Mergify

Rapport de stage

Mergify

Août 2020 - Février 2021

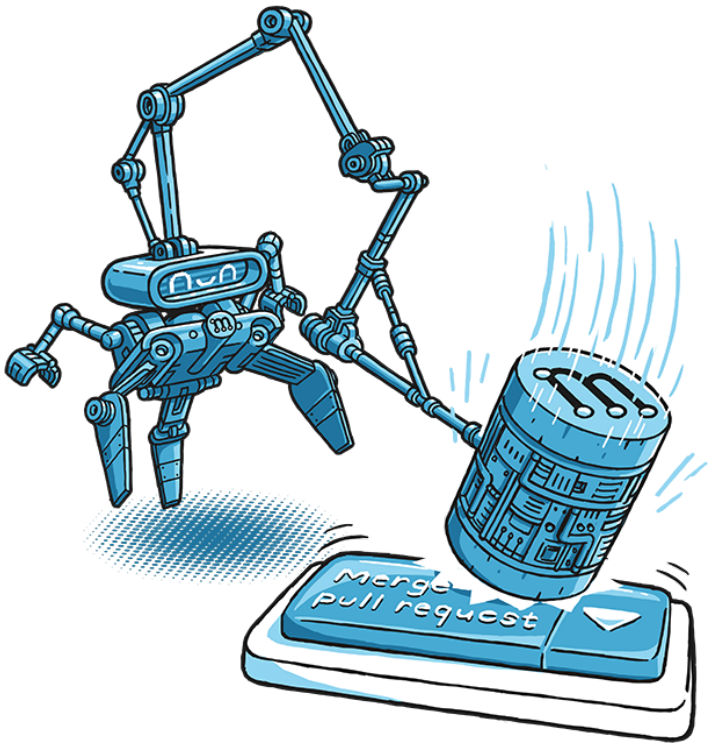


Table des matières

Remerciements

Je tiens vivement à remercier mon mentor de formation, Erwan Keribin, qui me suit depuis le début de mon parcours chez OpenClassrooms. Il a su me guider à travers l'ensemble des projets qui jalonnent ma formation, mais aussi me faire découvrir le métier de développeur en partageant ses expériences.

Je remercie aussi Julien Danjou pour m'avoir offert l'opportunité d'évoluer au sein de Mergify, de m'avoir accordé sa confiance pour l'ensemble des projets qu'il m'a attribués durant cette période de stage. Je le remercie aussi de m'avoir proposé de faire quelques sessions de travail chez lui, qui ont été des moments d'échanges et de partages enrichissants.

Merci à Mehdi Abaakouk pour son accueil à Mergify. Il m'a permis de découvrir sereinement le framework ReactJs. Il m'a été d'une grande aide sur beaucoup de sujets pointus et a su faire preuve de pédagogie et de patience lorsque j'étais coincé.

Enfin, merci aussi à Camille sans qui je n'aurais pas pu faire ce stage, et qui a été ma co-stagiaire durant une courte période.

Contexte

Présentation

Dessinateur technique dans le domaine du bâtiment durant une dizaine d'années, puis formateur pour un logiciel de modélisation 3D, j'ai décidé d'orienter ma carrière vers le monde du développement.

En juillet 2019, j'ai entamé le parcours de formation **Développeur d'Application Python** proposé par OpenClassrooms.

Dans le cadre de cette formation, j'ai décidé de suivre un stage de six mois en entreprise. Ce rapport relate mon expérience au sein de cette structure.

L'entreprise

Mergify est une société fondée en 2018 par Julien Danjou et Mehdi Abaakouk. Ensemble, ils ont créé un outil d'automatisation autour des pull-requests de GitHub.

Cet outil permet aux utilisateurs d'automatiquement merger, commenter, rebaser, mettre à jour, étiqueter, backporter, fermer, assigner les pull-requests.

Pendant mon stage, Julien et Mehdi conservaient une activité salariée en parallèle de Mergify. A la fin de mon stage, Mehdi était à mi-temps pour Mergify avec l'objectif de passer à plein temps dans le courant de l'année 2021.

