Chapitre 4
Le Raspberry Pi en console avec urwid

1. Introduction

`urwid` est une bibliothèque assez récente dans le paysage UNIX. Elle repose essentiellement sur une autre bibliothèque qui est relativement ancienne, la bibliothèque `curses`. À l'origine de `curses`, on retrouve les premiers développeurs informatiques. À l'époque, la console était le seul moyen d'interagir avec la machine. Ainsi, les premiers développeurs ont très rapidement dû faire face à la nécessité d'assembler un jeu de fonctions pour dessiner des courbes et écrire du texte afin de créer des interfaces utilisateurs. La bibliothèque `curses` était née et les premières interfaces graphiques avec. Cependant, l'apprentissage de cette bibliothèque est déroutant et relativement complexe. De ce fait, ce chapitre se concentrera sur l'étude de `urwid`, une bibliothèque beaucoup plus simple à prendre en main et avec laquelle le développeur peut rapidement créer des interfaces console riches. Dernier point important : contrairement à la bibliothèque `curses`, qui est livrée avec la bibliothèque standard de Python, la bibliothèque `urwid` doit être installée via le gestionnaire de paquets `pip`.

2. urwid, les fondamentaux

La bibliothèque `urwid` s'articule principalement autour de l'utilisation de widgets. Un widget est simplement une classe Python avec des attributs pré-conçus : un champ texte, une liste déroulante, un espace vide, etc. Dans le jargon `urwid` cependant, un widget est en charge d'afficher du texte, de représenter un bouton ou de servir de conteneur pour encapsuler d'autres widgets. Comme avec d'autres frameworks graphiques, la construction d'une application avec `urwid` s'effectue en imbriquant successivement des widgets les uns dans les autres.

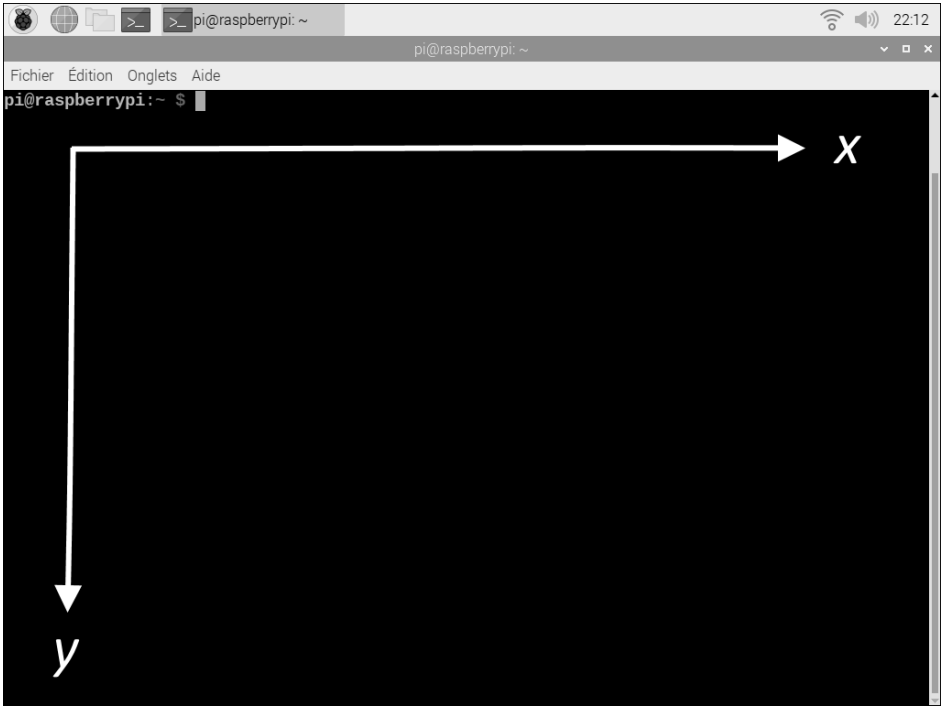
La bibliothèque `urwid` n'est malheureusement pas parmi les bibliothèques standards Python. Vous devez l'installer via l'outil `pip3` :

```
pi@raspberrypi:~ $ sudo pip3 install urwid
```

