

Description du package

Kiamo

EchantillonS de connecteur CRM

Date : 20190510

Version : v1.2.0

Author : S. Iniesta



Please make sure you have the right documentation version for your software version. Nevertheless, depending on the Kiamo version used, the screen shots in this document may be different from those of your interface.

The full documentation is available on [mykiamo.fr](http://mykiamo.fr/).

**Intellectual property and copyright**

Version documentation : 1.2 (6)

Date de dernière mise à jour : 10/05/2019

Use of this document is covered by French and international legislation on copyright and intellectual property. No part of this document may be reproduced, stored or transmitted for any purpose by any means whatsoever without the express written permission of Conecteo. Unless the license agreement of Conecteo expressly stipulates otherwise, provision of this document does not give you any license to these patents, trademarks, copyrights or other intellectual property rights.

Kiamo and Conecteo are registered trademarks of Conecteo.

© Conecteo 2019. All rights Reserved. IDDN.FR.001.300028.011.S.P.2010.000.20100

TABLE DES MATIERES

[1. Introduction 5](#_Toc8399203)

[1.1 Objectifs 5](#_Toc8399204)

[1.2 Description 5](#_Toc8399205)

[2. DESIGN 7](#_Toc8399206)

[3. CONNECTEUR 9](#_Toc8399207)

[3.1 Connecteur Kiamo (module) 9](#_Toc8399208)

[3.1.1 Création 9](#_Toc8399209)

[3.1.2 Configuration 10](#_Toc8399210)

[3.1.3 Logs du module 13](#_Toc8399211)

[3.1.4 Ressources du module 13](#_Toc8399212)

[3.1.5 Utilisation 13](#_Toc8399213)

[3.2 Interaction Manager (sous module) 15](#_Toc8399214)

[3.3 Entities Manager (sous module) 16](#_Toc8399215)

[3.4 Customization Manager (sous module) 17](#_Toc8399216)

[4. FACILITATEURS PRINCIPAUX 18](#_Toc8399217)

[4.1 Module 18](#_Toc8399218)

[4.2 ConfManager 19](#_Toc8399219)

[4.3 Logger 19](#_Toc8399220)

[4.4 SubModule 21](#_Toc8399221)

[5. Outils 22](#_Toc8399222)

[5.1 datetimes 22](#_Toc8399223)

[5.2 dict 22](#_Toc8399224)

[5.3 files 23](#_Toc8399225)

[5.4 resources 23](#_Toc8399226)

[5.5 strings 23](#_Toc8399227)

[5.6 userdata 24](#_Toc8399228)

[5.7 uuids 24](#_Toc8399229)

[5.8 webs 24](#_Toc8399230)

[6. Outils Connexes 25](#_Toc8399231)

[6.1 Testeur en ligne de commande 25](#_Toc8399232)

[6.2 Vérificateur de connecteur 25](#_Toc8399233)

[6.3 Exemple de script SVI 26](#_Toc8399234)

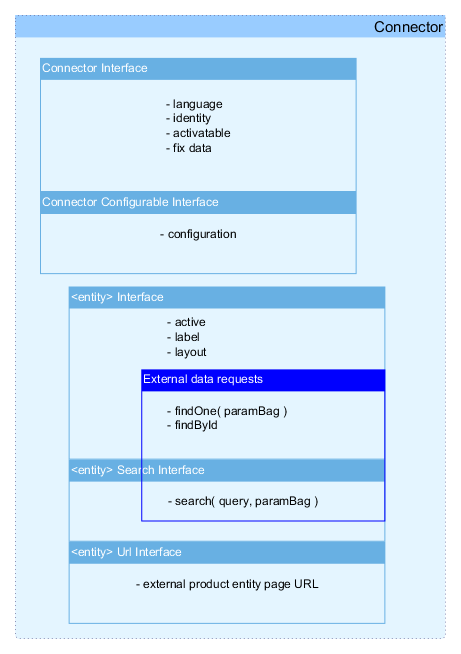
# Introduction

## Objectifs

Le but de ce package est de fournir des échantillons de connecteur CRM à destination d’une équipe de développement externe, sur la base d’une architecture épurée, avec un outillage intégré et extensible, et un certain nombre de facilitateurs. Chaque échantillon pourra tout autant servir de simple illustration, que de socle à une implémentation spécifique.

## Description

Un connecteur CRM Kiamo se présente synthétiquement comme le montre le schéma suivant :



Il est essentiellement dédié à la récupération d’entrées externes correspondant à des données de recherches reçues par Kiamo.

Le dossier d’un échantillon de connecteur CRM se présente comme une petite arborescence :

* <racine> : le module principal (implémentation du connecteur CRM), et ses sous-modules,
  + - * l’interaction manager, qui porte la responsabilité d’établir la session et les communications avec le CRM externe,
      * l’entities manager, qui porte la responsabilité des mappings nécessaires entre les données provenant du CRM et leur représentation dans Kiamo,
      * le customization manager, qui portera les implémentations spécifiques (traitements des données comme le formattage des numéros, le tri de listes de tickets, etc.)
* conf : le dossier contenant les fichiers de configuration,
* data : le dossier dédié aux fichiers de ressource,
* docs : le dossier contenant les docs,
* logs : le dossier où seront écrits les logs du connecteur,
* tools : le dossier contenant l’implémentation des outils fournis dans le package.

Chaque échantillon pointe un CRM externe particulier.

La documentation de référence pour l’implémentation d’un connecteur reste « Développement d’un connecteur CRM - ERP », de Kiamo.

Le répertoire où doivent être déployés les connecteurs est :

<Kiamo Folder>/data/userfiles/class/Connectors/<ConnectorName>

Le code est auto-descriptif et la structure assez simple pour être abordée à travers un survol rapide. Il sera intéressant de se pencher sur la documentation pour saisir l’esprit et les détails de l’implémentation, pour une meilleure appropriation.

Note : dans ce document ainsi que dans le code des échantillons, on évoquera les notions :

* d’ entity , une entité représentant un type d’item. Par exemple un contact, une société, un ticket.
* d’ entry , une entrée représentant une instance d’entité. Par exemple, la fiche contact de Mr Foo Bar.

