Licence informatique

La Rochelle Université

Galactic Market

Développement d'une interface graphique avec GTK

Djénaba BARRY

2025

Contents

Liste des figures

1 Cadre général du projet

1.1 Introduction

GALACTIC, acronyme de **GA**lois **LA**ttices **C**oncept **T**heory **I**mplicational systems and **C**losures, est un projet de recherche développé au sein du laboratoire L3i de La Rochelle Université. Il s'inscrit dans le domaine de l'intelligence artificielle explicable, avec pour objectif de produire des raisonnements à la fois transparents et compréhensibles.

Le projet repose principalement sur deux approches fondamentales : **L'Analyse Formelle des Concepts (AFC)**, introduite en 1982, qui permet d'organiser les données en treillis de concepts afin d'en faciliter l'interprétation et **le clustering hiérarchique** basé sur **les treillis**, qui permet une exploration structurée des relations entre groupes de données.

Le nom ***« Galois »*** fait référence à Évariste Galois, célèbre mathématicien français, reconnu pour ses travaux pionniers en algèbre, bien qu'il soit décédé très tôt, à l'âge de 20 ans.

En 2020, deux verrous scientifiques ont été levés par l'équipe du **L3i**, ouvrant la voie à l'analyse de données complexes et hétérogènes, notamment des données sous forme de séquences.

Le cœur technique du système, appelé **Galactic Core**, repose sur une architecture modulaire composée de plusieurs couches extensibles, décrites ci-dessous.

La couche des **caractéristiques** permet d'extraire tout type de valeur à partir des données, tandis que celle des **descriptions** permet de représenter des ensembles de valeurs au moyen d'**enveloppes convexes généralisées**.
Les **stratégies** explorent les données en ciblant des groupes d'éléments satisfaisant certaines propriétés définies sur chaque élément pris individuellement, alors que les **mesures** viennent alimenter les **méta-stratégies** qui orientent les parcours d'analyse.
Enfin, les **lecteurs de données** assurent la compatibilité avec différents formats de fichiers pour construire les jeux de données.

L'algorithme central utilisé dans **GALACTIC** est appelé **NextPriorityConcept**. Il permet la génération efficace de concepts pertinents en tenant compte de priorités définies.

La démarche scientifique du projet repose sur la notion d'**enveloppes** et s'articule autour de deux grandes familles de prédicats.

- Les **prédicats de descriptions**, qui définissent les ensembles de données à caractériser,
- Les **prédicats de stratégies**, qui orientent leur exploration au sein de l'espace des concepts.

Cette organisation conceptuelle se reflète dans l'architecture générale de **GALACTIC**, présentée ci-dessous :

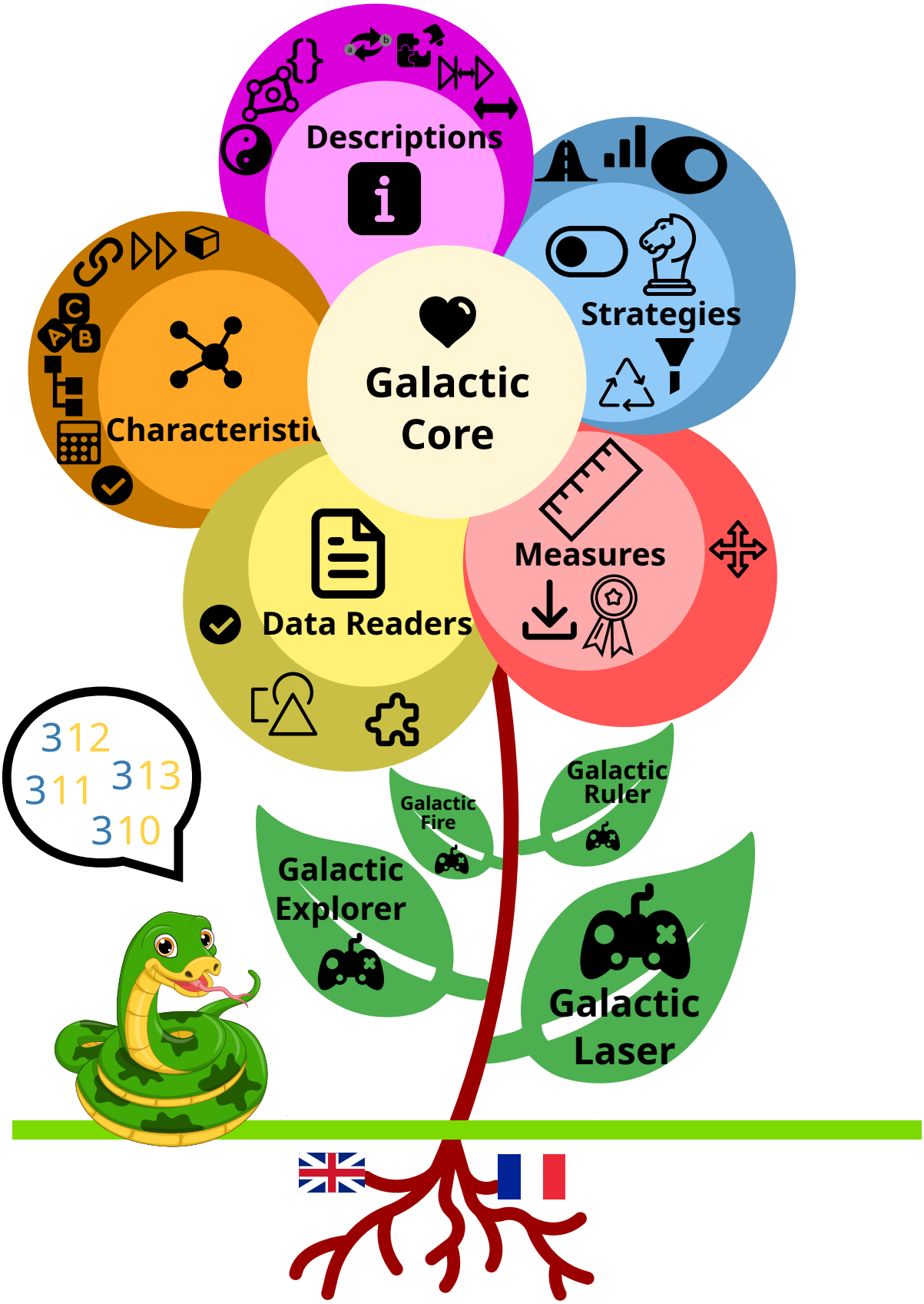


Figure 1: Architecture de GALACTIC

1.2 Résumé du projet

Le projet **GALACTIC Market** vise à offrir une infrastructure logicielle pour la gestion et la consultation d'un marché virtuel de paquets *Python*. Ce marché s'inscrit dans le cadre du cadriciel **GALACTIC**, développé selon les spécifications de la **Simple Repository API** (PEP503).
Contrairement aux paquets classiques, ceux du **GALACTIC Market** ne contiennent pas de code exécutable propre. Ils agissent comme des ***proxies*** vers des paquets hébergés sur des dépôts externes compatibles, qui peuvent nécessiter une authentification.
Afin de faciliter la recherche et l'exploitation de ces paquets, chacun est enrichi par des **classifiers Python** spécifiques ainsi que des métadonnées (icônes, liens vers la documentation, pages web, etc.), contribuant à une meilleure expérience utilisateur, notamment au sein d'interfaces graphiques.

L'architecture logicielle du projet, présentée ci-dessous, illustre l'organisation des composants et les interactions entre les différents modules.

