ENTWANNE

Geste de savoir

Do not use requirements.txt

vendredi 01 mars 2024

Table des matières

Introduction 1
1. Le packaging en Python 1
2. Ne pas confondre packaging et déploiement 2
3. Et les environnements virtuels dans tout ça ? 4
Conclusion 4

Introduction

Cette semaine je tombais sur cet article en anglais nommé « Do not use requirements.txt » à propos du packaging en Python, qui donne deux conseils :

— Ne pas utiliser `pip` et les fichiers `requirements.txt` pour gérer les dépendances Python.
— Utiliser Poetry à la place.

Intrigué par le titre et ces conseils, je continuais ma lecture mais n'allais pas plus être convaincu.

1. Le packaging en Python

L'article commence par expliquer que « la manière traditionnelle de gérer les dépendances pour un projet Python était de les lister dans un fichier `requirements.txt` et d'utiliser `pip install -r requirements.txt` pour les installer » puis se base là-dessus pour continuer son argumentaire.

Mais ce n'est pas ainsi que fonctionne le packaging en Python.

Un projet Python définit normalement un fichier `pyproject.toml`[1] qui comporte toutes les métadonnées liées à ce projet (nom du projet, version, auteurs, et même des configurations pour des outils) ainsi que les dépendances nécessaires à son bon fonctionnement.

Ces dépendances sont des noms de paquets Python accompagnés de spécificateurs de versions pour préciser un intervalle de versions compatibles.

```toml
[project]
name = "zds-site"
version = "30.5.0"
```

1. Anciennement on pouvait aussi trouver des fichiers `setup.cfg` et/ou `setup.py`.

```
dependencies = [
    "Django>3.0",
    "requests==2"
]
requires-python = ">=3.8"
authors = [
  {name = "Clémentine Sanpépins", email =
      "clem@zestedesavoir.com"},
]
description = "Zeste de Savoir"

[project.urls]
Homepage = "https://zestedesavoir.com/"
Repository = "https://github.com/zestedesavoir/zds-site"
```

Listing 1 – Exemple simplifié de fichier `pyproject.toml`

Tout ce beau monde s'installe ensuite avec `pip`. `pip` est l'outil standard pour installer des dépendances en Python et même s'il n'est pas le seul , il reste la référence sur la question. Il repose sur les dépôts PyPI (Python package Index) où sont publiés la majorité des projets Python.

Il suffit par exemple d'un `pip install Django` pour installer le framework Django ainsi que ses dépendances.

Dans le cas d'un projet en cours de développement on va chercher à l'installer depuis les sources locales plutôt qu'un paquet distant (puisque la version en train d'être développée n'existe pas encore sur le dépôt) et `pip` gère pour cela les chemins locaux.

Ainsi, `pip install .`[2] depuis le répertoire du projet analysera le fichier `pyproject.toml` pour installer le projet décrit ainsi que ses dépendances.

Il n'est donc nulle question ici d'un fichier `requirements.txt` pour lister les dépendances du projet. Ce fichier peut exister et a son utilisé, mais j'y reviendrai plus tard.

On note d'ailleurs que Poetry, préconisé dans l'article que je cite, fonctionne sur le même mode (métadonnées et dépendances listées dans le fichier `pyproject.toml`) même s'il a son propre format pour les exprimer et ses propres outils pour installer le projet ensuite (en remplacement de `pip` donc).

2. Ne pas confondre packaging et déploiement

Je pense que mon désaccord avec l'article vient de la différence entre packaging et déploiement.

Je détaillais le packaging dans la section précédente : il s'agit de décrire le projet et les versions de dépendances avec lesquelles il est compatible. L'idée étant qu'un projet puisse être installé à différents endroits avec des versions différentes (il peut exister plusieurs instances d'un même projet). Et suivant l'endroit, le système d'exploitation ou les bibliothèques système installées, toutes les dépendances ne seront pas disponibles dans les mêmes versions. On veut donc faire en sorte que le panel de dépendances avec lesquelles le projet est compatible soit le plus large possible.

2. On utilisera plutôt la syntaxe `pip install -e .`. Cette option `-e` signifie « éditable » et indique de ne pas copier les sources du projet pour construire le paquet mais d'utiliser un lien symbolique afin que les modifications locales se répercutent sur la version installée.

Mais dans le cas d'un déploiement, pour la mise en ligne d'un projet, on veut assurer un environnement cohérent et reproductible. Pour cela il nous faut des versions précises des dépendances installées, afin de pouvoir installer exactement les mêmes versions dans des endroits différents :

- Si deux serveurs peuvent répondre aux requêtes pour un même site web, on veut qu'ils utilisent la même version du projet.
- Dans mon environnement local, je veux pouvoir débuguer le projet identique à celui déployé en production.

