Licence informatique

La Rochelle Université

Création de plug-ins - GALACTIC CLI

Alexy Lafosse

2024

Table des matières

Liste des figures

Remerciements

Je souhaite principalement remercier mon maître de stage, Monsieur Christophe DEMKO, maître de conférences à l'université de La Rochelle, pour avoir accepté de me prendre en tant que stagiaire, ainsi que pour toute l'aide qu'il m'a apportée durant mon stage. Il m'a permis d'apprendre énormément de choses, que ce soit sur le langage Python ou sur l'utilisation d'outils tels que Poetry ou GitLab. Il m'a également permis de faire évoluer ma vision d'un projet, ce qui m'a permis de découvrir de nouvelles possibilités.

Je souhaite ensuite remercier Monsieur Yacine GHAMRI-DOUDANE, directeur du laboratoire L3i, pour m'avoir accepté au sein du laboratoire pour ce stage.

Je remercie également ma tutrice, Madame Karell BERTET, professeure des universités à l'université de La Rochelle, pour avoir accepté de m'accompagner durant ce stage.

Résumé en français

Dans le cadre de la fin de mon cursus de Licence en Informatique à La Rochelle Université, j'ai eu l'occasion de faire un stage au laboratoire de recherches L3i (Laboratoire Informatique, Image et Interaction) de l'université de La Rochelle sous la supervision de Monsieur Christophe Demko.

Ce stage a une durée de 33 jours, et ce rapport est incomplet du fait qu'il est écrit à la moitié de celui-ci. Il contient une présentation du laboratoire L3i et de la plateforme GALACTIC. On y retrouve la structure de la commande `galactic` ainsi que les avancées que j'ai faites sur la création d'extensions pour le framework de cette commande. En particulier le développement de l'extension `io-data` permettant de créer des plug-ins pour la plateforme GALACTIC. On retrouve également une perspective de création de l'extension *meta*, qui sera développée dans la suite de mon stage.

Abstract in English

As part of completing my Bachelor's degree in Computer Science at La Rochelle University, I had the opportunity to intern at the L3i research laboratory (Laboratoire Informatique, Image et Interaction) of La Rochelle University under the supervision of Mr. Christophe Demko.

This internship lasts for 33 days, and this report is incomplete as it is written halfway through. It contains a presentation of the L3i laboratory and the GALACTIC platform. It outlines the structure of the `galactic` command as well as the progress I have made on creating extensions for this command framework. Specifically, the development of the `io-data` extension which allows the creation of plugins for the GALACTIC platform is discussed. There is also a perspective on creating the *meta* extension, which will be developed further in the remainder of my internship.

1 Introduction

1.1 Laboratoire L3i

Figure 1: Logo de l'entreprise

Le Laboratoire Informatique, Image et Interaction (L3i) a été crée en 1993. Il s'agit d'un laboratoire de recherche de Sciences du Numérique de l'Université de La Rochelle, il associe les chercheurs en Informatique de l'IUT ainsi que les chercheurs du Pôle Sciences de l'Université de La Rochelle. Il est dirigé par Monsieur Yacine GHAMRI-DOUDANE.

Le laboratoire L3i fait parti de réseaux de recherches régionaux (Fédération CNRS MIRES, ERT "Interactivité Numérique"), nationaux (GDR I3 et GDR IRIS) et internationaux (IAPR) dans les secteurs de visibilité de son action scientifique, autour d'un projet stratégique lié à la gestion intelligente et interactive des contenus numériques. Cela est renforcé grâce à une politique volontariste de participation ou de pilotages de projets de recherches labélisés (ANR, PCRD, …), au sein desquels le laboratoire occupe couramment une position de leadership.

Le Laboratoire est également membre de l'Alliance Big data, lancée en 2013, pour favoriser le développement en France de nouveaux services et projets dans ce domaine.

Ses travaux sont menés en partenariat avec une dizaine de centres de recherches nationaux (dont l'INRIA, institut spécialisé). Le L3i entretient par ailleurs des liens privilégiés avec de nombreux centres de recherche à travers le monde. Il est également engagé dans près d'une vingtaine de partenariats industriels sur l'ensemble du territoire français.

Le laboratoire du L3i est divisé en 3 équipes:

- Modèles et Connaissances
- Images et Contenus
- Dynamique des systèmes et Adaptativité

Mon stage se déroule au sein de l'équipe Modèles et Connaissances et plus précisément autour du projet GALACTIC de cette équipe.

1.2 Plateforme GALACTIC

GALACTIC est l'acronyme de GAlois LAttices Concept Theory Implicational system and Closures. L'objectif de la plateforme est de pouvoir mettre en place un ensemble d'outils permettant de travailler

sur l'Analyse Formelle des Concepts ainsi que sur la Théorie des treillis. La plateforme GALACTIC est écrit en Python et est structurée de la façon suivante:

- Un noyau, représenté par le cœur de la fleur, qui contient les opérations de base et structures de données, et qui implémente un nouvel algorithme (NextPriorityConcept) inspiré des pattern structures.
- Des plugins, représentés par les pétales de la fleur:
  - Caractéristiques : Booléen, Numérique, Catégorique, String, Séquentielle, Chain, Triadic. Ils permettent de définir les caractéristiques.
  - Descriptions : Booléen, Logique, Catégorique, String (regex), String (distance), Chain, Séquentielle, Séquentielle (distance), Triadic. Ils définissent les prédicats et les espaces de description utilisés pour représenter et définir les données avec précision.
  - Stratégies : Booléen, Logique, Catégorique, Numérique, String, String (distance), Chain, Séquentielle, Séquentielle (distance), Triadic. Ils définissent la manière utilisée pour explorer les données, il utilise des descriptions pour générer des prédécesseurs pour chaque concept dans le treillis.
  - Mesures : predecessor Cardinality, sucessor Cardinality, Confidence. Ce sont des paramètres des stratégies de filtrage prédéfinis dans la librairie core.
  - Data Readers : Ils sont utilisés pour lire différents types de fichiers de données. Le moteur de base détecte le type de fichier en utilisant son extension. Les plugins Data Readers existants sont: YAML, JSON, CSV, TOML, INI, TXT, SLF, DAT, CXT.
  - Localization: Ils sont utilisés pour traduire les applications dans d'autres langues.
- Des applications, représentées par les feuilles de la fleur, qui sont développées afin d'utiliser la librairie de la plateforme. Ce sont des interfaces pour l'utilisateur:
  - GALACTIC Laser : pour construire les treillis et explorer les données.
  - GALACTIC Explorer : pour explorer de façon interactive les treillis contruits.
  - GALACTIC Ruler : pour extraire les règles d'implication.
  - GALACTIC Fire : pour exécuter un système de règles.

