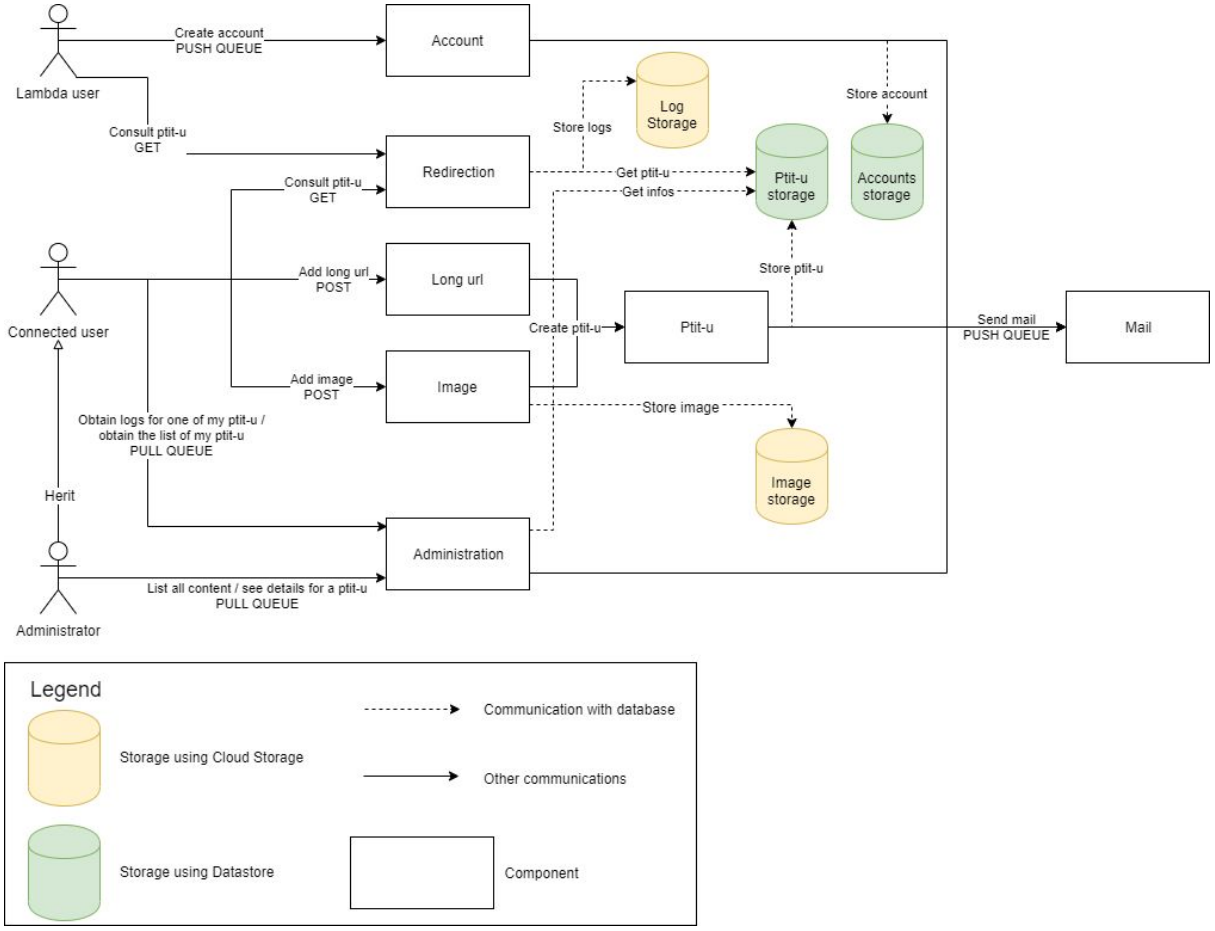
Second rendu projet Tiny-Poly

Composition de l'équipe

Benazet Laurent
Frère Baptiste
Marilier Cyril (défaillant)
Tetevi Fiacre Togni

Diagramme d'architecture



Description des composants

- **Account :** gère la création des comptes utilisateurs. Une fois créé, un mail est envoyé à l'utilisateur et le compte est stocké dans Accounts storage.

- **Redirection :** redirige automatiquement l'utilisateur vers l'url correspondant a la ptit-u. Lors d'un accès à une ptit-u, on stocke les logs correspondants dans Log Storage.

- **Ptit-u :** composant interne (on ne peut pas y accéder par une requête directe), crée une ptit-u, la stocke dans Ptit-u storage et envoie un mail la contenant.

- **Long url :** transmet la long url qu'on lui fournit à Ptit-u.

- **Image :** stocke une image dans Image storage puis demande la création d'une ptit-u correspondante à Ptit-u.

- **Administration :** fournit une interface permettant d'accéder à l'ensemble des ptit-u correspondant à un utilisateur, ainsi qu'aux détails des accès d'une ptit-u choisie.

- **Mail :** envoie des mails en fonction des différentes actions faites par l'utilisateur.

