Rapport de projet CAPI - Zyad NAISSALI / Jérémy PELLISSIER

https://github.com/Zzzyad/Projet-CAPI.git

1. Introduction

1.1 Présentation du projet

Le projet consiste en la création d'une application de planning poker, un outil utilisé pour estimer la complexité des tâches dans les projets de développement logiciel. Cette application permet à des joueurs (les membres d'une équipe) de collaborer pour attribuer des estimations aux tâches du backlog via un système de votes. Le projet comprend quatre modes de jeu principaux :

- **Mode strict** : Validation nécessitant l'unanimité.
- **Mode moyenne** : Dans ce mode, la moyenne des votes est calculée pour déterminer l'estimation de la fonctionnalité.
- **Mode médiane** : Dans ce mode, la médiane des votes est calculée pour valider l'estimation, offrant ainsi une approche qui minimise l'impact des votes extrêmes.
- **Mode majorité absolue** : Une fonctionnalité est validée si la majorité absolue des joueurs (plus de 50%) sont d'accord sur une estimation, indépendamment des autres votes.

---

2. Architecture et choix technologiques

2.1 Technologies utilisées

Initialement, nous avions prévu de développer le projet avec Python et Flask, mais après quelques essais, nous avons rencontré des difficultés à mettre en place un environnement complètement fonctionnel. Pour garantir une mise en œuvre rapide et simplifiée, nous avons choisi de nous orienter vers **HTML**, **CSS**, et **JavaScript**. Cette décision repose sur l'accessibilité de ces technologies, la facilité de leur écosystème et leur capacité à s'adapter efficacement aux projets légers.

Langages et bibliothèques

- **HTML/CSS** : Pour structurer et styliser l'interface utilisateur.
- **JavaScript** : Pour gérer la logique applicative et les interactions utilisateurs.

Outils et environnements

- **GitHub Actions** : Automatisation des tests, intégration continue et déploiement.
- **JSDoc** : Génération de documentation détaillée basée sur les commentaires dans le code.
- **Jest** : Tests unitaires pour garantir le bon fonctionnement des fonctionnalités.

2.2 Organisation du code

L'application suit une architecture modulaire permettant une maintenance et une extension facilitées. Voici une explication des principaux fichiers et classes utilisées :

```
.github/workflows/      # Fichiers pour GitHub Actions
  Documentation_javascript.yml  # Workflow pour la documentation
  TU_javascript.yml            # Workflow pour les tests unitaires
assets/                 # Ressources graphiques
  cartes_0.svg, cartes_cafe.svg, etc. # Images des cartes
data/                   # Données JSON
  backlog.json          # Données du backlog
node_modules/           # Modules Node.js pour les dépendances
out/                    # Fichiers générés pour la documentation
  fonts/                # Polices
  scripts/              # Scripts compilés
  styles/               # Styles CSS
app.js                  # Logique principale du jeu
index.html              # Interface utilisateur
style.css               # Feuille de style
jest.config.js          # Configuration des tests unitaires
package.json            # Gestion des dépendances
package-lock.json       # Verrouillage des versions des dépendances
testunitaire.test.js    # Tests unitaires
```

Explication des fichiers principaux

- `index.html` : Structure de la page web contenant les éléments interactifs.
- `style.css` : Définit l'apparence et le design global de l'application.
- `app.js` : Contient toute la logique applicative pour le système de planning poker.
- `testunitaire.test.js` : Inclut les tests Jest pour valider les fonctionnalités de l'application.

