Groupe_1_6

Livrable R3.02

SAE S3.01

Le présent document vous est présenté par le **Groupe_1_6**, composé des membres suivants :

- XXXX
- LAURENS Thibaud
- YYYY

Celui-ci a pour but de vous présenter notre travail de réflexion autour de l'efficacité du code et l'analyse des résultats.

Nous vous présenterons dans l'ordre les points définis dans le sommaire suivant :

Sommaire :

Application correcte **2**

1) Fonctionnalités et précondition 2
2) Résultats 3
3) Assertions - tests 3

Application efficace **5**

1) Traitement des données 5
2) Montée en charge 5
3) Plan de dimensionnement 6
4) Choix d'hébergement du serveur des données 7

Application correcte

1) Fonctionnalités et précondition

Notre site, SUNDAY, dispose de multiples fonctionnalités adaptées aux clients, employés en charge de la gestion du site et administrateurs (des sortes de super-employés disposant d'un contrôle total sur tous les utilisateurs). Pour donner deux exemples :

- Pour les clients, passer une commande avec la méthode "commander()". Il y a trois pré-requis (précondition) : être connecté sur un compte client, avoir rempli le formulaire de la page "commande_formulaire.php" (unique moyen d'accéder à cette méthode) et disposer d'au moins un article dans son panier (car une commande sans article, c'est une commande terminée). Cela se traduit par les ligne suivantes :

```
// Vérifier qu'il y a une session avec rôle client
assert($this -> session -> get('isLoggedIn') && $this -> session
-> get('role') === 'client', "N'est pas connecté comme client");

// Infos récupérées par le formulaire
assert($this -> request -> getPost("adresse_livraison") &&
$this -> request -> getPost("adresse_facturation") &&
$this -> request -> getPost("nom_acheteur") &&
$this -> request -> getPost("prenom_acheteur") &&
$this -> request -> getPost("nom_receveur") &&
$this -> request -> getPost("prenom_receveur") , "N'a pas rempli
le formulaire");

// Vérifier qu'au moins un enregistrement du panier référence
l'id du client
assert(0 < $this -> modelPanier ->
where("id_client",$_SESSION["user_id"]) -> countAllResults(); ,
"Le panier est vide");
```

- Pour les administrateurs, supprimer n'importe quel utilisateur avec les méthodes "deleteEmp()" et "deleteClient()". Leur précondition comprend deux éléments : être connecté en tant qu'administrateur et avoir dans l'url "?id=x" où x est un nombre entier (correspondant à l'identifiant de l'utilisateur à supprimer). Cela s'implémente de la façon suivante :

```
// Vérifier qu'il y a une session avec rôle administrateur
```

```
assert($this -> session -> get('isLoggedIn') && $this -> session
-> get('role') === 'admin', "N'est pas connecté comme admin");

// Infos récupérées par le formulaire
assert($this -> request -> getGet('id'), "N'a pas donné
d'identifiant à supprimer");
```

Deux peuvent paraître peu, cependant il s'agit d'exemples traitant nos préconditions les plus fréquentes : informations par GET et POST, utilisateur authentifié avec un certain rôle et occurrences dans les tables. Présenter d'autres fonctionnalités (authentification, ajout de produit au panier, formulaires de création / modification / suppression de compte / produit / commande, etc.) n'apporterait pas de contenu pertinent à ce rapport.

2) Résultats

La méthode "commander()" a pour but de créer une nouvelle commande à partir des enregistrements du panier associés au client courant. Plus précisément, après vérification de la précondition, la méthode récupère les éléments du panier correspondant au client actuel. Ensuite, pour définir l'id de la commande (clé primaire), elle prend le dernier id connu de la table commande et l'incrémente (s'il n'y a pas encore de commande, l'id est 1). De cela, elle insère une ligne de commande (table LigneCommande) par élément récupéré (la table LigneCommande a les mêmes attributs que la table Panier) et somme le montant des lignes de commande pour calculer le montant total de la commande. Enfin, la commande est créée avec l'id calculé, le montant calculé et les informations récupérées par le formulaire. Il ne reste plus qu'à retirer du panier ce qui correspond au client et à le renvoyer dans sa liste de commandes.

