Architecture API REST & HTTP

La présente section vise à présenter les conventions communes à la construction des API REST. Elle est issue d'un ensemble de bonnes pratiques communément appliquées et d'expériences accumulées sur la création des API REST.

Architecture REST

Manipulation des ressources

Lors de la conception des API, les règles suivantes s'appliquent :

- Les URL doivent être construites conformément aux règles & bonnes pratiques de l'architecture REST. Les identifiants des ressources doivent être passés en route param.
- Les ressources présentes dans les URLS seront **systématiquement** écrites au pluriel, même si une seule ressource est accessible.
  ➤ *Exemple :* `/contract/v1/contracts/123456789`
- Toutes les ressources d'un même service doivent **impérativement** partager un vocabulaire commun. Un champ représentant une donnée (exemple : Prix HT) doit disposer de la même "traduction", peu importe l'API utilisé au sein de ce service et / ou le modèle utilisé, tant que la donnée possède le même sens.

Convention de nommage des URLs

Les conventions de nommages s'appliquent principalement à la nomenclature des URLs accessibles et composant les services API.

- Utilisation de la convention de nommage Kebab Case.
- Utilisation de la langue **anglaise** pour le nommage des services, fonctions, attributs, ressources …
- Des **ressources** (et non des fonctions) doivent être utilisés dans les URLs (exemple `/contracts` et non `/getallcontracts`)
- Le nom des attributs composant une ressource devrait être différent des noms des champs de la base de données auxquels ils font référence.

Utilisation HTTP

Verbes HTTP

L'utilisation des verbes HTTP devra respecter la spécification ci-dessous, et, plus généralement, le sens de chaque méthode HTTP tel que décrit dans la section 4.3 de la RFC 7231 :

VERBE	DESCRIPTION
PUT	Modification totale d'une ressource
POST	Création une ressource
PATCH	Modification partielle d'une ressource
GET	Récupération d'une ressource
DELETE	Suppression d'une ressource

Tableau 3 : Utilisation des verbes HTTP pour la construction des APIs

Entêtes HTTP

Pour chaque réponse retournée, celle-ci doit inclure, à minima :

- La description du format de réponse : Ajout de l'entête `Content-Type`
- La définition de l'encodage utilisé : Ajout de l'entête `Charset`

Code statut HTTP

L'utilisation des codes de retour HTTP devra respecter la spécification suivante, et, plus généralement, le sens de chaque code de retour tel que décrit dans la section 6 de la RFC 7231 :

CODE	DESCRIPTION
2xx	Succès
200	Succès. Des informations de retour sont disponibles.
201	Succès. Une ressource a été créée. Généralement, la réponse contient la ressource qui vient d'être créée.
204	Succès. La réponse ne contient aucune donnée.
4xx	Échec à cause d'un problème dans la requête (exemple : création d'un utilisateur avec un e-mail déjà existant ou paramètre de requête manquant).
5xx	Échec dû à une erreur du serveur

Tableau 4 : Utilisation des codes de statut dans les réponses HTTP

Convention de création pour les API

Afin de maintenir une cohérence forte entre tous les services, certains besoins doivent utiliser une syntaxe commune décrite ci-dessous.

Règles communes

- Le résultat de la requête devra **toujours** être retourné dans le champ `data` d'un objet JSON. Les autres attributs peuvent servir à ajouter des métadonnées à la requête.

API Paginées

Les API paginées acceptent toujours deux paramètres optionnels :

- `page` - Numéro de la page à retourner (défaut : 1)
- `size` - Nombre de résultats par page à retourner (défaut : dépend de l'API, généralement 20)

Il doit être possible de manipuler ces paramètres pour obtenir les pages suivantes ou augmenter le nombre de résultat dans une seule page.

API de recherche

Dans le cas d'une API permettant d'effectuer une recherche (Recherche exclusive) sur une ressource :

- Il doit être possible, en spécifiant des valeurs en `query params`, de filtrer les résultats uniquement sur un critère précis de la ressource.
  ➠ *Exemple :* `users/?email=john.doe@contoso.com` - Liste des utilisateurs dont le nom est égal à la valeur indiquée.

- Les arguments de recherches de type string peuvent être préfixés/suffixés d'un `*` pour rendre la recherche non-limitative.
  ➠ *Exemple :* `users/?username=Sandbob*` - Tous les utilisateurs dont le nom commence par "Sandbob".

- Une logique similaire existe pour les champs de type date, avec les préfixes : `<` et `>`.
  ➠ *Exemple :* `users/?createdAt=>2020-01-15` - Tous les utilisateurs créés après le 15/01/2015.