Note
La bulle au dessus du serpent indique les différentes versions de Python avec lesquelles la plateforme GALACTIC est compatible.

Note
Au niveau des racines on peut voir les différentes langues disponibles sur la plateforme.

Tip
La présentation de la plateforme GALACTIC est issue du diaporama[1] de présentation de l'architecture de la plateforme ainsi que de la vidéo[2] faites par M. Demko.

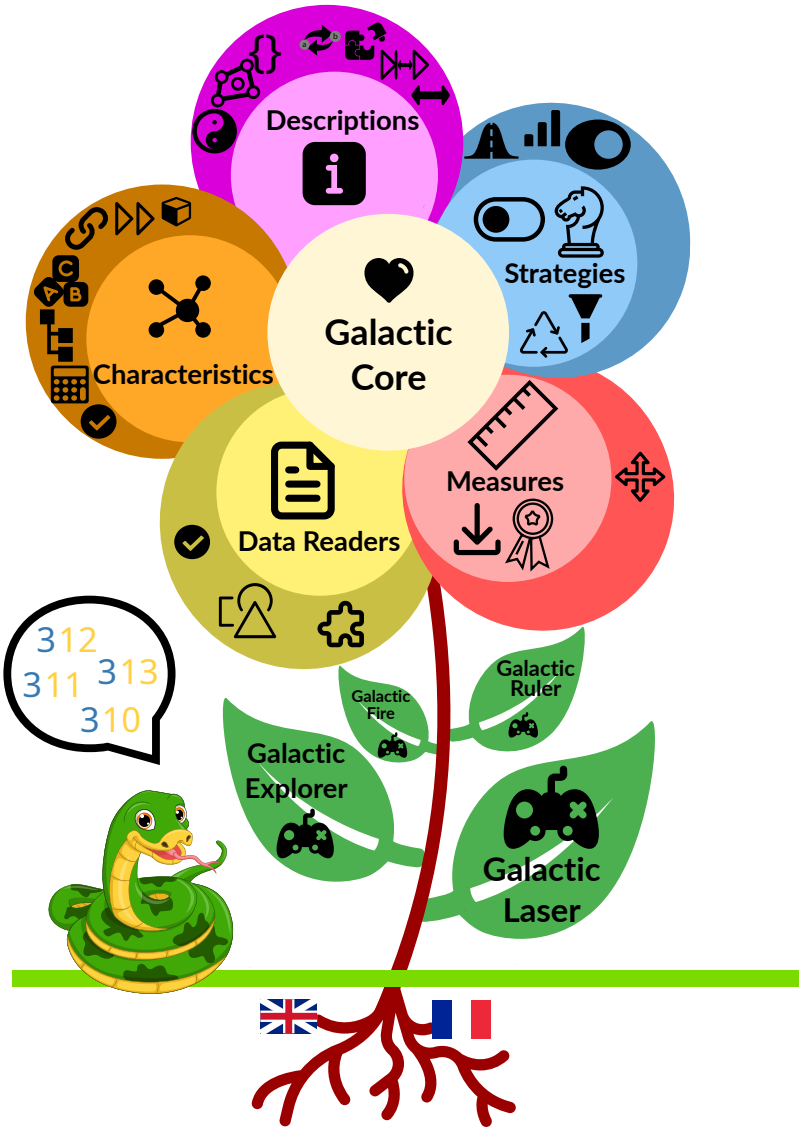

Figure 2: Architecture GALACTIC

1.2.1 Commande `galactic`

À cette structure vient s'ajouter une **interface en ligne de commande**, avec pour coeur la commande `galactic`. Cette commande, seule, ne peut rien faire. Cependant, elle est extensible, c'est à dire que l'on peut lui rajouter des fonctionnalités sous forme de plug-in.

[1]https://galactic.univ-lr.fr/slides/architecture/Galactic-Architecture-slides-20min-complete.pdf
[2]https://pod.univ-lr.fr/video/1942-architecture-de-galactic/

Figure 3: Schéma Framework-core

Par exemple, l'application *GALACTIC Laser* peut être une extension de cette commande. La commande permettant cet effet n'a pas encore été créé mais la structure permet un tel dévelopement. Par exemple, la commande pourrait être de la forme `galactic framework laser run`, ce qui exécuterait l'application.

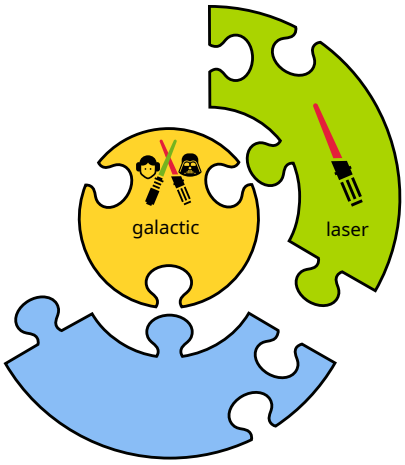

Figure 4: Schéma Framework-laser

Une de ces extensions, le *framework*, est une extension elle même extensible par plug-in. Initialement, elle ne contient que la commande de base, mais ses extensions viennent hériter de cette commande afin d'apporter des fonctionnalités. Chaque extension a pour objectif d'être un générateur de plug-in pour GALACTIC. Prenons l'exemple de l'extension `io-data`. Elle s'appelle avec la commande

`galactic framework io data create`. Son objectif est de générer des squelettes de plug-in permettant la lecture et/ou écriture d'un fichier (plug-in Data Readers).

