Développement d’un connecteur avec Passerelle

Passerelle est le module en charge de l'interconnexion de Publik vers les API d'autres applications; cette documentation concerne le développement d'un connecteur en utilisant Passerelle et s'appuie pour cela sur l'installation d'un environnement de développement local.

 Passerelle n'est pas l'unique possibilité d'interconnexion de Publik vers les autres applications, tous les échanges de Publik se font en HTTPS, cela permet à toute application tierce, en exploitant les actions, formats et API de Publik, d'interopérer directement avec celui-ci, sans connecteur particulier, ou à travers d'un module indépendant (ESB, proxy d'API externe, etc.).

Démarrage

Il faut d'abord créer un répertoire qui contiendra le connecteur, appelons ce répertoire "passerelle-test" (mkdir passerelle-test).

Dans ce répertoire (cd passerelle-test), il faut créer le module Python qui correspondra au connecteur, ici les tirets ne sont pas autorisés dans les noms, appelons-le donc "passerelle_test", un module Python est un simple répertoire (mkdir passerelle_test) contenant un fichier __init__.py, qui peut être vide (touch passerelle_test/__init__.py).

On a alors cette arborescence de fichier :

```
passerelle-test/
passerelle-test/passerelle_test/
passerelle-test/passerelle_test/__init__.py   # fichier vide
```

Code du connecteur

Le code du connecteur se crée dans le fichier passerelle-test/passerelle_test/models.py :

```
passerelle-test/
passerelle-test/passerelle_test/
passerelle-test/passerelle_test/__init__.py
passerelle-test/passerelle_test/models.py      # code du connecteur
```

Ce fichier models.py décrit un modèle Django, dans Passerelle on le fait hériter du modèle "BaseResource" qui contient déjà tout ce qui structure un connecteur :

```
from passerelle.base.models import BaseResource

class TestConnector(BaseResource):
    category = 'Divers'

    class Meta:
        verbose_name = 'Connecteur de test'
```

La catégorie où se range le connecteur se définit dans l'attribut category et le nom du connecteur va lui dans les métadonnées du modèle.

Très clairement ce connecteur ne fait encore rien mais il y a désormais assez de code pour l'instancier ; mais avant cela, il faut qu'il puisse être installé sur le système.

Installation du connecteur

Pour que le connecteur puisse être installé, il faut créer un fichier setup.py à la racine du projet (i.e. directement dans le répertoire passerelle_test/); son contenu peut être minimal :

```
#! /usr/bin/env python

from setuptools import setup, find_packages

setup(
    name='passerelle-test',
```

```
    author='John Doe',
    author_email='john.doe@example.net',
    url='http://example.net/',
    packages=find_packages(),
)
```

Le code d'installation exige la présence d'un fichier de documentation README, il doit donc être créé pour l'occasion (il peut très bien être vide, touch README).

À ce stade, l'arborescence du projet est donc celle-ci :

```
passerelle-test/
passerelle-test/README
passerelle-test/setup.py
passerelle-test/passerelle_test/
passerelle-test/passerelle_test/__init__.py
passerelle-test/passerelle_test/models.py
```

Il est maintenant possible de lancer l'installation, en mode développement pour permettre à nos modifications ultérieures d'être prises en compte sans demander à chaque fois une réinstallation (/home/utilisateur/envs/publik-env-py3/bin/python setup.py develop@).

```
$ /home/utilisateur/envs/publik-env-py3/bin/python setup.py develop
running develop
running egg_info
writing passerelle_test.egg-info/PKG-INFO
writing top-level names to passerelle_test.egg-info/top_level.txt
writing dependency_links to passerelle_test.egg-info/dependency_links.txt
reading manifest file 'passerelle_test.egg-info/SOURCES.txt'
writing manifest file 'passerelle_test.egg-info/SOURCES.txt'
running build_ext
Creating /home/utilisateur/envs/publik-env-py3/lib/python3.7/site-packages/dist-packages/passerell
e-test.egg-link (link to .)
Adding passerelle-test 0.0.0 to easy-install.pth file

Installed /home/test/passerelle-test
Processing dependencies for passerelle-test==0.0.0
Finished processing dependencies for passerelle-test==0.0.0
```

