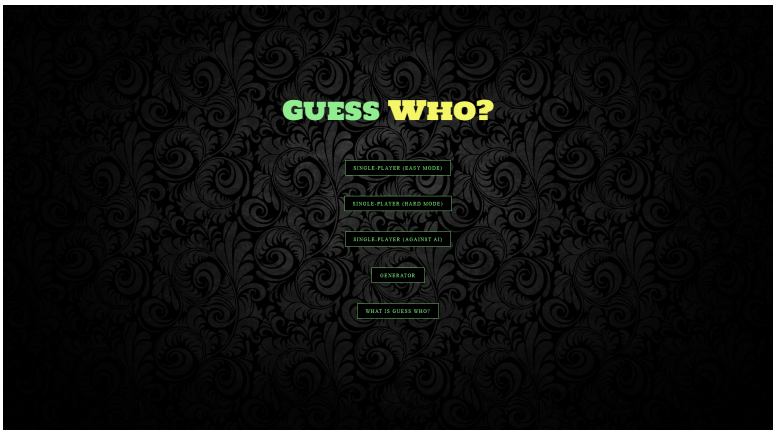
Qui est-ce ?
Projet de programmation

Groupe *W*
Almallouhi Mohamad Satea, Dayioglu Gurgun, El Gargouri Mariem, Karame Hiba
https://github.com/MariemElG/Projet-de-programmation
L2 informatique
Faculté des Sciences
Université de Montpellier.

16 avril 2022



Résumé

"Qui est-ce ?" un jeu de société, de logique et de déduction. Notre tâche était de créer une version numérique et interactive de ce dernier. Pour l'accomplir, on a commencé par travailler sur sa conception, les langages qu'on allait choisir, et le fonctionnement du groupe qu'on allait suivre.
Ensuite, on a petit à petit construit le code commençant par le jeu principal, passant par un générateur nous offrant la possibilité de personnaliser le jeu et finissant par des extensions sous la forme d'un minuteur ou d'un mode difficile où notre adversaire est la machine utilisée.

1 Technologies utilisées et organisation

1.1 Choix du langage

Le langage qu'on a choisi est le JavaScript. C'est un langage de programmation de scripts principalement employé dans les pages web. HTML : HyperText Markup Language qui représente le code utilisé pour structurer une page web et son contenu, était aussi présent dans notre projet.De plus, on a employé les CSS : Cascading Style Sheets, qui forment un langage informatique décrivant la présentation des documents HTML. On a principalement utilisé aussi REACT : une bibliothèque JavaScript libre. Et, enfin, Electron : un environnement permettant de développer des applications multi-plateformes de bureau avec des technologies web .

1.2 Organisation du travail

En ce qui concerne la répartition des tâches, chacun avait son rôle mais en soyant présent à chaque étape du projet.
En effet, Almallouhi Mohamad Satea s'occupait des fichier CSS et a réussi à faire le design du jeu. Ensuite,il a créé notre extension "minuteur".
Dayioglu Gurgun a guidé le groupe tout au long de la partie logique et comme il avait plus d'expérience dans react, il a pu corriger nos erreurs de code et a réussi à le faire marcher.
El Gargouri Mariem a contribué principalement à la création du jeu de base et était respondable du rapport.
Enfin, Karame Hiba a aidé à la conception du jeu, le tester et à nous donner des idées d'amélioration.

Quant au rythme de travail et mode de fonctionnement, il était comme suivant :
A chaque semaine il y a une tâche à finir. On se rencontre à l'université et on travaille dessus. Si un jour si quelqu'un avait des empêchements pour y aller, il rejoignait le groupe à distance.

Chacun avait ses tâches, et même si elles ne se représentaient pas sous la forme d'un code à modifier, on chercherchaient des méthodes qui permettront d'améliorer notre jeu.

Quant au travail sur GitHub, tout membre du groupe a créé une branche où il pouvait modifier le code du jeu ou bien ajouter des éléments. Et, à la fin de notre pépiple, on a rassemblé toutes les parties concevant notre projet.