Principales Fonctions JavaScript

```javascript
function addPlayers() {
 // Ajouter des joueurs au jeu en fonction du nombre saisi.
}

function loadBacklog(event) {
 // Charger un fichier JSON contenant un backlog.
}

function loadSave(event) {
 // Charger un fichier JSON contenant une sauvegarde.
}

function startGame() {
 // Démarrer la partie en initialisant les joueurs et en affichant le jeu.
}
```

```
function showCardsForPlayer() {
 // Afficher les cartes pour le joueur actuel et permettre le vote.
}

function voteForCurrentPlayer(card) {
 // Enregistrer le vote pour le joueur actuel et passer au joueur suivant.
}

function showFeature() {
 // Afficher la fonctionnalité actuelle pour le vote.
}

function showPreviousFeature() {
 // Afficher la fonctionnalité précédente.
}

function addPreviousButton() {
 // Ajouter un bouton pour revenir à la fonctionnalité précédente.
}

function disableCards() {
 // Désactiver les cartes après la fin du jeu.
}

function startTimer() {
 // Démarrer un chronomètre pour le jeu.
}

function validateVote() {
 // Valider le vote en fonction des règles sélectionnées (unanimité, moyenn
e, médiane, majorité absolue).
}

function resetVotesAndPlayers() {
 // Réinitialiser les votes et les joueurs pour une nouvelle fonctionnalit
é.
}

function saveResults() {
 // Sauvegarder les résultats finaux dans un fichier JSON.
}

function saveProgress() {
 // Sauvegarder la progression du jeu dans un fichier JSON.
}

function showReturnToMenuButton() {
```

```
 // Afficher un bouton pour retourner au menu principal.
}

function resetGame() {
 // Réinitialiser le jeu et revenir au menu principal.
}
```

2.3 Gestion des fichiers JSON

- Les fonctionnalités sont chargées depuis un fichier JSON au format personnalisé pour constituer le backlog initial.
- Un fichier JSON final est exporté à la fin de la partie, contenant les estimations validées.

Exemple de backlog JSON

```
[
    { "name": "Créer un formulaire d'inscription" },
    { "name": "Mettre en place une API REST" },
    { "name": "Optimiser la base de données" },
    { "name": "Implémenter une authentification par OAuth" },
    { "name": "Ajouter un tableau de bord analytique" },
    { "name": "Effectuer des tests de charge sur le serveur" }
]
```

Exemple de Sauvegarde JSON

Si tous les joueurs votent pour la **carte café**, la partie est suspendue et son état actuel est sauvegardé, permettant une reprise ultérieure.

```
{
  backlog: [
    { name: "Créer un formulaire d'inscription", estimate: 30 },
    { name: "Mettre en place une API REST", estimate: 70 },
    { name: "Optimiser la base de données", estimate: 4 },
    { name: "Implémenter une authentification par OAuth" },
    { name: "Ajouter un tableau de bord analytique" },
    { name: "Effectuer des tests de charge sur le serveur" }
  ],
  currentFeatureIndex: 3,
  players: ["QQq", "qQQQ"],
  votes: [null, null],
  rule: "average"
}
```

Exemple de Resultat JSON

Une fois que toutes les fonctionnalités ont été entièrement votées, l'application génère un fichier `results.json` contenant les résultats des votes

```
[
  { name: "Créer un formulaire d'inscription", estimate: 30 },
  { name: "Mettre en place une API REST", estimate: 70 },
  { name: "Optimiser la base de données", estimate: 4 },
  { name: "Implémenter une authentification par OAuth", estimate: 30 },
  { name: "Ajouter un tableau de bord analytique", estimate: 70 },
  { name: "Effectuer des tests de charge sur le serveur", estimate: 3 }
]
```

---

3. Intégration continue et automatisation

3.1 Pipelines CI/CD

Un pipeline d'intégration continue a été mis en place avec **GitHub Actions**, permettant de :

- Exécuter automatiquement les tests unitaires avec Jest lors de chaque push.
- Générer et déployer la documentation JSDoc sur GitHub Pages.

Prérequis en local :

- **Node.js** : Assurez-vous que Node.js est installé sur votre machine ou votre serveur.
- **Gestionnaire de paquets : npm** : npm est utilisé pour gérer les dépendances du projet.

Étapes :

1. **Installer les dépendances** :
   - Exécutez la commande suivante pour installer toutes les dépendances nécessaires définies dans le fichier `package.json` :

     ```
     npm install
     ```

2. **Lancer les tests** :
   - Pour exécuter les tests unitaires et vérifier le bon fonctionnement du code, utilisez cette commande :

     ```
     npm run test
     ```

3. **Documentation** :
   - La documentation est générée automatiquement grâce à **JSDoc**. Pour la générer, suivez ces étapes :
     - Installez JSDoc :

```
npm install -g jsdoc
```

- Générez la documentation avec la commande suivante :

```
jsdoc app.js
```

Après avoir configuré le setup local, voici les deux fichiers de configuration **YAML** nécessaires pour GitHub Actions afin d'automatiser l'intégration continue (CI) et le déploiement de la documentation (CD).

