**RAPPORT DE CONCEPTION DÉTAILLÉE**

Table des matières

[1. Architecture Générale 3](#_Toc116282839)

[ Frontend 3](#_Toc116282840)

[ Backend 4](#_Toc116282841)

[2. Gestion d’état 4](#_Toc116282842)

[a. Définition 4](#_Toc116282843)

[b. Difficultés Rencontrés et Solutions 4](#_Toc116282844)

[3. Les types d’exercices 5](#_Toc116282845)

[**a.** **Exercice de type « sélection »** 5](#_Toc116282846)

[**b.** **Exercice de type « drag and drop »** 7](#_Toc116282847)

[c. **Exercice de type « Memory »** 9](#_Toc116282848)

[**d.** **Exercice de type « Link »** 12](#_Toc116282849)

[**e.** **Exercice de type « Fill » ou « phrases à trous »** 13](#_Toc116282850)

[**f.** **Exercice de type « Sériation » :** 16](#_Toc116282851)

[4. Le design pattern Strategy 18](#_Toc116282852)

[5. Le routage 22](#_Toc116282853)

[a. Le routage du dashboard professeur. 22](#_Toc116282854)

[b. Le routage du dashboard Parent 24](#_Toc116282855)

[c. Le routage du dashboard enfant 25](#_Toc116282856)

# Architecture Générale

Le schéma suivant représente l’architecture générale du site ainsi que les différents composants qui le permettent de fonctionner correctement.

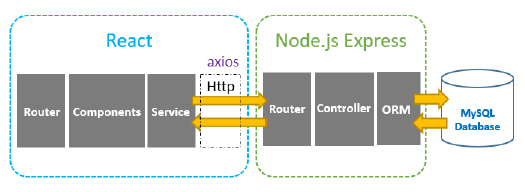
****



Figure 1 : Architecture Générale du site

## Frontend

Se compose de 3 parties :

* **Le Router : ReactRouter** permet de naviguer entre les différentes pages du site.
* **Les Composants :** constitue l’aspect visuel du site, ils sont répartis selon une architecture appelée « **atomic design** » qui décompose la page en 3 parties
  + **Atoms :** les composants les plus petits et qui ne dépend d’aucun autre composant, par exemple un bouton, un titre …
  + **Molecules :** ils se composent d’un ensemble d’Atom, par exemple le header qui se compose de l’Atom Logo et d’une liste de liens.
  + **Organisms :** sont constitué de plusieurs Molecules, par exemple le contenu de l’exercice (liste des cartes) ainsi que la barre qui permet de valider ou réinitialiser une question.
* **Les services :** chaque service est lié généralement à un Model donc il est lié indirectement à une table de la base de données, le rôle du service et d’envoyer des requêtes au serveur via Axios (protocole HTTP) et de fournir les données reçues aux composants qui s’en charge de la visualisation.

## Backend

Se compose de 3 parties :

* **Le Router :** le router intercepte les requêtes provenant du frontend et selon la requête appel le contrôleur adéquat pour la traiter et retourner une réponse qui est envoyé à la fin au frontend.
* **Le contrôleur :** interroge la base de données à travers l’ORM et récupère les données, puis les envois au frontend.
* **ORM :** c’est la partie qui lie chaque model à la table correspondante de la base de données.

# Gestion d’état

## Définition

Un état est une instance d’un composant React et peut être défini comme un objet d'un ensemble de propriétés qui contrôlent le comportement du composant. Autrement dit, l'état d'un composant est un objet qui contient des informations susceptibles de changer au cours de la durée de vie du composant.

## Difficultés Rencontrés et Solutions

* React peut regrouper plusieurs appels de setState() dans une mise à jour unique pour la performance. Étant donné que les props et le state peuvent être mis à jour de manière asynchrone, il ne faut pas faire confiance à leurs valeurs pour calculer l'état suivant.

Un exemple simple serait d’itérer sur la liste des matières reçu, et de stocker cette liste dans l’état. Dans ce cas il faut itérer sur la liste toute en se basant sur l’état précédant pour le mettre à jour à chaque fois.

La solution c’est d’utiliser la version surchargée du la fonction setState(), qui consiste à passer une fonction à celle-ci en passant l’état précédant en paramètre.

* Dans certain cas, on peut avoir besoin d’une donnée considéré comme globale dans plusieurs composants différents.

Comment faire par exemple pour stocker une référence vers la première carte choisie (exercice memory), et de comparer celle-ci avec la deuxième carte choisie.

Une idée est d’utiliser **Context API** qui permet de rendre les données globales, dans ce cas une référence vers la première carte sera stockée et au moment où la deuxième carte est choisi on peut la comparer avec la première carte.

**Context API** permet aussi d’éviter ce qu’on appelle le **Prop Drilling** qui est tout simplement le fait de devoir passer une information d’un composant parent à un composant enfant en passant par tous les composants entre eux qui n’ont pas besoin de cette information.

# Les types d’exercices

## **Exercice de type « sélection »**

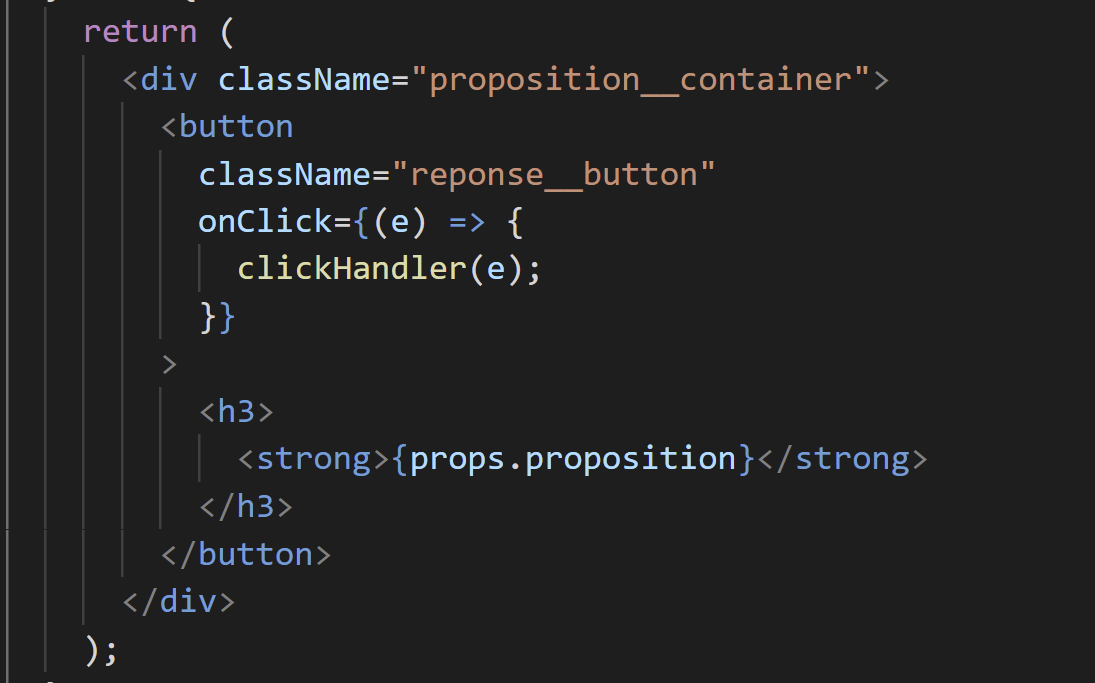
****

Figure 1 : Proposition de type Sélection

Le clic sur un bouton exécute la fonction **clickHandler** qui ajoute la chaine « **activeChoice** » à la liste des classeName du bouton (figure 2), ce qui permet de récupérer les éléments qui contient cette chaine au moment de l’évaluation et les comparer avec la liste des réponses juste (figure 3).