La méthode "deleteEmp()" (resp. "deleteClient()") supprime l'employé (resp. le client et toutes ses données) de la base de données si l'identifiant donné correspond. Après vérification de la précondition, la méthode récupère l'identifiant à supprimer. Pour les employés/admins, elle supprime directement de la table Employe l'entrée qui aurait comme clé primaire cet identifiant. Pour les clients, elle fait appel à une méthode du modèle ClientModel qui supprime toutes les références au client dans Panier, Favori et remplace son identifiant dans Commande par -1, avant de supprimer le client de la table Client s'il y est. Si la suppression n'est pas effective, aucune incidence ne se produira sur le système.

3) Assertions - tests

Les assertions de l'ensemble des méthodes présentées sont définies plus haut. Dans le code, nous les avons plutôt remplacées par des conditions "if" : si la précondition n'est pas validée, on renvoie l'utilisateur sur la page d'accueil du site (ou l'accueil de l'espace administratif dans le cas des employés et administrateurs).

La méthode se teste d'elle-même, car après exécution de son code le client est ramené sur une vue montrant ses commandes. On y voit alors directement que la nouvelle commande est ajoutée à la base de données, et on peut cliquer sur son id pour lister toutes les lignes de commande se rapportant à cette commande et constater qu'elles représentent bien ce

qui se trouvait dans le panier. Elle finissent dans tous les cas par rediriger l'administrateur vers la vue affichant tous les utilisateurs enregistrés du site.

Concernant "deleteEmp()" et "deleteClient()", les deux retournent par GET un attribut indiquant si la suppression est effective ou non ("?delete=error" ou "?delete=success"). Cette information est utilisée par la vue montrant les utilisateurs pour afficher un message confirmant le succès ou l'échec de l'opération, permettant de tester simplement. En principe, la suppression ne peut échouer que si l'administrateur a lui-même écrit l'url pour insérer l'identifiant qui n'existerait pas (autrement, il n'accède à cette méthode qu'en appuyant sur un bouton existant uniquement si un identifiant est associé).

Application efficace

1) Traitement des données

Le traitement des données au niveau de la base de données est optimal pour la quantité de données car les tables respectent la normalisation 3NF :

- Toute table possède une clé primaire (parfois concaténation de deux attributs)
- Les valeurs de tous les attributs des tables sont atomiques (nombre entier ou décimal, chaîne de caractères, booléen, date)
- Tous les attributs non-clés des tables dépendent entièrement de la clé primaire
- Tous les attributs non-clés des tables dépendent uniquement de la clé primaire (pas de transitivité)

(explications interprétées depuis https://formations.imt-atlantique.fr/bd_ihm/fr/normalisation/normal_forms/).

De plus, les requêtes vers ces BD utilisent exclusivement les méthodes prévues par le framework CodeIgniter4 (depuis les modèles : “where”, “find”, “findAll”, “insert”, “update”, “delete”, “save”), en principe optimisées (voir la documentation : https://codeigniter.com/user_guide/models/model.html).

En revanche, nous nous servons de SQLite3 comme SGBD. Conscients de ses limites par rapport au nombre d’écritures en simultané, nous l’avons choisi car il est nativement intégré à CodeIgniter4 et donc optimisé par ce dernier, gère très bien les accès concurrents en lecture, et que la communication réseau s’effectue avec un fichier au lieu d’un serveur de bases de données, ce qui augmente la vitesse d’exécution et réduit donc les coûts temporels tout en permettant de copier les données pour une sauvegarde externe ou une migration très facilement.

2) Montée en charge

Comme expliqué, le choix du SGBD par SQLite risque de poser des difficultés lors des montées en charge si trop d’écritures dans le fichier s’effectuent en simultanée.

Les fonctionnalités risquant de consommer le plus de temps sont, selon nous :

- méthode “*commander()*” : récupère le contenu du panier, crée une ligne de commande par produit récupéré, crée la commande, supprime les

enregistrements du panier associées au client courant. Chacune de ces actions est un accès à la BD.