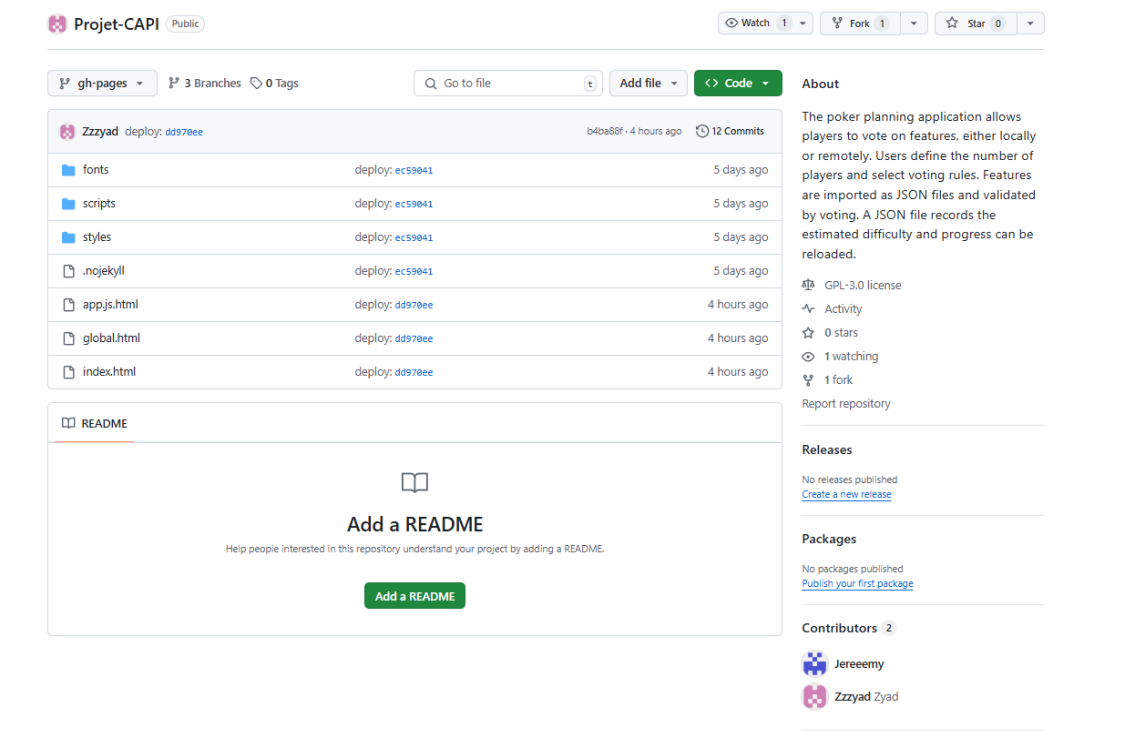
Workflow Documentation CI/CD GitHub Actions :

Ce workflow génère automatiquement la documentation à chaque fois qu'il y a un push sur la branche `main`.