L'entreprise est domiciliée à Toulouse et n'a pas de locaux officiels (Julien est sur Paris, Mehdi sur Toulouse). L'ensemble du stage s'est déroulé en télétravail.

Motivation

La réalisation d'un stage dans le cadre de cette formation est une option. Pour le projet 12, l'étudiant a le choix entre :

- créer un outil de veille technologique ;
- effectuer un stage d'une durée minimale de deux mois.

Le stage était pour moi l'occasion d'aiguiser mes compétences et d'arriver sur le marché de l'emploi avec plus de confiance.

L'opportunité de stage chez Mergify s'est présentée grâce à Camille, une étudiante du parcours.

La pile technique utilisée par Mergify présentait tous les critères que je m'étais fixés : Python, Flask, PostgreSQL. L'application front-end (tableau de bord) se base sur ReactJs, une nouveauté dans ma formation.

Le cœur de Mergify, que l'on appelle l'engine, est Open Source sous licence Apache 2.0. Cette particularité m'a d'autant plus incité à tenter ma chance !

Objectifs

Pour ce stage je m'étais fixé trois objectifs :

1. confirmer/infirmer ma préférence pour les projets orientés back-end ;
2. découvrir la vie d'un projet d'envergure ;
3. avoir une première expérience professionnelle, valorisable par la suite.

Problématique

La clientèle de Mergify se développait de façon croissante. Il fallait alors prendre en compte certains aspects mis de côté par manque de temps et considérés comme mineurs dans l'ordre des priorités.

Il était temps pour Julien et Mehdi de pouvoir alerter leurs clients en cas de problèmes avec leurs souscriptions, d'être en mesure de comprendre les raisons du départ d'un utilisateur, de récompenser les clients qui parlaient de Mergify autour d'eux, etc.

Pour cela, j'ai débuté mon stage dans l'optique de répondre à la question suivante :

Comment améliorer l'expérience client grâce au développement de nouvelles fonctionnalités ?

Missions

Durant le stage plusieurs missions m'ont été confiées. Celles-ci peuvent être classées suivant trois catégories :

1. résolution de bugs ;
2. création de nouvelles fonctionnalités ;
3. rédaction de documentations.

Mergify fonctionne avec deux dépôts de code :

1. un dépôt open source pour l'engine : https://github.com/Mergifyio/mergify-engine ;
2. un dépôt privé pour la partie tableau de bord : https://dashboard.mergify.io (le dépôt n'est pas accessible).

Dans les parties suivantes, nous allons détailler trois projets en particulier que j'ai réalisé durant cette période :

- Agir en fonction d'événements tiers.
- Créer un composant React.
- Créer un programme de parrainage.

Agir en fonction d'événements tiers

Cahier des charges

Pour certains événements, on souhaitait être en mesure d'envoyer un e-mail à nos clients. Exemples :

- un client du marketplace GitHub a plus d'utilisateurs que le nombre de sièges achetés ;
- un utilisateur résilie sa souscription via Stripe ;
- une erreur de paiement générée par Stripe ;
- etc.

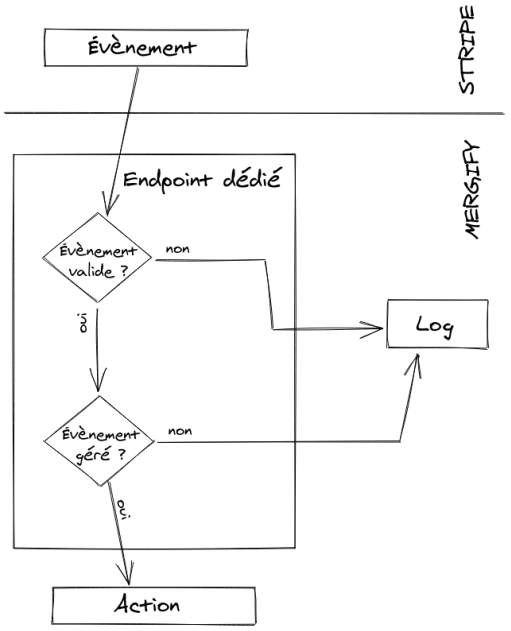
L'API du tableau de bord était déjà en mesure de recevoir les évènements envoyés par GitHub. Il nous manquait un endpoint pour la réception de ceux envoyés par Stripe.

