D'accord, avec ta structure **backend**, on remarque que les modèles `EventRoom`, `EventBus` et `EventMaterial` héritent d'un modèle abstrait `Event`, avec des champs spécifiques pour chaque type d'événement.

👉 **Cela renforce l'idée d'un composant frontend unique** (`AjouterEvent.jsx`) qui gère tous ces types d'événements en fonction des propriétés dynamiques passées (`type`).

📌 Adaptation du formulaire en React

Ton backend montre des différences **claires** entre chaque type d'événement, donc ton formulaire doit s'adapter dynamiquement :

- **Champ `resource`**
  - `EventRoom` → Sélectionner une `Room`.
  - `EventBus` → Sélectionner un `Bus`.
  - `EventMaterial` → Sélectionner un `Material`.
- **Champs spécifiques :**
  - `EventBus` → `available_seats`, `departure`, `destination`, `departure_time`, `arrival_time`.
  - `EventMaterial` → `available_stock`.

📌 Implémentation du composant générique `AjouterEvent.jsx`

```
import { useState, useEffect } from "react";

function AjouterEvent({ type, onSubmit }) {
  const [formData, setFormData] = useState({
    description: "",
    organizer: "",
    resource: "",
    ...(type === "bus" && {
      available_seats: "",
      departure: "",
      destination: "",
      departure_time: "",
      arrival_time: "",
    }),
    ...(type === "material" && { available_stock: "" }),
  });

  const [resources, setResources] = useState([]);

  useEffect(() => {
    // Charger les ressources spécifiques en fonction du type
    fetch(`/api/${type}-resources/`)
      .then((res) => res.json())
```

```
      .then((data) => setResources(data))
      .catch((err) => console.error("Erreur de chargement :", err));
  }, [type]);

  function handleChange(event) {
    setFormData({ ...formData, [event.target.name]: event.target.value });
  }

  function handleSubmit(event) {
    event.preventDefault();
    onSubmit(formData);
  }

  return (
    <form onSubmit={handleSubmit}>
      <h2>Ajouter un {type}</h2>
      <input
        type="text"
        name="description"
        placeholder="Description"
        value={formData.description}
        onChange={handleChange}
        required
      />

      <select name="resource" value={formData.resource} onChange=
{handleChange} required>
        <option value="">Sélectionner une ressource</option>
        {resources.map((res) => (
          <option key={res.id} value={res.id}>
            {res.name}
          </option>
        ))}
      </select>

      {type === "bus" && (
        <>
          <input type="number" name="available_seats" placeholder="Places
disponibles"
            value={formData.available_seats} onChange={handleChange}
required />
          <input type="text" name="departure" placeholder="Départ"
            value={formData.departure} onChange={handleChange} required />
          <input type="text" name="destination" placeholder="Destination"
            value={formData.destination} onChange={handleChange} required
/>
          <input type="datetime-local" name="departure_time"
            value={formData.departure_time} onChange={handleChange}
required />
          <input type="datetime-local" name="arrival_time"
            value={formData.arrival_time} onChange={handleChange} />
        </>
      )}
```

```
      {type === "material" && (
        <input type="number" name="available_stock" placeholder="Stock
disponible"
          value={formData.available_stock} onChange={handleChange} required
/>
      )}

      <button type="submit">Ajouter</button>
    </form>
  );
}

export default AjouterEvent;
```

📌 Utilisation du composant

```
import AjouterEvent from "./AjouterEvent";

function AjouterBusEvent() {
  return <AjouterEvent type="bus" onSubmit={(data) => console.log(data)}
/>;
}

function AjouterRoomEvent() {
  return <AjouterEvent type="room" onSubmit={(data) => console.log(data)}
/>;
}

function AjouterMaterialEvent() {
  return <AjouterEvent type="material" onSubmit={(data) =>
console.log(data)} />;
}
```

✅ Pourquoi c’est la bonne approche ?

1️⃣ **Un seul composant flexible**, réduit la duplication de code. 2️⃣ **Réutilisation des mêmes hooks (`useState`, `useEffect`)**, simplifiant la gestion des ressources (`Room`, `Bus`, `Material`). 3️⃣ **Scalabilité facile**, tu peux ajouter d’autres types d’événements sans modifier toute la structure.