Figure 2 : le bloc exécuté quand un bouton est appuyé

Pour l’évaluation, on vérifie tout d’abord si le nombre de réponses choisi est égale au nombre de réponses juste, si ce n’est pas le cas, on peut dire que l’élève a échoué à l’exercice.

Si le nombre de réponses choisi (liste 1) est égale au nombre de réponses juste (liste 2) on vérifie si chaque élément de la première liste existe dans la deuxième liste.

Le tableau **arr** contient les éléments sélectionnés par l’utilisateur, celui-ci est comparé avec les réponses juste stocké dans le tableau **reponses** (ligne 97).

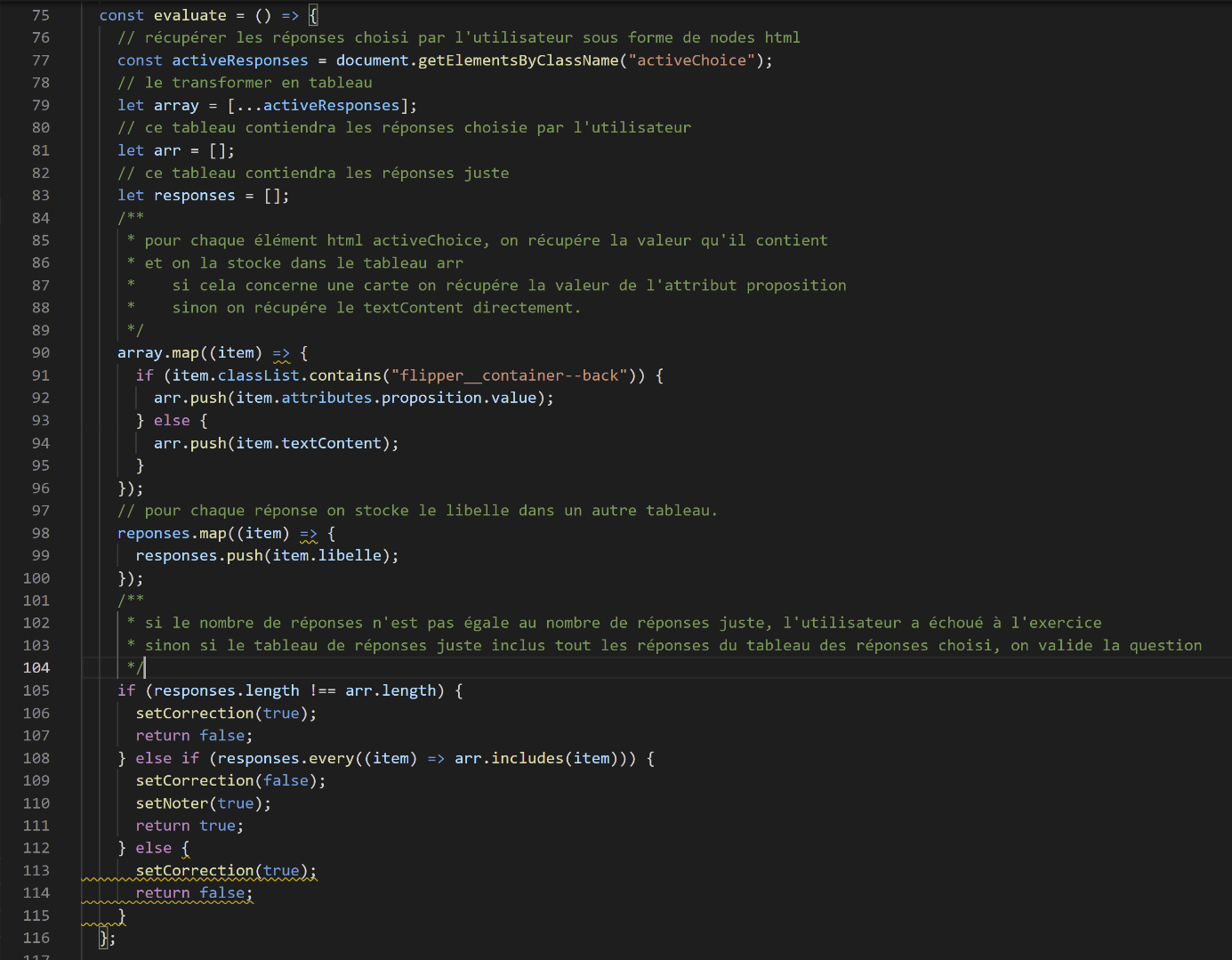


Figure 3 : fonction d'évaluation

## **Exercice de type « drag and drop »**

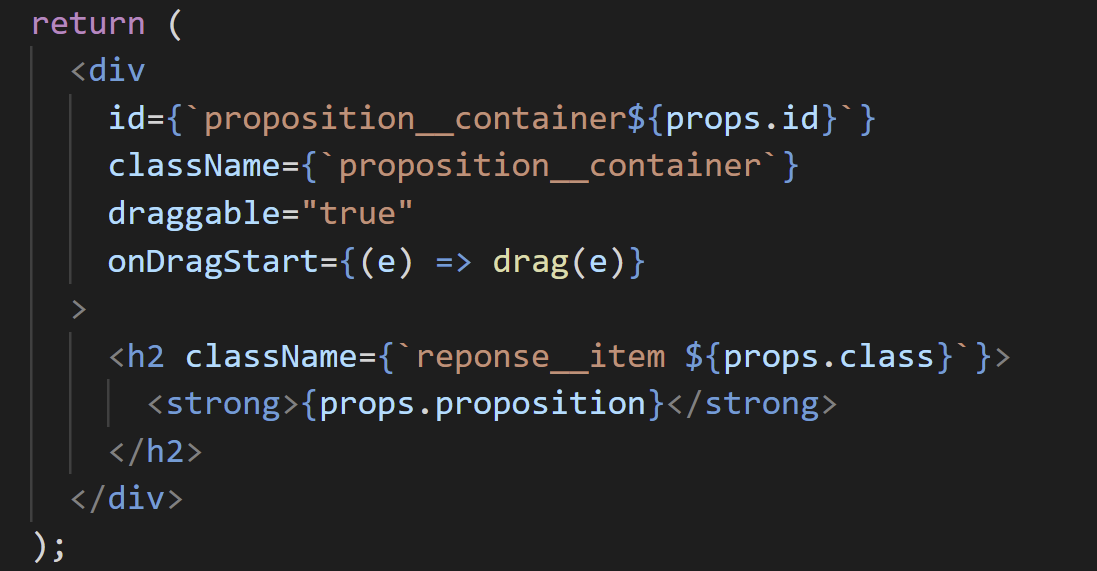
****

Figure 1 : Proposition de type drag and drop

Il faut tout d’abord permettre de glisser déposer dans les deux zones (zone des réponses possibles et la zone de réponses), donc on rend l’élément glissable avec **draggable= «true»** (figure 1).