Puisque l'on souhaitait envoyer des e-mails transactionnels, il fallait aussi choisir un fournisseur pour ce service.

Réalisation

Endpoint Stripe

Dans un premier temps, on a créé un endpoint pour recevoir les événements envoyés par l'API de Stripe. Cet endpoint fonctionne de la manière suivante :



Une fois l'endpoint créé, nous pouvions traiter les évènements qui nous intéressaient. Exemple avec l'évènement `customer.subscription.deleted` de Stripe :

```
def filter_and_dispatch_stripe_event(event: stripe.Event) -> None:
    with models.Session() as session:
        if event.type == "customer.subscription.deleted":
            stripe_customer = stripe.Customer.retrieve(event.data.object.customer)
            github_account = get_github_account_from_stripe_customer(
                session, stripe_customer
             )

            if github_account:
                mail.subscription_canceled(github_account, stripe_customer)
```

Lorsque nous recevions l'évènement `customer.subscription.deleted`, nous récupérions le compte de notre client via `get_github_account_from_stripe_customer` puis nous appelions la fonction `subscription_canceled`.

Cette fonction permettait de générer le template d'un e-mail puis de l'envoyer grâce à la librairie Python : postmarker.

```
def subscription_canceled(
    github_account: gh_account.GitHubAccount,
    stripe_customer: typing.Optional[stripe.Customer] = None,
) -> None:
    email_addresses = _get_email_adresses(github_account, stripe_customer)

    if email_addresses is None:
        return None

    to_email, from_email, bcc_email = email_addresses
    name = github_account.name or github_account.login
    template_alias = config.POSTMARK_SUBSCRIPTION_CANCELED_TEMPLATE
    template_model = {"name": name}

    try:
        POSTMARK.emails.send_with_template(
            TemplateAlias=template_alias,
            To=to_email,
            From=from_email,
            Bcc=bcc_email,
            TemplateModel=template_model,
        )
    except postmarker.exceptions.ClientError:
        LOG.error( "No email sent for %r", github_account, exc_info=True )
    else:
        LOG.info("Email sent for %r", github_account)
```

Choix du service d'e-mails transactionnels

Après la création de l'endpoint, nous avons décidé de prendre un fournisseur pour l'envoi des e-mails transactionnels.

Mergify utilise Mailchimp pour l'envoi des e-mails d'onboarding aux nouveaux utilisateurs. Malheureusement, ce fournisseur ne proposait pas d'offre gratuite pour les petits volumes (moins de 100 e-mails).

Nous avons réalisé un comparatif entre trois autres fournisseurs : Sendgrid, Mailgun et Postmark. Nous avons utilisé les critères suivants :

- offre gratuite pour les petits volumes ;
- documentation de l'API ;
- système de templates.

Notre choix s'est arrêté sur Postmark. Comme évoqué précédemment, nous avons utilisé la librairie Python postmarker pour la partie back-end.

La création des templates a été réalisée avec MailMason, un outil créé par Postmark.

Bilan du projet

Ce projet m'a fait comprendre l'intérêt d'avoir un intermédiaire comme Postmark. Ces fournisseurs s'assurent de la délivrabilité des e-mails et évitent ainsi d'avoir son nom de domaine blacklisté par les fournisseurs de messageries.

J'ai découvert l'importance du choix d'une technologie ou d'un fournisseur. Même s'il est toujours possible d'en changer, la transition vers un autre fournisseur demandera toujours du travail supplémentaire, potentiellement sur le long terme.