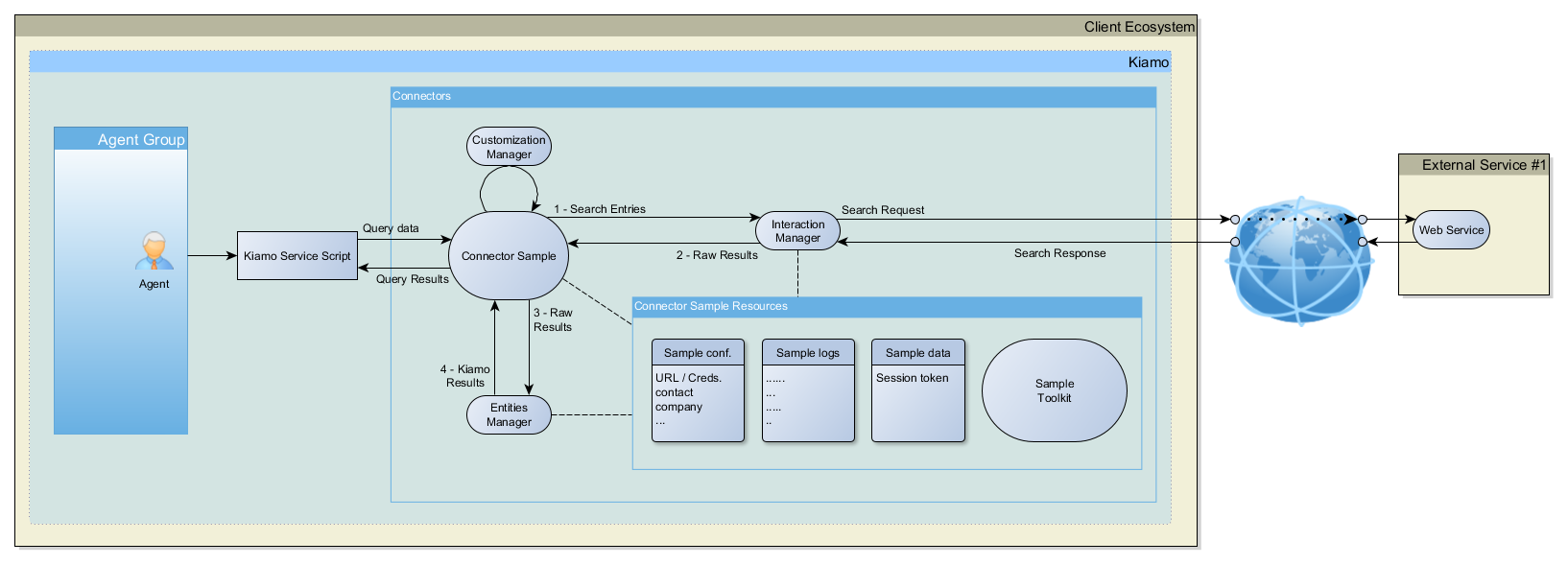
Ces deux notions sont différentes de celles présentes dans le nom des fonctions d’un connecteur Kiamo.

Dans la version actuelle (mai 2019), 3 échantillons de connecteurs CRM sont fournis :

* un connecteur générique ***MS Dynamics 365*** *(API v9.0, 2018)*,
* un connecteur générique ***Salesforce Lightning*** *(API v41.0, winter 2018)*,
* un connecteur générique ***Edeal*** *(API 2015)*.

# DESIGN

L’architecture type d’un échantillon de connecteur se présente de la manière suivante :



Très synthétiquement, le principe est le suivant :

* le connecteur Kiamo reçoit les critères de recherche de Kiamo, et lui retournera les résultats trouvés,
* il fait appel à un sous module Interaction Manager, qui porte la responsabilité de convertir les recherches en requêtes vers le service externe,
* les résultats sont confiés au Entities Manager, qui porte la responsabilité de convertir les résultats bruts du service externe au format requis par Kiamo,
* durant ce traitement, on fait appel au Customization Manager pour tout traitement spécifique : formattage et tri des données entrantes, tri des résultats, …
* L’ensemble se base sur un set de ressources :
  + une configuration, contenant toutes les données nécessaires à la connexion au service externe, et au mapping entre les données de ce service et les données Kiamo,
  + des logs,
  + des ressources physiques,
  + un toolkit (logger, confManager, webRequester, …).

Plus en détail, le design de ces échantillons se découpe en 3 parties principales :

* le connecteur CRM Kiamo :

Il s’agit de la classe principale.

* + Elle définit principalement le point pivot entre Kiamo et le CRM externe, en particulier le type de données que le connecteur pourra remonter et les fonctionnalités du connecteurs.
  + Au sein du package, cette classe est essentiellement une coque vide qui utilise les fonctionnalités de ses sous modules pour effectuer les tâches de :
    - récupération des données du CRM externe (session, accès),
    - mapping des données,
    - appel des fonctions spéciales (tri, pré-traitements, éligibilité, …).

Le pilotage de la recherche d’entités correspondant aux inputs passés (comme un numéro de téléphone entrant ou un id) est défini dans la configuration.

Les quelques lignes de code effectives de ce module portent essentiellement les appels aux fonctions de recherche et de mapping des données, la sélection d’une entrée parmi une liste retournée, et l’appel aux fonctions de customisation spécifiques.

* l’architecture de l’échantillon est constituée :
  + de la classe connecteur CRM Kiamo, à la racine (voir ci-dessus).

Cette classe porte d’une part l’implémentation classique d’un connecteur CRM Kiamo, d’autre part des facilitateurs du package :

* + - un gestionnaire et facilitateur d’accès à la configuration,
    - un gestionnaire de logs.

Les instances de ces facilitateurs seront portées par ce module, et propagées à ses sous-modules.

* + de ses sous-modules :
    - l’interaction manager : responsable des échanges avec le CRM externe, il établit la session, construit les requêtes vers ce dernier et renvoie les données brutes.

Une des finesses de ce sous module est de gérer la persistance du token de session de manière à éviter des appels systématiques au CRM externe lorsqu’il est invoqué par Kiamo.

* + - l’entities manager : il connait le format brut des données renvoyées par l’interaction manager, et le format nécessaire à Kiamo.

Sur la base de la conf, il effectue les vérifications des retours et le mapping entre ces données brutes et le format nécessaire à Kiamo (EntityLayout, EntityInstance, EntitiesCollections).

* + - le customization manager : c’est une classe dédiée à porter les méthodes client spécifiques, ainsi que les méthodes par défaut facilitant le traitement des entrées et des sorties.

Ainsi, on y trouvera des méthodes d’éligibilité de données (par exemple, un chaine de caractères de recherche agent comportant des lettres ne pourra pas être un numéro de téléphone ; il est donc inutile de solliciter le CRM externe avec des requêtes portant sur ce type de donnée – fixe, mobile, pro - non éligibles), des pré-traitements (élimination des caractères inutiles d’un numéro de téléphone entrant), des post-traitements (choix du ticket client le plus pertinent dans la liste retournée par le CRM), etc.

Ces sous-modules bénéficient des facilitateurs du module parent, le conf manager et le logger.

* + de l’arborescence du package :
    - répertoire conf : regroupe les fichiers de configuration, en particulier des logs et du connecteur.
    - répertoire data : ressources du module. Dans le cas présent, il abrite un petit fichier de persistance contenant le dernier token de session valide.
    - répertoire docs : contient la documentation du package.
    - répertoire logs : logs du package.
    - répertoire tools : l’outillage du package (voir ci-dessous).
* les facilitateurs :

Leur implémentation est située dans le répertoire tools du package. Il y a trois niveaux de facilitateurs :

* + module et submodule : fournissent des capacités d’accès à la configuration et de logs. Les modules et sous modules étendent ces classes.
  + autoload, confManager, logger : outils principaux liés aux modules et aux sous modules.
  + Tous les autres outils : fournissent diverses capacité d’accès et de traitement des données.