On a besoin maintenant d’une fonction **drag** (figure 1 et 2) de glissement qui permettra d’enregistrer l’élément glissable dans un évènement.



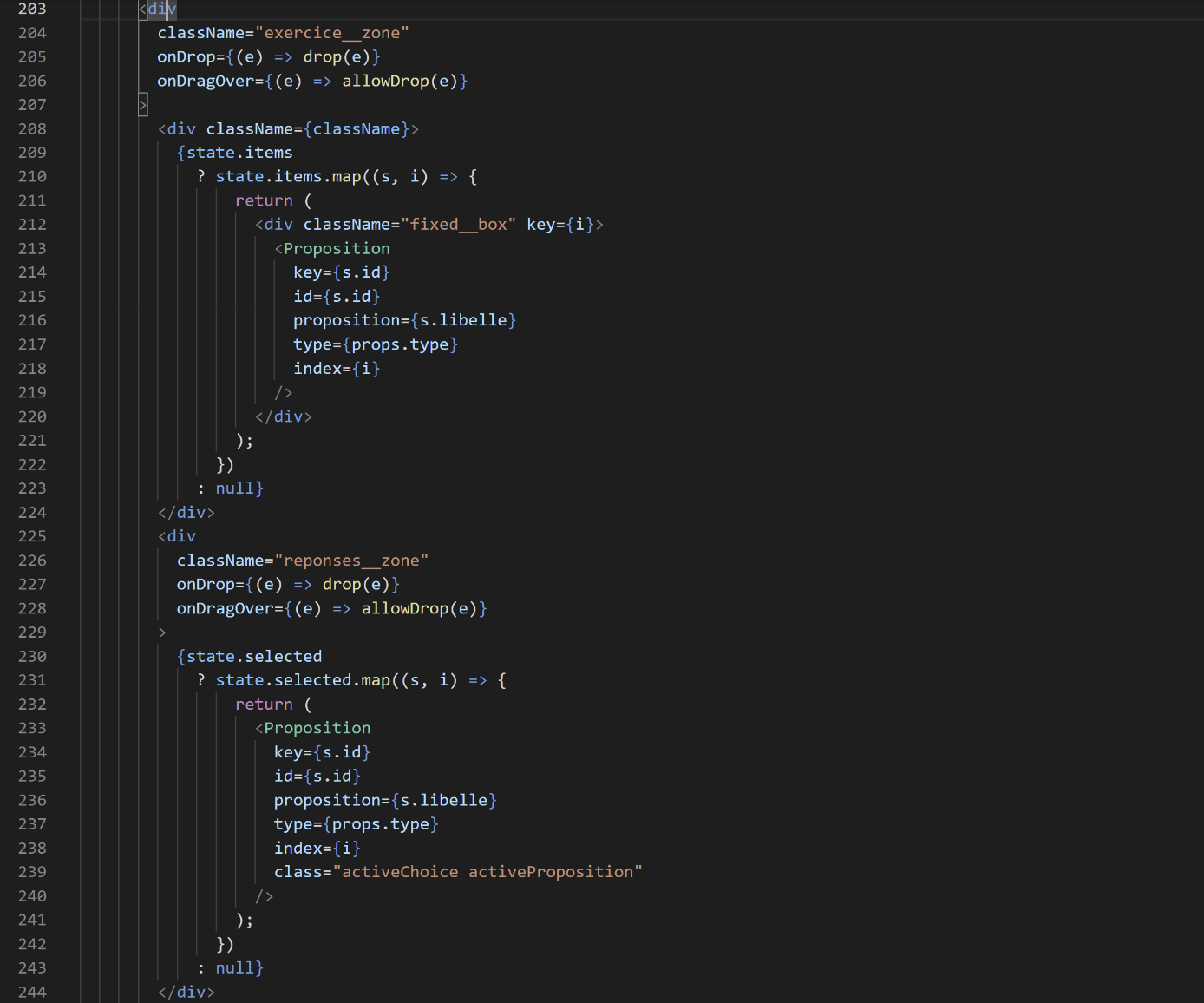
Figure 2 : fonction permettant de glisser un élément

On peut maintenant définir la méthode **drop** (figure 3) qui permettra de déposer l’élément à la position souhaité.

Comme vous pouvez le remarquer, la chaine **« activeChoice »** est utilisé ici pour identifier les réponses dans la zone de réponses.



Figure 3 : fonction qui permet de déposer un élément

****

L’état **state** est un objet qui contient deux listes, une liste **items** qui correspond aux éléments non sélectionnés et une deuxième liste **selected** qui correspond aux éléments sélectionnés.

Le fait de glisser une réponse possible vers la zone de réponses ajoute la chaine « activeChoice » à la liste des classeName du bouton, ce qui permet de récupérer les éléments qui contient cette chaine au moment de l’évaluation et de les comparer avec la liste des réponses juste.

## **Exercice de type « Memory »**

Le jeu se compose de paires de cartes portant des illustrations identiques. L’élève retourne deux cartes de son choix. S'il découvre deux cartes identiques, ils restent visibles. Si les cartes ne sont pas identiques, ils sont retournés faces cachées.

Le jeu se termine quand toutes les paires de cartes ont été découvertes.

On utilise l’élément SingleCard pour l’affichage des différentes cartes. Le jeu commence après avoir lancé la partie grâce au bouton Nouvelle partie.

* **Affichage**

Une image contenant texte

Description générée automatiquement

Figure 4: Rendu visuel de la carte

La fonction shuffleCards permet la création du tableau des cartes qui sera affiché suivant la difficulté créé dans la variable niveau. C’est aussi dans cette fonction que les cartes seront melangé avant affichage.