2 Étape 1 : permettre à l'utilisateur de jouer

Notre projet commence, comme la plupart des projets en informatique par des instructions *import*. Cettes dernières sont utilisées pour importer des liens qui sont exportés par un autre module.
Notre pemière importation concerne *les Hooks*. En effet, en react, on a deux types de composants : composants de classe qui s'étandent de React et des composants fonctionnels. Et, avec la version 16.8.0 des fonctions permettant de bénéficier de fonctionnalités react sans devoir créer une classe, sont apparus. Ce sont les Hooks. Dans notre projet, on a fait recours au Hook «UseState» qui permet d'avoir des variables d'état dans les composants fonctionnels. Et, «UseEffect» permettant d'avoir un effet de bord sur le composant choisi.
A l'étape suivante, on a importé le *countdown* employé au niveau de création du minuteur, *les données* des fichers du format JSON qui seront utilisé dans de différentes sections du jeu, et, enfin la collection *Bootstrap et le fichier CSS* responsables du design et de l'allure du jeu.

Notre jeu donne à l'utilisateur des possibilités diverses. Effectivement, il peut choisir de jouer tout seul, jouer contre sa machine, jouer en mode difficle avec un minuteur, générer un fichier de format JSON pour avoir le même jeu mais personnalisé ou, s'il n'a jamais essayé le "Qui est-ce ?", lire les informations nécessaires à comprendre le jeu.

Alors, comment a-t-on réussi à afficher le jeu tel qu'il est à l'utilisateur ?
Commençons par notre page d'acceuil. C'est une fonction fléchée dans laquelle on commence par intialiser les états des boutons du menu du jeu, son mode et notre minuteur.
Dans un deuxième temps, nous avons travaillé sur ce que cette fonction va retourner. Dans cette partie, l'usage du langage HTML est présent. Il est détectible à travers des éléments comme , conteneur générique du contenu du flux , <ul>, créateur d'une liste d'éléments sans ordre particulier , <li> représentant d'un élément dans une liste et <button> qui permet de créer des boutons. Et, pour faire appel à la liste des classes qui sont positionnées sur nos éléments HTML, la propriété className était la solution. Et grâce à cette partie du code, nous avons pu afficher les boutons et les possibilités.
Passons à la logique. Pour chaque possibilité, on voulait cliquer sur son bouton et avoir une fenêtre qui s'ouvre instantanément. Pour mettre cela en oeuvre, on a créé un composant *PopUp* . Dans ce composant, il y a le <trigger>, c'est à dire, le déclencheur qui est reponsable de l'apparition de la fenêtre et qui dépend de l'état du bouton de la section choisie par l'utilisateur. Ensuite, pour sa fermeture, <setTrigger> contenait l'état de notre bouton fermant de cette section. Ce qu'il faut remarquer cest que le choix de l'utilisateur est mis en action par un click. C'est ce que désigne la fonction *onClick()* présente dans chaque section dédiée à nos boutons de menu.

Parlons du mode facile, ou bien, le jeu standard. La logique dérrière ce mode est comme suit : on affiche les personnages du jeu, l'un parmi eux est choisit par hasard par la machine. Par suite, l'utilisateur sélectionne des attributs avec des spécifications de son choix. Si ces derniers correspondent aux spécifications du personnage choisit au début, il gagne, sinon, on élimine ceux ayant les mauvaises spécifications.

Du côté programmation, l'organisation de notre code ne suit pas la logique mentionnée juste avant. En effet, le fonctionnement de notre jeu basique est traité dans la fonction *Board()*. Dans cette dernière, on commence par initialiser la liste de nos questions. Cette liste dépend des questions posées du jeu sauvegardé.
Mais, comment le jeu choisit-il le personnage ?
Une question à laquelle on a répondu par une condition ternaire. En effet, si notre jeu est sauvegardé, on récupère le personnage déja choisit grâce à la méthode *json.parse()*. Cette méthode est capable d'analyser une chaîne de caractères JSON et de construire la valeur JavaScript ou l'objet décrit par cette chaîne. Sinon, un personnage est choisit au hasard.