J'ai pris conscience de l'importance d'avoir une documentation claire et détaillée. Elle garantit une bonne prise en main par les nouveaux utilisateurs et peut être un critère de choix pour les prospects.

Créer un composant React

Mergify utilise un système de chat accessible depuis le site ou le tableau de bord.

Cet outil facilite la relation avec les clients. Le service est fourni par Crisp.

Cahier des charges

Pour intégrer le client chat dans une application React, la documentation de Crisp propose d'intégrer le code suivant dans le composant principal de l'application :

```
componentDidMount () {
  window.$crisp = [];
  window.CRISP_WEBSITE_ID = "YOUR_WEBSITE_ID";

  (function() {
    var d = document;
    var s = d.createElement("script");

    s.src = "https://client.crisp.chat/l.js";
    s.async = 1;
    d.getElementsByTagName("head")[0].appendChild(s);
  })();
}
```

On doit par la suite utiliser les méthodes du SDK `$crisp` pour paramétrer la fenêtre de chat. Par exemple :

```
$crisp.push(["set", "user:email", [email]])
$crisp.push(["set", "user:phone", [phone]])
$crisp.push(["set", "user:nickname", [nickname]])
```

Sans la création d'un composant spécifique, le composant principal se trouve chargé de nombreuses lignes de code réservées au chat.

Nous avons décidé de créer ce composant de toute pièce et de le distribuer sous licence Open Source.

Réalisation

Avant d'utiliser Crisp, Mergify utilisait un concurrent Drift. L'intégration dans le tableau de bord se faisait avec le composant react-drift.

Les deux clients présentaient des similarités pour l'intégration et le paramétrage. Nous nous sommes donc inspirés de react-drift pour créer react-crisp.

L'utilisateur insère le composant de cette manière :

```
<Crisp
  crispWebsiteId="the-website-id-given-by-crisp"
  crispTokenId="a-unique-token-for-the-user"
  attributes={{
    "user:email": ["foo@bar.com"],
    "user:nickname": ["foo42"],
  }}
  configuration={{
    "position:reverse": [true],
  }}
  safeMode
  crispRuntimeConfig={{
    session_merge: true,
  }}
/>
```

Le composant consiste en une fonction React :

```
function Crisp(props) {
  const {...} = props;
  ...

  if (global.$crisp === undefined) {
    global.$crisp = [['safe', safeMode]];
  }

  // Paramétrage du client avec
  // la méthode 'push' du SDK $crisp
  pushCrisp('set', attributes);
  pushCrisp('config', configuration);

  ...

  if (scripts === null) {
    // Insertion du client
    ...
  }

  return <></>;
}
```

La fonction `pushCrisp` permet de parcourir les `array` `attributes` et `configuration` puis d'appliquer la méthode `push` du SDK `$crisp`.

Une fois le composant testé et fonctionnel, il ne restait plus qu'à déployer le package sur npm.

Nous avons utilisé webpack pour créer le package :

```
module.exports = {
  entry: './src/Crisp.jsx',
  output: {
      path: path.resolve('dist'),
      filename: 'crisp.js',
      libraryTarget: 'commonjs2',
  },
  module: {
      rules: [...],
  },
};
```

A l'issue de ce projet, j'ai rédigé un article sur le blog de Mergify pour présenter notre démarche. Cet article est disponible ici : A React Crisp Component.

Bilan du projet

Participer à la création d'un projet Open Source au cours d'un stage est une expérience intéressante et atypique.

Au-delà de la création du composant, j'ai appris à publier un package sur un registre et m'assurer qu'il soit utilisable par d'autres projets.

L'exercice de la rédaction d'un article de blog m'a permis de comprendre l'importance dans le métier de développeur de savoir synthétiser et expliquer simplement sa démarche.

Créer un programme de parrainage

La croissance de Mergify se base principalement sur le système de bouche à oreille. Il était donc important pour les fondateurs de pouvoir récompenser les utilisateurs.

Le programme de parrainage est une solution simple pour répondre à ce besoin.

