

Rapport Technique Complet : Projet Roomies

Introduction

Roomies est une application web de jeux en ligne multijoueurs, avec une communication en temps réel, un système de chat global et par room, une authentification JWT, un espace personnel personnalisable, et des fonctionnalités sociales évolutives.

Le projet repose sur une architecture moderne :

- Frontend : **Vue.js 3 + PrimeVue**
 - Backend : **Symfony 7.2.6**
 - Temps réel : **Socket.IO (Node.js)**
 - Orchestration : **Docker Compose**
 - Tests : **PHPUnit, Vitest, Cypress**
-

Sommaire du Rapport

1. Architecture Générale & Choix Techniques
 2. Frontend Vue.js
 3. Backend Symfony
 4. Communication Temps Réel (WebSocket)
 5. Tests d'application
-

1. Architecture Générale & Choix Techniques

- Architecture en conteneurs Docker (frontend, backend, websocket, db)
- Communication par réseaux internes (frontend-backend, backend-db)
- JWT pour authentification, MySQL pour stockage
- WebSocket découplé (Socket.IO)
- Jeux gérés : Morpion (synchro), 2048 et Echecs (localement)
- Fonctionnalités optionnelles intégrées : chat amélioré, favoris, profil, amis

2. Frontend Vue.js

- Vue.js 3 + Composition API + TypeScript
- PrimeVue + Tailwind pour design moderne et responsive
- Composants : Navigation, Chat, Modales, GameBoard, etc.
- Stores Pinia : **useAuthStore**, **useChatStore**

- Vue Router avec guards par rôle et token
- WebSocket Client : Socket.IO pour le chat et les jeux
- Fonctionnalités : SPA, profil, rooms, chat global, modales de jeux

3. Backend Symfony

- API REST sécurisée (routes `/api`) avec JWT
- Entités : `User`, `Game`, `Room`, `Friendship`
- Repositories personnalisés Doctrine
- Contrôleurs : `UserController`, `RoomController`, `AuthController`, etc.
- Routes commentées avec `#[Route]`
- Fixtures de démarrage pour les jeux disponibles
- Validation stricte via `Assert`
- Architecture claire : Controller → Service → Repository

4. Communication Temps Réel (WebSocket)

- Serveur indépendant en Node.js (port 3000)
- WebSocket via Socket.IO
- Canaux : `chat_<roomId>`, `room_<roomId>`, `global`
- Attribution des rôles X/O dans le jeu Morpion
- Spectateurs si room pleine, synchronisation du plateau en temps réel
- Nettoyage des connexions, mise à jour dynamique des rooms
- Architecture décentralisée : gestion des états via `Map`

5. Tests d'application

- PHPUnit : tests unitaires (`UserTest`, `GameTest`), fonctionnels (controllers), et d'intégration (auth, room)
- Cypress : tests e2e complets sur l'auth, le profil, les rooms, et le chat
- Vitest : unitaires sur composants Vue et stores (auth, login)
- Couverture large : cas valides, erreurs, statuts HTTP, validations
- CI exécutable via `make test` (Makefile inclus)

Partie 1 — Architecture Générale & Choix Techniques

Objectif du projet

Le projet **Roomies** a pour vocation de proposer une plateforme de jeux multijoueurs en ligne intégrant :

- La **création et gestion de rooms** (salons de jeu),
- L'**interaction en temps réel** via WebSocket (chat et jeu),
- Des **jeux embarqués** (morpion fonctionnel, 2048 et échecs),
- Un **chat global** ainsi que des chats par room,
- Un **système d'authentification sécurisé**,
- Un **profil utilisateur personnalisable**,
- Et un **système social en cours** (ajout d'amis, favoris).

Le projet a été conçu en respectant les exigences d'un environnement **fullstack moderne**, conteneurisé avec **Docker**, orienté API, réactif et scalable.