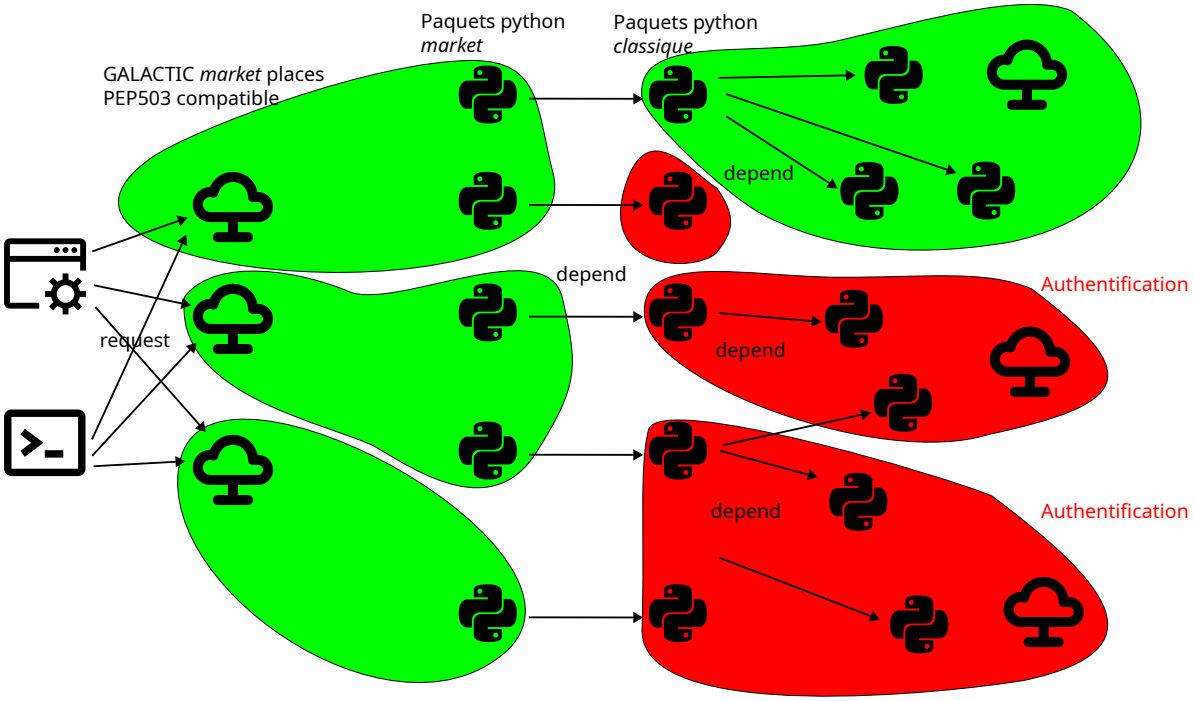


Figure 2: Architecture logicielle

Dans le cadre de ce projet, mes travaux se sont articulés principalement autour de trois axes complémentaires.

Tout d'abord, avec mon binome nous avons conçu le ***backend*** commun du système, cela a impliqué le développement d'une bibliothèque en Python permettant d'interagir avec des dépôts respectant **la norme PEP503**. Nous avons pris en charge la récupération, l'analyse (***parsing***) et l'indexation des métadonnées des paquets, incluant les ***classifiers*** et les **URLs** spécifiques.
Nous avons également intégré sur ce ***backend*** une gestion des ***proxies***, afin de rediriger les requêtes vers d'autres dépôts si nécessaire, ainsi qu'un mécanisme d'authentification dont l'implémentation n'est pas terminé permettant l'accès à des dépôts protégés.

Ensuite, nous avons crée un gestionnaire de cache qui assure le stockage des métadonnées et des

informations associées aux paquets récupérés. Il permet également d'effectuer des recherches avancées, notamment par ***classifiers*** ou par d'autres critères définis par l'utilisateur.

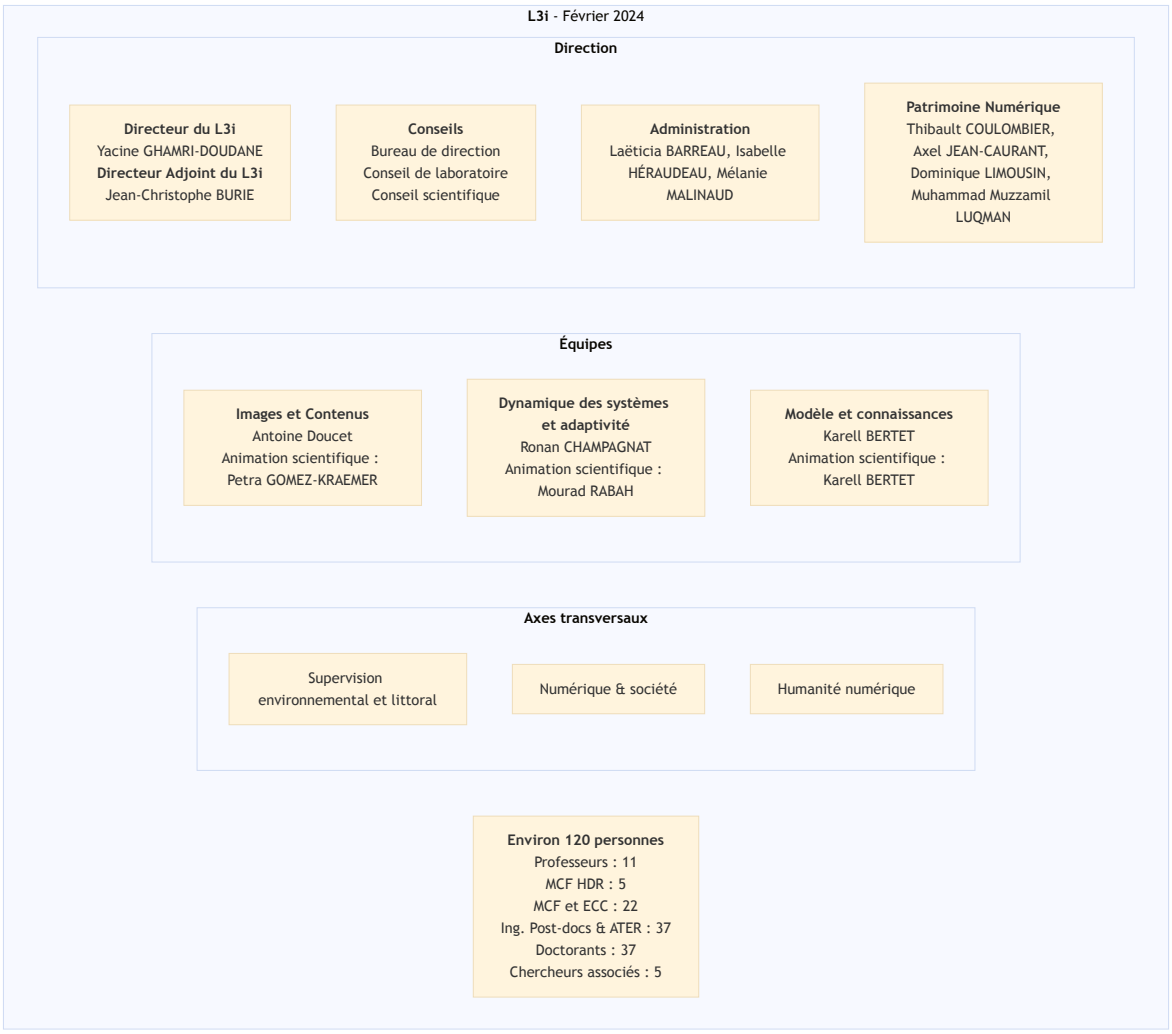
Enfin, j'ai développé une interface graphique à l'aide de la bibliothèque **GTK**, dans le but de proposer une interaction conviviale avec le système. Elle permet de consulter les paquets disponibles dans le **GALACTIC Market**, de les visualiser sous forme de listes ou de tableaux, d'effectuer des recherches dynamiques à l'aide de filtres et d'accéder à une fiche détaillée pour chaque paquet (incluant l'icône, la description et la documentation).
Elle permet aussi de gérer les dépôts configurés localement avec ou sans authentification, de télécharger ou de configurer les paquets et inclut divers éléments graphiques enrichis tels que des onglets, des boutons d'action ou encore des icônes.
Un soin particulier a été apporté à l'optimisation de cette interface, afin d'offrir une expérience utilisateur fluide et intuitive.

1.3 Présentation de l'entreprise

J'ai effectué mon stage sous la supervision de **M. Christophe DEMKO**, au sein de l'équipe Modèles et Connaissances du Laboratoire **L3i** (Laboratoire Informatique, Image et Interaction), une structure de recherche en informatique de l'Université de La Rochelle créée en 1993.