Activation du connecteur

Le connecteur a désormais tout le nécessaire pour être activé, il reste juste à le déclarer dans la configuration, pour cela il faut créer un fichier /home/utilisateur/.config/publik/settings/passerelle/settings.d/connecteur.py (ou tout autre nom en .py), avec ces deux lignes :

```
INSTALLED_APPS += ('passerelle_test',)
TENANT_APPS += ('passerelle_test',)
```

Migration initiale

Django dispose d'un système automatique de migration des données des différents modèles, qui facilite grandement les mises à jour. Dans la majorité des cas ce système est automatique, il est temps de l'initialiser en créant une première migration :

```
$ /home/utilisateur/envs/publik-env-py3/bin/passerelle-manage --forceuser makemigrations passerell
e_test
Migrations for 'passerelle_test':
  0001_initial.py:
    - Create model TestConnector
```

Si cette commande échoue sur un problème d'accès, vérifiez que votre utilisateur est bien dans le groupe "passerelle".

Instanciation du connecteur

Passerelle peut désormais être rédémarré, sudo supervisorctl restart passerelle. Et voilà, il est maintenant possible de naviguer, le nouveau connecteur apparaîtra dans la liste des connecteurs disponibles. Créons-le et donnons-lui le nom de "démo", sa première instance devient disponible : https://passerelle.dev.publik.love/passerelle-test/demo/

Endpoints

C'est ici que tout commence car bien sûr un connecteur vide n'a aucun intérêt, il faut désormais lui ajouter le code qui le rendra utile, sous forme de "endpoints", les URL qui seront utilisées pour exposer les webservices.

Il faut importer le code permettant de déclarer les endpoints from passerelle.utils.api import endpoint en haut du fichier models.py, et ensuite, dans le modèle TestConnector créé plus haut, nous pouvons alors déclarer un premier endpoint,

```
class TestConnector(BaseResource):
    […]

    @endpoint()
    def info(self, request):
        return {'hello': 'world'}
```

Redémarrons Passerelle pour que le code soit pris en compte (sudo supervisorctl restart passerelle) et voilà, accéder à l'URL du endpoint, https://passerelle.dev.publik.love/passerelle-test/demo/info retourne la donnée au format JSON :

```
{
  "hello": "world"
}
```

Il est aussi possible pour un endpoint d'attendre des paramètres, déclarons donc un deuxième endpoint, faisant l'addition de nombres entiers et retournant le résultat :

```
class TestConnector(BaseResource):
    […]

    @endpoint()
    def addition(self, request, a, b):
        return {'total': int(a) + int(b)}
```

Visiter https://passerelle.dev.publik.love/passerelle-test/demo/addition?a=2&b=3 retournera au format JSON :

```
{
  "total": 5
}
```

Journalisation

Passerelle suit la configuration de la journalisation établie au niveau de Django (cf Journalisation dans la documentation officielle de Django) et ajoute à celle-ci une journalisation interne, consultable depuis la page de visualisation du connecteur.

L'objet logger d'un connecteur dispose des méthodes standards de log (debug, info, warning, critical, error, fatal); ainsi un connecteur opérant une division pourrait logguer au niveau "debug" toutes les demandes et logguer au niveau "error" les divisions par zéro :

```
class TestConnector(BaseResource):
    […]

    @endpoint()
    def division(self, request, a, b):
        self.logger.debug('division de %s par %s', a, b)
        try:
            return {'result': int(a) / int(b)}
        except ZeroDivisionError:
            self.logger.error('division par zéro')
            raise
```

Autorisations d'accès

Par défaut les endpoints ajoutés sont privés, limités à une permission can_access, un tableau de gestion des autorisations d'accès est automatiquement inclus à la page de gestion du connecteur.

can_access est le nom par défaut utilisé pour limiter l'accès mais il est possible d'augmenter la granularité des autorisations, on

pourrait ainsi avoir un autre endpoint dont l'accès serait différencié :

```
class TestConnector(BaseResource):
    […]

    _can_compute_description = 'Accès aux fonctions de calcul'

    @endpoint(perm='can_compute')
    def addition(self, request, a, b):
        return {'total': int(a) + int(b)}
```

À noter également l'ajout dans la classe d'un attribut _can_compute_description, qui servira au niveau de l'interface à fournir une description pour cette autorisation d'accès.