Il est naturellement possible, voire encouragé, de modifier et d’étendre cet outillage au gré des besoins de développement.

# CONNECTEUR

Un connecteur CRM Kiamo peut être implémenté dans une seule classe.

Dans le cadre de cet échantillon, l’implémentation se fait à travers l’implémentation d’un module principal, le connecteur lui-même, et de trois sous modules aux responsabilités bien établies :

* Le connecteur :
  + implémente les méthodes requises par la documentation de développement d’un connecteur,
  + est piloté par sa configuration pour faire appel au CRM externe et retrouver les items correspondant aux inputs Kiamo. Il implémente donc la lecture et l’exécution de la configuration.
* Les sous modules :
  + L’Interaction Manager :
    - Implémente les méthodes d’authentification au CRM externe, le maintient d’une session valide et l’accès aux ressources de ce CRM.
    - Il renvoie les résultats sous forme brute, tels que le CRM les lui a renvoyé.
  + L’Entities Manager :
    - effectue toutes les « traductions » et les mapping entre les données provenant du CRM externe et les entités Kiamo, sur la base de la configuration.
  + Le Customization Manager :
    - regroupe toutes les fonctions d’éligibilité, de modification et de tri des données, en particulier les traitements spécifiques au besoin client.

## Connecteur Kiamo (module)

### Création

La création d’un connecteur CRM Kiamo doit suivre les prérequis décrits par la documentation de développement d’un connecteur CRM – ERP de Kiamo.

De plus, pour bénéficier des capacités et facilitateurs du package, il doit :

* inclure l’autoloader,
* effectuer un use de chaque outil utilisé,
* étendre la classe Module,
* initialiser ses SubModule (interaction, entities et customization managers).

«

namespace <ConnectorNamespace> ;

require\_once \_\_DIR\_\_ . DIRECTORY\_SEPARATOR . "tools" . DIRECTORY\_SEPARATOR . "autoload.php" ;

use <ConnectorNamespace>\Module ;

(...)

class <ConnectorName> **extends Module** implements KiamoConnectorInterface, (...)

{

(...)

public function \_\_construct()

{

parent::\_\_construct() ;

$this->interactionMgr = new InteractionManager( $this ) ;

$this->entitiesMgr = new EntitiesManager( $this ) ;

$this->customizationMgr = new CustomizationManager( $this ) ;

}

(...)

}

»

L’autoloader permet de se passer du require d’inclusion des fichiers du toolkit. Cependant, un use est, sinon obligatoire, au moins recommandé pour éviter d’éventuelles collisions.

Le Module est, en arrière-plan, initialisé avec le path racine, et le nom du module qui est, par défaut, le nom de la classe.

Sur cette base :

* le path <racine>, socle de toute l’implémentation du module, sera fixé,
* le nom du module permettra :
  + l’utilisation (éventuelle) d’un fichier de config dédié (<racine>/conf/u\_<ConnectorName>.php), à condition que ce dernier soit déclaré dans le fichier racine de la configuration <racine>/conf/\_config.php.
  + la création d’un folder <racine>/logs/<ConnectorName>/ dédié, où tous les logs du module seront gérés.
  + L’utilisation simplifiée de ressources dans le répertoire <racine>/data/<ConnectorName>/, à condition que ce dernier soit créé et alimenté par les ressources souhaitées.

### Configuration

La configuration du Module se situe dans le répertoire conf. Elle est constituée, essentiellement :

* du fichier racine de la conf \_config.php : le Configuration Manager se base sur ce fichier pour connaitre les éléments de configuration à prendre en compte, et le chemin pour y accéder.
* du fichier configuration du tool Logger t\_logger.php : il définit le niveau et les détails des logs, et le comportement du Logger vis-à-vis de ses fichiers anciens.
* du fichier configuration utilisateur du connecteur u\_<ConnectorName>.php : ce fichier pilote intégralement l’accès au CRM externe, la forme que devront prendre les entrées retournées et la manière de les rechercher et de les traiter.

Le fichier racine de la configuration est donc un fichier php qui contient :

<?php

return [

'tools' => [

'key' => 't',

'items' => [

'logger',

],

],

'user' => [

'key' => 'u',

'items' => [

'<ConnectorName>',

],

],

] ;

?>

Il est chargé par le ConfManager, et décrit les configurations déclarées (et donc accessibles), ici un tool Logger et un user module <ConnectorName>.

Si l’on souhaite utiliser un fichier de conf pour un module ou un sous-module donné, il faut le déclarer dans ce fichier.

Voir « **4.2 ConfManager** » pour les détails.

Le fichier de configuration du connecteur lui-même se présente sous la forme suivante :

* bloc self : nom du service externe et version du connecteur
* bloc protocol : version de l’API de Web Services permettant l’accès au CRM externe
* bloc environments : liste des environnement CRM (test, preprod, prod, …). Pour chaque environnement :
  + bloc accessdata : toute la configuration nécessaire à l’accès et à l’établissement d’une session : url, clés, jetons de sécurité, …
  + bloc urls : toutes les données nécessaire à la construction des urls permettant l’accès aux ressources du CRM : page de recherche pour une entité donnée (contact, société, …), fiche d’une entrée, etc.
* bloc entities : contient tout le mapping entre les entités du CRM externe et Kiamo, ainsi que la logique de recherche d’entrées correspondant aux inputs dans Kiamo.
  + bloc map : table de correspondance <entiteCRM> => <entiteKiamo> (ex : ‘Account’ => ‘company’).
  + bloc <kiamoEntityType> : description complète du mapping et des logiques de recherche :
    - bloc labels : toutes les données nécessaires à la construction des labels de l’entité et de ses entrées.

Dans le cas du label d’une entrée, il s’agit d’une concaténation de chaines de caractères (type string) et de valeurs de champs de l’entrée (type field, valeur key interne du champs), séparées par le separator.

Attention : si le script Kiamo comporte plusieurs blocs connecteur s’enchaînant sur des entités différentes, ce sera le label du dernier de ces blocs qui déterminera le label de l’interaction.

* + - bloc map : liste de tous les champs formant une entrée, qu’ils soient affiché dans Kiwi ou simplement nécessaires à l’assemblage des données de cet affichage.

Exemple : dans un contact, il n’est pas nécessaire d’afficher l’Id CRM Externe du la société du contact ; mais cet Id est nécessaire à la récupération de cette société, si celle-ci doit aussi être affichée dans Kiwi). Il faut donc la récupérer lors de la récupération des données du contact.

