses prestataires à structurer ses nouvelles applications.

Plus précisément, ce document didactique présente et définit les relations à l'environnement architectural, démystifie les postulats de conception, et fournit un modèle de dossier technique d'architecture pour accompagner à la création de nouvelles applications logicielles.

Pour chaque projet, il sera nécessaire de faire un DAG (Dossier d'Architecture Générale) se basant sur les recommandations de ce guide d'architecture.

1.2. Lexique

Dans ce document, les abréviations suivantes seront utilisées :

Abréviation	Signification
API	C'est une interface de programmation applicative (Application Programming Interface). La notion d'API peut avoir plusieurs matérialisations en fonction du contexte : - Le contexte « librairie applicative » lors du développement : il s'agit d'un ensemble de services proposées sous la forme de fonctions ou méthodes permettant de résoudre un problème particulier. L'API est en général exposée sous la forme de spécification (interface de programmation), et l'implémentation concrète reste cachée au code appelant. - Le contexte plus général de l'Architecture Orientée Service (SOA), et plus précisément l'Architecture Orientée API. La philosophie globale reste la même : exposer un service. Cette fois le service est un service distant, qui peut être écrit dans un autre langage, et donc normalisé non pas par une interface de programmation, mais une interface de communication : adresse, format d'appel, de message et de réponse à envoyer/recevoir. La plupart du temps, à la date de mise à jour de ce guide (en 2020), la technologie de communication utilisée est REST. Cela permet de bénéficier de plusieurs avantages décrits plus tard dans ce guide.
HATEOAS	Hypermédia en tant que moteur de l'état d'application (Hypermedia as the Engine of Application State. Cela veut dire que l'état de l'application est géré par le client, le serveur étant dépourvu d'état lui-même. Les actions possibles dans le cadre de l'HATEOAS dépendent de l'état des ressources accédées par les API.
REST	RE presentational S tate T ransfer (REST) est un style architectural qui définit un ensemble de contraintes et de propriétés basées sur HTTP. Il s'agit donc dans cette architecture de reposer sur l'adressage et les verbes

	prévus par le protocole http pour localiser des ressources, sur lesquels on va pouvoir agir grâce à ces verbes (POST,PUT,DELETE,GET,PATCH). C'est un protocole orienté « ressource », qui s'appuie sur les modifications d'état de ces ressources pour pouvoir réaliser le fonctionnel de l'application.
SOAP	SOAP est un protocole de RPC orienté objet bâti sur XML. Il permet la transmission de messages entre objets distants, ce qui veut dire qu'il autorise un objet à invoquer des méthodes d'objets physiquement situés sur un autre serveur. C'est un protocole orienté « opération » qui s'appuie sur une liste d'opérations proposées par un service. La force de SOAP est le niveau de maturité et d'industrialisation qu'il propose, sa faiblesse réside dans la lourdeur inhérente à son implémentation (formalisme xml, non aisé dans certaines technologies tels que le javascript), ainsi que dans la dispersion fonctionnelle des opérations pour lesquelles il est parfois difficile de garder une cohérence globale. L'organisation par « ressources fonctionnelles » de REST aide théoriquement à se prémunir de cette dispersion.
Commit à deux phases	Lorsqu'une transaction concerne la mise à jour de données sur plusieurs serveurs indépendants, et comme chaque base de données possède son propre journal de transactions, il est nécessaire d'ajouter une couche de coordination des transactions distribuées. Le principe du COMMIT à deux phases passe donc par un superviseur (ou coordinateur de transactions distribuées) qui scrute l'état des serveurs et ordonne la finalité globale de la transaction.

2. Présentation Générale

2.1. Contexte et enjeux

Le schéma directeur 2012-2017 (SDSI) a acté que le socle applicatif devait être renouvelé. Le parc applicatif composé d'une trentaine d'applications Java/J2EE a un fort couplage (voir schéma ci-dessous & concerne aussi les bases de données avec duplications d'informations) et entraîne une faible indépendance fonctionnelle. Le SDSI a aussi acté des travaux d'urbanisation pour rendre les applications moins dépendantes les unes des autres. Actuellement, les composantes logicielles sont difficilement réutilisables et testables.

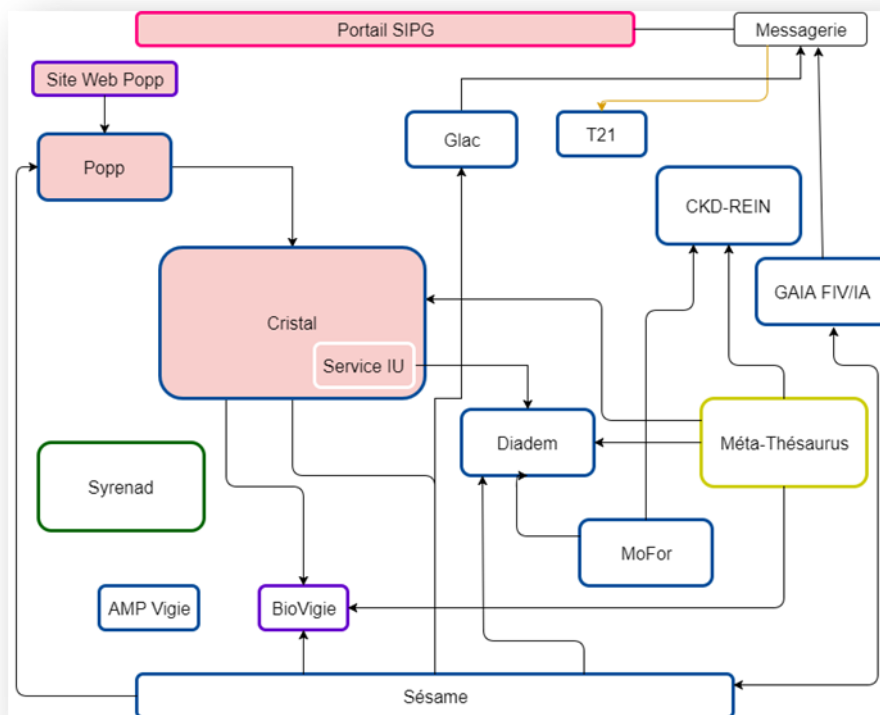


Figure 1: Schéma de la représentation des applications du système d'information de l'agence Biomédecine

Ainsi, l'ajout de nouvelles fonctionnalités sur les applications existantes entraîne un coût supplémentaire.

L'objectif principal poursuivi est de réduire les coûts par la rationalisation des moyens SI ; et l'uniformisation des processus. Le système d'information est souvent composé de développements spécifiques. Pour ces développements, de nombreuses entreprises se tournent vers des briques Open Source, disponibles par milliers sur Internet, répondant quasiment à toutes les problématiques techniques rencontrées et évitant ainsi de « réinventer la roue ».

Chaque DSI peut ainsi choisir les technologies les plus proches de ses besoins en cohérence avec sa stratégie. La difficulté réside alors dans une intégration efficace de ces technologies sous la forme d'un socle applicatif, afin d'industrialiser et d'homogénéiser la réalisation et la maintenance de ces applications.

En conséquence, afin d'optimiser et de réduire les coûts, un socle applicatif technique adapté à nos applications a été choisi.

2.2. Synthèse du choix de l'architecture

2.2.1. L'architecture comme outil au service du métier

Il est courant que l'architecture soit vécue par certains chefs de projets métiers comme une contrainte à respecter, pour un gain inexistant (tout du moins invisible pour eux). Afin que des changements d'architectures soient correctement adoptés par ces chefs de projets, il convient de respecter 2 prérequis :

- La compréhension de l'objectif et des impacts de ces changements d'architectures
- Le gain obtenu par l'application de ces changements.

Le gain peut se matérialiser suivant plusieurs axes :

- Amélioration de la productivité pour l'implémentation,
- Amélioration du niveau de fonctionnalités offert par le nouveau socle,
- Amélioration de la confiance dans l'application,
- Amélioration organisationnelle des projets ou des utilisateurs (création de synergies inter-projets, amélioration de l'organisation des utilisateurs, etc.),
- Amélioration des performances de l'application (qui peut se traduire par une augmentation de la productivité des utilisateurs).

Il faut donc absolument se poser la question si un quelconque changement d'architecture bénéficie au métier selon l'un de ces 5 axes avant de chercher à le réaliser. Dans la négative, il s'agit certainement d'un choix dogmatique qu'il conviendra d'éviter, car ce changement ne sera probablement pas appliqué correctement.

2.2.2. Les critères retenus

Le socle applicatif du système d'information a été déterminé en fonction de la liste des objectifs ci-dessous :

Critère retenu	Gains réalisés	Commentaire
Se Conformer à la tendance générale du marché	Productivité, fonctionnalités	Rester dans la tendance du marché permet d'être en phase avec les compétences des développeurs disponibles, ainsi que de bénéficier de certaines fonctionnalités non présentes dans les anciens systèmes
Notoriété de la solution chez les prestataires	Productivité, Fonctionnalités	Les prestataires auront plus de facilité à faire et cela permettra d'avoir une meilleure force de proposition pour les nouvelles fonctionnalités rendues possible par l'adoption de la nouvelle architecture
Amener les bonnes pratiques de l'ANSSI, le principe de Security By Design et les conseils de l'OWASP	Confiance	La sécurisation au centre du développement permet aux utilisateurs d'avoir plus confiance dans l'utilisation de ces applications.

Sûreté de la solution sur Mobile,	Confiance, Fonctionnalités, Organisation	Il ne serait pas possible d'ouvrir une solution mobile sans sécurité. Ainsi le fait de sécuriser cette partie permet aux utilisateurs d'accéder à cette fonctionnalité et d'améliorer leur organisation.
Fiabilité de la solution (données de santé sensibles)	Confiance	Garantir la fiabilité de la solution (non-divulgateur et absence de pertes de données) permet d'augmenter la confiance des utilisateurs dans la solution.
Rapidité du développement de la solution	Productivité, Confiance	La solution met moins de temps à converger vers une solution stable. Les utilisateurs qui voient l'avancement plus rapide restent mobilisés pour soutenir les tests.
Rentabilité en terme de MCO et de Run,	Productivité	A fonctionnalité équivalente, on produit moins cher, ou à budget constant on produit plus.
Facilité de maintenance et de mise à jour des applications,	Productivité, Confiance	Les évolutions/correctifs sont plus rapides, la confiance des utilisateurs dans le SI augmente.
Adaptée aux méthodes Agiles,	Organisation, Productivité	Permettra plus d'implication de la part des utilisateurs, une convergence plus rapide vers la solution définitive. La solution sera probablement plus adaptée à l'organisation réelle due au métier des utilisateurs.
Compatibilité avec les futurs usages mobiles,	Fonctionnalités, Productivité	Permet de prévoir un déploiement facilité des solutions mobiles lorsque celles-ci seront envisagées.
Uniformiser les échanges inter applicatifs	Productivité, Sécurité, Confiance	Permettra une meilleure réutilisabilité des échanges inter-applicatifs, par exemple dans le cas de la création d'un nouvel échange, qui pourrait réutiliser un échange existant. Le fait de normaliser les échanges permet d'augmenter le niveau de sécurité de ces

		échanges en imposant des règles de communication. La réutilisabilité réduit le délai de mise à disposition d'un service et le niveau de confiance des utilisateurs augmente.
Efficacité de la solution,	Confiance, Performance	L'ergonomie et la fluidité de l'application (en termes de temps de réponse) permettent d'augmenter la productivité des utilisateurs ainsi que la confiance de ces derniers en la capacité du SI à produire un service pertinent.
Capacité à supporter de gros volumes de données,	Performance	La capacité du système à supporter de grands volumes de données, notamment l'historique et le croisement de plusieurs types de données, permettent d'augmenter la productivité des utilisateurs.
Supporter l'intranet et l'Internet (connexion),	Organisation	Permet de satisfaire aux contraintes de l'agence : des utilisateurs en intra et certains services consommateurs en internet

Dans ce contexte, le socle applicatif retenu est basé sur une architecture d'APIs REST. L'intérêt majeur de l'architecture orientée API est sa capacité d'abstraire les technologies d'implémentation des services existants.

Il est ainsi possible, en exposant une API « par-dessus » un service existant de pouvoir exploiter le métier sans avoir besoin de le ré-écrire ou de le modifier en profondeur. Cette approche permet également de naturellement découpler l'interface graphique (front-end) et les services métiers (back-end), permettant à chaque expertise de se concentrer sur son domaine de compétence.

L'interface utilisateur et la logique métier peuvent ainsi être développées par des tiers, probablement plus experts dans les technologies d'interface et les problématiques ergonomiques. Le système expose donc une API, c'est-à-dire un ensemble de ressources. Les applications finales reposent sur la composition de plusieurs ressources provenant éventuellement de systèmes disjoints.

2.3. Exigences et Contraintes

Chaque nouvelle application utilisant le nouveau socle applicatif doit suivre une liste d'exigences et de contraintes :

2.3.1. Temps de réponse

Ci-dessous les trois catégories de temps de réponse pour une application et par conséquent, l'effet généré pour un utilisateur :

- Un temps inférieur à 200ms permet de donner une impression d'immédiateté.
- Un temps inférieur à 2 secondes correspond à la limite pour que l'utilisateur ait l'impression que le système réagit de manière correcte, ce qui signifie qu'aucune rétroaction spéciale n'est nécessaire, sauf pour afficher le résultat.
- Un temps supérieur à 2 secondes aura des chances de rompre le fil de pensée de l'utilisateur, l'utilisateur remarque le retard. Normalement, aucune rétroaction spéciale n'est nécessaire pendant les retards de moins de 1,0 seconde, mais l'utilisateur perd la sensation de fonctionner directement sur les données.
- 7 secondes est la limite pour garder l'attention de l'utilisateur concentré sur le dialogue. Pour des retards plus longs, les utilisateurs voudront effectuer d'autres tâches en attendant la fin du traitement, de sorte qu'ils devraient recevoir des commentaires indiquant quand le traitement doit se terminer.

Les applications qui utiliseront le socle applicatif de l'Agence de Biomédecine ne devront pas dépasser des temps de réponse de plus de 2 secondes. Cette limite reste un objectif, dans le cas où, pour des raisons fonctionnelles, la quantité de données à exploiter, même après optimisation, ne permet pas un temps de réponse inférieur à 2 secondes, un affichage spécifique pourra être mis en place :

- Affichage asynchrone : l'utilisateur peut continuer à utiliser l'application et sera prévenu ultérieurement que le traitement déclenché est complet
- Affichage avec indicateur d'attente : l'utilisateur est bloqué tant que l'action n'est pas terminée, pour les actions très longues, un indicateur d'attente avec niveau de progression peut être mis en place.

2.3.2. Niveau de robustesse

Il s'agit pour une application de supporter une charge importante d'utilisateurs sur une durée relativement longue.

Des tests de robustesse seront réalisés pour déterminer si l'application est capable de supporter une activité intense, sur une longue période, sans dégradation des performances et des ressources applicatives ou systèmes.

Les exigences de l'Agence de Biomédecine en la matière peuvent varier d'un projet à l'autre. Le niveau de robustesse à atteindre doit être défini en accord avec la cellule architecture et le chef de projet de l'application concernée.