Ce laboratoire mène des travaux de recherche dans les domaines de l'informatique, de l'image et de l'interaction. Il regroupe des enseignants-chercheurs, chercheurs, ingénieurs et doctorants en informatique issus de différentes composantes de l'Université de La Rochelle, notamment de l'**IUT** (Institut Universitaire de Technologie)

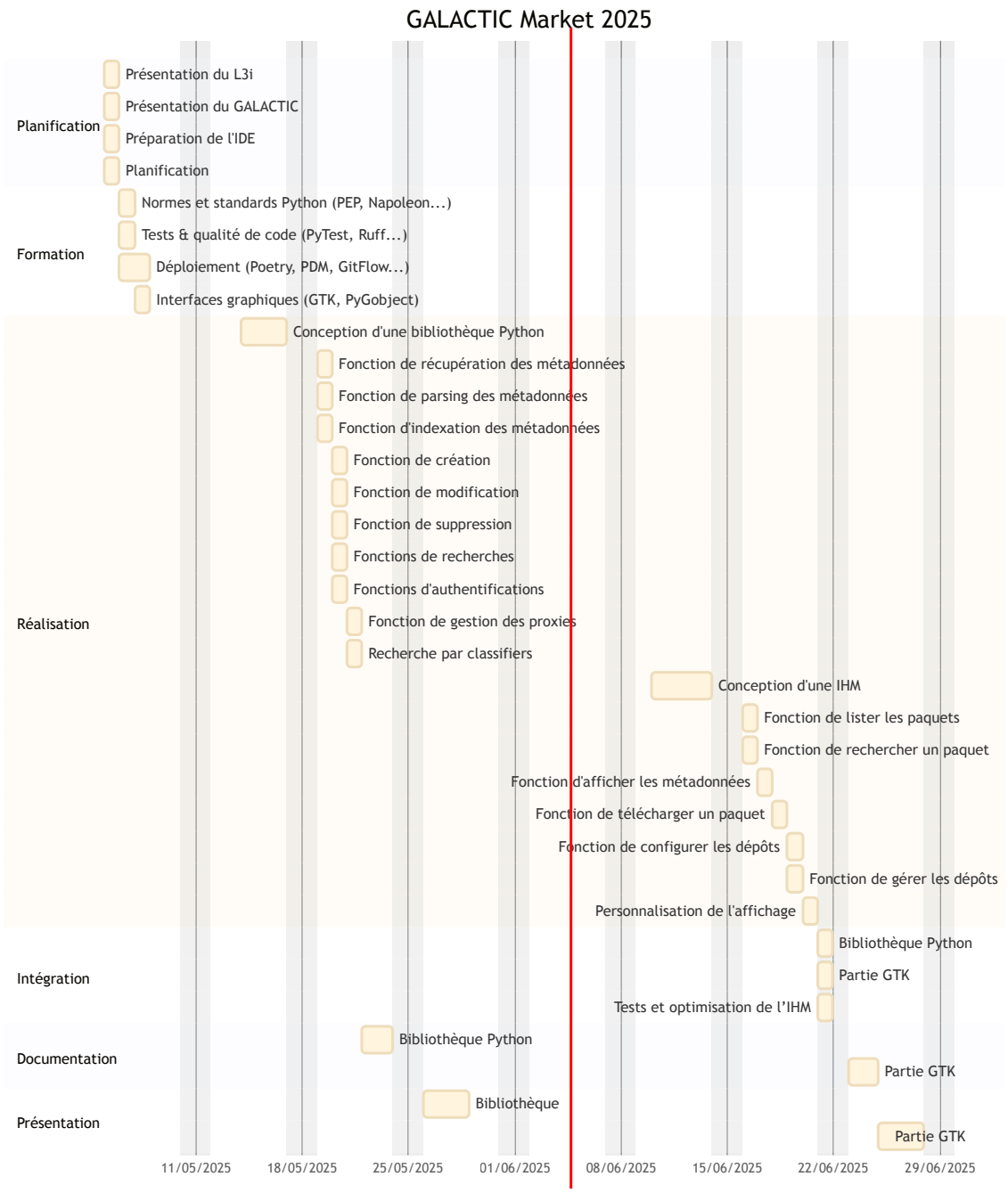
Il est organisé comme suit :



1.4 Organisation du travail

Pour planifier les différentes tâches du projet **GALACTIC Market**, j'ai utilisé Mermaid, un outil de visualisation permettant de créer facilement des diagrammes à partir de texte brut. L'utilisation d'un diagramme de Gantt nous a été suggérée, car ce type de diagramme permet de visualiser clairement les différentes phases du projet, leur durée, ainsi que leur enchaînement dans le temps. Le diagramme de Gantt permet de structurer visuellement les principales phases du projet, à savoir :

- La planification
- La formation
- La réalisation
- L'intégration
- La documentation
- La présentation


GALACTIC Market 2025
Planification
Présentation du L3i
Présentation du GALACTIC
Préparation de l'IDE
Planification
Formation
Normes et standards Python (PEP, Napoleon...)
Tests & qualité de code (PyTest, Ruff...)
Déploiement (Poetry, PDM, GitFlow...)
Interfaces graphiques (GTK, PyGobject)
Réalisation
Conception d'une bibliothèque Python
Fonction de récupération des métadonnées
Fonction de parsing des métadonnées
Fonction d'indexation des métadonnées
Fonction de création
Fonction de modification
Fonction de suppression
Fonctions de recherches
Fonctions d'authentifications
Fonction de gestion des proxies
Recherche par classifiers
Conception d'une IHM
Fonction de lister les paquets
Fonction de rechercher un paquet
Fonction d'afficher les métadonnées
Fonction de télécharger un paquet
Fonction de configurer les dépôts
Fonction de gérer les dépôts
Personnalisation de l'affichage
Intégration
Bibliothèque Python
Partie GTK
Tests et optimisation de l'IHM
Documentation
Bibliothèque Python
Partie GTK
Présentation
Bibliothèque
Partie GTK
11/05/2025
18/05/2025
25/05/2025
01/06/2025
08/06/2025
15/06/2025
22/06/2025
29/06/2025

2 Formation

Avant d'entamer la conception du projet **GALACTIC Market**, nous avons suivi plusieurs formations sur les bonnes pratiques de développement en Python, ainsi que sur les outils indispensables pour coder, tester, organiser, documenter et déployer efficacement une application.
Cette phase d'apprentissage nous a permis d'acquérir une base technique solide, couvrant la programmation, la gestion des dépendances, le respect des normes de qualité et la conception d'interfaces utilisateur ergonomiques.
C'est dans cette optique que cette section est structurée en quatre sous-parties, chacune présentant en détail un ensemble d'outils ou de méthodes abordés au cours de notre formation.

2.1 Normes et standards Python

Cette partie regroupe les normes officielles et les spécifications qui structurent l'écosystème Python. Elles établissent des conventions essentielles pour garantir la compatibilité, la lisibilité et la cohérence des projets, en particulier lors de leur emballage et de leur distribution.

Les principaux outils et références abordés ci-dessous participent à cette structuration commune.

L'**index PEP** (Python Enhancement Proposals) est un ensemble de documents qui décrivent les évolutions du langage Python, les bonnes pratiques et les standards adoptés par la communauté. Chaque **PEP** propose une nouvelle idée, améliore une bibliothèque ou change la manière de développer avec Python.

Il existe trois types de PEP :

- Les **PEP** de standardisations, qui introduisent de nouvelles fonctionnalités ou modifient les bibliothèques.
- Les **PEP** informatives, qui donnent des conseils ou des conventions à suivre.
- Les **PEP** de processus, qui expliquent comment le développement de Python est organisé.

Ces documents sont très importants car ils assurent une évolution cohérente du langage et influencent la création des outils Python.

Par exemple, la **PEP 503** définit une manière standard d'organiser les index de paquets comme **PyPI**. Elle explique comment structurer les URL et les pages HTML pour que les paquets soient faciles à trouver et à installer avec **pip**.
Elle impose une normalisation des noms de paquets pour éviter les erreurs, ce qui rend la gestion des paquets plus simple et plus fiable.
Mais pour bien distribuer un paquet, il faut aussi fournir des **métadonnées**.

Les **métadonnées** sont des informations qui décrivent un paquet à travers son nom, sa version, sa description, ses dépendances, ses auteurs, sa licence, etc.
Elles sont enregistrées dans des fichiers comme **setup.py** ou **pyproject.toml** et utilisées par des outils comme **build** ou **PDM** pour construire ou publier un paquet et **pip** pour les installer.
Ces informations sont essentielles pour gérer les projets Python et les rendre compatibles avec son écosystème.