Chaque ligne d’un bloc map correspond à un champs de l’entrée dans le CRM externe. Elle se présente sous la forme :

<crmFieldName> => [ key, label, type, display, map ], où :

* + - * <crmFieldName> est le nom du champs de l’entrée dans la base CRM,
      * key est le nom de la clé interne, qui sera utilisée par l’échantillon pour décrire ce champs. key peut être une chaine vide, si ce champs n’est pas amené à être manipulé régulièrement par l’implémentation. Les clés internes classiquement utilisées sont id, firstname, lastname, phone, mobile, email, contactId, companyId, ticketId, status, … Le entity manager fournit une méthode facilitant l’accès au champs d’une entrée via cette clé interne ; cela permet d’abstraire les noms des champs externes, variables. Par exemple, la récupération de l’id de toute entrée se fera toujours par la clé ‘id’, quel que soit le nom du champs de l’entité externe correspondante.
      * label : label du champs à afficher dans Kiwi, si ce champs est affiché.
      * type : type de la donnée. Ce type interne sera mappé avec les types Kiamo, influençant l’affichage et les fonctionnalités additionnelles.

Par exemple, un type interne phone sera mappé vers le type Kiamo EntityField::TYPE\_PHONE, qui rendra le champs clickable et pourra déclencher un appel depuis Kiwi vers ce numéro.

Les types internes principaux sont id, string, phone, email, date, datetime, time, birthday, text.

* + - * display : booléen signalant si ce champs doit être affiché dans Kiwi, ou pas.
      * map : nom de la variable Kiamo dans laquelle doit être mappée la valeur du champs. Ce nom peut bien entendu rester vide.
    - bloc search : ce bloc décrit la logique de recherche d’entrées correspondant aux inputs de Kiamo. Dans l’échantillon, l’implémentation de ces logiques est présente dans le connecteur CRM Kiamo. Il y a deux types de recherche, et une méthode additionnelle :
      * recherche kiamoInput : il s’agit d’une recherche lancée à partir du ParameterBag de Kiamo. Ce sac de paramètres est rempli par les inputs ayant déclenché l’interaction, puis l’appel au connecteur à travers le script voix, mail, … où est positionné le bloc connecteur. Il peut être enrichi tout au long de la traversée du script.

Typiquement, le ParameterBag contiendra initialement un numéro de téléphone entrant, dans le cadre d’un appel client.

Ce bloc contient un certain nombre de lignes. Chaque ligne sera traitée dans l’ordre, de la première à la dernière, excluant les suivantes en cas de résultat positif, et correspondra à l’étude du contenu d’une variable du ParameterBag, et d’un appel à un Web Service du CRM externe pour vérifier si des entrées correspondent à ce paramètre.

Exemple :

Une première ligne pourrait vérifier si le CustNumber du ParameterBag contient un numéro de téléphone, et appelle le Web Service externe pour vérifier si une entrée du CRM contient un champs « téléphone fixe » correspondant à ce numéro.

Une deuxième ligne pourrait vérifier la correspondance entre ce même paramètre et le champs « téléphone mobile ».

Enfin une troisième ligne effectuerait une vérification comparable sur la base du EMailSender du ParameterBag.

Chaque ligne contient une array composée des champs suivants :

* + - * + varName : nom de la variable du ParameterBag
        + entityField : clé interne d’un champs de l’entité, correspondant à un champs externe.
        + operation : clé / nom de l’opération de recherche dans le CRM externe. Par exemple like ou equals, pour des requêtes typées SQL.
        + preTrt : pré-traitement à effectuer sur la valeur de la variable passée brute.

Par exemple, si l’on sait que les numéros entrant sont mal formattés mais qu’ils sont tous représentés de la même manière dans le CRM externe, on passera une fonction de pré-traitement sur les inputs bruts pour ajuster leur format à une recherche correcte dans le CRM externe.

Ce champs peut rester vide si aucun pré-traitement n’est nécessaire.

Si le champs n’est pas vide, il doit correspondre au nom d’une méthode de pré-traitement implémentée dans le Customization Manager.

* + - * + eligibility : méthode appelée sur la donnée brute éventuellement pré-traitée, pour déterminer si l’input est éligible ou non à un type de recherche donné, et ainsi éviter des appels inutiles vers le CRM externe.

Par exemple, il est inutile d’effectuer une recherche via un numéro de téléphone si la chaine de caractères passée contient des lettres.

Ce champs peut rester vide si aucune vérification d’éligibilité n’est nécessaire.

Si le champs n’est pas vide, il doit correspondre au nom d’une méthode d’éligibilité implémentée dans le Customization Manager.

* + - * + postTrt : post-traitement à effectuer sur le résultat de la recherche, dans le cadre d’une recherche sur ce champs précis.

Par exemple, si on a cherché une correspondance partielle sur un numéro de téléphone dont on ne connaissait qu’une partie, et que plusieurs résultats sont retournés, il peut être intéressant de déterminer la meilleure correspondance via l’exécution d’un post-traitement dédié.

Ce champs peut rester vide si aucun post-traitement n’est nécessaire.

Si le champs n’est pas vide, il doit correspondre au nom d’une méthode de post-traitement implémentée dans le Customization Manager.

* + - * recherche agentQuery :

Il s’agit exactement de la même logique que le bloc kiamoInput précédent, sauf :

* + - * + que cette logique sera appliquée aux recherches manuelles des agents sur des entités données (bouton « Rechercher » de Kiwi), par chaine de caractère dont on ne connait donc, par nature, pas d’avance le type, et qui peut être partielle.
        + qu’il n’y a pas de champs « varName », vu que la variable est toujours la chaine de recherche manuelle.

Pour le reste, tout fonctionne à l’identique des recherches de type kiamoInput.

* + - * fonction additionnelle getOneOfList : il s’agit de la fonction par défaut, si une recherche (de tout type) a renvoyé une liste d’éléments et que Kiamo n’en attend qu’un seul.

Par défaut, si cette chaine reste vide, c’est simplement le premier élément de la liste qui sera renvoyé.

Si la chaine n’est pas vide, elle devra correspondre au nom d’une méthode de filtre sur une liste d’entrées correspondantes, implémentée par le sous-module Customization Manager.

Cette méthode sera appliquée après tout post-traitement effectué sur le résultat d’une recherche, le cas échéant.

### Logs du module

Les logs du modules seront écrits dans le folder <racine>/logs/<ConnectorName>/ .

La configuration de l’ensemble des logs du package et des modules utilisateurs se trouve dans le fichier <racine>/conf/t\_logger.php. La configuration permet l’effacement ou le zip de logs anciens (plus vieux qu’un certain nombre de jours).

Voir « **4.3 Logger** » pour les détails.

### Ressources du module

On peut choisir de placer les ressources du modules dans le folder <racine>/data/<ConnectorName>/ , pour bénéficier de l’outil helper de ressources.

