Sklep – lista

1. Nagłówek HTML
```
html
KopiujEdytuj
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <link href="./style.css" rel="stylesheet" />
  <title>Books</title>

  <!-- Biblioteki React i ReactDOM -->
  <script src="https://unpkg.com/react@18/umd/react.production.min.js"></script>
  <script src="https://unpkg.com/react-dom@18/umd/react-
dom.production.min.js"></script>

  <!-- Babel do przetwarzania JSX (tylko do testów) -->
  <script src="https://unpkg.com/@babel/standalone"></script>
</head>
```

- **DOCTYPE i html lang="en"** — deklaruje język i typ dokumentu.
- **meta charset** i **viewport** — ustawienia kodowania i responsywności.
- **link do style.css** — zewnętrzny plik CSS.
- **title** — tytuł strony.
- Ładowanie bibliotek React (React i ReactDOM) z CDN.
- Ładowanie Babel standalone, by móc pisać JSX bez kompilacji (tylko do testów, nie do produkcji).

---

2. Body i struktura HTML

```
html
KopiujEdytuj
<body>
  <div class="container">
    <h1>Books</h1>
    <div id="root"></div>
  </div>
```

- Prosty kontener z nagłówkiem "Books".
- **div#root** — tutaj React „przyczepi” całą aplikację.

---

3. Skrypt React w JSX (type="text/babel")

```
jsx
KopiujEdytuj
<script type="text/babel">
  const { useState, useEffect } = React;
```

```
  const App = () => {
    ...
  };

  ReactDOM.render(<App />, document.getElementById("root"));
</script>
```

- **useState** i **useEffect** — dwa popularne hooki Reacta.
- Definicja komponentu funkcyjnego **App**.
- Renderowanie **App** do elementu o id "root".

---

4. Wnętrze komponentu `App`

a) Stan komponentu:

```
jsx
KopiujEdytuj
const [items, setItems] = useState([]);
const [search, setSearch] = useState("");
const [sortKey, setSortKey] = useState("name");
const [newItem, setNewItem] = useState({ name: "", description: "", image: "",
rating: 1 });
```

- **items** — lista książek (początkowo pusta).
- **search** — fraza do filtrowania książek.
- **sortKey** — kryterium sortowania ("name" lub "rating").
- **newItem** — obiekt reprezentujący książkę dodawaną przez użytkownika.

---

b) `useEffect` — pobieranie danych z pliku `items.json`

```
jsx
KopiujEdytuj
useEffect(() => {
  fetch("items.json")
    .then((res) => res.json())
    .then((data) => setItems(data))
    .catch((err) => console.error("Error loading items:", err));
}, []);
```

- `useEffect` z pustą tablicą zależności `[]` oznacza, że wykona się tylko raz po zamontowaniu komponentu.
- Pobiera dane z pliku **items.json** (musisz mieć ten plik w folderze z aplikacją).
- Parsuje JSON i ustawia dane jako stan `items`.
- W przypadku błędu wypisuje go w konsoli.

---

c) Funkcje do zarządzania stanem:

Dodawanie nowej książki

```
jsx
KopiujEdytuj
const handleAddItem = () => {
  const id = Date.now(); // unikalne ID na bazie aktualnego czasu
  setItems([...items, { ...newItem, id }]); // dodajemy nową książkę do listy
  setNewItem({ name: "", description: "", image: "", rating: 1 }); // czyścimy
formularz
};
```

- Generuje unikalne id.
- Dodaje nowy element do tablicy `items` (zachowując poprzednie).
- Resetuje `newItem`, aby formularz był pusty.

Usuwanie książki

```
jsx
KopiujEdytuj
const handleDelete = (id) => {
  setItems(items.filter(item => item.id !== id));
};
```

- Usuwa książkę o podanym `id` filtrując listę.

Zmiana oceny książki

```
jsx
KopiujEdytuj
const handleRatingChange = (id, rating) => {
  setItems(items.map(item => item.id === id ? { ...item, rating:
parseInt(rating) } : item));
};
```

- Modyfikuje ocenę książki o podanym `id` na nową wartość.

---

d) Filtrowanie i sortowanie elementów

```
jsx
KopiujEdytuj
const filteredItems = items
  .filter(item => item.name.toLowerCase().includes(search.toLowerCase()) ||
item.description.toLowerCase().includes(search.toLowerCase()))
  .sort((a, b) => {
    if (sortKey === "name") return a.name.localeCompare(b.name);
    if (sortKey === "rating") return b.rating - a.rating;
    return 0;
  });
```

- Filtrowanie po `search`: wyświetla tylko te książki, które w nazwie lub opisie zawierają wpisany tekst (case insensitive).

