ESGI 4 IW

Projet Roomies

Florian Lebon

21/07/2025

Introduction

Ce projet est un projet transversal dont le sujet était de développer une webapp permettant aux utilisateurs de jouer à un jeu et de discuter entre eux.

Il a pour but de nous évaluer sur les compétences suivantes :

- Vuejs avec typescript
- Symfony
- Test unitaires, fonctionnels et E2E
- Communication web temps réel (websocket, SSE)

Ce document reprend les différents points importants de ce projet pour les présenter plus en détail. Vous pouvez toujours retrouver le code complet sur github : https://github.com/Yoseinoo/ESGI4-ROOMIES.git

VueJS

Router Vue

L'utilisation du routeur Vue est effectué dans le fichier src/router/index.ts.

On y retrouve les différentes routes liées a leur composant à charger lorsque cette dernière est appelée.

On retrouve également un genre de middleware qui permet de vérifier avant chaque redirection si l'utilisateur a le droit de s'y rendre sans être authentifié. Si l'utilisateur doit être identifié alors il est redirigé vers la page de login.

```
const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  routes: [
    {
      path: '/',
      name: 'home',
      component: () => import('../views/HomeView.vue'),
    },
    {
      path: '/login',
      name: 'login',
      component: () => import('../views/LoginView.vue'),
    },
    {
      path: '/room/:id',
      name: 'room',
      component: () => import('../views/RoomView.vue'),
    },
    {
      path: '/profile',
      name: 'profile',
      component: () => import('../views/UserView.vue'),
    },
  ],
})
```

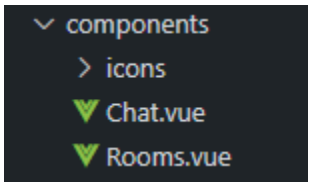
```
// Navigation Guard
router.beforeEach(async (to, _, next) => {
  const user = localStorage.getItem('user');
  console.log(user)

  // Skip guard for public routes
  const publicPages = ['login']
  const authRequired = !publicPages.includes(to.name as string);

  if (authRequired && !user) {
    next({ name: 'login' })
  } else {
    next()
  }
})
```

Composants

L'application dispose de 2 composants principaux et réutilisables



Un composant pour le chat qui peut être ajouté à n'importe quelle page et un composant pour afficher les rooms disponibles utilisable de la même manière.

Ces composants se gèrent tout seul et sont complètement réutilisables n'importe où dans l'application.

Authentification

Pour gérer l'authentification, un store est utilisé en plus de la page de login.

L'utilisateur connecté est stocké dans le local storage.

Le store contient les fonctions "login" et "fetchUser" qui permettent de se connecter et de récupérer les informations de l'utilisateur connecté.

La page de connexion contient juste un formulaire qui appelle la fonction "login" du store lorsqu'il est soumit et redirige vers la page d'accueil.

```
async login(email: string, password: string): Promise<void> {
  this.loading = true
  try {
      const res = await fetch('https://localhost:8000/login', {
          method: 'POST',
          headers: {
              'Content-Type': 'application/json',
          },
          body: JSON.stringify({ email, password })
      })

      if (!res.ok) throw new Error('Échec de la connexion')

      const data = await res.json();
      console.log('connected as : ', data);
      this.user = data;
      localStorage.setItem('user', data.email)
      this.error = null
  } catch (err: any) {
      this.error = err.message ?? 'Erreur inconnue'
  } finally {
      this.loading = false
  }
}
```

```
var Error: ErrorConstructor
new (message?: string) => Error
```

```
async fetchUser() {
  let email = localStorage.getItem('user');

  if (!email || this.user != null) {
    return;
  }

  const res = await fetch('https://localhost:8000/my-profile', {
      method: 'POST',
      headers: {
          'Content-Type': 'application/json',
      },
      body: JSON.stringify({ email })
  })

  if (!res.ok) throw new Error('Échec de la connexion')

  const data = await res.json();
  console.log('got user : ', data);
  this.user = data;
},
```

```vue
<template>
  <div class="login-page">
    <h2>Connexion</h2>
    <form @submit.prevent="handleLogin">
      <input v-model="email" type="email" placeholder="Email" required />
      <input v-model="password" type="password" placeholder="Mot de passe" required />
      <button type="submit">Se connecter</button>
    </form>

    <p v-if="store.error" class="error">{{ store.error }}</p>
  </div>
</template>

<script setup lang="ts">
import { ref } from 'vue'
import { useRouter } from 'vue-router'
import { useUserStore } from '@/stores/userStore'

const email = ref('')
const password = ref('')
const router = useRouter()
const store = useUserStore()

const handleLogin = async () => {
  await store.login(email.value, password.value)
  if (store.user) {
    router.push({ name: 'home' })
  }
}
</script>
```

Rooms et chat global

La gestion du chat global se passe dans le composant réutilisable ‘Chat’.