On remarque donc des objectifs contradictoires entre packaging et déploiement : dans le premier cas on veut spécifier les versions de dépendances les plus larges possibles et dans l'autre on veut être le plus restreint/précis possible.
Et c'est pour répondre à cette problématique de déploiement qu'interviennent les *lock files*. Ce sont des fichiers générés qui précisent (verrouillent) les dépendances (directes et indirectes) du projet dans un environnement donné. Un fichier `requirements.txt` peut remplir ce rôle.
Ce fichier est en fait une liste de dépendances dans une syntaxe comprise par `pip`.

```
asgiref==3.7.2
Django==5.0.2
requests==2.0.0
sqlparse==0.4.4
```

Listing 2 – Exemple de fichier `requirements.txt`

Chaque ligne du fichier est un argument valide à placer derrière un `pip install`.
Et c'est d'ailleurs ce que fait `pip install -r requirements.txt` : il analyse le contenu du fichier et traite chaque ligne comme s'il s'agissait d'un argument supplémentaire.[1]
Le fichier `requirements.txt` que je montre en exemple est généré par la commande `pip freeze` après avoir installé le projet d'exemple. Cette commande liste tous les paquets Python installés avec leurs versions. On y trouve ainsi `Django` et `requests` qui sont des dépendances directes de mon projet, mais aussi `asgiref` et `sqlparse` qui sont des dépendances de `Django`.
Précédemment, mon `pip install .` a donc résolu les dépendances du projet pour récupérer les versions les plus à jour répondant aux critères et les a installées. Ce sont ces versions qui sont listées ici.
La génération d'un tel fichier peut aussi se faire à l'aide de l'outil `pip-compile` issu de la suite `pip-tools` qui prend le fichier `pyproject.toml` en entrée et résout les versions des dépendances sans nécessiter de les installer.
Si je dispose de différents environnements qui font tourner le projet dans des conditions / versions différentes, je peux avoir des *lock files* différents. Par exemple `requirements_dev.txt` et `requirements_prod.txt`.

La confusion entre packaging et déploiement vient notamment du fait que beaucoup de projets (particulièrement les projets SaaS) n'existent qu'en une seule instance : le projet n'est pas distribué à l'extérieur de l'entreprise et celui-ci est toujours déployé dans des environnements similaires.
Ainsi il n'est pas nécessaire de spécifier des versions de dépendances larges (personne d'autre

1. Le fichier pourrait ainsi contenir `-r other_requirements.txt` pour inclure un second fichier de dépendances.

n'installera le projet) et les deux besoins convergent.
Il n'empêche qu'il s'agit de problématiques différentes et cela explique que l'outillage soit différent.

On notera enfin que le format du fichier `requirements.txt` n'est pas forcément optimal pour remplir le rôle de *lock file* (d'autres formats stockent une somme cryptographique du paquet et d'autres attributs).
Des discussions sont en cours pour convenir d'un format standard en Python pour cet usage.

3. Et les environnements virtuels dans tout ça ?

Le second point abordé par l'article cité concerne les environnements virtuels. Il s'agit d'un mécanisme de Python pour le « tromper » (via des variables d'environnement) en configurant des répertoires d'installation différents des répertoires systèmes pour les paquets. Ce qui permet de faire coexister sur la machine des versions différentes de même paquets dans des environnements différents.
L'outil standard pour gérer cela est `venv` (fourni avec Python), qui s'utilise via `python -m venv`.
La critique émise étant que `pip` ne crée pas automatiquement d'environnements virtuels pour installer les paquets.
Pourtant, est-ce vraiment souhaitable ?
Oui, dans le développement, un environnement virtuel sera nécessaire pour installer le projet. Il est d'ailleurs probable qu'un `pip install` exécuté en dehors d'un tel environnement lève une erreur (afin de ne pas mettre le bazar avec les paquets Python installés au niveau système).
Pour le déploiement c'est moins sûr : si on a le contrôle sur le système, sur les versions utilisées, et qu'on ne risque pas de conflit : on peut se passer d'un environnement virtuel.
Mais ensuite, est-ce qu'un environnement virtuel est suffisant ? On en revient au cas précédent des versions multiples.
Quand on travaille sur des versions différentes d'un projet / de ses dépendances, on aura besoin d'un environnement virtuel par ensemble de versions, pour pouvoir passer de l'un à l'autre sans tout réinstaller à chaque fois. D'autant plus si on travaille aussi avec des versions de Python différentes.
Alors à la question de savoir si c'est au gestionnaire de paquets (`pip`) de gérer les environnements virtuels, j'ai envie de répondre que non. Python s'inscrit plutôt dans la philosophie Unix d'avoir un outil qui fait une chose et qui la fait bien.

Conclusion

Alors bon, les fichiers `requirements.txt` : pourquoi pas ?
En l'absence d'un standard de *lock file* ils remplissent en tout cas bien ce rôle et sont assez lisibles. En plus ils sont faciles à gérer pour faire coexister plusieurs versions / environnements d'un projet.

Icône : Logo du projet PyPI sous licence GPL

Liste des abréviations

SaaS Software as a Service. 3