Contribution à Software Heritage

Projet 5éme année

2019 -2020

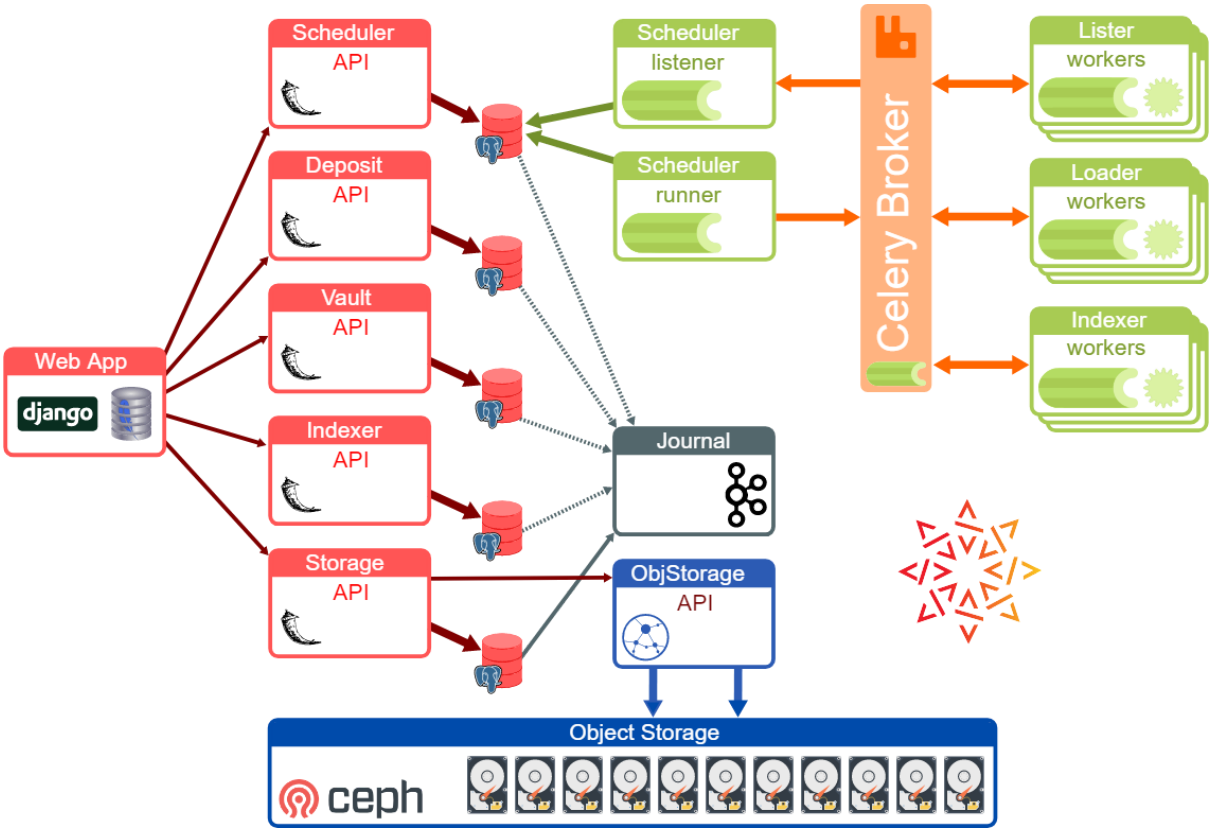
DALAINE Nathan

FONTFREYDE Joachim

GAUFFIER Léni

GAUTIER-PIGNONBLANC Yann

Architecture



Source: Software Heritage

L'architecture est très complexe et la documentation n'est pas forcément à jour un véritable challenge de la comprendre. On accède aux différents services depuis une interface web qui communique aux différents services via des API REST.

Il existe trois types différents de tâches:

- Les Listers, qui ont pour but de récupérer tous les codes sources à partir d'un site internet ou d'une forge donnée. La plupart du temps ils fonctionnent en utilisant les API fournies par les hébergeurs de forge, il n'y a pas de "scrapping à la main".
- Les Loaders: seront appelés après les listers et vont eux s'occuper d'importer ou bien de mettre à jour le code source présent dans une forge donnée. Ils sont spécifiques à chaque outil de versioning utilisé, aujourd'hui Git, Mercurial et SVN sont supportés.
- Les Indexers: ce composant va en extraire des informations complémentaires. Pour ce faire différents Indexers sont appelés en parallèle et vont chercher des fichiers connus comme par exemple package.json pour NPM, PKG-INFO pour Python… Les packages trouvés seront également importés dans Software Heritage.

Ces trois composants sont intimements liés et ne peuvent pas fonctionner l'un sans l'autre. Par exemple dans le cas de l'importation de l'ensemble des projets d'un hébergeur de forge, une première tâche de lister est programmée. Cette tache appellera elle même l'API de gitlab afin de récupérer les adresses des forges de tous les projets puis les ajouteras dans la base de donnée de Software Heritage via l'API de storage.