Voyons les éléments nécessaires à l'écriture d'une application avec `urwid`. Ouvrez une console Python et tapez :

```
>>> import urwid
>>> texte = urwid.Text("Bonjour")
>>> filler = urwid.Filler(texte)
>>> main = urwid.MainLoop(filler)
>>> main.run()
```

Le message "Bonjour" apparaît alors au milieu de l'écran. Le positionnement par défaut des composants d'une application s'effectue toujours de la gauche vers la droite. Un autre aspect à prendre en compte est la mise en forme des éléments dans la console. Pour mieux comprendre, ouvrez un terminal et agrandissez celui-ci au maximum. Imaginez deux axes X et Y. L'axe X est horizontal et commence au coin gauche supérieur. L'axe Y est vertical et commence au même point. Ils représentent respectivement l'axe des abscisses et l'axe des ordonnées :



La disposition de ces axes s'inspire de la bibliothèque `curses` sur laquelle `urwid` repose en grande partie.

Sans plus tarder, voici l'exemple classique du "Hello world" (*Chapitre_4/urwid_1.py*) :

```
#!/usr/bin/env python3
import urwid


def sortie(touche):
    if touche in ("q", "Q"):
        raise urwid.ExitMainLoop()


def main():
    texte = urwid.Text("Hello world!")
    padding = urwid.Padding(texte, align="center", width="pack")
    interface = urwid.Filler(padding, valign="middle")
    main = urwid.MainLoop(interface, unhandled_input=sortie)
    main.run()


if __name__ == "__main__":
    main()
```

Le résultat de l'exécution de ce script est le suivant :

Cet exemple s'articule autour de quatre widgets, qui sont respectivement :

- `Text`
- `Padding`
- `Filler`
- `MainLoop`

Le widget `Text` prend en paramètre une chaîne de caractères et l'affiche. Celui-ci s'imbrique ensuite dans deux autres widgets : `Padding` et `Filler`. Ces widgets remplissent la zone d'affichage en prenant soin de placer les éléments qu'ils contiennent au centre de cette zone, notamment en indiquant les alignements horizontal et vertical via les attributs `align` et `valign`. Enfin, l'élément final est passé au widget `MainLoop`. Parmi ces quatre widgets, `MainLoop` est le plus important, car il permet de démarrer la boucle événementielle `urwid`. Notez que ce widget accepte une fonction pour capturer les touches appuyées au clavier. Dans cet exemple, les touches **q** et **Q** ferment le programme.

Ce bref tour d'horizon pose les bases nécessaires pour mieux appréhender la bibliothèque `urwid`.

3. Projet #1 : une horloge en console

Voyons comment construire une application légèrement plus complexe, à savoir une horloge géante affichée dans la console (*Chapitre_4/urwid_2.py*) :

```
1 #!/usr/bin/env python3
2 import urwid
3 import time
```

La logique du programme est encapsulée au sein de la classe Horloge afin d'en réduire la complexité. L'initialisation de la classe configure l'horloge, définit la palette de couleurs, instancie la boucle événementielle et attache une alarme de 1 seconde à une fonction :

```
6 class Horloge:
7     def __init__(self):
8         self.configurer_horloge()
9         self.palette = [("horloge", "dark blue", "")]
10         self.boucle = urwid.MainLoop(
11             self.horloge,
12             palette=self.palette,
13             unhandled_input=self.quitter)
14         self.boucle.set_alarm_in(1, self.actualiser)
```

