Documentation Technique - School Agent

Guide complet expliquant l'architecture, les choix techniques et le fonctionnement du code.

📚 Table des matières

🏛️ Architecture globale

Pourquoi cette architecture ?

Le projet utilise une architecture **MVC (Model-View-Controller)** personnalisée en PHP natif. Ce choix a été fait pour :

- **Apprentissage** : Comprendre les fondamentaux avant d'utiliser un framework
- **Légèreté** : Pas de dépendances lourdes pour un MVP
- **Contrôle** : Maîtrise totale du code et du fonctionnement
- **Performance** : Code optimisé sans overhead de framework

Structure du projet

```
School_Agent/
├── app/
│   ├── Config/              # Configuration (BDD, Auth)
│   ├── Controllers/         # Logique métier
│   ├── Models/             # Accès aux données
│   └── Views/              # Templates HTML/PHP
├── public/
│   ├── index.php           # Point d'entrée unique
│   ├── css/                # Styles
│   └── js/                 # Scripts
├── documents/              # Documentation SQL
└── vendor/                 # Dépendances Composer
```

Principe du point d'entrée unique : - Toutes les requêtes passent par `public/index.php` - Le routing analyse l'URL et appelle le bon contrôleur - Cela permet de centraliser la sécurité et la configuration

---

🎯 Structure MVC

Model (Modèle)

Rôle : Gérer les données et les interactions avec la base de données

Exemple avec `UserModel.php` :

```php
namespace SchoolAgent\Models;

use SchoolAgent\Config\Database;
use PDO;

class UserModel
{
    private $db;

    public function __construct()
    {
        $this->db = Database::getInstance()->getConnection();
    }

    public function getUserByEmail($email)
    {
        $stmt = $this->db->prepare("SELECT * FROM user WHERE email = :email");
        $stmt->execute(['email' => $email]);
        return $stmt->fetch(PDO::FETCH_ASSOC);
    }
}
```

Pourquoi cette approche ?

1. **Séparation des responsabilités** : Le modèle ne fait QUE de l'accès aux données
2. **Réutilisabilité** : Les méthodes peuvent être appelées depuis n'importe quel contrôleur
3. **Sécurité** : Requêtes préparées (PDO) contre les injections SQL
4. **Singleton Database** : Une seule connexion BDD partagée = performance optimale

Requêtes préparées - Pourquoi ?

```php
// ❌ DANGEREUX (injection SQL possible)
$query = "SELECT * FROM user WHERE email = '$email'";

// ✅ SÉCURISÉ (requête préparée)
$stmt = $this->db->prepare("SELECT * FROM user WHERE email = :email");
$stmt->execute(['email' => $email]);
```

Si un utilisateur entre `admin@test.com' OR '1'='1`, la première méthode retournerait tous les utilisateurs. La seconde traite l'input comme une simple chaîne.

View (Vue)

Rôle : Afficher les données à l'utilisateur (HTML/CSS/JS)

Exemple avec `login.php` :

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <title>Connexion - School Agent</title>
    <link rel="stylesheet" href="/css/front/login.css">
</head>
<body>
    <div class="login-container">
        <form method="POST" action="/login">
            <input type="email" name="email" required>
            <input type="password" name="mot_de_passe" required>
            <button type="submit">Se connecter</button>
        </form>
    </div>
    <script src="/js/front/login.js"></script>
</body>
</html>
```

Pourquoi séparer les vues ?

1. **Designer vs Développeur** : Le designer peut modifier le HTML/CSS sans toucher au PHP
2. **Réutilisation** : Templates header/footer inclus partout (`require header.php`)
3. **Maintenance** : Changement de design centralisé
4. **Lisibilité** : Pas de mélange logique/présentation

Templates partiels :

```
// En-tête réutilisable
<?php require __DIR__ . '/../templates/header.php'; ?>

<main>
    <!-- Contenu de la page -->
</main>

// Pied de page réutilisable
<?php require __DIR__ . '/../templates/footer.php'; ?>
```

Avantage : Modification du header une seule fois = changement sur toutes les pages.

Controller (Contrôleur)

Rôle : Coordonner Model et View, gérer la logique métier

Exemple avec `AuthController.php` :

```php
namespace SchoolAgent\Controllers;

use SchoolAgent\Models\UserModel;
use SchoolAgent\Config\Authenticator;

class AuthController
{
    private $model;

    public function __construct()
    {
        $this->model = new UserModel();
    }

    public function login()
    {
        // Vérifier si déjà connecté
        if (Authenticator::isLogged()) {
            // Rediriger selon le rôle
            header('Location: /home');
            exit;
        }