Ensuite, on initialise aussi le personnage éliminé. Ce personnage est représenté par une constante, accessible uniquement en lecture, que signifie la déclaration *const*. Elle contient un tableau vide au début qui se remplit au cours du jeu. Effectivement, pour remplir ce tableau, on avait besoin d'un fichier léger d'échange de données et le format JSON était parfait pour pourvoir de cette proriété.

Pour mieux expliquer notre démarche, il faut expliquer ce que contiennet les fichiers sous ce format pratique. En effet, ils contiennent des objets. Le premier désigne l'emplacement des images qu'on veut afficher. Et, le deuxième représente un object de type tableau contenant les données des personnages, c'est à dire, leurs attributs et toutes les descriptions. Dans notre projet, nous avons choisi d'appeler cet objet de type tableau *possibilités*.
Retournons à notre code d'application. On était en train de trouver une solution pour éliminer des personnages. Notre idée était d'importer nos données des fichiers JSON et les utiliser en les désignant par *data*, *dataGen*, *dataOG* qui seront déployés respectivement dans la partie du code concernant le jeu principal, le générateur et le choix standard des personnages au niveau du générateur.
Alors, en utilisant ces données, on est maintenant capable d'éliminer quelques éléments. En effet, dans notre jeu, les questions sont les attributs des personnages. Si, la description de cet attribut, choisit par l'utilisateur, ne correspond pas à celle du personnage souhaité et que la liste des personnages éliminés ne contient pas ce dernier, on le rajoute au tableau avec la méthode *push()*.

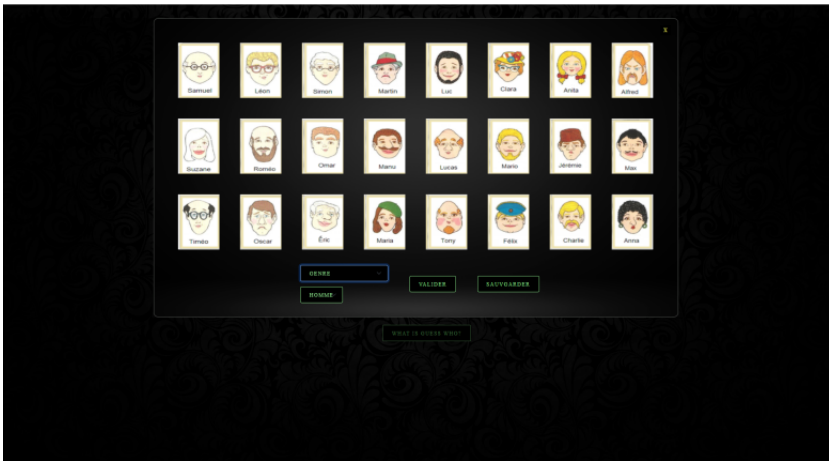
Il faut expliquer maintenant notre fonction *handlesubmit()* qui traite la question choisie. En effet, on annule le choix par défaut du jeu car il ne contient pas de spécifications. Ceci est fait

grâce à *event.preventDefault()*. Ensuite, si l'utilisateur ne donne pas de spécifications non plus, on lui demande de le faire. S'il suit notre consigne, on compare ses spécifications à celles du personnage désiré et on lui affiche un résultat contenant "vrai" ou "faux". Après, on modifie la liste des questions possibles restantes avec*props.SetQuestions* et à l'aide des trois points... représentant la notation de propagation de propriété, on ajoute la question posée précedemment.

Dans ce jeu, l'utilisateur a la possibilité de sauvegarder son jeu. Pour cela, on à créé une fonction *handleSaveClick()*. Dans cette fontion, on définie *session* qui va être stockée dans notre constante de sauvegarde.
En effet, tout au début, on a créé la constante de sauvegarde à l'aide du *localStorage* , qui est une technique d'enregistrement des données et *get.item*, qui est une méthode permettant de spécifier l'item à récupérer.
Ensuite, on tranforme notre personnage choisit et les questions en string grâce à *json.stringify()* et avec *window.location.reload(false)*, on recharge la page en utilisant sa version mise en cache.