2.2 Bibliothèques pour l'interrogation des métadonnées et dépôts Python

Cette section regroupe les outils ou bibliothèques qui facilitent la création, la gestion et la mise en production de projets Python. Ils permettent notamment d'interagir avec des registres de paquets comme **PyPI**, d'automatiser les processus de publication et d'intégrer Python dans des plateformes collaboratives telles que `GitLab`.

L'utilisation d'un `serveur PyPI` local ou privé permet d'héberger des paquets internes à une organisation. Cela garantit la distribution sécurisée de bibliothèques propriétaires, sans dépendre de l'index public. Cette approche est particulièrement adaptée aux entreprises développant des outils en interne, car elle assure un meilleur contrôle sur la gestion des dépendances.

Parmi les outils utiles dans ce contexte, `pkginfo` joue un rôle important. Cette bibliothèque permet d'extraire les métadonnées d'un paquet Python sans avoir à l'installer. Elle se montre très utile dans les processus d'intégration continue, les systèmes de ***build*** ou les audits de dépendances. Grâce à elle, on peut obtenir rapidement des informations telles que le nom, la version, les dépendances (***requires_dist***), la licence ou encore le format d'archive. Elle fonctionne avec les formats **.egg**, **.whl** et **.tar .gz**, ce qui en fait une solution polyvalente pour de nombreux cas d'usage.

Dans le même registre, `distlib` offre des fonctionnalités de bas niveau pour la manipulation des distributions Python. Elle est utilisée en interne par des outils comme **pip** et **virtualenv** . Cette bibliothèque permet notamment de lire et écrire les métadonnées de paquets selon les normes **PEP**, de gérer les fichiers ***Wheel*** ou encore de définir des points d'entrée (***entry points***) pour les exécutables et plugins. En assurant la conformité avec les standards de l'écosystème, `distlib` contribue à la fiabilité des chaînes de construction et de déploiement.

Pour gérer les fichiers de configuration, les caches ou les données utilisateur dans des contextes multiplateformes, `platformdirs` fournit une abstraction unifiée. Cette bibliothèque génère automatiquement des chemins adaptés au système d'exploitation sous-jacent. Par exemple, elle utilise **AppData** sur Windows, **~/Library** sur macOS et les répertoires définis par la spécification **XDG** sous Linux. Cela permet de développer des applications respectueuses des conventions de chaque environnement, sans avoir à gérer manuellement les spécificités de chaque plateforme.

Lorsqu'il s'agit d'interagir avec un dépôt de paquets Python, qu'il soit public ou privé, `pypi-simple` constitue une solution légère et efficace. Elle permet d'interroger l'index **PyPI** via le protocole **"Python Simple Repository"**, utilisé également par **pip**. En plus de la consultation des paquets disponibles, cette bibliothèque facilite le téléchargement de fichiers et l'accès aux métadonnées, incluant les contraintes de version, les types de fichiers (source ou wheel) ou encore les informations de signature **PGP**. Elle s'intègre aisément dans des outils internes ou des systèmes de gestion de dépendances personnalisés.

Enfin, pour intégrer Python à une plateforme collaborative comme `GitLab`, la bibliothèque `python-gitlab` propose une interface complète avec les **API REST** v4 et **GraphQL** de GitLab. Elle peut être utilisée de manière synchrone ou asynchrone et inclut un outil en ligne de commande (gitlab) qui simplifie les interactions avec l'API. Grâce à sa grande flexibilité, elle permet d'automatiser la gestion de projets, groupes, utilisateurs, pipelines et autres ressources GitLab. Elle prend en charge l'authentification, les connexions persistantes, l'usage de ***proxies***, la gestion des certificats, ainsi

que la reconnexion automatique en cas de coupures réseau. La pagination des résultats est gérée efficacement via des ***lazy iterators*** et l'outil encode automatiquement les paramètres des requêtes. De plus, la configuration peut être centralisée à partir de fichiers, de variables d'environnement ou d'arguments en ligne de commande, ce qui en fait un outil puissant pour l'automatisation des workflows GitLab.

Comparaison entre `python-gitlab` et `pypi-simple`

Ces deux outils peuvent parfois sembler proches lorsqu'on souhaite interagir avec des paquets hébergés sur GitLab, mais ils répondent en réalité à des besoins très différents.

Critère	pypi_simple	python-gitlab
Objectif principal	Interroger un index PyPI (public ou privé) pour obtenir des paquets Python	Interagir avec l'API GitLab (gestion de projets, paquets, CI/CD, etc.)
Avantages	Simple, léger, installation facile, spécialisé PyPI, téléchargement direct	API GitLab complète, support des objets GitLab, gestion CI/CD, artefacts, etc.
Inconvénients	Usage limité aux paquets PyPI, ne gère pas les projets GitLab	Plus complexe, nécessite une bonne compréhension de l'API GitLab
Authentification	Authentification HTTP de base (token simple)	Token GitLab personnel (plus sécurisé, plus complexe)
Téléchargement de paquets	Oui, simple avec les URLs fournies	Possible mais indirect (artefacts ou endpoints spécifiques)
Installation de dépendances	Très légères, aucun besoin de GitLab	Dépendances plus lourdes, requiert `python-gitlab`, parfois `requests`
Facilité de prise en main	Très facile, quelques lignes suffisent	Moyen à difficile selon l'action
Utilisation recommandée pour	Télécharger un paquet Python depuis un dépôt privé, lister les versions	Gérer un projet GitLab, CI/CD, issues, utilisateurs, etc.

Recommandations selon le besoin

Le choix de l'outil Python à utiliser dépend des actions à réaliser côté back-end, notamment en ce qui concerne l'accès à des paquets, la gestion de dépôts privés ou l'automatisation de processus CI/CD. Le tableau suivant présente les cas d'usage types et l'outil le plus adapté pour chacun d'eux

Besoin principal	Outil recommandé	Justification
Télécharger et lister les versions d'un paquet Python privé depuis GitLab	`pypi-simple`	Spécialisé, simple, rapide à mettre en place

Besoin principal	Outil recommandé	Justification
Gérer un projet GitLab (issues, pipelines, releases, permissions, etc.)	`python-gitlab`	Plus complet, adapté aux workflows DevOps complets
Déployer automatiquement des paquets ou artefacts avec CI/CD	`python-gitlab`	Permet d'interagir avec les pipelines et les artefacts
Créer une interface ou un script pour accéder à un paquet Python privé	`pypi-simple`	Très léger et sans surcharge inutile

2.3 Outils de test, d'automatisation et de qualité de code

Cette section décrit les outils utilisés pour garantir un développement Python structuré, fiable et maintenable. Elle couvre les aspects essentiels du cycle de vie du code, allant du formatage à la documentation, en passant par les tests, l'automatisation, la gestion des dépendances et la gestion des versions.

Afin de valider automatiquement le code avant chaque ***commit***, l'outil `pre-commit` est mis en place. ce gestionnaire de ***hooks*** multi-langages permet d'exécuter automatiquement des actions telles que le ***linting***, le formatage ou les tests, avant l'enregistrement des modifications. Il se charge aussi de gérer les dépendances nécessaires à l'exécution de ces actions, même lorsque certains outils, comme **Node**, ne sont pas installés localement. Cela garantit une cohérence dans les contributions, tout en évitant les erreurs dues à des environnements mal configurés.

La qualité du code et le respect des conventions sont assurés par deux outils complémentaires : ***ruff*** et ***Black*** .

`ruff` écrit en Rust, combine plusieurs outils de vérification en un seul, ce qui en fait un ***linter*** et un formateur très performant. Il remplace efficacement des solutions comme `flake8`, `isort`, `pylint` ou encore `mccabe`, tout en offrant une configuration simplifiée et une vitesse d'exécution remarquable.

`Black` De son coté, adopte une approche stricte et déterministe du formatage. Il reformate automatiquement le code Python selon des règles fixes, ce qui élimine les débats sur le style et garantit une lisibilité constante à travers tout le projet. Son intégration dans les pipelines CI/CD en fait un outil incontournable pour standardiser le code dans des équipes de toutes tailles.

