# Planning Poker


Université de Technologie

Département d'Informatique


Projet de Gestion de Projet


Réalisé par :

Maxime Coux

Antoine Vuillet

Steven Grenier

Enseignant:

Valentin Lachand-Pascal


Date :

20 décembre 2024

# Sommaire:

# Introduction:

Dans le cadre des projets Agile, le Planning Poker est une méthode collaborative et efficace pour estimer l'effort nécessaire à la réalisation des tâches d'un projet. Cette étape est essentielle pour garantir une planification précise et réaliste des sprints.

Notre projet visait à concevoir et développer un outil en ligne qui soit à la fois intuitif, interactif et performant, permettant à des équipes distribuées de collaborer en temps réel pour estimer leurs tâches. L'objectif principal était de renforcer la communication, l'engagement et la fluidité des échanges entre les membres, même à distance.

Pour atteindre ces objectifs, nous avons fait des choix stratégiques tant sur le plan technique que méthodologique. L'application repose sur une architecture moderne intégrant un backend robuste pour gérer les sessions en temps réel, un frontend ergonomique pour offrir une expérience utilisateur optimisée, et l'utilisation de WebSocket pour assurer une synchronisation rapide et fiable entre les participants. Par ailleurs, nous avons mis en place un pipeline d'intégration continue, garantissant la qualité, la stabilité et l'évolutivité du projet à chaque étape de son développement.

# Choix techniques et fonctionnels:

Pour le développement de l'application Planning Poker, le choix des technologies s'est appuyé sur une analyse qualitative des besoins, tant fonctionnels que techniques :

### Besoins fonctionnels:

-Interface utilisateur intuitive et responsive : Une interface claire et ergonomique est essentielle pour offrir une expérience fluide, notamment pour des équipes réparties sur différents dispositifs (ordinateurs, tablettes, smartphones).

-Interaction en temps réel : Les estimations doivent être synchronisées instantanément entre les participants, rendant la communication fluide et interactive.

-Accessibilité et simplicité de déploiement : L'application doit être accessible via un navigateur web sans nécessiter d'installation locale.

### Critères de choix technologiques:

-Nous avons choisi HTML et CSS pour le front-end car ces technologies standards du web sont universelles et permettent de concevoir des interfaces responsives adaptées à tous les écrans. En plus de cela, leur simplicité d'utilisation et la disponibilité d'outils modernes (frameworks CSS, bibliothèques graphiques) facilitent le développement rapide d'un design attractif.

-Pour le back-end, nous avons choisi JavaScript (Node.js) car cet outil est particulièrement adapté pour gérer des applications en temps réel grâce à son modèle événementiel.

-Il offre une large gamme de bibliothèques (comme Socket.io dont nous nous sommes servi pour le temps réel) qui simplifient la mise en œuvre de fonctionnalités.

-L'utilisation de JavaScript pour le back-end et le front-end assure une homogénéité dans code, facilitant la collaboration et le partage de connaissances entre développeurs.

-Les framework de tests choisis sont Jest et Cypress pour les tests unitaires et fonctionnels respectivement. Ils sont tous les deux fiables et efficaces, et maîtrisés par le membre du groupe écrivant les tests.

-JsDoc a été choisi pour générer la documentation du code. Etant donné qu'aucun membre du groupe n'avait d'expérience dans la génération automatique de documentation en Javascript, l'exemple préconisé par le cours a été choisi.

Compétences des développeurs:

-Les technologies susnommées étaient maîtrisées par tous les développeurs de l'équipe ce qui nous permet de réduire le temps d'apprentissage et de le concentrer sur les technologies que nous ne maîtrisons pas (WebSocket).

## Conception:

Notre code est séparé en deux parties majeures, le client et le serveur.
-Le client reçoit toutes les informations et actions de l'utilisateur et les transmet au serveur. Qu'il s'agisse du choix d'une estimation (carte) ou de la création d'un salon de vote, le client envoie un événement à l'aide des sockets contenant toutes les informations nécessaires au serveur. Le client gère aussi la réception en temps réel des réponses du serveur ce qui permet que tous les utilisateurs voient l'actualisation en temps réel.