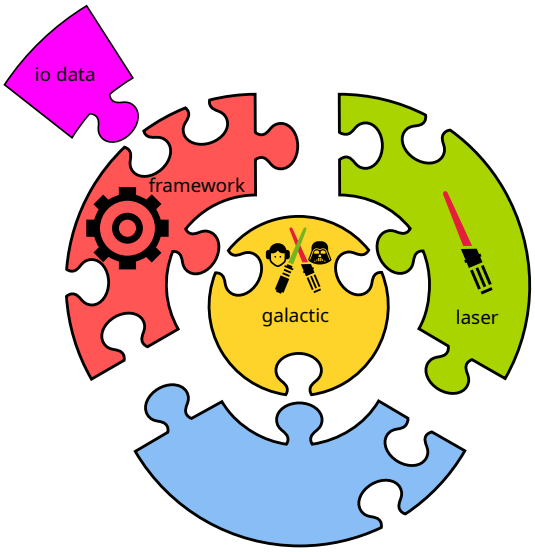

Figure 5: Schéma Framework-io-data

Ce framework a une particularité car il est auto-extensible, c'est à dire qu'une de ses extensions permet de créer d'autres extensions. Cette extension *meta* génère un squelette d'extension pour le framework, ce qui permet à des développeurs de créer une nouvelle catégorie de plug-in.

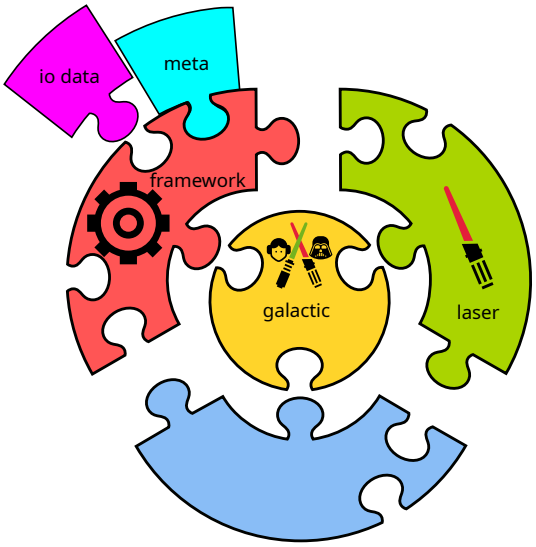

Figure 6: Schéma Framework-meta

Note
L'extension *meta* est rattaché au framework car elle s'installe avec le même paquet Python.

L'objectif de ce *framework* est donc de simplifier la création de nouveau plug-in de GALACTIC.

1.3 Objectif du stage

L'objectif du stage est de créer des plug-ins du framework en ligne de commande de GALACTIC.

Au début du stage, le framework ne possédait aucun plug-in, seulement la commande de base, qui ne pouvait rien accomplir seule. L'objectif est donc, d'ici la fin de mon stage, de créer le plus grand nombre possible de plug-ins.

1.4 Organisation

Durant ma première semaine de stage, j'ai été occupé à comprendre :

- les objectifs et enjeux du stage
- la plateforme GALACTIC
- les outils nécessaire au développement du framework et de ses plug-ins.

Après avoir bien compris les objectifs et les outils, j'ai commencé à travailler sur le framework. J'ai fais dans l'ordre :

- la création d'une première version du plug-in `io data`
- l'amélioration de la commande de base `core`
- l'amélioration du plug-in `io data`
- la préparation du rapport

Après le dépôt du rapport, j'envisage de faire :

- la préparation de la soutenance
- la création du plug-in `meta`
- la création du plus grand nombre possible de plug-in implémentant d'autres fonctionnalités de Galactic

2 Préparation

2.1 Compréhension et mise en place des outils de développement

Lorsque l'on travaille sur un projet, il est important de produire un travail propre, et le codage n'y fait pas exception. Pour simplifier le travail des développeurs, de nombreux outils ont été créés. Dans le cadre du framework sur lequel je travaille, plusieurs de ces outils ont été utilisés. Ma première mission a été de les comprendre et de savoir les utiliser. Je vais donc les présenter ci-dessous.

2.1.1 Gitflow

Gitflow est un workflow Git qui consiste à créer plusieurs branches avec des objectifs différents :

- `main`

- `develop`
- `feature`
- `release`
- `hotfix`

La branche principale est la branche `main`, qui contient les versions publiques de l'application. Créée à partir de la branche `main`, la branche `develop` contient le projet dans son ensemble, avec toutes les évolutions qui ont pu se produire. Depuis la branche `develop`, il est possible de créer des branches `feature`, chacune dédiée au développement d'une fonctionnalité. Lorsque l'on souhaite mettre à jour la version publique de l'application se trouvant sur la branche `main`, on utilise une branche `release` dans laquelle seules des tâches liées à la livraison du produit, comme la correction de bugs ou la documentation, sont effectuées. Enfin, il y a les branches `hotfix`, qui permettent la maintenance et la correction des bugs d'une version du produit final. Elles sont donc créées à partir de la branche `main` si celle-ci nécessite des corrections.

En conclusion, l'utilisation du workflow Gitflow est très utile pour améliorer la performance et la maintenabilité d'une application grâce à son système de branches. De plus, il permet d'accroître l'efficacité et la rapidité des équipes en facilitant le travail simultané. Par exemple, lors de la mise à jour du produit via une branche `release`, une autre équioetrye peut toujours travailler sur une nouvelle fonctionnalité.

2.1.2 Black

Black est un formateur de code Python qui permet aux développeurs de coder sans se soucier des conventions de codage Python (PEP8). En effet, lorsque le développeur utilise Black, celui-ci reformate et réadapte les fichiers Python en suivant ces conventions. Le développeur a donc simplement besoin d'exécuter Black après avoir terminé de coder. Black permet ainsi d'être plus efficace et rapide dans le processus de développement.

2.1.3 Slotscheck

Slotscheck est une extension Python qui vérifie l'utilisation des `__slots__` dans un fichier Python et signale les éventuels problèmes.

2.1.4 Flake8

Flake8 est une extension Python conçue pour tester la qualité du code. Elle utilise trois outils :

- PyFlakes, qui vérifie la syntaxe du code
- Pycodestyle, qui vérifie si le code respecte les conventions de style PEP8
- Le script McCabe de Ned Batchelder, qui calcule la complexité cyclomatique du code, permettant d'identifier les parties de code potentiellement difficiles à comprendre ou à maintenir en raison de leur complexité

2.1.5 Ruff

Ruff est également une extension Python destinée à tester la qualité du code, mais elle présente l'avantage d'être beaucoup plus rapide à exécuter.

