151 - Intégrer des éléments de base de données dans des applications Web

Rapport de Projet

Date de création : 29.01.2024  
Version 2 du 26.02.2024

Simon Gendre

|  |  |  |
| --- | --- | --- |
|  | Module du : | 29.01.2024 |
| Au : | 05.03.2024 |

Table des matières

1. Introduction 3

1.1. Le Projet 3

1.2. Objectifs du module 3

2. Analyse 4

2.1. Le projet 4

2.2. Planning 4

2.2.1. Use Cases 5

2.3. Maquettes 6

2.3.1. Écran de login 6

2.3.2. Mode invité 6

2.3.3. Mode connecté 7

2.4. Diagrammes d’activités (une action) 7

2.5. Diagrammes de séquences (une action) 8

2.6. Schéma ER 8

3. Conception 9

3.1. Diagramme de classe 9

3.1.1. Client 9

3.1.2. Serveur 9

3.2. Diagramme de séquence interactions 9

3.3. Schéma relationnel de la base de données 10

3.3.1. Table des utilisateurs 11

3.3.2. Table des salles de tchat 11

3.3.3. Table des messages 11

4. Réalisation 13

4.1. Descente de code 13

4.2. Tests fonctionnels 13

4.3. Problèmes rencontrés 13

4.4. Hébergement 13

5. Synthèse 14

5.1. Différence de timing 14

5.2. Conclusion 14

5.2.1. Ce que je retiens de ce module 14

5.2.2. Amélioration / proposition 14

5.2.3. Mes points forts et faibles 14

6. Glossaire et Bibliographie 15

6.1. Bouts de code 15

6.2. documentation 15

# Introduction

## Le Projet

Durant ce module 151, nous allons créer une application web client-serveur en PHP et JavaScript.

## Objectifs du module

# Analyse

## Le projet

Mon projet s’intitule Semaphor et consiste en une application de messagerie en direct. Les utilisateurs peuvent lire les anciens messages et doivent se loguer pour en envoyer.

Il y aura une chatroom principale et éventuellement d’autres chatroom

## Planning

Voici le planning que j’ai estimé pour ce projet. L’analyse et la conception seront fait en semaine 1 et 2. La réalisation sera faite entre les semaines 2, 3, 4 et 5.



### Use Cases

Voici les Use Cases de l’application. Il y aura trois acteurs différents : les visiteurs, les utilisateurs et les administrateurs. Voici ce qu’ils peuvent faire :

|  |  |
| --- | --- |
| Acteur | Actions |
| Visiteur | Se loguer / créer un compte / voir les anciens messages de la room principale |
| Utilisateur | Se déconnecter / envoyer des messages / rejoindre et créer des room |
| Administrateur | Même chose que les utilisateur / supprimer des messages |



## Maquettes

Voici les maquettes du site WEB. Ce ne sont que des squelettes qui permettent de se donner une idée du résultat final. Ce dernier risque d’être légèrement différent dans le design mais pas dans les options.

### Écran de login

Une fois arrivé sur le site, les visiteurs seront accueillis par une page de login. Ils pourront, soit se connecter/créer un compte, soit accéder au site en mode invité.

Une image contenant texte, capture d’écran, logiciel, Icône d’ordinateur

Description générée automatiquement

### Mode invité

Le mode invité permet de voir les messages envoyés dans la room principale. C’est tout.

Une image contenant texte, capture d’écran, nombre, ligne

Description générée automatiquement

### Mode connecté

Le mode connecté permet aussi de voir les messages et d’en envoyer mais il sera alors possible de changer de room en en créant une ou en en rejoignant une.

Une image contenant texte, capture d’écran, nombre, logiciel

Description générée automatiquement

## Diagrammes d’activités (une action)

Voici le diagramme d’activité pour envoyer un message.



## Diagrammes de séquences (une action)



## Schéma ER

Voici à quoi ressemblera la base de données du serveur. Elle aura trois tables qui seront relié entre-elles. La table « user » contiendra les informations de login (nom et hash de mot de passe), la table room servira à séparer les messages en salles de chat différentes et pour finir, les messages contiendront un texte, une date et les informations qui les relie aux autres tables.



# Conception

## Diagramme de classe

### Client

Une image contenant texte, diagramme, capture d’écran, ligne

Description générée automatiquement

### Serveur

Une image contenant texte, capture d’écran, diagramme, Police

Description générée automatiquement

## Diagramme de séquence interactions

Une image contenant texte, diagramme, Plan, Parallèle

Description générée automatiquement

## Schéma relationnel de la base de données

Voici les trois tables définit à l’analyse. La table centrale est celle des messages, elle est reliée avec un utilisateur et une salle de tchat.

Une image contenant texte, capture d’écran, nombre, Police

Description générée automatiquement

### Table des utilisateurs

La table des utilisateurs contient simplement les champs, username, hash et un booléen pour les droits d’admin. Tout ces champs sont non-nul.

Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement

### Table des salles de tchat

La table de tchat ne contient qu’un nom qui est aussi une PK. J’ai décidé d’en faire une table à part (à la place d’un attribut dans le message) afin de pouvoir évoluer en ajoutant des champs liés aux salles dans le futur (comme le créateur).

Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement

### Table des messages

La table des messages stockera le texte du message, son auteur, la date d’envoi et la salle où il a été envoyé. Tous les champs sont non-nul.

Une image contenant texte, capture d’écran, nombre, Police

Description générée automatiquement

# Réalisation

## Descente de code (envoi de messages)

### Client

VueCtrl.js

Dans le contrôleur de l’interface, lorsque que le chat est chargé, deux listeners sont définis. Les deux font la même action, c’est-à-dire demander au contrôleur d’envoyer le message mais un des deux est sur le bouton « envoyer » et l’autre permet d’envoyer en appuyant sur « entrée »

loadChat() {

$("#send-btn").click((event) => {

var texte = $("#message-input").val();

$("#message-input").val("");

ctrl.sendMessage(texte, localStorage.getItem("currentRoom"));

});

$("#message-input").keypress(function(event) {

// le code 13 correspond à la touche entrée

if (event.which === 13) {

var texte = $("#message-input").val();

$("#message-input").val("");

ctrl.sendMessage(texte, localStorage.getItem("currentRoom"));

}

});

ctrl.loadRoom(localStorage.getItem("currentRoom"));

ctrl.autoReloadMessage();

}

Ctrl.js

Dans le contrôleur, le message est vérifié pour qu’il ne soit pas vide et ne contienne pas d’injection HTML avant d’être transféré vers le service http. Les méthode « retour » sont aussi définies ici et seront appelé une fois que le serveur enverra une réponse.

sendMessage(texte, room\_id) {

if (!texte || !texte.trim()) {

console.error("Text is null, empty, or contains only spaces.");

return;

}

if (room\_id === null) {

console.error("Room ID is null");

return;

}

// empeche les injection HTML (merci chatGPT)

texte = $("<div>").text(texte).html();

console.log("Sending message:", texte, "to room:", room\_id);

this.http.sendMessage(

texte,

room\_id,

ctrl.sendMessageSuccess,

ctrl.sendMessageError

);

}

sendMessageSuccess(data, text, jqXHR) {

console.log("message envoyé !");

ctrl.loadRoom(localStorage.getItem("currentRoom"));

}

sendMessageError(data, text, jqXHR) {

if (data.status == 413) {

// Le message est trop long

ctrl.vue.afficheStatut("Le message est trop long. La longueur maximale est de 160 caractères.", "red");

} else if (data.status == 500) {

// Une erreur s'est produite

ctrl.vue.afficheStatut("Petit problème sur le serveur...", "red");

} else if (data.status == 403) {

// Pas logué

ctrl.vue.afficheStatut("Pas autorisé ! Va te loguer petit chenapan.", "red");

} else {

ctrl.vue.afficheStatut("Le serveur ne répond pas...", "red");

}

}

http.js

le service http permet d’envoyer des requêtes GET, PUT, POST, DELETE au serveur. Dans notre cas, c’est une requête POST qui est envoyée avec les paramètres suivants : action de type « message », puis le texte et la salle.

sendMessage(texte, roomId, successCallback, errorCallback) {

$.ajax({

type: "POST",

url: this.BASE\_URL,

data: {

action: "message",

texte: texte,

room\_id: roomId

},

xhrFields: {

withCredentials: true

},

success: successCallback,

error: errorCallback,

});

}

### Serveur

Index.php

Sur l’index, un switch permet d’exécuter du code en fonction du type de requête (GET, PUT, POST, DELETE) puis dans le cas d’un POST, en fonction du paramètre « action » certains bouts de code sont exécutés. Voici un zoom sur le cas d’un « POST » avec l’action « message » qui permet de stocker un nouveau message.

case 'POST':

if (isset($\_POST['action'])) {

$action = $\_POST['action'];

switch ($action) {

case [...]

case 'message':

parse\_str(file\_get\_contents("php://input"), $vars);

if (isset($vars['texte']) && isset($vars['room\_id'])) {

$texte = $vars['texte'];

$room\_id = $vars['room\_id'];

// Envoyer un message

// Forward vers MessageManager.php

// test le message avant de le transférer.

