Processus de décision du projet ShortUrl

Ce document relate les questions et mon processus de réflexion relatif au projet ShortURL

● Base de données vs Algorithme de compression sans perte

Une question s'est posée, lors de ma première lecture du projet : faut-il utiliser une base de données ou un algorithme de compression sans perte.

Algorithmiquement parlant, la solution de compression sans perte me semble bien plus élégante. Elle réponds aussi à certaines demandes :

- *À partir d'une URL complète, générez une URL raccourcie.*
- *À partir d'une URL raccourcie précédemment générée, fournissez l'URL complète (originale).*
- *Le système doit être persistant, capable de « survivre » à un redémarrage.*

  o En d'autres termes, si je génère une URL raccourcie à l'aide du programme, je devrais pouvoir fermer le programme, le relancer et être capable d'obtenir l'URL précédemment raccourcie.

- *Deux URL complètes identiques doivent donner la même URL raccourcie*

Cependant, un élément me semble assez difficile à réaliser

- *Une URL raccourcie ne devrait pas compter plus de 10 caractères.*

Après avoir regardé, de manière infructueuse, s'il existe des algorithmes de compression sans perte dont on peut limiter le nombre de caractères, j'ai relu les demandes et me suis rendu compte que cette solution ne corresponds pas, car la fonctionnalité suivante ne peut pas être réalisé sans base de données:

- *Si l'URL raccourcie fournie au système ne correspond pas à une URL précédemment générée, une erreur doit être renvoyée.*

La solution algorithmique est donc à écarter et me suis orienté vers un système avec base de données.

● SQL vs No SQL

Une question s'est posée : quelle est la structure du projet est ce que je veux mettre en place.

J'ai spontanément pensé à faire un petit projet compte tenu du faible temps à ma disposition. Mais cela ne m'a pas empêché de réfléchir à une autre solution.

La solution alternative plus coûteuse que j'ai envisagé est de faire un micro-service distribué REST en Java Spring avec une base de données NoSQL (MongoDB). Le MongoDB est un NoSQL de sous-catégorie document. Dans ce cas, j'aurais désigné la structure suivante : la collection s'appellerait "URL" et les documents individuels possèderaient tous comme propriété l'url d'origine, l'identifiant du serveur ainsi qu'un timestamp en nanoseconde.

Cette solution a comme avantage d'être scalable horizontalement à l'infini et aussi de pouvoir supporter une quantité de données URL gigantesque (plus grande qu'avec de la base de données SQL).

Cependant en plus (+) d'être un peu plus (+) coûteux en temps de développement, cette solution me semble exagéré.

De plus deux autres aspects du projet me semble ne pas coller avec la solution noSql :

- Mes données à stocker étaient structurées (je devais avoir une url d'origine, une url raccourcie et un id unique) ce qui va à l'encontre de l'esprit du NoSql, qui permet un certain laisser-aller vis-à-vis du format des données

- Pouvoir garantir que mes données soient ACID (et ainsi éviter les données en base incohérentes. Cela est important entre autres car le projet utilise la propriété d'unicité de l'id. Je souhaite donc être sûr qu'une seule donnée est ajoutée à la fois.)

J’ai donc finalement opté pour une solution monolithique avec une base de données SQL.

● Synchronisation vs Non Synchronisation

Une question s’est posée : comment gérer la concurrence d’accès lors de l’insertion de données.

La première solution est de ne pas en tenir compte et de faire comme si de rien n’était. Les traitements étant rapides, les risques de transactions simultanées sont extrêmement faibles. Cependant, cela ne constitue pas une solution viable, car les risques, même si faibles, sont quand même existants.

Ensuite, je me suis interrogé. Faut-il considérer qu’il y a plusieurs instances de mon programme en parallèle ? Compte tenu de l’envergure du projet, j’ai considéré que cela n’est pas une possibilité. En effet, ce projet est limité dans le temps (une semaine) et la charge prévue est minime (et le projet ne sera jamais déployé).

Donc, il faut quand même gérer pour la seule instance du projet la concurrence d’accès.

La solution que j’ai préférée a été de rendre synchrone la fonction de recherche d’identifiant et d’insertion.

Il aurait été possible aussi de faire un “select for update” et donc de donner la responsabilité de la concurrence d’accès à la base de données. Mon projet ayant été codé avec comme base de données un H2 (base de données pas super puissante), j’ai préféré mettre un synchronize sur ma fonction et donc donner la responsabilité de la concurrence à l’application plutôt qu’à la base de données.

● Post vs Get

Une question s’est posée pour résoudre ce problème : serait-il plus intuitif de faire un get ou un post pour la requête pour aller chercher une nouvelle URL ?