2.3.3. Disponibilité

Deux niveaux de disponibilités selon les besoins applicatifs :

- **Forte disponibilité** : L'application doit être disponible 24/7,
- **Moyenne disponibilité** : L'application doit être disponible pendant la journée de 7 heures à 21 heures du lundi au vendredi,

2.3.4. Volumétrie

La volumétrie qu'une application doit pouvoir supporter en termes de bandes passantes doit être définie en accord avec la cellule architecture et le chef de projet de l'application concernée.

2.3.5. Sécurité

Voici la liste des contraintes de l'agence de biomédecine :

- Suivre [les 10 règles OWASP](#)
- Niveau d'authentification
- Traçabilité sur les données confidentielles
- Eviter le développement spécifique si une librairie reconnue existe
- Utiliser un outil d'analyse de code (type SonarQube) pour vérifier la prise en compte des exigences de sécurité

3. Architecture Applicative

3.1. Présentation générale de l'architecture

Le concept d'une interface de programme d'application (API) existe depuis longtemps. Le concept a été utilisé dans la définition POSIX de 1988 pour la compatibilité entre les variantes d'Unix et d'autres systèmes d'exploitation. CORBA et DCOM ont utilisé le concept à partir des années 1980. ODBC a utilisé le concept de connectivité de base de données à partir de la fin des années 1980.

Le concept d'API est lié à l'information cachée du génie logiciel qui remonte au moins au début des années 1980.

Le terme « dissimulation » implique qu'une modularité efficace peut être obtenue en définissant un ensemble de modules indépendants qui ne communiquent entre eux que les informations nécessaires à la réalisation de la fonction logicielle.

Les API permettent la création d'une interface minimale relativement stable pouvant être utilisée par d'autres systèmes logiciels pour accéder ou manipuler les systèmes ou données sous-jacents. Cela permet d'améliorer la maintenabilité des systèmes ou des données sous-jacents sans perturber les systèmes logiciels qui utilisent l'API.

Ils sont généralement implémentés à l'aide de services Web tels que SOAP, REST

3.2. Focus sur l'architecture orientée API

Lors d'une création d'une application, des problèmes de conception se posent. Une conception robuste et solide est un facteur clé du succès pour le projet. Une API mal conçue entraînera en effet une mauvaise utilisation, des problèmes de performances ou, pire encore, aucune utilisation par les développeurs d'applications front

Créer et fournir une API de pointe nécessite de prendre en compte les points ci-dessous :

3.2.1. Définition d'une API

Une API est un point d'entrée pour établir un dialogue entre deux parties (traditionnellement, le « front-end » et le « back-end » même si des dialogues « back-end – back-back-end » peuvent avoir lieu), ce dialogue est composé de messages qui peuvent avoir plusieurs formes. L'API est exposée par le back-end, qui tourne sur un serveur. Dans le cas de l'agence de la biomédecine, le back est implémenté en Java, et le front est développé en angular.

Le but de construire un ensemble d'API est d'établir de manière ordonnée l'ensemble des fonctionnalités du système d'information, de même que de laisser une possibilité relativement ouverte, standardisée et peu coûteuse d'utiliser ces fonctionnalités.

Note – différence entre ressource et API

Une API permet d'exposer un ensemble de ressources. Il est possible de fragmenter une API contenant beaucoup (trop ?) de ressources en plusieurs APIs intégrant moins de ressources. Ces plus petites APIs sont plus faciles à partager, car elles remplissent un rôle spécifique et leur remplacement impacte moins le reste de l'API.

La question de la couverture fonctionnelle de l'API sera discutée en section 4.1.6.

3.2.2. Impacts de ce changement de modèle

Cela tranche assez nettement avec l'approche suivie jusqu'à maintenant, qui était de concentrer toutes les fonctionnalités d'un domaine applicatif dans un seul bloc (souvent appelé monolithe) constituant un seul livrable, regroupant l'interface graphique, les traitements métiers et la gestion des données.

Ces monolithes applicatifs, pouvaient à la marge exposer certains services pour ouvrir leurs fonctionnalités à d'autres systèmes, mais cela était souvent fait à la demande et nécessitait un travail supplémentaire. En orienté API, l'ensemble des fonctionnalités sont exposées par défaut, ce qui permet une interopérabilité relativement aisée.

Ci-après une figure qui illustre cette approche monolithique (implémentée en MVC) :

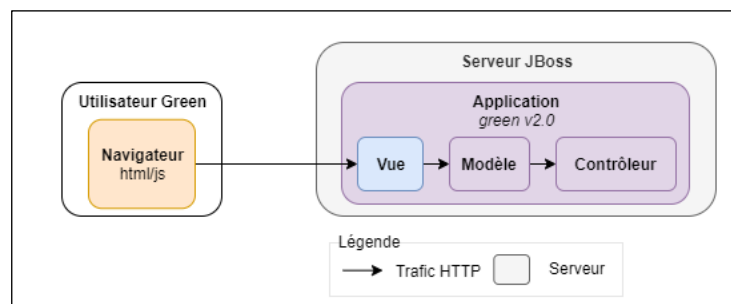


Fig. Approche monolithique

Comme on peut le voir sur cette figure, cette application monobloc contient la vue, qui se charge d'effectuer la partie présentation, les traitements métiers et les accès aux données (non représentés ici).

Avec cette architecture applicative, l'on passe d'une application monolithique à au moins deux applications distinctes :

- Le front-end : qui régit toutes les interactions utilisateurs, effectue des contrôles métiers « de surface » et présente les données.
- Le back-end : qui absorbe les messages en provenance du front-end, effectue certains traitements métiers, et propage les modifications dans la base de données.

Cette séparation est illustrée dans le schéma suivant

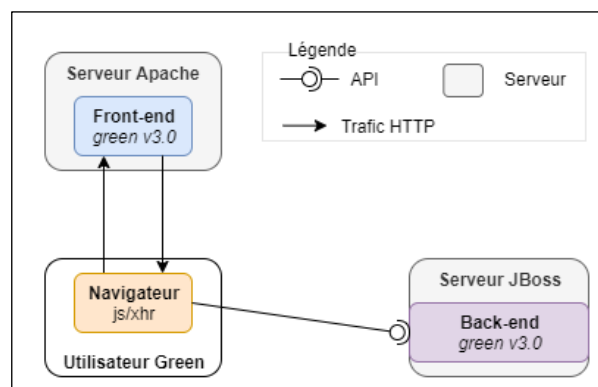


Fig. Approche Orientée API

Cette interopérabilité et cette inventorisation permanentes des fonctionnalités du SI permettent de gagner en maîtrise sur la gestion de ces fonctionnalités, tout en assurant un découplage entre l'application front et les applications back. Ce découplage s'effectue en revanche au détriment de la cohésion de ces deux parties du système. En effet, on avait auparavant toute la chaîne de l'action utilisateur jusqu'au code de persistance au même endroit (monolithe). Ce qui permettait de faire des ajustements relativement faciles à tester. Dorénavant, il faut synchroniser les efforts pour ces mêmes ajustements avec des problématiques :

- **De développements** à cause de développeurs (et d'équipes) probablement différents : le fait qu'il y ait 2 applications et donc plusieurs équipes spécialisées dans chacune leurs domaines (interface graphique ou traitements serveurs) crée un effort de communication supplémentaire. Il y a également un problème de disponibilité de la fonctionnalité d'un côté ou de l'autre : Le développement de la fonctionnalité peut avoir lieu à des moments différents, ce qui provoque du temps de changement de contexte perdu, car les sujets sont laissés « en attente » de l'autre partie.
- **D'infrastructure**, car pour les déploiements, il faut maintenant deux serveurs, qui sont probablement sur des machines différentes, et donc potentiellement gérées par des personnes différentes (pas le même emplacement sur le réseau).
- **De conception** : il faut définir les services de telle manière à ce qu'ils puissent être consommés par le front et servis par le back de la bonne manière.
- **De sécurisation des échanges** : les problématiques de sécurisation sont facilement gérés par l'application monolithique, notamment grâce à un mécanisme de session utilisateur qui permet de garder l'état de l'application, en particulier l'authentification et les autorisations de l'utilisateur.
- **D'organisation pour la maîtrise d'ouvrage**, pour instaurer un processus permettant de ne pas dupliquer certaines APIs ou de les retravailler.
- **D'urbanisation des APIs** : Afin de pouvoir gérer correctement quelles sont les APIs exposées (et dans quelles versions) utilisées par ces applications front, de même que les problématiques d'interdépendances (entre un front et plusieurs backs, ou entre le back et plusieurs fronts)

Les sections suivantes détaillent l'approche nécessaire point par point pour ne pas tomber dans les écueils de l'architecture orientée API.

4. Architecture orientée API et API Management

Tout ce qui est relatif à la bonne gestion des APIs exposées par un SI relève de ce que l'on appelle l'« API Management ». C'est une méthodologie, appuyée par un outil, l'API Manager, qui permet de garantir une organisation efficace dans le développement et la maintenance d'une telle architecture. L'API Manager devient un point central, permettant d'industrialiser le développement d'API.

L'API Manager se positionne comme il suit dans le SI

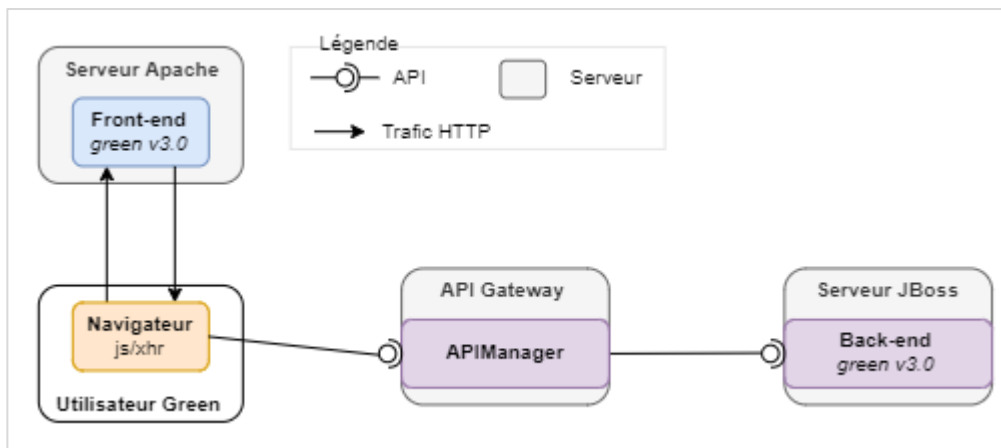


Fig. *Un API Manager pour la gestion d'API.*

L'API Manager est une application web qui se déploie sur un serveur (idéalement physiquement à part) et qui permet de résoudre un certain nombre de problématiques identifiées dans la partie précédente, notamment grâce à ses capacités de proxy, de filtrage, et d'applications automatiques de traitements sur les requêtes qui lui sont envoyées.

L'API Manager choisi est WSO2.

Note Importante

La configuration et le provisionnement des API sur l'API Manager constitue un projet à part entière. C'est la pierre angulaire de l'ensemble du système, et une dérive sur cette partie peut conduire à une situation très compliquée pour la maintenabilité du SI.

4.1. Gestion des développements

Comme évoqué précédemment dans ce document, il n'y a plus qu'une seule équipe en charge des développements, il peut y avoir plusieurs équipes, tout du moins, il y a au moins 2 projets (front et back) qu'il faut faire communiquer.

Cette séparation peut entraîner des déphasages entre les disponibilités de fonctionnalités, comme indiqué dans le schéma suivant :

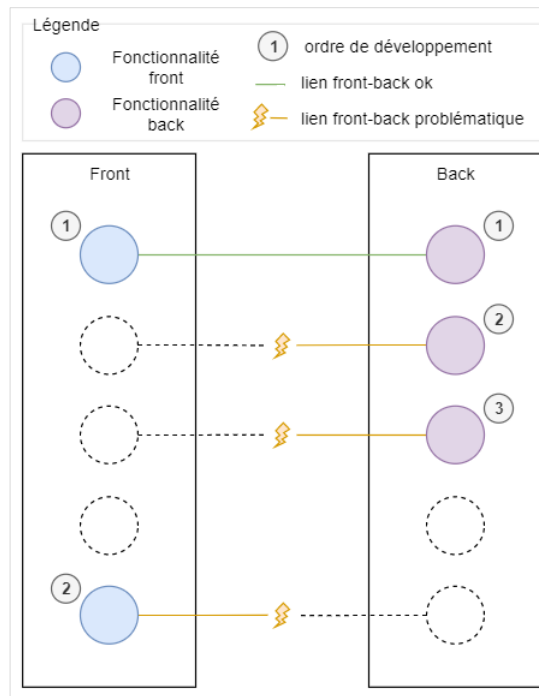


Fig. Désynchronisation de l'avancement des fonctionnalités

La bonne approche est de standardiser l'approche pour les développements en se mettant d'accord sur un contrat commun, avant la mise à disposition d'une API par un service REST, et avant d'en avoir fait l'exploitation, même « bouchonnée » côté front. Cette contractualisation se fait idéalement par un document OpenAPI qui détaille les ressources disponibles, les messages que l'on peut leur envoyer, les réponses attendues, etc.

Une fois ce contrat établi, pour toutes les ressources à développer ou à exploiter, il devient facile et beaucoup moins risqué de réaliser les développements. La convergence devrait être accélérée par cette approche.

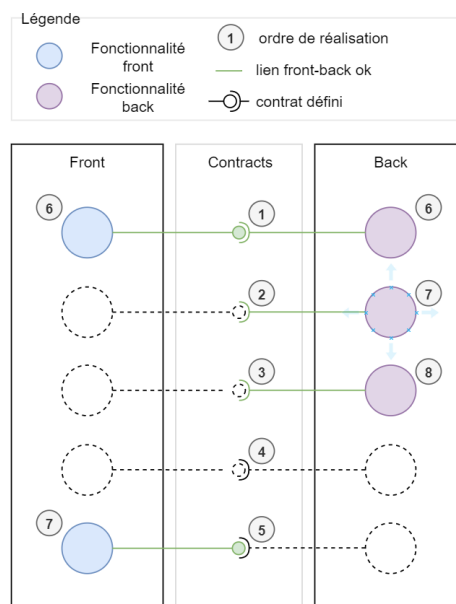


Fig. La définition des contrats doit se faire de manière prioritaire, avant toute réalisation

Afin de pouvoir travailler en avance de phase sur l'implémentation du contrat, il convient pour les développeurs back-end de réaliser des tests d'intégration sur les services exposés, afin de valider la conformité avec les spécifications, avant que les développeurs backend puissent l'exploiter.

Dans l'autre sens, si les services qu'il faut consommer via le front ne sont pas prêts, les développeurs front peuvent bouchonner leurs appels en générant des clients à partir du contrat défini préalablement.

Les contrats peuvent également être définis par spécifications moins technique

Le cas idéal, lorsque c'est possible, est que ce contrat provienne de l'Agence de la Biomédecine, traduisant ainsi sa maturité sur le fonctionnel de l'application. Cela permet également à l'Agence de rester pilote dans l'urbanisation des APIs (voir section 4.1.6).

Note

Les spécifications fonctionnelles étant données sous formes d'exigences utilisateur, la plupart du temps, c'est bien l'expérience Utilisateur (UX) qui possède un ascendant sur le backend. Il faut définir des APIs qui conviennent au moins à la réalisation de la spécification, ce qui indique qu'en cas de discussion sur les formats d'échange (contrat cité ci-avant), c'est la satisfaction des contraintes du front-end qui est prioritaire.