Le websocket utilisé est Mercure qui fait du SSE (Server-sent events) et qui est le système de websocket natif a symfony.

Dans le composant, on essaye donc de se connecter au websocket en utilisant l’interface EventSource. On va alors souscrire à un channel et attendre que ce dernier nous envoie des évènements pour mettre à jour le front.

On a ensuite un formulaire qui envoie une méthode post au back avec le message et le back se charge de push vers le websocket.

```
onMounted(() => {
  const url = new URL('http://localhost:3000/.well-known/mercure');
  url.searchParams.append('topic', 'chat/general');

  eventSource = new EventSource(url.toString());

  eventSource.onmessage = event => {
    const data = JSON.parse(event.data);
    messages.value.push(data);
  };

  eventSource.onerror = err => {
    console.error('Erreur Mercure:', err);
    eventSource.close();
  };
});

onBeforeUnmount(() => {
  if (eventSource) {
    eventSource.close();
  }
});
```

```
const sendMessage = async () => {
  if (!newMessage.value.trim()) return;

  try {
    await fetch(api_url + '/chat/send', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ message: newMessage.value }),
    });
    newMessage.value = '';
  } catch (error) {
    console.error('Erreur envoi message:', error);
  }
};
```

Pour la gestion des rooms cela se passe à 2 endroits.

Le premier est dans la page de profil de l'utilisateur dans laquelle il peut ajouter et supprimer des rooms.

```
//Créer une room et l'ajoute dans la liste
const createRoom = async () => {
    if (!newRoomName.value.trim()) return
    const res = await axios.post('https://localhost:8000/create-room', {
        name: newRoomName.value,
        email: user.value.email
    })

    rooms.value.push(res.data)
    newRoomName.value = ''
}

//Supprimer une room et la supprime de la liste
const deleteRoom = async (roomId) => {
    await axios.delete(`https://localhost:8000/rooms/${roomId}`, {
        data: { email: 'user@example.com' }
    })

    console.log('Room deleted: ', roomId);
    rooms.value = rooms.value.filter(room => room.id !== roomId)
}
```

Enfin, dans la page de la room elle même, on gère la connexion au websocket pour le jeu et la capacité de la room.

```
//Connexion à la room du jeu
const connectToMercure = () => {
  const url = new URL('http://localhost:3000/.well-known/mercure')
  url.searchParams.append('topic', `/rooms/${roomId}`)

  eventSource = new EventSource(url.toString());

  eventSource.onmessage = (event) => {
    const data: GameUpdate = JSON.parse(event.data)
    board.value = data.board
    currentTurn.value = data.currentTurn
    winner.value = data.winner
    console.log('new game state : ', data);
  }

  eventSource.onerror = () => {
    console.error('Mercure connection error')
  }
}
```

```
/**
 * Permet de signifier au back que la room a un utilisateur supplémentaire
 */
const joinRoom = async () => {
    const res = await axios.post(`https://localhost:8000/rooms/${roomId}/join`, {
        email: user
    });

    console.log(res.data);
}

/**
 * Permet de leave la room
 */
const leaveRoom = async () => {
    const res = await axios.post(`https://localhost:8000/rooms/${roomId}/leave`, {
        email: user
    });

    console.log(res.data);
}
```

```
async function play(row: number, col: number): Promise<void> {
  error.value = null

  if (winner.value) {
    error.value = 'Game is over.'
    return
  }

  if (currentTurn.value !== user) {
    error.value = 'Not your turn!'
    return
  }

  if (board.value[row][col] !== '') {
    error.value = 'Cell already taken.'
    return
  }

  try {
    const response = await fetch(`https://localhost:8000/rooms/${roomId}/play`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        row,
        col,
        user: user,
      }),
    })

    const data = await response.json()
```

Symfony

Entités et relations

La base de données pour cette application est constituée de 2 entités.