Une image contenant texte

Description générée automatiquement

Quand l’un de ces deux états globaux est initialisé, la fonction **useEffect** est déclenché, ce qui permettra de vérifier si les deux cartes retournées ont la même valeur de match et donc corresponde entre elles

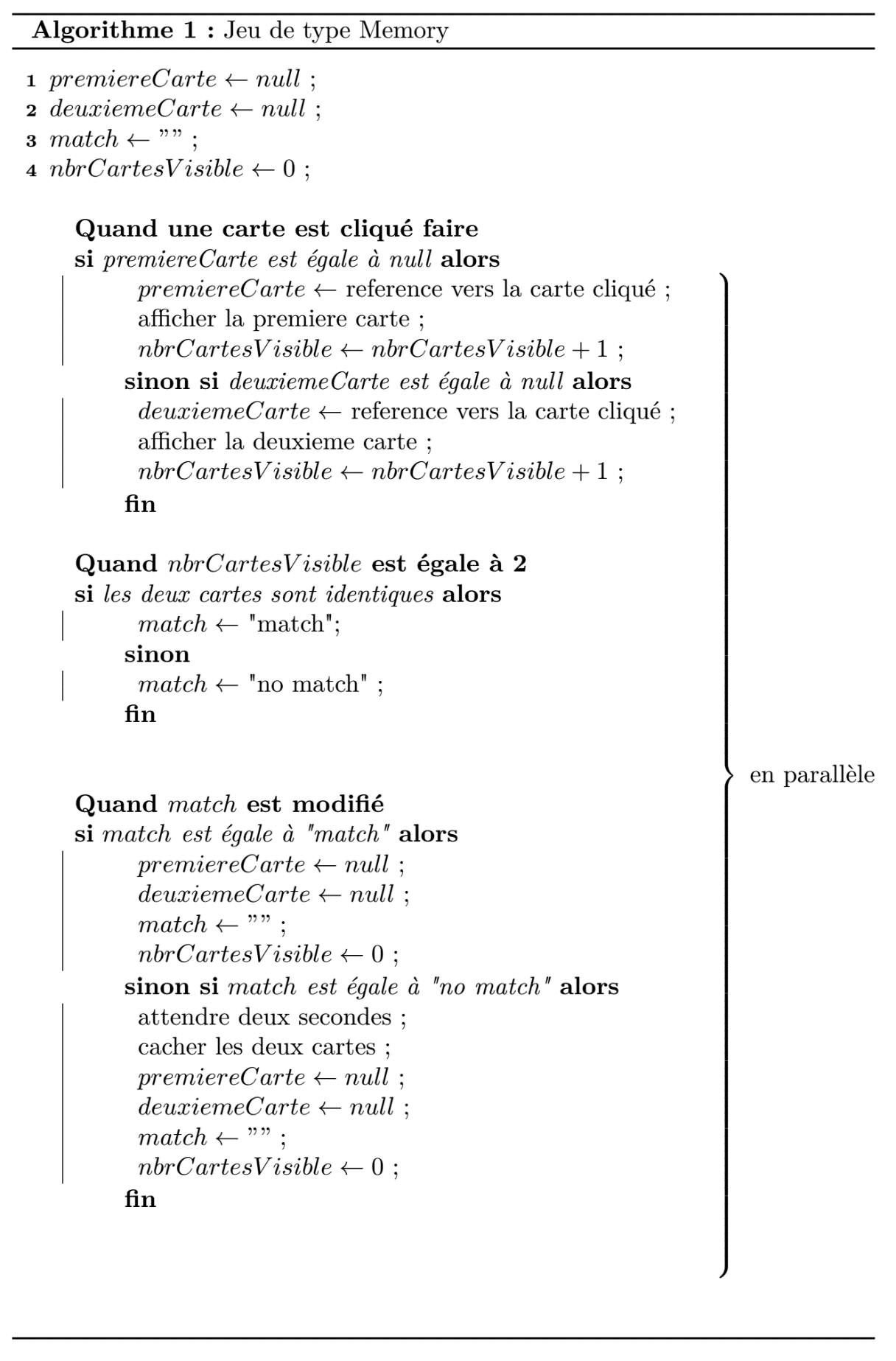
Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

* **Pseudo-code**

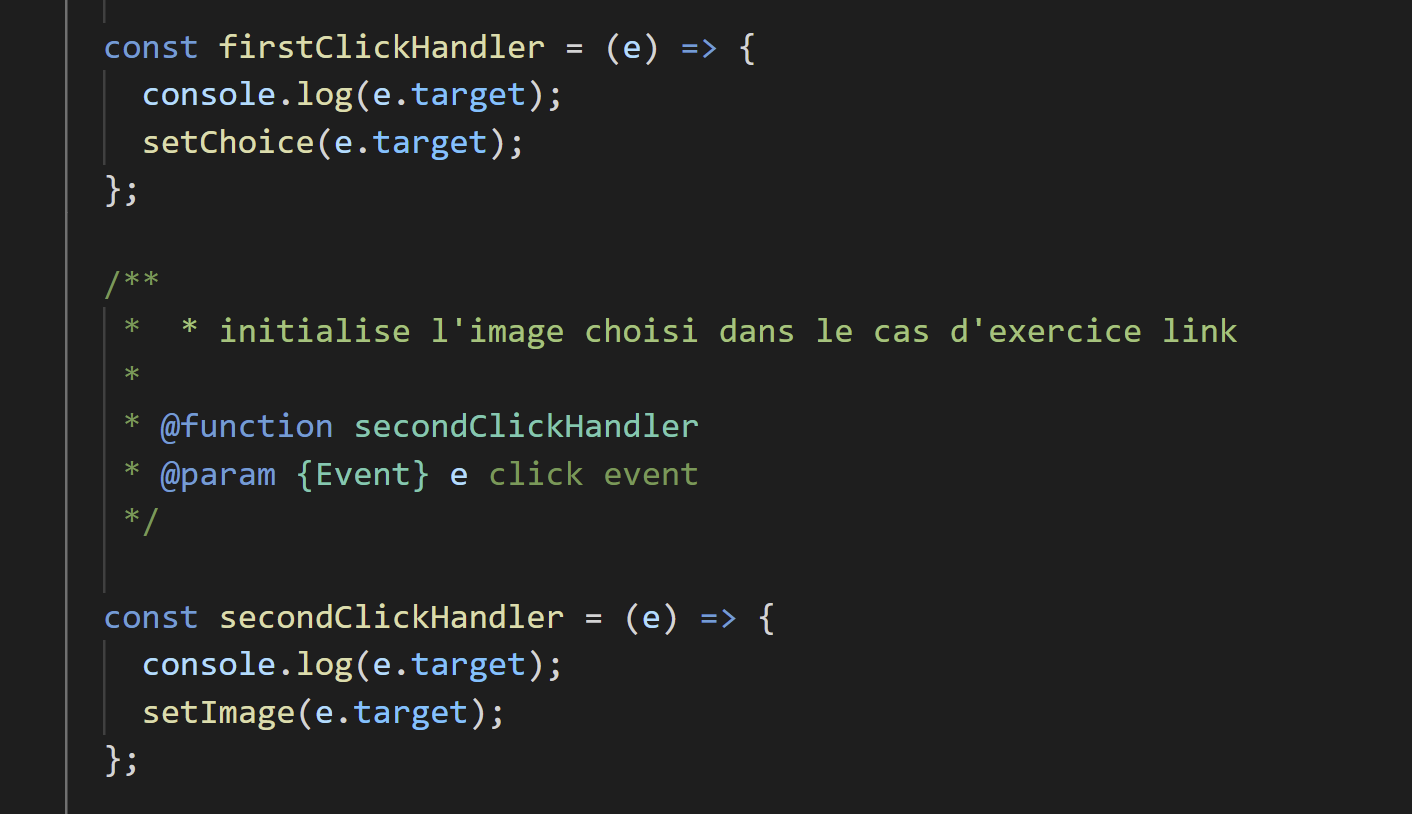
****

## **Exercice de type « Link »**

L’exercice de type link permet de lier les éléments qui se correspond, sachant que les éléments à gauche sont des textes et les éléments à droite sont des images.