Notre utilisateur gagne lorsque tous les personnages, autre que celui choisit, sont éliminés. Cette condition était représentée par la comparaison de longeurs de deux tableaux. Le premier est le tableau des personnages éliminés, et le deuxième est les possibilités, c'est à dire, les personnages restant comme choix possible dans le jeu.
Suite au gain, un message de félicitation apparaît au joueur en lui rappelant le nombre de questions posées. Ce message apparait grâce à la méthode *alert()*.

A ce stade là, on peut dire qu'on a expliqué toute la logique derrière les éléments de notre jeu basique. Cependant, il faut aussi parler de l'affichage que regarde l'utilisateur.
Effectivement, nous traitons cet affichage dans la partie *return* de la fonction *Board()* et après les modifications du joueur, nous proposons deux cas. Si un personnage est éliminé, une nouvelle photo apparaît donnant l'abréviation *R.I.P : Rest In Peace* qui signifie en français *Reposez en paix* . Sinon, on a un affichage de nos données des fichiers du format JSON.
Ce qu'on peut aussi expliquer est le code permettant l'affichage des photos. En effet, la source est donnée grâce à l'attribut *src* dans la balise <img>.Et, une description textuelle est présente grâce à l'attribut *alt* dans la même balise. Ici, c'est le nom du personnage en question.



Le jeu basique

3 Étape 2 : aider à la saisie des personnages

Passons à notre deuxième étape de réalisation du projet : réalisation du générateur.
Notre objectif était de permettre à l'utilisateur de créer un jeu personnalisé. La démarche pour arriver à cet objectif était de créer un générateur de fichiers de format JSON contenant les nouveaux données désirés par notre joueur.
Ce générateur créé est accessible à l'utilisateur à travers le bouton *GENERATOR*. Quand il clique dessus, une nouvelle fenêtre s'affiche. Dans cette fenêtre, la première chose qu'on remarque est les cartes avec les points d'interrogation dessus :ce sont nos prochains personnages. Après, on peut remarquer quelques foncionnalités comme *AJOUTER UN ATTRIBUT*, *CHOISIR DES IMAGES*, *VALIDER LA CONFIGURATION* ou même *RESET GAME*.
Expliquons alors ces fonctionnalités et leur code.

Commençons par l'ajout des images. C'est dans la fonction *ChoisirImages()* qu'on traite la logique derrière cette étape. En effet, dans cette partie du code, on a du utiliser Electron. Les images que l'utilisateur veut jouer doivent être dans un dossier existant sur sa machine. Alors, on a d'abord notre configuration de "dialogue" entre l'utilisateur et le jeu à travers laquelle on va renseigner le repértoire souhaité où se place notre dossier. Et, pour pouvoir sélectionner le répertoire, on utilise la méthode *window.dialog.openDialog()* qui est une extension de *window.open()* de javascript. Ensuite, avec *fs.readdirSync ()* on réussit à lire les composants de notre répertoire choisit . S'ils ont l'extension sous la forme ".png" ou ".jpg" ou ".jpeg", on fait une copie de ces fichiers de la source à la destination grâce à la méthode *window.fs.copyFileSync()*. Enfin, avec la méthode *window.fs.writeFileSync()*, si notre fichier JSON contenant ces images n'existe pas, on le crée

S'il a ses images, l'utilisateur a besoin maintenant d'ajouter des attributs et les specifier. Pour effectuer des modifications sur les personnage, nous avons créé une fonction spécifique : *ModifierPersonnage()*. Dedans, on a une sous fonction *handleSubmit()* qui traite la soumission de requête. En effet, on empêche le choix par défaut du jeu de se faire et on crée une varibale "testPassed". Cette dernière permettra de vérifier si toutes les spécifications des personnages sont toutes renseignées après l'ajout des attributs. Quand sa valeur est "false" et que l'utilisateur essaye de confirmer ses choix, on lui indique qu'il doit tout remplier avant de continuer. Dans une autre étape, si un personnage est choisi et notre attribut saisie est différent de l'attribut "fichier", on ajoute ce dernier à l'objet "possibilités", qui existe dans le fichier JSON concernant cette partie du jeu. Et comme mentionné avant, grâce à la méthode *window.fs.writeFileSync()*, si notre fichier du format JSON n'exite pas, elle le crée.