Voir « **4.4 resources** » pour les détails.

### Utilisation

Le fait que le module de fonctionnalités soit de type Module, lui ouvre automatiquement les capacités du Logger et du ConfManager :

* Logger :
  + Logs :

Depuis n’importe quel emplacement d’un Module ou d’un SubModule, il est possible d’invoquer le Logger.

$this->log( "---------", Logger::LOG\_INFO, \_\_METHOD\_\_ ) ;

$this->log( "INIT : OK", Logger::LOG\_INFO, \_\_METHOD\_\_ ) ;

Les lignes de logs sont écrites dans le fichier <racine>/logs/<ConnectorName>/<YYYYMMDD>.log, sous la forme :

[20190429\_102557][INFO ][ ][Logger::\_\_construct ] ---------[20190429\_102557][INFO ][ ][Logger::\_\_construct ] INIT : OK

Le 1er bloc est la date du log, le 2nd le niveau du log, le 3ème l’actionId, le 4ème le module et la fonction ayant générés le log, suivis du log lui-même.

* + Action Id :

Il est possible de « décorer » une section des logs avec un "actionId" soit automatique, soit défini par le développeur. Pour cela, il suffit d’encadrer les logs voulus avec les méthodes correspondantes :

$this->setActionId() ;

$this->log( "Log #01", Logger::LOG\_INFO, \_\_METHOD\_\_ ) ;

$this->log( "Log #02", Logger::LOG\_INFO, \_\_METHOD\_\_ ) ;

$this->clearActionId() ;

Les lignes de logs sont alors écrites sous la forme :

[20190429\_102557][INFO ][5bd6d225ccee2][Logger::\_\_construct ] Log #01[20190429\_102557][INFO ][5bd6d225ccee2][Logger::\_\_construct ] Log #02

L’ "actionId" peut, quand c’est pertinent, être setté avec un id externe.

$myId = (...)

$this->setActionId( $myId ) ;

$this->log( "Log #01", Logger::LOG\_INFO, \_\_METHOD\_\_ ) ;

$this->log( "Log #02", Logger::LOG\_INFO, \_\_METHOD\_\_ ) ;

$this->clearActionId() ;

* ConfManager :
  + Configuration du module :

Accéder à un item de la configuration du module peut se faire, depuis n’importe quel endroit de l’implémentation d’un Module ou SubModule, via la méthode getConf :

$var = $this->getConf( 'server.port' ) ;

L’item sera automatiquement extrait du fichier <racine>/conf/u\_<ConnectorName>.php. Ce dernier n’est loadé qu’une fois par instanciation du module.

* + Autre configuration :

Accéder à n’importe quel item de la configuration peut se faire, depuis n’importe quel endroit de l’implémentation, via la méthode getGlobalConf :

$var = $this->getGlobalConf( 'tools.logger.behavior.dateFormat' ) ;

Dans ce cas, le ConfManager suit son fichier de configuration racine pour résoudre le chemin de la configuration : tools 🡺 logger, puis le contenu recherché.

Pour obtenir le même item de la configuration du module que l’exemple précédent, on aurait simplement fait :

$var = $this->getGlobalConf( 'user.<ConnectorName>.server.port' ) ;

## Interaction Manager (sous module)

Ce sous-module porte la responsabilité des interactions avec le CRM externe.

Dans le cadre de l’échantillon fourni, il implémente :

* les deux méthodes de recherche socles :
  + getSingleEntry : récupère une entrée du CRM externe sur la base :
    - de son type (entité),
    - de son id.
  + getEntriesList : récupère n entrées du CRM externe sur la base :
    - de leur type (entité),
    - du champs sur lequel va s’effectuer cette recherche,
    - de la valeur connue (éventuellement partielle) de ce champs,
    - de l’opération à effectuer (like, equals par exemple dans le cadre de requêtes de type SQL),
    - de la liste des champs de l’entité que l’on souhaite récupérer pour chaque entrée.

Ces deux méthodes masquent la demande d’autorisation de l’utilisation des APIs du CRM, et son éventuel renouvellement. On peut donc y faire appel directement, sans explicitement récupérer cette autorisation.

Ces deux méthodes retournent les données brutes telles que les ont renvoyées les API du CRM. C’est l’impléméntation du connecteur qui gère la logique sur la base de laquelle ces méthodes seront appelées, et le Entities Manager qui prendra la responsabilité du mapping depuis ces données brutes vers le format attendu par Kiamo.

* l’obtention de l’autorisation d’utilisation des APIs du CRM, généralement via un token de session. Dans ce cadre :
  + il implémente la récupération de cette autorisation,
  + si pertinent, la conservation du token de session, et sa réutilisation tant que ce dernier est valide (pour éviter des appels inutiles au CRM),
  + le renouvellement de l’autorisation, si cette dernière s’invalide.

## Entities Manager (sous module)

Ce module porte la responsabilité du mapping entre les données provenant du CRM externe et les entités Kiamo, en se basant sur la configuration.

Dans le cadre de l’échantillon fourni, il implémente :

* les deux méthodes de mapping socles :
  + getEntryInstance : à partir du type d’entité et du résultat d’une entrée brute retournée par le CRM, construit une entité Kiamo.
  + getEntriesCollection : même chose que la méthode précédente, pour une liste d’entrées.
* les méthodes requises par le connecteur CRM Kiamo :
  + getEntityLayout : modèle de présentation des entités Kiamo, sous la forme d’une liste de champs et quelques données additionnelles.
  + getEntitiesSearchUrl : construit l’URL de la page de recherche d’entités données dans le CRM externe.
  + getEntryExternalUrl : construit l’URL de la fiche de l’entrée fournie.
  + getEntryLabel : construit le label d’une entrée donnée.
* des raccourcis d’accès à différents points de la configuration, et le mapping entre les clés internes, les noms de champs CRM externe et types Kiamo.

## Customization Manager (sous module)

La responsabilité du Customization Manager est de porter :

* les méthodes d’éligibilité des inputs.

Ces méthodes scrutent les entrants et déterminent s’ils sont éligibles ou non à une recherche sur un type donné (un numéro de téléphone, un nom, un email, …). L’objectif est d’éliminer des sollicitations inutiles du CRM externe.

* les méthodes de pré-traitement des inputs, et de post-traitement des réponses du CRM.

Les méthodes de pré-traitement des inputs ont pour but d’uniformiser le format des données qui vont être utilisées dans un appel, quand c’est nécessaire, pour le faire correspondre au format stocké dans le CRM externe.

Par exemple, rajouter des espaces chaque 2 caractères dans un numéro de téléphone à 10 chiffres.

Les méthodes de post-traitement des résultats de recherche, quand c’est nécessaire.

Par exemple, enrichir ou pré-filtrer une liste de résultats.

* les filtres de liste n => 1.

Méthodes nécessaires à la sélection de l’entrée la plus pertinente, quand une liste a été retournée et que Kiamo n’attend qu’un seul résultat.