2.1.6 Mypy

Mypy est un outil de vérification de types statiques. Il permet de s'assurer que les variables et les fonctions sont utilisées correctement. Pour que Mypy puisse détecter les erreurs de type, il est nécessaire d'annoter le code en précisant les types des variables et des valeurs de retour.

Exemple d'annotation :

```python
def print_hello(name: str) -> str:
    return 'Hello ' + name
```

2.1.7 Pylint

Pylint est un analyseur de code statique, c'est-à-dire qu'il teste le code sans l'exécuter. Il cherche des erreurs dans le code et peut suggérer des modifications au développeur.

2.1.8 Darglint

Darglint est un vérificateur de documentation qui s'assure, entre autres, que les paramètres des fonctions sont correctement définis dans la documentation. Il suit les directives du Google Python Style Guide, du Sphinx Style Guide ou du Numpy Style Guide.

Sur la plateforme GALACTIC, c'est la documentation Sphinx qui est utilisée.

2.1.9 Doc8

Doc8 est un vérificateur de documentation pour Sphinx.

2.1.10 Sphinx

Sphinx est un générateur de documentation. Il permet de produire une documentation propre et élégante pour des projets, principalement en Python, en utilisant le langage ReStructuredText (.rst).

2.1.11 Teyit

Teyit est un analyseur de tests unitaires. Il vérifie la qualité des tests unitaires.

2.1.12 Refurb

Refurb analyse le code Python et propose des suggestions d'amélioration pour moderniser le code.

2.1.13 Pytest

Pytest est un framework Python qui simplifie et automatise la mise en place de tests.

Lors de son exécution, pytest recherche de manière récursive tous les fichiers dans le dossier spécifié en paramètre commençant par "test_" ou se terminant par "_test", puis exécute les tests présents dans ces fichiers et signale les erreurs éventuelles.

2.1.14 Tox

Tox simplifie la gestion des environnements virtuels et la création de tests. Il permet d'exécuter plusieurs commandes en une seule. Par exemple, si l'on souhaite exécuter plusieurs outils d'analyse de code tels que Ruff, Flake8 et d'autres, il est possible de le faire avec une seule commande tox et un fichier de configuration `tox.ini`, qui exécute ces outils les uns après les autres.

2.1.15 Hooks

Les hooks sont des scripts liés à des actions Git, comme avant ou après un commit, ou encore avant ou après un merge. La plupart des outils énumérés ci-dessus ont des hooks associés. Dans notre cas, ces hooks seront utilisés avec l'outil qui suit. Ils nous permettront d'exécuter nos outils avant chaque commit.

2.1.16 Pre-commit

Pre-commit est un framework qui permet d'exécuter une liste de hooks automatiquement avant chaque commit. Cette liste de hooks doit être fournie dans un fichier de configuration `.pre-commit-config.yaml`. Dans notre cas, cela nous permet d'exécuter tous les outils souhaités avant de réaliser un commit afin de vérifier qu'il n'y a pas d'erreurs et que tout est conforme.

2.1.17 Gitlab runners

Gitlab runner est une application qui exécute des actions dans des pipelines. Ces actions sont définies dans un fichier de configuration `.gitlab-ci.yml`.

Dans notre cas, nous l'utilisons pour effectuer différentes actions telles que les tests sur la construction du fichier wheel, la vérification de la couverture des tests effectués, la construction de la documentation, ainsi que d'autres tâches.

2.2 Compréhension et mise en place des outils liés au framework

2.2.1 Poetry

Poetry est un outil de gestion de dépendances et de paquets pour Python. Il peut également créer un environnement virtuel.

Le framework Poetry est utilisé dans la plateforme GALACTIC car il simplifie la gestion des dépendances et des environnements virtuels avec Python.

Poetry utilise un fichier `pyproject.toml` comme fichier de configuration, remplaçant souvent le fichier `setup.py`.

Plusieurs commandes sont importantes :

```
poetry shell
```

Note
Cette commande crée un environnement virtuel lié au dossier courant la première fois, et s'y connecte s'il existe déjà. Pour en sortir, il faut utiliser la commande `exit`.

```
poetry install
```

Note
Cette commande installe les dépendances précisées dans le fichier `pyproject.toml`. Elle crée un fichier `poetry.lock` s'il n'existe pas déjà, contenant les informations sur les versions exactes des dépendances. Si un fichier `poetry.lock` est déjà présent, alors les dépendances sont installées avec les versions spécifiées.

```
poetry check
```

Note
Cette commande vérifie le contenu du fichier `pyproject.toml` et s'assure qu'il n'y a pas d'erreurs de cohérence ou de compatibilité entre les dépendances.

```
poetry lock
```

Note
Cette commande met à jour le fichier `poetry.lock` à partir du `pyproject.toml`.

```
poetry build
```

Note
Cette commande crée le fichier source `.tar.gz`, qui contient les fichiers source du projet, ainsi que le fichier wheel `.whl`, permettant l'installation rapide du projet.

```
poetry run
```

Note
Cette commande permet d'exécuter des scripts ou des commandes dans l'environnement virtuel créé par Poetry. Si l'environnement est déjà activé avec la commande `poetry shell`, l'utilisation de `poetry run` n'est pas nécessaire.

2.2.1.1 Poeblix

Poeblix est une extension de Poetry qui permet, entre autres, d'inclure des fichiers autres que Python dans le fichier wheel, ce qui est très utile pour la génération de templates.

Des commandes sont utiles :

```
poetry up
```

Note
Met à jour les dépendances installées par poetry.

```
poetry blixbuild --no-lock
```

Note
Cette commande fait la même chose que la commande poetry build, mais en incluant des fichiers autres que Python dans le fichier wheel par le biais d'une configuration dans le fichier `pyproject.toml`.

Warning
L'option `--no-lock` est nécessaire pour éviter des conflits. Avec cette option, la commande exclut du fichier wheel les dépendances du fichier `poetry.lock`.

```toml
[tool.blix.data]
data_files = [
    {
        destination = "share/data/",
        from = [ "data_files/test.txt", "data_files/anotherfile" ]
    },
    {
        destination = "share/data/threes",
        from = [ "data_files/athirdfile" ]
    }
]
```

2.2.1.2 Poetry-dynamic-versioning

L'extension *poetry-dynamic-versioning* permet de mettre à jour la version du projet dynamiquement en utilisant des informations telles que les commits Git.