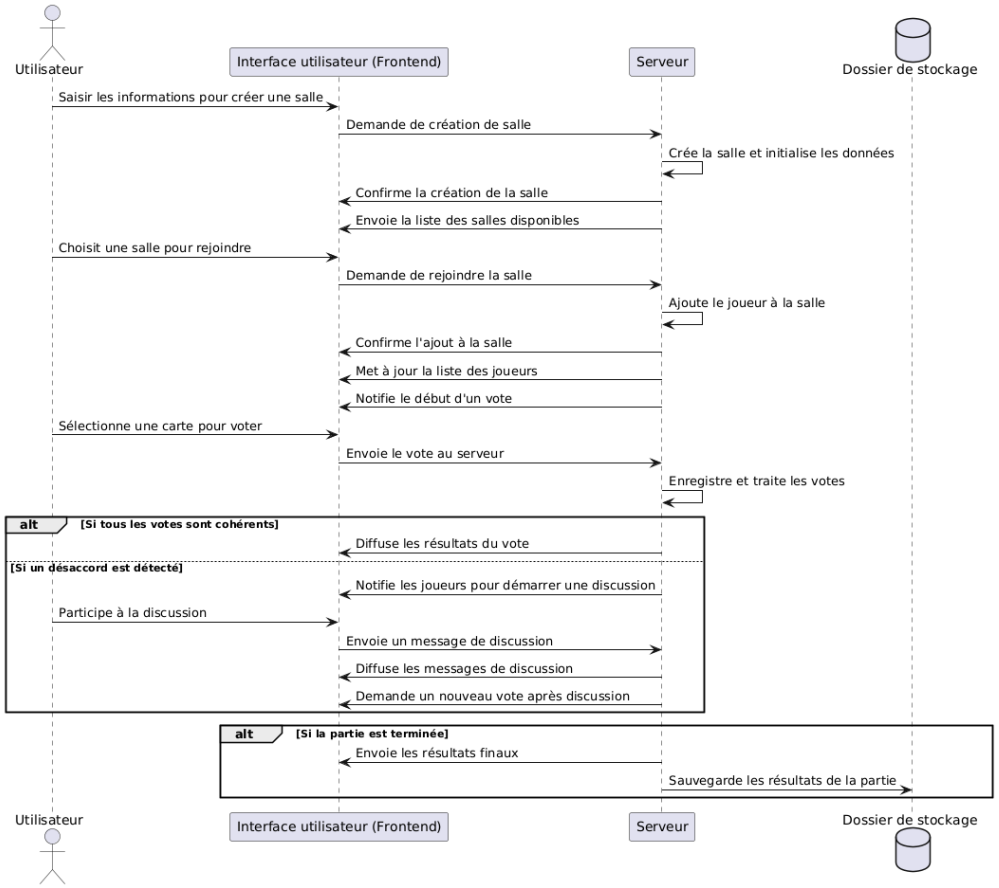
L'interface utilisateur a été conçue pour être intuitive et réactive, garantissant une expérience fluide quel que soit le terminal utilisé.

-Le serveur lui effectue les opérations nécessaires au bon fonctionnement de l'application. Il reçoit les événements créés par les clients et les traite directement, stockant les informations sur les différents utilisateurs et salon créés. Il gère aussi la synchronisation en temps réel de tous les utilisateurs.