La méthode **firstClickHandler** sert à récupérer le texte choisi et la méthode **secondClickHandler** sert à récupérer l’image choisi correspondante.



Dès que les deux éléments sont choisis, on crée une flèche du premier texte choisi vers l’image correspondante avec la bibliothèque **React LeaderLine** et on réinitialise les deux états correspondant au texte et à l’image sélectionnés.



## **Exercice de type « Fill » ou « phrases à trous »**

Cet exercice concerne les phrases à trous mais aussi le calcul posé, l’idée est d’avoir des cases vides à certains positions, ces positions sont stockées dans la table **reponse** avec les réponses justes.

Pour une phrase à trous, on spécifie les réponses justes et ses positions de cette façon dans la base de données :

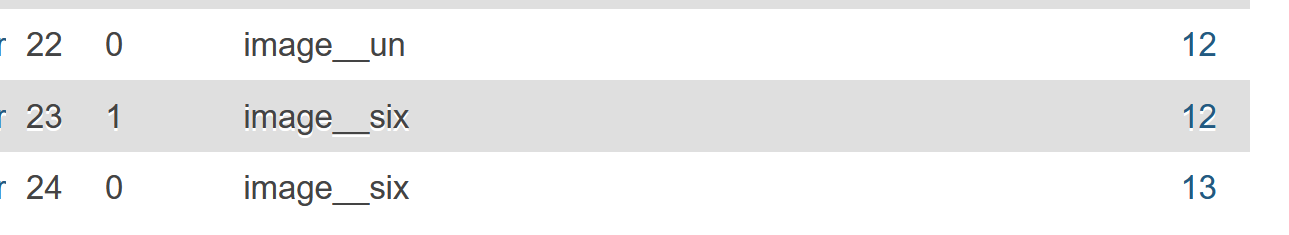


En récupérant ces réponses, on peut savoir où insérer les cases vides. Le code ci-après permet de transformer la phrase en question à un tableau de mots, d’insérer la case vide à la bonne position et de reconstruire la phrase.

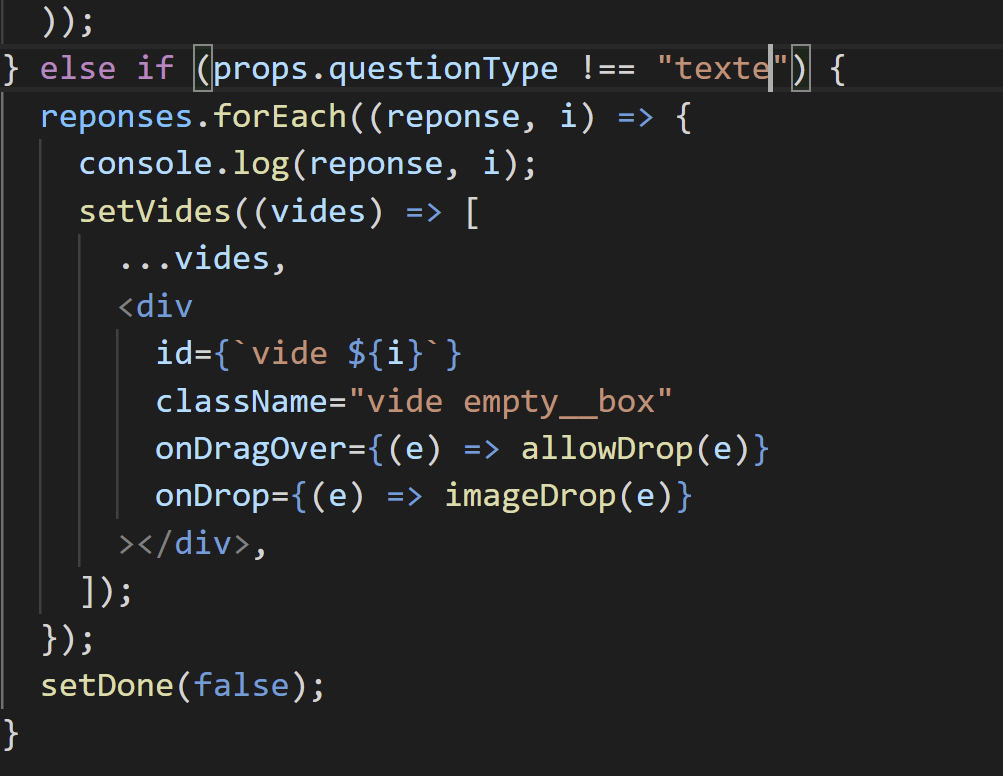


L’évaluation se fait en vérifiant la position du texte glissé mais aussi en vérifiant l’intitulé du texte.

Concernant le calcul posé, une liste de réponses correspondant à des identifiants d’images est stockée dans la table **reponse** de la base de données, ces images représentent les chiffres de 0 à 9 avec lesquelles l’enfant peut former un nombre.



Le code suivant initialise les cases vides qu’on va ajouter par la suite à l’ensemble du calcul posé.



Le code suivant analyse le texte et récupère tout d’abord le signe de l’opération, on récupère par la suite les deux nombres de l’opération et on initialise le texte de l’opération toute en insérant les cases vides à la fin.



L’évaluation se fait en vérifiant la position du l’image glissé mais aussi en vérifiant si son identifiant correspond au texte enregistré dans la base de données comme le montre le code ci-dessous.