Explications de l'architecture et des communications

Nous avons considéré que les administrateurs possèdent les mêmes droits que les utilisateurs connectés, en plus de leurs droits de lister toutes les ptit-u et d'accéder aux détails de chacune d'entre elles. Les utilisateurs connectés et les administrateurs ne sont pas censés pouvoir créer un compte.

Nous avons décidé d'utiliser le protocole REST pour l'ajout d'image et d'url longue puisque l'utilisateur veut recevoir la ptit-u générée directement : la communication est synchrone. De même, lors de la redirection, l'utilisateur veut arriver sur la page visée directement.

Pour le reste des communications, nous avons décidé d'utiliser des queue, pour plusieurs raisons :

- pour les mails : nous souhaitons envoyer les mails dès que possible, car bien que la communication soit asynchrone, les utilisateurs attendent une réponse rapide. Ainsi, l'utilisation d'une push queue permet d'envoyer les mails au fur et à mesure que les requêtes arrivent sans perturber le reste du fonctionnement du système, et les requêtes seront en attente si le service de mail n'est pas disponible. Pour ce cas d'utilisation, les pull queue n'étaient pas adaptées car les mails seraient alors envoyés en batch.

- pour la création de compte : nous souhaitons que la création de compte se fasse rapidement. Cependant, le compte d'un utilisateur ne sera considéré comme actif que lorsqu'un mail aura été reçu. Ainsi, il n'est pas nécessaire d'utiliser une communication synchrone, et une push queue est adaptée. Cela permettra de plus d'augmenter la robustesse de notre système, car même si le service Account est temporairement indisponible, la création de compte pourra tout de même s'effectuer à son retour en activité.

- pour l'administration : la requête est asynchrone et la création des logs peut prendre un long moment à être traitée, puis les résultats sont envoyés par mail une fois les

données agrégées. Le protocole REST n'est pas adapté pour une requête asynchrone, et les requêtes présentes dans une push queue risquent de disparaître avant que la création des logs soit finie. Ainsi, certaines requêtes pourraient être perdues. C'est pour cela que nous avons décidé d'utiliser des pull queue pour l'administration.

Pour le moment, nous avons décidé d'utiliser la même push queue pour tous les envois de mails. Cependant, il serait peut être plus intéressant de créer une push queue pour l'envoi de mails suite à la création d'un compte, une suite à la création d'une ptit-u et une pour les réponses aux requêtes pour l'administration. Cela permettrait de séparer plus clairement les responsabilités des push queue, et la montée en charge serait plus simple à gérer, car toutes les requêtes déclenchant un envoi de mail ne sont pas utilisées à la même fréquence. Ainsi, nous ne mettrions pas en place la même élasticité au niveau de chacune des push queue.

Elasticité

Nous avons choisi d'activer l'élasticité automatique fournie par Google, avec un nombre minimum d'instance à 0. Cela nous permet de ne rien payer lorsque l'application n'est pas utilisée.

Cette élasticité est nécessaire car certains services risquent de recevoir une charge bien plus importante que les autres : nous estimons que le service de Redirection sera appelé très fréquemment, donc il est important qu'il puisse tenir la charge. De plus, sans utilisation de cache, la taille des données envoyées par ce service sera très grand. Il est donc important de mettre en place de l'élasticité pour ce service.

De même, le service d'ajout d'image va recevoir une charge importante en terme de taille des données reçues. L'élasticité est donc nécessaire.

Les autres services (ajout de long url, création de compte et administration) recevront certainement un nombre de requêtes moins important, mais comme nous souhaitons concevoir pour parer à la panne, il est important que l'élasticité automatique puisse intervenir afin de tenir un pic de charge éventuel.

Pour résumer, nous souhaitons pour le moment mettre en place de l'élasticité aux endroits critiques du système (les plus fréquemment utilisés et ceux sur lesquels la charge sera la plus grande, par exemple l'ajout d'image), mais nous avons décidé d'activer l'élasticité automatique pour parer à toute éventualité.

Types de stockage

Voici ce que nous avons décidé de stocker pour chacune des bases de données présentes sur le diagramme :

- **Ptit-u storage :** ptit-u, type de contenu, nom du contenu (URL longue ou nom de l'image), créateur du contenu. Ce stockage est important car il permet d'accéder plus

rapidement aux long url, puisqu'il les contient directement. Nous avons décidé de ne pas stocker les images ici car nous souhaitons séparer les rôles des stockages.

- **Accounts storage :** données sous forme JSON. Pour chaque compte, nous stockerons l'adresse mail correspondante, de type String, et un booléen indiquant si le compte en question est un compte administrateur ou non.

- **Log storage :** simplement un fichier texte contenant pour chaque consultation d'une ptit-u les informations importantes (ptit-u, type de contenu, créateur de cette ptit-u, IP et date de la demande, succès/échec de la demande).

- **Image storage :** id autogénérée, nom de l'image et l'image en elle même.