Il est aussi possible de créer des APIs permettant de sélectionner **un ensemble** de ressources, correspondant aux différentes valeurs des éléments indiqués dans les `query params` de la requête HTTP (Recherche inclusive).

- Une syntaxe basée sur des crochets (`[ ]`) permet de spécifier la liste des différentes valeurs séparées par des virgules (,).
  ➠ *Exemple :* `contracts/?id=[1124521,1124550,2102450]` - Obtient une liste des contrats indiqués.

- Une syntaxe supplémentaire peut être implémentée, permettant une sélection sur un range de valeurs, en utilisant le séparateur `...`
  ➠ *Exemple :* `users/?id=[1…5]` (Les utilisateurs dont l'id est contenu entre 1 et 5).

Authenticated API

Certains services API doivent disposer de end-points **adaptant leur retour en fonction du contexte d'identité** véhiculé à travers le jeton d'authentification. Dans ce scénario, les API doivent répondre aux règles suivantes :

- L'url contient toujours, juste après le nom du service API et de sa version, le nom de l'identité utilisée.
  ➠ *Exemple :* `/user/contracts` - Les contrats de l'utilisateur xxx.
- Le nom de l'identité doit être **au singulier**.
- Une authentification de type `Authorization Code` ou `Resource Owner Password Credential` est requise.
- Une erreur `401` (`Unauthorized`) doit être levée si le jeton ne contient pas d'identité ou que celle-ci ne peut pas être vérifiée via le serveur d'autorisation.

Données manipulées par les services

Format d'échange

Les règles suivantes s'appliquent concernant les données des services API :

- Les services API doivent être conçus pour accepter des données d'entrée au format `application/json`.
- Les données retournées par les services API doivent être au format `application/json`.

Par ailleurs, les formats suivants doivent être toujours respectés (en entrée comme en sortie) :

TYPE DE DONNÉES	FORMAT ATTENDU	STOCKAGE BDD
Dates & heures	Date conforme à la RFC 3339 ➠ *Exemple* : 2005-08-15T15:52:01+01:00	`DATETIME`
Chaîne de caractères	Les `strings` doivent toujours être : - Débarrassées des espaces blancs inutiles (trim) - Utiliser le `null` si elles sont vides, sauf contrainte métier.	Variable (`VARCHAR`, `TEXT`…)
Nombre	Les nombres doivent être représentés sous la forme `integer` ou `float` et non de chaîne de caractères.	Variable (`INT`, `NUMERIC`…)
Booléen	Les booléens doivent être échangés sous leur forme originelle : `true` et `false`. L'utilisation du `0` ou `1` est proscrite.	`BIT`
Mot de passe	Les mots de passe doivent être hachés en utilisant l'algorithme `SHA256`	Variable (`VARCHAR`, `TEXT`…)

Tableau 5 : Format d'échanges des données au sein des APIs

Gestion des E/S

Contrôles d'intégrité des données

Pour chaque API, les données d'entrée / sortie doivent être contrôlées sur deux aspects :

- Présence ou non de la donnée.
- Respect du format attendu selon les contraintes de la base de données et selon les règles métiers.

Manipulation des données par les services API

Lors de la manipulation des données dans les services API (connexions aux bases de données …), les règles suivantes devraient être respecter :

- Chaque service est propriétaire de ses données. Il est le seul à pouvoir les consommer.
  ➡ *Quelques exceptions peuvent exister pour les données métiers transverses.*
- L'utilisation d'un ORM doit être privilégiée pour manipuler les données.
- Les transactions doivent être utilisées chaque fois que nécessaire.
- Les requêtes sollicitant régulièrement la base de données devraient utiliser des connexions persistantes.
- Les chaînes de connexion devraient spécifier selon le modèle l'option de lecture / écriture.

Gestion des erreurs

Normalisation de la sortie d'erreur (API Problem)

Toutes les erreurs API, qu'elles soient techniques ou fonctionnelles, devront être formatées selon les spécifications de la RFC 7807.

Ainsi, une erreur sera toujours conforme, à minima, au format suivant :

```
{
    "title": "string",
    "type": "string",
    "status": int
}
```

Où :

- `title` : contiendra le nom de l'erreur dans un format lisible par un être humain. (Généralement en anglais)
- `type` : contiendra l'identifiant du type de l'erreur. Deux erreurs ayant la même cause renverront **toujours** un type similaire. Le type peut être utilisé pour permettre d'adapter le retour à afficher à l'utilisateur.
- `status` : Contiendra le code HTTP de l'erreur. Sauf mention contraire, ce code **sera toujours** identique à celui de la réponse `HTTP`.