2.2.2 Cléo

Cléo est un framework Python qui simplifie la création d'interfaces de lignes de commandes CLI. Il joue un rôle central dans le fonctionnement de la commande `galactic` en nous permettant de créer facilement nos lignes de commande.

Voici rapidement son fonctionnement.

Tout d'abord, il est nécessaire d'avoir une classe `Application` hérité de l'application de Cléo. Cette classe crée une instance d'une application où les commandes seront stockées.

Avec cette application, la commande de base `galactic` est créée. Grâce à la classe parente de Cléo, cette commande possède automatiquement des commandes et options de base, telles que `help` et `list`. Pour les options, on peut retrouver `--version` ou encore `--verbose`.

Note

La commande `galactic` seule affiche la liste des commandes et options disponibles. Cela revient à faire `galactic list`.

```
xy galactic-apps-cli-framework-io-data % galactic
GALACTIC main application (version 0.0.0.post41.dev0+c325698)

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display help for the given command. When no command
is given display help for the list command.
  -q, --quiet           Do not output any message.
  -V, --version         Display this application version.
      --ansi            Force ANSI output.
      --no-ansi         Disable ANSI output.
  -n, --no-interaction  Do not ask any interactive question.
  -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for normal out
put, 2 for more verbose output and 3 for debug.

Available commands:
  help                      Displays help for a command.
  list                      Lists commands.

 framework
  framework io data create  Create a data-io plugin
  framework self create     Create a plugin for the framework itself
```

Figure 7: Exemple - Commande galactic

Ensuite, une commande principale `framework` a été créée mais n'est pas enregistrée. Cela signifie que toutes les autres commandes vont hériter de celle-ci, commençant donc toutes par `galactic framework ...`, mais la commande `galactic framework` en elle-même n'existe pas. C'est ce que j'ai appelé dans l'introduction, *la commande de base du framework*.

À partir de là, chaque plug-in implémente les commandes nécessaires à son fonctionnement.

Par exemple pour le générateur de plug-in `io data`, la commande `io data` est créée, héritée de la commande `framework`. Elle est ensuite enregistrée, ce qui implique que l'on peut écrire la commande `galactic framework io data` en ligne de commande.

Pour chaque commande, il est possible d'ajouter des arguments ainsi que des options. Il est également possible de poser des questions à l'utilisateur.

2.2.3 Jinja2

Jinja est un framework Python qui permet de générer des templates dynamiquement.

Dans notre cas, Jinja est utilisé dans les extensions du framework, lors de la création des plug-ins, pour générer les templates des différents fichiers.

Voilà succintement son fonctionnement.

Il est tout d'abord nécessaire de créer les templates des fichiers à générer dans un dossier particulier. Nous avons choisi de les mettre dans le dossier `share/galactic/apps/cli/framework/-templates/`.

Lors de l'exécution d'une commande, une instance de la classe `Environment` de Jinja est créée. Cette instance permet d'indiquer où se trouvent les templates à générer. Elle permet également de

récupérer ces templates et de leur passer des variables, ce qui permet ensuite de créer des fichiers personnalisés.

3 Mission

Ma mission est de créer des extensions pour le framework de la commande `galactic`. Ces extensions sont des générateurs de plug-ins qui permettront aux utilisateurs de créer plus facilement de nouveaux plug-ins.

3.1 Interaction

Il est nécessaire de questionner l'utilisateur à propos de certaines informations. Certaines sont indispensables pour tous les plug-ins, telles que le nom de l'auteur ou la description du plug-in, tandis que d'autres sont spécifiques aux plug-ins. Par exemple, pour un plug-in permettant de lire un certain type de fichiers, il faudra préciser le type des données du fichier.

Ces questions posées à l'utilisateur ont une réponse par défaut dans la majorité des cas pour accélérer et simplifier la prise en main.

De plus, ces questions peuvent être évitées à l'aide des options dans la commande.

3.2 Base de la commande : *core*

La base de la commande est héritée de la classe `Command` de Cléo et sert de classe parente à toutes les commandes de plug-in. Elle contient également les arguments et options communs à tous les plug-ins.

La commande prend un argument obligatoire : un chemin où le plug-in s'installera. De plus, elle prend en options les auteurs du plug-in ainsi que sa description.

Note
Comme indiqué précédemment, la commande seule n'existe pas. Elle sert uniquement de base pour les autres commandes.

3.2.1 Spécificités

3.2.1.1 Auteurs Les auteurs sont sauvegardés d'une commande à une autre dans un fichier `YAML`. Pour ce faire, j'ai utilisé l'outil *platformdirs* qui permet de récupérer le chemin du dossier où sont stockées les données utilisateur des applications. J'ai donc enregistré dans des sous-dossiers de ce dossier un fichier `YAML` avec les noms des auteurs.

Ensuite, j'ai fait en sorte de demander à l'utilisateur s'il souhaitait garder les auteurs sauvegardés ou non. Si non, alors une série de questions répétitives apparaissent, lui demandant de rentrer les auteurs un à un.

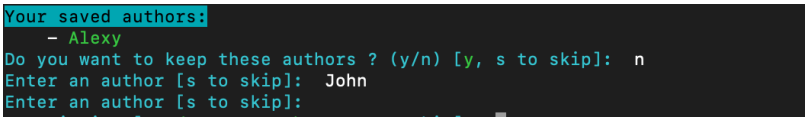

Figure 8: Exemple - Auteurs

Tip

Si le fichier YAML est vide ou inexistant, le programme va chercher le nom et l'e-mail Git de l'utilisateur s'il les a paramétrés. S'il ne l'a pas fait, le programme lui pose immédiatement la série de questions répétitives.

3.2.1.2 Description La particularité de la description ne se trouve pas dans une quelconque sauvegarde, mais dans le fait que lors de la question, une réponse par défaut est proposée en fonction du plug-in utilisé. Pour ce faire, j'ai utilisé une valeur par défaut définie dans les commandes des plug-ins mais utilisée dans la commande parente.