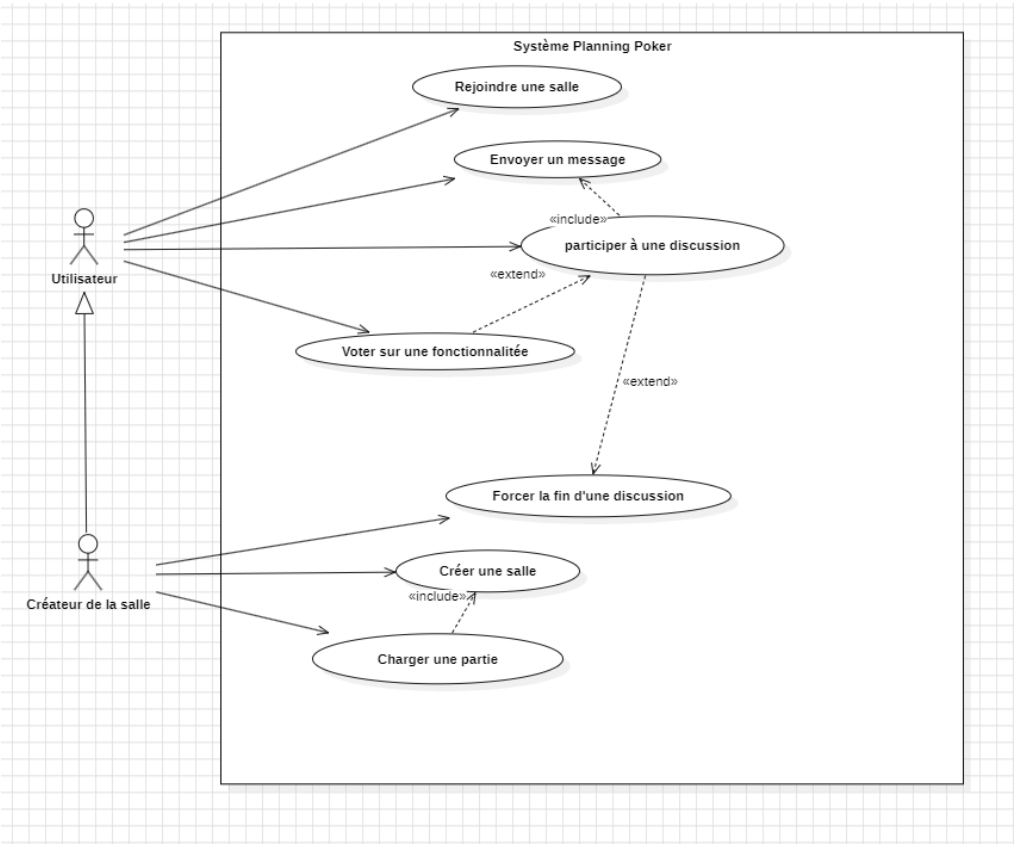
Le serveur, utilisant le système de websocket assure ainsi un échange en temps réel des données avec une très faible latence qui garantit une expérience utilisateur fluide et réactive.

Cette conception en deux parties distinctes mais interconnectées offre une architecture simple, robuste et adaptée aux besoins de l'application. Le découpage clair entre les responsabilités du client et du serveur facilite non seulement le développement, mais également l'évolution future du projet.

## Diagramme de séquence:




## Diagramme de cas d'utilisation:



# Mise en place de l'intégration continue :

## Tests unitaires :

Comme précisé dans les choix techniques, le framework de tests unitaires utilisé est Jest.
La toute première étape de la mise en place de l'intégration continue est tout d'abord l'installation du package avec npm, puis la création du fichier de test unitaire servant à tester leur exécution :


Puis l'ajout d'un script NPM dans le projet afin de lancer les test et de récupérer la couverture de code :

```
"scripts": {
  "test": "jest --coverage",
```