Architecture technique

Conteneurisation Docker

L'application repose sur une architecture conteneurisée avec **docker-compose**. Elle est composée des services suivants :

Service	Description
frontend	Application Vue.js 3 servie sur le port 5173
backend	API Symfony 7 sur le port 8000, connecté à la base de données
realtime-server	Serveur WebSocket Socket.IO, écoute sur le port 3000

mysql SGBD MySQL 8, persistance des données

phpmyadmin Interface d'administration de la base pour développement/test

Deux réseaux sont définis :

- **frontend-backend** : communication entre front, back et serveur WebSocket
- **backend-db** : communication backend ↔ base de données

Structure des répertoires

L'application est structurée ainsi :

```
roomies/
├── backend/      → API Symfony (routes REST, services, entités, sécurité)
├── frontend/     → Vue.js 3 (pages, composants, chat, jeux)
├── realtime-server/ → Serveur WebSocket Socket.IO (Node.js)
├── bdd/          → Scripts d'initialisation SQL
└── docker-compose.yml → Orchestration multi-conteneurs
```

Stack Technique détaillée

Frontend : Vue.js 3

- **Vue.js 3** avec **Composition API**
- **PrimeVue** pour les composants UI réutilisables
- **Vue Router** pour la navigation (Home, Room, Profil)
- **Gestion d'état** via **store** (Pinia)
- **Authentification** :
 - JWT stocké côté client
 - Appels sécurisés vers l'API Symfony via **axios**
- **Connexion WebSocket** avec **socket.io-client**
- **Stockage persistant des messages** via **localStorage**

Backend : Symfony 7.2.6

- Architecture RESTful (routes sous `/api`)
- **LexikJWTAuthenticationBundle** pour l'authentification par token
- Sécurité :
 - Accès REST uniquement pour les utilisateurs authentifiés
 - CORS restreint au développement (`localhost`, `127.0.0.1`)
- ORM : **Doctrine**, avec mapping des entités (User, Room, etc.)
- Configuration centralisée via fichier `.env`

```

DATABASE_URL="mysql://admin_user:admin_password@mysql:3306/roomies?serverVersion=8.0.32&charset=utf8mb4"
JWT_SECRET_KEY=/config/jwt/private.pem
JWT_PUBLIC_KEY=/config/jwt/public.pem
JWT_PASSPHRASE=test_passphrase

```

WebSocket : Node.js + Socket.IO

Serveur développé avec :

- `http.createServer()` + `new Server()` de **Socket.IO**
- CORS activé pour le développement
- **Rooms Socket.IO** :
 - `chat_<roomId>` : canal de chat
 - `room_<roomId>` : canal de jeu
- **Attribution des rôles** dans le morpion (X / O)
- **Synchronisation d'état** : chaque changement dans le jeu est renvoyé à tous les membres connectés à la room
- **Gestion des connexions/déconnexions** :
 - Comptage des participants
 - Attribution/désaffectation des joueurs

- Diffusion d'état actualisé à chaque événement
-

Jeux intégrés

1. Morpion (Tic-Tac-Toe)

- **Fonctionnel**
- Communication en **temps réel** : synchronisation des mouvements, gestion des rôles, détection de victoire ou match nul
- État maintenu côté serveur via **Map JS**
- Données transmises via WebSocket à chaque interaction

2. 2048 (*partiellement intégré*)

- Logique du jeu présente et jeu fonctionnel
- Non encore connectée à WebSocket (jeu solo ou à synchroniser)

3. Échecs (*prototype*)

- Plateau et pions fonctionnel
- Règles en cours
- Communication en temps réel à implémenter

Chat

Chat global

- Accessible uniquement aux utilisateurs connectés
- Messages diffusés à tous les utilisateurs via WebSocket
- Persisté temporairement en local (stocké dans le store client)
- Mécanisme de **purge automatique** :

```
if (this.messages.length > 300) {  
  this.messages.splice(0, this.messages.length - 100) }  
}
```

Chat par room

- Accessible uniquement aux membres actifs d'une room
 - Basé sur un canal dédié `chat_<roomId>`
 - Messages transmis uniquement aux utilisateurs autorisés
-