Nous étudierons plus tard l'utilité d'une alarme. Mais en premier lieu, examinons la configuration de l'horloge :

```
21      def configurer_horloge(self):
22          text = time.strftime("%H:%M:%S")
23          font = urwid.font.HalfBlock7x7Font()
24          self.texte_horloge = urwid.BigText(text, font)
25          self.horloge = urwid.Filler(
26              urwid.AttrMap(
27                  urwid.Padding(
28                      self.texte_horloge,
29                      "center",
30                      width="clip"
31                  ), "horloge"
32              ), "middle"
33          )
```

La logique du programme consiste à imbriquer des widgets les uns dans les autres. Cependant, deux widgets retiennent notre attention : `BigText` et `HalfBlock7x7Font`. `BigText`, comme son nom l'indique, est un widget utilisé pour afficher un texte large en utilisant une police adéquate. La palette de couleurs à la ligne 9 indique qu'un élément ayant la propriété `horloge` est de couleur bleu foncé. Enfin, la construction de l'horloge est une suite d'appels aux widgets `Filler`, `AttrMap` et `Padding`. À cet instant, l'horloge est pratiquement opérationnelle. Il reste à gérer l'actualisation du temps :

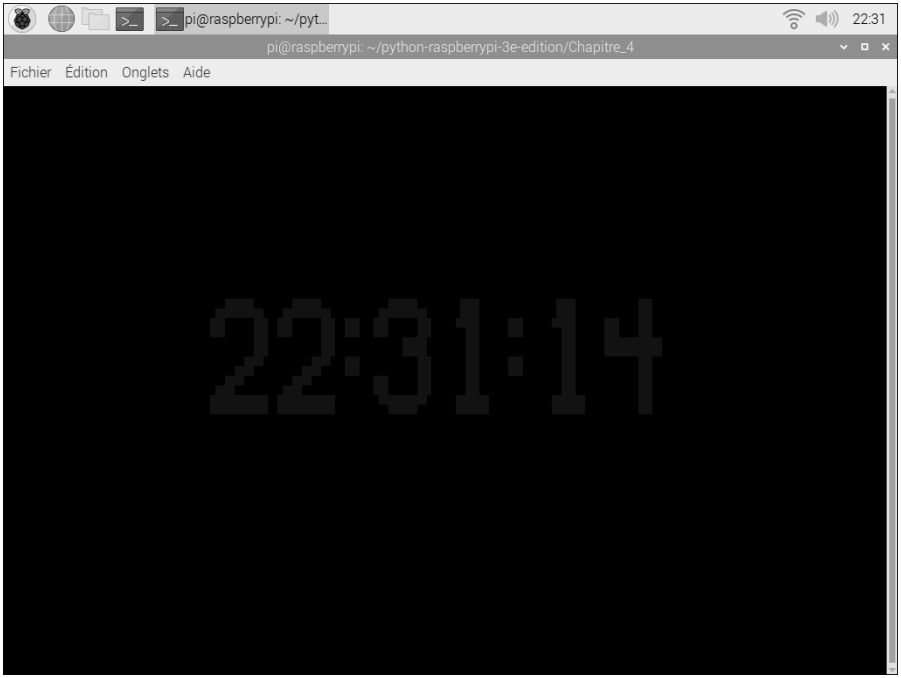
```
16      def actualiser(self, boucle=None, data=None):
17          self.configurer_horloge()
18          boucle.widget = self.horloge
19          boucle.set_alarm_in(1, self.actualiser)
```

La fonction `actualiser` est attachée à une alarme appelée lorsque la seconde est écoulée, à la manière d'un compte à rebours. L'astuce de la mise à jour réside ensuite à reconstruire l'horloge, ce qui incrémentera le temps de 1 seconde, et à attacher à nouveau la fonction `actualiser` à une alarme de 1 seconde. Ainsi, l'horloge est mise à jour automatiquement à chaque seconde !

Enfin, le programme se lance lorsque la fonction `demarrer` de notre classe est appelée :

```
35      def demarrer(self):
36          self.boucle.run()
37
38      def quitter(self, touche):
39          if touche in ("q", "Q"):
40              raise urwid.ExitMainLoop()
41
42
43  if __name__ == "__main__":
44      horloge = Horloge()
45      horloge.demarrer()
```

Et voici le résultat en image :



Vous pouvez maintenant lire l'heure depuis la ligne de commande !

