Architecture 3-tiers

Skander Zannad et Judicaël Courant

2014-05-01

1 Résumé des épisodes précédents

On a vu :

- calcul relationnel ;
- algèbre relationnelle ;
- (une partie de) SQL ;
- le lien entre les trois.

2 Et en pratique ?

1. Architecture client-serveur ;
2. Architecture trois-tiers ;
3. Applications web.

3 Architecture client-serveur

3.1 Principe

Accès à une base de données SQLite :

- se fait **localement** à la machine par le frontal (front-end) interactif en mode texte `sqlite3` ;
- situation atypique.

Situation typique :

- La base de données tourne sur une machine (`mabd.example.org`), appelée *serveur* ;

- On la consulte/modifie depuis une autre machine (par exemple `monpc.example.org`), dite *cliente* avec un logiciel, dit *client*.

D'où le terme : architecture client-serveur.

Typiquement :

- un (voire quelques) serveurs ;
- plusieurs (voire nombreux) clients.

3.2 Remarque culturelle

L'architecture client-serveur est omniprésente aujourd'hui, pas seulement pour les bases de données.

Exemples :

Web Pour la consultation de pages hypertextes (protocole HTTP).

Messagerie Courrier électronique (POP, IMAP, SMTP).

DHCP Pour se faire attribuer une IP par sa box.

DNS Pour trouver l'IP d'une machine dont on connaît le nom.

Temps Pour mettre à l'heure son ordinateur (NTP).

Systèmes de fichiers Accès à des fichiers distants (SMB, NFS).

3.3 Problèmes soulevés

Le passage en réseau complique tout :

Et si...

(Sécurité) un méchant veut écrire n'importe quoi dans la base ?

(Sûreté) le réseau plante au milieu d'une modification ?

(Concurrence) deux personnes modifient en même temps la base ?

NB : différence sûreté/sécurité (absence/présence d'un adversaire) Solutions :

Sécurité Contrôle d'accès

Sûreté/Concurrence Gestion de transactions.

Sécurité : difficile à assurer mais on a une idée de ce que c'est.

Transactions : qu'est-ce ?

3.4 Notion de transaction

Transaction :

- séquence de commandes SQL à exécuter ;
- sur lesquels on garantit les propriétés ACID¹.

1. Rien à voir avec *Lucy in the Sky with Diamonds*.

ACID :

- Atomicité
- Cohérence
- Isolation
- Durabilité

3.4.1 Atomicité

Une transaction échoue en entier ou réussit en entier

Exemple : On enlève 100 euros d'un compte bancaire pour les ajouter à un autre. En cas de problème sur la deuxième opération, on veut éviter que la première soit enregistrée.

3.4.2 Cohérence

À la fin d'une transaction, on doit avoir une base cohérente.

Exemple : on ne veut pas supprimer une personne de notre base cinématographique si la table des films y fait référence.

3.4.3 Isolation

Les états intermédiaires de la base durant une transaction sont invisibles à tout autre utilisateur de la base.

Exemple sur un système de réservation de places de spectacles. Vous avez une place au balcon, vous la changez pour une place dans l'orchestre.

Deux scénarios à éviter :

1. Vous libérez la place, un autre client réserve les deux places restantes et vous n'avez plus de place !
2. Vous prenez la place orchestre, un autre client vient, constate qu'il n'y a plus de place disponible, vous libérez la place balcon et une place reste invendue !

Autrement dit : le système doit garantir que l'exécution *parallèle* (simultanée) de deux transactions se passe comme si les deux transactions avaient eu lieu de façon *séquentielle* (l'une après l'autre).

3.4.4 Durabilité

Une fois que la transaction a été acceptée, elle reste enregistrée même si un problème survient.

Exemple : on vous confirme votre achat d'un billet de train (par le web), une coupure électrique a lieu à la SNCF. Avez-vous perdu 100 euros ?

3.4.5 Les SGBD sont ACID

Les SGBD :

- comportent une notion de transaction ;
- permettent de valider la transaction en cours (*commit*) ;
- ou de l'annuler (*rollback*) ;

Les SGBD garantissent les propriétés ACID à condition :

- d'une mise en œuvre correcte sur le plan logiciel (en particulier utilisation correcte des transactions) ;
- d'une mise en œuvre correcte sur le plan matériel ;
- sous des conditions opérationnelles raisonnables (coupure électrique ok, bombe atomique ko).