        // Traiter le formulaire POST
        if ($_SERVER['REQUEST_METHOD'] === 'POST') {
            $email = $_POST['email'];
            $password = $_POST['mot_de_passe'];

            // Récupérer l'utilisateur (Model)
            $user = $this->model->getUserByEmail($email);

            // Vérifier le mot de passe
            if ($user && password_verify($password, $user['mot_de_passe'])) {
                Authenticator::login($user['id_user']);
                header('Location: /home');
                exit;
            } else {
                $error = "Email ou mot de passe incorrect.";
            }
        }

        // Afficher la vue
        require __DIR__ . '/../Views/front/login.php';
    }
}
```

Pourquoi cette logique ?

1. **Sécurité d'abord** : Vérifier si l'utilisateur est déjà connecté avant tout
2. **Validation côté serveur** : Ne jamais faire confiance aux données POST
3. **Hashage des mots de passe** : `password_verify()` compare le hash en toute sécurité
4. **Redirection selon rôle** : Admin → dashboard, User → accueil
5. **Séparation Model/View** : Le contrôleur orchestre, ne fait pas le travail

🔐 Authentification et Sécurité

Classe `Authenticator.php`

Rôle : Gérer les sessions et l'authentification utilisateur

```php
namespace SchoolAgent\Config;

class Authenticator
{
    public static function login($userId)
    {
        session_start();
        $_SESSION['user_id'] = $userId;
        $_SESSION['logged_in'] = true;
    }

    public static function logout()
    {
        session_start();
        session_unset();
        session_destroy();
    }

    public static function isLogged()
    {
        session_start();
        return isset($_SESSION['logged_in']) && $_SESSION['logged_in'] === true;
    }

    public static function getUserId()
    {
        session_start();
        return $_SESSION['user_id'] ?? null;
    }
}
```

Pourquoi cette approche ?

1. **Méthodes statiques** : Pas besoin d'instancier, accessible partout (`Authenticator::isLogged()`)
2. **Session PHP native** : Stockage sécurisé côté serveur (pas de cookies exposés)
3. **Centralisation** : Une seule classe gère toute l'authentification
4. **Simplicité** : 4 méthodes claires (login, logout, isLogged, getUserId)

Comment ça fonctionne ?

```php
// 1. L'utilisateur se connecte
Authenticator::login($user['id_user']);
// → Stocke l'ID en session

// 2. Sur chaque page protégée
```

```php
if (!Authenticator::isLogged()) {
    header('Location: /login');
    exit;
}
// → Vérifie la session

// 3. Récupérer l'utilisateur actuel
$userId = Authenticator::getUserId();
$user = $userModel->getUser($userId);
// → Charge les données de l'utilisateur

// 4. Déconnexion
Authenticator::logout();
// → Détruit la session
```

Protection des routes

Middleware dans les contrôleurs :

```php
public function index()
{
    // Vérifier l'authentification
    if (!Authenticator::isLogged()) {
        header('Location: /login');
        exit;
    }

    // Vérifier le rôle (pour pages admin)
    $user = $this->model->getUser(Authenticator::getUserId());
    if ($user['role'] !== 'admin') {
        header('Location: /home');
        exit;
    }

    // Suite du code...
}
```

Pourquoi cette protection ?

- Empêche l'accès direct aux pages par URL
- Vérifie à chaque requête (pas de cache côté client)
- Peut être amélioré avec un middleware centralisé (en Symfony par exemple)

Hashage des mots de passe

Lors de l'inscription :

```php
$hashedPassword = password_hash($password, PASSWORD_DEFAULT);
// Stocke le hash en BDD, JAMAIS le mot de passe en clair
```

Lors de la connexion :

```php
if (password_verify($password, $user['mot_de_passe'])) {
    // Mot de passe correct
}
```

Pourquoi `password_hash()` ?

- **Sécurité** : Hash bcrypt avec salt automatique
- **Évolutif** : `PASSWORD_DEFAULT` s'adapte aux nouveaux algorithmes
- **Standard PHP** : Fonction native, pas besoin de bibliothèque externe
- **Protection** : Même si la BDD est compromise, les mots de passe restent sécurisés

---

🎮 Contrôleurs

Structure d'un contrôleur

```php
namespace SchoolAgent\Controllers\Front;

use SchoolAgent\Models\AgentModel;
use SchoolAgent\Models\ConversationModel;
use SchoolAgent\Models\MessageModel;
use SchoolAgent\Config\Authenticator;

class IaController
{
    private $agentModel;
    private $conversationModel;
    private $messageModel;

    public function __construct()
    {
        $this->agentModel = new AgentModel();
        $this->conversationModel = new ConversationModel();
        $this->messageModel = new MessageModel();
    }

    public function index()
    {
        // Protection
        if (!Authenticator::isLogged()) {
            header('Location: /login');
            exit;
        }