4.2. *Gestion des déploiements*

Les déploiements s'effectuent dorénavant sur plusieurs serveurs, comme montré dans la figure « Approche Orientée API ». Il faut donc s'outiller pour pouvoir facilement déployer un ensemble cohérent (front-back). Le déploiement devrait se faire de manière automatique, à partir de la chaîne d'intégration afin d'éviter les problèmes de manipulation (le déploiement front est relativement peu industriel, en angular tout du moins), et de cohérence entre les fonctionnalités déployées des deux côtés.

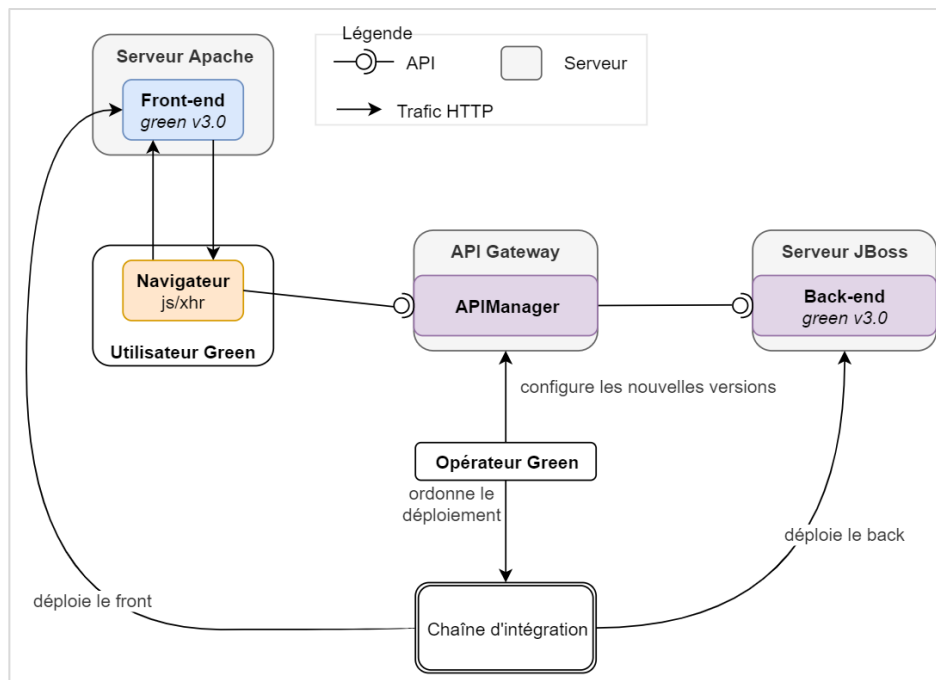


Fig. Les déploiements doivent être automatisés, la configuration de l'API Manager reste manuelle

4.3. Conception des APIs

La problématique de conception des APIs réside dans le fait de les rendre appropriées pour le frontend, mais également de penser à une réutilisabilité pour les autres applications censées se fournir auprès de ce domaine

Les questions de l'accessibilité et de la granularité des APIs sont centrales, elles seront discutées dans les sections dédiées aux bonnes pratiques de développement dans ce document, et complété par le document « Guide de développement ».

4.4. Sécurisation des échanges

Les applications monolithiques permettent d'obtenir un niveau de sécurisation de base : les échanges sont faits dans le cadre d'une session qui contient, entre autres informations, les informations d'accès et d'autorisation de l'utilisateur.

Dans le cas d'une API, il est impossible de gérer ces informations de manière « stateful », l'API étant réutilisée de manière massive, chaque API ne peut pas retenir toutes les informations d'utilisation. Une API est donc dite « stateless » (sans état). Cela implique que les échanges ne sont plus protégés par ce mécanisme de session.

Les données d'utilisations courantes sont quant à elle mémorisées par l'application frontend.

L'approche à mettre en place pour la sécurisation de ces APIs repose sur l'intégration d'un serveur d'autorisation, qui va délivrer un jeton contenant certaines informations de l'utilisateur, que l'application cliente appose à toutes ces requêtes. Il y a deux possibilités afin de vérifier la légitimité de ce jeton :

- Chaque Back-end applicatif aurait la charge de vérifier la légitimité de ce jeton ;
- Un API Manager (WSO2) aurait la charge de vérifier la légitimité de ce jeton pour l'ensemble des Back-end de l'infrastructure.

L'intégration de ce serveur d'autorisation se fait suivant le schéma suivant :

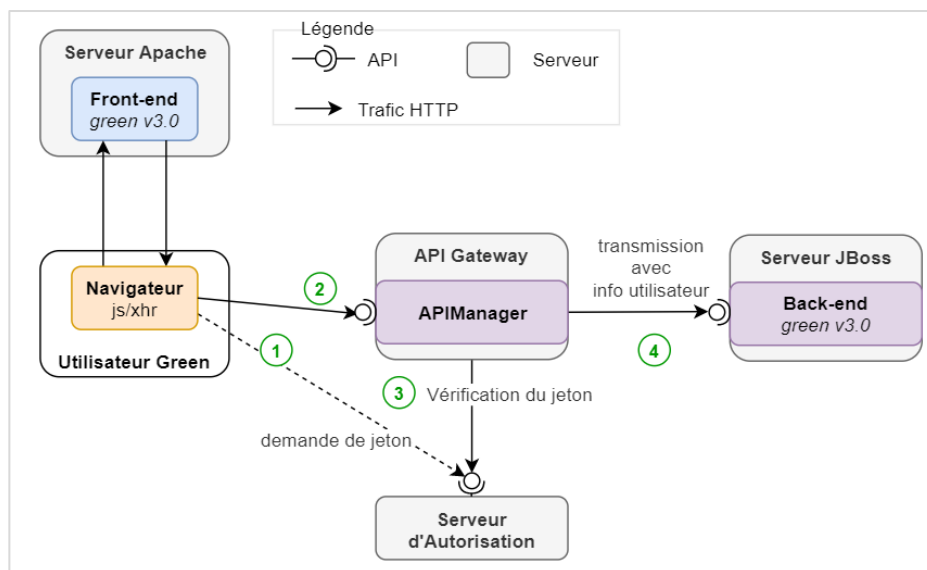


Fig. Inclusion d'un serveur d'autorisation pour la sécurisation de l'API

Dans ce schéma :

1. Le client demande un jeton en envoyant des informations d'authentification. Cette flèche est en pointillé, car l'appel peut se faire au travers de l'API Gateway afin de ne garder qu'un seul destinataire d'appel. Un jeton (encodé sous forme de chaîne de caractères) est produit par le serveur d'autorisation si ces informations sont validées par le serveur d'autorisation.
2. Le client consomme une API par invocation d'une url. Il aura préalablement intégré son jeton à l'appel pour que l'appel soit valide.
3. L'API Gateway intercepte la requête et vérifie le jeton auprès du serveur d'autorisation.
4. Si le jeton est authentifié, l'appel original est redirigé vers le backend. L'API Gateway peut également transmettre les informations contenues dans le jeton.

Pour rappel, depuis le 1^{er} Janvier 2022, l'API Manager est WSO2 en version 3.2.0 et le serveur d'autorisation est Keycloak en version 12.0.4.

Les informations contenues dans le jeton sont personnalisables, c'est une configuration du serveur d'autorisation qui remplira les champs du jeton prévus à cet effet avec les données en provenance de la source d'authentification (LDAP, BDD, etc...).

L'access token de keycloak utilisé dans le cadre de l'infrastructure de l'Agence ne contient pas beaucoup plus d'informations complémentaires relatives à l'utilisateur connecté qu'un « ID token » de l'OpenID Connect. Ces informations relatives à l'utilisateur connecté sont le login de l'utilisateur, le code acteur de Sésame et prochainement les rôles d'accès aux applications et/ou les rôles fonctionnels de l'utilisateur connecté.

4.5. Passage en Orienté API : Impact organisationnel

En orienté API, il y a 3 projets : les applications front et back, et l'API Management. Cette brique « du milieu » doit faire l'objet d'une surveillance particulière. Les prestataires doivent livrer ce

qui est attendu d'eux en cette matière, notamment la documentation de la configuration qu'ils ont utilisée pour pouvoir effectuer la recette. Les prestataires peuvent également être force de proposition lorsqu'ils identifient des ressources à exposer qui n'étaient pas mentionnées dans les spécifications originales, mais les choix doivent être faits en concertation avec la cellule architecture.

La communication entre les équipes projets et les architectes doivent être prépondérantes dans le cadre de cette architecture. Les équipes projets doivent communiquer certains éléments aux architectes tels que :

- La liste des services dont ils ont besoin, ce qui aura un impact sur l'url d'entrée et de sortie et le format de communication entre l'application front et l'API Manager ;
- La fourniture éventuellement d'une documentation technique de l'API « apidocs » définissant les ressources que l'on souhaite exposer ;
- La communication de la mise à jour de l'application back (numéro de version de l'API et/ou du service) ;
- Les besoins en termes d'éléments/headers souhaités à communiquer aux applications back dans le cadre d'un message de médiation dans WSO2.

Les architectes fournissent aux équipes projets le message de médiation adapté aux besoins.

La définition de la liste des services accessibles pour une application WSO2 doit être construite avec la participation des équipes projets concernées et les architectes.

Dans le cadre de besoins non répertoriés, la solution sera définie par la collaboration des équipes projets concernées et les architectes.

A ce niveau-là, l'ensemble des décisions apportées au niveau de l'API Manager peuvent avoir une incidence sur le fichier de configuration des applications Frontend. C'est la raison pour laquelle, les équipes projets et les architectes doivent communiquer afin de permettre une intégration simplifiée.

5. Architecture orientée API : Trajectoire d'implémentation et bonnes pratiques

Cette section s'attache à établir une trajectoire de mise en place pour cette architecture orientée API.

Dans le cadre d'une nouvelle application, les choix d'implémentation sont simplifiés. En effet, dans le cas d'une migration, le poids de l'historique et l'éventuelle cohabitation entre les deux versions (ancienne et nouvelle) rendent la tâche difficile.

Un des intérêts de l'architecture orientée API est également de permettre un découplage au niveau des éventuels liens forts qui peuvent exister entre les applications.

5.1. *Abstraction du modèle historique*

Dans le cadre de la refonte de certaines applications historiques, le modèle de données ne permet pas une exploitation aisée de ces données. Il peut alors y avoir une première étape qui constitue la mise en place d'interfaces qui permettent d'abstraire la difficulté d'exploitation causée par l'empilement successif des modifications (parfois un peu anarchiques) des structures de données.

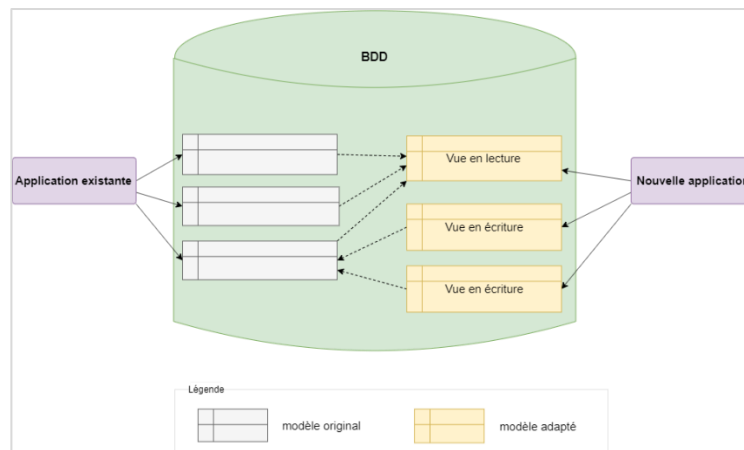


Fig. Abstraction du modèle historique.

Cette proposition permet de gérer une migration en douceur, en proposant des formats de données plus proches du métier.

La nouvelle application (une API exposant une partie du domaine fonctionnel couvert par l'application) bénéficie de concepts métiers d'assez haut niveau, faciles à exploiter.

5.2. **Exploitation des APIs dans l'application existante**

Une fois que cette modélisation d'API est faite, il est possible de passer à l'implémentation de ces dernières, théoriquement facilitée par la mise en œuvre des préconisations de transformation progressive du modèle physique des données.

Les invocations d'APIs pourront être intégrées progressivement dans les applications existantes, en lieu et place des mécanismes de gestion des données.

Cela veut concrètement dire que les applications verraient leur code changer progressivement à mesure que les APIs disponibles deviendraient plus nombreuses.

Cette démarche n'est probablement pas incompatible avec le fonctionnel de l'application existante dans la plupart des cas, et permet de construire petit-à-petit le socle de la nouvelle application.

Il s'agit d'un motif de conception appelé « façade » permettant une transformation progressive du SI.

5.3. **Refonte de l'application cliente**

Une fois les APIs correctement définies, et peut être même totalement implémentées, il est possible de passer à l'implémentation de l'application cliente.

Cette refonte doit se faire progressivement et peut induire des changements limités dans les contrats d'APIs mises à disposition. Il est éventuellement possible de découper l'application à refondre en module et de faire cohabiter des modules de l'ancienne application avec ceux de la nouvelle. Cette approche permet une migration en douceur.

5.4. **Raffinement des services**

Lorsque l'on met au point une API, on l'expose par le biais d'un livrable (application web java qui se matérialise sous la forme d'un fichier « war »). Il peut être judicieux de découper et de répartir les ressources dans des war différents, en les regroupant par fonctionnalité par exemple. Cela permet de converger vers une approche micro-service, rendant l'ensemble des applications Back-end plus facile à maintenir, car extrêmement modulaire et précis...

Les communications Back-end vers Back-end sont à considérer sérieusement, le motif façade peut également être utilisé pour diminuer la combinatoire d'appel et l'interdépendance entre les

applications Back-end, bien que l'utilisation d'un ESB ou d'un autre outil avec des fonctionnalités d'intégration puisse être souhaitable. Voir la section « composants mutualisés »

6. Les règles d'implémentation des API Rest

6.1. *Maturité de l'approche orientée API basée sur REST*

Le modèle de maturité de Richardson (Richardson Maturity Model) est un modèle qui décompose l'approche REST en quatre étapes qui introduisent progressivement les principaux éléments de REST (Ressources ; Verbes et Codes retours HTTP ; Contrôles hypermédia) pour passer d'un modèle RPC sur HTTP à un modèle RESTFull.

Le but de cette section est de valider la « maturité REST » de l'implémentation existante dans un projet, afin d'évaluer l'effort restant pour atteindre un niveau satisfaisant les standards de l'agence.

6.1.1. Niveau 0 : Marais de POX (Plain Old XML)

Le niveau 0 utilise son protocole d'implémentation (normalement HTTP, mais ce n'est pas obligatoire) comme un protocole de transport. C'est-à-dire qu'il canalise les requêtes et les réponses via son protocole sans utiliser le protocole pour indiquer l'état de l'application. Il utilisera seulement un point d'entrée (URI) et un type de méthode (en HTTP, c'est normalement la méthode POST). Des exemples de ceux-ci sont SOAP et XML-RPC.

La métaphore du marais est indicative du manque d'urbanisation ressentie lorsque les opérations et les ressources sont réparties dans le SI de manière non contrôlée.