Gestion utilisateur

- **Création de compte / Connexion / Déconnexion**
 - **JWT** délivré à l'authentification, utilisé dans les appels API
 - **Page de profil personnalisable :**
 - Modification des données utilisateur
 - Changement d'avatar
 - Liste des jeux favoris
 - **Système d'amis** débuté (ajout possible, fonctionnalités à compléter)
-

Fonctionnalités optionnelles implémentées

Fonctionnalité	État	Détail
Jeux supplémentaires	Partiellement OK	2048 et Échecs intégrés localement, WebSocket à implémenter
Améliorations du chat	OK	Historique local, purge automatique, multi-canal

Fonctionnalités
sociales

En cours

Ajout d'amis, profil personnalisé, favoris de jeux

Sécurité & Accès

- Utilisation des clés privées/publics JWT
- Middleware de sécurité sur routes `/api/**`
- CORS contrôlé (restreint au développement local)
- Pas d'accès aux endpoints API sans token valide

Conclusion de la partie

Le projet Roomies repose sur une architecture modulaire et évolutive adaptée aux exigences d'une application web interactive temps réel. Il allie les **bonnes pratiques de conteneurisation**, une **communication fluide WebSocket**, et une **séparation claire des responsabilités** entre frontend, backend et moteur de synchronisation.

Le reste du rapport détaillera le fonctionnement précis de chaque brique (Vue.js, Symfony, WebSocket, tests).

Partie 2 — Frontend Vue.js avec TypeScript

Vue d'ensemble

L'application frontend de **Roomies** est développée avec **Vue.js 3** utilisant la **Composition API**, écrite en **TypeScript**, et intègre la bibliothèque **PrimeVue** pour les composants UI. Le projet est modulaire, bien structuré et suit les bonnes pratiques de conception Vue.

L'ensemble de l'application est encapsulé dans un conteneur Docker, permettant une intégration fluide avec le backend Symfony et le serveur WebSocket.

Architecture et organisation des composants

Structure des dossiers Vue :

```
src/
├── assets/           # Fichiers CSS globaux (main.css)
├── components/      # Composants réutilisables (chat, jeu,
navigation)
│   ├── games/       # Morpion, Echecs, 2048
│   └── icons/        # Icônes personnalisées
├── lib/             # Fonctions utilitaires globales (utils.ts)
├── plugins/          # Socket.IO (socket.ts)
├── router/           # Configuration du routeur (index.ts)
├── stores/           # Pinia stores (auth, chat)
├── types/            # Types TypeScript
├── views/            # Pages principales (About, Home, Room, etc.)
├── App.vue           # Composant racine
└── main.ts           # Point d'entrée de l'application
```

Principaux composants personnalisés :

- **Navigation.vue** : Header et menu de navigation
- **GlobalChat.vue** / **GameChat.vue** : Composants de chat global et par room
- **Morpion.vue**, **2048.vue**, **Echecs.vue** : Composants de jeux
- **FavoriteModal.vue** : Modale pour ajout de jeux favoris
- **GameBoard.vue**, **GameCell.vue** : Composants graphiques de plateau

Exemple : **ProfileView.vue**

Permet à l'utilisateur de modifier son profil, son mot de passe, choisir un avatar (avatars pré-chargés via `/public/img/avatar`) et gérer ses jeux favoris.

Routage et navigation

La navigation est gérée via **vue-router** dans `router/index.ts` avec des **guards globaux** pour la sécurité.

Routes principales

```
/ // HomeView
/auth // AuthView (login/register)
/profile // ProfileView
/rooms // RoomListView
/rooms/:id // RoomView (accès à une room spécifique)
/rooms/game/:slug // Liste de rooms d'un jeu
/admin // AdminView (protégé par
ROLE_ADMIN)
/contact // ContactView
```

Sécurité via **meta**

```
meta: { requiresAuth: true, requiresAdmin: true }
```

Les routes sensibles sont protégées par des guards. Le token JWT est vérifié à chaque navigation.