Dans l'exemple suivant, on constate que la commande `io-data` a été appelée et parmi les informations fournies, l'extension principale est `yaml`. Par conséquent, la proposition par défaut de la description est `Io data yaml package`.

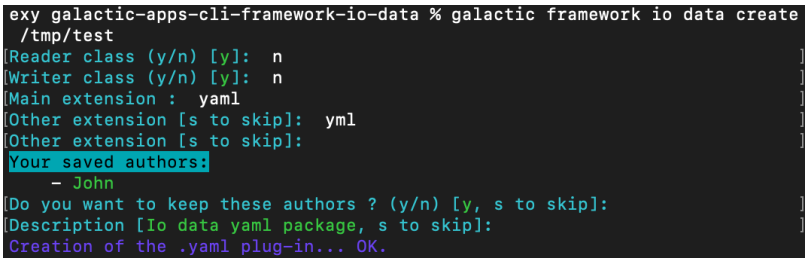

Figure 9: Exemple - Description

Note

Initialement, cette commande demandait uniquement l'auteur et ne le sauvegardait pas. Ce sont des améliorations que j'ai apportées à la commande.

3.3 Extension `io-data`

L'extension `io-data` est la première extension que j'ai décidé de créer. Son objectif est de générer un squelette de plug-in permettant de lire et/ou écrire dans un fichier, simplifiant ainsi la création de plug-ins *Data Readers* pour la plateforme GALACTIC.

Le nom complet de la commande est : `galactic framework io data create`

Cette commande prend un argument obligatoire, à savoir le chemin où le plug-in va s'installer. Cet argument est fourni par la commande de base.

Elle peut également accepter des options. Outre les options de base générées par Cléo telles que `--help` ou `--verbose`, ainsi que celles fournies par la commande de base, il est possible de passer le résultat de chaque question en option (voir exemple ci-dessous).

Les informations spécifiques à demander sont les suivantes :

- Est ce que le plug-in doit avoir une classe reader (oui/non) ?
- Est ce que le plug-in doit avoir une classe writer (oui/non) ?
- Quelle est l'extension principale des fichiers ?
- Quelles sont les autres extensions (question répétitive) ?
- Quel est le nom de la classe reader (s'il y en a une) ?
- Quel est le nom de la classe writer (s'il y en a une) ?

S'ajoutent à ces informations spécifiques les informations communes :

- Souhaitez-vous garder les auteurs sauvegardé (s'il y en a) ? Si non :
  - Quelles sont les auteurs (question répétitive) ?
- Quelle est la description ?

Une fois toutes les informations obtenues, que ce soit via les options ou les questions, le plug-in est créé au chemin précisé en argument, et un message d'information s'affiche.

Dans ce nouveau plug-in, on retrouve :

- `pyproject.toml`
- `README.md`
- `test/`
- `galactic/io/data/'extensionPrincipale'/`
  - `__init__.py`
  - `__init__.pyi`
  - `_command.py`
  - `py.typed`

3.3.1 Exemples

Commande `io-data` sans options :

```
exy galactic-apps-cli-framework-io-data % galactic framework io data create
 /tmp/test
[Reader class (y/n) [y]:                                                    ]
[Writer class (y/n) [y]:                                                    ]
[Main extension :  yaml                                                     ]
[Other extension [s to skip]:  yml                                          ]
[Other extension [s to skip]:                                               ]
[Name of the reader class [yamlDataReader]:                                 ]
[Name of the writer class [yamlDataWriter]:                                 ]
Your saved authors:
    - Alexy Lafosse <xla.lafosse@gmail.com>
[Do you want to keep these authors ? (y/n) [y, s to skip]:                  ]
[Description [Io data yaml package, s to skip]:                             ]
Creation of the .yaml plug-in... OK.
```

Figure 10: Exemple - Commande galactic io data

Commande `io-data` avec options :

```
(galactic-apps-cli-framework-io-data-py3.12) lafossealexy@MacBook-Air-de-Al
exy galactic-apps-cli-framework-io-data % galactic framework io data create
 /tmp/test --author Alexy --description test  --reader --readerClass reader
 --extension csv
Writer class (y/n) [y]:  n
Creation of the .csv plug-in... OK.
Authors saved.
```

Figure 11: Exemple - Commande galactic

Commande `io-data` avec l'option `--help` :

```
(galactic-apps-cli-framework-io-data-py3.12) lafossealexy@MacBook-Air-de-Al
exy galactic-apps-cli-framework-io-data % galactic framework io data create
 /tmp/test --help

Description:
  Create a data-io plugin

Usage:
  framework io data create [options] [--] <path>

Arguments:
  path                          The path to create the project at.

Options:
      --author=AUTHOR           Name of an author of the package. (multipl
e values allowed)
      --description=DESCRIPTION  Description of the package.
  -r, --reader                  Create a data reader class.
  -w, --writer                  Create a data writer class.
  -e, --extension=EXTENSION     An extension allowed in the plug-in. (mult
iple values allowed)
      --readerClass=READERCLASS  Set the name of the reader class.
      --writerClass=WRITERCLASS  Set the name of the writer class.
  -h, --help                    Display help for the given command. When n
o command is given display help for the list command.
  -q, --quiet                   Do not output any message.
  -V, --version                 Display this application version.
      --ansi                    Force ANSI output.
      --no-ansi                 Disable ANSI output.
  -n, --no-interaction          Do not ask any interactive question.
  -v|vv|vvv, --verbose          Increase the verbosity of messages: 1 for
normal output, 2 for more verbose output and 3 for debug.
```

Figure 12: Exemple - Commande galactic io data avec options

Arbre des fichiers du plug-in nouvellement créé :

```
.
├── README.md
├── galactic
│   └── io
│       └── data
│           └── yaml
│               ├── __init__.py
│               ├── __init__.pyi
│               ├── _main.py
│               └── py.typed
├── pyproject.toml
└── tests
```

Figure 13: Arbre du plug-in créé par io-data

3.4 Extension générateur d'extensions : *meta*

L'extension *meta* n'est pour l'instant pas encore créée. Seule la commande existe, mais elle n'affiche que "Hello World".

Cette extension a pour but de créer des extensions pour le framework, plus précisément de créer un squelette de générateur de plug-in.