Une seconde tâche de type Loader est alors ajoutée à la queue, celle-ci va télécharger le contenu (code source, historique des révisions…) et l'ajouter à la base de données. Puis finalement une tâche de type indexer sera programmée afin d'extraire les metadatas du projet nouvellement importé.

Un modèle définit la façon dont les projets importés sont stockés dans la base de données, sont donc conservés: les répertoires, les commits, les tags (releases), les blobs. Des informations sont également ajoutées: l'origine qui représente l'url du repository qui a été cloné, les snapshots qui représentent l'état actuel (commit) des branches au moment de l'importation et un champ visit qui représente le moment auquel l'importation d'un projet à eu lieu, chaque visit est lié à une snapshot.

Réalisations techniques

Tâches "Easy hack"

Nous avons tout d'abord commencé ce projet par une revue des tâches taggées *easy hack* dans la forge Phabricator. Ces tâches ainsi annotées par la communauté indiquent une certaine facilité et un point d'entrée adéquat pour les développeurs nouvellement intéressés par le projet SH. Notre objectif a été d'en choisir quatre pour mettre le pied à l'étrier rapidement.

Parmi la quinzaine de tâches disponibles beaucoup portaient sur l'annotation du code Python. En effet le langage Python est un langage à typage dynamique c'est à dire que le type d'une variable est induit par l'assignation. Ce genre de typage est pratique pour ne pas avoir à se soucier de la déclaration d'un type à l'avance. En revanche lorsqu'un programme se complexifie il devient dur de suivre quel genre d'entrées et de sorties une fonction quelconque va nécessiter. Ces informations pouvaient déjà se trouver dans les commentaires, mais la mise à jour Python 3.5 a formalisé une notation permettant d'annoter les variables et fonctions. Cette version est sortie en 2015 et a dû attendre plusieurs années pour être réellement adoptée, il y a donc beaucoup du code de Software Héritage qui a été développé avant le passage à cette version et aujourd'hui la communauté essaie de pallier les manques d'annotations. Étant un bon moyen d'explorer le code, nous avons choisi deux tâches d'annotations.

Une autre tâche consistait à ajouter des compteurs dans le storage component afin de garder un décompte précis des éléments ajoutés à la base de données. A l'origine seul le nombre de snapshots/répertoires etc étaient pris en compte ces champs sont ajoutés à un tableau et récupérés grâce à leur hash. Cependant trois champs ajoutés à ce même tableau n'ont pas de hash associé: person, skipped_content, origin_visit ils n'étaient pas donc pas décomptés. Nous les avons ajoutés aux compteurs.

Un dernier easy hack réalisé a été l'ajout d'un endpoint de l'API swh-graph, un outil de visualisation de la représentation en graphe de l'archive Software Heritage. Il s'agissait d'améliorer l'endpoint /randomwalk qui effectue une traversée de graphe d'un noeud source à un noeud destination et renvoie les noeuds traversés. Il existait un endpoint /randomwalk/last qui renvoyait le dernier noeud traversé (avant celui de destination), l'objectif était de généraliser cela pour avoir les N derniers noeuds (ou les N premiers). Nous avons donc remplacer /last par un paramètre de requête ?limit=N avec N $\in \mathbb{Z}$, à la manière de l'indexation Python une valeur négative renverra les N derniers noeuds et une valeur de N positive renverra les N premiers noeuds. Enfin 0 est la valeur par défaut et renvoie tout.

Nouveau lister pour la plateforme Launchpad

Notre principale contribution au projet Software Heritage a été le nouveau lister pour une nouvelle forge logicielle : Launchpad. Cette forge est développée est maintenue par Canonical Ltd. qui est également mainteneur de Ubuntu. Ainsi le code source d'Ubuntu est tout d'abord développé et hébergé en utilisant cette plateforme. Canonical est également à l'origine d'un logiciel de gestion de version appelé Bazaar, Launchpad a donc énormément de projets développé avec Bazaar et non Git. À l'heure actuelle Launchpad héberge publiquement plus d'un million de branches Bazaar et 17000 repos Git.

Notre intérêt pour ce lister vient d'une tâche ouverte il y a près d'un an en mai 2019. Sur cette tâche un contributeur Google Summer of Code était assigné et il proposait certaines solutions, vinrent ensuite un groupe d'étudiant de master de l'Université de Montpellier qui avaient également des propositions sur ce lister en particulier. Mais en juillet la discussion prit fin et aucun code n'avait été proposé. Au début nous pensions pouvoir nous appuyer sur les études et propositions des différents contributeurs qui avaient participé à la discussion. A priori le stagiaire et le groupe de master avaient le même constat : Launchpad proposait une API HTTP ainsi qu'une librairie Python launchpadlib et il serait intéressant d'utiliser l'un et l'autre pour arriver à nos fins.

Mais intéressons nous d'abord sur ce que fait réellement un lister, quels sont les prérequis pour qu'il fonctionne et comment utiliser l'existant pour en développer un nouveau. Un lister c'est un package de swh.lister qui contient : une classe lister (ici ce sera LaunchpadLister), un modèle SQLalchemy décrivant les informations qu'un projet listé doit avoir, et une ou plusieurs tasks Celery qui sont celles qui seront effectivement utilisées par SH. Notre LaunchpadLister doit avoir une fonction run(), c'est celle là qui sera exécutée par les tasks Celery et qui contient tout le mécanisme de listing.