Gestion de l'état avec Pinia

Deux stores principaux sont définis :

useAuthStore.ts

- Authentification : login/logout/token
- Stockage des données utilisateur : id, email, avatar, rôles
- Décodage du JWT pour extraire les infos
- Vérification d'expiration de token

useChatStore.ts

- Stockage des messages
- Persistant localement (via `localStorage`)
- Nettoyage auto des anciens messages si plus de 300

Fonctionnalités implémentées

Fonctionnalité Vue.js	Statut
Authentification JWT	Fonctionnel (formulaire, token, guards)
Navigation SPA	Full Vue Router, sans rechargement
Liste des rooms	Liste avec filtres et accès direct
Jeux embarqués	Morpion (synchro), 2048 et Echecs présents
Chat global & room	Socket.IO intégrés, composants réactifs
Profil utilisateur	Avatar, infos, favoris

Qualité du code

- Tous les composants sont typés avec **TypeScript**
- Utilisation de `defineProps`, `defineEmits`, `ref`, `computed`, `onMounted`
- Code commenté et clair (ex : `FavoriteModal.vue`)
- Styles cohérents avec TailwindCSS + effets visuels personnalisés (neon, modales, transitions)

Tests Vue.js

Le projet utilise **Vitest** pour les tests unitaires, avec la structure suivante :

```
tests/
├── integration/
│   └── ProfileView.integration.test.ts
├── unit/
│   ├── components/Login.test.ts
│   └── stores/useAuthStore.test.ts
```

- Test unitaire du store d'authentification (gestion du token)
- Test du composant `Login.vue`
- Test d'intégration de `ProfileView.vue` (fetch, interactions formulaire)

Possibilité d'ajouter des tests pour : `GlobalChat.vue`, `FavoriteModal.vue`, `RoomView.vue`, etc.

Conclusion

La partie frontend est robuste, moderne et bien organisée :

- Bonne architecture (pages/écrans + composants réutilisables)
- Authentification et routage sécurisés
- Communication WebSocket efficace
- Design cohérent grâce à PrimeVue + TailwindCSS
- Tests unitaires/intégration présents

Cette architecture permet une évolutivité simple (ajout de nouveaux jeux, thèmes, ou interactions sociales).

Partie 3 — Backend Symfony

Vue d'ensemble

Le backend du projet **Roomies** repose sur le framework **Symfony 7.2.6**, configuré comme une API REST sécurisée, modulaire et stateless. Il est conçu autour d'une architecture claire et maintenable, avec une séparation nette des responsabilités :

- **Contrôleurs REST personnalisés** pour chaque domaine fonctionnel (users, rooms, jeux, admin, auth)
- **Doctrine ORM** pour la gestion des entités relationnelles
- **Authentification JWT** avec LexikJWTAuthenticationBundle
- **Gestion des rôles**, validation des données et système de favoris
- **Tests unitaires et fonctionnels** structurés et pertinents

Le backend ne persiste pas les messages de chat : cette logique est entièrement déportée sur le frontend et sur le serveur WebSocket.

Architecture Symfony

Structure du code dans **src/**

```
src/  
├── Controller/           # Contrôleurs REST (User, Room, Game, Auth,  
Admin)  
├── Entity/              # Entités Doctrine : User, Room, Game,  
Friendship, StatusType  
├── Repository/          # Repositories Doctrine personnalisés  
└── DataFixtures/# Jeux préchargés (GameFixtures)
```

Configuration (dans **config/**)

- **security.yaml** : définit les firewalls, providers, rôles et access control
- **lexik_jwt_authentication.yaml** : config des clés JWT (pub/priv)
- **nelmio_cors.yaml** : CORS configuré pour localhost
- **services.yaml** : injection automatique des dépendances
- **routes.yaml** : mapping des routes (automatique via annotations)

Gestion des utilisateurs (User)

Entité **User**

- Champs : `id`, `email`, `username`, `password`, `roles`, `avatar`, `createdAt`, `lastActive`
- Relations :
 - `rooms` (OneToMany)
 - `favoris` (ManyToMany avec Game)
 - `sentFriendRequests` / `receivedFriendRequests` (OneToMany sur Friendship)

