

# Quickstart API Python

## Base de données

### Module db\_common

Le module db\_common propose des fonction utilitaires pour effectuer des requêtes sur la base de données.

### Lire plusieurs lignes

Nous allons utiliser la fonction `dbgenericgetrows()` dont la signature est la suivante:

```
def dbgenericgetrows(usession, itbl, icond=None, ifldorder=None,
ifldordersens="A", limit=None, limistart = 0)
    """
    Return rows from a table. Select all the fields.

    :type usession: session
    :param usession: NCHP Session
    :type itbl: string
    :param itbl: table to fetch
    :type icond : list
    :param icond: list of conditions

    :rtype: list
    :return: A list of row, each row described by a dictionnary.
    """
```

Exemple d'utilisation pour sélectionner toutes les lignes de la table test

```
rows = db_common.dbgenericgetrows(
    usession = gses,
    itbl = "factfour",
)

pprint.pprint(rows)
>>>
[{'FACTFOUR_CPPID': 0L,
  'FACTFOUR_DATE': datetime.datetime(2022, 6, 10, 11, 59, 55),
  'FACTFOUR_DATEDEPOTSAE': None,
  'FACTFOUR_DATEFACT': datetime.datetime(2016, 9, 29, 0, 0),
  'FACTFOUR_DEPOTSAE': 0,
  'FACTFOUR_FOURNISSEURID': 36L,
  'FACTFOUR_HT': 1350.0,
```

```
'FACTFOUR_ID': 13L,  
'FACTFOUR_NUMFACTURE': 'QUAL_00000000003821',  
'FACTFOUR_TTC': 1620.0,  
'FACTFOUR_TVA': 270.0  
, {  
'FACTFOUR_CPPID': 0L,  
'FACTFOUR_DATE': datetime.datetime(2022, 6, 10, 14, 36, 39),  
'FACTFOUR_DATEDEPOTSAE': datetime.datetime(2022, 6, 10, 14, 45, 14),  
'FACTFOUR_DATEFACT': datetime.datetime(2016, 9, 29, 0, 0),  
'FACTFOUR_DEPOTSAE': 1,  
'FACTFOUR_FOURNISSEURID': 36L,  
'FACTFOUR_HT': 1350.0,  
'FACTFOUR_ID': 14L,  
'FACTFOUR_NUMFACTURE': 'QUAL_00000000003821',  
'FACTFOUR_TTC': 1620.0,  
'FACTFOUR_TVA': 270.0  
, {  
'FACTFOUR_CPPID': 380066452L,  
'FACTFOUR_DATE': None,  
'FACTFOUR_DATEDEPOTSAE': None,  
'FACTFOUR_DATEFACT': None,  
'FACTFOUR_DEPOTSAE': 0,  
'FACTFOUR_FOURNISSEURID': 0L,  
'FACTFOUR_HT': 0.0,  
'FACTFOUR_ID': 15L,  
'FACTFOUR_NUMFACTURE': '0',  
'FACTFOUR_TTC': 0.0,  
'FACTFOUR_TVA': 0.0  
}  
}  
]
```

Un exemple maintenant en ajoutant des conditions

```
rows = db_common.dbgenericgetrows(  
    usession = gses,  
    itbl = "factfour",  
    icond = db_common.buildcondandor(  
        tbl = "factfour",  
        fields = ["FACTFOUR_CPPID", "FACTFOUR_DEPOTSAE"],  
        values = ["0", "1"]  
    )  
)  
  
pprint.pprint(rows)  
>>>  
[ {  
    'FACTFOUR_CPPID': 0L,  
    'FACTFOUR_DATE': datetime.datetime(2022, 6, 10, 11, 59, 55),  
    'FACTFOUR_DATEDEPOTSAE': None,  
    'FACTFOUR_DATEFACT': datetime.datetime(2016, 9, 29, 0, 0),
```

```
'FACTFOUR_DEPOTSAE' : 0,  
'FACTFOUR_FOURNISSEURID' : 36L,  
'FACTFOUR_HT' : 1350.0,  
'FACTFOUR_ID' : 13L,  
'FACTFOUR_NUMFACTURE' : 'QUAL_00000000003821',  
'FACTFOUR_TTC' : 1620.0,  
'FACTFOUR_TVA' : 270.0  
}  
]
```



La fonction `db_common.buildcondandor(...)` qui est utilisée pour construire les conditions ne gère que des conditions utilisant le même opérateur qui par défaut est l'opérateur `=`.

## Insérer une ligne

## Classe `sql_db`

Une autre façon de faire des requêtes est d'utiliser directement la classe `sql_db` qui est implémentée spécifiquement par chaque module dédié à un SGBD donné.

Par exemple pour MySQL l'API EzGED propose le module `db_mysql`

En général vous n'avez pas besoin d'instancier directement la classe `sql_db` car une instance est attachée à la session EzGED que vous aurez ouverte.

```
>>> import common  
>>> gses = common.dbses()  
>>> gses.db  
<db_mysql.sql_db instance at 0x0684F8F0>
```

Pour exécuter une requête on peut alors procéder ainsi

```
import common  
  
gses = common.dbses()  
result = gses.db.query("SELECT * FROM societycache;")  
  
with result:  
    for row in result:  
        print(row)
```



L'utilisation de `with` et l'itération sur l'objet `result` ne sont possibles qu'à partir de la version **3.4.23127**



Pour les versions antérieures voir [db](#) qui présente d'autres manières de procéder.