// si il est trop long (>160 char) ça passe pas. (oui c'est une référence à la limite historique des SMS)

if (strlen($texte) > 160) {

http\_response\_code(413);

echo '<error>Le texte est trop long. Maximum 160 caractères. Longueur du message : ' . strlen($texte) . '</error>';

} else {

// echo "enregistrer un message avec : texte ($texte), user ($user) et room\_id ($room\_id) .<br>";

$messageManager = new MessageManager();

echo $messageManager->send($room\_id, $texte);

}

} else {

echo '<error>Paramètre texte, user ou room\_id manquant pour un nouveau message</error><br>';

}

MessageManager.php

Voici la méthode « send » du gestionnaire de messages qui est appelée par l’index. Cette méthode contrôle que la session actuelle a le droit d’envoyer un message (est loguée) avant d’écrire le message avec une autre méthode.

function send($room\_id, $message)

{

$session = SessionManager::getSessionInfo();

//check session

if ($session["isLogged"] == true) {

// user dans la session

$user = $session["username"];

http\_response\_code(200);

return $this->writeMessage($room\_id, $user, $message);

} else {

//pas logué

http\_response\_code(403);

}

}

La méthode « write » va sauvegarder le message dans la DB en évitant les injections SQL.

private function writeMessage($room\_id, $user, $message)

{

$connection = WrkDb::getInstance();

//définit les champs manquant

$fk\_user = $connection->executeQuery("SELECT \* FROM t\_user WHERE username = ?", array($user))->fetch(PDO::FETCH\_ASSOC)["pk\_user"];

$dateEnvoi = date('Y-m-d H:i:s'); // Format: YYYY-MM-DD HH:MM:SS

// Preparation du SQL

$sql = "INSERT INTO t\_message (texte, dateEnvoi, fk\_user, fk\_room) VALUES (:texte, :dateEnvoi, :fk\_user, :fk\_room)";

$params = array(':texte' => $message, ':dateEnvoi' => $dateEnvoi, ':fk\_user' => $fk\_user, ':fk\_room' => $room\_id);

$query = $connection->executeQuery($sql, $params);

return $query->rowCount();

}

SessionManager.php

Le gestionnaire de message appelle le gestionnaire de session pour savoir si l’utilisateur a le droit d’envoyer le message. Cette méthode retourne simplement les informations stockées dans la session.

static function getSessionInfo()

{

if (isset($\_SESSION["isLogged"])) {

return array(

'username' => $\_SESSION['username'],

'isLogged' => $\_SESSION['isLogged'],

'admin' => $\_SESSION['isAdmin']

);

} else {

return array('username' => "", 'isLogged' => "", 'admin' => "");

}

}

WrkDb.php

La méthode executeQuery permet, comme son nom l’indique, d’exécuter une requête. Elle utilise les prepared statements pour contrer les injections SQL.

public function executeQuery($query, $params = array())

{

try {

$stmt = $this->pdo->prepare($query);

$stmt->execute($params);

return $stmt;

} catch (PDOException $e) {

http\_response\_code(500);

error\_log("Erreur lors de l'exécution de la requête: " . $e->getMessage());

die("Erreur lors de l'exécution de la requête: " . $e->getMessage());

}

}

## Tests fonctionnels

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Utilisateur | Cas | Tests | Attendu | Obtenu | OK/NOK |
| Visiteur | Envoyer un message | Un visiteur non connecté essaye d'envoyer un message | Un message s'affiche lui disant qu'il doit se connecter pour envoyer un message. | Un message apparait disant qu’il faut aller se loguer. | OK |
| Client | Envoyer un message | Un visiteur envoie un message en étant connecté | L'envoi du message est possible et il s’affiche à la suite des autres messages | Les Messages sont rafraichis et le nouveau est affiché dans la liste | OK |
| Envoyer un message | Un visiteur envoie un message contenant une injection SQL | L'injection SQL n'est pas interprétée / un message d'erreur s'affiche | L’injection n’est pas exécutée mais est affiché comme n’importe quel autre message | OK |
| Envoyer un message | Un visiteur envoie un message vide | Le message n'est pas envoyé, il ne se passe rien (un message d'erreur s’affiche ?) | Une erreur (dans la console, rien n’est visible sur le site) nous dit que le texte est vide | OK |
| Administrateur | Envoi/Gestion d'un message | Un administrateur va sur la page de chat où figurent les messages | Il a la possibilité en plus d'envoyer lui aussi des messages de supprimer les messages qui figurent déjà dans le chat | Le bouton « passer en mode admin est en plus par rapport à un utilisateur | OK |
| Envoyer un message | Un administrateur envoie un message contenant une injection SQL | L'injection SQL n'est pas interprétée / un message d'erreur s'affiche | L’injection n’est pas exécutée mais est affiché comme n’importe quel autre message | OK |
| Envoyer un message | Un administrateur envoie un message vide | Le message n'est pas envoyé, il ne se passe rien (un message d'erreur s’affiche ?) | Les messages des admins sont traités de la même manière que les utilisateurs, les messages vides ne sont pas envoyés | OK |

## Problèmes rencontrés

## Hébergement

# Synthèse

## Différence de timing

## Différence avec la conception

## Conclusion

### Ce que je retiens de ce module

### Amélioration / proposition

### Mes points forts et faibles

# Glossaire et Bibliographie

## Bouts de code

## documentation