Ces lignes ont été ajoutés au fichier package.json
Cela permet d'exécuter les tests unitaires comme suit :

```
PS C:\Users\Maxime Coux\source\repos\CAPIMAS> npm run test

> test
> jest --coverage

 PASS  tests/test.js
  √ Le fichier de test est executé (3 ms)

-------------------------|---------|----------|---------|---------|-------------------
File                     | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------------------|---------|----------|---------|---------|-------------------
All files                |       0 |        0 |       0 |       0 |
 CAPIMAS                 |       0 |        0 |       0 |       0 |
  cypress.config.js      |       0 |      100 |       0 |       0 | 1-3
  jest.config.js         |       0 |      100 |     100 |       0 | 1
  server.js              |       0 |        0 |       0 |       0 | 2-544
 CAPIMAS/cypress/e2e     |       0 |      100 |       0 |       0 |
  create_room.cy.js      |       0 |      100 |       0 |       0 | 1-43
  pause_room.cy.js       |       0 |      100 |       0 |       0 | 2-16
 CAPIMAS/cypress/support |       0 |        0 |       0 |       0 |
  commands.js            |       0 |        0 |       0 |       0 |
  e2e.js                 |       0 |        0 |       0 |       0 |
 CAPIMAS/public          |       0 |        0 |       0 |       0 |
  script.js              |       0 |        0 |       0 |       0 | 2-332
-------------------------|---------|----------|---------|---------|-------------------

============================== Coverage summary ==============================
Statements   : 0% ( 0/508 )
Branches     : 0% ( 0/127 )
Functions    : 0% ( 0/83 )
Lines        : 0% ( 0/484 )
==============================================================================
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        2.283 s
Ran all test suites.
PS C:\Users\Maxime Coux\source\repos\CAPIMAS>
```

La deuxième étape est ensuite de créer une Github Action qui va :

- s'exécuter uniquement sur la branche main, en cas de push ou de pull request
- installer tous les packages Node nécessaire au bon fonctionnement de l'application
- exécuter les tests unitaires

```yaml
# This workflow will do a clean installation of node dependencies, cache/restore them, build t
# For more information see: https://docs.github.com/en/actions/automating-builds-and-tests/bui

name: Node.js CI

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  build:

    runs-on: ubuntu-latest

    strategy:
      matrix:
        node-version: [18.x, 20.x, 22.x]
        # See supported Node.js release schedule at https://nodejs.org/en/about/releases/

    steps:
    - uses: actions/checkout@v4
    - name: Use Node.js ${{ matrix.node-version }}
      uses: actions/setup-node@v4
      with:
        node-version: ${{ matrix.node-version }}
        cache: 'npm'
    - run: npm install
    - run: npm test
```

Code du fichier “unit-tests.yml”

Le but était ensuite d’écrire les tests unitaires au fur et à mesure de l’avancement du projet. Cependant, nous avons été très vite limité par la structure du code du projet. En effet, le code Javascript du projet est contenu dans deux fichiers :

- server.js : code du back-end
- script.js : code du front-end

Un problème (pour les tests) commun entre les deux fichiers, est que la plupart des fonctions sont définies directement dans les sockets :

```
// Gestion de la réception des messages de chat
socket.on('sendMessage',(data)=>{
    const { roomName, message } = data;
    const room = rooms[roomName];
    if (!room) return;
    const player = room.players[socket.id];
    if (!player) return;

    // Si on est en phase de discussion, vérifier si le joueur est autorisé à envoyer des messages
    if (room.state === 'discussion') {
        if (!room.extremes.includes(socket.id)) {
            // Le joueur n'est pas autorisé à discuter
            socket.emit('error', 'Vous ne pouvez pas envoyer de messages pendant la discussion.');
            return;
        }
    }

    io.to(roomName).emit('receiveMessage', { username: player.username, message });
});
```

Les définir autre part empêche le code de bien fonctionner. De ce fait, il est impossible de les importer dans un fichier de tests. Et même si cela était possible, l’utilisation des packages socket.io et express dans les deux fichiers fait que le fichier importé ne peut bien s'exécuter dans l’environnement de tests unitaires.
Une solution testé sur les fonctions testables était de les définir dans un fichier à part, et de faire en sorte que le fichier de code utilisant la fonction et le fichier de tests unitaires important tous les deux la fonction, qui peut s'exécuter dans les deux environnement, et peut donc être testée. Cependant, dans ce cas de figure, nous ne sommes pas arrivés à faire en sorte de faire marcher cette solution :

- Soit la fonction était bien importer dans le code, mais l’import ne marchait pas dans les tests
- Soit était bien testé, mais impossible de l’exporter vers le code qui en avait besoin.

Ces contraintes ont fait en sorte que l’écriture d’autres tests unitaires fut abandonnée, au profit des tests fonctionnels avec Cypress.