        // Récupérer les données
        $agents = $this->agentModel->getAllAgents();
        $userId = Authenticator::getUserId();
        $user = $this->userModel->getUser($userId);

        // Variables pour la vue
        $isLogged = true;
```

```
        // Afficher la vue
        require __DIR__ . '/../../Views/front/ia/ia.php';
    }
}
```

Principes appliqués :

1. **Injection de dépendances** : Les modèles sont instanciés dans le constructeur
2. **Protection systématique** : Vérification auth sur chaque action
3. **Préparation des données** : Le contrôleur récupère tout ce dont la vue a besoin
4. **Pas de logique dans la vue** : Tout est calculé avant le `require`

Contrôleur `AuthController.php` - Analyse détaillée

```
public function login()
{
    // 1. Vérifier si déjà connecté
    if (Authenticator::isLogged()) {
        $user = $this->model->getUser(Authenticator::getUserId());
        if ($user && $user['role'] === 'admin') {
            header('Location: /admin');
        } else {
            header('Location: /home');
        }
        exit;
    }

    // 2. Traiter la soumission du formulaire
    if ($_SERVER['REQUEST_METHOD'] === 'POST') {
        $email = $_POST['email'];
        $password = $_POST['mot_de_passe'];

        // 3. Valider les données (simple ici, à améliorer)
        $user = $this->model->getUserByEmail($email);

        // 4. Vérifier le mot de passe
        if ($user && password_verify($password, $user['mot_de_passe'])) {
            // 5. Connexion réussie
            Authenticator::login($user['id_user']);

            // 6. Rediriger selon le rôle
            if ($user['role'] === 'admin') {
                header('Location: /admin');
            } else {
                header('Location: /home');
            }
            exit;
        } else {
            // 7. Erreur de connexion
            $error = "Email ou mot de passe incorrect.";
        }
    }
```

```
    // 8. Afficher le formulaire (GET ou POST avec erreur)
    require __DIR__ . '/../Views/front/login.php';
}
```

Flux de données :

1. **GET /login** → Affiche le formulaire
2. **POST /login** → Traite les données
3. **Si succès** → Crée la session + redirection
4. **Si échec** → Affiche erreur + reformulaire

Améliorations possibles :

- Validation plus stricte (regex email, longueur mot de passe)
- Limite de tentatives de connexion (protection brute force)
- Logs des tentatives de connexion
- CSRF token (protection contre les attaques CSRF)

💾 Modèles

Classe `Database.php` - Pattern Singleton

```php
namespace SchoolAgent\Config;

use PDO;
use PDOException;

class Database
{
    private static $instance = null;
    private $connection;

    private function __construct()
    {
        require_once __DIR__ . '/database.config.php';

        try {
            $this->connection = new PDO(
                "mysql:host=" . DB_HOST . ";port=" . DB_PORT . ";dbname=" . DB_NAME,
                DB_USER,
                DB_PASS,
                [
                    PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
                    PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
                    PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES utf8"
                ]
            );
        } catch (PDOException $e) {
            die("Erreur de connexion : " . $e->getMessage());
        }
```

```
    }

    public static function getInstance()
    {
        if (self::$instance === null) {
            self::$instance = new Database();
        }
        return self::$instance;
    }

    public function getConnection()
    {
        return $this->connection;
    }
}
```

Pourquoi le pattern Singleton ?

1. **Une seule connexion** : Évite de créer 50 connexions BDD pour 50 requêtes
2. **Performance** : Réutilisation de la connexion existante
3. **Économie ressources** : Moins de charge sur le serveur MySQL
4. **Contrôle** : Point d'entrée unique pour la BDD

Comment ça marche ?

```
// Première utilisation
$db1 = Database::getInstance(); // Crée la connexion
$db2 = Database::getInstance(); // Réutilise la même connexion
// $db1 === $db2 (même objet)
```

Options PDO expliquées :

- `PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION` : Lance des exceptions en cas d'erreur (mieux pour le debug)
- `PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC` : Retourne des tableaux associatifs (pas d'index numériques)
- `SET NAMES utf8` : Support des caractères spéciaux (accents, emojis)

Modèle `UserModel.php` - Gestion des utilisateurs

```
namespace SchoolAgent\Models;

use SchoolAgent\Config\Database;
use PDO;

class UserModel
{
    private $db;

    public function __construct()
    {
        $this->db = Database::getInstance()->getConnection();
    }
```