- Sortowanie według:
    - nazwy (alfabetycznie)
    - lub oceny (malejąco).

---

e) JSX zwracany przez komponent (UI)

```
jsx
KopiujEdytuj
return (
  <div>
    {/* Sekcja wyszukiwania i sortowania */}
    <div>
      <input
        type="text"
        placeholder="Search..."
        value={search}
        onChange={(e) => setSearch(e.target.value)}
      />
      <select onChange={(e) => setSortKey(e.target.value)} value={sortKey}>
        <option value="name">Sort by Name</option>
        <option value="rating">Sort by Rating</option>
      </select>
    </div>

    {/* Formularz dodawania nowej książki */}
    <div className="add-item-form">
      <h3>Add New Item</h3>
      <input
        type="text"
        placeholder="Name"
        value={newItem.name}
        onChange={(e) => setNewItem({ ...newItem, name: e.target.value })}
      />
      <textarea
        placeholder="Description"
        value={newItem.description}
        onChange={(e) => setNewItem({ ...newItem, description:
e.target.value })}
      />
      <input
        type="text"
        placeholder="Image URL"
        value={newItem.image}
        onChange={(e) => setNewItem({ ...newItem, image: e.target.value })}
      />
      <select
        value={newItem.rating}
        onChange={(e) => setNewItem({ ...newItem, rating: e.target.value })}
      >
        {[1, 2, 3, 4, 5].map(n => <option key={n} value={n}>{n} Star{n > 1 &&
's'}</option>)}
      </select>
      <button onClick={handleAddItem}>Add Item</button>
    </div>

    {/* Lista książek */}
    <div className="item-list">
      {filteredItems.map(item => (
        <div className="item-card" key={item.id}>
```

```
          <img src={item.image} alt={item.name} />
          <h4>{item.name}</h4>
          <p>{item.description}</p>
          <select
            value={item.rating}
            onChange={(e) => handleRatingChange(item.id, e.target.value)}
          >
            {[1, 2, 3, 4, 5].map(n => (
              <option key={n} value={n}>{n} Star{n > 1 && 's'}</option>
            ))}
          </select>
          <button onClick={() => handleDelete(item.id)}>Delete</button>
        </div>
      ))}
    </div>
  </div>
);
```

Wyjaśnienie poszczególnych części UI:

- **Wyszukiwarka:** `<input>` pozwala wpisać tekst i aktualizuje `search`.
- **Sortowanie:** `<select>` pozwala wybrać według czego sortujemy (nazwa lub ocena).
- **Formularz dodawania:**
  - Input na nazwę książki
  - Textarea na opis
  - Input na URL obrazka
  - Select na ocenę (1-5)
  - Przycisk, który wywołuje `handleAddItem`.
- **Lista książek:**
  - Dla każdego elementu z `filteredItems` tworzy kartę.
  - Pokazuje obraz, nazwę, opis.
  - Można zmienić ocenę w select.
  - Można usunąć książkę przyciskiem.

Podsumowanie

- To jest aplikacja React do wyświetlania listy książek z możliwością:
  - Pobierania danych z pliku JSON.
  - Wyszukiwania książek po nazwie/opisie.
  - Sortowania po nazwie lub ocenie.

    - Dodawania nowych książek z formularza.
    - Usuwania książek.
    - Zmiany oceny książki.
- Kod jest napisany w JSX, w przeglądarce jest konwertowany przez Babel.
- Stan aplikacji zarządzany jest przy pomocy hooków `useState` i `useEffect`.