En effet, les instructions requièrent la fonctionnalité suivante :

- *À partir d'une URL complète, générez une URL raccourcie.*

Ayant statué qu'il faut sauvegarder les url raccourcies pour répondre à la problématique du :

- *Deux URL complètes identiques doivent donner la même URL raccourcie*
- *Le système doit être persistant, capable de « survivre » à un redémarrage.*

  o *En d'autres termes, si je génère une URL raccourcie à l'aide du programme, je devrais pouvoir fermer le programme, le relancer et être capable d'obtenir l'URL précédemment raccourcie.*

Il faut donc pouvoir persister l'information de quelle "URL complète" donne quelle "URL raccourcie"

Ainsi, l'url raccourcie générée doit donc être persistée lorsque l'appel est effectué.

Techniquement, il s'agit donc d'un POST si l'on s'en réfère au standard. En effet, il s'agit d'une insertion dans une base de données. Alors pourquoi avoir fait un Get?

Je me suis mis à la place de l'utilisateur "naïf" de l'application. Je me suis dit qu'il est plus (+) intuitif pour lui de faire une demande GET (il n'a pas à savoir que derrière il s'agit effectivement d'une insertion en base de données). Cela lui simplifie la tâche.

Ensuite, il y avait cette demande :

- *Deux URL complètes identiques doivent donner la même URL raccourcie*

Si un premier utilisateur fait une demande pour raccourcir une URL, prenons par exemple https ://www.google.ca. Avec un post, il recevra en réponse 201 CREATED avec l'URL raccourcie. Si un deuxième utilisateur demande à faire raccourcir https ://www.google.ca, il recevra une réponse erreur 409 CONFLICT avec la réponse. Pour un utilisateur averti, il comprendra que l'URL a déjà été raccourcie. Cependant, un utilisateur naïf pourrait ne pas comprendre pourquoi il reçoit une erreur alors qu'il a une réponse.

C'est pour cela que j'ai finalement arbitré pour un get qui va à l'encontre des standards, mais qui est plus simple à s'en servir pour un utilisateur moyen.

● Redirection vs Get

Une question s’est posée : comment démontrer que la fonctionnalité de fournir l’URL originale ?

En effet, les instructions requièrent la fonctionnalité suivante :

- *À partir d'une URL raccourcie précédemment générée, fournissez l'URL complète (originale).*

D’un point de vue technique rien n’empêche de faire un simple “GET” prenant en paramètre l’URL raccourcie et restituant l’url originale.

Cependant, me plaçant du point de vue d’un utilisateur, je me suis demandé : quel est le but d’une telle application ? L’URL générée devrait être utilisable et me rediriger vers le même site que l’URL de base. À mon avis, c’est là tout le but de l’application.

Pratiquement, dans le problème donné, le service doit démontrer être capable d’aller chercher l’URL de base avec l’url raccourcie. Qu’il y ait une redirection ou non, cela démontre que la fonctionnalité est là.

N’arrivant pas à arbitrer pour un point ou un autre et sachant que faire les deux fonctionnalités ne sont pas coûteuses, j’ai décidé de ne pas décider et de faire les deux options.

● Hashage de l’Id vs Suite Mathématique

Une question s’est posée quant à la forme que devait prendre l’URL raccourcie : Comment la raccourcir ?

Il me faut une chaine de caractère compatible avec le protocole HTTP (par exemple pas de caractères non imprimables dedans), sans risque de collision avec une autre chaine de caractère précédemment générée.

Ma première solution est de faire un simple hachage (tronqué si nécessaire) de l’url originale. Rapidement, j’ai écarté la possibilité tout comme celle de générer une chaîne de caractère aléatoire. Cela m’expose trop au risque de collision “à terme”. Même si avec une quantité de 36

[ensemble des caractères alphanumériques] puissance 10 [parce que 10 caractères maximum] cas possibles, les risques, bien que minimes, existent et plus (+) l'application vivra, plus (+) elle sera exposée à ce risque. Pire, si je décide de valider que la chaîne est bien unique, les performances de l'application seront décroissantes.

Ainsi, je décide de laisser la responsabilité de générer un identifiant unique pour chaque donnée à la base de données (sachant que la base de données relationnelle est ACID et peut donc garantir mon unicité).

Cependant, cela m'expose à un autre problème : la prévisibilité des urls raccourcies précédentes et suivantes. Faut-il que j'utilise une suite mathématique ?

La suite doit avoir comme propriétés que du passage de N à N+1, les valeurs de U(n) et U(n+1) soient très différentes et que quelque soit N il n'est pas de valeur M tel que U(N) = U(M).

N'ayant pas trouvé "chaussure à mon pied", j'ai finalement opté pour un hachage de l'identifiant unique : transformer le "long" de l'identifiant unique auto incrémenté de la base de données qui est en base 10 en un String qui est en base 36.

Cette solution n'est donc pas optimale, mais rien dans l'énoncé ne m'imposait cette contrainte.