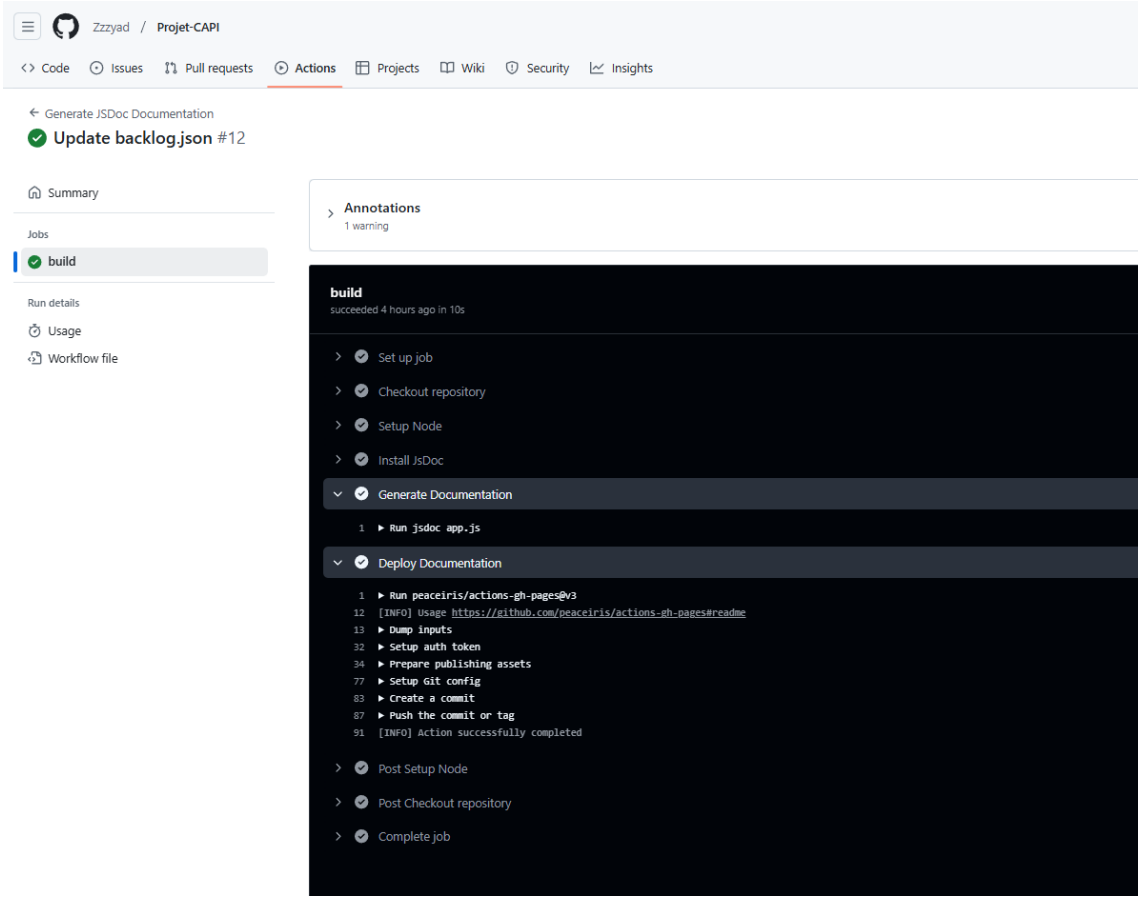
```yaml
name: Generate JSDoc Documentation

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 app.js

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

Ce fichier installe JSDoc et génère la documentation à partir du fichier `app.js.`

Une fois la documentation déployée, elle est accessible via la branche `gh-pages`, qui contient le code généré de la documentation, permettant ainsi aux utilisateurs de consulter la documentation directement à partir de GitHub Pages.



On peut également observer dans GitHub Actions la génération automatique de la documentation, qui s'effectue à chaque push sur la branche principale.



Workflow Tests Unitaires CI/CD GitHub Actions :

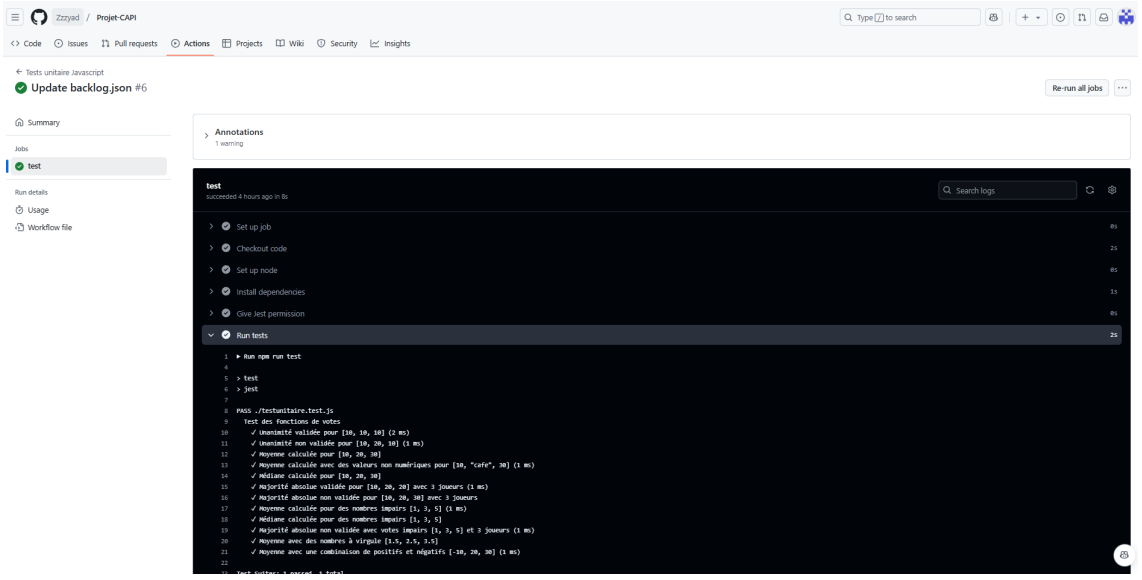
Ce workflow exécute les tests unitaires à chaque fois qu'il y a un push ou une pull request sur la branche `main`.