- méthode "*deleteData(int $idClient)*" : supprime les éléments du panier associés au client, supprime les favoris associés au client, passe l'id_client de toutes les commandes associées au client à "-1", supprime le client. Cela peut demander en effet beaucoup de suppressions.

- méthode "*updateProduitPost()*" : peut, selon les actions de l'utilisateur : mettre à jour la table produit, chercher un produit, ajouter des enregistrements dans la table produit, supprimer des éléments de la table produit, supprimer de la table Image, ajouter dans la table Image.

En analysant le temps de chargement de ces fonctions, on voit qu'ajouter un client prend 120ms, "*commander()*" avec un seul produit dure 100ms, "*updateProduitPost()*" avec un maximum de modifications un peu moins de 100ms aussi.

Au maximum, nos vues chargent les données de deux tables complètes (et cela n'arrive qu'une seule fois). Si la taille des tables est petite, cela ne produit aucune latence (actuellement, ces pages se chargent en 50 - 60 ms). En revanche, des tables avec des centaines de données pourraient bien augmenter le temps de chargement pour : l'affichage de tous les produits du catalogue, l'affichage de toutes les commandes passées par les clients, l'affichage de tous les clients, l'affichage de tous les utilisateurs du site possédant un compte.

Actuellement, la vue la plus lente à charger est la page d'accueil, pour 160ms (source : clic droit sur une page -> inspecter -> réseau).

3) Plan de dimensionnement

L'organisation des données est défini par le framework, lequel respecte la structure MVC : les utilisateurs n'ont accès qu'à la Vue ; leurs actions sur cette dernière déclenche en interne des Contrôleurs ; ces contrôleurs effectuent diverses actions en s'appuyant sur les Modèles ; les modèles sont les uniques accès aux bases de données (un modèle par table exactement). (résumé de https://codeigniter.com/user_guide/concepts/mvc.html). Seules les données nécessaires sont traitées selon le contrôleur employé et sa méthode, signifiant que certaines actions de l'utilisateur n'ont même pas recours à un modèle (pas d'accès inutile).

En principe, l'ensemble du site et de ses données est contenu sur un unique serveur ; le déploiement est facilité par quelques fichiers explicatifs (l'un présente les actions à effectuer pour un déploiement local, un autre un exemple de configuration du

réseau local et de la machine sur lesquels déployer le site en question). Ils seront joints au présent rapport.

À long terme, si la quantité de données et le nombre d'utilisateurs venait à croître fortement, une migration des données vers un réel serveur de bases de données serait nécessaire. Cette migration ne se fera pas en deux ou trois étapes, mais des tutoriels sur Internet expliquent de manière claire la chose (voir https://www.geeksforgeeks.org/mariadb/how-to-migrating-data-from-sqlite-to-mariadb/). Une migration vers MariaDB serait le meilleur choix car ce SGBD est globalement plus rapide que MySQL (source : https://aws.amazon.com/fr/compare/the-difference-between-mariadb-vs-mysql/).

4) Choix d'hébergement du serveur des données

Comme expliqué jusque-là, le site tel quel ne se sert pas d'un serveur de données afin de réduire les coûts temporels. Cependant, s'il est amené à en nécessiter un (pour traîter plus de données), le choix entre un hébergement local et cloud se poserait.

Si un hébergement en local sur le même réseau que le serveur hébergeant le site est la solution la plus efficace en temps (les données ont moins de chemin à parcourir), c'est une méthode impliquant des connaissances en gestion d'infrastructure réseau et dangereuse : d'une part, en cas de montée en charge, il faudrait soi-même agir pour trouver des mesures correctives. D'autre part, la sécurisation du serveur reviendrait au gestionnaire du serveur.

L'hébergement cloud est une solution évolutive (utilise plusieurs serveurs dispersés), scalable (pas de problème avec la montée en charge), sécurisée et maintenue par les fournisseurs du service cloud (source : https://cloud.google.com/learn/what-is-cloud-hosting?hl=fr). De ce fait, le cloud est une solution infiniment plus simple et moins risquée, bien qu'elle soit en pratique plus lente.