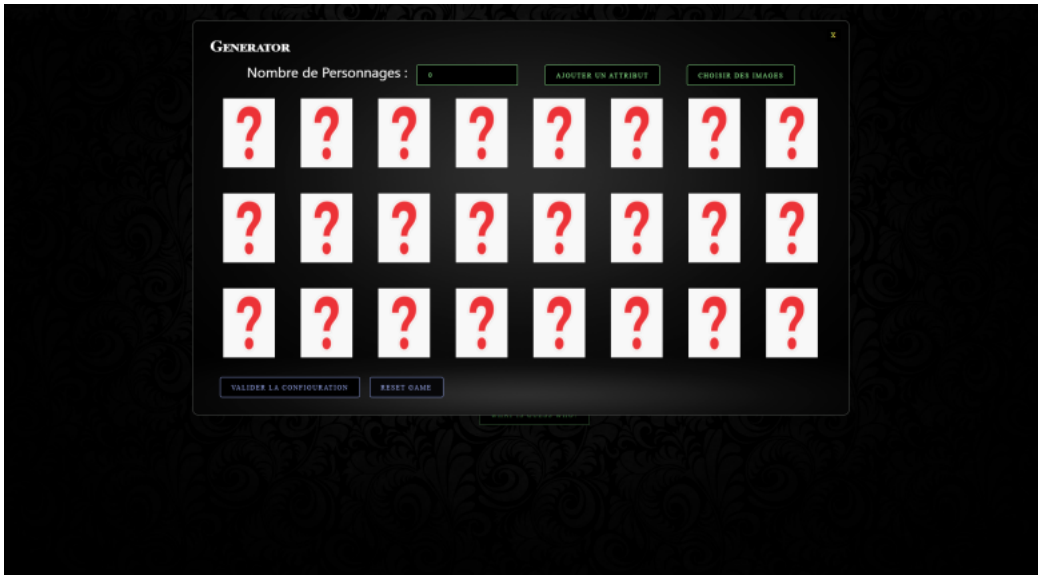
Passons alors maintenant à ce que notre fonction va retourner.Dans un premier temps, on peut 'modifier le personnage' , c'est à dire, lui ajouter des attributs. Pour cela, on affiche non seulement des champs de saisie à l'utilisateur qu'il pourra remplir mais aussi le bouton "valider" sur lequel il cliquera pour soumettre sa proposition.

Cequ'il faut souligner avant l'ajout des attributs, c'est la vérification de l'existance des images. C'est le rôle du code dans notre fonction *AjouterAttribut()* après l'empêchement de réalisation du choix par défaut. Si tout est bien, l'attribut est ajouté et copié aussi dans le fichier du format JSON concerné. Et si ce dernier n'existe pas, il sera créé avec la méthode *window.fs.writeFileSync()*.
Ce qu'on retounre à la fin de cette fonction est bien sûr les champs de saisie nécessaires pour pouvoir ajouter des attributs.

Pour valider notre configuration, on a créé la fonction *ValiderConfig()*. Elle vérifie qu'il y a au moins un attribut, que tous les attributs sont remplis. Enfin, ajoute les éléments au fichier du format JSON s'il existe, sinon elle le crée de la même manière expliquée précedemment. Ensuite, on retourne les outils, c'est à dire les boutons, indispensable à l'utilisateur pour qu'il puisse valider sa configuration et jouer.

On affiche dans le generateur aussi le nombre de personnages du jeu. Effectivement, notre fonction créé verifie s'il y a des images,si c'est le cas, elle affiche leur nombre, sinon elle affiche 0.