Une entité pour les utilisateurs et une entité pour les rooms. Les rooms comportent les champs pour la logique du jeu qui devraient être mis dans une autre table idéalement.

```php
#[ORM\Entity]
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
    #[ORM\Id, ORM\GeneratedValue, ORM\Column(type:"integer")]
    private ?int $id = null;

    #[ORM\Column(type:"string", length:180, unique:true)]
    private string $email;

    #[ORM\Column(type:"json")]
    private array $roles = [];

    #[ORM\Column(type:"string")]
    private string $password;

    #[ORM\OneToMany(mappedBy: 'owner', targetEntity: Room::class, orphanRemoval: true)]
    private Collection $rooms;
```

```php
#[ORM\Entity]
class Room
{
    #[ORM\Id, ORM\GeneratedValue, ORM\Column(type: 'integer')]
    private ?int $id = null;

    #[ORM\Column(type: 'string')]
    private string $name;

    #[ORM\Column(type: 'integer')]
    private int $capacity = 2;

    #[ORM\Column(type: 'integer')]
    private int $playerCount = 0;

    #[ORM\ManyToOne(targetEntity: User::class, inversedBy: 'rooms')]
    #[ORM\JoinColumn(nullable: false)]
    private ?User $owner = null;

    /**
     * GAME STATE
     */
    #[ORM\Column(type: 'json')]
    private array $board = []; // 6x7 grid, default empty

    #[ORM\Column(type: 'string', nullable: true)]
    private ?string $currentTurn = null;

    #[ORM\Column(type: 'string', nullable: true)]
    private ?string $winner = null;

    #[ORM\Column(type: 'json')]
    private array $players = []; // [player1_id, player2_id]
```

Routes et contrôleurs

L'application est constituée de 4 contrôleurs principaux. Les routes sont définies par annotation au-dessus de chaque méthode de contrôleur.

Un contrôleur pour la mécanique de jeu : il contient une méthode "play" qui sert à jouer un tour et notifier le websocket que le tour a été joué pour qu'il informe le client.

```php
/**
 * Permet de jouer un tour sur le jeu
 */
#[Route('/rooms/{id}/play', methods: ['POST'])]
public function play(Request $request, EntityManagerInterface $em, HubInterface $hub, string $id): JsonResponse
{
    $data = json_decode($request->getContent(), true);
    $row = $data['row'];
    $col = $data['col'];
    $playerId = $data['user'];

    $room = $em->getRepository(Room::class)->find($id);
    if (!$room) return new JsonResponse(['error' => 'Room not found'], 404);

    if ($room->getWinner() || $room->getCurrentTurn() !== $playerId) {
        return new JsonResponse(['error' => 'Not your turn or game over'], 403);
    }

    $board = $room->getBoard();

    // Check if cell is empty
    if ($board[$row][$col] !== '') {
        return new JsonResponse(['error' => 'Cell already occupied'], 400);
    }

    // Determine player's mark: X or O (assuming players[0] = X, players[1] = O)
    $players = $room->getPlayers();
    if (count($players) < 2) {
        return new JsonResponse(['error' => 'Not enough players'], 400);
    }
```

Un contrôleur pour la gestion des rooms : Il contient les méthodes "createRoom", "getRooms", "getRoomsByEmail", "deleteRoomByEmail", "joinRoom", "leaveRoom". Ce contrôleur est utilisé pour toutes les actions effectuées sur les rooms.

```php
    * Permet de mettre à jour la room et le game state quand un joueur rejoins
    */
#[Route('/rooms/{id}/join', name: 'join_room', methods: ['POST'])]
public function joinRoom(Request $request, Room $room, EntityManagerInterface $em, HubInterface $hub): JsonResponse {
    $data = json_decode($request->getContent(), true);
    $email = $data['email'] ?? null;

    if (!$email) {
        return $this->json(['error' => 'Missing email'], 400);
    }

    if ($room->getPlayerCount() >= $room->getCapacity()) {
        return $this->json(['error' => 'Room is full'], 400);
    }

    if (!in_array($email, $room->getPlayers()) && $room->getPlayerCount() < $room->getCapacity()) {
        $players = $room->getPlayers();
        $players[] = $email;
        $room->setPlayers($players);
        $room->incrementPlayerCount();
    }

    if (!$room->getCurrentTurn()) {
        $room->setCurrentTurn($room->getPlayers()[0] ?? null);
    }

    $em->flush();

    $update = new Update("/rooms/{$room->getId()}", json_encode([
        'board' => $room->getBoard(),
        'currentTurn' => $room->getCurrentTurn(),
        'winner' => $room->getWinner(),
        'playerCount' => $room->getPlayerCount()
    ]));
    $hub->publish($update);
```