Il est possible de créer des accès ouverts, en passant OPEN comme permission d'accès :

```
class TestConnector(BaseResource):
    […]

    @endpoint(perm='OPEN')
    def info(self, request):
        return {'hello': 'world'}
```

Documentation

Pour renforcer l'utilité de la page d'information d'un connecteur il est important de documenter le connecteur et ses différents endpoints. Le connecteur en lui-même peut être décrit en ajoutant un attribut api_description. Pour les endpoints, cela passe par l'ajout de paramètres à la déclaration @endpoint :

```
class TestConnector(BaseResource):
    […]
    api_description = "Ce connecteur propose quelques opération arithmétiques élémentaires."

    @endpoint(description='Addition de deux nombres entiers',
              parameters={
                'a': {'description': 'Un premier nombre', 'example_value': '7'},
                'b': {'description': 'Un second nombre', 'example_value': '2'}
              })
    def addition(self, request, a, b):
        return {'total': int(a) + int(b)}
```

Dans description se place donc une description général de l'endpoint et dans parameters se place les informations sur les différents paramètres autorisés par l'appel, pour chacun d'eux une description et une valeur qui sera reprise en exemple peuvent être fournies.

Paramétrages supplémentaires

À l'instanciation d'un connecteur quelques paramètres sont demandés, un titre et une description, ainsi que le niveau de journalisation souhaité. Il est possible d'ajouter d'autres paramètres, il faut pour cela ajouter de nouveaux attributs à la classe du connecteur. Comme noté en début de document cette classe correspond à un modèle Django, il s'agit ainsi d'y ajouter des champs.

Comme exemple, créons un connecteur dont le rôle sera de nous renseigner sur le bon état de fonctionnement, ou pas, d'une adresse, en faisant en sorte que cette adresse soit un paramètre supplémentaire du connecteur.

Il faut donc ajouter un attribut au modèle, url = models.URLField(...) et cette information sera par la suite disponible dans self.url :

```
import requests
from passerelle.base.models import BaseResource

class TestConnector(BaseResource):
    category = 'Divers'
    url = models.URLField('URL', default='http://www.example.net')

    class Meta:
        verbose_name = 'Connecteur de test'

    @endpoint(description='Teste une adresse')
```

```
    def up(self, request):
        try:
            response = self.requests.get(self.url)
            response.raise_for_status()
        except requests.RequestException:
            return {'result': '%s est en panne' % self.url}
        return {'result': '%s est ok' % self.url}
```

Comme le modèle se trouve modifié, il est nécessaire de prévoir pour la base de données une migration qui ajoutera une colonne pour ce champ url, /home/utilisateur/envs/publik-env-py3/bin/passerelle-manage --forceuser makemigrations passerelle_test. Cette migration sera exécutée automatiquement au redémarrage de Passerelle, qui est donc nécessaire à cette étape.

Note : self.requests utilisé ici est un wrapper léger au-dessus du module Requests, qui ajoute une journalisation et une expiration automatique des appels.

Exécution de tâches planifiées (cron)

Un connecteur peut souhaiter exécuter des tâches de manière régulière, comme par exemple lancer une synchronisation avec un référentiel externe. Passerelle prend en charge quatre fréquences, une tâche peut être planifiée pour s'exécuter toutes les heures, tous les jours, toutes les semaines ou tous les mois. Pour ce faire il faut définir dans le connecteur une fonction avec la tâche à exécuter et la nommer selon la fréquence (hourly / daily / weekly / monthly).

```
from passerelle.base.models import BaseResource

class TestConnector(BaseResource):
    […]

    def hourly(self):
        pass  # code exécuté toutes les heures
```

 Les tâches planifiées sont lancées automatiquement dans un environnement de production debian mais pas dans un environnement de développement local.

Pour lancer manuellement une tâche planifiée dans un environnement de développement local (remplacer hourly pas daily / weekly / monthly selon le cas) :

```
/home/utilisateur/envs/publik-env-py3/bin/passerelle-manage tenant_command cron -d passerelle.dev.publik.love hourly
```

Suivi de la disponibilité

Dans le prolongement des tâches planifiées se trouve le suivi de la disponibilité. Un connecteur peut régulièrement (toutes les cinq minutes) interroger le service distant pour s'assurer de sa disponibilité et marquer celui-ci si jamais ce n'était pas le cas.