En lisant notre code, on remarque qu'à la fin, on a une fonction *table()*dans cette fonction, on a l'initialisation du fichier de format JSON du générateur. Ensuite, avec la fonction *personnageNone()*, on trouvera ce que le générateur affiche avant les modifications de l'utilisateur. Enfin, grâce a la fonction *personnage1()*, on est capable d'afficher nos nouveaux éléments du jeu.

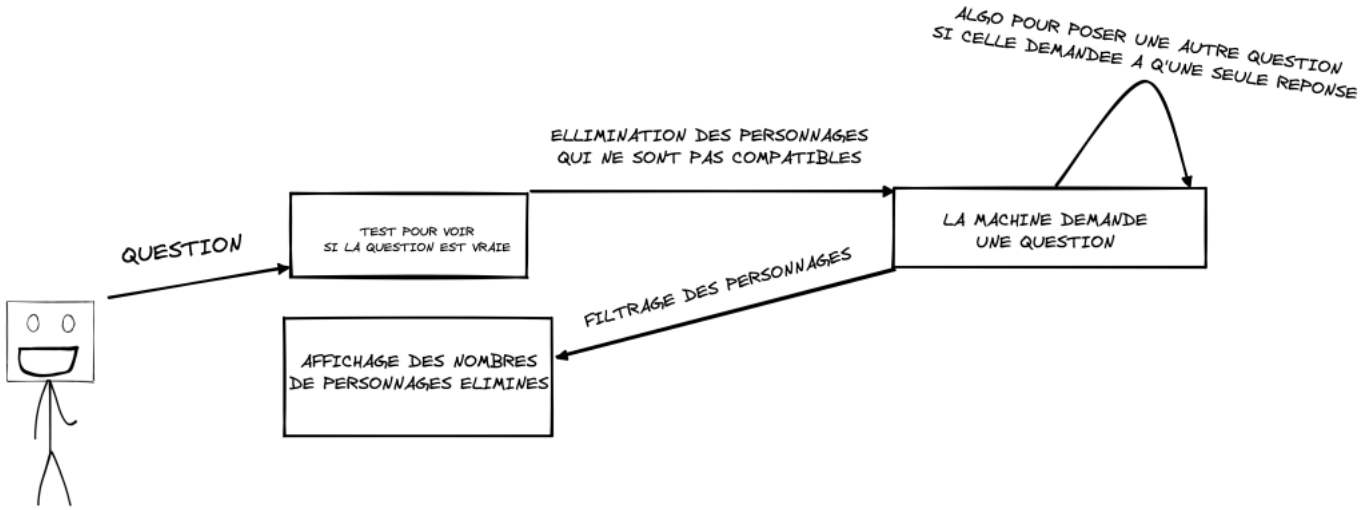


Générateur

4 Étape 3 : Ajout du jeu contre la machine et du mode difficile

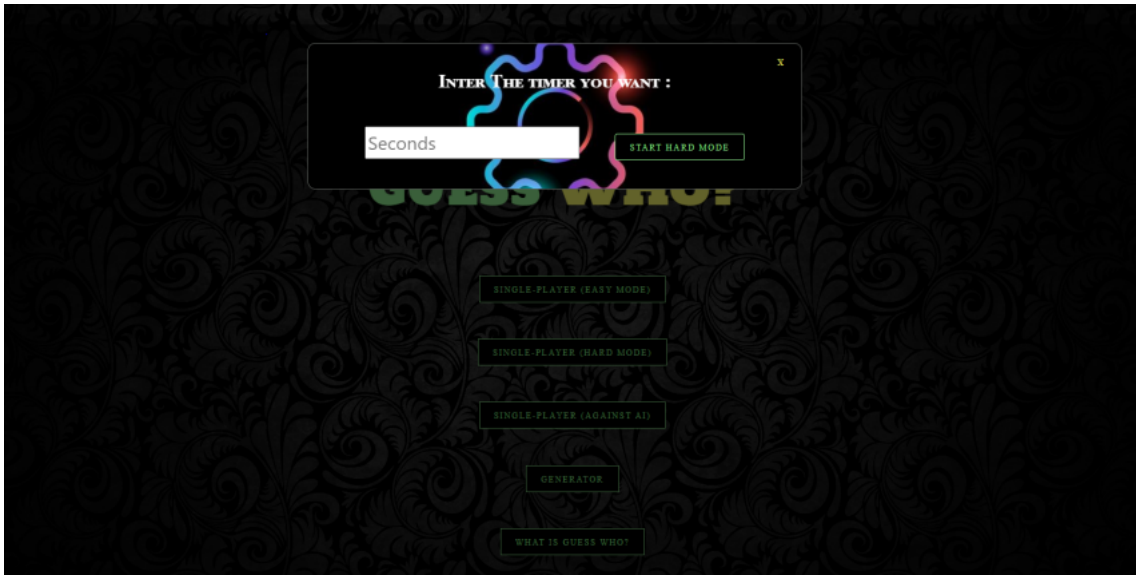
Dans la dernière étape du programmation du jeu, on avait comme but ajouter deux extensions. Le premièr se représente sous forme d'un jeu contre la machine, et, le deuxième sous forme d'un mode difficile avec un minuteur limitant le temps du joueur.

Si, on parle du jeu contre la machine, on aura beaucoup de ressemblances avec notre jeu basique. En effet, le choix des questions est fait par hasard, on a la même démarche d'ajout des éléments aux fichiers JSON et on élimine les personnages de la même manière. Cependant, la machine possède un algorithme qui peut améliorer le choix des questions.
On peut représenter le comportement de notre jeu contre machine à travers la figure suivante :