Historyjka:
1. Sekcja HTML i nagłówka (head)

```
html
KopiujEdytuj
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link href="style.css" rel="stylesheet">
  <title>Cult</title>

  <script src="https://unpkg.com/react@18/umd/react.production.min.js"></script>
  <script src="https://unpkg.com/react-dom@18/umd/react-
dom.production.min.js"></script>
  <script src="https://unpkg.com/@babel/standalone"></script>
</head>
```

- `<!DOCTYPE html>` — standardowa deklaracja mówiąca, że to dokument HTML5.
- `<html lang="en">` — element HTML z atrybutem języka.
- `<meta charset="UTF-8">` — ustawienie kodowania znaków.
- `<meta name="viewport"...>` — ustawienie responsywności na urządzeniach mobilnych.
- `<link href="style.css" rel="stylesheet">` — podłączenie zewnętrznego pliku CSS (stylów).
- `<title>Cult</title>` — tytuł strony wyświetlany w karcie przeglądarki.
- Następnie trzy skrypty:
  - React 18 w wersji produkcyjnej
  - ReactDOM 18 — do renderowania Reacta w DOM
  - Babel standalone — pozwala pisać JSX bez konieczności budowania projektu (kompiluje JSX w przeglądarce)

---

2. Sekcja body

```
html
KopiujEdytuj
<body class="bg-gray-900 text-white">
  <div id="root"></div>
```

- `<body>` ma klasy `bg-gray-900` i `text-white` — to zapewne klasy z Tailwind CSS, czyli ciemne tło i biały tekst.
- `<div id="root"></div>` — tutaj React wyrenderuje całą aplikację.

---

3. Skrypt React w JSX (typ `text/babel`)

```
jsx
KopiujEdytuj
<script type="text/babel">
  const { useState, useEffect } = React;
```

- Destrukturyzacja hooków Reactowych — `useState` i `useEffect` są później wykorzystywane.

---

4. Komponent `Cult`

```
jsx
KopiujEdytuj
const Cult = () => {
  const [story, setStory] = useState(null);
  const [sceneKey, setSceneKey] = useState(null);
  const [stats, setStats] = useState({ strength: 0, agility: 0 });
  const [result, setResult] = useState(null);
```

- `story` — przechowuje całą historię z pliku `story.json`.
- `sceneKey` — klucz aktualnej sceny w historii.
- `stats` — stan postaci (siła i zręczność).
- `result` — wynik walki (success/failure).

---

5. Ładowanie historii z pliku JSON

```
jsx
KopiujEdytuj
useEffect(() => {
  fetch("story.json")
    .then((res) => res.json())
    .then((data) => {
      setStory(data);
      setSceneKey(data.start);
    })
    .catch((err) => console.error("Error loading story:", err));
}, []);
```

- Po pierwszym renderze (pusty array jako drugi argument `useEffect`) pobieramy plik `story.json`.
- Po poprawnym załadowaniu:
  - ustawiamy `story` na dane z JSON
  - ustawiamy `sceneKey` na `data.start` — czyli na startową scenę (np. `"intro"`).
- W przypadku błędu wyświetlamy go w konsoli.

---

6. Sprawdzanie stanu i ładowanie sceny

```
jsx
KopiujEdytuj
if (!story || !sceneKey || !story.scenes[sceneKey]) {
  return <div>Loading...</div>;
}
```

- Jeśli nie ma jeszcze danych historii, lub klucza sceny, lub scena nie istnieje — wyświetlamy "Loading...".

---

7. Pobieramy obiekt aktualnej sceny

```
jsx
KopiujEdytuj
const scene = story.scenes[sceneKey];
```

- Z obiektu `story.scenes` bierzemy obiekt aktualnej sceny.

---

8. Opcjonalna scena — losowy warunek

```
jsx
KopiujEdytuj
if (scene.optional !== undefined) {
  if (Math.random() > scene.optional) {
    setSceneKey(scene.next);
    return null;
  }
}
```

- Jeśli scena ma właściwość `optional` (liczbę od 0 do 1), to losujemy liczbę.
- Jeśli wylosowana jest większa niż `optional`, to przechodzimy automatycznie do następnej sceny (klucz `scene.next`).
- `return null;` powoduje, że komponent nic nie renderuje w tym momencie.

---

9. Funkcja `applyGains`

```
jsx
KopiujEdytuj
const applyGains = (gain = {}) => {
  setStats((prev) => ({
    strength: prev.strength + (gain.strength || 0),
    agility: prev.agility + (gain.agility || 0),
  }));
};
```

- Funkcja aktualizuje statystyki, dodając wartości podane w `gain`.
- Jeżeli `gain` nie zawiera np. `strength`, to dodajemy 0.

---

10. Obsługa wyboru gracza

```
jsx
KopiujEdytuj
const handleChoice = (nextKey, gain) => {
  if (gain) applyGains(gain);
  setSceneKey(nextKey);
};
```

- Po wyborze (kliknięciu) użytkownika:
  - Jeżeli są statystyki do dodania (`gain`), to je zastosuj.
  - Ustaw nową scenę na `nextKey`.