### Tests Fonctionnels :

Afin de pallier le manque de tests unitaire, des tests fonctionnels ont été créés afin d’effectuer un contrôle qualité sur l’application. Ce type de test vérifie le bon fonctionnement de l’application en effectuant diverses actions sur son interface (clics de boutons, entré de valeurs dans des champs…) et en vérifiant que l’interface affiche les bonnes données. L’installation du framework se fait avec npm, comme pour Jest. Ensuite, deux tests ont été écrits.Tout d’abord, un test vérifiant la création d’une salle à un joueur, puis le bon fonctionnement du vote pour chaque tâche. Le deuxième vérifie le bon fonctionnement de la mise en pause d’une salle.

```
//Test de la création d'une salle et du vote de chaque fonctionnalitée
describe("Création d'une salle et vote des fonctionnalitées", () => {
  it('passes', () => {
    cy.visit('http://localhost:3000');
    cy.get('#roomNameInput').type("test");
    cy.get('#maxPlayersInput').type("1");
    cy.get('#usernameInput').type("test1");
    cy.get('#backlogInput').selectFile("./cypress/e2e/test_configs/backlog-test.json");
    cy.get('#createRoomBtn').click();
    cy.get('#roomNameDisplay').contains("test");
    for(let i=0; i<cards_values.length;i++){
      cy.get('.cardBtn:nth-child('+Number(i+1)+')').contains(cards_values[i]);
    }

    for(let i = 0;i<10;i++){
      cy.get('.cardBtn:nth-child('+Number(i+1)+')').click();
      cy.get('#swal2-html-container').contains('La fonctionnalité "Task'+Number(i+1)+'" a été estimée à '+cards_values[i]+'.');
      cy.get('.swal2-confirm').click();
    }
    cy.get('.cardBtn:nth-child(1)').click();
    cy.get('#swal2-html-container').contains('Le backlog a été entièrement estimé. Les résultats ont été sauvegardés.');
    cy.get('.swal2-confirm').click();
  })
})
```

```
//Test de la création d'une salle et de l'activation de la pause
describe("Création d'une salle et vote des fonctionnalitées", () => {
    it('passes', () => {
      cy.visit('http://localhost:3000');
      cy.get('#roomNameInput').type("test");
      cy.get('#maxPlayersInput').type("1");
      cy.get('#usernameInput').type("test1");
      cy.get('#backlogInput').selectFile("./cypress/e2e/test_configs/backlog-test.json");
      cy.get('#createRoomBtn').click();
      cy.get('#roomNameDisplay').contains("test");
      cy.get('.cardBtn:nth-child(11)').click();
      cy.get('#swal2-html-container').contains('Tous les joueurs ont choisi "Café". La partie est sauvegardée.');

      cy.get('.swal2-confirm').click();
      for(let i=0; i<11;i++){
        cy.get('.cardBtn:nth-child('+Number(i+1)+')').should('be.disabled');
      }
    })
  })
```

Une fois les tests ajoutés au dépôt en ligne, une Github Action a été créée, afin d'exécuter automatiquement les tests fonctionnels à chaque push ou pull request sur la branche main :

```
name: Cypress Tests

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  cypress-run:
    runs-on: ubuntu-24.04
    steps:
      - name: Checkout
        uses: actions/checkout@v4
      # Install npm dependencies, cache them correctly
      # and run all Cypress tests
      - name: Cypress run
        uses: cypress-io/github-action@v6
        with:
          build: npm install
          start: npm start
```

### Génération de la documentation :

Afin d'utiliser JSDoc pour générer la documentation, il faut annoter le code JS avec des commentaires spéciaux pour que la documentation du code soit correcte. Les différentes fonctions de script.js et server.js (les seuls fichiers de code js du projet) ont donc été tous deux annotés avec JSDoc :