6.1.2. Niveau 1 : Ressources

Lorsque votre API peut distinguer différentes ressources, elle peut être de niveau 1. Ce niveau utilise plusieurs URI, chaque URI étant le point d'entrée d'une ressource spécifique. Au lieu de passer par

<http://api.agence-biomedecine.fr/doctors>

il est possible de faire la distinction entre

<http://api.agence-biomedecine.fr/doctors/1>

et

<http://api.agence-biomedecine.fr/doctors/2>.

Cependant, ce niveau utilise une seule méthode comme POST pour transmettre toutes les modifications d'état, et GET pour consulter l'état d'une ressource.

C'est un niveau de base que l'on atteint de facto avec l'utilisation d'une servlet en Java, c'est une prémisse de REST.

6.1.3. Niveau 2 : verbes HTTP

Chaque nouvelle application de l'Agence de Biomédecine doit utiliser ce niveau. Dans ce niveau pour être RESTful, votre API doit utiliser les verbes HTTP. C'est-à-dire que dans ce niveau votre API doit utiliser les propriétés du protocole pour gérer l'évolutivité et les échecs. N'utilisez pas une seule méthode POST pour tous, mais utilisez GET lorsque vous demandez des ressources et utilisez la méthode DELETE lorsque vous souhaitez supprimer des ressources. En outre, utilisez les codes de réponse de votre protocole d'application. N'utilisez pas le code 200 (OK) lorsque quelque chose s'est mal passé. En faisant cela pour le protocole d'application HTTP, ou tout autre protocole d'application que vous souhaitez utiliser, vous avez atteint le niveau 2.

6.1.4. Niveau 3 : contrôles hypermédia

Le niveau 3 est le plus haut niveau, l'API est destinée à être consommée par tout type de développeurs.

HATEOAS (Hypermedia As The Engine Of Application State) retourne d'autres informations comme :

- Des liens de mise à jour ;
- Des liens qui permettent de récupérer des données liées
- Des codes retour HTTP cohérent, par exemple :

```
> GET https://api.agence-biomedecine.fr/doctor/42
200 OK

> PUT /doctor/42 {...}
409 Conflict

> DELETE /doctor/42
204 No Content

> GET /doctor/42
404 Not Found
```

6.2. KISS

Keep It Simple and Stupid (en français, dans le sens de « Ne compliquez pas les choses »). Le principe KISS recommande de maintenir les systèmes simplement : par conséquent, la simplicité est un objectif clé dans la conception et donc la complexité inutile doit être évitée.

Il est primordial que l'API soit la plus simple possible et qu'elle s'auto-décrive, de manière que le développeur ait à consulter la documentation le moins possible. On parle alors d'affordance : c'est-à-dire de la capacité de l'API à suggérer son utilisation.

Lors de la conception d'une application, il convient de garder à l'esprit les principes suivants :

- La sémantique d'une application doit être intuitive, qu'il s'agisse de l'URI, du « Payload »(enveloppe d'attribut) de la requête ou des données retournées : un utilisateur devrait pouvoir exploiter l'application en ayant recours le moins possible à la documentation de l'application.
- L'application est conçue pour les clients : les développeurs, et non pour les « data » (représentation interne des données de l'entreprise).

- L'application exposée doit exposer des fonctionnalités simples qui répondent aux besoins du client. Une erreur fréquemment rencontrée est de désigner son application à partir d'un modèle de données existant, qui est souvent complexe. Enfin, en phase de conception, il convient de se focaliser en premier lieu sur les cas d'utilisations principaux, et traiter les cas exceptionnels dans un second temps.

Note – L'API est conçue pour rendre un service métier, et non d'opérer un simple CRUD sur une table

Les deux derniers points des remarques ci-avant sont très importants. Le travers classique dans l'exposition de ressources par API est de croire que ces APIs ne servent qu'à opérer un CRUD (par l'intermédiaire des verbes HTTP POST, GET, PUT, DELETE) sur une table de la base de données.

Certains développeurs utilisent des générateurs (type DAL, Data Access Layer) qui définissent exactement ces expositions directes de données.

Il ne **doit pas** y avoir d'exposition directe des données provenant de la base de données, mais il **faut** absolument que les ressources soient définies pour permettre la résolution de problématiques métiers.

Un exemple classique est celui de la personne et l'adresse courante. En base, il s'agit certainement de deux tables différentes, or pour l'utilisateur souhaitant exploiter ces deux structures de données, l'ensemble ne fait qu'une ressource (personne + adresse), l'entité adresse n'ayant aucun sens fonctionnel sans rattachement à une personne. Modéliser les personnes et les adresses comme deux ressources différentes rendraient leur exploitation difficile pour l'application cliente, en plus de générer un trafic inutile.

Il faut donc veiller à garder cette approche fonctionnelle pour les définitions d'API, cette remarque est consolidée dans la section 5.4 « granularité de l'API »

6.3. *Modèle métier et granularité des ressources*

En théorie, une ressource est égale à une URL mais cela pousse à découper l'ensemble des ressources, il semble important de garder une limite raisonnable dans ce découpage.

Exemple à éviter : les informations d'une personne contiennent une adresse courante qui elle-même contient un pays. Il s'agit d'éviter d'avoir 3 appels à réaliser :

```
>https://api.agence-biomedecine.fr/doctors/1234
<200 OK
< {"id":"1234", "Name":"Antoine Jaby", "address":"https://api.agence-
biomedecine.fr/addresses/4567"}

> https://api.agence-biomedecine.fr/addresses/4567
< 200 OK
< {"id":"4567", "street":"sunset bd", "country": "https://api.agence-
biomedecine.fr/countries/98"}

> https://api.agence-biomedecine.fr/countries/98
```

```
< 200 OK  
< {"id": "98", "name": "France"}
```

Exemple à suivre :

```
> https://api.agence-biomedecine.fr/doctors/1234/addresses/4567/country/  
< 200 OK  
< {"id": "98", "name": "France"}
```

Plus concrètement, il est préconisé :

- De ne regrouper que les ressources qui seront accédées à la suite de manière quasi systématique,
- De ne pas regrouper les collections pouvant avoir de nombreux composants. Par exemple, la liste des emplois courants sont limités (une personne ne peut pas cumuler beaucoup plus que 2 ou 3 emplois à temps partiel), en revanche, la liste des expériences professionnelles peut être très longue,
- De se limiter à deux niveaux d'imbrication : /doctors/addresses/countries

6.4. Veiller à l'aspect Stateless de l'API

Stateless signifie que votre API ne doit pas conserver de contexte entre deux appels successifs. Chaque appel est autoportant et son effet ne dépend que de l'état des ressources au moment de l'appel. On ne parle pas d'absence d'état, mais au contraire de deux états bien distincts qu'il est recommandé de séparer :

- Le client est responsable de l'état de l'application (navigation, session utilisateur authentifiée, enrichissement de l'expérience utilisateur).
- Le serveur est responsable de l'état de ses ressources (persistance, cohérence, transitions entre états).

Les principaux bénéfices de cette séparation sont

- Simplifier l'implémentation du serveur (on m'envoie X, je réponds Y, rien d'autre à gérer, je suis user agnostique),
- Rendre les requêtes cachables (on m'envoie X, je répondrai toujours Y => mettons cela en cache),
- Gagner en résilience et scalabilité : plus de "session utilisateur" consommatrice de mémoire côté serveur,
- Plus de "Sticky session" au niveau du Load Balancer, pas de problématique de réplication de session entre nœuds applicatifs, impactant la complexité, la robustesse et la performance globale de l'architecture.

La mise en place d'architectures REST est enfin l'occasion de revenir dans l'esprit original du protocole http : un client qui interagit de manière transparente avec un serveur via un protocole sans état.

6.5. Aspect non transactionnel

Les transactions par commit à deux phases sont un réflexe souvent culturel mais un mécanisme de compensation fonctionnelle peut s'avérer être une meilleure solution.

Une ressource `https://api.agence-biomedecine.fr/doctors` représente une liste de docteurs. Quand un hôpital souhaite transférer deux docteurs.

Il réalise deux requêtes successives :

- **PATCH** `https://api.agence-biomedecine.fr/doctors/1`
- **PATCH** `https://api.agence-biomedecine.fr/doctors/2`

Si une exception se produit entre les deux actions, alors la mise à jour est potentiellement perdue.

Dans le cadre d'une API REST, pour résoudre cette problématique, il existe deux solutions :

- Utiliser une compensation fonctionnelle. Côté client, appeler une action de rollback. Par exemple un appel à :
 - **PATCH** `https://api.agence-biomedecine.fr/doctors/1` avec les informations originales
 - si **PATCH** `https://api.agence-biomedecine.fr/doctors/2` échoue).
- Annuler la transaction côté serveur (commit à deux phases) via un batch par exemple, et avertir l'utilisateur du comportement exceptionnel de manière asynchrone, tout en rétablissant la cohérence du système(s) modéliser cette suite d'actions interdépendantes par une ressource unique (un message) qui fera alors l'objet d'un seul appel.

La garantie de l'atomicité et éventuellement de la réversibilité de l'action relève de l'implémentation de l'API et non de son utilisation correcte par les utilisateurs. Il est recommandé d'utiliser un mécanisme de compensations fonctionnelles.

6.6. Utilisation de la grammaire REST

La conception d'une API est une étape primordiale car elle constitue un enjeu majeur, dans la mesure où une API mal conçue sera vraisemblablement peu ou pas utilisée par les clients : les développeurs d'applications.

6.6.1. Verbes de la grammaire REST

Le protocole HTTP définit un certain nombre de méthodes qui affectent une signification sémantique à une requête. Les méthodes HTTP utilisées par la plupart des API web RESTful sont les suivantes :

- **GET** récupère une représentation de la ressource à l'URI spécifié. Le corps du message de réponse contient les détails de la ressource demandée,
- **POST** crée une ressource à l'URI spécifié. Le corps du message de requête fournit les détails de la nouvelle ressource.
- **PUT** crée ou remplace la ressource à l'URI spécifié. Le corps du message de requête spécifie la ressource à créer ou mettre à jour.
- **PATCH** effectue une mise à jour partielle d'une ressource. Le corps de la requête spécifie l'ensemble de modifications à appliquer à la ressource.
- **DELETE** supprime la ressource à l'URI spécifié.

L'effet d'une requête spécifique sera différent, selon que la ressource est une collection ou un élément individuel. Le tableau suivant résume les conventions courantes adoptées par la plupart des implémentations RESTful suivant l'exemple du commerce électronique.

<i>Ressource</i>	<i>POST</i>	<i>GET</i>	<i>PUT / PATCH</i>	<i>DELETE</i>
/doctors	Créer un docteur	Récupérer tous les docteur	Mettre à jour des docteurs en bloc	Supprimer tous les docteurs
/doctors/1	Error	Récupérer les détails du docteur 1	Mettre à jour les détails du docteur 1 s'il existe	Supprimer le docteur 1
/doctors/1/orders	Créer une commande pour le docteur 1	Récupérer toutes les commandes pour le docteur 1	Mettre à jour des commandes en bloc pour le docteur 1	Supprimer toutes les commandes pour le docteur 1

6.6.2. Documentation des ressources et de leur utilisation

Il est **obligatoire** de fournir une documentation d'implémentation de la solution. Plusieurs types de documentations peuvent être disponibles :

- Une documentation formelle comme celle-ci

2.2.3 GET evenements-indesirables/{id}			
Ce service permet la récupération d'une ressource représentant un événement indésirable à partir de son identifiant.			
Il renvoie l'événement indésirable et la liste des agents infectieux.			
Titre	Récupère un événement indésirable		
URL	/evenements-indesirables/{id}		
Méthode	GET		
Paramètre du Header			
Path Variable	Type	Obl.	Désignation
id	long	Oui	id de l'événement indésirable
Paramètre d'URL	Type	Obl.	Désignation
Données en entrée			
Code réponse en succès	200 Success		
Code erreur			
Données en sortie (EvenementIndesirableMapper)	<pre>evenementindesirable • [agentinfectieux] • utilisateurCreateur • [acteursConcernes] = [dossierPublications] • [documentPublications]</pre> Les acteurs concernés renvoyés à ce niveau ne correspondent qu'aux enregistrements provenant de la table ACTEUR_CONCERNE. Les informations sont incomplètes (il n'y a pas par exemple l'email).		

Fig. Extrait de documentation de ressource issu des STD Green v3.0

- Une documentation plus orientée développement, telle que celle-ci :

1. Récupérer un groupe de travail

Dans l'application Contact-Everyone V5 **tout est organisé par groupe**. L'utilisation des groupes permet, entre autre, de grouper ses contacts et ses listes de diffusion afin de les partager avec d'autres utilisateurs.

Pour commencer, il convient donc de **récupérer un identifiant de groupe**.

Pour faire cela, utilisez l'URL suivante permettant de lister vos groupes.

GET /api/v1.2/groups

Puis récupérez l'identifiant du groupe sur lequel vous allez effectuer votre envoi de messages.

Pour plus de détails sur la gestion des groupes, rendez-vous dans la section **Gérer ses groupes**

Format de requête

```
URL url = new URL("https://[SERVER_URL]/api/v1.2/groups");
HttpURLConnection con = (HttpURLConnection) url.openConnection();

con.setRequestMethod("GET");
con.setRequestProperty("Authorization", "Bearer [Access-Token]");
con.setRequestProperty("Content-Type", "application/json");

BufferedReader in = new BufferedReader(new InputStreamReader(con.getInputStream()));
String output;
StringBuilder response = new StringBuilder();

while ((output = in.readLine()) != null){
    response.append(output);
}

in.close();
```

Fig. Extrait de la documentation d'API CEO SMS (Orange Business Services)

Il est préconisé d'illustrer systématiquement vos appels d'application via des exemples (au moins des exemples cURL) dans la documentation de l'application, qui peuvent être utilisés en copier-coller, pour lever toute ambiguïté concernant les modalités d'appel.

Exemple :

```
> CURL -X POST \  
-H "Accept:application/json" \  
-d {"address": { "@type": "PostalAddress", "addressLocality": "France",  
"addressRegion": "Ile de France", "postalCode": "75000", "streetAddress": "20 rue  
de la paix" }} \  
https://api.agence-biomedecine.fr/doctors/1234/addresses/
```

6.6.3. Filtre et Pagination

L'exposition d'une collection de ressources par le biais d'un seul URI peut conduire les applications à extraire d'importantes quantités de données alors que seul un sous-ensemble d'informations est requis.

Des paramètres peuvent être utilisés afin de filtrer et paginer les éléments. Ci-dessous dans le tableau, les mots-clés qui permettent de filtrer et de paginer les données :

Une ressource <https://api.agence-biomedecine.fr/organ> représente une liste d'organes.