Ce qui nous affichera quelque chose comme ça:

```
{'SOCIETYCACHE_SIREN': '552ZZ081317', 'SOCIETYCACHE_SOCIETYNAME':  
'ELECTRICITE DE FRANCE', 'SOCIETYCACHE_ID': 5L}  
{'SOCIETYCACHE_SIREN': '515180115', 'SOCIETYCACHE_SOCIETYNAME': 'EZDEV',  
'SOCIETYCACHE_ID': 24L}  
...
```

## Logging

EzGED utilise le module python standard logging pour la journalisation de messages.

Les loggers suivants peuvent être utilisés:

- ezged ⇒ toute journalisation concernant EzGED.
- jobd ⇒ journalisation concernant le serveur de travaux
- convd ⇒ journalisation dans le cadre du serveur de conversion
- eztest ⇒ journalisation pour les scripts de test

Le logger a utiliser dépend donc du contexte dans lequel votre code est susceptible de s'exécuter.

Donc pour obtenir le logger ezged il suffirait de faire

```
import logging  
ezged_logger = logging.getLogger("ezged")
```

Mais vous n'êtes pas obligé de le connaître et le plus simple reste d'utiliser notre fonction `_common.get_logger()` afin de récupérer le logger correspondant au contexte d'exécution.

```
import _common  
app_logger = _common.get_logger()
```

`_common` est un module python EzGED qui contient de nombreuses fonctions utiles et globalement utilisées dans toute l'application. Dont notamment la fonction `get_logger()`

## Journaliser les activités

Dans EzGED un certain nombre d'entités tels que les fiches descriptives ou les fiches sont l'objet d'activités ou d'évènements.

Pour ajouter une activité à une fiche descriptive on peut par exemple utiliser la fonction `dblog.dblogaddindexifnotexist`. Exemple:

```
import dblog

dblog.dblogaddindexifnotexist(
    usession = gses,
    tbl = "facture",
    rsid = 12,
    secusrid = 1,
    action = "C",
    text = log_msg,
    tstamp = _common.tsatamp()
)
```

L'appel ci-dessus va créer une nouvelle "activité" pour la facture d'identifiant 12. Il s'agit d'une activité de type "Création" indiqué par le paramètre action = "C".

## Travaux

### Créer un travail EzGED

#### Script d'étape

La logique d'exécution des étapes d'un travail dans EzGED sont implémentées via des scripts pythons.

Le plus simple est de partir d'un [template de script](#).

#### Progression du script

On peut mettre à jour l'avancement de l'exécution de l'étape en utilisant la fonction `libjobdext.jobstateupdate_percent`:

```
libjobdext.jobstateupdate_percent(gses, jobqueueid, jobstepnumber,
jobstatepercent)
```

`jobstepnumber` est le numéro de l'étape. On peut passer 0 et laisser la fonction le déterminer. `value` est la valeur d'avancement du job où une valeur de 1 représente une progression de 100%.

Exemple pour positionner une progression de 30% sur le travail 1234

```
libjobdext.jobstateupdate_percent(gses, jobqueueid=1234, jobstepnumber=0,
jobstatepercent=0.3)
```

#### Lecture/Ecriture des fichiers de travail

Les fichiers de travail sont les fichiers qu'utilisent les scripts pour écrire ou lire des informations

nécessaires à la réalisation du script et donc de son travail.

Parmi les principaux on retrouve `indexes`, `fssto` et `docpak`

Ces fichiers sont des fichiers textes contenant des données structurées en colonnes de longueur fixe.

Pour faciliter leur manipulation on utilise le module `scriptfiles`

## Instancier un travail

```
import libjobdext
jobid, activated = libjobdext.jobcreatefromtpl(
    usession = usession,
    id = "MYJOBCODE",
    secusrid = usession.userid,
    params = ["param1", "param2"],
    paramvalues = ["value1", "value2"],
    active = 0
)
```

MYJOBCODE est le code du travail à instancier. Le code peut être retrouvé en allant consulter la liste des Travaux de référence.

La fonction retourne l'identifiant du travail et indique si le travail a été activé ou non.

## Activer un travail

Dans l'exemple précédent nous avons instancier un travail en passant le paramètre `active` à 0 donc le travail a été créé (on peut le voir dans la liste des travaux depuis l'interface web par exemple) mais il n'est pas actif.

Ceci permet par exemple de copier des fichiers dans le répertoire du travail (qui aura été créé au moment de son instanciation). Une fois que la mise en place est terminée on peut finalement l'activer.

```
libjobdext.job_activate(usession, jobid)
```

## Fichiers

### Attacher un fichier à une fiche

Nous avons besoins d'importer les modules suivants

```
import fs
import docpak
```

La première consiste à stocker le fichier

```
new_fsfileid = fs.fsfilesto(gses, filepath, ifssaid = 0)
```

Si le fichier a bien été stocké alors new\_fsfileid est un entier strictement positif.

Nous pouvons ensuite attacher le fichier stocké à l'enregistrement (la fiche).

```
docpakid = docpaknew(gses, itbl="matable", irsid=1234 ,  
ifiles=[new_fsfileid])
```

## Créer une nouvelle version d'un fichier

From:

<http://wiki.ezdev.fr/> - **EzGED Wiki**

Permanent link:

<http://wiki.ezdev.fr/doku.php?id=docs:dev:api:python:quickstart&rev=1709048665>

Last update: **2024/02/27 15:44**