Un contrôleur pour la gestion de l'utilisateur : Il contient les méthodes classique "login" et "logout" ainsi qu'une méthode permettant de récupérer les infos utilisateurs "profile".

```php
#[Route('/login', name: 'login', methods: ['POST'])]
public function login(Request $request, EntityManagerInterface $em, UserPasswordHasherInterface $hasher): JsonResponse {
    $data = json_decode($request->getContent(), true);
    $email = $data['email'] ?? null;
    $password = $data['password'] ?? null;

    if (!$email || !$password) {
        return new JsonResponse(['error' => 'Missing credentials'], 400);
    }

    $user = $em->getRepository(User::class)->findOneBy(['email' => $email]);

    if (!$user || !$hasher->isPasswordValid($user, $password)) {
        return new JsonResponse(['error' => 'Invalid credentials'], 401);
    }

    return new JsonResponse([
        'id' => $user->getId(),
        'email' => $user->getUserIdentifier(),
        'roles' => $user->getRoles(),
    ]);
}

#[Route('/logout', name: 'logout', methods: ['POST', 'GET'])]
public function logout(Request $request): JsonResponse
{
    return new JsonResponse(['message' => 'Logged out']);
}
```

Un contrôleur pour le chat : Ce contrôleur ne contient qu'une méthode qui permet d'envoyer un message et le transmettre au websocket.

```php
/**
 * Route utilisée pour envoyer un message dans le websocket
 */
#[Route('/chat/send', name: 'chat_send', methods: ['POST'])]
public function send(Request $request, HubInterface $hub): JsonResponse
{
    $data = json_decode($request->getContent(), true);
    $message = $data['message'] ?? '';

    if (!$message) {
        return new JsonResponse(['error' => 'Message is empty'], 400);
    }

    // Publier le message sur le topic "chat"
    $update = new Update(
        'chat/general', // topic
        json_encode(['message' => $message, 'timestamp' => time()])
    );

    $hub->publish($update);

    return new JsonResponse(['status' => 'Message sent']);
}
```

Il y a également un service pour les mécaniques de jeu : Ce service contient des fonctions génériques liées au jeu comme “checkWin” ou “isBoardFull”.

```php
// Utility to check win condition
public function checkWin(array $board, string $mark): bool
{
    // Check rows and columns
    for ($i = 0; $i < 3; $i++) {
        if (
            ($board[$i][0] === $mark && $board[$i][1] === $mark && $board[$i][2] === $mark) ||
            ($board[0][$i] === $mark && $board[1][$i] === $mark && $board[2][$i] === $mark)
        ) {
            return true;
        }
    }

    // Check diagonals
    if (
        ($board[0][0] === $mark && $board[1][1] === $mark && $board[2][2] === $mark) ||
        ($board[0][2] === $mark && $board[1][1] === $mark && $board[2][0] === $mark)
    ) {
        return true;
    }

    return false;
}

public function isBoardFull(array $board): bool
{
    foreach ($board as $row) {
        foreach ($row as $cell) {
            if ($cell === '') return false;
        }
    }
    return true;
}
```

Authentification

L’authentification est gérée à la main sans utilisation des mécanismes existants dans symfony. L’utilisateur est comparé avec les utilisateurs en bdd. S’il existe et que le mot de passe est bon, il est alors envoyé au front pour le stocker dans le localStorage. Voir partie précédente pour le code.

Rooms et chat global

La gestion des rooms se fait à travers le contrôleur ‘RoomController’.