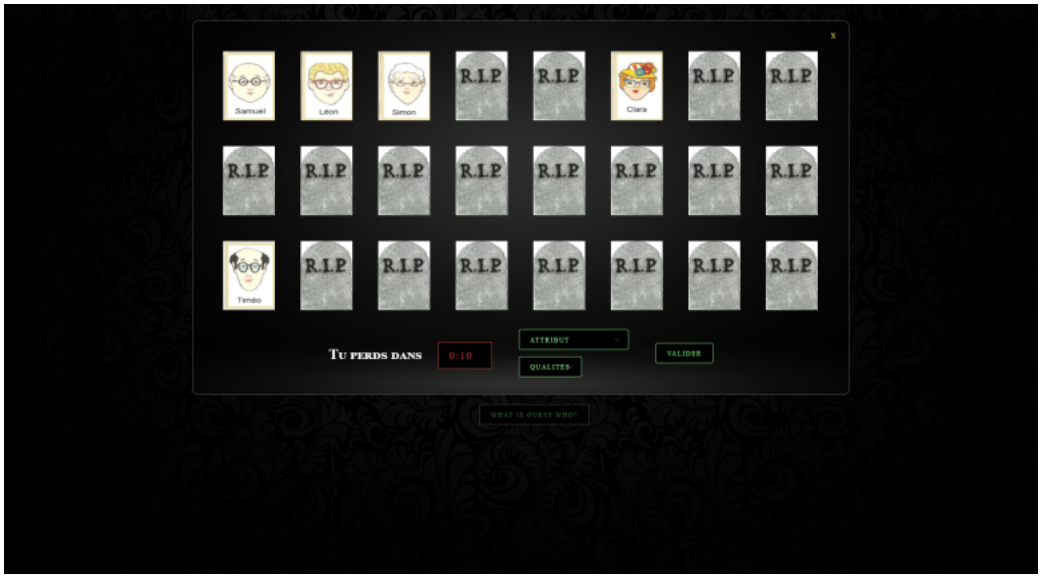


Si on passe au mode difficile maintenant, on aura de quoi parler. En effet, c'est un mode qui s'ouvre dans une nouvelle fenêtre grâce à la fonction *popUpTimer()*. La première étape consiste à donner le temps limitant voulu par l'utilisateur et la deuxième consiste à essayer de gagne avant que ce temps se coule.
Traitons cela du côte du code. D'abord, on a fait une fontion *handleSubmit1()* qui n'accepte que les valeurs supérieures à 1. Ensuite, on a changé quelques états sur lesquels dépend le jeu, comme par exemple l'activation du mode difficile, l'apparition de notre fenêtre et le temps choisi pour finir le jeu. Enfin, on a procédé à construire la partie que retourne notre fonction *popUpTimer()*. Ici, nous avons fait appel à un formulaire à travers la balise <form> dans lequel <onSubmit>, signifiant "sous soumission", active notre fontion *handleSubmit1()*.

Passons à l'explication de notre deuxième partie de ce mode.
En créant le compte à rebours, il fallait créer une constante, qu'on a appelé "Completionist" que le jeu retourne au cas du fin du comptage. Mais, au cours de ce dernier, on a ajouté "time" avec l'instruction var qui permet de déclarer une variable et éventuellement d'initialiser sa valeur. c'est cette variable qui va être initialisé par l'utilisateur et qui décrémente au cours du temps. Dans cette section, on utilise le countdown qu'on a importé au début du fichier. Il existe déjà dans l'API react, et est responsable de l'incrémentation du temps. Dans sa balise, on appelle la fonction renderer, qui prends en paramètres les minutes, secondes et l'état du temps.Ensuite, on remarque un premier *useeffect()*. Ce dernier est responsable de réinitialieser le minuteur. Quant au deuxième, il s'occupe de récuperer le temps quand la page rafraichit suite à une validation. Et comme d'habitude, après la logique, on vous explique l'affiche. En effet, pour que l'utilisateur puisse faire rentrer le temps du jeu voulu, nous avons dû appeler l'élément HTML <input> qui permet de saisir des données. Pour décrire le champ de saisie, on a fait recours à <placeholder> afin de guider l'utilisateur à mettre le temps en secondes. Par suite, on a donné "timer" comme identifiant de cet élément du code.