```php
    public function getAllUsers()
    {
        $stmt = $this->db->query("SELECT * FROM user ORDER BY nom, prenom");
        return $stmt->fetchAll();
    }

    public function getUser($id)
    {
        $stmt = $this->db->prepare("SELECT * FROM user WHERE id_user = :id");
        $stmt->execute(['id' => $id]);
        return $stmt->fetch();
    }

    public function getUserByEmail($email)
    {
        $stmt = $this->db->prepare("SELECT * FROM user WHERE email = :email");
        $stmt->execute(['email' => $email]);
        return $stmt->fetch();
    }

    public function createUser($data)
    {
        $stmt = $this->db->prepare("
            INSERT INTO user (nom, prenom, email, mot_de_passe, role)
            VALUES (:nom, :prenom, :email, :password, :role)
        ");
        return $stmt->execute([
            'nom' => $data['nom'],
            'prenom' => $data['prenom'],
            'email' => $data['email'],
            'password' => password_hash($data['password'], PASSWORD_DEFAULT),
            'role' => $data['role'] ?? 'etudiant'
        ]);
    }

    public function updateUser($id, $data)
    {
        $stmt = $this->db->prepare("
            UPDATE user
            SET nom = :nom, prenom = :prenom, email = :email, role = :role
            WHERE id_user = :id
        ");
        return $stmt->execute([
            'id' => $id,
            'nom' => $data['nom'],
            'prenom' => $data['prenom'],
            'email' => $data['email'],
            'role' => $data['role']
        ]);
    }

    public function deleteUser($id)
    {
```

```
        $stmt = $this->db->prepare("DELETE FROM user WHERE id_user = :id");
        return $stmt->execute(['id' => $id]);
    }
}
```

Architecture CRUD :

- **C**reate : `createUser()`
- **R**ead : `getAllUsers()`, `getUser()`, `getUserByEmail()`
- **U**pdate : `updateUser()`
- **D**elete : `deleteUser()`

Pourquoi plusieurs méthodes de lecture ?

- `getAllUsers()` : Liste d'administration
- `getUser($id)` : Profil utilisateur
- `getUserByEmail($email)` : Authentification

Chaque méthode a un usage spécifique, cela rend le code plus lisible.

🎨 Vues et Templates

Template de base

Structure avec header/footer réutilisables :

```
<!-- header.php -->
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>School Agent</title>
    <link rel="stylesheet" href="/css/front/home.css">
</head>
<body>
    <header class="main-header">
        <nav>
            <a href="/home">Accueil</a>
            <a href="/ia">Assistants IA</a>
            <?php if ($isLogged): ?>
                <span>Bonjour <?= htmlspecialchars($user['prenom']) ?></span>
                <a href="/logout">Déconnexion</a>
            <?php else: ?>
                <a href="/login">Connexion</a>
            <?php endif; ?>
        </nav>
    </header>

<!-- footer.php -->
    <footer>
        <p>© 2025 School Agent. Olivier / Nicolas / Flavie</p>
```

```
    </footer>
</body>
</html>
```

Utilisation dans une page :

```php
<?php
// Variables nécessaires pour le header
$isLogged = Authenticator::isLogged();
$user = $isLogged ? $userModel->getUser(Authenticator::getUserId()) : null;
?>

<!-- Inclure le header -->
<?php require __DIR__ . '/../templates/header.php'; ?>

<main class="content">
    <h1>Ma page</h1>
    <!-- Contenu -->
</main>

<!-- Inclure le footer -->
<?php require __DIR__ . '/../templates/footer.php'; ?>
```

Avantages :

- Modification du design en un seul endroit
- Cohérence visuelle sur tout le site
- Navigation toujours identique
- Maintenance facilitée

Protection XSS dans les vues

Toujours échapper les données utilisateur :

```php
<!-- ❌ DANGEREUX -->
<p>Bonjour <?= $user['prenom'] ?></p>

<!-- ✅ SÉCURISÉ -->
<p>Bonjour <?= htmlspecialchars($user['prenom']) ?></p>
```

Pourquoi `htmlspecialchars()` ?

Si un utilisateur entre `<script>alert('hack')</script>` comme prénom : - Sans protection : Le script s'exécute - Avec protection : Affiché comme texte `&lt;script&gt;...`

Raccourci utile :

```php
<?php
function h($text) {
    return htmlspecialchars($text, ENT_QUOTES, 'UTF-8');
}
?>
```

```
<!-- Utilisation -->
<p>Bonjour <?= h($user['prenom']) ?></p>
```

⚙️ Configuration

Fichier `database.config.php`