Je vais maintenant vous présenter la commande telle que je l'imagine actuellement. La version finale risque certainement de différer de cette version.

Les informations spécifiques à demander sont :

- Quel est le nom de la commande à créer ?
- Souhaitez-vous ajouter un argument ? (question répétitive) Si oui :
  - Quel est son nom ?
  - Quel est sa description ?
  - Peut-il être multiple (oui/non) ?
  - Peut-il être optionnel (oui/non) ?
- Souhaitez-vous ajouter une option ? (question répétitive) Si oui :
  - Quel est son nom ?
  - Quel est sa description ?
  - Prend-elle une valeur (oui/non) ?
  - Peut-elle être multiple (oui/non) ?

Cette commande créera, dans un dossier, précisé les fichiers nécessaires au fonctionnement d'un générateur de plug-ins. Ces fichiers sont :

- `pyproject.toml`
- `README.md`
- `test/`
- `share/galactic/apps/cli/framework/templates/`
- `galactic/apps/cli/framework/'nomCommande'/`
  - `__init__.py`
  - `__init__.pyi`
  - `_command.py`
  - `py.typed`

3.5 Autres plug-ins

Il faudra ensuite créer les autres plug-ins liés aux différentes fonctionnalités de la plateforme GALACTIC. Par exemple, les caractéristiques.

3.6 Amélioration possible

Actuellement, les plug-ins doivent être installés par une commande `pip install`.

Je pense qu'il pourrait être intéressant de permettre l'ajout de chaque plug-in par une commande. Par exemple : `galactic framework add io-data`.

Cela permettrait de n'avoir à installer que la commande de base avec `pip`.

4 Tests

Avant de faire un commit, il est nécessaire de :

- Vérifier que le code est propre et qu'il n'y a pas de modification à apporter. On utilise pour cela des outils d'analyse de code.
- Créer des tests afin d'assurer le bon fonctionnement des commandes et faire en sorte que la couverture des tests soit la plus proche possible de 100%.

Note
La couverture des tests représente le pourcentage de lignes parcourues par les tests sur le nombre total de lignes du script.

Tip
Nous utilisons pre-commit, qui va exécuter entre autres ces commandes avant le commit et annuler celui-ci s'il rencontre une erreur.

Les commandes utilisées sont :

```
poetry run tox
```

Note
Cette commande exécute les tests du projet.

```
poetry run tox -e style
```

Note
Cette commande exécute les outils liés à la forme et à la structure des fichiers du projet afin de vérifier qu'ils sont conformes.

```
poetry run tox -e linter
```

Note
Cette commande exécute l'outil pylint indépendamment des autres car il est plus long à s'exécuter.

```
poetry run -e coverage
```

Note
Cette commande exécute les tests du projet et affiche le pourcentage de couverture des tests sur les différents fichiers Python.

5 Utilisation

Pour pouvoir utiliser un plug-in du framework `galactic`, il suffit de l'installer avec une commande ressemblant au code ci-dessous.

Note
Il n'est pas nécessaire d'installer la commande de base au préalable.

```
pip install \
    --index-url https://gitlab.univ-lr.fr/api/v4/groups/galactic/-
↪   /packages/pypi/simple
↪   \
    galactic-apps-cli-framework-io-data
```

Tip

Pour pouvoir utiliser l'extension `meta`, il suffit d'installer la commande de base `core`.

Note

Tous les plug-ins existants sont disponibles ici[3].

6 Choix effectués

6.1 Schéma de réponse

Lors de la phase de questions posées à l'utilisateur pour obtenir des informations, j'ai décidé d'intégrer des réponses par défaut pour simplifier l'utilisation. J'ai également cherché à respecter le même schéma de réponse pour permettre à l'utilisateur de s'habituer.

Par exemple :

- y/yes -> oui
- n/no -> non
- la touche 'entrée' -> réponse par défaut
- s/skip -> ne pas répondre

La seule particularité repose dans les questions répétitives avec une chaîne de caratères comme réponse.

Prenons l'exemple des auteurs :

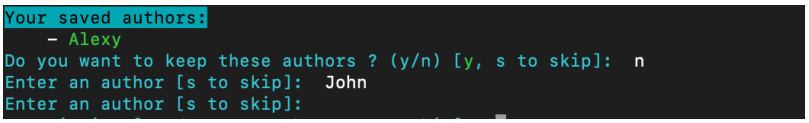

Figure 14: Exemple - Schéma de réponse

Lorsque l'on décide de saisir les auteurs, étant une question répétitive, il est possible, lorsqu'on n'a plus d'auteurs à saisir, de saisir `s` ou `skip`, mais également la touche 'entrée'. Cette option n'est pas écrite mais suit une certaine logique : nous n'allons pas sauvegarder un auteur sans nom. De plus, cela évite à l'utilisateur de taper sur son clavier. Malgré cela, j'ai décidé de laissé la touche `s` affichée afin de conserver ce schéma de réponse.

[3] https://galactic.univ-lr.fr/docs/apps/cli/framework/

6.2 Gestion des extensions

Pour la gestion des extensions pour les plug-ins de type `data-io`, j'ai fait le choix de demander à l'utilisateur l'extension principale, puis les autres extensions une à une pour simplifier l'expérience. Pas besoin de mettre des crochets ou d'utiliser une quelconque mise en forme.

En ligne de commande, l'option `--extension` peut être multiple et prendre plusieurs extensions facilement. Par exemple : `galactic framework command path --extension yaml --extension yml`

6.3 Méthodes de classes

Dans le code Python des commandes, suite à un conseil de M. Demko, j'ai décidé pour les méthodes de classes d'utiliser `@classmethod` plutôt que `@staticmethod`. Cela permet plus de liberté au développeur qui pourra appeler la méthode de deux manières :

```python
NomDeLaClasse.nomDeLaMethode
# ou
self.nomDeLaMethode
```

6.4 Gestion des erreurs

Lorsqu'une erreur survient à cause d'une mauvaise saisie de la part de l'utilisateur, plusieurs actions peuvent se produire :

- Si l'erreur survient lors de la saisie de la commande, alors la commande n'est pas exécutée.
- Si l'erreur survient lors de la phase de questions, alors un message d'erreur s'affiche et la question est posée à nouveau.