Saisie du temps limitant



Jeu avec temps limitant

5 Bilan et Conclusions

Tout au long du projet, nous avons été confrontés à de nombreux problèmes techniques au cours du développement. Heureusement, nous avons pu résoudreun bon nombre de ces problèmes en utilisant des bibliothèques supplémentaires ou en faisant preuve de créativité.

Le premier problème est venu de notre choix d'outils. Comme nous utilisions un serveur web local pour jouer au jeu, nous nous bloquions l'accès à tout fichier sur le système d'exploitation de l'utilisateur, ce qui nous empêchait d'implémenter un générateur. Nous avons résolu ce problème en passant à Electron, comme expliqué précédemment.

Le deuxième problème était de faire fonctionner l'IA. Plus précisément, jouer contre la machine n'était pas assez agréable. La machine posait parfois la même question, ou n'était pas capable de poser de "bonnes" questions. Pour ce problème, nous avons ajouté un petit algorithme qui vérifie si une question donnée a une seule réponse. Si c'est le cas, la machine pose une autre question. Nous voulions également ajouter une condition qui empêche la machine de poser une question dont le nombre de réponses est supérieur au nombre moyen de réponses de toutes les questions possibles. Cependant, cela aurait pour effet de casser le jeu sur certaines configurations. Par exemple, 5 questions ayant toutes 2 réponses entraîneraient une boucle infinie de tentatives.

Un autre problème auquel nous avons été confrontés concerne le mode de jeu chronométré. Dans REACT, ce n'est pas toujours possible de stocker l'état, le mettre à jour ET afficher la version mise à jour instantanément. Pour résoudre le problème, nous avons utilisé l'API LocalStorage du navigateur - puisque nous fonctionnions sur Electron, nous pouvions le faire. Après le démarrage du minuteur, toutes les secondes, le minuteur enregistre le compte dans LocalStorage et le récupère lorsqu'une question est posée. De cette façon, nous pouvions mettre à jour l'interface utilisateur, puis reprendre le minuteur avec la dernière valeur enregistrée.