Authentification et JWT

- Bundle utilisé : **LexikJWTAuthenticationBundle**
- Clés dans `/config/jwt/`
- Auth via `/api/login` et `/api/register`
- Routes `/api/**` protégées par firewall JWT (stateless)

```
access_control:  
- { path: ^/api, roles: IS_AUTHENTICATED_FULLY }
```

Contrôleur : **UserController**

- `GET /api/users/profile` : infos utilisateur courant
- `POST /api/users/update` : modifie email, username, password, avatar (+ renvoi token)
- `GET /api/users/favorites` : liste des jeux favoris
- `POST /api/users/favorites/{id}` : ajout d'un jeu aux favoris
- `DELETE /api/users/favorites/{id}` : suppression d'un favori

Gestion des rooms et des jeux

Entités Doctrine

- `Room` : identifiant, titre, max joueurs, propriétaire (User), statut
- `Game` : `name`, `image`, `description`
- `Friendship` : système d'amis (en cours d'évolution)

Contrôleurs

- **RoomController** :
 - CRUD des rooms (create/list/delete)
 - Liens avec les utilisateurs créateurs (owner)
- **GameController** :
 - Exposition des jeux disponibles
 - Intégration des favoris via User

Fixtures

- **GameFixtures.php** : précharge les jeux 2048, morpion, echecs
 - Insertion via : `php bin/console doctrine:fixtures:load`
-

Sécurité et accès

Configuration **security.yaml**

```
firewalls:
  api:
    pattern: ^/api
    stateless: true
    jwt: ~
```

- Authentification **stateless** par token JWT uniquement
- Provider : **username**
- Password hashing automatique via **password_hashers**

Gestion des rôles

- Par défaut : **ROLE_USER**
- Support des rôles additionnels : **ROLE_ADMIN**
- Guard de route pour `/admin` via `meta.requiresAdmin`

Tests Backend

Structure

```
tests/
├── Controller/           # Tests fonctionnels HTTP des endpoints
├── Integration/Controller # Tests avec base de données simulée
└── Unit/Entity           # Tests des entités et logique métier
```

Exemple : `UserTest.php`

- Test des setters/getters
- Test de validité du JWT identifier
- Test des rôles, avatar, date de création auto, collection de favoris

Exemple : `UserControllerTest.php`

- Accès au profil, modification des données, gestion des favoris
 - Couverture des cas classiques et erreurs (ex : jeu non trouvé)
-


Documentation & Qualité

- **Code commenté** dans tous les contrôleurs (User, Auth, Room, Game, Admin)
 - Routes déclarées via **annotations Symfony 7** (`#[Route]`)
 - Sérialisations via **Groups** (`@Groups`) pour structurer les JSON
 - **Validation** des données utilisateurs via `Assert` et `ValidatorInterface`
 - **Fixtures** et scripts de restauration pour la BDD (`restore-db.sh`)
 - Gestion évolutive avec `RoomRepository`, `GameRepository`, `FriendshipRepository`
-

Fonctionnalités Symfony validées

Fonctionnalité	✓ Statut	Détail
CRUD Utilisateurs	✓ Complet	Profil, update, token JWT renvoyé
CRUD Rooms	✓ Complet	Création, suppression, listing
CRUD Jeux / favoris	✓ Complet	Via GameController + UserController
Authentification & JWT	✓ Complet	Token JWT avec rôles et avatar
Rôle admin / route <code>/admin</code>	✓ Présent	Protections à l'accès
Tests Symfony unitaires + intégration	✓ Présents	User, Game, Auth, Room

Messages via WebSocket (non persistés)

 Non gérés par le backend

Conclusion

Le backend Symfony est rigoureusement conçu :

- Architecture claire, code bien structuré
- Authentification robuste via JWT
- Relations Doctrine modélisées précisément
- Couverture test acceptable (unitaires et fonctionnels)
- Commentaires clairs, conventions respectées

Partie 4 — Communication Temps Réel (WebSocket)