Les images sont des fichiers lourds. Ainsi, en utilisant la documentation fournie par Google pour présenter Datastore, nous avons remarqué qu'ils conseillent d'utiliser Cloud Storage pour les données volumineuses. Nous avons donc décidé de suivre leurs conseils. De plus, nous souhaitons que les images ne soient disponibles que cinq minutes. Cloud Storage possède une cohérence forte, qui nous assure que les images ne seront plus disponibles dès qu'on les supprimera.

Le fichier de log sera également un fichier lourd, c'est pour cela que Cloud Storage nous semble le plus adapté.

Pour le stockage des ptit-u, nous avons décidé d'utiliser Datastore. Datastore promet une "eventual consistency", c'est à dire que la consistance des données est assurée dans le futur. Cela nous suffit amplement, car il y a peu de cas de figures où cela peut nous poser problème.

Pour la création des comptes, nous utilisons également Datastore pour les mêmes raisons.

Nous considérons qu'aucune donnée de notre application n'est réellement critique. Ainsi, nous n'avons pas besoin de choisir un système proposant une disponibilité très élevée.

Nous avons décidé de ne pas mettre en place de système de cache. Cela ne nous semble pas nécessaire car les images ne seront disponibles et accessibles que pendant 5 minutes après leur création. Ainsi, le cache n'aurait aucune utilité sur un temps si court.

Description des interfaces

- **Create account :** Requête Post avec 3 paramètres. Name pour le nom de l'utilisateur, mail qui servira comme authentification dans le reste de l'application, et admin qui indique si l'utilisateur a le rôle d'admin ou non.

- **Store long url :** Requête Post avec 2 paramètres. Mail pour l'authentification et longurl pour l'url à raccourcir.

- **Store image :** Requête Post avec 2 paramètres. Mail pour l'authentification et image pour l'image à uploader.

- **Consult ptit-u :** Requête Get avec 1 seul paramètre, url pour la ptit-u qui va être redirigée.

- **List all content :** Requête Post avec 1 seul paramètre, mail pour savoir quelles url récupérer, qui envoie les informations dans une queue.

- **See detail :** Requête Post avec 2 paramètres. Mail pour l'authentification et purl pour la ptit-u dont on veut récupérer les informations. Elle envoie les informations dans une queue.

- **Send mail :** Requête Post avec 4 paramètres qui envoie les informations dans la push queue pour l'envoi de mail. senderMail et recipientMail pour l'expéditeur et destinataire du mail, subject pour le sujet et message pour le contenu du mail.

Estimation des coûts avec stockage

1er scénario, optimiste en coût :

Nous considérons que nous avons 1 000 utilisateurs par mois. Chacun d'entre eux ajoute une url longue (au maximum 50Ko) par jour. La taille de ces urls sont surestimés, en effet les urls qui seront raccourcies feront moins de 2 000 caractères (le maximum possible pour Internet Explorer), même si cela ne poserait pas de problème pour notre application. Nous décidons quand même de surestimer la taille pour garder un peu de marge.

Nous avons donc 1000x30x50 Ko = 1,5 Go par mois d'urls envoyés sur notre site.

En considérant que chaque ptit-u est vue 5 fois, on a donc 7,5Go d'url redirigés par notre site par mois.

En ce qui concerne le stockage, nous utilisons un Datastore qui a un tier gratuit de 50000 lectures et 20000 écritures par jour. On prend donc juste en compte le stockage des urls.

Avec le simulateur de google, nous obtenons un prix de 0,70€ par mois environ. Cela représente moins d'un centime par mois par utilisateur.

2ème scénario, pessimiste en coût :

Nous considérons que nous avons 10 000 utilisateurs par mois. Chacun d'entre eux ajoute cinq fois par jour une image de taille maximum, c'est à dire 4Mo.

Nous avons donc 10000x30x5x4 Mo = 6 To par mois d'images envoyés sur notre site.

En considérant que chaque ptit-u est vue 5 fois, on a donc 30 To d'images redirigés par notre site par mois.

Les images ne restant disponibles que pendant 5 minutes, on va considérer qu'elles sont réparties homogènement sur le mois, on a donc 6 To / 8640 intervalles de 5min = 700 Mo de stockage d'images en moyenne par mois, nous restons donc dans le tier gratuit du Cloud Storage que nous utilisons.

Nous devons quand même garder en mémoire les urls pour le système de logs, nous gardons la même base de 50Ko par url. Nous avons donc 10000x30x5x50 Ko = 75 Go pour les logs qui sont stockés sur un Cloud Storage.

Avec le simulateur de google, nous obtenons un prix de 3300€ par mois environ. Cela représente environ 0,30€ par mois par utilisateur.

Ces deux scénarios montrent que le prix de notre application dépendra énormément de l'usage qu'en feront nos utilisateurs. Sur une utilisation normale avec 10 000 utilisateurs mensuels on peut estimer que le prix à payer sera de moins de 500 euros par mois, puisque nous estimons que les utilisateurs normaux n'ajoutent en moyenne même pas une image par jour. Cela représenterait donc moins de 5 centimes par mois par utilisateur.