Pour ce qui est des tests automatisés, `pytest` est utilisé en raison de sa simplicité d'utilisation, de sa lisibilité et de sa grande flexibilité. Il permet aussi bien l'écriture de tests unitaires que de tests d'intégration ou fonctionnels. Grâce à son système de fixtures, il facilite la configuration et la réutilisation de l'environnement de test, ce qui améliore la productivité des développeurs. De plus, sa compatibilité avec de nombreux plugins et outils d'intégration continue en fait une solution particulièrement adaptée aux projets évolutifs.

En ce qui concerne la gestion des dépendances et des environnements, plusieurs outils modernes sont adoptés.

uv écrit en Rust, se distingue par sa rapidité et sa compatibilité avec les fichiers **pyproject.toml**. Il vise à accélérer l'installation des dépendances et à améliorer les performances globales de l'environnement Python.

hatch propose une approche tout-en-un de la gestion de projets Python. Il permet de créer et de gérer des environnements virtuels, de construire des paquets et d'automatiser des tâches courantes comme les tests ou l'analyse statique, le tout via une interface unique et extensible.

Poetry centralise également la gestion des dépendances, la création d'environnements isolés, l'empaquetage, ainsi que la publication sur PyPI. Il offre un workflow simple et cohérent grâce à une configuration centralisée dans le fichier **pyproject.toml** et un système de verrouillage qui assure des installations reproductibles.

PDM Adopte une approche moderne conforme à la norme **PEP 582**, en stockant les dépendances dans un dossier local ***pypackages***, propre à chaque projet, au lieu de recourir à un environnement virtuel classique (**venv**). Il facilite ainsi la gestion des dépendances sans activation manuelle d'environnement, tout en conservant une configuration claire dans **pyproject.toml**. Il est conçu pour être rapide, simple et standardisé et offre une alternative légère et efficace pour les projets modulaires ou les environnements de développement portables.

Pour la documentation, l'outil de référence est `Sphinx`, qui permet de générer automatiquement une documentation à partir des **docstrings** du code source et de fichiers au format **.rst** ou **.md**. Il prend en charge de multiples formats de sortie comme HTML, PDF ou LaTeX, ce qui facilite la publication de contenus clairs et professionnels. Grâce à l'extension `Napoleon`, Sphinx peut interpréter des docstrings écrites selon les conventions **Google** ou **NumPy**, ce qui est particulièrement pratique si ces styles sont déjà adoptés dans le projet.

Enfin, pour organiser le développement, le modèle de branches `GitFlow` est appliqué. Ce modèle introduit une structure claire dans le cycle de vie des branches Git, avec des rôles spécifiques attribués à chacune. La branche ***main*** correspond à la version stable en production, tandis que ***develop*** est dédiée au développement courant. Des branches temporaires sont créées pour les fonctionnalités (***feature***), les corrections urgentes (***hotfix***) ou encore la préparation des versions (***release***). Ce processus favorise une meilleure coordination entre les équipes, assure une traçabilité des évolutions et permet de gérer efficacement les cycles de développement parallèles.

2.4 Outils pour interfaces graphiques

Cette section présente la bibliothèque utilisée pour développer une interface utilisateur graphique (**GUI**) en Python. Elle permet de créer des applications interactives, ergonomiques et multiplateformes. L'outil **GTK** est retenu dans ce contexte.

GTK (**G**imp **T**ool**k**it) est une bibliothèque destinée à la création d'interfaces graphiques multiplateformes. Elle est notamment utilisée dans des environnements de bureau comme `GNOME`. Compatible avec plusieurs langages, dont le `C` et Python (via `PyGObject`), elle offre une grande souplesse aux développeurs souhaitant concevoir des applications modernes.
Dotée d'un ensemble complet de widgets interactifs (boutons, listes, champs de texte…) et d'un moteur de rendu performant. `GTK` constitue une alternative robuste à `Qt` pour le développement

d'applications desktop élégantes et réactives.
Dans l'écosystème Python, GTK est généralement utilisé via le module `PyGObject`.
`PyGObject` est un module Python fournissant des liaisons (bindings) pour les bibliothèques basées sur `GObject`, telles que `GTK`, `GStreamer`, `WebKitGTK`, `GLib` ou `GIO`. Il permet d'utiliser ces bibliothèques écrites en C directement depuis Python, de manière fluide et intuitive.
Ce package est compatible avec Linux, Windows et macOS, il fonctionne avec `Python 3.9+` ainsi que `PyPy3`.
Distribué sous licence **LGPL v2.1+**, `PyGObject` bénéficie d'une documentation complète.
Techniquement, il repose sur les bibliothèques `GLib`, `GObject`, `GIRepository`, `libffi`, entre autres. Il interagit avec les bibliothèques C (comme `libgtk-4.so`) en s'appuyant sur des métadonnées décrites dans des fichiers `typelib` (par exemple : `Gtk-4.0.typelib`).
Ces métadonnées permettent à `PyGObject` de générer dynamiquement une interface Python, simplifiant l'utilisation des bibliothèques natives.

Note

Parmi les outils étudiés au cours de la formation, nous avons sélectionné certains outils ou librairies spécifiques en fonction de nos besoins :
`pypi-simple` a été préféré à **`python-gitlab`** pour sa simplicité de mise en place d'un index PyPI local.
Pour le développement, nous avons utilisé **`hatch`** pour la gestion de projet, **`black`** pour le formatage automatique du code, **`ruff`** pour l'analyse statique, ainsi que **`pkginfo`** et **`platformdirs`** pour la gestion des métadonnées et des chemins spécifiques à l'environnement.
Pour les interfaces graphiques, **GTK avec PyGObject** a été retenu, un choix pertinent pour les applications desktop modernes.
Côté documentation, nous avons utilisé **Sphinx** avec l'extension **Napoleon**, qui facilite la génération de documentation à partir de docstrings au format *Google* ou *NumPy*.
Enfin, **`uv`** a été utilisé pour la gestion des environnements virtuels et l'exécution des scripts dans un contexte d'intégration continue.

3 Conception du *backend*

Le ***backend*** a été conçu de manière structurée pour permettre la gestion efficace de plusieurs dépôts de paquets Python, notamment grâce à l'intégration de ***proxies*** personnalisés compatibles avec le format **PyPI**. Ce système autorise la communication simultanée avec plusieurs dépôts, en s'appuyant sur la bibliothèque `pypi_simple`, qui facilite l'interaction avec les serveurs respectant le standard **PyPI Simple API**.

L'ensemble des paquets téléchargés est stocké localement dans un cache organisé, ce qui rend possible leur consultation ultérieure sans nécessiter de nouvelle connexion Internet. Grâce à une architecture modulaire, le ***backend*** reste à la fois centralisé et évolutif. Chaque composant du système dispose d'une responsabilité clairement définie, ce qui améliore sa maintenabilité à long terme.

3.1 Développement de la bibliothèque Python gérant le market

La bibliothèque Python au cœur du ***backend*** repose sur le **modèle Singleton**, cela signifie qu'une seule instance du système est crée et utilisée pendant toute la durée de l'exécution. Cette instance unique est incarnée par une classe centrale appelée `Market`.

La bibliothèque se compose principalement de deux modules : `PackageManager` et `ProxyManager`.

Le module **PackageManager** est chargé de la gestion des paquets Python présents localement. Il est capable de lire les métadonnées des fichiers **.whl**, de dresser la liste des paquets disponibles et de fournir des informations détaillées ou synthétiques sur chacun d'eux.

Quant au module **ProxyManager**, il se consacre à la gestion des **proxies PyPI**. Il prend en charge la connexion à plusieurs sources, le téléchargement des paquets, ainsi que l'organisation du cache local. Cette division fonctionnelle suit le principe de responsabilité unique, ce qui rend l'architecture plus claire, plus facile à maintenir et plus simple à faire évoluer.

3.2 Gestion des métadonnées et des dépôts

Cette section détaille les mécanismes mis en place pour gérer efficacement les métadonnées des paquets Python ainsi que les dépôts distants (***proxies***). L'objectif est de permettre à l'utilisateur de parcourir, filtrer et télécharger des paquets de manière fluide et sécurisée, tout en offrant une architecture modulaire, capable d'évoluer vers des fonctionnalités plus avancées comme l'authentification. L'ensemble de ces composants assure une compatibilité optimale avec PyPI et les dépôts privés, dans le respect des standards de l'écosystème Python.