```php
<?php
define('DB_HOST', 'localhost');
define('DB_PORT', '3308');
define('DB_NAME', 'schoolia');
define('DB_USER', 'root');
define('DB_PASS', '');
```

Pourquoi des constantes ?

- Accessible partout avec `DB_HOST`
- Ne peut pas être modifié accidentellement
- Conventions PHP pour la configuration

Sécurité :

- Fichier dans `.gitignore` (ne pas versionner)
- Permissions restrictives sur le serveur (chmod 600)
- Pas de credentials en dur dans le code

Fichier `config.php` - Configuration globale

```php
<?php
return [
    'database' => [
        'host' => 'localhost',
        'port' => '3308',
        'dbname' => 'schoolia',
        'user' => 'root',
        'password' => ''
    ],
    'app' => [
        'name' => 'School Agent',
        'url' => 'http://localhost',
        'environment' => 'development'
    ],
    'ai' => [
        'api_key' => 'VOTRE_CLE_API_GROQ_ICI',
        'api_url' => 'https://api.groq.com/openai/v1/chat/completions',
        'model' => 'llama-3.3-70b-versatile',
        'temperature' => 1.0
    ],
    'session' => [
```

```
        'lifetime' => 3600,
        'cookie_secure' => false,
        'cookie_httponly' => true
    ]
];
```

Pourquoi un tableau de configuration ?

- Organisation par catégories
- Facile à charger : `$config = require 'config.php';`
- Peut être converti en objet ou classe Config
- Préparation pour Symfony (.env)

🎨 Frontend (CSS/JS)

Organisation CSS

```
public/css/
├── front/
│   ├── home.css          # Page d'accueil
│   ├── login.css         # Page connexion
│   ├── ia.css            # Liste des agents
│   ├── conversations.css # Liste conversations
│   └── chat.css          # Interface chat
```

Pourquoi séparer les CSS ?

- Performance : Charger uniquement ce dont on a besoin
- Maintenance : Modifications ciblées
- Organisation : Facile de retrouver le style d'une page

Design System utilisé

Couleurs :

```
:root {
    --color-primary: #667eea;      /* Violet principal */
    --color-primary-dark: #764ba2; /* Violet foncé */
    --color-bg: #f8f9fa;           /* Fond clair */
    --color-bg-dark: #e9ecef;      /* Fond plus foncé */
    --color-white: #ffffff;        /* Blanc */
    --color-shadow: rgba(0,0,0,0.06); /* Ombre légère */
}
```

Gradients :

```
/* Gradient violet pour boutons et cartes */
background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);

/* Gradient fond de page */
background: linear-gradient(to bottom, #f8f9fa, #e9ecef);
```

Pourquoi un design system ?

- Cohérence visuelle sur tout le site
- Facile de changer une couleur partout
- Variables CSS réutilisables
- Maintenance simplifiée

JavaScript - Chat

Fichier `chat.js` :

```
document.addEventListener('DOMContentLoaded', function() {
    const messagesContainer = document.querySelector('.messages-container');
    const chatInput = document.querySelector('.chat-input');
    const sendButton = document.querySelector('.send-btn');

    // Auto-scroll vers le bas
    function scrollToBottom() {
        if (messagesContainer) {
            messagesContainer.scrollTop = messagesContainer.scrollHeight;
        }
    }

    // Auto-resize du textarea
    function autoResizeTextarea() {
        chatInput.style.height = 'auto';
        chatInput.style.height = Math.min(chatInput.scrollHeight, 150) + 'px';
    }

    // Événements
    chatInput.addEventListener('input', autoResizeTextarea);

    chatInput.addEventListener('keydown', function(e) {
        if (e.key === 'Enter' && e.ctrlKey) {
            sendButton.click();
        }
    });

    // Scroll initial
    scrollToBottom();
});
```

Fonctionnalités implémentées :

1. **Auto-scroll** : Toujours voir le dernier message
2. **Auto-resize textarea** : S'adapte au contenu (max 150px)
3. **Ctrl+Enter** : Envoyer le message rapidement
4. **DOMContentLoaded** : Attendre le chargement complet de la page

Pourquoi du JavaScript natif ?

- Pas besoin de jQuery pour des fonctionnalités simples
- Performance optimale

- Moins de dépendances
- Apprentissage des bases

🤖 Intégration API Groq

Appel API dans le contrôleur

```php
public function sendMessage($conversationId)
{
    if ($_SERVER['REQUEST_METHOD'] === 'POST') {
        $userMessage = $_POST['message'] ?? '';

        // 1. Sauvegarder le message de l'utilisateur
        $this->messageModel->create([
            'id_conversation' => $conversationId,
            'role' => 'user',
            'contenu' => $userMessage,
            'timestamp' => date('Y-m-d H:i:s')
        ]);