Cahier des charges

Les parrains et filleuls devaient se voir offrir, chacun, un siège d'une durée d'un an.

Le parrainage devait être validé à condition que le filleul souscrive une offre payante.

Le parrainage devait prendre fin soit au bout de l'année soit lorsque le filleul résiliait son abonnement.

Chaque utilisateur devait se voir attribuer un code unique de parrain qu'il pouvait partager avec son filleul.

Réalisation

Le projet a été découpé par sous-projets répartis entre le back-end et le front-end du tableau de bord.

Back-End	Front-End
Modèle pour le parrainage	Page supplémentaire sur le tableau de bord
Code de parrainage unique	Formulaire pour récupérer son code parrainage
Endpoint "referral" pour le front-end	Formulaire pour entrer un code parrainage
Prise en compte pour la facturation	Liste des utilisateurs parrainés
	Afficher le nom du parrain et la date limite du parrainage

Modèle parrainage

Le modèle créé pour le parrainage est composé de trois colonnes :

1. l'`id` du `referee` ;
2. l'`id` du `referrer` ;
3. et la date du `referral`.

```
referee_github_id = sqlalchemy.Column(
    sqlalchemy.Integer,
    sqlalchemy.ForeignKey("github_account.id"),
```

```
    nullable=False, primary_key=True,
)

referrer_github_id = sqlalchemy.Column(
    sqlalchemy.Integer,
    sqlalchemy.ForeignKey("github_account.id"),
    nullable=False,
)

referred_at = sqlalchemy.Column(
    sqlalchemy.DateTime,
    nullable=False,
    server_default=sqlalchemy.func.now(),
)
```

La colonne `referee_github_id` est une clef primaire pour éviter qu'un utilisateur ait plus d'un `referrer`.

Une contrainte a été ajoutée entre `referee_github_id` et `referrer_github_id` pour empêcher un utilisateur de se parrainer lui même :

```
__table_args__ = (
    sqlalchemy.CheckConstraint(
        "referrer_github_id != referee_github_id",
        name="check_referrer_not_referee",
    ),
 )
```

Code de parrainage

Le code de parrainage a été ajouté comme propriété du modèle utilisateur :

```
@property
def referral_code(self) -> typing.Optional[str]:
    if self.login is None:
        return None

    login = self.login.lower()

    s = login + config.REFERRAL_SECRET + login

    return f"{login}-{sum(map(ord, s)) * 8529 % 1000000:06d}"
```