Par exemple, sélection du ticket client le plus pertinent, si plusieurs ouverts en même temps ou si uniquement des tickets clos sont retournés.

* toute autre méthode client nécessaire.

# FACILITATEURS PRINCIPAUX

## Module

Un Module est une classe dont l’objectif est de rassembler les capacités essentielles au développement d’une fonctionnalité, comme un gestionnaire de configuration ou un logger par exemple, et de les propager aux classes implémentant la fonctionnalité, ce de la manière la plus transparente possible.

Il n’est pas nécessaire de décrire en détail le fonctionnement de la classe Module.

Tout module implémentant une fonctionnalité principale (donnant lieu à des logs dédiés), possède les capacités suivantes :

* Logger :
  + log( $logStr, [ $logLevel, $caller ] ) 🡺 écrire un log dans le fichier de log du jour,
  + actionId :
    - setActionId( [ $actionId ] ) 🡺 déclarer une section des logs décorée par un actionId,
    - clearActionId() 🡺 cesser la décoration des logs avec l’action Id,
    - getActionId() 🡺 récupérer l’action Id courant,
* ConfManager :
  + getGlobalConf( [ $confPath ] ) 🡺 récupérer un item depuis la racine de la conf.
    - Si $confpath n’est pas renseigné, retourne toute la conf ayant été loadée à cet instant (le load ne se faisant qu’ « on demand ») ;
    - $confPath peut se présenter sous la forme d’une array ou d’une chaine de caractères avec le séparateur ‘.’ :

getConf( [ 'tools', 'logger', 'behavior', 'dateFormat'] )

getConf( 'tools.logger.behavior.dateFormat' )

* + getConf( [ $confPath ] ) 🡺 récupérer un item de la conf du module courant.

Remarque :

Si la classe étendant le Module définit un constructeur et ne se situe pas directement dans le répertoire du Toolkit, il est nécessaire d’appeler le constructeur du Module en première ligne du contructeur de la classe fille.

Le cas échéant, on peut souhaiter spécifier le path des logs de cette classe, et/ou le path de la configuration de cette classe. On peut les passer en paramètre du constructeur de Module.

Ex :

class MyClass extends Module

{

public function \_\_construct()

{

parent::\_\_construct( <PathToTheToolkit>,

<LogsFolderPath>,

<ConfFolderPath> ) ;

(...)

## ConfManager

Il n’est pas nécessaire de décrire en détail le fonctionnement de la classe ConfManager.

Cette classe est un helper de gestion de configuration :

* il se base sur sa configuration racine, le fichier <racine>/conf/\_config.php, pour résoudre et loader automatiquement les configurations demandées.
* un fichier de configuration ne sera loadé qu’à la demande, si on cherche à accéder à un item de ce fichier. Une fois le fichier loadé, on n’accèdera plus qu’à l’image mémoire de la conf.
* les fonctions utilisables par les modules de fonctionnalité sont principalement :
  + getConf( $itemPath ) : accès à un item de la conf. Si la conf n’a pas encore été loadée, ce sera fait à ce moment là. C’est un moyen de forcer le load d’une conf.
  + isInConfig( $itemPath ) : vérification de la présence ou non d’un item dans la conf. Si la conf contenant l’item n’a pas été préalablement loadée, le retour sera false même si l’item est présent dans l’arborescence des fichiers de conf.
  + loadConf( $confPath ) : forcer le load d’une conf. Le paramètre doit être le chemin de l’objet de configuration, comme 'tools.logger' ou 'user.MyModule' par exemple.

Note : sauf exception, forcer le load d’une conf par ce biais est inutile.

* + addToConf( $itemPath, $key, $value ) : permet d’ajouter un sous arbre de configuration, ou un item de configuration, à la configuration actuelle, manuellement. Si on utilise le système de fichiers de conf, a priori cette fonction est inutile.

## Logger

Il n’est pas nécessaire de décrire en détail le fonctionnement de la classe Logger.

Cette classe est un helper de gestion des log. Elle est instanciée automatiquement pour tout module créé, et écrira ses logs dans un répertoire dédié au module, et à ses sous-modules.

Il est aussi possible d’invoquer la fonction principale "log" comme une méthode de classe, sans instance de Logger. Le cas échéant, les logs seront écrits dans des fichiers à la racine du dossier de logs. Tout le comportement défini par la conf est préservé.

* Le Logger se base sur le fichier de conf <racine>/conf/t\_logger.php, qui définit :
  + le comportement (clé ‘behavior’) :
    - niveau de logs global (clé ‘globalLogLevel’), parmi (ordre croissant d’importance) :
      * Logger::LOG\_VERBOSE,
      * Logger::LOG\_TRACE,
      * Logger::LOG\_DEBUG,
      * Logger::LOG\_INFO,
      * Logger::LOG\_WARN,
      * Logger::LOG\_ERROR,
      * Logger::LOG\_CRITICAL.

Seuls les logs d’un niveau supérieur ou égal à ce niveau global seront écrits.

* + - format de la date (clé ‘dateFormat’),
    - nom de l’appelant « intelligent » (clé ‘smartMethodName’) :
      * engagé (clé ‘enabled’) :
        + si false, le nom de la méthode appelante sera écrite sans transformation,
        + si true, le nom de la méthode sera ajusté pour rendre la lecture des logs plus aisée (voir ci-dessous).
      * taille fixe (clé ‘strictLength’) : le bloc contenant le nom de la méthode sera forcé à cette taille fixe. Si le nom est plus court, le reste est rempli de blancs ; si le nom est plus long, il est tronqué à droite.
  + la gestion des fichiers de log (clé ‘files’) :
    - dossier racine des logs (clé ‘folder’) : chemin relatif et nom du dossier racine des logs.
    - gestion des fichiers de logs anciens (clé ‘obsolete’) :
      * zip (clé ‘zipOlderThan’) : les fichiers de log plus vieux que le nombre de jours défini, sont zippés (individuellement).
      * effacement (clé ‘deleteOlderThan’) : les fichiers de log plus vieux que le nombre de jours défini, sont effacés.

Note : tout item de cette configuration peut être absent du fichier. Le cas échéant, le logger se rabattra sur sa configuration par défaut (voir le code du logger).