4. Projet #2 : un navigateur de fichiers en console

Pour continuer l'étude de la bibliothèque, voici un programme légèrement plus complexe, à savoir un navigateur de fichiers (*Chapitre_4/urwid_3.py*) :

```
1 #!/usr/bin/env python3
2 import urwid
3 import os
```

Comme dans l'exemple précédent, toute la logique de création du navigateur de fichiers est encapsulée dans une classe appelée `NavigateurFichiers`. À sa création, cette classe initialise un certain nombre de variables et procède à un listing des fichiers du répertoire courant :

```
6 class NavigateurFichiers(urwid.WidgetPlaceholder):
7     def __init__(self, chemin):
8         self.chemin_courant = chemin
9         self.chemin_prec = None
10         self.liste_fichiers = sorted(os.listdir(self.chemin_courant))
11         if self.chemin_courant != "/":
12             self.liste_fichiers.insert(0, "..")
13         super(NavigateurFichiers, self).__init__(self.creer_contenu())
```

La variable `liste_fichiers` est stockée comme membre de l'instance afin d'être réutilisée plus tard. Le code prend aussi soin d'ajouter le chemin `..` en début de liste pour que l'utilisateur puisse remonter d'un répertoire lors de la navigation. Voyons en détail la fonction `creer_contenu` :

```
30     def creer_contenu(self):
31         contenu = []
32         for fichier in self.liste_fichiers:
33             bouton = urwid.Button(fichier)
34             if os.path.isdir(os.path.join(self.chemin_courant, fichier)):
35                 urwid.connect_signal(
36                     bouton, "click", self.maj_contenu, fichier
37                 )
38                 contenu.append(urwid.AttrMap(
39                     bouton, "rep", focus_map="rep_focus")
40                 )
41             else:
42                 contenu.append(urwid.AttrMap(
43                     bouton, None, focus_map="fichier"))
44         return urwid.AttrMap(
45             urwid.LineBox(
46                 urwid.ListBox(
47                     urwid.SimpleFocusListWalker(contenu)),
48                 self.chemin_courant), "contenu"
49         )
```

Cette fonction parcourt la liste des éléments du répertoire courant. Lorsqu'il s'agit d'un répertoire, un bouton est créé et, lorsqu'il est sélectionné, appelle la fonction `maj_contenu`. Si l'élément est un fichier, un simple bouton est créé sans aucune fonction particulière attachée. Ces boutons sont à chaque fois ajoutés dans une liste : `contenu`. La fin de l'exécution de la fonction retourne alors quatre widgets imbriqués les uns dans les autres :

- `AttrMap`
- `LineBox`
- `ListBox`
- `SimpleFocusListWalker`

Chapitre 6
Développement web en Python

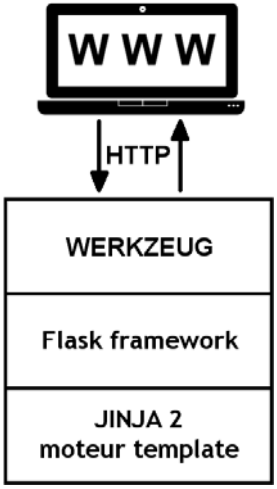
1. Présentation de Flask

Flask est un micro framework de développement web écrit en Python.