Par ailleurs, les règles suivantes s'appliquent :

- Les requêtes invalides se traduiront toujours par l'envoi d'une réponse `400 Bad Request` et d'un type d'erreur : `validation-error`.
- Les requêtes vers des ressources manquantes se traduiront toujours par l'envoi d'une réponse `404 Not Found` et d'un type d'erreur : `resource-not-found`.
- Si une erreur technique survient durant le traitement, la réponse sera toujours une `500 Internal Error` et le type d'erreur : `internal-error`.
    - Aucun détail de l'erreur ne devrait être visible en production.
    - L'erreur doit être loguée.

Stockage des erreurs (logs)

Les règles suivantes s'appliquent lorsqu'une erreur survient au sein d'un service API :

- Une solution de log centralisée (ex : `Graylog`) doit être utilisé pour inscrire tous les logs de l'application.
    - Cependant, une application peut disposer **en complément** (duplication) d'un mécanisme de log interne.
- Tous les services API doivent pouvoir activer un mode `debug` (via fichier d'environnement) permettant de loguer toutes les requêtes atteignant le service.
- Les données sensibles devront être anonymisées (exemple : Mot de passe).
- Toute erreur doit être identifiée et doit être traitée sous forme d'exception. Ces exceptions devront toutes être transmises au gestionnaire de log.
- Une requête en erreur doit être, **à minima**, être composée de logs permettant de retrouver :
- Une identification de la cause (détail de l'erreur, cause de l'erreur, fichier, n° de ligne …)
- Des informations sur la requête ayant générée l'erreur (Données d'entrée, fonction appelée etc…)
- Les logs d'erreurs doivent être conservés au moins **30 jours**.

Les codes erreurs utilisés pour identifier le niveau de criticité devront être conforme aux recommandations de la section 6.2 de la RFC Syslog Protocol :

CODE	GRAVITÉ	DESCRIPTION
0	Emergency	Système inutilisable.
1	Alert	Une intervention immédiate est nécessaire.
2	Critical	Erreur critique pour le système.
3	Error	Erreur de fonctionnement.
4	Warning	Avertissement (une erreur peut intervenir si aucune action n'est prise).
5	Notice	Événement normal méritant d'être signalé.
6	Informational	Pour information.
7	Debugging	Message de debug.

Tableau 6 : Liste des codes erreurs - Syslog Protocol

Version des API

Version d'une API

Chaque service API est **obligatoirement** préfixé d'un numéro de version. Un même service API peut exister sous plusieurs versions.

Un changement de version peut avoir lieu :

- Lors d'une modification du format d'E/S d'un ou plusieurs end-points constituant un service API.
- Lors de modification profonde des fonctionnalités associées à un ou plusieurs end-points d'un service API, et dont la modification peut avoir un impact sur les applicatifs consommateurs.

Toutes les autres mises à jour (amélioration de performance, correctif de sécurité …) d'un service API qui n'ont pas d'impact sur les éléments décrit ci-dessus doivent être appliquées de manière transparente sur l'API sans changement de version.

Tenue d'un CHANGELOG

La mise en place d'un **manifeste d'historique de version est obligatoire**.

- Utilisation du format Keep a Changelog & utilisation du Semantic Versioning.
- Ce manifeste doit être présent à la racine du dépôt **Git** sous la forme d'un fichier CHANGELOG.md (Fichier reconnu par **Gitlab**).

Documentation API

Pour chaque service, il est nécessaire de fournir :

Une spécification complète de l'API au format Open API 3.0 (Swagger), qui doit inclure à minima :

- Pour chaque `end-point` :
    - Une description claire de son rôle. Cette description doit indiquer si des filtrages implicites sont appliqués sur la ressource retournée.
    - Une description des paramètres de recherches s'ils existent.
    - La liste complète des réponses (erreurs / succès) qui peuvent être retournées par la fonction.
    - Pour chaque réponse, un exemple de valeurs retournées et/ou le `model` associé.
- Pour chaque `model` :
    - Une description générale du model.
    - Une description de la signification de chaque champ. Cette description est **obligatoire** même si le nom du champ semble être suffisamment explicite.

Si nécessaire, une documentation contextuelle supplémentaire (par exemple : Liste des règles métiers spécifiques à cette fonction) sera fournie et accessible au même endroit que la documentation **Swagger**.