* Les logs sont regroupés par fichiers journaliers : <racine>/logs/<ConnectorName>/YYYYMMDD.log
* La fonction principale est log( $str, [ $level, $method, $actionId, $indentLevel ] ) :
  + $str est le log lui-même,
  + $level, le niveau du log (par défaut, Logger::LOG\_DEBUG),
  + $method, le nom de la méthode appelante. Par facilité, il est conseillé d’utiliser la « constante magique » \_\_METHOD\_\_.
  + $actionId : l’action Id du log. Sauf besoin spécifique, il est conseillé d’utiliser les méthodes setActionId et clearActionId pour décorer une section de logs, et d’ignorer ce paramètre manuel.
  + $indentLevel : valeur entière indiquant le nombre d’espace à ajouter en préfixe du log (après le bloc du nom de la méthode appelante). Sauf besoin spécifique, c’est un paramètre à ignorer.
* Les lignes de logs se présentent sous la forme :

[20190429\_102557][INFO ][ ][Logger::\_\_construct ] ---------[20190429\_102557][INFO ][ ][Logger::\_\_construct ] INIT : OKoù :

* + le 1er bloc est la date du log
  + le 2nd le niveau du log
  + le 3ème l’actionId
  + le 4ème le module et la fonction ayant générés le log
  + Puis le log lui-même.

Exemple d’utilisation du logger de manière statique, dans une bibliothèque de fonction Kiamo :

require\_once "kiamoServices/ToolkitPHP/tools/autoload.php" ;

use KServicesTools\Logger ;

Logger::log( 'Here is a first log', Logger::LOG\_INFO ) ;

$functLibName = "MaFonctionBibliotheque01" ;

$currentCallRef = (...) ;

Logger::log( 'Here is a second log tagged with the function name and the callref',

Logger::LOG\_INFO, $functLibName, $ currentCallRef ) ;

## SubModule

Le but de cette classe est de propager à tout sous-module, et de manière transparente, les capacités Logger et ConfManager du Module parent (le module principal), ainsi que la capacité d’avoir sa propre configuration dédiée.

Un développeur peut, pour son module principal, souhaiter définir l’implémentation d’un sous-module, tout en lui propageant ses capacités de log et de gestion de conf.

Par exemple, pour une fonctionnalité devant utiliser une base de donnée MySQL et une autre MongoDB, le développeur peut écrire le corps de son implémentation dans son module principal, et créer deux sous-modules accesseurs des DB.

A l’instar du module principal devant étendre la classe Module, un sous-module doit étendre la classe SubModule :

<?php

namespace <ConnectorNamespace> ;

require\_once( \_\_DIR\_\_ . DIRECTORY\_SEPARATOR . "tools" . DIRECTORY\_SEPARATOR . 'autoloader.php' ) ;

use <ConnectorNamespace>\ConfManager ;

use <ConnectorNamespace>\SubModule ;

class MySubModule **extends SubModule**

{

public function \_\_construct( $\_parent ) // The \_parent must be a module

{

parent::\_\_construct( $\_parent, get\_class( $\_parent ), ConfManager::UserConfType ) ;

}

public function doSomething( $args )

{

(...)

}

(...)

}

?>

Le sous-module bénéficiera alors des même fonctionnalités que le Module : log, setActionId, clearActionId, getConf, getGlobalConf, pour accéder à la configuration du module parent.

Note : dans le \_\_construct\_\_, le deuxième paramètre de l’instanciation est le nom du sub module. Si ce nom est différent de celui de la classe parent, cela ouvre les possibilités de bénéficier de logs, d’une configuration et de ressources spécifiques au SubModule ; par contre, l’accès transparent à la conf, aux logs et aux ressource du Module parent est alors coupé. Il faut alors passer par $this->\_parent-><capability> pour y accéder.

# Outils

Les outils ont pour but de fournir de facilitateurs de manipulation des données, et ont pour objectif d’être enrichis au fil de l’eau en fonction des besoins et des développements.

## datetimes

La lib datetimes fournit un set de fonctions permettant de manipuler aisément des dates, en divers formats : string formatée, timestamp (en secondes), millis (en millis), …

Le socle de cet outil est la génération de dates sous la forme : YYYYMMSS\_hhmnss. Cela dit, tout format de date est possible.

Les fonctionnalités de datetimes sont des fonctionnalités de génération, opérations, comparaison et conversions, principalement.

## dict

La lib dict fournit la classe Dict, qui est un helper assez complet de manipulation d’arrays sous la forme key 🡺 value, où value est une donnée ou un sous arbre.

Si l’on possède une array dont on veut se faciliter la manipulation, on crée une instance de Dict à partir de cette array. Le Dict offre alors les capacités principales suivantes :

* get : accès à n’importe quel point de l’array par path.

Le path est, au choix, soit une array, soit une chaine avec séparateur ‘.’ :

get( [ 'tools', 'logger', 'behavior', 'dateFormat'] )

get( 'tools.logger.behavior.dateFormat' )

Un paramètre additionnel, le booléen $strict à false par défaut, est disponible :

* + s’il est est false, la méthode renverra null si le path ne mène à rien (non présent dans le Dict),
  + s’il est est true, la méthode génèrera une exception dans ce même cas.
* set : pendant du get, setting de la valeur de n’importe quel point de l’array.

set( [ 'tools', 'logger', 'behavior', 'dateFormat'], "Ymd\_His" )

set( 'tools.logger.behavior.dateFormat' , "Ymd\_His" )

Le paramètre additionnel $strict est présent, à false par défaut :

* + s’il est est false, la méthode crée l’intégralité du path si nécessaire (absent du Dict),
  + s’il est est true, la méthode génèrera une exception dans ce même cas.

Ainsi, on peut directement, en un seul set, créer le path 'tools.logger.behavior.dateFormat' dans une array vide.

* del : efface un path dans l’array.

Comportement comparable aux get et set. Ne fait rien si le path n’existe pas (pas de mode strict).

* exists : vrai si un path existe dans l’array, false autrement.

Comportement comparable aux get et set. Pas de mode strict.

* fromFile : load un fichier contenant un dictionnaire (array formée de clé 🡺 valeur où valeur peut être une donnée ou un dictionnaire), dans le dictionnaire courant.

fromFile( $filepath )

le contenu du fichier $filepath remplace le contenu du Dict courant.

fromFile( $filepath, $atKeyPath )

le contenu du fichier $filepath est setté au path $atKeyPath dans le Dict courant (mode non strict).

* json : retourne la chaine de caractères JSON représentant le Dict courant.

La lib fournit d’autre part des fonctionnalités communes aux Dict :

* Dict::splitKey( $key ) : transforme, si nécessaire, un path de clé de Dict sous la forme d’une array (utilisée pour la mécanique interne des fonctions des objets Dict).

Exemple : 'tools.logger.behavior.dateFormat' 🡺 [ 'tools', 'logger', 'behavior', 'dateFormat']

* Dict::joinKey( $key ) : transforme, si nécessaire, un path de clé de Dict sous la forme d’une string (utilisation principale : logging ou affichage).

Exemple : [ 'tools', 'logger', 'behavior', 'dateFormat'] 🡺 'tools.logger.behavior.dateFormat'

## files

files définit la classe statique Files, permettant la manipulation de fichiers :

* Files::existsFile( $filepath ) : comme son nom l’indique, true ou false.
* Files::srm( $filepath ) : efface un fichier. Aucune action si le fichier n’existe pas.
* Files::fileInfos( $filepath ) : retourne les informations système du fichier.
* Files::folderFiles( $pattern [, $flags = 0 ] ) : retourne les fichiers correspondant au pattern fourni.
* Files::zipFile( $filepath [, $deleteSource = true ] ) : zippe le fichier fourni dans son propre répertoire, et efface la source si demandé.