Logo du projet Flask

Allant à contre-pied d'autres solutions de développement web, Flask est livré avec le strict minimum, à savoir :

- un moteur de template (Jinja 2)
- un serveur web de développement (Werkzeug)
- un système de distribution de requête compatible REST (dit RESTful)
- un support de débogage intégré au serveur web
- un micro framework doté d'une très grande flexibilité
- une très bonne documentation



Structure du micro framework Flask

Disposer d'un micro framework implique donc l'absence de certains éléments *out-of-the-box* tels que :

- une solution d'authentification
- le support de base de données ou un ORM
- la gestion sécurisée de formulaires HTML
- une interface d'administration

Cela n'est cependant pas un frein, car la grande flexibilité du micro framework permet l'adjonction d'une pléthore d'extensions Flask couvrant ces manques apparents, extensions dont certaines sont décrites plus loin dans ce chapitre.

1.1 Pourquoi Flask ?

Pour commencer, parce qu'il utilise Python, ce qui reste dans les objectifs du présent ouvrage.

Flask n'est cependant pas le seul framework de développement web Python disponible. Il existe d'autres alternatives comme Bottle (http://bottlepy.org), Django (https://www.djangoproject.com/) ou CherryPy (https://cherrypy.org).

Django est une perle dans le domaine du développement web Python. Il dispose de nombreuses fonctionnalités et rien n'y manque pour réaliser un développement de niveau professionnel. La contrepartie, c'est une solution plus lourde à mettre en œuvre avec une courbe d'apprentissage plus longue. Django sera plutôt réservé à des projets d'envergure.

Flask et Bottle quant à eux permettent de produire les premières pages en moins de 5 minutes (installation comprise). Ces solutions plus légères en termes de ressources conviendront mieux à notre projet sur Raspberry Pi.

Ce qui a guidé le choix entre Bottle et Flask est la popularité, la documentation et les extensions de Flask. L'expérience ayant par ailleurs démontré que ce dernier permet de réaliser des développements de qualité professionnelle.

1.2 La flexibilité de Flask

Ce qui rend Flask si populaire et attractif, c'est sa grande flexibilité. Dès le début du développement, le micro framework a intégré de nombreux mécanismes facilitant le développement d'extensions.

Ainsi, Flask intègre des mécanismes comme les hooks, extend (développement d'extension), Signals (notification par signaux), dérivation de la classe Flask et middleware (introduction de corrections entre le serveur HTTP et l'application Flask).

Un document est disponible en ligne pour découvrir ces différents mécanismes :
http://flask.pocoo.org/docs/1.0/becomingbig/

1.3 Les nombreuses extensions Flask

Ces dernières années ont vu l'apparition de très nombreuses extensions permettant d'ajouter, au cas par cas, les fonctionnalités nécessaires sur le micro framework.

Un catalogue des extensions est disponible sur le lien suivant :
http://flask.pocoo.org/extensions/

Les extensions sont hébergées sur des pages PyPI (*Python Package Index*), ce qui permet de les installer avec l'utilitaire `pip` comme n'importe quel autre paquet Python.

Le catalogue d'extensions est une liste modérée d'extensions Flask suivant les recommandations du guide des extensions Flask. De nombreuses ressources sont également disponibles sur Internet.

La liste suivante reprend les extensions les plus intéressantes. Savoir quelles extensions existent permet d'éviter de nombreux efforts et des recherches inutiles.

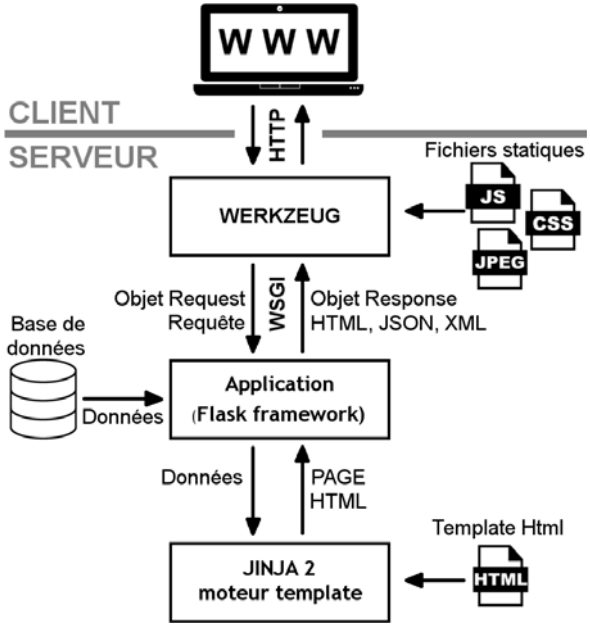
- Flask-Login : permet d'ajouter la gestion de sessions utilisateur à Flask. Cette extension prend en charge le login, la déconnexion et mémorisation de la session utilisateur durant un certain temps.
- Flask-User : gestion personnalisable des utilisateurs (enregistrement, confirmation, login, changement de mot de passe, etc.).