```
name: Tests unitaires Javascript

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v4
```

```
    - name: Set up node
      uses: actions/setup-node@v4

    - name: Install dependencies
      run: npm install

    - name: Give Jest permission
      run: chmod +x node_modules/.bin/jest

    - name: Run tests
      run: npm run test
```

Ce fichier installe les dépendances, donne les permissions nécessaires à Jest et lance les tests unitaires en exécutant `npm run test`.

Une fois le code poussé et déployé, les résultats des tests effectués seront disponibles sur GitHub Actions, permettant de consulter directement les rapports de tests réalisés.



3.2 Tests unitaires

Nous avons réalisé des tests généraux sur des fonctions basiques telles que le calcul de la moyenne, de la médiane, de l'unanimité et de la majorité absolue pour vérifier leur bon fonctionnement dans des situations standards. En parallèle, j'ai testé **des cas plus aberrants**, comme des valeurs négatives, des nombres à virgule, afin de voir comment le code réagit face à des entrées inhabituelles ou invalides. Cela permet de s'assurer que les fonctions gèrent correctement ces cas limites sans provoquer d'erreurs ou de comportements inattendus.

Les tests unitaires sont écrits avec **Jest dans le fichier testunitaire.test.js** :

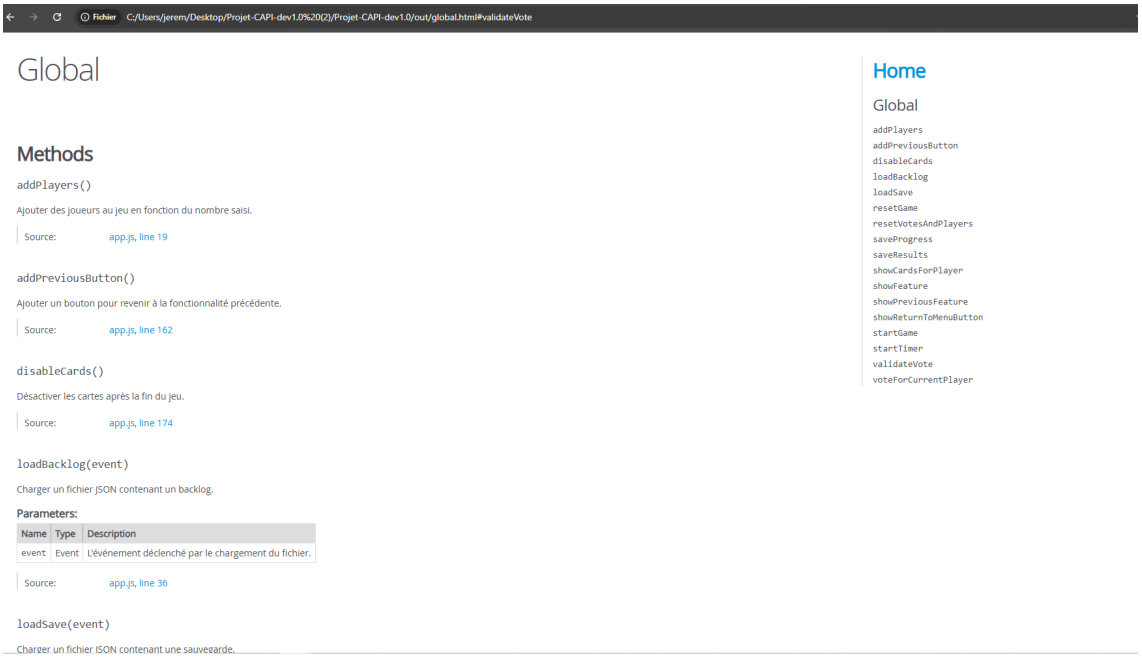
```
// testunitaire.test.js

function testUnanimite(votes) {
```

```
    // Test de l'unanimité lorsque tous les votes sont identiques.
}

function testMoyenne(votes) {
    // Test du calcul de la moyenne pour une série de votes numériques.
}