## resources

Cette classe statique Resources fournit les même fonctionnalités que la classe Files (à l’exception de la méthode folderFiles).

La différence est que la classe Resources ne manipule que des fichiers, individuellement, et ajoutera automatiquement comme path le chemin <racine>/data/<ModuleName>/ , en se basant sur le ModuleName de l’appelant. Le module pourra ainsi manipuler directement ses fichiers par nom de fichier, sans se préoccuper du path.

Exemple :

Resources::existsFile( ‘data.json’ ) retournera true si le fichier ‘data.json’ est présent dans le répertoire <racine>/data/<ModuleName>/ .

## strings

Cette classe statique fournit différentes méthodes de manipulation de chaines de caractères.

* Strings::strStartsWith( $sourceStr, $searchedStr ) : comme son nom l’indique, true ou false.
* Strings::adjust( $sourceStr, $size = -1, $border = RIGHT, $fill = ' ' ) : retourne une chaine de caractère ajustée selon les paramètres passés :
  + $size  : taille fixe. Si nécessaire, le remplissage sera assuré par le caractère $fill.
  + $border  : ajustement à droite, gauche ou centré.
  + $fill  : caractère de remplissage
* Strings::getJson( $data, $pretty = false ) : retourne la chaine de caractères JSON représentant la donnée passée.
* Strings::hprint( $data, $prefix = '', $postfix = '' ) : echo de la donnée dans la page Web.

## userdata

Classe statique UserData de manipulation des données utilisateur : numéro de téléphone, email, etc.

* fonctions couldBe : vérification d’éligibilité à un type de donnée utilisateur :
  + UserData::couldBeId( $str ) : vrai si alphanum,
  + UserData::couldBePhone( $str ) : vrai si éligible comme numéro de téléphone,
  + UserData::couldBeEmail( $str ) : vrai si éligible comme email.
* fonctions format :
  + formatTelNumberForSearch : élimine les caractères non digit et les ‘0’ préfixes d’un numéro de téléphone,
  + formatTelNumberForDisplay : ne garde que les digits, et rajoute un ‘0’ préfixe si nécessaire.

## uuids

Classe statique Uuids de génération d’id uniques.

* Uuids::get( $strong = false, $length = DEFAULT\_SIZE ) : génère un id unique.
  + si $strong est false, cet id est simplement basé sur le timestamp ; sinon, l’unicité de l’id est renforcée par un générateur de nombre aléatoire fort.
  + si $length n’est pas fourni, la taille par défaut est 32 caractères.
* Uuids::realrand( $min, $max ) : génère un entier aléatoire (algo random fort) entre $min et $max.

## webs

Facilitateur statique d’envoi et packageur de résultat d’appels REST, via curl.

* restRequest( $url, $data = null, $header = null ) :

renvoie [ okFlag, curl\_error, http\_code, jsonResponse ]

* + si $data n’est pas null, la requête sera un POST, autrement un GET.

# Outils Connexes

Le package est fourni avec quelques outils connexes :

* un outil de test en ligne de commande,
* un outil de vérification du connecteur,
* un exemple de script SVI basique.

En effet, il est pratique d’une part de pouvoir effectuer des tests de son implémentation au cours de la phase de développement sans avoir à effectuer de stimulation via Kiamo ; d’autre part il est important de vérifier que le connecteur qu’on s’apprête à installer dans Kiamo ne va pas provoquer une erreur 500 sur la page d’intégration des connecteurs, ce qui arrive dès qu’un parse error ou une char encoding issue survient dans le connecteur, ce qui est courant durant la phase de mise au point.

## Testeur en ligne de commande

Il s’agit d’un petit bout de code php à la racine de chaque échantillon : CommandLineTester.php

Sa syntaxe d’appel est la suivante :

> php CommandLineTester.php -f --test=<testNb>

Son fonctionnement est le suivant :

* il vérifie que le numéro de test passé correspond bien à une fonction de test définie,
* le cas échéant, il instancie le connecteur et exécute cette fonction.

Les fonction de test définies sont positionnées dans une array. Chaque fonction de test se basera sur l’instance du connecteur générée, et le stimulera comme souhaité par le développeur.

Des fonctions de test sont fournies pour l’exemple.

## Vérificateur de connecteur

Il s’agit d’un répertoire fourni à la racine du package, contenant un bout de code php qui devra être appelé via un Web Browser.

Le but de cet outil est de faciliter le déploiement d’un connecteur.

Lorsque le connecteur est chargé par Kiamo, si n’importe quel type de problème d’implémentation est présent (parse error, bug, encodage des caractères – UTF-8 est obligatoire pour l’ensemble code et fichiers de conf), différents effets indésirables se produisent dans Kiamo : erreur 500, page blanche, page affichée partiellement, …).

Pour éviter ce type de désagrément, cet outil va effectuer un ensemble de vérifications basiques :

* Compilation : aucune parse error n’est détectée dans la classe du connecteur,
* Instanciation : il est possible de créer une instance de connecteur,
* Encodage des caractères : tous les fichiers doivent être encodés en UTF-8, ou en ASCII ANSI si aucun caractère étendu n’est présent. Le checker vérifiera ces points sur :
  + le dossier du connecteur (non récursif)
  + le dossier conf du connecteur, si présent (non récursif),
  + le dossier tools du connecteur, si présent (non récursif).

Le dossier du checker peut être déployé dans le répertoire Kiamo :

<KIAMO>\Kiamo\data\userfiles\public

Pour vérifier un connecteur, si le Kiamo Manager est lancé, taper la ligne suivante dans un Web Browser :

[http://127.0.0.1/public/ConnectorSimpleChecker/?name=<YouConnectorFolderName](http://127.0.0.1/public/ConnectorSimpleChecker/?name=%3cYouConnectorFolderName)>

Le checker signalera toute erreur inattendue.

Note : la détection d’encodage de caractère n’étant pas une science exacte, il arrive que le checker remonte des faux positifs. Le cas échéant une fois vérification faite, ignorer les faux positifs.

## Exemple de script SVI

Un exemple de script SVI basique est fourni à la racine du package.

Il met en œuvre un connecteur appelé 4 fois :

* recherche d’un contact lié à l’appel entrant,
* recherche d’une société, celle du contact trouvé dans le bloc précédent ou via le numéro entrant,
* recherche d’un ticket ouvert,
* mise à jour du label d’interaction au nom du contact, via l’appel au bloc contact (qui utilisera l’id de contact déjà trouvé).

C’est une simple illustration de l’utilisation d’un connecteur dans Kiamo, pour l’exemple.