URL	Description
/organs?fields=organid,quantity	Retourne les données avec seulement l'identifiant de l'organe et la quantité
/organs?limit=25	Retourne les 25 premières lignes.
/organs?sort=organid	Retourne les lignes triées par l'identifiant de l'organe
/organs?limit=5&offset=5	Retourne la ligne 6 à 10
/organs?sort_by=asc(organid)	Retourne les lignes trier dans l'ordre croissant
/organs?organid=10	Retourne la ligne dont l'identifiant de l'organe est égal à 10.
/organs?quantity=10&fields=organid&limit=5&offset=5	Retourne l'identifiant de l'organe de la ligne 6 à 10 dont la quantité est égale à 10

7. Architecture logicielle des composants

7.1. *Back-end*

Le backend est structuré comme le montre le schéma suivant :

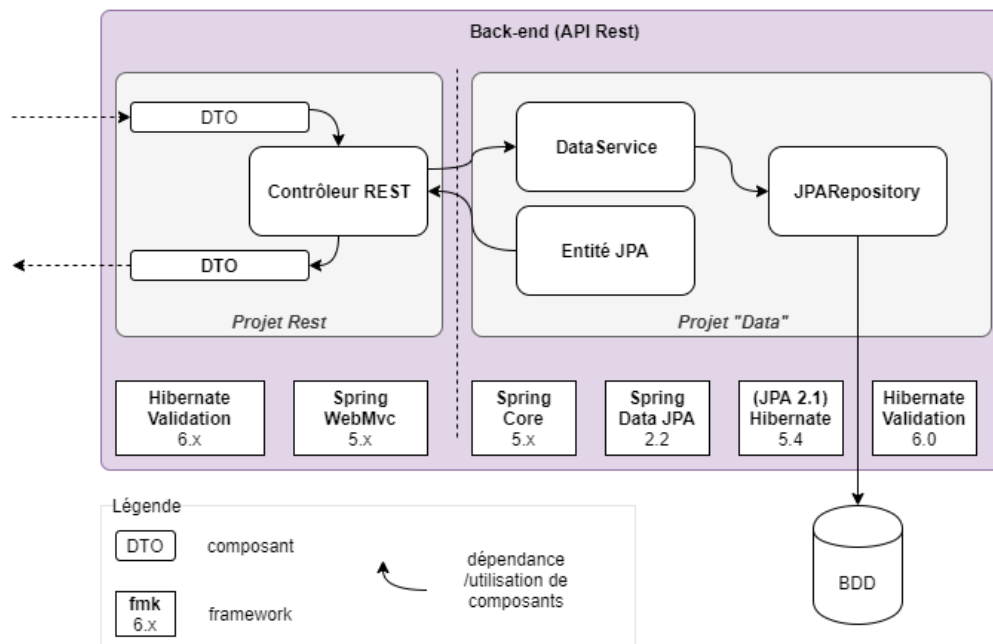


Fig. Structure du backend (versions utilisées en 2020)

Commentaires sur les DTO et les méthodes HTTP utilisées.

- Les DTO (Data Transfer Object) permettent de matérialiser l'état de la ressource à créer ou mettre à jour, ou à récupérer.
- Dans le cas d'opérations GET, il n'y a pas de DTO en entrée, mais un ensemble de paramètres permettant de récupérer les ressources appropriées.
- Dans le cas d'opérations POST, PUT, ou DELETE, il faut qu'un message indiquant le succès de l'opération (200 si aucun retour n'est nécessaire, 201 si une ressource a été créée) soit retourné, pas besoin de DTO en sortie.
Noter que DELETE est une opération idempotente : l'invoquer sur un objet métier déjà détruit ou inexistant n'a pas d'effet indésirable.
- Dans le cas de PATCH (mise à jour partielle), il convient de définir spécifiquement ce que l'on veut retourner. Il semble qu'une version complète de la ressource modifiée, telle qu'elle est en base après la modification soit appropriée.

7.2. Frontend

L'organisation logicielle de l'application frontend

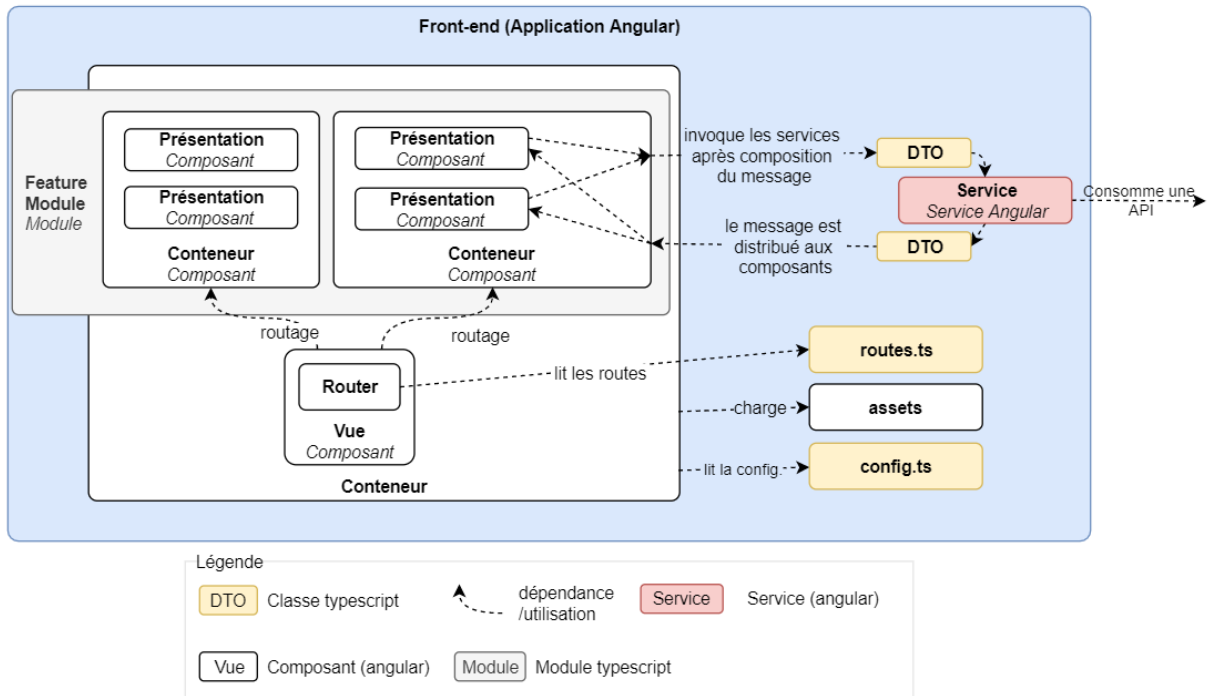


Fig. Organisation interne d'une application Angular

Commentaires :

- Il y a 3 types de composants différents :
 - o Les composants de présentation en charge de présenter les données et de les modifier. Ces données sont ensuite remontées au niveau d'un composant « conteneur ».
 - o Les composants conteneurs qui invoquent les services, distribuent les données dans les composants de présentation et les récupèrent afin d'invoquer un autre service.
 - o Les composants « vue », qui orchestrent les navigations et les agrégations de conteneurs dans l'application
- Les services sont assez symétriques par rapport au backend, les DTO sont donc assez semblables à ceux qui peuvent être présents sur le Back-end.
- Les routes permettent de piloter la navigation dans l'application
- La configuration est externalisée dans le fichier config.ts elle contient entre autres choses, les endpoints de destination pour le Back-end.

7.3. Batches

Les traitements batches doivent être réalisés avec Spring Batch, ce framework est structurant et permet d'isoler correctement les opérations de lectures, de traitement et d'écriture. Ces opérations sont regroupées dans un objet « étape » (step). Une succession d'étapes constituent un job, et plusieurs jobs peuvent appartenir à une tâche.

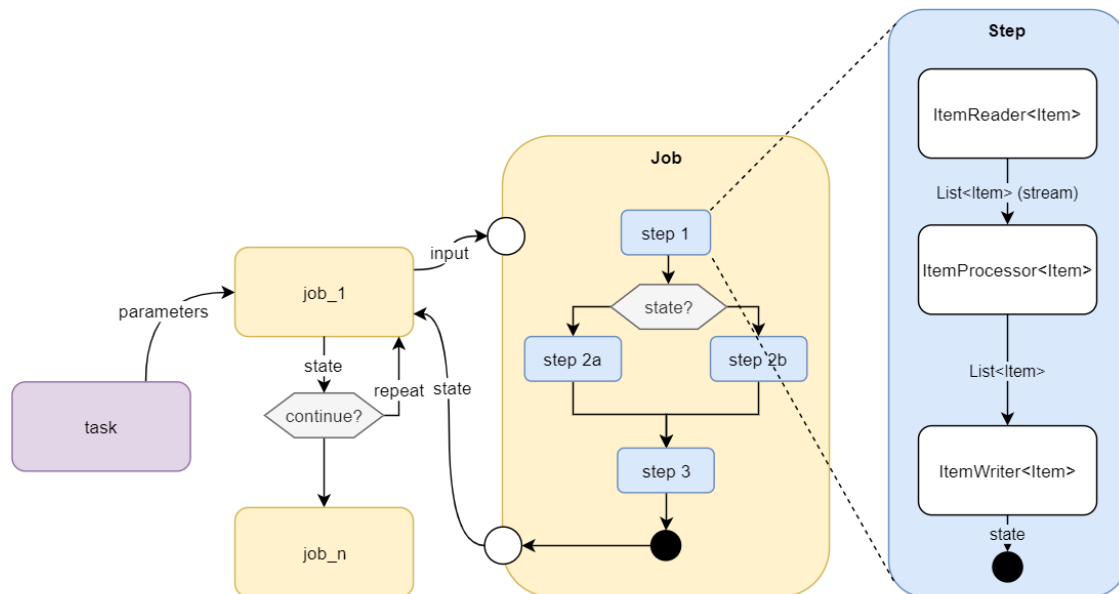


Fig. structure d'un programme écrit avec Spring Batch

La modularité imposée par le framework permet une bonne réutilisabilité (même si dans les faits cette dernière est rarement exploitée).

A propos de la consommation des services existants : Les opérations en volume sont mal adaptées à la réutilisation des services REST, cela peut être fait dans une certaine mesure pour la récupération de listes (recherches des éléments à traiter), mais la plupart des opérations de mise à jour proposées par des services REST sont unitaires.

Pour ces traitements en volume, il est possible de s'appuyer sur la brique d'accès aux données (identifié comme « Projet Data » dans la section décrivant le backend), afin de ne pas réécrire cette partie-là.

Il faut noter que le code existant pour les services REST peut être sous-optimal en cas de traitement batch et que certains accès nécessitant d'être plus proches de la source de données, il faudra écrire spécifiquement ces parties. Ce dernier point ne constitue pas une infraction aux bonnes pratiques.

Pour pouvoir ouvrir les traitements batches à l'intégration au sein du SI, il conviendra d'utiliser Spring Integration. Cette surcouche permet des lancements de batches par réception de messages, tels que ceux que peuvent envoyer un ESB.

Il est également possible de prévoir une surcouche REST pour piloter les batches, permettant ainsi de déclencher un batch, d'en interroger l'état ou encore de recommencer un step.

Si un tel besoin survient, il conviendra d'en spécifier les contours avec la cellule architecture afin de préciser le cadre à respecter pour se raccorder à l'administration.

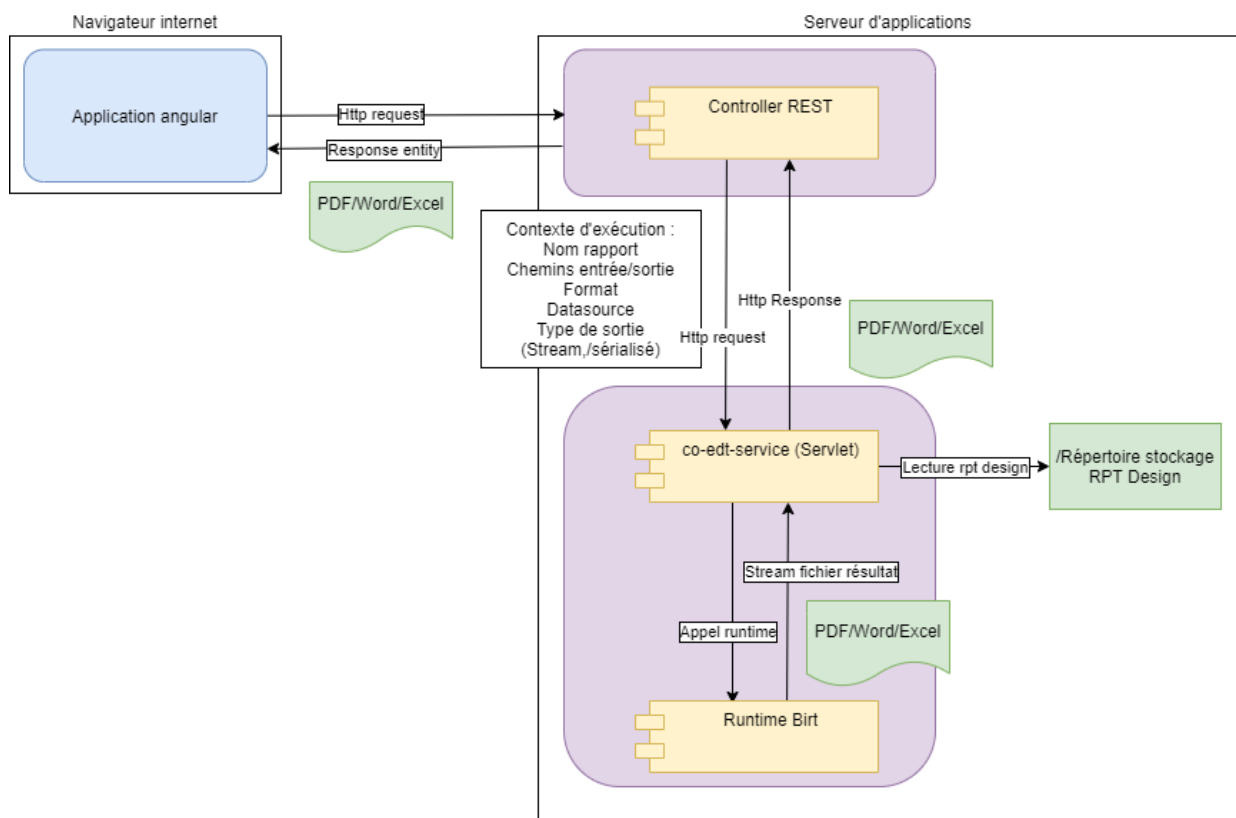
7.4. Serveur d'édition Birt

Birt (Business Intelligence and Reporting Tools) est un outil de mise en forme des données utilisé par des clients riches ou par des applications Web, et permettant également la production de rapports d'impression.

Il est constitué d'un outil (Basé sur Eclipse) permettant de créer les rapports (RPT Design) et d'un runtime permettant de les générer dynamiquement à la demande des utilisateurs, au format souhaité (PDF, Word, Excel). La mise en place d'un rapport s'effectue de la manière suivante :

- L'équipe de développement se charge de créer le fichier RPT Design répondant aux besoins de la MOA (Par exemple pour l'édition du dossier donneur : dossier-donneur.rptdesign) ;
- Le fichier est intégré au livrable de l'application correspondante et livré à l'Agence ;
- Le fichier est déployé dans un répertoire spécifique du serveur d'application afin d'être servi par un service d'édition ; ce répertoire doit être en dehors des répertoires de déploiement de JBoss afin de ne pas être écrasé ;
- Lorsqu'un utilisateur souhaite éditer un rapport, la demande est faite par l'application utilisée au service d'édition encapsulant le runtime Birt ;
- Le service lit le rapport correspondant, le transmet au runtime Birt avec les paramètres nécessaires ; ce dernier crée alors le rapport au format souhaité ;
- Le rapport est ensuite renvoyé à l'application appelante pour y être affiché.