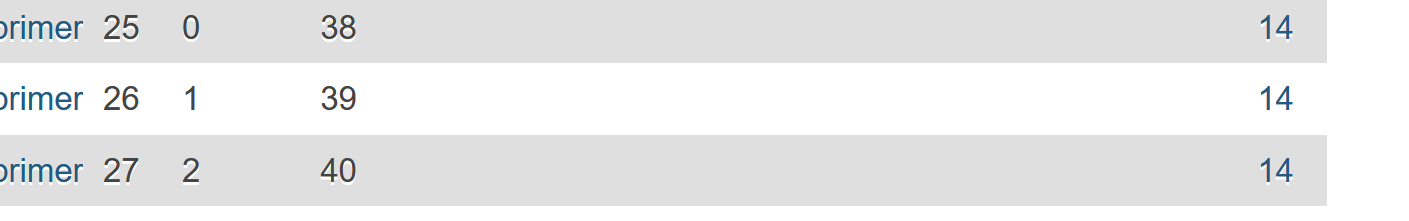


## **Exercice de type « Sériation » :**

Ce type d’exercice permet de ranger des éléments suivant la taille, le poids, la valeur …, pour le faire fonctionner on doit stocker les noms des images dans la table de choix, ces images doivent être stocké dans le dossier **src/Data/Images.**



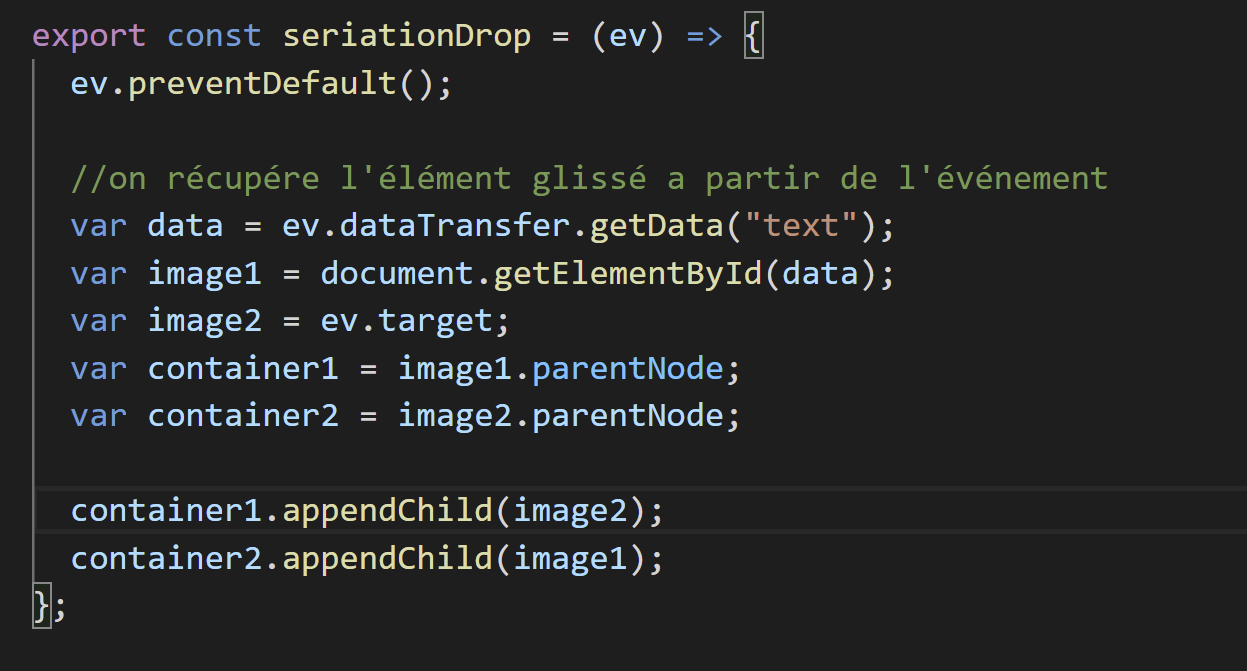
La prochaine étape est de stocker l’ordre et les identifiants des images dans la table réponse.



Pour pouvoir afficher l’image il faut lire le nom de l’image à partir de la base de données et l’insérer par la suite dans le dom donc il faut appeler la méthode require dans la méthode map.



La fonction **seriationDrop** permet de récupérer l’image glissé ainsi que l’image de destination, on récupère par la suite le conteneur de chacune des images, et pour chaque conteneur on définit l’image opposé comme enfant du conteneur.



Pour évaluer le travail de l’enfant il faut vérifier tout d’abord si les positions des images sont bonnes ont comparant avec les valeurs dans la table reponse.

On vérifie par la suite si l’identifiant de chaque image est égale à la valeur stocké dans la table réponse.



# Le design pattern Strategy

Au tout début du projet, un switch-case permettais de définir l’affichage selon le type de l’exercice, et a chaque fois qu’on voulait ajouter un autre type d’exercice, le code du switch-case devenait de plus en plus long.

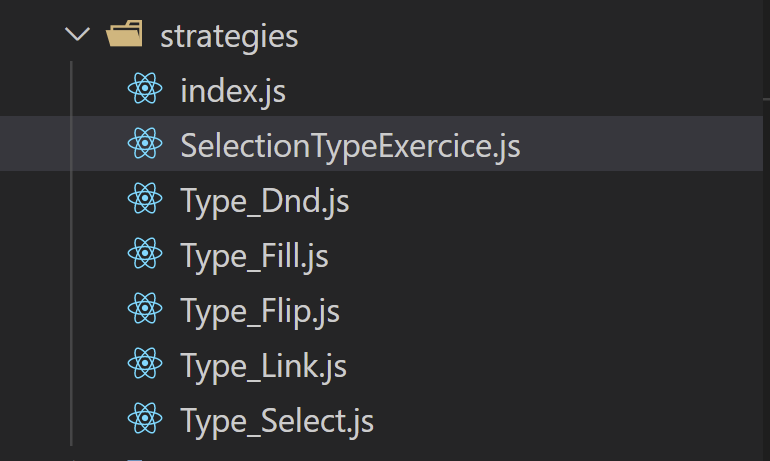
Cette implémentation ne respectait pas le principe open/closed, ce qui rendait le code de plus en plus non maintenable.

La solution était d’implémenter le design pattern Strategy qui prend en compte le type de l’exercice et définit le type d’affichage lors de l’exécution.