Ce système permet de ne pas avoir à ressaisir toutes les informations précédemment fournies.

Pour la gestion des erreurs, j'ai utilisé le système de questionnement de Cléo ainsi que des expressions régulières.

On peut voir dans l'exemple suivant les solutions qui fonctionnent et celles qui ne fonctionnent pas ainsi que le message d'erreur.

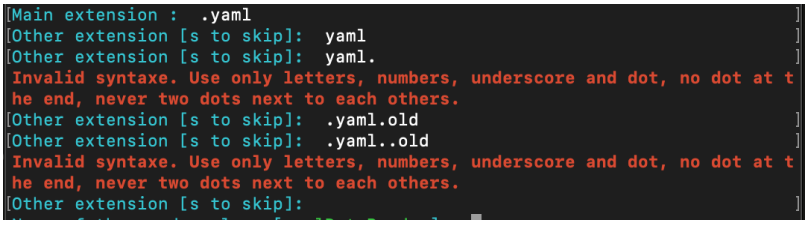

Figure 15: Exemple - Gestion des Erreurs

7 Difficultés rencontrées

7.1 Compréhension de la structure

Ma plus grande difficulté a été de comprendre l'objectif du stage, la structure de la plateforme GALACTIC, et surtout la structure autour de la commande `galactic`. Le début du stage a été particulièrement compliqué à cause de cette structure que je n'arrivais pas à bien identifier. J'ai fini par la comprendre de manière générale, après avoir posé de nombreuses questions à M. Demko et y avoir réfléchi moi-même.

Malgré cette compréhension globale, il restait toujours des zones d'ombre. Cependant, au fur et à mesure que j'avançais, les informations s'assemblaient, me donnant une vision plus complète de la structure. Il m'est aussi arrivé de réaliser que ce que je croyais correct était en réalité incorrect, ce qui m'a parfois contraint à reprendre entièrement l'étude de cette structure.

En conclusion, ma vision de la structure a constamment évolué durant le stage jusqu'à la présentation que j'en ai faite précédemment.

7.2 Mauvaise méthode

Il m'est arrivé de me tromper dans mon raisonnement à cause d'informations ou d'utilisations de commandes que je n'avais pas prises en considération. Par exemple, lors des tests de la commande de base (core). Afin de tester les différentes possibilités concernant les auteurs, je supprimais le dossier contenant le fichier de sauvegarde. Cependant cette méthode était inadéquate car elle risquait d'effacer d'autres informations importantes. Grâce à l'intervention de M. Demko, j'ai réexaminé la question et compris pourquoi cette approche n'était pas la bonne. J'ai ensuite découvert par moi-même la méthode appropriée, qui consistait à renommer temporairement le fichier.

7.3 Prise de décision

Une autre difficulté que j'ai rencontrée a été la nécessité de faire des choix, notamment pour améliorer l'expérience utilisateur. À mon avis, une interface en ligne de commande doit être intuitive et rapide à utiliser. Si elle exige de répondre à un trop grand nombre de questions, ce n'est pas idéal. Il est donc crucial de poser des questions utiles et de proposer des solutions par défaut adéquates. Dans certains cas, cela ne pose pas de problème du fait du peu de questions posées, mais pour des éléments plus complexes comme le meta plug-in, cela devient plus significatif.

8 Conclusion

En conclusion, mon objectif durant ce stage est de développer des extensions pour le framework de la commande `galactic`. Pour l'instant, une extension a été créée et est fonctionnelle, l'extension `io-data`. De plus, la commande de base a été enrichie pour offrir une fonctionnalité plus complète. Il me reste, d'ici la fin de mon stage, à finaliser l'extension *meta* et à concevoir le plus grand nombre

possible d'extensions pour les autres fonctionnalités de la plateforme GALACTIC. Ce projet est utile car il simplifiera la tâche des développeurs souhaitant créer des plug-ins pour GALACTIC, accélérant ainsi le processus de développement.

Mon expérience au début de ce stage a été marquée par une certaine difficulté à comprendre la structure du projet. Une fois cette obstacle surmontée, j'ai pu apprendre énormément. J'ai non seulement fait évoluer ma vision de la structure d'un projet, mais j'ai également acquis des compétences avec certains outils tels que Poetry. Ce stage a véritablement été une période d'évolution pour moi, enrichissant mes compétences en Python et en gestion de projet, notamment grâce à l'utilisation de Git et GitLab. Je suis convaincu que ces compétences seront extrêmement bénéfiques pour mes projets personnels à venir, notamment un projet d'application de gestion et de génération de fichier, dans lequel je souhaite me lancer.

9 Annexes

9.1 Liens

https://galactic.univ-lr.fr/slides/architecture/Galactic-Architecture-slides-20min-complete.pdf

https://pod.univ-lr.fr/video/1942-architecture-de-galactic/

https://galactic.univ-lr.fr/docs/apps/cli/framework/

9.2 Sources

9.2.1 Introduction

https://l3i.univ-larochelle.fr/Presentation-317

https://galactic.univ-lr.fr/slides/architecture/Galactic-Architecture-slides-20min-complete.pdf

https://pod.univ-lr.fr/video/1942-architecture-de-galactic/

9.2.2 Outils de développement

https://www.atlassian.com/fr/git/tutorials/comparing-workflows/gitflow-workflow

https://github.com/psf/black

https://pypi.org/project/slotscheck/

https://pypi.org/project/flake8/

https://flake8.pycqa.org/en/latest/

https://github.com/astral-sh/ruff

https://mypy.readthedocs.io/en/stable/index.html

https://pypi.org/project/pylint/

https://pypi.org/project/darglint/

https://doc8.readthedocs.io/en/latest/readme.html

https://www.sphinx-doc.org/en/master/

https://pypi.org/project/teyit/

https://pypi.org/project/refurb/

https://docs.pytest.org/en/8.2.x/

https://tox.wiki/en/4.15.0/

https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks

https://pre-commit.com

https://docs.gitlab.com/runner/

9.2.3 Outils liés au framework galactic

https://python-poetry.org/docs/

https://github.com/python-poetry/poetry

https://github.com/spoorn/poeblix

https://pypi.org/project/poetry-dynamic-versioning/

https://github.com/python-poetry/cleo

https://jinja.palletsprojects.com/en/3.1.x/