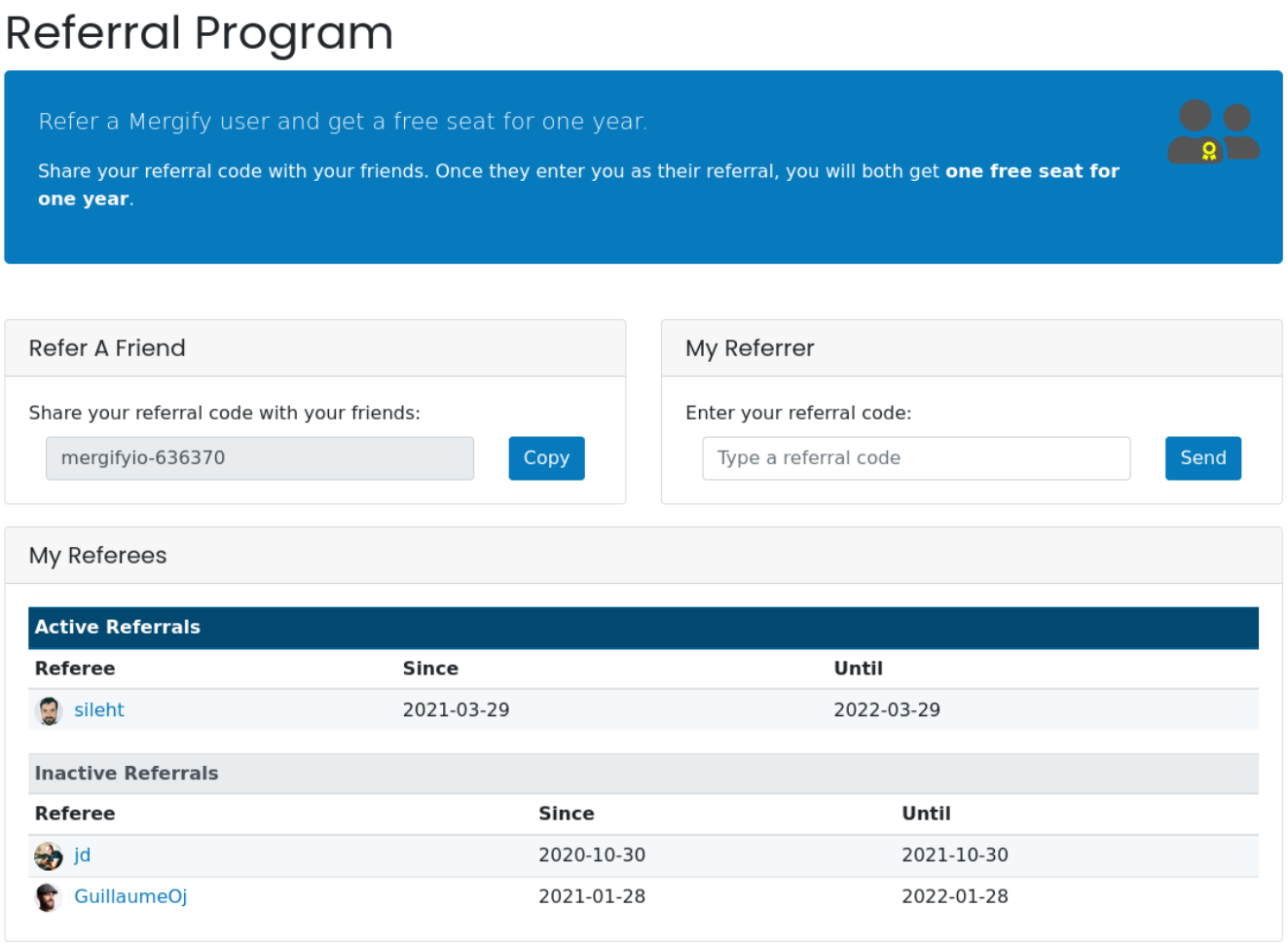
API de parrainage

Un endpoint d'API a été ajouté pour permettre la création des nouveaux parrainage depuis l'application React.

Front-End

La suite des opérations a consisté à créer une page dédiée au programme de parrainage.

Voici le résultat final :



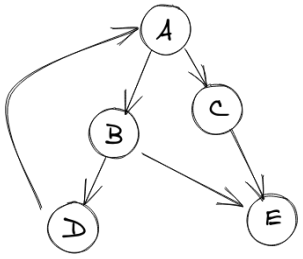
Références circulaires

Alors que le projet était sur le point d'être mis en production, Julien a soulevé un point majeur :

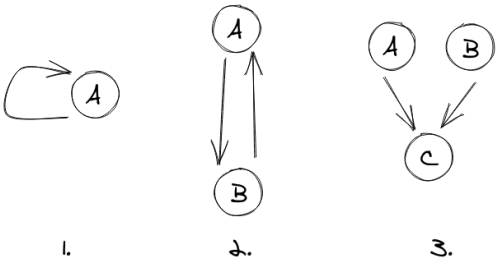
> *Que se passe-t-il si le referee essaye de parrainer le referrer de son propre referrer ?*

Etat des lieux

Pour résoudre ce problème le système de parrainage a été schématisé par un *Graphe Orienté Acyclique* :

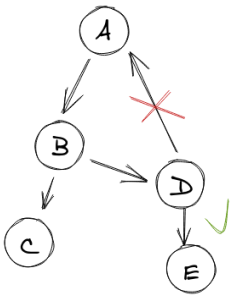


Avant de mettre en place une détection de cycle, le système de parrainage permettait déjà d'éviter les trois situations suivantes :



1. l'utilisateur se parraine lui même ;
2. l'utilisateur parraine son propre parrain ;
3. l'utilisateur est parrainé par deux parrains.