On y retrouve :

- Une méthode de création de room

```php
/**
 * Créé une room associé à l'utilisateur donné dans la requête POST
 */
#[Route('/create-room', name: 'create_room', methods: ['POST'])]
public function createRoom(Request $request, EntityManagerInterface $em): JsonResponse {
    $data = json_decode($request->getContent(), true);

    // Validate incoming data
    if (!isset($data['name'], $data['email'])) {
        return $this->json(['error' => 'Missing room name or email'], 400);
    }

    $user = $em->getRepository(User::class)->findOneBy(['email' => $data['email']]);

    if (!$user) {
        return $this->json(['error' => 'User not found'], 404);
    }

    $room = new Room();
    $room->setName($data['name']);
    $room->setOwner($user);

    $em->persist($room);
    $em->flush();

    return $this->json([
        'id' => $room->getId(),
        'name' => $room->getName(),
        'owner' => $user->getEmail(),
    ]);
}
```

- Une méthode de suppression de room

```php
//Supprime la room avec l'ID joint pour l'utilisateur donné dans la requête
#[Route('/rooms/{id}', name: 'delete_room_by_email', methods: ['DELETE'])]
public function deleteRoomByEmail(int $id, Request $request, EntityManagerInterface $em): JsonResponse {
    $data = json_decode($request->getContent(), true);

    if (!isset($data['email'])) {
        return $this->json(['error' => 'Missing email'], 400);
    }

    $user = $em->getRepository(User::class)->findOneBy(['email' => $data['email']]);
    if (!$user) {
        return $this->json(['error' => 'User not found'], 404);
    }

    $room = $em->getRepository(Room::class)->find($id);
    if (!$room) {
        return $this->json(['error' => 'Room not found'], 404);
    }

    if ($room->getOwner() !== $user) {
        return $this->json(['error' => 'You do not own this room'], 403);
    }

    $em->remove($room);
    $em->flush();

    return $this->json(['success' => true]);
}
```

- Deux méthodes pour récupérer des rooms (toutes les rooms et toutes les rooms d'un utilisateur)

```php
/**
 * Récupère toutes les rooms disponibles
 */
#[Route('/rooms', name: 'get_rooms', methods: ['GET'])]
public function getRooms(EntityManagerInterface $em): JsonResponse
{
    $rooms = $em->getRepository(Room::class)->findAll();

    $data = array_map(fn(Room $room) => [
        'id' => $room->getId(),
        'name' => $room->getName(),
        'owner' => $room->getOwner()?->getEmail(),
    ], $rooms);

    return $this->json($data);
}

/**
 * Récupère les rooms de l'utilisateur
 */
#[Route('/my-rooms', name: 'get_my_rooms', methods: ['POST'])]
public function getRoomsByEmail(Request $request, EntityManagerInterface $em): JsonResponse {
    $data = json_decode($request->getContent(), true);

    if (!isset($data['email'])) {
        return $this->json(['error' => 'Missing email'], 400);
    }

    $user = $em->getRepository(User::class)->findOneBy(['email' => $data['email']]);

    if (!$user) {
        return $this->json(['error' => 'User not found'], 404);
    }

    $rooms = $user->getRooms();

    $result = array_map(fn($room) => [
        'id' => $room->getId(),
        'name' => $room->getName(),
    ], $rooms->toArray());

    return $this->json($result);
}
```

- Une méthode pour rejoindre une room qui envoie un event par le websocket

```php
/**
 * Permet de mettre à jour la room et le game state quand un joueur rejoins
 */
#[Route('/rooms/{id}/join', name: 'join_room', methods: ['POST'])]
public function joinRoom(Request $request, Room $room, EntityManagerInterface $em, HubInterface $hub): JsonResponse
{
    $data = json_decode($request->getContent(), true);
    $email = $data['email'] ?? null;

    if (!$email) {
        return $this->json(['error' => 'Missing email'], 400);
    }

    if ($room->getPlayerCount() >= $room->getCapacity()) {
        return $this->json(['error' => 'Room is full'], 400);
    }

    if (!in_array($email, $room->getPlayers()) && $room->getPlayerCount() < $room->getCapacity()) {
        $players = $room->getPlayers();
        $players[] = $email;
        $room->setPlayers($players);
        $room->incrementPlayerCount();
    }

    if (!$room->getCurrentTurn()) {
        $room->setCurrentTurn($room->getPlayers()[0] ?? null);
    }

    $em->flush();

    $update = new Update("/rooms/{$room->getId()}", json_encode([
        'board' => $room->getBoard(),
        'currentTurn' => $room->getCurrentTurn(),
        'winner' => $room->getWinner(),
        'playerCount' => $room->getPlayerCount()
    ]));
    $hub->publish($update);

    return new JsonResponse(['status' => 'joined room']);
}
```