        // 2. Récupérer le prompt système de l'agent
        $conversation = $this->conversationModel->getConversation($conversationId);
        $agent = $this->agentModel->getAgent($conversation['id_agent']);
        $systemPrompt = $agent['prompt_system'];

        // 3. Appeler l'API Groq
        $apiResponse = $this->callGroqApi($systemPrompt, $userMessage);

        if ($apiResponse['success']) {
            // 4. Sauvegarder la réponse de l'IA
            $this->messageModel->create([
                'id_conversation' => $conversationId,
                'role' => 'assistant',
                'contenu' => $apiResponse['content'],
                'timestamp' => date('Y-m-d H:i:s')
            ]);
        }

        // 5. Rediriger vers le chat
        header("Location: /ia/chat?id=$conversationId");
        exit;
    }
}

private function callGroqApi($systemPrompt, $userMessage)
{
    $config = require __DIR__ . '/../../Config/config.php';

    $data = [
        'model' => $config['ai']['model'],
        'messages' => [
```

```
            ['role' => 'system', 'content' => $systemPrompt],
            ['role' => 'user', 'content' => $userMessage]
        ],
        'temperature' => $config['ai']['temperature'],
        'max_tokens' => 1024
    ];

    $ch = curl_init($config['ai']['api_url']);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    curl_setopt($ch, CURLOPT_POST, true);
    curl_setopt($ch, CURLOPT_POSTFIELDS, json_encode($data));
    curl_setopt($ch, CURLOPT_HTTPHEADER, [
        'Content-Type: application/json',
        'Authorization: Bearer ' . $config['ai']['api_key']
    ]);

    $response = curl_exec($ch);
    $httpCode = curl_getinfo($ch, CURLINFO_HTTP_CODE);
    curl_close($ch);

    if ($httpCode === 200) {
        $result = json_decode($response, true);
        return [
            'success' => true,
            'content' => $result['choices'][0]['message']['content'] ?? ''
        ];
    } else {
        return [
            'success' => false,
            'error' => 'Erreur API'
        ];
    }
}
```

Flux de données :

1. Utilisateur envoie un message (POST)
2. Message sauvegardé en BDD (rôle ‘user’)
3. Récupération du prompt système de l’agent
4. Appel API Groq avec cURL
5. Réponse sauvegardée en BDD (rôle ‘assistant’)
6. Redirection vers le chat actualisé

Pourquoi sauvegarder les messages ?

- Historique de conversation
- Possibilité de reprendre une conversation
- Analyse des interactions
- Backup en cas d’erreur API

Structure de l’appel API :

```
{
  "model": "llama-3.3-70b-versatile",
  "messages": [
    {
      "role": "system",
      "content": "Tu es un professeur de mathématiques..."
    },
    {
      "role": "user",
      "content": "Comment résoudre une équation du second degré ?"
    }
  ],
  "temperature": 1.0,
  "max_tokens": 1024
}
```

- **system** : Contexte et personnalité de l'IA
- **user** : Question de l'utilisateur
- **temperature** : Créativité (0 = précis, 2 = créatif)
- **max_tokens** : Longueur maximale de la réponse

🗺️ Routing

Fichier `public/index.php` - Point d'entrée

```php
<?php
require_once __DIR__ . '/../vendor/autoload.php';

use SchoolAgent\Controllers\AuthController;
use SchoolAgent\Controllers\Front\HomeController;
use SchoolAgent\Controllers\Front\IaController;
use SchoolAgent\Controllers\Admin\AdminController;

// Démarrer la session
session_start();

// Récupérer l'URI
$requestUri = $_SERVER['REQUEST_URI'];
$requestUri = strtok($requestUri, '?'); // Enlever les paramètres GET

// Routing
switch ($requestUri) {
    case '/':
    case '/home':
        $controller = new HomeController();
        $controller->index();
        break;

    case '/login':
        $controller = new AuthController();
```

```
        $controller->login();
        break;

    case '/logout':
        $controller = new AuthController();
        $controller->logout();
        break;

    case '/ia':
        $controller = new IaController();
        $controller->index();
        break;

    case '/ia/conversations':
        $controller = new IaController();
        $controller->showConversations();
        break;

    case '/ia/chat':
        $controller = new IaController();
        $controller->showChat();
        break;

    case '/admin':
        $controller = new AdminController();
        $controller->index();
        break;

    default:
        http_response_code(404);
        echo "Page non trouvée";
        break;
}
```

Comment ça marche ?

1. **Autoload Composer** : Charge automatiquement les classes
2. **Session** : Démarrer avant tout routing
3. **URI parsing** : Récupérer l'URL demandée
4. **Switch/Case** : Router vers le bon contrôleur
5. **404** : Gérer les URLs inconnues

Pourquoi ce système ?

- Simple et compréhensible pour apprendre
- Pas de dépendances externes
- Facile à débuguer
- Peut être amélioré progressivement

Améliorations possibles :