---

11. Obsługa walki (battle)

```
jsx
KopiujEdytuj
const handleBattle = () => {
  const totalPower = stats.strength + stats.agility;
  const successChance = totalPower >= 4 ? 0.8 : 0.3;
  const outcome = Math.random() < successChance;
  setResult(outcome ? "success" : "failure");
};
```

- Obliczamy łączną siłę (`strength + agility`).
- Jeżeli jest większa lub równa 4 — mamy 80% szans na sukces.
- Jeśli mniej — tylko 30%.
- Losujemy wynik, ustawiamy `result` na `"success"` albo `"failure"`.

---

12. Wyświetlanie wyniku walki (jeśli jest)

```
jsx
KopiujEdytuj
if (result) {
  return (
    <div className="p-12 max-w-xl mx-auto">
      {result === "success" ? (
        <p className="text-center">
          ...duży tekst opisujący zwycięstwo...
        </p>
      ) : (
        <p className="text-center center-box-single">
          ...tekst przegranej...
        </p>
```

```
      )}
    </div>
  );
}
```

- Jeśli `result` jest ustawiony, wyświetlamy tekst opowiadający o sukcesie lub porażce w walce.

---

13. Renderowanie sceny

```
jsx
KopiujEdytuj
return (
  <div className="p-12 max-w-xl mx-auto bg-gray-900 text-white rounded-xl
shadow-xl">
    <p className="mb-12 text-center">{scene.text}</p>
```

- Kontener ze stylem, w którym pokazujemy tekst sceny (`scene.text`).

---

14. Renderowanie wyborów

```
jsx
KopiujEdytuj
{scene.choices && (
  <div className="space-y-2 center-box">
    {scene.choices.map((choice, i) => (
      <button
        key={i}
        className="p-2 bg-blue-600 rounded hover:bg-blue-500"
        onClick={() => handleChoice(choice.next, scene.gain)}
      >
        {choice.text}
      </button>
    ))}
  </div>
)}
```

- Jeśli scena ma wybory (`choices`), wyświetlamy je jako przyciski.
- Każdy przycisk po kliknięciu wywołuje `handleChoice` z kluczem następnej sceny i zyskiem `scene.gain` (czyli dodaniem statystyk).

---

15. Renderowanie przycisku "Continue"

```
jsx
KopiujEdytuj
{scene.next && !scene.choices && (
  <div className="space-y-2 center-box">
    <button
      className="mt-12 p-2 bg-green-600 rounded hover:bg-green-500"
      onClick={() => handleChoice(scene.next, scene.gain)}
    >
```

```
      Continue
    </button>
  </div>
)}
```

- Jeśli scena ma `next` (następną scenę), ale nie ma wyborów, pokazujemy jeden przycisk "Continue" do przejścia dalej.

---

16. Renderowanie przycisku walki

```
jsx
KopiujEdytuj
{scene.battle && (
  <div className="space-y-2 center-box">
    <button
      className="mt-12 p-2 bg-red-600 rounded hover:bg-red-500"
      onClick={handleBattle}
    >
      Face the Cult Leader
    </button>
  </div>
)}
```

- Jeśli scena zawiera pole `battle`, pokazujemy przycisk do rozpoczęcia walki.

---

17. Zakomentowany fragment pokazujący statystyki

```
jsx
KopiujEdytuj
{/*<div className="mt-6 text-center">
  <p><strong>Strength:</strong> {stats.strength}</p>
  <p><strong>Agility:</strong> {stats.agility}</p>
</div>*/}
```

- Komentarz, który gdyby odkomentować, pokazywałby aktualne statystyki.

---

18. Renderowanie komponentu

```
jsx
KopiujEdytuj
ReactDOM.render(<Cult />, document.getElementById("root"));
```

- Na końcu aplikacja React jest renderowana do `div#root`.

Podsumowanie

Ten kod to **prosta gra tekstowa napisana w React**:

- Ładuje historię z pliku `story.json`.
- Każda scena ma tekst, opcjonalnie wybory, dalsze sceny, walkę, lub zyski statystyk.
- Gracz podejmuje decyzje, które zmieniają stan (scenę i statystyki).
- W pewnym momencie może nastąpić walka z prostą mechaniką sukcesu opartą na statystykach.
- Po walce pokazuje się opis zwycięstwa lub porażki.