- Une méthode pour quitter une room qui envoie aussi un évènement.

```php
/**
 * Permet de mettre à jour la room et le game state quand un joueur quitte
 */
#[Route('/rooms/{id}/leave', name: 'leave_room', methods: ['POST'])]
public function leaveRoom(Request $request, Room $room, EntityManagerInterface $em, HubInterface $hub): JsonResponse
{
    $data = json_decode($request->getContent(), true);
    $email = $data['email'] ?? null;

    if (!$email) {
        return $this->json(['error' => 'Missing email'], 400);
    }

    if ($room->getCurrentTurn() == $email) {
        $room->setCurrentTurn(null);
    }
    $room->decrementPlayerCount();
    $em->flush();

    $update = new Update("/rooms/{$room->getId()}", json_encode([
        'board' => $room->getBoard(),
        'currentTurn' => $room->getCurrentTurn(),
        'winner' => $room->getWinner(),
        'playerCount' => $room->getPlayerCount()
    ]));
    $hub->publish($update);

    return $this->json("{'message' => 'Successfully left room'}");
}
```

Pour le chat global c'est le contrôleur 'ChatController' qui s'en charge avec une seule méthode qui permet d'envoyer le message au websocket qui va redistribuer vers les clients abonnés au canal.

```php
class ChatController extends AbstractController
{
    /**
     * Route utilisée pour envoyer un message dans le websocket
     */
    #[Route('/chat/send', name: 'chat_send', methods: ['POST'])]
    public function send(Request $request, HubInterface $hub): JsonResponse
    {
        $data = json_decode($request->getContent(), true);
        $message = $data['message'] ?? '';

        if (!$message) {
            return new JsonResponse(['error' => 'Message is empty'], 400);
        }

        // Publier le message sur le topic "chat"
        $update = new Update(
            'chat/general', // topic
            json_encode(['message' => $message, 'timestamp' => time()])
        );

        $hub->publish($update);

        return new JsonResponse(['status' => 'Message sent']);
    }
}
```

Communication Temps Réel

Dans ce projet le websocket utilisé est Mercure qui est une fonctionnalité native de symfony.

Cependant Mercure n'est pas un vrai websocket mais un protocole permettant de faire du SSE (Server-sent events). Le serveur va pouvoir envoyer des événements vers les clients qui suivent des canaux précis. Cependant les clients ne peuvent pas discuter directement avec Mercure, on doit passer par une méthode du back.

Le Hub Mercure est installé et lancé par un docker compose qui a une image mercure prête à l'emploi.

```yaml
services:
  mercure:
    image: dunglas/mercure
    restart: unless-stopped
    environment:
      SERVER_NAME: ':80'   # désactive HTTPS, Mercure écoute sur le port 80
      MERCURE_PUBLISHER_JWT_KEY: '!ChangeThisMercureHubJWTSecretKey!'
      MERCURE_SUBSCRIBER_JWT_KEY: '!ChangeThisMercureHubJWTSecretKey!'
      MERCURE_JWT_SECRET: '!ChangeThisMercureHubJWTSecretKey!'
      MERCURE_EXTRA_DIRECTIVES: |
        cors_origins *
        allow_anonymous
        anonymous
        allow_anonymous_subscription true
    ports:
      - '3000:80'          # expose le port 80 container vers port 3000 host
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:80/healthz"]
      timeout: 5s
      retries: 5
      start_period: 60s
    volumes:
      - mercure_data:/data
      - mercure_config:/config

volumes:
  mercure_data:
  mercure_config:
```

Pour prendre l'exemple du chat :

- Le client a un formulaire qui envoie une requête au back quand il est soumis

```
const sendMessage = async () => {
  if (!newMessage.value.trim()) return;

  try {
    await fetch(api_url + '/chat/send', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ message: newMessage.value }),
    });
    newMessage.value = '';
  } catch (error) {
    console.error('Erreur envoi message:', error);
  }
};
```

- Le back notifie mercure qu'un message est arrivé

```
/**
 * Route utilisée pour envoyer un message dans le websocket
 */
#[Route('/chat/send', name: 'chat_send', methods: ['POST'])]
public function send(Request $request, HubInterface $hub): JsonResponse
{
    $data = json_decode($request->getContent(), true);
    $message = $data['message'] ?? '';

    if (!$message) {
        return new JsonResponse(['error' => 'Message is empty'], 400);
    }

    // Publier le message sur le topic "chat"
    $update = new Update(
        'chat/general', // topic
        json_encode(['message' => $message, 'timestamp' => time()])
    );

    $hub->publish($update);

    return new JsonResponse(['status' => 'Message sent']);
}
```