Il suffit pour assurer cela de définir une méthode check_status, le service sera considéré indisponible quand la méthode lèvera une exception.

```
from passerelle.base.models import BaseResource

class TestConnector(BaseResource):
    […]

    def check_status(self):
        response = self.requests.get('http://example.net')
        response.raise_for_status()
```

 Le suivi de disponibilité est lancé automatiquement dans un environnement de production debian mais pas dans un environnement de développement local.

Pour lancer manuellement le suivi de disponibilité dans un environnement de développement local :

```
/home/utilisateur/envs/publik-env-py3/bin/passerelle-manage tenant_command cron -d passerelle.dev.publik.love availability
```

Tâches asynchrones

Les tâches planifiées s'exécutent de manière régulière, il peut également être utile d'exécuter des tâches de manière ponctuelle, par exemple pour transférer en tâche de fond un fichier important. La méthode add_job permet d'ajouter une tâche de fond, qui sera exécutée après la requête.

```
class TestConnector(BaseResource):
    […]

    @endpoint(description='Transfert de données',
              parameters={
                  'src': {'description': 'Source'},
                  'dst': {'description': 'Destination'},
              })
    def transfert(self, request, src, dst):
        self.add_job('execution_transfert', src=src, dst=dst)
        return {'msg': 'Transfert planifié', 'err': 0}

    def execution_transfert(self, src, dst):
        # ici obtenir les données depuis la source puis les transférer à destination
        pass
```

La méthode add_job prend comme paramètres le nom de la méthode à exécuter puis une série libre de paramètres nommés, qui seront utilisés lors de l'exécution.

Quand des tâches existent, la page du connecteur les reprend dans un tableau.

 Dans un environnement de développement local, les tâche asynchrones ne sont exécutées que si passerelle utilise le serveur UWSGI .

Tests unitaires

Ils sont indispensables pour dormir tranquille, ils se créent dans un nouveau répertoire, tests et utilisent les modules pytest, pytest-django et django-webtest (ces trois modules doivent donc être installés).

Il y a d'abord à définir un fichier tests/settings.py particulier, qui assurera la présence du connecteur :

```
INSTALLED_APPS += ('passerelle_test',)
```

Puis ensuite créer un fichier pour les tests en eux-mêmes, ça peut être tests/test_connecteur.py, il commencera par les imports des modules souhaités puis la définition des objets qui seront ensuite utiles.

```
# -*- coding: utf-8 -*-

import pytest
import django_webtest

from django.contrib.contenttypes.models import ContentType
from passerelle_test.models import TestConnector
from passerelle.base.models import ApiUser, AccessRight

@pytest.fixture
def app(request):
    # création de l'application Django
    return django_webtest.DjangoTestApp()

@pytest.fixture
def connector(db):
    # création du connecteur et ouverture de la permission "can_access" sans authentification.
    connector = TestConnector.objects.create(slug='test')
    api = ApiUser.objects.create(username='all', keytype='', key='')
    obj_type = ContentType.objects.get_for_model(connector)
    AccessRight.objects.create(
            codename='can_access', apiuser=api,
            resource_type=obj_type, resource_pk=connector.pk)
    return connector
```

Vient alors enfin le temps de tester le connecteur, pour un test sur la fonction d'addition :

```
def test_addition(app, connector):
    resp = app.get('/passerelle-test/test/addition?a=5&b=3')
    assert resp.json.get('total') == 8
```

Les tests s'exécutent ensuite via la commande py.test, en pointant le fichier settings.py créé spécialement :

```
 $ DJANGO_SETTINGS_MODULE=passerelle.settings PASSERELLE_SETTINGS_FILE=tests/settings.py py.test
==================================== test session starts ====================================
=
platform linux2 -- Python 2.7.14+, pytest-3.3.1, ...
cachedir: .cache
Django settings: passerelle.settings (from environment variable)
rootdir: ..., inifile:
plugins: ...
collected 1 item

tests/test_connecteur.py::test_addition PASSED                                          [100%]

================================== 1 passed in 0.48 seconds ==================================
=
```

Au-delà

- gestion des POST.
- pattern sur les endpoints.
- Squelette HTML de description du connecteur.