Vue d'ensemble

La communication en temps réel de l'application **Roomies** repose sur un serveur WebSocket dédié, implémenté en **Node.js** avec **Socket.IO**. Ce serveur est totalement découplé du backend Symfony et fonctionnellement centré sur :

- Le **chat global** (accessible à tous les utilisateurs connectés)
- Le **chat de chaque room** (canal privé pour les membres)
- La **gestion temps réel du jeu Morpion** (attribution de rôle, synchronisation des coups, détection de fin de partie)
- La **gestion des connexions / déconnexions**, et des participants actifs

Le tout est orchestré depuis un conteneur **realtime-server** dans Docker, qui écoute sur le port **3000**.

Stack technique WebSocket

Composant	Détail
Serveur WS	Node.js + socket.io
Client WS	socket.io-client dans Vue.js frontend
Communication	bidirectionnelle, événements custom
Authentification	basée sur l'ID utilisateur + username (non JWT)
Structure	room_<id> pour le jeu, chat_<id> pour le chat

Gestion des canaux et événements

Chat global

- Canal : broadcast global (pas de room Socket.IO)
- Événement reçu : **chat:message**
- Diffusé à tous les sockets connectés

```
socket.on("chat:message", (msg) => {
  io.emit("chat:message", msg)
})
```

Chat de room

- Canal privé `chat:<roomId>`
- Événement : `room:join-chat`, `leave-chat-room`
- Seuls les utilisateurs connectés à une room reçoivent les messages

Morpion temps réel

- Attribution des rôles `X` et `O` lors de la connexion
- Gestion de l'état via Map : `morpionStates`, `roomParticipants`
- Événements :
 - `room:join`
 - `morpion:start`
 - `morpion:move`
- Détection des patterns de victoire ou d'égalité et diffusions

```
socket.on("morpion:move", ({ roomId, index, userId }) => {
  // Vérifications + mise à jour du plateau
  // Changement de joueur ou détection de victoire
  io.to(`room_${roomId}`).emit('morpion:state', state)
})
```

Fonctionnalités WebSocket réalisées

Fonctionnalité	Statut	Détail
Chat global	OK	messages diffusés à tous
Chat par room	OK	canaux <code>chat:<roomId></code>

Attribution rôles dans les rooms	OK	gestion <code>morpionPlayersMap</code>
Synchronisation du jeu morpion	OK	état émis en temps réel à chaque coup
Gestion spectateur si room pleine	OK	<code>spectator: true</code> avec message d'info
Rafraîchissement auto participants	OK	via <code>room:update</code>
Gestion des connexions / déconnexions	OK	suppression des sockets et clean maps

Code et qualité de l'implémentation

- Code entièrement contenu dans `realtime-server/`
- Gestion centralisée des états via `Map` (ex : `gameState`, `morpionStates`, `userSockets`)
- Logs précis des événements sur le serveur pour debug
- Aucune persistance : les états sont en mémoire temporaire
- Le code est commenté, modulaire, et proprement structuré

Exemple de canal de jeu

```
io.on("connection", (socket) => {
  socket.on("room:join", ({ roomId, userId, maxPlayers }) => {
    // Ajout de l'utilisateur dans la room
    // Attribution rôle X/O si disponibles
    // Envoi de l'état courant du jeu
  })
})
```

Comportement utilisateur

- Lorsqu'un utilisateur se connecte :
 - un ID unique de socket est enregistré
 - il rejoint automatiquement les bons canaux (chat, room)
 - il reçoit l'état du jeu ou un message "mode spectateur"
 - Lorsqu'il quitte une room ou se déconnecte :
 - son ID est supprimé
 - les autres utilisateurs sont notifiés via `room:update`
-

Grille d'évaluation : Communication Temps Réel

Catégorie	Critères	Statut
Utilisation de WebSocket	Serveur Socket.IO fonctionnel, multiplexage, événements	✓
Connexions / déconnexions	Gestion de la présence, nettoyage, mise à jour des rooms	✓
Performance / stabilité	Événements clairs, code stable, messages broadcast optimisés	✓
Qualité du code	Maps, événements structurés, journalisation	✓
Documentation	Logs, commentaires, rôles explicites	✓
Fonctionnalités optionnelles	Spectateur, attribution rôles, chat par room, morpion sync	✓

