You're absolutely right; having a single component, such as Combat.js, handle both player actions and monsters' logic can make the file quite large and potentially less maintainable over time. To keep your codebase organized and maintainable, consider adopting a modular and component-based approach. Here are some strategies you can use to manage your game's logic and maintain code readability:

**Separate Game Logic into Modules:**

Create separate modules or files for different aspects of the game logic. For instance, you can have a module for player actions, another for monsters, and yet another for general game state management.

**Use a State Management Library:**

Consider using state management libraries like **Redux or Mobx** to handle the game's state and actions. These libraries help you manage complex application state in a more organized and scalable manner.

**Component Composition:**

Break down your user interface into smaller, reusable components. For example, you can have a PlayerHand component, a Monster component, and a GameBoard component.

Components are primarily responsible for the user interface, rendering, and handling user interactions. They should remain as focused as possible on the **presentation and view aspects** of your application.

**Service Layer:**

Create a service layer to encapsulate game logic that doesn't necessarily belong in the UI components. This can include actions like card playing, monster interactions, and other game-specific functionality. This approach helps maintain a clear separation of concerns.

Here's an example of how your project structure might look with these suggestions:

Une image contenant texte, capture d’écran, logiciel, ordinateur

Description générée automatiquement

**state Folder:**

The state folder is used for managing application state. It's a common pattern in React applications to centralize the management of various parts of your application's state. This can include data that needs to be shared between different components, user-specific information, game-related data, and more.

**playerState Module:**

The playerState module is one of the state management modules, and it typically contains the state and logic related to the player in your game. Here's what you might find in the playerState module:

**Player Data:**

This module might store player-related data such as player health, mana, player level, and any other relevant player attributes.

**Player Actions and Logic:**

It can include methods or functions for handling player actions and interactions, such as playing cards, taking damage, gaining experience, and other player-specific game mechanics.

**PlayerHand Component:**

The PlayerHand component is responsible for rendering and handling the player's hand of cards. Here's what you might find in the PlayerHand component:

**Rendering the Hand:** The component should render the player's hand of cards. This can be a list of card components, each representing a card in the player's hand.

**Handling Card Actions:**

The component can handle user interactions with the cards, like clicking on a card to play it. When a card is played, it can call the relevant game logic (e.g., invoking the playCard function in your GameLogicService) to apply the card's effects.

Updating UI: It should update the UI to reflect the player's current hand and any changes that occur as a result of playing cards. For example, removing a card from the hand after it's played or updating the player's mana.

The idea is to keep components like PlayerHand focused on their specific responsibilities, such as rendering and user interactions, and delegate game-related logic to the appropriate state management modules like playerState. This separation of concerns makes your code more maintainable and allows for easier testing of individual components and game logic.

**Each user gets the data of his player from the database :**

For implementing a system where each user can save and load their player's data, which includes attributes like health points and level that can vary, you will need to handle both local state (for real-time interactions) and database state (for persistent data storage).

Here's a high-level approach to handle this in your playerState module:

**Local State**: Maintain local state within the playerState module to manage the player's attributes in real-time while they play the game. This state will be used for immediate updates, and you can leverage the useState hook or similar state management solutions for this.

**Database State:** Store the player's data, including their health points, level, and other attributes, in your database. You can create an API or endpoints on your server to save and load the player's data to and from the database. You can use a database solution like MongoDB, which you mentioned you're using, to store this information.

**Syncing Local and Database State:**

When a player interacts with the game and their attributes change (e.g., they level up or lose health), update **the local state immediately for a smooth user experience.**

**Periodically or when specific game events occur (e.g., the end of a level or when the player decides to save their progress), sync the local state with the database.** This involves sending a request to your server API to update the player's data in the database.

Here's an example of how your playerState module might look to handle both local and database state:

**Une image contenant texte, Appareils électroniques, capture d’écran, logiciel

Description générée automatiquement**

**Gérer les state avec REDUX**

*Can I, for instance, fetch some data from the db when the user wants to start the fight, and then manage the state of the values that I fetched locally with Redux ?*

Yes, you can absolutely use Redux to manage the state of data fetched from a database for your game. Here's a general outline of how you can achieve this:

**Fetch Data from the Database**: When the user wants to start a fight or load game-related data, you can make an asynchronous API request to your database using technologies like Axios, Fetch, or a state management library like Redux Thunk or Redux Saga to manage asynchronous actions.

**Dispatch Actions:** Once you've fetched the data, you can dispatch Redux actions to update the game state. For example, you might dispatch actions to store player attributes, enemy attributes, and the available cards.

**Reducers:** Create reducers to handle these actions. Reducers should update the state in response to the dispatched actions. For example, you'd have a reducer for player data, one for enemy data, and one for card data.

**Selectors:** Implement selectors to access and extract specific data from the Redux store. You can use these selectors to access player stats, enemy stats, and card data when rendering the game interface.

**User Interface**: Build your game's user interface to display the fetched and managed data. Connect the UI components to the Redux store using tools like react-redux.

**Game Logic**: Implement game logic based on the data you've fetched and stored in your Redux state. For example, when the player initiates a fight, you can use the data in Redux to determine the outcome of the fight.

This approach allows you to keep the game's data consistent and easily accessible throughout the game, as Redux serves as a centralized data store. Fetching the data from a database initially and then managing it locally with Redux is a common and effective strategy, especially for web-based games. It keeps your data and game state organized and allows for easy updates and interactions.