Son intégration dans le SI s'effectue par sa mise en service selon l'architecture suivante :

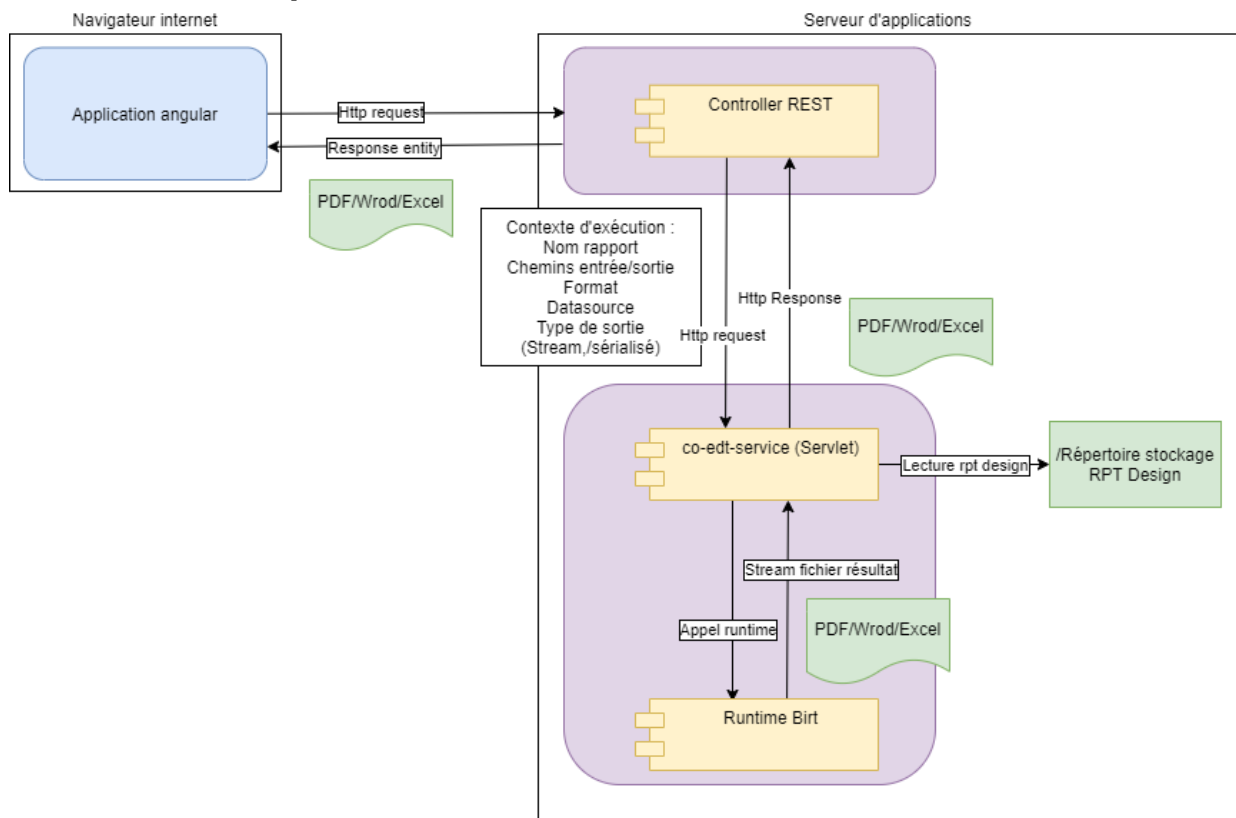


- L'application angular, qui s'exécute dans un navigateur internet, appelle le endpoint du service dédié aux éditions, à la demande de l'utilisateur en lui transmettant toutes les informations nécessaires ;
- Le contrôleur REST, en fonction de l'action demandée :
 - Vérifie que l'utilisateur a le droit d'éditer le document demandé ;
 - Prépare le contexte d'exécution demandé (nom du rapport, datasource, paramètres, etc.) ;

- Appelle la servlet dédiée ;
- Cette servlet (co-edt-service) est un composant générique, déjà utilisée dans le cadre de Cristal Receveur, qui encapsule le runtime Birt chargé de générer les rapports ;
- Les rapports sont stockés dans un répertoire dédié du serveur d'application ; ce répertoire est externalisé, c'est-à-dire qu'il n'est pas dans le contexte de déploiement du JBoss hébergeant le service ; de ce fait, les rptdesign ne sont pas inclus dans le war des applications mais en tant que ressources externes à ajouter dans le répertoire dédié.
- Le runtime lit le rapport (RPT Design), et l'exécute avec le contexte transmis depuis le contrôleur REST). Le résultat (Document PDF, Word ou Excel) est alors retourné sous forme d'un Stream d'octets à la servlet (réponse http) qui elle-même le retourne à contrôleur REST ;
- Le contrôleur le retourne à l'application Angular (http response de type entity response) qui a à sa charge de son affichage dans le navigateur.

8. Mutualisation des composants

8.1. Editique



8.2. Upload (Stockage des Documents)

Un module d'Upload serait une interface standard pour la gestion des documents de l'agence de biomédecine permettant aux utilisateurs de les stocker, de les récupérer et de les partager. Ce module garantirait le contrôle de la sécurité et de la version des documents. À l'aide d'un système de gestion de documents, vous pouvez créer un document en utilisant votre traitement de texte favori et le stocker dans le système de gestion de documents dans l'un des emplacements disponibles.

Ce module pourrait être utilisé pour charger un fichier PDF ou un fichier image correspondant au résultat d'un examen d'imagerie médicale. Ces fichiers pouvant être volumineux le module de chargement prend en charge la complexité technique du chargement de tels volumes.

La mise à disposition de ces services est un objectif à relativement court-terme pour l'agence. Il faudra par conséquent vérifier la disponibilité de ce service au moment où la fonctionnalité est identifiée au sein d'un projet.

8.3. Utilisation d'un ESB

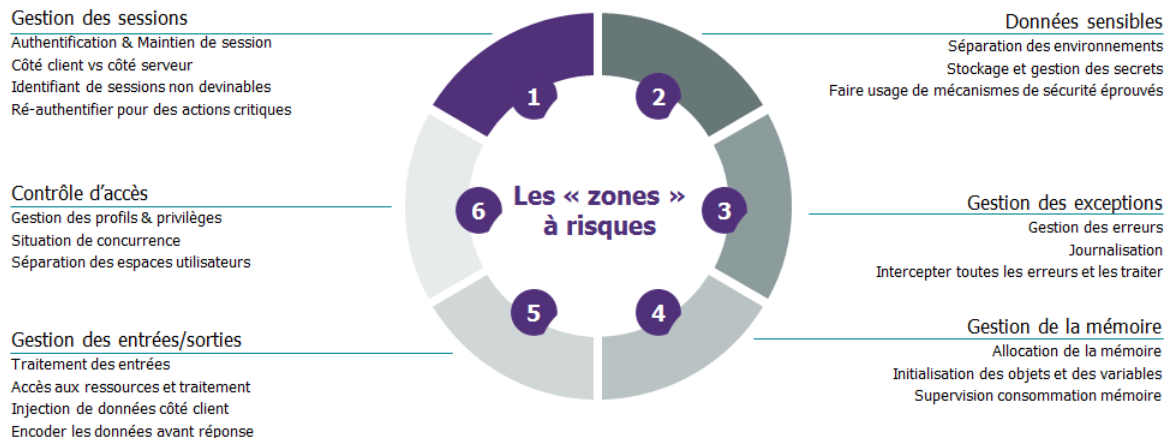
Dans certains cas, où la consommation de composants appartenant à d'autres domaines est fréquente et où le nombre d'interconnexions est important, il peut apparaître que l'utilisation d'un ESB est souhaitable. La situation aura dans la plupart des cas été identifiée par la cellule

architecture. Dans le cas contraire, le prestataire peut être force de proposition et indiquer la situation éligible à l'ESB.

9. Sécurité des applications

9.1. OWASP

En ce qui concerne, la sécurité des API, il s'agit essentiellement de s'assurer de couvrir les principales zones à risques d'une application web et de déterminer les mesures de sécurité appropriées.



[En suivant les règles OWASP](#), cela permet d'éviter d'avoir des problèmes dans les zones à risques.

9.2. OAuth2

Il est préconisé de sécuriser l'API via le protocole OAuth2. [Plus d'informations](#)

Rendre les API disponibles du l'internet ne signifie pas qu'elles sont en accès libre, de manière non sécurisée. Il existe des protocoles Web standardisés pour gérer l'authentification et l'habilitation des applications appelantes et des utilisateurs connectées à ces applications. La pierre angulaire étant le protocole OAuth2, qui est un Framework d'autorisation proposant un mécanisme de délégation d'autorisation permettant à une application tierce d'accéder à une ressource protégée pour le compte d'un utilisateur considéré comme le propriétaire de la ressource, sans avoir à communiquer à l'application l'identité (login/mdp) de l'utilisateur connecté. OAuth2 peut être étendu pour prendre en charge l'authentification via le protocole OpenID Connect. Ces protocoles simples sont bien éloignés des solutions complexes généralement utilisées par les entreprises (WS-Security, SAML2...).

L'objectif étant de permettre aux développeurs de consommer facilement l'API, ces protocoles n'incluent pas de mécanismes de signature ou de chiffrement applicatif des messages (ils reposent uniquement sur la couche de transport TLS qui garantit l'intégrité et la confidentialité des échanges) et permettent une consommation de l'API par tout type d'appareil : non seulement des serveurs, mais aussi des navigateurs, des applications natives...

Notons que JWT est fréquemment utilisé conjointement à une implémentation OAuth2. Il s'agit d'une RFC séparée ajoutant une couche de chiffrement qui complexifie les appels OAuth2 mais apporte une optimisation fondamentale permettant de récupérer les informations liées à l'utilisateur connecté et aux habilitations, sans effectuer de nombreux appels serveurs. Son utilisation est normalisée via la spécification OpenID Connect.

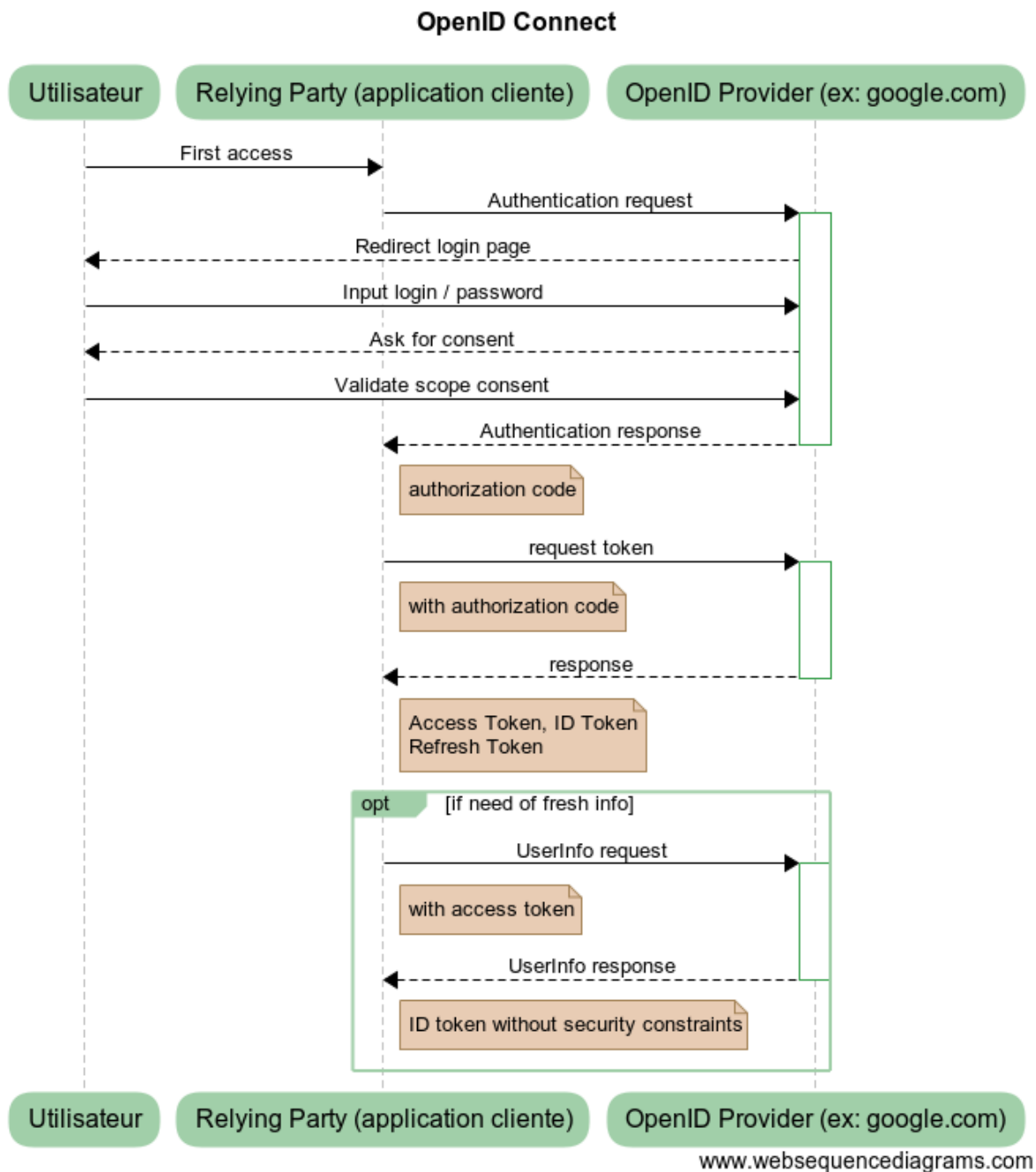
Enfin, le mécanisme de sécurité à implémenter dépend du type de ressources. Il existe en effet deux types de ressources :

- Les ressources publiques : dont les données ne dépendent pas d'un utilisateur final. L'application appelante est habilitée via une **api_key**. C'est le mécanisme utilisé pour l'identification d'une application cliente (section 4.1.6).
- Les ressources privées : dont les données dépendent d'un utilisateur final. L'application appelante est habilitée via un **access_token** OAuth2 qui permet d'identifier l'utilisateur connecté (section 4.1.5)
- L'**access_token** peut être renouvelé sans nouvelle authentification grâce à un **refresh_token**, transmis en même temps que l'**access_token**.

Les six recommandations suivantes sont essentielles pour une mise en place sécurisée du Framework :

- **Authorization code:** Valider strictement les *authorization codes* et clients associés
- **State and PKCE:** À utiliser pour garantir l'intégrité d'une cinématique complète
- **Authorization ≠ Authentication:** Utiliser OpenID Connect pour authentifier, OAuth pour déléguer l'accès
- **Secret local :** L'application est munie d'identifiants lui permettant de s'authentifier auprès du serveur OAuth : ne pas mettre ce secret (identifiant du service) dans l'application mobile ou le considérer compromis
- **Redirect URI:** Valider strictement les URLs de redirection vers l'application, sans wildcard
- **Implicit:** Éviter le « *Implicit grant* » dans la mesure du possible (et se tourner vers le *proxy pattern*).