Conclusion

La couche temps réel de Roomies est **robuste, isolée et modulaire** :

- La logique WebSocket est clairement séparée du backend
- Le jeu en temps réel (morpion) est pleinement interactif et synchronisé
- Le chat global et par room est fluide, performant, et bien géré

Partie 5 — Tests d'application

Vue d'ensemble

Le projet **Roomies** intègre une stratégie de test couvrant l'ensemble des couches de l'application :

- **Tests unitaires** sur les entités et composants critiques (Vue et Symfony)
- **Tests fonctionnels** sur les routes REST (Symfony)
- **Tests end-to-end** avec **Cypress** sur les parcours utilisateurs (Vue.js)

Les tests sont organisés, reproductibles, et intégrables dans une chaîne CI/CD via la commande `make test` définie dans le Makefile.

Tests unitaires

Symfony (PHPUnit)

Entité **User**

- Couverture des getters/setters (username, email, avatar...)
- Vérification des rôles par défaut (`ROLE_USER`)
- Initialisation des collections Doctrine (favoris, demandes d'amis)
- Test de la création automatique de `createdAt`
- Validation du comportement de `getUserIdentifier()` (intégration avec Symfony Security)

Entité **Game**

- Tests sur les attributs `name`, `description`, `image`
 - Comportements attendus avec champs vides ou partiels
 - Validation des types de retour (`is_string`, `is_null`, etc.)
-

Vue.js (Vitest)

- `Login.test.ts` : teste le rendu du formulaire, la réactivité aux champs, et la logique de validation
- `useAuthStore.test.ts` : vérifie le comportement du store d'authentification (token, isAuthenticated, logout, parsing JWT)

Ces tests vérifient l'initialisation, les mutations d'état, la présence de rôles et la résilience aux cas limites (token invalide).

Tests fonctionnels Symfony

Les tests fonctionnels sont écrits avec **WebTestCase** (Symfony) et visent à vérifier les réponses HTTP de l'API.

Contrôleurs testés :

- `UserControllerTest`
- `RoomControllerTest`
- `AuthControllerTest`

Exemples de vérifications :

- Statut HTTP attendu (200, 201, 400, 401, 403)
- Structure de la réponse JSON
- Création d'une room en POST (avec ou sans token)
- Connexion utilisateur avec identifiants valides / invalides
- Erreurs de validation lors de l'inscription

Préparation

- Base nettoyée à chaque `setUp()` / `tearDown()`
 - Utilisation de `POST /register` pour créer un utilisateur proprement
 - Authentification simulée pour récupérer un token et le réutiliser
-

Tests End-to-End (E2E)

Framework utilisé : Cypress

Les tests E2E simulent des comportements réels d'utilisateurs dans le navigateur et vérifient les parcours fonctionnels complets.

Tests écrits :

- `auth-connexion.cy.ts` : test de login avec utilisateurs valides / erreurs d'authentification
- `auth-inscription.cy.ts` : test du formulaire d'inscription, validation des erreurs
- `home.cy.ts` : affichage de la home, vérification du chat global et des rooms

- `profile-management.cy.ts` : accès au profil, modification des données, changement d'avatar, gestion des favoris
- `room-management.cy.ts` : création de room, accès au chat de room, vérification du fonctionnement en WebSocket

Points vérifiés :

- Navigation SPA entre les routes Vue.js
- Présence de JWT dans `localStorage`
- Affichage des messages, erreurs serveur, validations frontend
- Fonctionnement WebSocket observé via effets visibles (chat en direct, rafraîchissement des participants)

Intégration et automatisation

- Tous les tests peuvent être exécutés via la commande :

`make test`

- Cela lance les tests Symfony avec PHPUnit
- Les tests Cypress peuvent être lancés via :