function testMediane(votes) {
    // Test du calcul de la médiane pour une série de votes numériques.
}

function testMajoriteAbsolue(votes, playerCount) {
    // Test de la majorité absolue lorsque l'une des cartes est choisie par
    //plus de la moitié des joueurs.
}

test('Unanimité validée pour [10, 10, 10]', () => {
    // Vérifie que l'unanimité est validée lorsque tous les votes sont
    //identiques (ici 10).
});

test('Unanimité non validée pour [10, 20, 10]', () => {
    // Vérifie que l'unanimité n'est pas validée lorsque les votes
    //sont différents (10, 20, 10).
});

test('Moyenne calculée pour [10, 20, 30]', () => {
    // Vérifie que la moyenne est correctement calculée pour des
    //votes numériques (10, 20, 30).
});

test('Moyenne calculée avec des valeurs non numériques pour [10, "cafe", 30]'
    // Vérifie que la moyenne est correctement calculée malgré la
    //présence d'une valeur non numérique (ici "cafe").
});

test('Médiane calculée pour [10, 20, 30]', () => {
    // Vérifie que la médiane est correctement calculée pour des
    //votes numériques (10, 20, 30).
});

test('Majorité absolue validée pour [10, 20, 20] avec 3 joueurs', () => {
    // Vérifie que la majorité absolue est validée lorsqu'une carte
    //est choisie par plus de la moitié des joueurs.
});

test('Majorité absolue non validée pour [10, 20, 30] avec 3 joueurs', () => {
    // Vérifie que la majorité absolue n'est pas validée si aucune
```

```
    //carte n'est choisie par plus de la moitié des joueurs.
});

test('Moyenne calculée pour des nombres impairs [1, 3, 5]', () => {
    // Vérifie que la moyenne est correctement calculée pour des
    //nombres impairs (1, 3, 5).
});

test('Médiane calculée pour des nombres impairs [1, 3, 5]', () => {
    // Vérifie que la médiane est correctement calculée pour des
    //nombres impairs (1, 3, 5).
});

test('Majorité absolue non validée avec votes impairs [1, 3, 5] et 3 joueurs'
    // Vérifie que la majorité absolue n'est pas validée lorsque les
    //votes sont impairs et qu'aucune carte n'a plus de la moitié des voix.
});

test('Moyenne avec des nombres à virgule [1.5, 2.5, 3.5]', () => {
    // Vérifie que la moyenne est correctement calculée pour des
    //nombres à virgule (1.5, 2.5, 3.5).
});

test('Moyenne avec une combinaison de positifs et négatifs [-10, 20, 30]', ()
    // Vérifie que la moyenne est correctement calculée pour une
    //combinaison de votes positifs et négatifs (-10, 20, 30).
});
```

3.3 Documentation

Nous avons utilisé JSDoc pour documenter le code afin de fournir des descriptions claires et détaillées des fonctions et de leurs paramètres. La documentation générée sera disponible dans le dossier "out", où l'on trouve un fichier "index.html". En ouvrant ce fichier, l'utilisateur pourra accéder à la documentation de chaque fonction du code "app.js", avec des explications sur leur rôle, leurs entrées et leurs sorties. Cela permet de rendre le code plus compréhensible et d'assurer une meilleure maintenance à long terme.


Fichier C:/Users/jerem/Desktop/Projet-CAPI-dev1.0%20(2)/Projet-CAPI-dev1.0/out/global.html#validateVote
Global
Methods
addPlayers()
Ajouter des joueurs au jeu en fonction du nombre saisi.
Source: app.js, line 19
addPreviousButton()
Ajouter un bouton pour revenir à la fonctionnalité précédente.
Source: app.js, line 162
disableCards()
Désactiver les cartes après la fin du jeu.
Source: app.js, line 174
loadBacklog(event)
Charger un fichier JSON contenant un backlog.
Parameters:
Name Type Description
event Event L'événement déclenché par le chargement du fichier.
Source: app.js, line 36
loadSave(event)
Charger un fichier JSON contenant une sauvegarde.
Home
Global
addPlayers
addPreviousButton
disableCards
loadBacklog
loadSave
resetGame
resetVotesAndPlayers
saveProgress
saveResults
showCardsForPlayer
showFeature
showPreviousFeature
showReturnToMenuButton
startGame
startTimer
validateVote
voteForCurrentPlayer