- Mercure envoie une événement à tous les clients qui écoutent pour dire qu'un message est arrivé ➜ le front se met à jour avec le nouveau message.

```
onMounted(() => {
  const url = new URL('http://localhost:3000/.well-known/mercure');
  url.searchParams.append('topic', 'chat/general');

  eventSource = new EventSource(url.toString());

  eventSource.onmessage = event => {
    const data = JSON.parse(event.data);
    messages.value.push(data);
  };

  eventSource.onerror = err => {
    console.error('Erreur Mercure:', err);
    eventSource.close();
  };
});
```

Le client s'abonne automatiquement au canal 'chat/general' si le composant 'Chat.vue' est dans la page. Quand le composant est démonté, la connexion avec Mercure est fermée.

```
onBeforeUnmount(() => {
  if (eventSource) {
    eventSource.close();
  }
});
```

C’est le même principe pour le canal du jeu avec quelques petites subtilités pour permettre au jeu de fonctionner :

- Le client avertit le serveur que le joueur a joué

```
try {
  const response = await fetch(`https://localhost:8000/rooms/${roomId}/play`, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({
      row,
      col,
      user: user,
    }),
  })

  const data = await response.json()

  if (!response.ok) {
    error.value = data.error || 'Failed to play move'
  }
} catch {
  error.value = 'Network error'
}
```

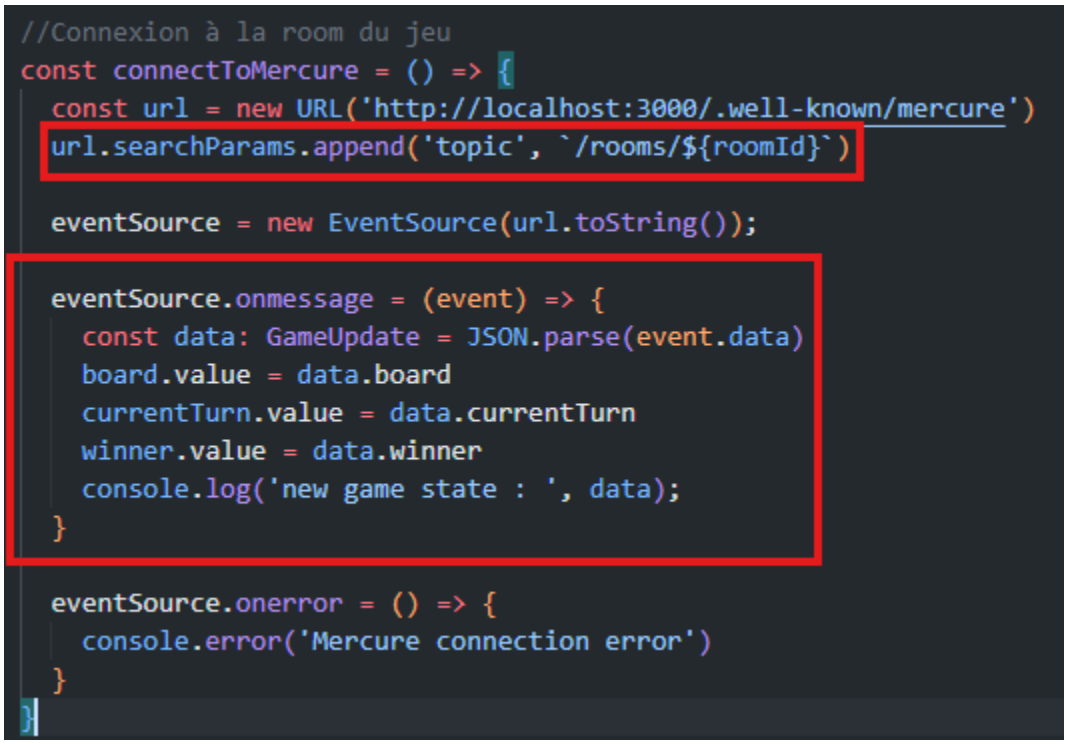
- Le serveur s’occupe de la gestion du jeu et envoie à mercure le nouvel état de jeu pour que mercure s’occupe de notifier les clients

```
// Switch turn
$nextPlayer = ($players[0] === $playerId) ? $players[1] : $players[0];
$room->setCurrentTurn($nextPlayer);

$room->setBoard($board);
$em->flush();

// Notify via Mercure
$update = new Update("/rooms/{$id}", json_encode([
    'board' => $board,
    'currentTurn' => $room->getCurrentTurn(),
    'winner' => $room->getWinner(),
    'players' => $players,
]));
$hub->publish($update);
```