3.5 Résumé : client-serveur

Le passage en réseau soulève des problèmes :

de sécurité qu'on sait à peu près résoudre ;

de concurrence pour lequel on utilise des transactions.

Mais une architecture client SQL/serveur SQL ne répond en général pas à tous les besoins, c'est pourquoi on l'étend en une architecture dite 3-tiers.

4 Architecture 3-tiers

4.1 Les tiers selon Pagnol (*Marius*)

CÉSAR : Tu mets d'abord un tiers de curaçao. Fais attention : un tout petit tiers. Bon. Maintenant, un tiers de citron. Un peu plus gros. Bon. Ensuite, un BON tiers de Picon. Regarde la couleur. Regarde comme c'est joli. Et à la fin, un GRAND tiers d'eau. Voilà.

MARIUS : Et ça fait quatre tiers.

CÉSAR : Exactement. J'espère que cette fois, tu as compris. (Il boit une gorgée du mélange).

MARIUS : Dans un verre, il n'y a que trois tiers.

CÉSAR : Mais, imbécile, ça dépend de la grosseur des tiers !

Attention : en matière de bases de données, « tiers » ne signifie pas 1/3 mais « couche ». On peut donc avoir des architecture 4-tiers.

4.2 Motivations

Besoins pour une application classique utilisant une BD :

- Être utilisable par des gens réfractaires à SQL ordinaires ;
- Gérer la logique de l'application (par exemple, valider un paiement bancaire) ;
- Gérer les données.

Ces besoins conduisent à une structuration en trois couches :

1. Couche présentation : l'interface utilisateur ;
2. Couche logique (ou applicative, ou métier) : gère la logique de l'application (par exemple, déclencher l'envoi d'un bien acheté après paiement) ;
3. Couche accès aux données.

4.3 En pratique

On trouvera :

1. Un (logiciel) client graphique (interface graphique) ;
2. Un serveur applicatif ;
3. Un serveur de base de données.

Le serveur applicatif répond aux requêtes du client en consultant le serveur de base de données.

Le client et les serveurs peuvent être ou non sur la même machine. En général :

1. Le client est sur un ordinateur de bureau ;
2. Les deux serveurs sont sur une même machine puissante ;
3. On met les serveurs sur des machines différentes si le travail à fournir est trop important pour une seule machine.

4.4 Problème

Programmer une interface graphique est très long et coûteux :

- Parce que c'est difficile à tester automatiquement ;
- Parce qu'il y a une grande variété de machines clientes ;
- Parce que programmer une interface graphique sur l'OS le plus répandu (MS-Windows) est particulièrement pénible (sans comparaison avec son concurrent principal) ;

Et de plus, c'est compliqué à déployer (il faut l'installer sur chaque machine).

4.5 Un peu d'histoire

1993 Invention d'un système de documentation hypertexte appelé *world wide web*. Permet de distribuer des documents qui peuvent avoir des liens vers d'autres documents distribués de la même façon.

1995 (env.) Extensions au protocole pour permettre d'interagir avec le serveur et non juste de lire des documents.

4.6 Le client graphique universel

Le navigateur web est devenu le client graphique universel :

- Disponible sur toutes les plates-formes ;
- Déjà installé sur toutes les machines de bureau.

4.7 Applications web

Nom donné aux applications dont l'interface est le navigateur web.

Évolution à mentionner :

- Le navigateur web ne connaît rien à l'application ;
- Donc il faut lui dire quelles données présenter et comment : ça s'ajoute au travail du serveur applicatif.

Défaut des applications web : souvent moins réactives que des vraies applications. Mais ça s'améliore nettement.

Technologies les plus souvent utilisées : LAMP

Linux pour l'OS ;

Apache comme serveur Web ;

PHP langage spécialisé pour les applications web (utilisé en conjonction avec Apache) ;

MySQL comme serveur de base de données.

5 Projet : réalisation d'une application

But : réaliser une application permettant de consulter (voire de modifier) une base de données.

Le cadre :

- On utilisera SQLite et non un serveur de base de données ;
- On utilisera python pour la partie applicative ;
- Chacun fera tourner l'application sur sa machine (celle où tourne le navigateur) et non sur un serveur ;
- On se préoccupera au minimum des questions de sécurité et de gestion des transactions.