Sur le schéma ci-après, le lien entre **D** et **A** représente le cas d'une référence circulaire non souhaitée :



Prototype

Nous avons créé un système de détection de références circulaires. Le prototype de cette détection se trouve sur le dépôt suivant : [Acyclic Graph Draft].

Ce prototype se base sur deux méthodes `check_referral` et `get_referrer` :

```
def check_referral(self, referee, referrer):
    if self.get_referrer(referee):
        raise RefereeIsAlreadyReferred

    previous_referrer = self.get_referrer(referrer)

    while previous_referrer:
        if previous_referrer == referee:
            raise CircularRefer

        previous_referrer = self.get_referrer(previous_referrer)


@classmethod
def get_referrer(cls, referee):
    if not cls.referrals:
        return None

    left = 0
    right = len(cls.referrals) - 1

    while left <= right:
        middle = (left + right) // 2

        if referee.id < cls.referrals[middle].referee.id:
            right = middle - 1
        elif referee.id > cls.referrals[middle].referee.id:
            left = middle + 1
        else:
            return cls.referrals[middle].referrer
```

La méthode `get_referrer` retourne le `referrer` pour un `referee` donné.

Cette méthode se base sur un algorithme de type **Diviser pour régner**. Elle effectue une recherche dans une liste de `referrals` ordonnée par `referee`. L'algorithme est de complexité `O(log n)`.

La méthode `check_referral` vérifie un parrainage (un couple `referrer` / `referee`) en deux étapes :

1. Le `referee` a-t-il déjà un `referrer` ?
2. Le `referee` est-il dans la lignée de son `referrer` ?

Application

Nous avons décidé de mettre en application la détection au niveau de la base de données.

Pour cela nous avons utilisé une fonction récursive déclenchée à chaque insertion dans la table `referral`.

Cette fonction vérifie que le `referee` est absent de la lignée du `referrer`. Cette approche offre plusieurs avantages :

- la simplification de l'algorithme de recherche;
- l'utilisation des contraintes existantes de la table pour éliminer les cas où le `referee` aurait déjà un `referrer`.

Un article rédigé par Julien sur le blog de Mergify détaille de manière précise cette approche et la mise en place de cette fonction : Cycle detection in PostgreSQL.

Bilan du projet

Ce projet m'a permis de créer une nouvelle fonctionnalité de la conception à la mise en production.

J'ai retiré plusieurs apprentissages de ce projet :

- Une fonctionnalité aussi importante doit être découpée en plusieurs commits distincts permettant la création de "petites" pull-requests. N'ayant pas procédé de cette façon, la pull-request finale du projet est restée pratiquement deux semaines en review. Ceci s'explique par les allers-retours entre Julien et moi, mais aussi par les conflits avec les commits réguliers dans la branche master.
- Si je devais refaire ce projet, j'essaierais d'être plus rigoureux sur le découpage des fonctionnalités attendues pour avoir un rythme de travail plus fluide et un cap à tenir.
- Lorsqu'une nouvelle fonctionnalité est déployée, même si les tests sont au vert, il est important de prévoir du temps pour corriger les bugs dans les semaines suivantes.
- J'ai éprouvé de la satisfaction d'avoir mené à bien un projet en autonomie grâce à la confiance accordée par Julien et Mehdi.

Bilan

J'ai beaucoup appris auprès de Julien et Medhi et ce stage a confirmé ma volonté de travailler avec les technologies autour de Python.

Les projets sur lesquels j'ai été amené à travailler étaient très stimulants. Ils m'ont permis de découvrir comment mettre en place des outils orientés clients.

Le but affiché était de fidéliser les clients et de montrer la réactivité de Mergify face à des situations singulières.