Gestion des métadonnées

Pour commencer, la gestion des métadonnées repose sur une analyse fine des paquets disponibles localement.
La bibliothèque utilise **pkginfo** afin d'extraire les métadonnées des paquets Python au format wheel (.whl), sans qu'il soit nécessaire de les installer.
Lors de ce processus, le système vérifie que le fichier est bien un fichier wheel valide. Il analyse ensuite les métadonnées présentes dans le fichier `PKG-INFO` et retourne une structure de données contenant les informations les plus pertinentes.
Les champs extraits incluent notamment le nom du paquet, sa version (selon la norme **PEP**), un résumé descriptif, le nom de l'auteur et le type de licence associé.
Cette méthode permet de naviguer efficacement parmi les paquets disponibles localement, avec des performances élevées puisqu'il n'est pas nécessaire de procéder à une installation préalable des paquets.

Gestion des proxies

En complément de la gestion locale, la plateforme intègre également une gestion centralisée des dépôts distants à l'aide d'un composant dédié.
Le composant **ProxyManager** offre une gestion avancée des ***proxies*** compatibles avec PyPI. Il permet

d'ajouter ou de supprimer des ***proxies***, d'en consulter la liste complète et de gérer automatiquement le téléchargement des paquets.
Lorsqu'un ***proxy*** est exploré, le système effectue une analyse complète du dépôt, filtre les fichiers au format `.whl`, procède à leur téléchargement en parallèle et gère les éventuelles erreurs réseau.
Le cache local est organisé de manière hiérarchique, avec un classement par ***proxy***, par paquet et par version. Les noms de fichiers sont explicites, ce qui simplifie la maintenance et évite les conflits entre différentes versions d'un même paquet.

Chaque ***proxy*** est enrichi d'un ensemble de métadonnées structurées, ce qui facilite son identification et sa configuration, notamment dans des environnements complexes ou professionnels.
Parmi les informations stockées figurent le nom du ***proxy***, une description facultative en Markdown, le type de ***proxy*** (défini par l'utilisateur ou par le système), l'URL complète du dépôt, la nécessité ou non d'une authentification, ainsi que la méthode d'authentification à utiliser (token ou basic).

Authentification (prévue)

Pour anticiper les besoins en sécurité et en accès restreint, le système prévoit également un mécanisme d'authentification évolutif.
Bien que l'authentification ne soit pas encore implémentée, l'architecture actuelle prévoit déjà les points d'extension nécessaires à son intégration future.
Le système sera ainsi capable de gérer l'accès à des dépôts privés, en supportant les tokens d'API ou l'authentification de type Basic. Il pourra également exploiter des fichiers de configuration standard tels que `.pypirc` ou encore utiliser des variables d'environnement pour la gestion sécurisée des identifiants.
Cette anticipation facilite l'intégration de ces mécanismes sans perturber le fonctionnement actuel.

Explication du schéma de publication des paquets

Pour mieux comprendre l'intégration des paquets dans l'écosystème **GALACTIC Market**, il est utile de revenir sur le processus de publication d'un paquet Python.
Le schéma présente le cycle de publication d'un paquet Python, depuis sa création jusqu'à son intégration dans un dépôt distant ou local.

Le développeur commence par structurer son projet en suivant les standards de l'écosystème Python (comme `pyproject.toml` pour la configuration). Le code source, les dépendances et les métadonnées sont organisés dans un répertoire conforme.

Le projet est ensuite compilé au format ***wheel*** (`.whl`) ou **sdist** (source distribution). Ces formats standardisés contiennent tout le nécessaire pour l'installation et la distribution du paquet.

Le paquet est téléversé vers un dépôt distant comme **PyPI** (Python Package Index) ou vers un dépôt privé via des outils comme **twine**. Ce dépôt centralise les versions et les métadonnées des paquets publiés.

Une fois publié, le paquet est indexé et devient accessible via des outils comme `pip`. Les métadonnées sont utilisées pour permettre des recherches efficaces, des filtres par version, licence ou auteur.

C'est à cette étape que le gestionnaire **GALACTIC Market** intervient : il explore les dépôts, télécharge les paquets (format `.whl`), extrait leurs métadonnées via **pkginfo** et les stocke localement selon une structure optimisée pour les performances.

Il convient de distinguer deux types de serveurs **PyPI** pouvant accueillir les paquets : le **serveur PyPI public**, ouvert à la communauté et largement utilisé pour les bibliothèques ***open-source*** et les **serveurs PyPI privés**, souvent déployés en entreprise pour héberger des paquets internes. Bien que le processus de publication reste identique (via des outils comme ***twine***), les serveurs privés nécessitent une configuration spécifique pour l'authentification et la gestion des accès. Le gestionnaire **GALACTIC Market** est capable d'explorer ces deux sources pour enrichir son catalogue local.

Le schéma ci-dessous illustre l'ensemble de ce processus de publication et d'intégration.

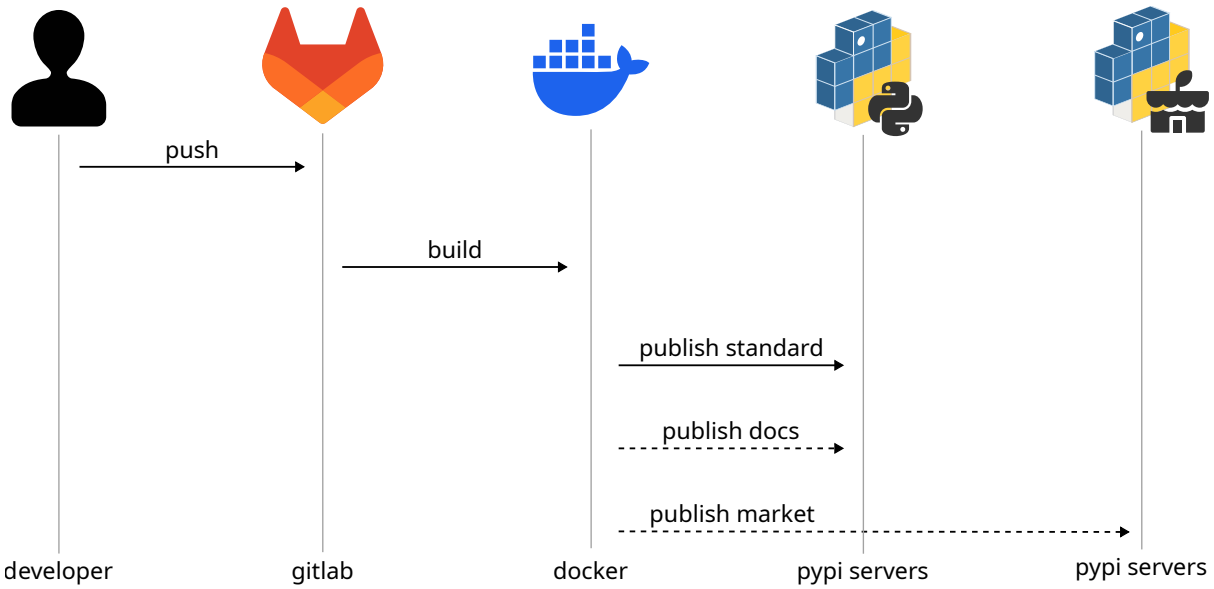


Figure 3: Publication de paquets

3.3 Création et utilisation du gestionnaire de cache

Dans cette partie, nous présentons le fonctionnement du gestionnaire de cache, qui joue un rôle central dans l'optimisation des performances et la réduction des dépendances réseau. En organisant localement les informations relatives aux paquets téléchargés, le système assure un accès rapide, une meilleure résilience hors ligne, et une base solide pour les futures fonctionnalités de recherche avancée. Cette approche permet également une compatibilité multiplateforme soignée grâce à l'usage de conventions standardisées pour le stockage des fichiers.

Gestion des données locales

Pour optimiser l'accès aux paquets et limiter les appels réseau, le ***backend*** s'appuie sur un système de stockage local structuré. Ce système fonctionne comme une base de données légère, permettant de consulter les paquets même hors ligne.
Les informations sont organisées dans une hiérarchie de dossiers standardisés, ce qui garantit la compatibilité avec les principaux systèmes d'exploitation, à savoir Linux, Windows et macOS.
Outre la consultation hors ligne, cette organisation permet également une amélioration des performances, car les recherches locales sont bien plus rapides que les requêtes réseau.
À terme, une évolution vers une base de données relationnelle comme SQLite pourrait permettre une recherche encore plus fine parmi les paquets.