```
// Routing plus avancé avec regex
$routes = [
    '#^/ia/chat/(\d+)$#' => ['IaController', 'showChat'],
```

```
    '#^/admin/users/(\d+)/edit$#' => ['AdminUserController', 'edit']
];

foreach ($routes as $pattern => $action) {
    if (preg_match($pattern, $requestUri, $matches)) {
        [$controller, $method] = $action;
        $ctrl = new $controller();
        $ctrl->$method($matches[1]);
        break;
    }
}
```

🔄 Flux complet d'une requête

Exemple : Envoi d'un message dans le chat

1. L'utilisateur tape un message et clique sur "Envoyer"

```
<form method="POST" action="/ia/send-message?id=<?= $conversationId ?>">
    <textarea name="message"></textarea>
    <button type="submit">Envoyer</button>
</form>
```

2. Le navigateur envoie une requête POST à `/ia/send-message?id=123`

3. `index.php` route vers `IaController::sendMessage()`

```
case '/ia/send-message':
    $controller = new IaController();
    $controller->sendMessage($_GET['id']);
    break;
```

4. Le contrôleur traite la requête

```
public function sendMessage($conversationId)
{
    // Vérifier l'authentification
    if (!Authenticator::isLogged()) {
        header('Location: /login');
        exit;
    }

    // Récupérer le message
    $userMessage = $_POST['message'];

    // Sauvegarder en BDD via le modèle
    $this->messageModel->create([...]);

    // Appeler l'API
    $response = $this->callGroqApi(...);
```

```
    // Sauvegarder la réponse
    $this->messageModel->create([...]);

    // Rediriger
    header("Location: /ia/chat?id=$conversationId");
}
```

5. Redirection vers le chat avec les nouveaux messages

6. Affichage de la vue avec les données actualisées

```
<?php foreach ($messages as $message): ?>
    <div class="message <?= $message['role'] ?>">
        <p><?= htmlspecialchars($message['contenu']) ?></p>
        <span><?= date('H:i', strtotime($message['timestamp'])) ?></span>
    </div>
<?php endforeach; ?>
```

📊 Base de données - Structure

Tables principales

Table `user` :

```
CREATE TABLE user (
    id_user INT PRIMARY KEY AUTO_INCREMENT,
    nom VARCHAR(100) NOT NULL,
    prenom VARCHAR(100),
    email VARCHAR(180) UNIQUE NOT NULL,
    mot_de_passe VARCHAR(255) NOT NULL,
    role VARCHAR(50) DEFAULT 'etudiant',
    niveau_education VARCHAR(100)
);
```

Table `agent` :

```
CREATE TABLE agent (
    id_agent INT PRIMARY KEY AUTO_INCREMENT,
    nom VARCHAR(100) NOT NULL,
    type VARCHAR(100) NOT NULL,
    description TEXT NOT NULL,
    specialite VARCHAR(100) NOT NULL,
    status VARCHAR(50) DEFAULT 'active',
    prompt_system TEXT NOT NULL
);
```

Table `conversation` :

```
CREATE TABLE conversation (
    id_conversation INT PRIMARY KEY AUTO_INCREMENT,
```

```
    id_user INT NOT NULL,
    id_agent INT NOT NULL,
    date_creation DATETIME NOT NULL,
    statut VARCHAR(50) DEFAULT 'active',
    FOREIGN KEY (id_user) REFERENCES user(id_user),
    FOREIGN KEY (id_agent) REFERENCES agent(id_agent)
);
```

Table `message` :

```
CREATE TABLE message (
    id_message INT PRIMARY KEY AUTO_INCREMENT,
    id_conversation INT NOT NULL,
    role VARCHAR(50) NOT NULL,
    contenu TEXT NOT NULL,
    timestamp DATETIME NOT NULL,
    FOREIGN KEY (id_conversation) REFERENCES conversation(id_conversation)
);
```

Relations