- Flask-Themes : permet à une application Flask de supporter plusieurs thèmes.
- Flask-WTF : permet de simplifier le code nécessaire à la gestion de la saisie et de la validation de données sur des pages HTML. Avec WTForm, les caractéristiques et les validations des champs de données sont définies dans le script python tandis que le template s'occupe du rendu. Un must !
- Flask-Uploads : permet de prendre en charge le téléversement et le stockage de fichiers sur le serveur. Cette extension permet également de télécharger les fichiers stockés sur le serveur.
- Flask-Exceptionnal : permet d'ajouter le support d'Exceptionnal à une application Flask. Exceptionnal (https://pythonhosted.org/Flask-Exceptional/) collecte les erreurs et leurs informations associées survenant dans les applications et les rend disponibles en temps réel sur un site sécurisé.
- Flask-SQLAlchemy : apporte le support pour SQLAlchemy à Flask. SQLAlchemy est un ORM (*Object Relational Mapper*) très puissant supportant de nombreuses bases de données. Un must !
- Flask-Babel : ajout du support d'internationalisation i18n et localisation l10n à une application Flask en s'appuyant sur le projet Babel. La fonction `gettext()` et l'utilitaire `pybabel` permettent de traduire facilement le contenu de l'application.
- Flask-Bcrypt : algorithme de hashing utilisé pour éviter le stockage des mots de passe dans la base de données.
- Flask-RESTful : permet de réaliser rapidement des API REST et d'obtenir des ressources au format JSON.
- Flask-Restless : permet de fournir une API RESTful pour des modèles de base de données en utilisant SQLAlchemy.
- Flask-Mail : permet à une application Flask d'envoyer très facilement des e-mails.
- Flask-Creole : permet d'utiliser le langage de marquage Wikicréole dans les applications Flask. Similaire à MarkDown, Wikicréole est une syntaxe universelle pour les wikis destinée à transférer facilement du contenu entre différents moteurs wiki.
- Flask-Dance : permet d'utiliser la délégation d'autorisation OAuth dans un projet Flask. OAuth est un protocole libre qui permet aux utilisateurs d'un site A de donner l'autorisation à un tiers B (site ou logiciel) d'utiliser les données personnelles stockées sur le site A en son nom (généralement via une API). Le mécanisme OAuth permet de protéger les pseudonyme et mot de passe utilisateur du site A puisqu'ils ne sont jamais communiqués au tiers B. Voir aussi Flask-OpenID et Flask-OAuth.
- Flask-FlatPages : offre une collection de pages statiques à une application Flask. Les pages sont stockées sous forme de fichiers plats (en opposition aux pages générées à partir d'une base de données). Flask-FlatPages s'utilise en conjonction avec Frozen-Flask (https://pythonhosted.org/Frozen-Flask/).

- Flask-Genshi : permet à l'application Flask de supporter le langage de template Genshi pour HTML, XML et texte. Genshi met surtout l'accent sur la chaîne de traitement XML, ce qui renforce la conformité des contenus HTML/XML produits. En effet, la plupart des langages de template, y compris Jinja, ne traitent que des flux de caractères pour y réaliser des substitutions sans assurer la conformité du rendu.

1.4 Flask plus en détail

Le schéma suivant présente le fonctionnement général du micro framework Flask et les différents éléments intervenant dans la chaîne de production du contenu.



