**GUIDE DE RECOMMANDATIONS**

**- GitmeMoney -**

****

**\_**

**David EVAN**

**06/09/2022**

**Version 1.0**

**Crypto Plateforme – GitMeMoney**

**Historique des révisions**

|  |  |  |  |
| --- | --- | --- | --- |
| **Numéro de version** | **Auteur** | **Description** | **Date de modification** |
| 1.0 | EVAN David  *(Architecte logiciel)* | Livraison initiale | 06/09/2022 |
|  |  |  |  |
|  |  |  |  |

Tableau 1 - Historique des révisions

**Objectif du document**

Dans le cadre du projet « Crypto-plateforme » initié par GitmeMoney, ce guide de recommandation a pour objectif de fournir des recommandations à destination des développeurs visant à améliorer les pratiques de développement logiciel et la qualité de code.

Il rappelle les standards de développement à respecter, les méthodes de travail à appliquer et les réunions associées à la méthodologie de travail.

**Note :**

Ce document vise à présenter de manière synthétique les recommandations en matière de développement logiciel et de méthodologie de gestion à appliquer pour ce projet. Il ne s’agit pas d’une documentation sur les concepts présentés. Des références seront indiquées pour l’approfondissement des concepts.

**Table des matières**

[ORGANISATION DU PROJET 4](#_Toc113370532)

[Méthodologie « SCRUM » 4](#_Toc113370533)

[Les cérémonies 4](#_Toc113370534)

[ORGANISATION DES DÉVELOPPEMENTS 5](#_Toc113370535)

[Méthodes de travail 5](#_Toc113370536)

[Peer-programming 5](#_Toc113370537)

[Revue du code par les paires 5](#_Toc113370538)

[Tests unitaires 5](#_Toc113370539)

[STANDARDS DE DÉVELOPPEMENTS 6](#_Toc113370540)

[Introduction 6](#_Toc113370541)

[Conventions de code à respecter 6](#_Toc113370542)

[Frontend (HMTL / JavaScript / Angular …) 6](#_Toc113370543)

[Backend (Java / Spring) 6](#_Toc113370544)

[Architecture API REST & HTTP 7](#_Toc113370545)

[Architecture REST 7](#_Toc113370546)

[Utilisation HTTP 8](#_Toc113370547)

[Convention de création pour les API 9](#_Toc113370548)

[Données manipulées par les services 10](#_Toc113370549)

[Format d'échange 10](#_Toc113370550)

[Gestion des E/S 11](#_Toc113370551)

[Gestion des erreurs 11](#_Toc113370552)

[Normalisation de la sortie d’erreur (API Problem) 11](#_Toc113370553)

[Stockage des erreurs (logs) 12](#_Toc113370554)

[Version des API 13](#_Toc113370555)

[Version d’une API 13](#_Toc113370556)

[Tenue d’un CHANGELOG 13](#_Toc113370557)

[Documentation API 14](#_Toc113370558)

[TABLES DES RÉFÉRENCES 15](#_Toc113370559)

[Figures 15](#_Toc113370560)

[Tableaux 15](#_Toc113370561)

# ORGANISATION DU PROJET

## Méthodologie « SCRUM »

Rappel de la méthodologie (en synthèse) et de l’outillage utilisé (JIRA, scrum board)

## Les cérémonies

# ORGANISATION DES DÉVELOPPEMENTS

## Méthodes de travail

### Peer-programming

### Revue du code par les paires

### Tests unitaires

# STANDARDS DE DÉVELOPPEMENTS

## Introduction

Cette section vise à présenter un certain nombre de bonnes pratiques pour harmoniser l’écriture de code. Dans un esprit d’amélioration continue, ces recommandations peuvent être amenées à évoluer en fonction des pratiques et des usages.

## Conventions de code à respecter

### Frontend (HMTL / JavaScript / Angular …)

Le développement frontend en langage Javascript devra respecter les conventions décrites dans le guide de style et bonnes pratiques Angular :

* [Angular coding style guide](https://angular.io/guide/styleguide)

Notons que tout code HTML produit devra respecter les normes proposées par le W3C, notamment en matière d’accessibilité :

* [Validation HTML 5](https://validator.w3.org/)
* [WCAG3 : Web Content Accessibility Guidelines](https://www.w3.org/TR/WCAG22/)

### Backend (Java / Spring)

Le développement backend en langage Java s’appuiera sur les conventions de code standards à Java (habituellement maitrisés par tout développeur) et le guide de style Google pour Java pour la présentation :

* [Java Code Conventions](https://www.oracle.com/technetwork/java/codeconventions-150003.pdf)
* [Google Java Style Guide](https://google.github.io/styleguide/javaguide.html)

## 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’expérience accumulés 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](https://medium.com/better-programming/string-case-styles-camel-pascal-snake-and-kebab-case-981407998841).
* 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 différents des noms des champs de la base de données auxquelles 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](https://tools.ietf.org/html/rfc7231#section-4.3) :

|  |  |
| --- | --- |
| **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 2 : 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](https://tools.ietf.org/html/rfc7231#section-6) :

|  |  |
| --- | --- |
| **CODE** | **DESCRIPTION** |
| **2xx** | Succès |
| **200** | Succès. Des informations de retour sont disponibles. |
| **201** | Succès. Une ressource a été créer. 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 3 : 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 conçu 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](https://tools.ietf.org/html/rfc3339)  *Exemple*: 2005-08-15T15:52:01+01:00 | DATETIME |
| Chaine 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 chaine 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 4 : 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.*

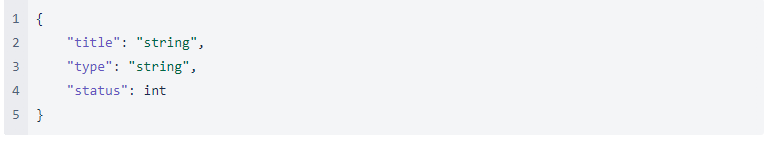
* L’utilisation d’un ORM doit être privilégiée pour manipuler les données.
* Les [transactions](https://fr.wikipedia.org/wiki/Transaction_informatique) doivent être utilisées chaque fois que nécessaire.
* Les requêtes sollicitant régulièrement la base de données devraient utiliser des connexions persistantes.
* 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](https://tools.ietf.org/html/rfc7807).

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



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 loggée.

## 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 logger 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 identifiés le niveau de criticité devront être conforme aux recommandations [de la section 6.2 de la RFC Syslog Protocol](https://www.rfc-editor.org/rfc/rfc5424.html#section-6.2.1) :

|  |  |  |
| --- | --- | --- |
| **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 5 : 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](https://keepachangelog.com/en/1.1.0/) & utilisation du [Semantic Versioning](https://semver.org/spec/v2.0.0.html).
* Ce manifeste doit être présent à la racine du dépôt **Git** sous la forme d’un fichier [CHANGELOG.md](http://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](https://swagger.io/docs/specification/about/) (Swagger), qui doit inclure à minima :

* Pour chaque end-point :
  + Une description claire se 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**.

# TABLES DES RÉFÉRENCES

## Figures

**Aucune entrée de table d'illustration n'a été trouvée.**

## Tableaux

[Tableau 1 - Historique des révisions 2](#_Toc113370413)

[Tableau 2 : Utilisation des verbes HTTP pour la construction des APIs 9](#_Toc113370414)

[Tableau 3 : Utilisation des codes de statut dans les réponses HTTP 9](#_Toc113370415)

[Tableau 4 : Format d'échanges des données au sein des APIs 11](#_Toc113370416)

[Tableau 5 : Liste des codes erreurs - Syslog Protocol 14](#_Toc113370417)