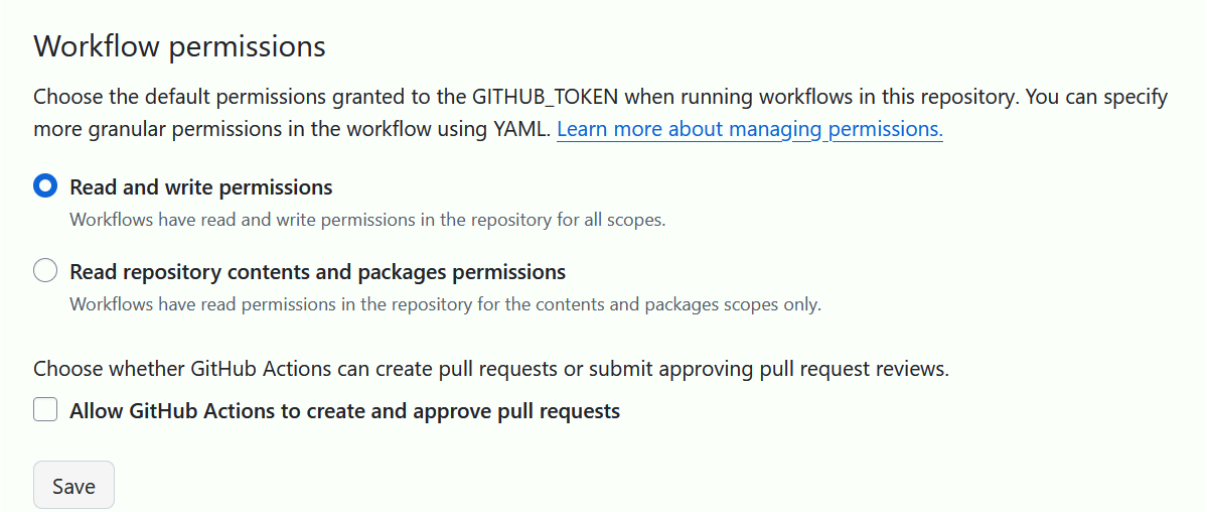
```
/**
 * Gère les résultats du vote d'une salle donnée
 * @param {string} roomName Le nom de la salle
 * @returns Le résultat du vote
 */
// Fonction pour gérer le résultat du vote
function handleVotingResult(roomName) {
```

```
/**
 * Formate le temps donné en paramètre
 * @param {Time} time Le temps à formater
 * @returns Le temps rentré en paramètre, formaté en mm:ss
 */
function formatTime(time) {
    const minutes = Math.floor(time / 60);
    const seconds = time % 60;
    return `${minutes}:${seconds.toString().padStart(2, '0')}`;
}
```