Pour prendre un exemple concret attardons nous sur le GithubLister. Au moment de sa conception il est important de se demander quelles informations le fournisseur nous propose et de quelle manière ces données sont structurées. Nous pouvons voir sur https://developer.github.com/v3/repos/#list-public-repositories que Github fournit un endpoint de son API listant tous ses dépôts en les indexant et les paginant selon un ID incrémental, on voit également que le lien vers la prochaine page est donnée dans les headers. Avec ça GithubLister peut utiliser le mixin IndexingHttpLister, cette classe est destinée au genre de lister qui utilisent des requêtes HTTP pour recevoir des listes de dépôts indexés. En implémentant IndexingHttpLister il suffit de spécifier la transformation de la réponse en données ingestibles selon le modèle défini et indiquer la page suivante à requêter. Il est

donc très simple de réaliser un nouveau lister suivant un schéma similaire, c'est d'ailleurs l'exemple donné dans le tutoriel de la documentation Software Heritage. IndexingHttpLister est également implémenté par les listers de Bitbucket et Phabricator. Il existe d'autres mixin prêts à l'emploi tels que PageByPageLister et SimpleLister. Toutes ces classes sont dérivées de ListerBase qui implémente toutes les fonctions liées à la base de données.

C'est en utilisant IndexingHttpLister que les contributeurs à la discussion pensaient développer ce nouveau lister. Or si l'on regarde de plus près ce que propose Launchpad on se rend vite compte que cette manière de faire est bancale et que Launchpad ne détaille à aucun moment l'indexation de ses réponses aux appels API. La proposition de base était de lister tous les projets grâce à un appel à l'API web https://api.launchpad.net/devel/projects puis de trier ceux qui seraient des projets Git et ensuite de lister tous les dépôts Git se trouvant dans ces projets grâce à un appel d'une fonction de leur librairie launchpadlib. Nous n'étions pas convaincus par cette méthode qui impliquait d'utiliser une indexation non fiable, c'est pourquoi nous avons décidé de contacter directement les développeurs de Launchpad pour avoir leur avis sur la question.

Nous sommes donc allés sur l'IRC des développeurs de Launchpad pour leur demander des détails concernant leur API. Ils étaient tout de suite très enthousiastes à l'idée de nous aider à implémenter ce lister dans Software Heritage. Par chance ils se trouvaient pendant cette semaine en congrès et ils purent aussi en discuter physiquement. Pour notre première interrogation sur l'indexation la réponse fut claire : il ne faut pas s'y fier. Ils nous ont par la suite expliquer en détail comment Launchpad marchait et que leur schéma de données était plus complexe que des projets contenant des dépôts, par exemple le code d'Ubuntu était hébergé en dehors des projets et aurait été omis par la solution initiale. Ainsi toute la discussion préalable sur la task par les autres contributeurs était caduque. Notre interlocuteur principal fut Colin Watson, il nous proposa de développer un nouvel endpoint de launchpadlib qui listerait tous les dépôts Git. C'est exactement ce qu'il nous fallait pour mener à bien ce lister.

Le nouveau lister consiste finalement d'appels à cette librairie uniquement et n'utilise donc pas de requêtes HTTP. Il a donc fallu construire par nous même le lister avec comme seul point d'appui ListerBase.

Les dépôts sont récupérés grâce à l'appel launchpad.git_repositories.getRepositories(order_by='most neglected first', modified_since_date=threshold), l'ordre de sortie est en fonction de l'activité du dépôt. Les premiers sortis sont les dépôts ayant été modifiés il y a le plus longtemps. On peut spécifier à partir de quand on recherche ces dépôts, ainsi en prenant la date de modification du dernier dépôt de la collection et en rappelant la fonction on peut itérer sur la collection.

Notre fonction run() comporte un paramètre max_bound qui permet de spécifier à partir de quelle date elle va commencer. La fonction run s'occupe d'ingérer les dépôts git itérativement de max_bound jusqu'à la fin et de mettre à jour la base de donnée en conséquence.

Avec cette configuration nous avons pu concevoir 3 tâches :

- IncrementalLaunchpadLister : tâche de base qui lance simplement run() à un max_bound spécifié.
- FullLaunchpadLister : qui utilise IncrementalLaunchpadLister avec comme max_bound None, ce qui va itérer sur la collection entière.

- NewLaunchpadLister : qui utilise IncrementalLaunchpadLister avec comme max_bound la dernière date modifiée maximale des entrées déjà présentes dans la base de donnée. Ainsi on n'a pas à réitérer sur des dépôts dont on sait qu'ils n'ont pas été modifiés.

À l'heure actuelle le lister est fonctionnel et à l'état de code review sur Phabricator. Les tests sont à modifier pour mocker les appels à launchpadlib, une fois effectué le lister devrait pouvoir être déployé en production.