Stockage des données

Le stockage des paquets suit la spécification `platformdirs`, ce qui garantit un emplacement de cache cohérent selon le système d'exploitation utilisé.
Sous Linux, les fichiers sont enregistrés dans le répertoire **.cache/galactic/market/**, tandis que sous Windows, ils sont placés dans **%LOCALAPPDATA%\galactic\market\cache**. Sur macOS, le cache est conservé dans **Library/Caches/galactic/market**.
Ce système de cache est géré automatiquement. Les fichiers corrompus peuvent être nettoyés, les anciennes versions supprimées, et l'espace disque est utilisé de manière optimisée.

Recherche avancée

Le ***backend*** prévoit également l'intégration de fonctionnalités de recherche avancée. Il sera ainsi possible de filtrer les paquets selon leur version (avec prise en charge de plages ou de jokers), leur type de licence (open source ou commerciale), ou encore en fonction de métadonnées spécifiques.
L'indexation des résultats permettra une navigation rapide, notamment grâce à un système de recherche en texte intégral et à l'ajout éventuel de tags pour faciliter le classement.
Une API de requêtage sera proposée, avec une syntaxe dédiée, la possibilité de sauvegarder les recherches effectuées, et l'exportation des résultats obtenus.

3.4 Diagramme de classes

Le diagramme de classes suivant illustre les relations entre les différents composants du ***backend***. La classe `Market` constitue le point d'entrée unique du système et centralise l'ensemble des opérations. Elle combine les fonctionnalités fournies par `PackageManager`, responsable de l'accès aux paquets locaux et à leurs métadonnées, et `ProxyManager`, chargé de la gestion des ***proxies***, des téléchargements et du cache.

Ce schéma permet de mieux comprendre l'architecture logique du projet et facilite la maintenance du code sur le long terme.

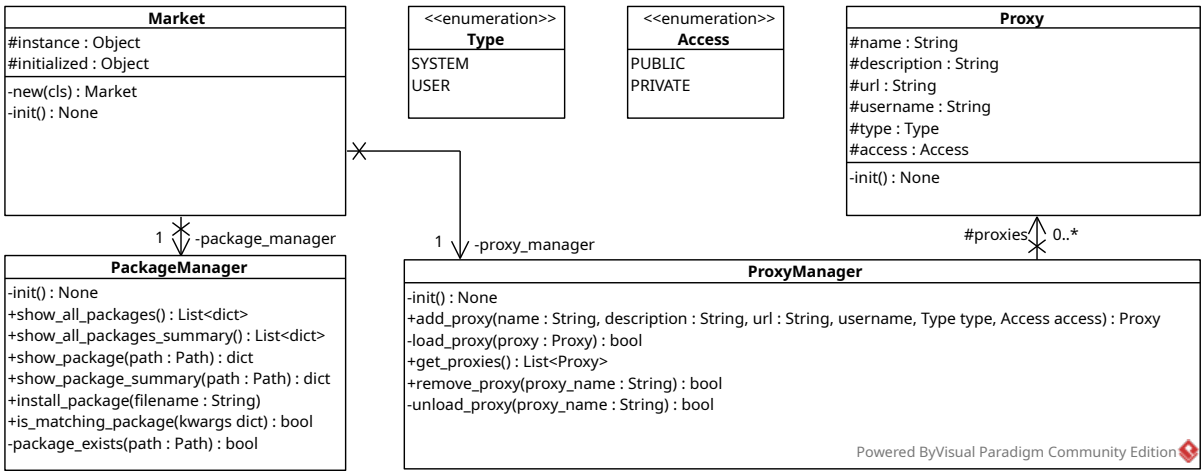


Figure 4: Diagramme des classes

Note

Ce ***backend*** permet de gérer des paquets Python de manière efficace et organisée.
Il facilite la connexion à plusieurs dépôts PyPI grâce à un système de ***proxies***, le téléchargement et le stockage local des paquets, la lecture des métadonnées sans avoir à installer les paquets, le fonctionnement même sans connexion internet.
Grâce à sa structure en modules séparés (comme PackageManager et ProxyManager), le code est facile à comprendre, à maintenir et à faire évoluer.
Ce système peut déjà répondre à beaucoup de besoins et il est prêt à accueillir de futures améliorations comme l'authentification ou des recherches plus poussées.

4 Développement d'une interface graphique avec GTK

Pour concevoir l'interface graphique du **GALACTIC Market**, nous avons choisi d'utiliser la bibliothèque **GTK 4** avec le langage de programmation **Python**.
L'application présente une interface moderne, organisée autour de plusieurs onglets distincts.

La page d'accueil ou **Home**, affiche le logo, le titre de l'application et un message de bienvenue.
L'onglet Packages, actuellement en cours de développement, permettra d'accéder à la liste des paquets disponibles et offrira des fonctionnalités de filtrage, de téléchargement et d'installation.
L'onglet Caches, également prévu, permettra de consulter les paquets stockés localement dans le cache de la machine. D'autres sections sont également en préparation, notamment **un onglet Language** pour permettre à l'utilisateur de choisir la langue de l'interface, **un onglet Display Mode** pour basculer entre les thèmes clair et sombre, et **un onglet About** destiné à présenter des informations générales sur le projet **GALACTIC Market**.

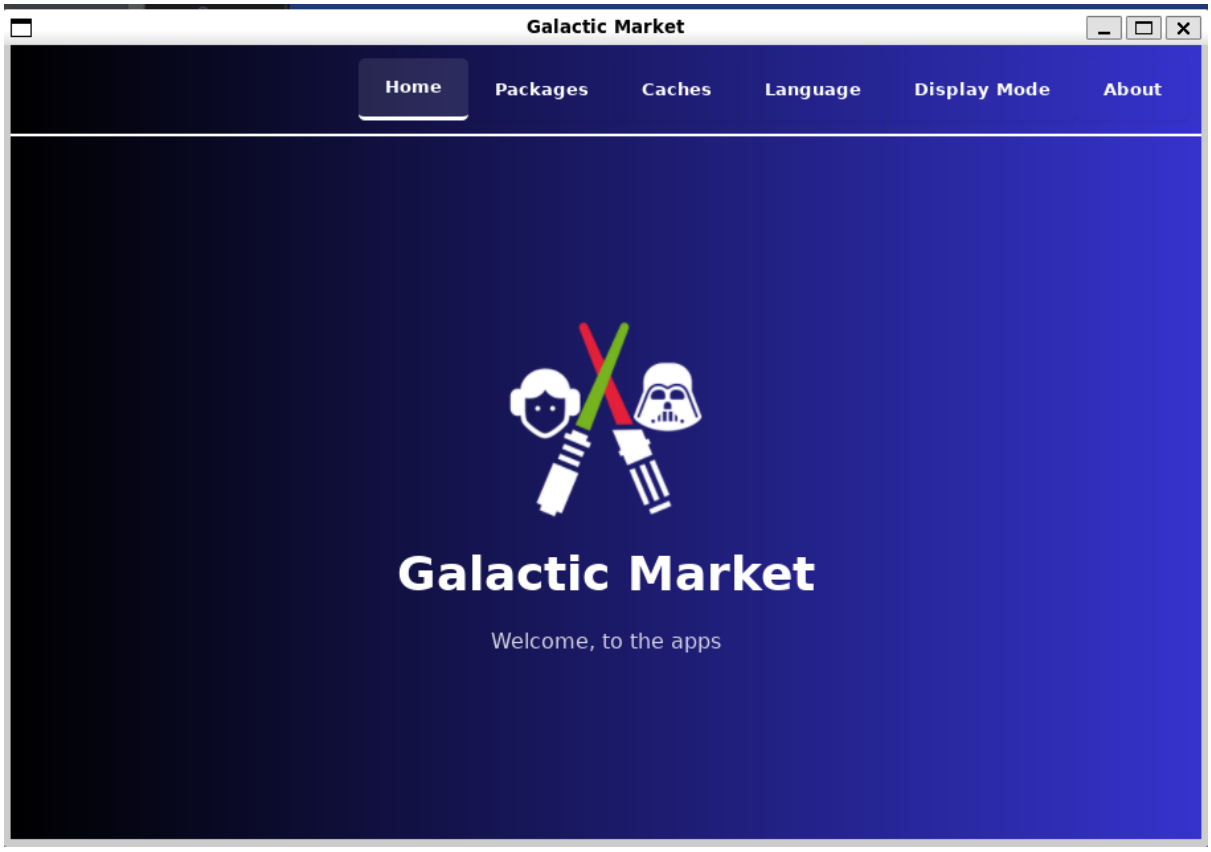


Figure 5: Page d accueil

4.1 Conception et développement de l'IHM

L'interface utilisateur repose sur une architecture utilisant **Gtk.Stack**, qui permet de gérer dynamiquement l'affichage des différentes pages selon l'onglet sélectionné. Chaque onglet correspond à une vue spécifique. La navigation s'effectue à travers une barre horizontale composée de **Gtk.ToggleButton**, garantissant une transition fluide entre les vues. L'application intègre également des éléments graphiques personnalisés, tels que **des boutons stylisés**, **des icônes** et une organisation claire de l'espace à l'écran pour renforcer l'ergonomie de l'ensemble.