`npx cypress run`

Grille d'évaluation : Tests

Catégorie	Critères	✓ Statut
Tests unitaires	Entités Symfony, Stores Pinia, composants Vue	✓
Cas de base/errors	Champs vides, rôles manquants, valeurs invalides	✓
Tests fonctionnels	Contrôleurs Symfony REST : auth, room, user	✓
Tests e2e	Cypress : parcours complets, auth, profil, rooms	✓
Scénarios réalistes	Login → Room → Chat → Favoris → Profil	✓

Conclusion

La stratégie de tests est **complète et alignée avec les critères attendus** :

- Tests unitaires pour garantir la logique métier
- Tests fonctionnels pour valider les flux REST
- Tests e2e pour simuler l'expérience utilisateur

Conclusion Finale

Ce projet de fin d'année, intitulé **Roomies**, a constitué bien plus qu'un simple exercice technique : il a été l'aboutissement d'un parcours d'apprentissage approfondi, mêlant conception logicielle, intégration temps réel, développement fullstack moderne et bonnes pratiques de l'ingénierie logicielle.

L'objectif initial était clair : créer une plateforme de jeux en ligne multijoueur, permettant aux utilisateurs de jouer, discuter, créer des rooms, gérer leur profil, et vivre une expérience interactive fluide. Pour cela, j'ai choisi d'architecturer le projet autour de technologies puissantes et complémentaires : **Vue.js 3** côté frontend pour sa flexibilité et sa réactivité, **Symfony 7** côté backend pour la robustesse de son écosystème PHP, et **Socket.IO** pour la communication en temps réel.

Chaque couche du projet a été pensée avec rigueur :

- Le **frontend** offre une interface moderne, responsive et modulaire, construite avec PrimeVue et organisée selon les principes de composants réutilisables.
- Le **backend** repose sur une API REST sécurisée, où chaque entité est définie avec précision, chaque route protégée par JWT, et chaque action testée pour assurer la stabilité de l'ensemble.
- Le **serveur WebSocket**, entièrement indépendant, gère les échanges en temps réel (chat, jeux) avec efficacité et scalabilité.

Au-delà des fonctionnalités attendues, j'ai choisi d'aller plus loin : système de favoris, personnalisation du profil, gestion d'amis (en cours), amélioration du chat global, historique local et nettoyage automatique des messages. Ces ajouts témoignent de ma volonté de livrer une application complète, pensée pour l'usage, et non uniquement pour la démonstration technique.

Un soin particulier a également été apporté à la **qualité logicielle**. L'ensemble est testé via **PHPUnit**, **Vitest**, et **Cypress**, garantissant ainsi la cohérence des comportements côté serveur, client et au niveau de l'expérience utilisateur. Le projet est conteneurisé via **Docker**, ce qui facilite son déploiement, sa maintenance et sa reproductibilité.

Ce travail m'a permis de mobiliser l'ensemble des compétences acquises au cours de mon Master, tout en me confrontant à des défis concrets : la synchronisation en temps réel, la gestion d'état distribué, la sécurité API, ou encore l'interopérabilité des services. Il a aussi renforcé ma capacité à concevoir une architecture modulaire, évolutive et maintenable.

En somme, **Roomies incarne ma vision d'un projet web complet, professionnel, fonctionnel et élégant (après chacun ses goûts)**. Il reflète non seulement mes compétences techniques, mais aussi ma capacité à penser un produit dans son ensemble, à anticiper ses usages, et à y répondre avec clarté, méthode et créativité.

Ce projet marque la fin d'un cycle de formation, mais je le considère aussi comme un **point de départ** vers de futures évolutions : ajout de nouveaux jeux, système de classement, persistance des messages, voire intégration de WebRTC pour des échanges audio ou vidéo.

Je suis fier du chemin parcouru, du projet livré, et de la maturité technique qu'il illustre. Roomies est aujourd'hui une plateforme fonctionnelle, mais aussi une preuve concrète de mes compétences en tant que développeur fullstack, et de mon engagement à livrer des projets solides, bien construits et pensés pour l'utilisateur.