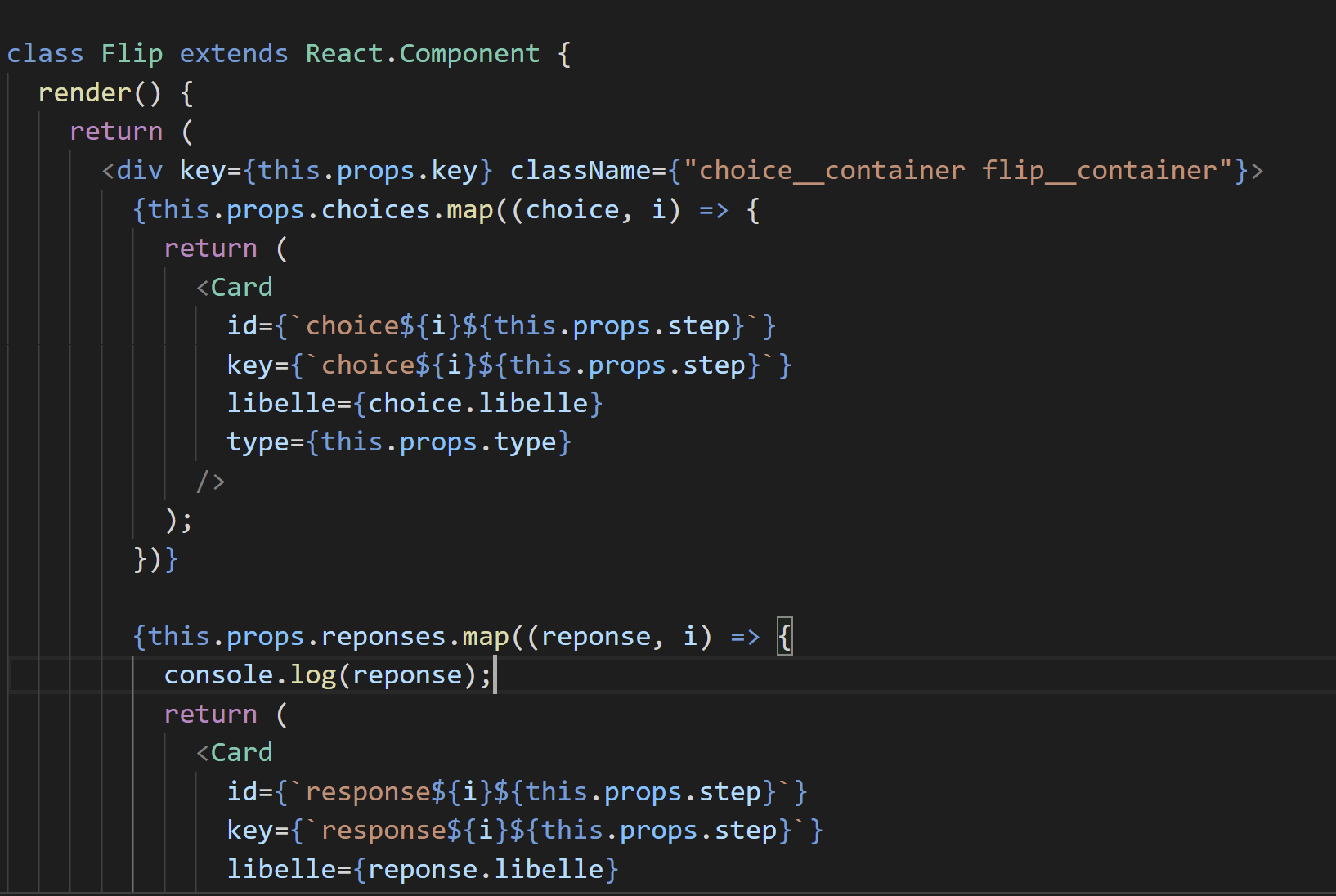
L’implémentation suivante représente la fameuse classe Context du design pattern Strategy, c’est elle qui permet de définir la stratégie au moment de l’exécution et donc d’appeler la méthode **render** adéquate.



L’image suivante montre les différentes stratégies qui représente chacune un type d’exercice.



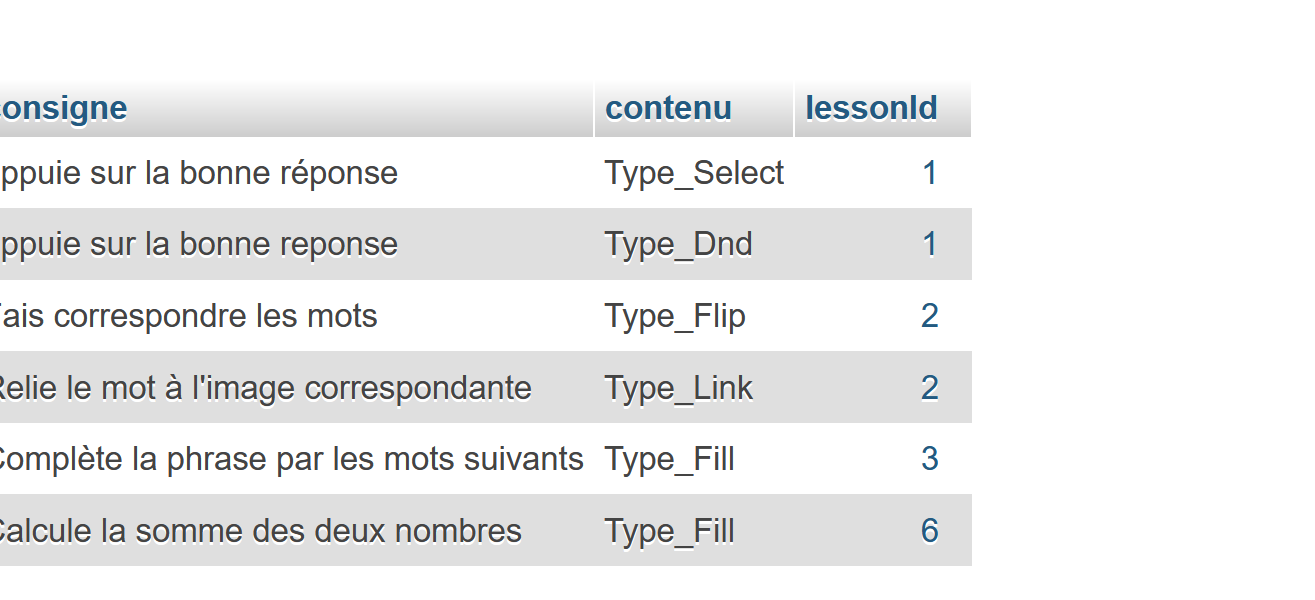
Chaque stratégie implémente la méthode **render** qui est appelé au moment de l’exécution.



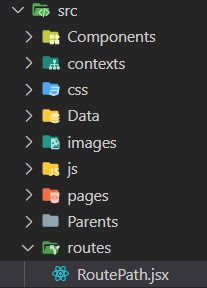
Pour appliquer ce design pattern on initialise tout d’abord le Context **TypeExerciceManager**, et selon le type on récupère la stratégie, il suffit après d’affecter cette dernière au context et d’appeler la méthode render du context.



**PS : il faut que le type d’exercice dans la base corresponde au nom de la stratégie crée.**



# Le routage



Dans le dossier src->routes on trouve un fichier RoutePath.jsx ou l’on trouve toutes les routes qui vont être utile dans l’api.

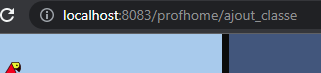
Ce fichier contient les différentes routes qui seront affichées dans la barre d’adresse du navigateur en fonction de la page affichée.

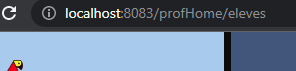
## Le routage du dashboard professeur.