4.2 Fonctionnalités de gestion des paquets

La gestion des paquets constitue un élément central de l'application. Dans les prochaines versions, l'utilisateur pourra consulter la liste des paquets disponibles sous forme de liste. L'interface sera enrichie par l'utilisation des **icônes GALACTIC**, conçues pour représenter visuellement chaque paquet de manière unique et intuitive.

Une fonction de recherche avancée permettra de filtrer les paquets selon différents critères, tels que la catégorie ou le type. Lorsqu'un paquet sera sélectionné, une fiche détaillée s'affichera, contenant son nom, une brève description et un lien vers sa documentation officielle. Depuis cette interface, l'utilisateur pourra interagir directement avec les paquets pour les télécharger, les installer ou les configurer.

4.3 Gestion des dépôts

Une section spécifique dédiée à la gestion des dépôts est en cours de développement. Elle offrira à l'utilisateur la possibilité de consulter les dépôts déjà configurés localement, de les activer ou de les désactiver selon ses besoins, et d'ajouter de nouveaux dépôts. L'interface prendra également en charge l'authentification pour les sources privées ou protégées, afin de garantir un accès sécurisé aux paquets.

4.4 Validation de l'interface et amélioration continue

Une phase de validation et d'amélioration continue de l'interface est prévue pour assurer une expérience utilisateur fluide et cohérente. Des tests utilisateurs seront réalisés afin d'évaluer la clarté de la navigation et la pertinence de l'organisation des contenus. L'interface sera optimisée pour s'adapter aux différentes tailles et résolutions d'écran, qu'il s'agisse d'un ordinateur portable ou d'un écran de bureau. Un soin particulier sera apporté à la gestion des erreurs, afin de garantir la stabilité de l'application en toutes circonstances. Enfin, l'ensemble des performances sera ajusté pour minimiser les temps de réponse et offrir un confort d'utilisation optimal.

5 Conclusion

Travailler sur ce projet durant mon stage m'a offert une expérience particulièrement enrichissante, tant sur le plan technique que personnel. J'ai pu contribuer activement au développement du projet **GALACTIC Market**, en participant à la conception d'une bibliothèque Python, à la gestion des métadonnées de paquets au format wheel, ainsi qu'à la mise en œuvre d'une interface graphique moderne et fonctionnelle avec l'outil GTK.

Ce stage m'a permis de renforcer mes compétences en programmation Python, en conception logicielle et en développement d'interfaces graphiques. Il m'a également sensibilisé à des aspects plus complexes tels que la structuration d'un projet modulaire, la gestion efficace des ressources locales, la communication avec des dépôts distants, ou encore la préparation de futures extensions comme l'authentification ou la recherche avancée.

Sur le plan méthodologique, j'ai appris à documenter rigoureusement mon travail, à structurer mon code selon des principes de maintenabilité et à travailler de manière plus autonome tout en respectant des objectifs concrets. J'ai aussi eu l'opportunité de collaborer dans un contexte professionnel, d'échanger avec des encadrants et de prendre en compte les retours pour faire évoluer les fonctionnalités développées.

Enfin, ce projet m'a donné une vision plus claire des problématiques rencontrées dans le domaine de la gestion de paquets logiciels et de leur intégration dans un environnement utilisateur.
Au cours de ce stage, j'ai été informée de l'événement IA NA, consacré à l'intelligence artificielle. Étant déjà déterminée à poursuivre un master dans ce domaine, cette participation représente pour moi une opportunité précieuse d'être plus informée sur ce secteur, de découvrir les innovations récentes et de consolider mon projet professionnel.

6 Remerciements

Je tiens à exprimer ma profonde gratitude à **mes parents**, qui ont toujours cru en moi et ont pris la décision courageuse de m'envoyer étudier à l'étranger. Leur soutien inconditionnel a été une source essentielle de motivation tout au long de ce parcours.

Je remercie les membres de **La Rochelle Université** pour m'avoir accueilli au sein de leur établissement, rendant ainsi cette aventure possible.
Ma reconnaissance va à **M. Thierry BOUWMANS**, mon tuteur pédagogique, pour son accompagnement tout au long de ce parcours.
Je tiens à remercier tout particulièrement **M. Christophe DEMKO**, pour sa confiance et son encadrement constant durant ce stage.
J'adresse également mes remerciements à Madame **Karell BERTET**, pour sa présentation claire et structurée du projet GALACTIC.
Je suis également reconnaissant envers **l'équipe du L3i** pour leur accueil chaleureux et leur disponibilité.

Un grand merci à mon binôme **Cyprien ROBINAUD**, pour sa collaboration précieuse et son excellent esprit d'équipe.
Enfin, je remercie sincèrement toutes les personnes formidables que j'ai eu la chance de rencontrer depuis mon arrivée en France.

Références

« A Successful Git Branching Model ». 2024. https://nvie.com/posts/a-successful-git-branching-model/.
« Black – The Uncompromising Python Code Formatter ». 2024. https://black.readthedocs.io/en/stable/.
« Distlib ». 2024. https://pypi.org/project/distlib/.
« Gitflow – Guide de gestion efficace des branches ». 2024. https://cyberinstitut.fr/gitflow-guide-gestion-efficace-branches/.
« GTK – GIMP Toolkit ». 2024. https://www.gtk.org/.
« Hatch – Modern Project, Package, and Environment Manager for Python ». 2024. https://hatch.pypa.io/latest/.
« Index PEP – Python Enhancement Proposals ». 2024. https://peps.python.org/topic/release/.
« Métadonnée ». 2024. https://www.digdash.com/fr/glossaire/metadonnee/.
« Napoleon Extension for Sphinx ». 2024. https://www.sphinx-doc.org/en/master/usage/extensions/napoleon.html.
« NextPriorityConcept: A new and generic algorithm computing concepts from complex and heterogeneous data ». 2024. https://hal.archives-ouvertes.fr/hal-02528679.
« PDM – Python Development Master ». 2024. https://pdm-project.org/en/latest/.
« PEP – Python Enhancement Proposals ». 2024. https://www.docstring.fr/glossaire/pep/.
« PEP 503 – Simple Repository API ». 2024. https://peps.python.org/pep-0503/.
« PkgInfo ». 2024. https://pypi.org/project/pkginfo/.
« Platformdirs ». 2024. https://pypi.org/project/platformdirs/.
« Poetry – Python Dependency Management and Packaging ». 2024. https://python-poetry.org/docs/.
« Pre-commit Framework ». 2024. https://pre-commit.com/.
« PyGObject ». 2024. https://pypi.org/project/PyGObject/.
« pypiserver ». 2024. https://pypi.org/project/pypiserver/.
« pypi-simple ». 2024. https://pypi.org/project/pypi-simple/.
« Pytest Documentation ». 2024. https://docs.pytest.org/en/stable/.
« python-gitlab ». 2024. https://pypi.org/project/python-gitlab/.
« Ruff – Python Linter ». 2024. https://docs.astral.sh/ruff/.
« Sphinx Documentation Generator ». 2024. https://www.sphinx-doc.org/en/master/.
« UV: Python Packaging Tool ». 2024. https://docs.astral.sh/uv/.