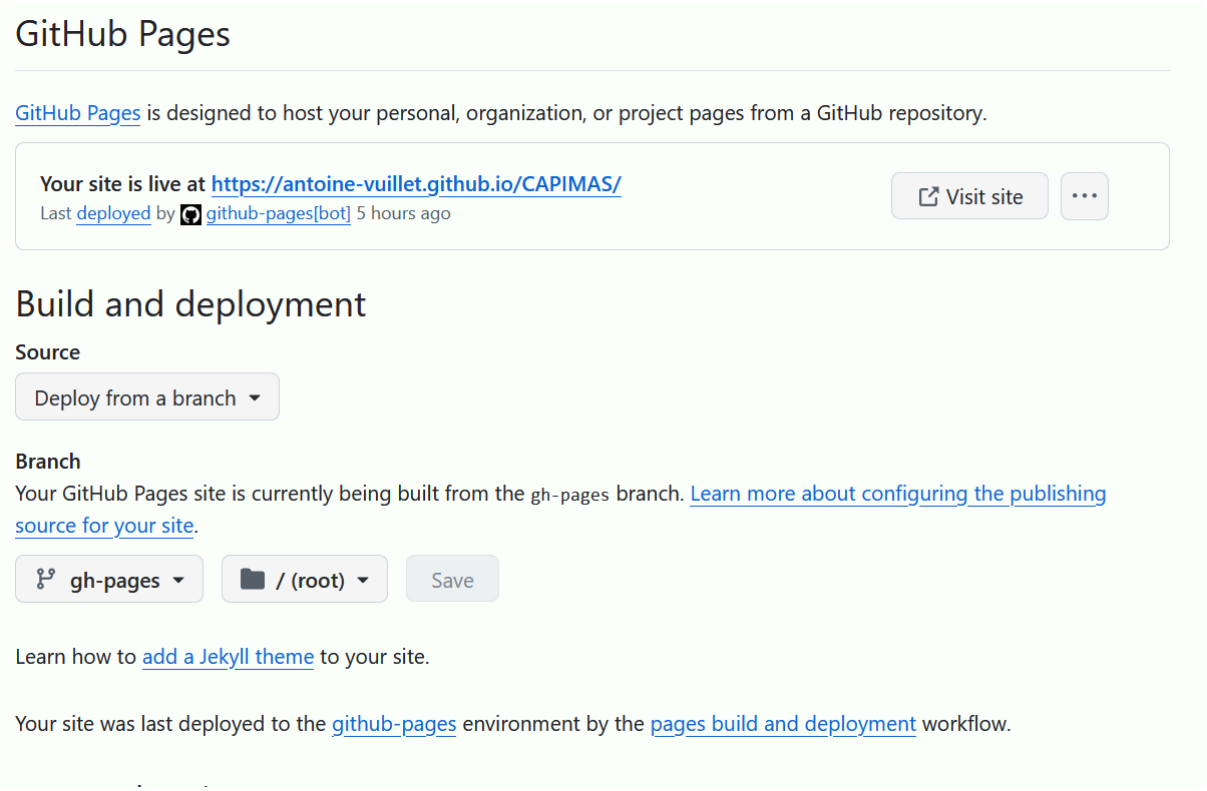
Un Github action a ensuite été mis en place afin d'automatiser la création de la documentation, puis de pouvoir y accéder depuis la Github Page du projet :

```
name: Generate JSDoc Documentation

on:
  push:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout repository
      uses: actions/checkout@v4

    - name: Setup Node
      uses: actions/setup-node@v4

    - name : Install JsDoc
      run : npm install -g jsdoc

    - name: Generate Documentation
      run: jsdoc *.js public/*.js

    - name: Deploy Documentation

      uses: peaceiris/actions-gh-pages@v3
      with:
        github_token: ${{ secrets.GITHUB_TOKEN }}
        publish_dir: ./out
```

Après cela, nous avons modifié le dépot GitHub pour lui donner les droits d'écriture et de lecture, lui permettant d'écrire la documentation.




Et enfin, nous avons paramétré GitHub pages pour lui permettre de stocker la documentation sur une adresse pages.




Permettant d'avoir dans la situation actuelle, notre documentation stockée à https://antoine-vuillet.github.io/CAPIMAS/ .

# Conclusion:

Le développement de ce projet nous a permis de mettre en pratique les concepts de la gestion de projet agile que nous avons étudiés. À travers ce projet, nous avons relevé plusieurs défis, comme l'implémentation d'interaction en temps réel et la création d'une pipeline d'intégration continue, qui nous ont permis de livrer un produit efficace, robuste, et vérifiable.

L'application que nous avons conçue et développée répond aux besoins identifiés : elle offre une interface intuitive et ergonomique, permet une collaboration fluide grâce à l'utilisation de WebSocket et intègre des outils de test et de documentation pour assurer sa pérennité. Malgré les contraintes techniques rencontrées, notamment pour l'écriture de tests unitaires, nous avons su nous adapter en misant sur des tests fonctionnels pour garantir le bon fonctionnement de l'application.

Notre application possède encore des pistes d'amélioration à envisager pour renforcer la qualité et l'utilisabilité de notre application, notamment réussir à implémenter des tests unitaires en rendant le code plus modulaire faciliterai l'écriture de tests et augmenterai la couverture du code et la fiabilité générale de l'application. Mais aussi améliorer l'interface utilisateur. Nous nous sommes concentré sur la lisibilité et la compréhension facile de l'application, mais elle gagnerait à être plus esthétique ou engageante avec des transitions ou animations gardant l'utilisateur investi.