5.1 SQLite et python

Module standard pour l'utilisation de sqlite3 :

```
import sqlite3
```

5.1.1 Connexion et création d'un curseur

On se connecte à la base de données. En général, on donne le nom de la machine où est le serveur de base de données mais pour SQLite, on donne juste le nom du fichier :

```
conn = sqlite3.connect('films.db')
```

On crée ensuite un curseur (objet servant à traiter les réponses de la base) :

```
c = conn.cursor()
```

5.1.2 Exécution de requêtes

On peut alors exécuter des requêtes :

```
c.execute("""SELECT*_FROM_PERSONNE
WHERE_prenom=? Clint """)
```

Attention : `c.execute` ne retourne pas la réponse mais demande juste à la base de données de calculer la réponse.

5.1.3 Récupération des résultats

On peut ensuite récupérer la réponse, ligne par ligne :

```
>>> c.fetchone()
(3, u'Eastwood', u'Clint', u'1930-05-31')
>>> c.fetchone()
>>>
```

(retourne None quand toutes les lignes ont été retournées).

Ou en une seule fois :

```
>>> c.execute("""SELECT*_FROM_PERSONNE""")
<sqlite3.Cursor object at 0x7f7ff6f375e0>
>>> c.fetchall()
[(1, u'Kubrick', u'Stanley', u'1928-07-26'),
 (2, u'Spielberg', u'Steven', u'1946-12-18'),
 (3, u'Eastwood', u'Clint', u'1930-05-31'),
 (4, u'Cumberbatch', u'Benedict', u'1976-07-19'),
 (5, u'Freeman', u'Martin', u'1971-09-08'),
 (6, u'Leone', u'Sergio', u'1929-01-03'),
 (7, u'McGuigan', u'Paul', u'1963-09-19'),
 (8, u'Sellers', u'Peter', u'1925-09-08')]
```

On peut aussi itérer sur le curseur :

```
>>> c.execute("""SELECT*_FROM_PERSONNE""")
<sqlite3.Cursor object at 0x7f7ff6f375e0>
>>> for r in c:
    print r

(1, u'Kubrick', u'Stanley', u'1928-07-26')
(2, u'Spielberg', u'Steven', u'1946-12-18')
(3, u'Eastwood', u'Clint', u'1930-05-31')
(4, u'Cumberbatch', u'Benedict', u'1976-07-19')
(5, u'Freeman', u'Martin', u'1971-09-08')
(6, u'Leone', u'Sergio', u'1929-01-03')
(7, u'McGuigan', u'Paul', u'1963-09-19')
(8, u'Sellers', u'Peter', u'1925-09-08')
>>>
```

5.1.4 Transactions

Si on a effectué des modifications, on peut les valider avec

```
conn.commit()
```

ou les annuler avec

```
conn.rollback()
```

Quand on a terminé, on peut fermer la connexion. Penser à valider les changements auparavant s'il y en a !

```
conn.close()
```

5.1.5 Requêtes paramétrées

Comment écrire une fonction prenant en argument un curseur et la clé primaire d'une personne et retournant l'enregistrement ayant cette clé primaire ?

La méthode `execute` accepte un second argument qui est un n -uplet de paramètres. Les « ? » dans la requête seront alors remplacés par les valeurs données :

```
def personne(c, id):
    p = (id, ) # le 1-uplet des paramètres
    req = """SELECT*_FROM_PERSONNE_WHERE_id=? """
    c.execute(req, p)
    r = c.fetchall()
    assert len(r) == 1 # id est une clé primaire
    return r[0]
```

Attention : essayer de construire soi-même la chaîne à la main (en concaténant ""SELECT*_FROM_PERSONNE_WHERE_id=_"" avec **str(id)**) est une **grave erreur** car risque d'ouvrir la porte à des attaques par injection de code SQL (cf <http://xkcd.com/327/>).

— problème de sécurité No 1 des applications web ;

— nombre de victimes : supérieur à 10^8 :

Fév. 2002 200 000 numéros de cartes de crédit exposés ;

Août 2009 vol de 130 millions de numéros de cartes de crédit ;

Déc 2009 vol de 32 millions de mots de passe de Rockyou ;

Juin 2011 vol d'un million de mots de passe chez Sony.