```
user (1) ----< (N) conversation (N) >---- (1) agent
                        |
                        | (1)
                        |
                        | (N)
                    message
```

Pourquoi ces relations ?

- Un utilisateur peut avoir plusieurs conversations
- Un agent peut être utilisé dans plusieurs conversations
- Une conversation contient plusieurs messages
- On garde l'historique complet

🚀 Déploiement et Production

Checklist avant mise en production

Sécurité : - [ ] Mots de passe hashés (password_hash) - [ ] Requêtes préparées partout (PDO) - [ ] htmlspecialchars sur toutes les sorties - [ ] HTTPS activé (certificat SSL) - [ ] Clés API dans variables d'environnement - [ ] .gitignore pour config.php

Performance : - [ ] CSS/JS minifiés - [ ] Images optimisées - [ ] Cache navigateur (headers) - [ ] Compression gzip activée - [ ] Index BDD sur colonnes fréquentes

Monitoring : - [ ] Logs d'erreurs activés - [ ] Monitoring serveur (CPU, RAM) - [ ] Alertes en cas d'erreur critique - [ ] Backup automatique BDD

🎓 Pourquoi migrer vers Symfony ?

Après avoir compris le code actuel, migrer vers Symfony apporte :

Avantages de Symfony

1. **Sécurité** :
   - Protection CSRF intégrée
   - Gestion avancée des permissions (Voters)
   - Encodage automatique des mots de passe
   - Protection contre injections (Doctrine)
2. **Performance** :
   - Cache optimisé
   - Lazy loading des services
   - HTTP cache intégré
   - Profiler pour détecter les lenteurs
3. **Productivité** :
   - Générateurs de code (make:controller, make:entity)
   - Débogage avancé (Profiler, VarDumper)
   - Tests unitaires et fonctionnels
   - Documentation complète
4. **Maintenabilité** :
   - Structure standardisée
   - Injection de dépendances
   - Bundle system (fonctionnalités réutilisables)
   - Communauté active
5. **Évolutivité** :
   - Microservices possible
   - API Platform pour REST API
   - Messenger pour tâches asynchrones
   - Support entreprise (SensioLabs)

Exemple de comparaison

Avant (PHP natif) :

```php
public function sendMessage($conversationId)
{
    if (!Authenticator::isLogged()) {
        header('Location: /login');
        exit;
    }

    $message = $_POST['message'];
    // ... logique métier
}
```

Après (Symfony) :

```php
#[Route('/conversations/{id}/send', methods: ['POST'])]
#[IsGranted('ROLE_USER')]
public function sendMessage(
    Request $request,
    Conversation $conversation,
    MessageRepository $messageRepo,
    GrokApiService $grokApi
): Response {
    $form = $this->createForm(MessageType::class);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $message = $form->getData();
        // ... logique métier
    }

    return $this->redirectToRoute('conversation_chat', ['id' => $conversation->getId()]);
}
```

Avantages : - Routing par annotations - Authentification par attribut `#[IsGranted]` - Validation automatique des formulaires - Injection de dépendances dans les paramètres - Type-hinting pour la sécurité

📚 Ressources pour aller plus loin

Documentation

- **PHP officiel** : https://www.php.net/manual/fr/
- **PDO** : https://www.php.net/manual/fr/book.pdo.php
- **Sécurité PHP** : https://www.php.net/manual/fr/security.php
- **Composer** : https://getcomposer.org/doc/

Tutoriels

- **Grafikart** : https://grafikart.fr/formations/php
- **OpenClassrooms** : https://openclassrooms.com/fr/courses/918836-concevez-votre-site-web-avec-php-et-mysql
- **PHP The Right Way** : https://phptherightway.com/

Migration Symfony

- Guide complet dans `README_Symfony.md`
- Documentation Symfony : https://symfony.com/doc/current/index.html
- Symfony Casts : https://symfonycasts.com/

✅ Points clés à retenir

Architecture MVC

✅ **Séparation des responsabilités** : Model (données), View (affichage), Controller (logique) ✅ **Modèles** : Une classe par table, méthodes CRUD ✅ **Vues** : Templates réutilisables, échappement XSS ✅ **Contrôleurs** : Orchestration, pas de logique métier

Sécurité

✅ **Authentification** : Sessions PHP, pas de cookies exposés ✅ **Mots de passe** : `password_hash()` et `password_verify()` ✅ **Injections SQL** : Requêtes préparées PDO ✅ **XSS** : `htmlspecialchars()` sur toutes les sorties

Bonnes pratiques

✅ **Singleton BDD** : Une seule connexion réutilisée ✅ **Namespaces** : Organisation du code en packages ✅ **Autoload** : Composer gère les imports ✅ **Configuration** : Fichiers séparés, .gitignore

Évolution

✅ **Code actuel** : Bon pour apprendre les fondamentaux ✅ **Prochaine étape** : Migrer vers Symfony ✅ **Objectif** : Application professionnelle, scalable, sécurisée

---

Dernière mise à jour : 12 novembre 2025
Auteurs : Olivier / Nicolas / Flavie