Fonctionnement détaillé du micro framework Flask

Voici une description des différents éléments composant le mini framework Flask :

- Werkzeug
- WSGI
- Jinja
- la base de données

1.4.1 Werkzeug

Werkzeug est un serveur web WSGI écrit en Python.

Il reçoit les requêtes HTTP et fait le nécessaire pour décoder les demandes et envoyer les réponses en retour. Werkzeug s'occupe des problématiques de gestion des sockets et connexions, traitement des tâches en parallèles, prise en charge des aspects sécuritaires (au niveau HTTP), etc.

Werkzeug est également un serveur compatible WSGI. Il est donc capable de transmettre la requête à l'application en suivant le protocole WSGI puis de récupérer la réponse et de la transmettre au client via HTTP.

Bien que ce projet ait débuté comme une simple boîte à outils WSGI, Werkzeug est aujourd'hui l'un des modules WSGI les plus puissants existant sur le marché.

Werkzeug inclut :

- la gestion complète des requêtes et des réponses HTTP,
- un support unicode,
- un débogueur interactif (utilisant JavaScript),
- un contrôle de l'en-tête HTTP,
- un contrôle du cache (dans l'en-tête),
- la gestion des cookies,
- la transmission de fichiers (téléchargement et téléversement),
- un système très puissant de routage des URL,
- de nombreux greffons produits par la communauté,
- un support WSGI compatible avec Python 2.7 et 3.3.

Dans le cadre d'une solution Flask, Werkzeug :

- transmet les requêtes HTTP à l'application Flask,
- délivre directement les ressources statiques telles que les fichiers CSS, JavaScript, les images.

1.4.2 WSGI

WSGI (*Web Server Gateway Interface*) est une norme et un protocole de communication qui définissent comment un serveur web peut interagir avec une application Python pour envoyer des requêtes et recevoir des réponses.

WSGI est une norme pour serveur web comme l'est FastCGI, SCGI, ou WebSocket destinée au langage Python.

Un serveur web supportant WSGI doit être capable de transformer une requête HTTP en objet Python en suivant la norme WSGI. Il doit également être capable de recevoir la réponse sous forme d'objet Python pour renvoyer celle-ci vers le client web.

L'application Python (comme Flask) doit donc être capable de supporter les spécificités du protocole WSGI pour renvoyer des réponses par l'intermédiaire du serveur web via WSGI. L'application doit pouvoir recevoir une requête HTTP sous forme d'un objet Python (à la norme WSGI) et renvoyer la réponse avec un objet Python tel que défini dans la norme.

Tous les serveurs compatibles WSGI le sont avec toutes les applications compatibles WSGI. La norme WSGI assure que ces différents éléments sont interchangeables.

Avantages de WSGI

- Pas besoin de prendre en charge le support HTTP dans l'application.
- Normalisation de l'interface entre l'application et le serveur web. Il est possible de changer de serveur web si cela est nécessaire.
- Le module WSGI reste chargé en mémoire (économie de ressources, meilleur temps de réponse).
- WSGI permet l'interfaçage direct d'un code Python. Une application rudimentaire ne nécessite pas l'utilisation d'un quelconque framework pour produire une réponse pour le web serveur.

Inconvénients du WSGI

- WSGI ne supporte pas encore la norme WebSocket, devenue rapidement populaire car elle permet au serveur de pousser des données vers le client web en fonction des besoins. WebSocket permet une communication dans les deux sens. Il existe cependant des solutions Python alternatives telles que Tornado ou Twisted.
- Certains serveurs comme Nginx ou lighttpd ne supportent pas WSGI, ce qui empêche l'application Python de communiquer directement avec ces serveurs web. Il existe cependant des solutions middlewares, comme par exemple gunicorn qui permet de réaliser une installation nginx (serveur web) <==> gunicorn (middleware WSGI) <==> Flask / Django / ... (Python).