- Le front est abonné au canal du jeu et se met à jour quand il reçoit une événement. Chaque room a un canal séparé, le client qui entre dans une room s'abonne au canal de la room en question.



```
//Connexion à la room du jeu
const connectToMercure = () => {
  const url = new URL('http://localhost:3000/.well-known/mercure')
  url.searchParams.append('topic', `/rooms/${roomId}`)

  eventSource = new EventSource(url.toString());

  eventSource.onmessage = (event) => {
    const data: GameUpdate = JSON.parse(event.data)
    board.value = data.board
    currentTurn.value = data.currentTurn
    winner.value = data.winner
    console.log('new game state : ', data);
  }

  eventSource.onerror = () => {
    console.error('Mercure connection error')
  }
}
```

Test d'Application

Tests unitaires

Les tests unitaires côté backend se sont concentrés sur la mécanique de jeu notamment sur le 'GameService'.

```php
class GameServiceTest extends TestCase
{
    private GameService $game;

    protected function setUp(): void
    {
        $this->game = new GameService();
    }

    public function testWinByRow(): void
    {
        $board = [
            ['X', 'X', 'X'],
            ['', '', ''],
            ['', '', ''],
        ];
        $this->assertTrue($this->game->checkWin($board, 'X'));
    }

    public function testWinByColumn(): void
    {
        $board = [
            ['O', '', ''],
            ['O', '', ''],
            ['O', '', ''],
        ];
        $this->assertTrue($this->game->checkWin($board, 'O'));
    }
```

Les tests unitaire côté frontend se sont focalisés sur le chat :

```
describe('Chat.vue', () => {
  beforeEach(() => {
    vi.resetAllMocks()
  })

  it('renders messages from EventSource', async () => {
    const wrapper = mount(Chat)

    const mockEvent = {
      data: JSON.stringify({ message: 'Hello', timestamp: 1234567890 }),
    }

    // Simulate receiving a message from Mercure
    wrapper.vm.eventSource.onmessage(mockEvent)
    await flushPromises()

    expect(wrapper.text()).toContain('Hello')
  })

  it('sends a message and clears input', async () => {
    global.fetch = vi.fn(() =>
      Promise.resolve({ ok: true, json: () => Promise.resolve({}) })
    ) as any

    const wrapper = mount(Chat)

    const input = wrapper.find('input')
    await input.setValue('Bonjour le monde')
    await input.trigger('keyup.enter')

    await flushPromises()

    expect(fetch).toHaveBeenCalled()
    expect(wrapper.find('input').element.value).toBe('')
  })
```

Tests End-to-End

Les tests end-to-end sont faits en utilisant Playwright.

Test connexion + page d'accueil :

```
test('homepage loads and shows title', async ({ page }) => {
    //Auth first then test
    await page.goto('http://localhost:5173/login');
    await page.fill('input[type="email"]', 'test@test.fr');
    await page.fill('input[type="password"]', 'test');
    await page.click('button[type="submit"]');
    await page.waitForURL('http://localhost:5173'); // wait after login

    // Check the title
    await expect(page).toHaveTitle("Vite App")

    // Check for h1
    await expect(page.locator('h1')).toContainText('Roomies')

    // Check if "Room" heading exists
    await expect(page.locator('h2')).toContainText('Rooms disponibles')
})
```

Test du chat :

```
test('user can send a message', async ({ page }) => {
    //Auth first then test
    await page.goto('http://localhost:5173/login');
    await page.fill('input[type="email"]', 'test@test.fr');
    await page.fill('input[type="password"]', 'test');
    await page.click('button[type="submit"]');
    await page.waitForURL('http://localhost:5173'); // wait after login

    const input = page.locator('input[placeholder="Tape ton message"]')
    await input.fill('Hello from test')
    await input.press('Enter')

    // Wait for the message to appear
    await expect(page.locator('.message')).toContainText('Hello from test')
})
```