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquementUne image contenant texte

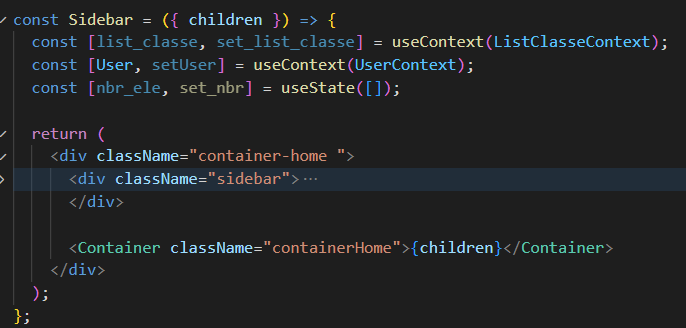
Description générée automatiquementDans le fichier RoutePath.jsx on trouve les différentes routes qui seront utilisées et les éléments qu’elles doivent afficher

Les éléments des routes s’affichent grâce au composant <Outlet />du fichier ProfHome.jsx composant du react-router-dom

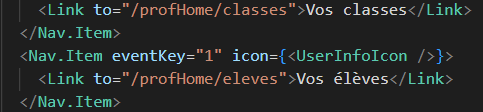
Une image contenant texte

Description générée automatiquement

L’affichage des différentes vues se fait dans le composant <Sidebar/> dans Sidebar.jsx à l’aide de children.



Dans Sidebar.jsx on trouve les liens vers profHome/classes et profHome/eleves à travers des éléments <Link/> de react- router-dom



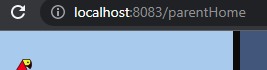
On utilise encore des éléments <Link/> vers /profhome/ajout\_classe et /profhome/ajout\_eleve pour afficher les composants <FormAddEleve /> et <FormAddClasse/>.

## Le routage du dashboard Parent

Une image contenant texte

Description générée automatiquement

On retrouve dans le fichier RoutePath.jsx les différentes routes attachées aux différents éléments nécessaires pour le dashboard Parent.

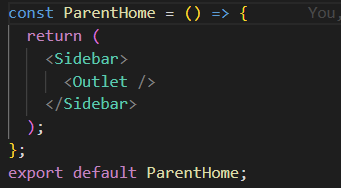


Une image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquement

Comme pour l’écran professeur un utilise le composant <Outlet/> du react-router-dom pour afficher les différentes routes.



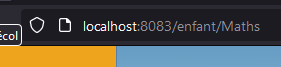
De la même manière dans Sidebar.jsx on utilise children pour afficher les éléments et on trouve des composants <Link/> pour naviguer entre les éléments.

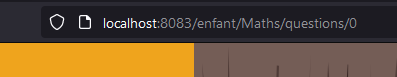
## Le routage du dashboard enfant

Une image contenant texte

Description générée automatiquement

Ici aussi on retrouve les différentes routes associées aux composants qu’elles doivent afficher.



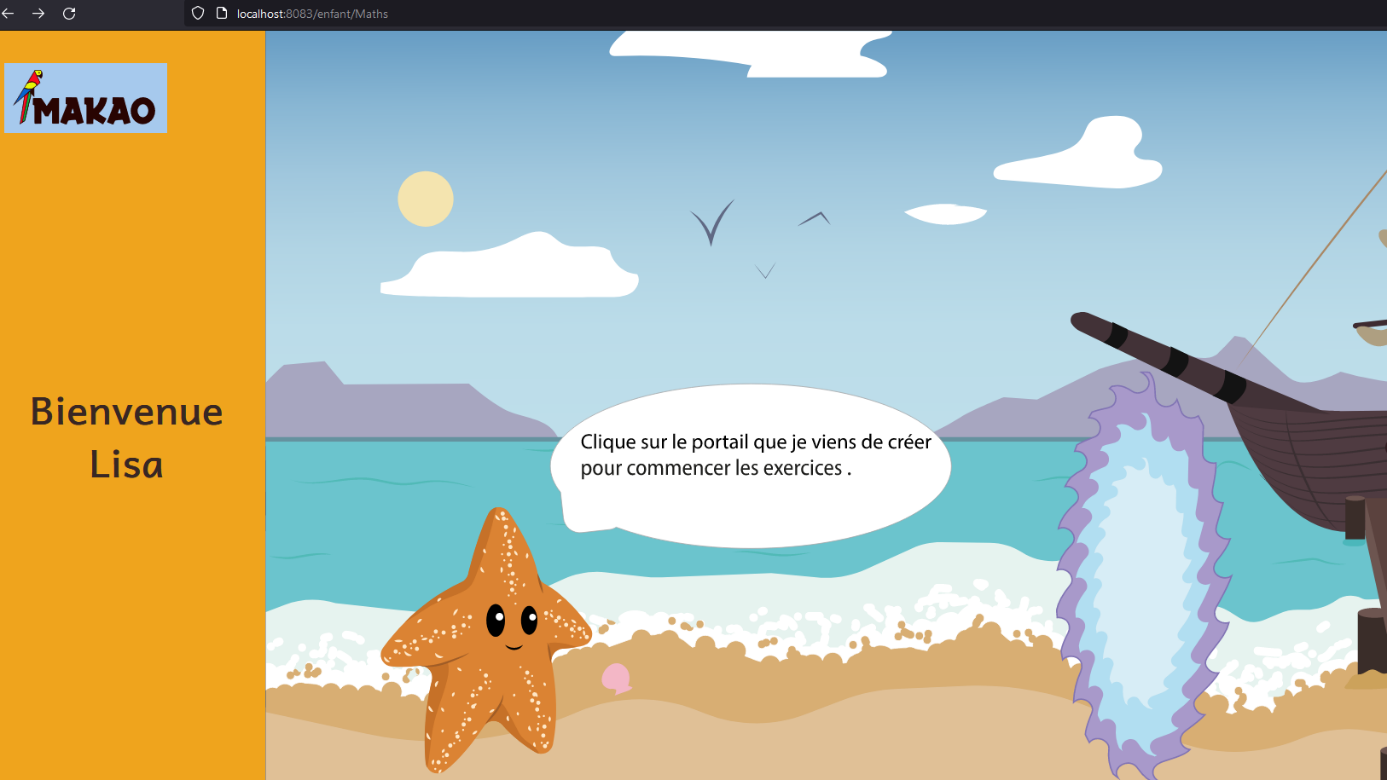


Pour l’affichage de l’écran d’une matière on utilise dans le fichier Home.js la fonction navigate() de react-router-dom en fonction du choix de la matière.

Une image contenant texte

Description générée automatiquement

Cela nous amène à l’écran matière :



Et au fichier exercice.js ou l’on utilise à nouveau la fonction navigate() pour afficher un exercice en fonction de sa matière et de son index.

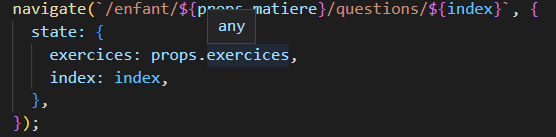
Une image contenant texte

Description générée automatiquement

On met index et matiere dans le state car ces deux variables nous servirons dans la navigation dans le fichier NavBar.js. Il faut penser à les mettre en props dans les composants <Step/> et <Steps/>

Cela nous permet d’afficher le composant <Question/>.

La navigation vers l’exercice suivant se fait grâce au composant <NavBar/> qui est un composant de <Step/> lui-même composant de <Steps/> composant de question

Donc dans le fichier NavBar.js on utilise de nouveau navigate avec en state exercices et index.