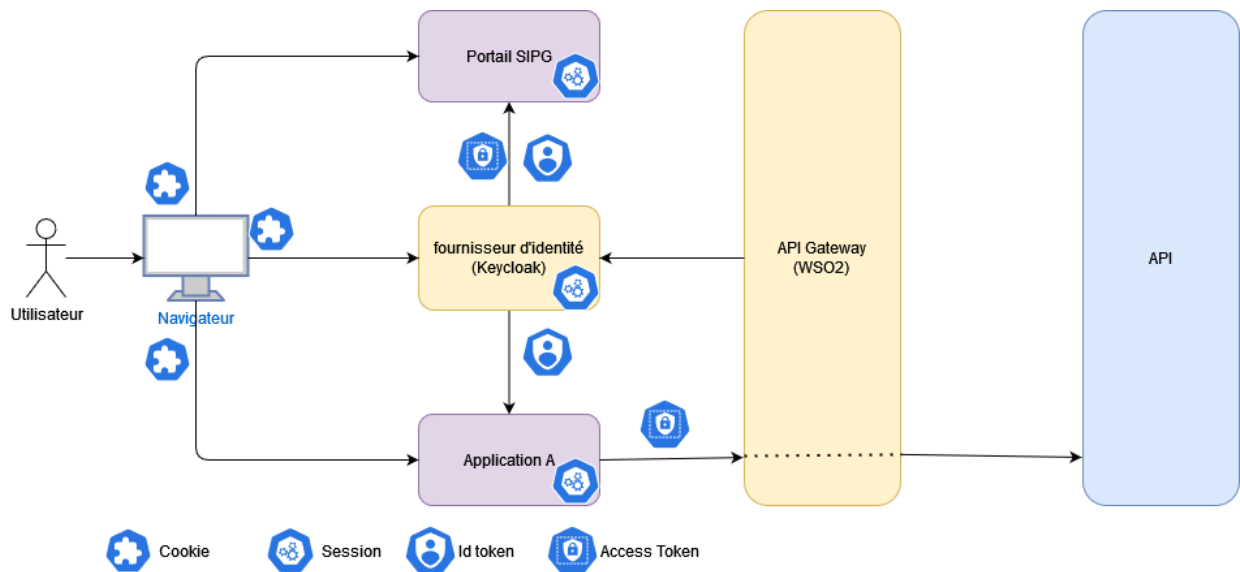
Pour rappel le protocole d'authentification par Authorization Code Flow avec PKCE doit être effectué de la manière suivante :



9.3. Implémentation d'un Single Sign On (SSO) respectant le protocole OAuth2

Le Single Sign On (SSO) est la possibilité pour un utilisateur de s'authentifier une fois et d'avoir accès à plusieurs applications sans avoir à le refaire. Ceci est généralement rendu possible par l'utilisation d'un unique fournisseur d'identité gérant sa session, et en opérant à partir du même navigateur.

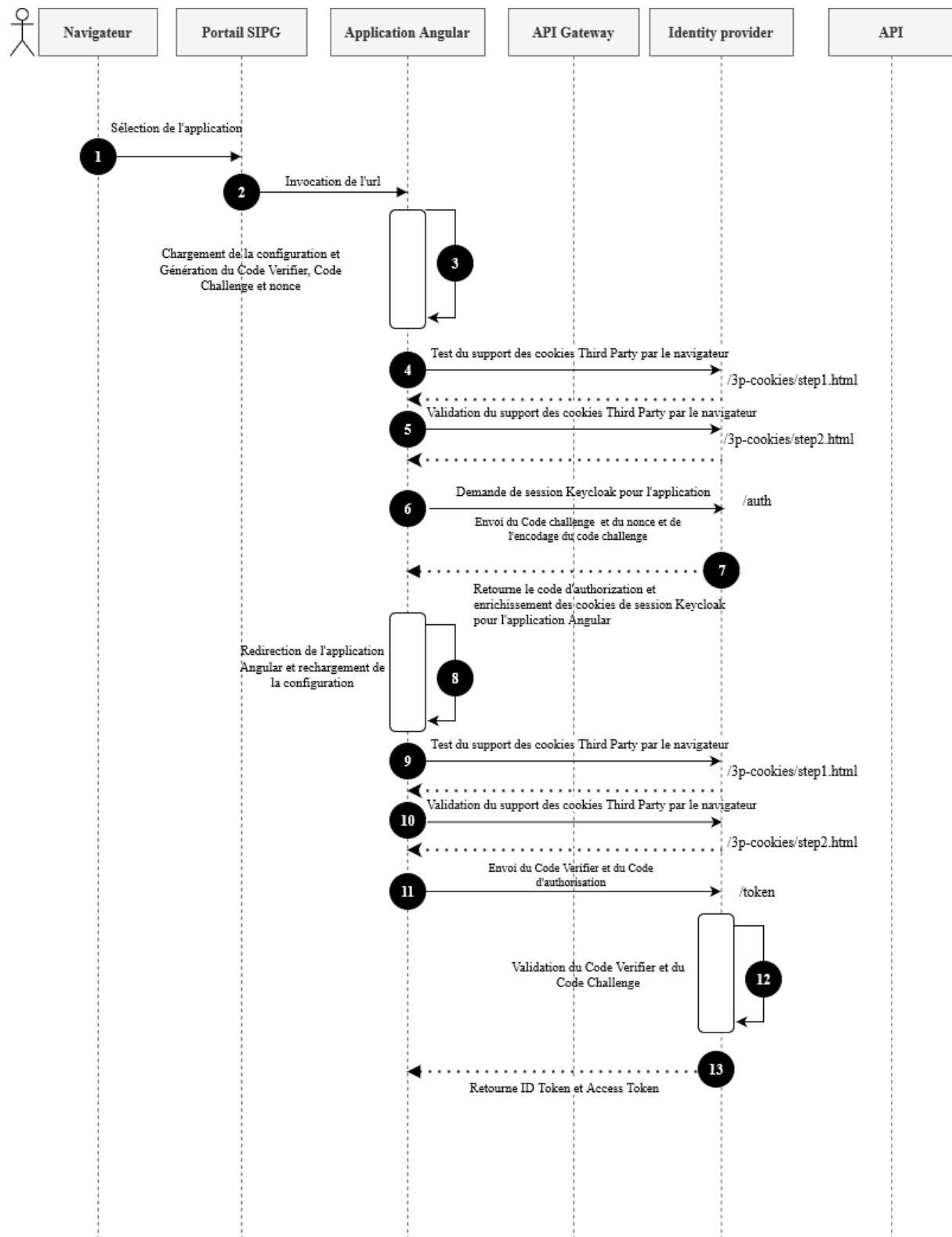
Le SSO est possible du fait du maintien de la session par le fournisseur d'identité ; le mécanisme est le suivant :



- L'utilisateur accède à l'application "Portail SIPG" qui redirige le navigateur vers la mire de connexion du fournisseur d'identité.
- Il saisit ses informations de connexion et en cas de succès, le fournisseur d'identité crée une session, un cookie avec ces informations de session et redirige le navigateur vers le portail SIPG en lui transmettant les tokens ;
- Si l'utilisateur accède depuis le même navigateur à l'Application A, ce dernier sera authentifié par le cookie de session contenant les informations de session, le fournisseur d'identité ajoutera les informations de session complémentaire liée à l'application A et redirige le navigateur vers l'application A en lui transmettant les tokens ;
- Si l'utilisateur souhaite afficher des nouvelles informations, il effectue un appel à une API REST avec l'access token, l'API Gateway va vérifier la validité du jeton d'accès et rediriger si oui vers l'API qui transmet les résultats de la requête à l'application A.

L'implémentation d'un Single Sign On SSO doit respecter le protocole d'Autorisation Code Flow avec PKCE.

Le diagramme de séquence suivant représente le protocole d'autorisation Code flow mis en place dans le cadre du Single Sign On :



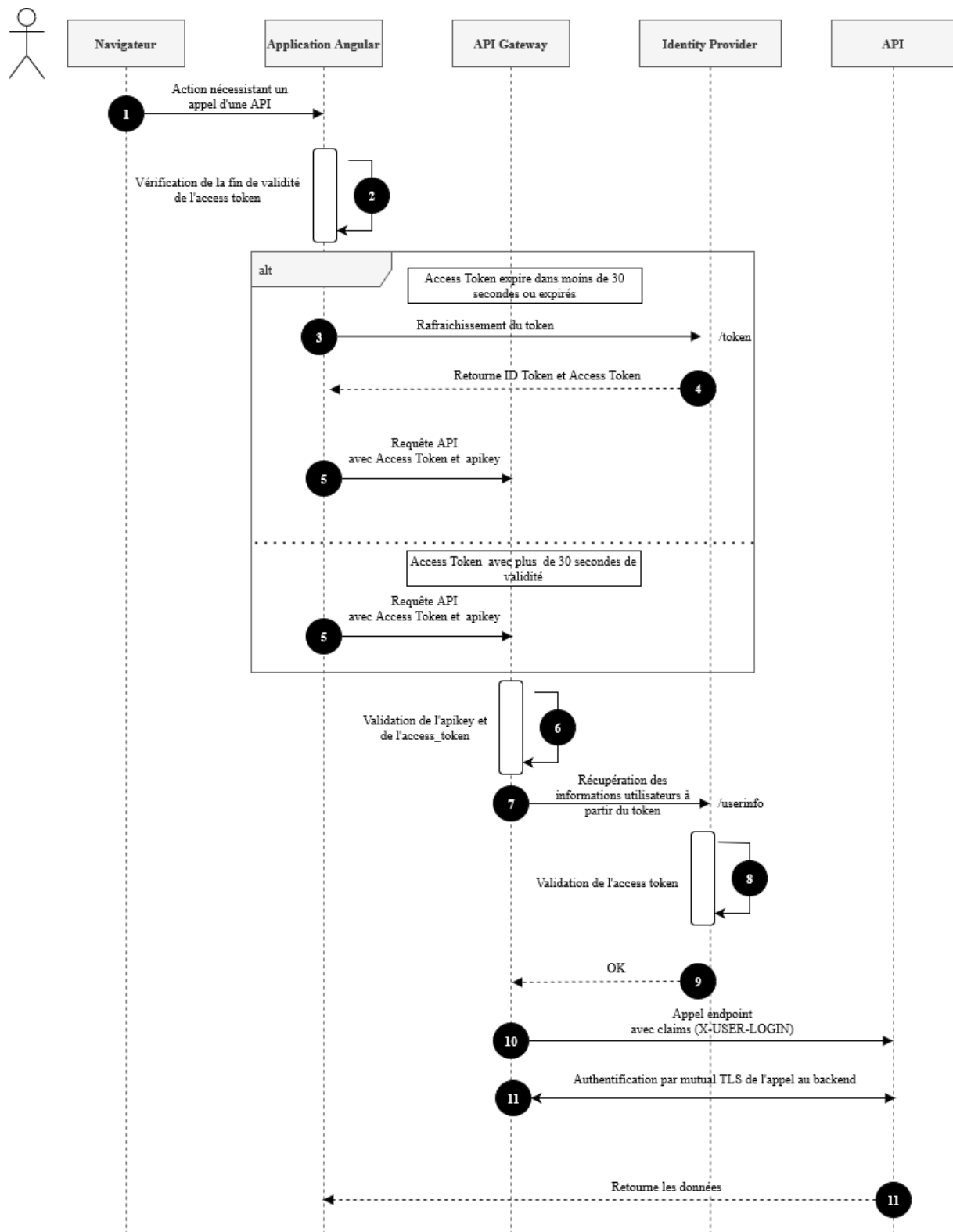
9.4. Sécurisation des Appels aux services REST par WSO2 (API Manager)

A l'instar du protocole d'Authorization Code Flow, pour l'authentification des utilisateurs, la sécurisation des API peut être renforcé par l'application de certaines recommandations de l'ANSSI :

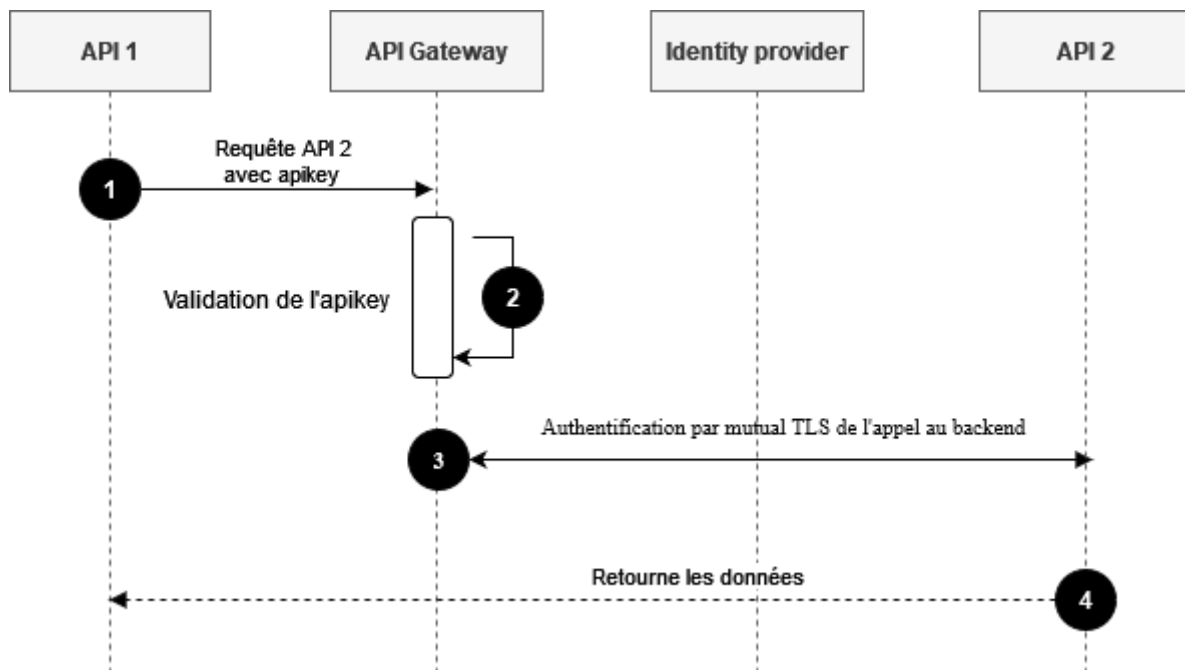
- La mise en œuvre du Mutual TLS entre l'API Manager et le serveur du backend ;
- L'autorisation de certaines ressources au niveau de l'API Manager et l'utilisation de définitions de ces ressources dans WSO2 ;
- La mise en place de traitement de requêtes, le contrôle de la présence de certains rôles contenus dans les jetons d'accès du fournisseur d'identité et la restriction d'accès à certaines APIs dans ces cas-là.

En effet, WSO2 permet de mettre à disposition seulement certaines ressources sur une url de WSO2 en se basant sur les documentations techniques des applications.

La sécurisation des appels aux APIs depuis les applications de type SPA sont représentées par le diagramme de séquences suivant :



Le diagramme de séquence ci-dessous représente quant à lui, les appels de type Backend à Backend :



9.5. WAF

Il est conseillé de mettre en place un pare-feu applicatif (WAF). Afin de garantir la sécurité des applications.

Un Web Application Firewall (WAF) est un type de pare-feu qui vérifie les données des paquets afin de protéger la couche application du modèle OSI. Dans l'architecture globale du système, un WAF est placé en avant de l'application Web qui doit être protégée. Chaque demande envoyée est d'abord examinée par le WAF avant qu'elle n'atteigne l'application Web. Si cette demande est conforme avec l'ensemble de règles du pare-feu, ce dernier peut alors transmettre la demande à l'application. D'une manière générale, un WAF peut être défini comme une politique de sécurité mise en place entre une application web et l'utilisateur final.

Par exemple : Mod Security est un pare feu applicatif, dont le rôle est de filtrer les requêtes entrant sur un serveur HTTP Apache ou NGINX.

Le rôle de WAF « bas-niveau » ne doit pas être opéré par l'API Gateway dans la mesure du possible. Il faut que cette dernière se concentre exclusivement sur le routage et de vérification de la légitimité des appels.

9.6. Intégration avec le portail SIPG

Dans la nouvelle version du mécanisme d'authentification, c'est le portail qui assure la récupération du jeton.

Cette section résume les étapes permettant cette mise en place dans le portail.

9.6.1. Utilisation d'OPENID Connect (OIDC)

OpenID Connect (OIDC) permet d'utiliser la technologie OAuth afin d'y ajouter les informations nécessaires pour la gestion fine des accès et des autorisations.

Dans ce modèle, l'utilisateur (aussi appelé **Requesting Party**) tente d'utiliser l'application (**Relying Party**) qui elle-même va adresser sa demande d'accès aux ressources de l'application

(**Services REST**). L'accès à cette ressource est protégé par le serveur d'autorisations (Authorization Server) auprès duquel l'utilisateur s'est déjà authentifié au préalable.

Le schéma ci-après montre le flux OIDC adapté à la configuration du portail

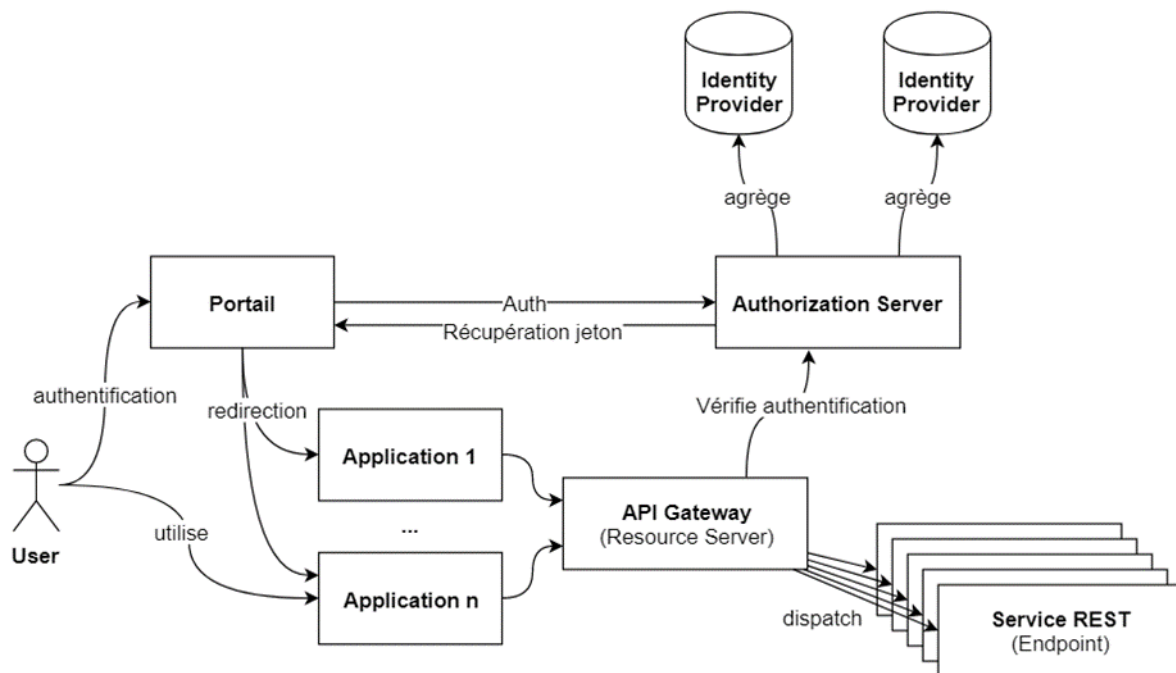


Fig. Flux Open ID Connect avec un portail

Comme on peut le voir, OIDC s'intègre particulièrement bien dans cette architecture incluant le portail.

9.6.2. Fourniture d'une application Blanche

L'application blanche permet d'avoir un exemple d'utilisation de l'authentification (ici il s'agit d'une application mimant le fonctionnement du portail).

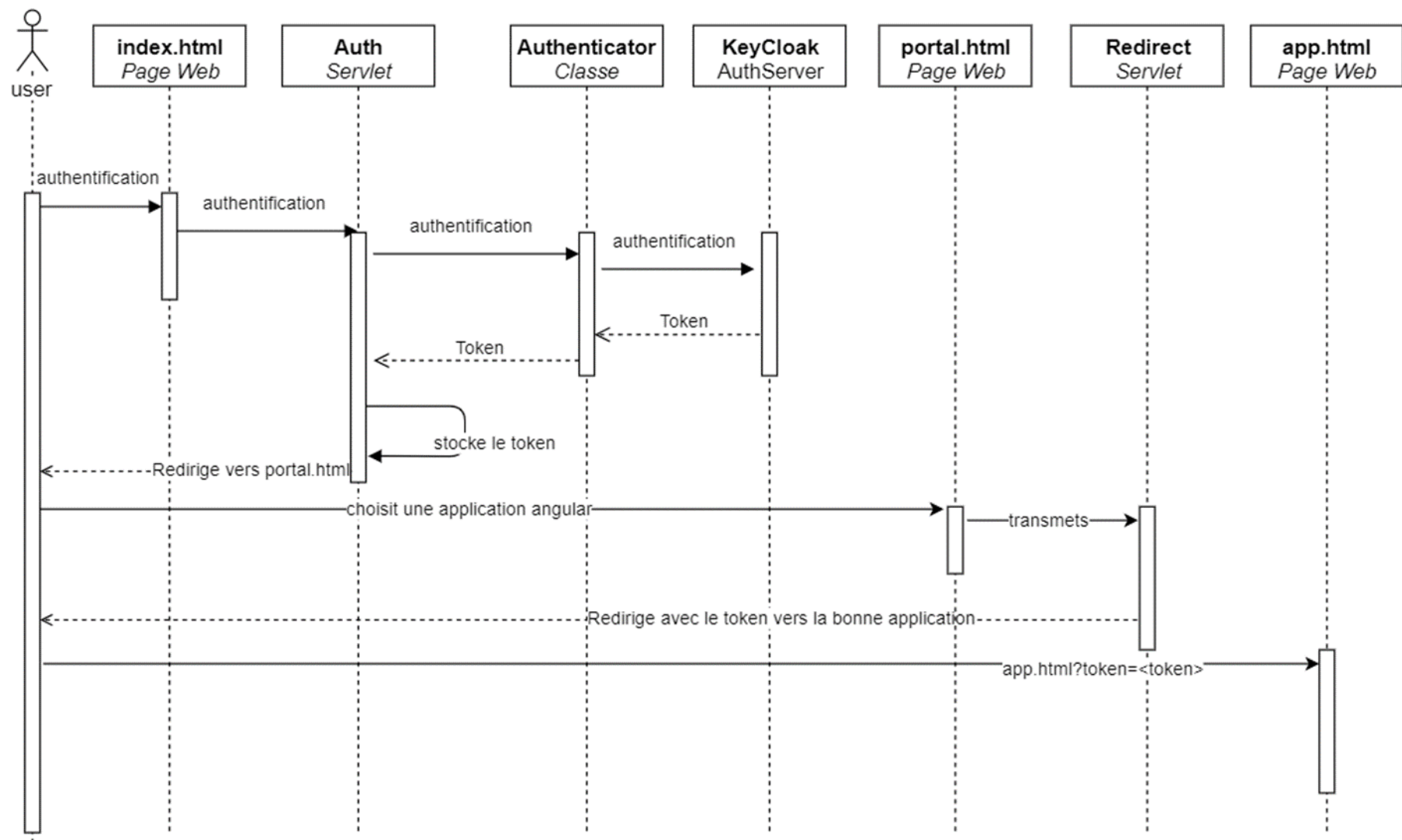


Fig. transmission du token vers une application « à token ».

Le code de cette application blanche est fourni par le responsable du portail.

Il est important pour le portail de « retenir » l'association du login et du token afin de pouvoir vérifier que l'utilisation d'une application nécessitant le token est bien autorisée pour le login considéré.

Cela peut être fait dans la session, ou plus généralement dans le contexte de servlet, si l'on veut pouvoir avoir des durées de session et de validité du token différentes.

Note – sur le passage du token à l'application Frontend

Dans le schéma, il est noté que le token est transmis à l'application cliente dans l'url. Cette solution mise en place ne constitue pas de « menace » (pour l'exposition de données sensibles) pour plusieurs raisons :

- La durée de vie du token est faible, et même si des personnes arrivaient à l'intercepter, ils ne pourraient probablement rien en faire.
- Ce token n'est visible dans l'url que lors du premier appel.

10. Infrastructure de production

L'infrastructure de production prévu pour février 2022 est la suivante :

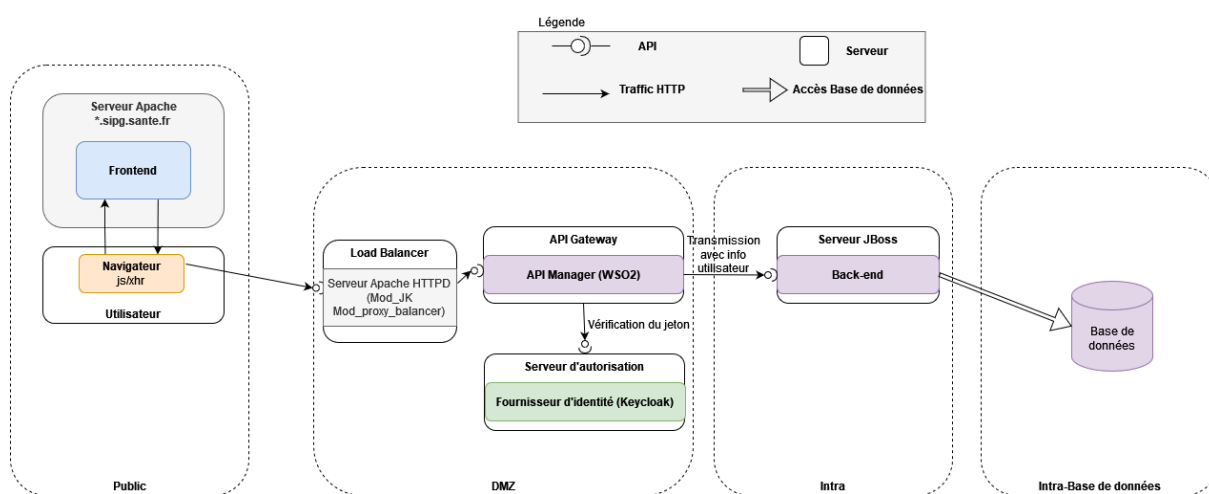


Fig. Infrastructure de Production

L'infrastructure de production ainsi défini repose sur plusieurs zones définies pour le réseau de l'Agence. Tout d'abord, la zone Public correspond au serveur Apache Frontaux avec un Load Balancer Actif/Passif. Les requêtes ainsi effectuées arrivent sur un serveur Apache servant de Load Balancer pour l'API Gateway (WSO2) et pour le fournisseur d'identité Keycloak. Les différents nœuds du cluster WSO2 et Keycloak sont en Haute disponibilité et en mode Actif/Actif.

Cette architecture clustérisée a démontré son efficacité lors de différentes études menées. Cette infrastructure permet la continuité de services et la répartition des charges de façon

active sur les différents nœuds de la structure. De plus, une session Keycloak n'est pas liée à un nœud Keycloak. Elle est indépendante du nœud par un système réplication de sessions.

La zone réseau « DMZ » communique avec une zone réseau « Intra-Base de données » afin de récupérer les vérification d'identité des utilisateurs à la connexion par le serveur d'autorisation. Mais aussi, d'accéder à la configuration de WSO2 et Keycloak sur une base de données PostgreSQL.

La zone réseau « DMZ » communique avec une zone réseau « Intra » correspondant à celle contenant les serveurs d'applications. Cette dernière correspond à l'ensemble des différents serveurs JBoss pour lesquels, les services sont déployés. De plus, la zone réseau « Intra » authentifie les demande d'accès aux backends WSO2 par Mutual TLS.

Enfin la zone réseau « Intra » échange avec la zone réseau « Intra-Base de données » pour les différentes données fonctionnelles demandées si ces dernières existent.

11. Annexe : Utilisation particulière d'une API

Il est parfois nécessaire d'exposer certaines ressources (*endpoints*) d'une API vers l'extérieur de façon moins sécurisée, sans protection par un *token* OAuth2/OIDC. Par exemple, si une API métier contient le *endpoint* « login » ou d'autres ressources qui doivent être accessibles dans le contexte d'un utilisateur non authentifié, alors il faut créer une autre API dite *publique* contenant uniquement les ressources que l'on souhaite exposer à des utilisateurs non authentifiés. Cette API *publique* contient alors un sous-ensemble de ressources de l'API initiale. Il faut tout de même protéger cette API *publique* avec une *api-key*.

Il est important de bien paramétrer l'API *publique* ainsi définie sur l'API-manager (WSO2), de telle sorte que le paramètre « *endpoint* » de l'APIM pointe, non pas sur la « racine » de l'API (le *context root* du war), mais de façon plus ciblée sur le groupe de ressources « publiques ».

Illustration avec WSO2

API initiale :

Production Endpoint → <https://<host>:<port>/myAPI-rest> , « myAPI-rest » étant le *context root* ou le « *runtime name* » (JBoss) du war de l'API.

API Definition → basePath = /myAPI-rest

Ce paramétrage sur l'APIM (WSO2) permet d'accéder à l'ensemble des ressources de l'API, avec les éléments de sécurité nécessaires (*token* OAuth2/OIDC, *api-key*).

API publique :

Production Endpoint → <https://<host>:<port>/myAPI-rest/public/fonctions-demo> .

API Definition → basePath = /myAPI-rest/public/fonctions-demo

Ce paramétrage sur WSO2 restreint les accès à un sous-ensemble des ressources de l'API, placées sous un *path* plus spécifique « /myAPI/public/fonctions-demo » (à l'instar d'un sous-répertoire). Il faut protéger les accès à ce sous-groupe de ressources avec une *api-key*, à moins que l'on ne souhaite pas du tout sécuriser cette API (openbar).

Cette solution permet de réutiliser l'apidoc initial sur l'APIM (WSO2), en faisant un minimum de modifications, sans avoir à enlever les ressources *non publiques* et les DTOs liés (opération fastidieuse, source d'erreurs). Surtout, le paramétrage WSO2 de cette API publique peut rester stable en dépit des évolutions de l'API initiale au niveau du *backend*, si le groupe de ressources publiques n'évolue pas.