TO-DO WEBSITE DOKUMENTATION
BEREICHE KOMPONENTE
VON
SRIBRIDDHI BANERJEE

1. Projektüberblick

Diese Dokumentation beschreibt die Umsetzung der Kategorienverwaltung im Rahmen einer Todo-App. Inhalt sind Funktion, Einbindung ins Gesamtprojekt, Quellcode, zugrundeliegende Ideen, Entscheidungen sowie Backend-Anbindung.

2. Frontend – CategoriesComponent

2.1 Funktion & Zielsetzung

Die Komponente ermöglicht das Anlegen, Anzeigen, Bearbeiten, Löschen und Navigieren von Kategorien („Bereiche“) innerhalb der Anwendung. Sie bietet eine visuell ansprechende Karussellansicht, intuitive Nutzerfeedbacks via Popover und modulare Struktur durch Separation von UI, Logik und Services.

2.2 Einbindung ins Projekt

- Implementiert als Standalone Angular Component.
- Eingebunden in das Modul mit Imports:
  - **FormsModule** (für ngModel)
  - **RouterModule** (für Navigation)
  - **CommonModule** (für gängige Direktiven)
  - **PopupComponent** (für die Bearbeitung).

- Kommuniziert mit:
  - **UserService** (Aktueller Nutzer)
  - **CategoriesService** (CRUD-Operationen)
  - **TodoService** (Handling von Todos)
  - **TodosComponent** (Nachladen der Todos bei Navigation)
  - Dem Routing System (Router).

2.3 Komponentenstruktur & Methoden

Wesentliche Logik:

- **loadBereiche():** Lädt Kategorien vom Server für den aktuellen Nutzer.
- **addBereiche()**: Fügt neue Kategorie hinzu, aktualisiert UI und zeigt Popover zur Bestätigung.
- **goToTodos():** Navigiert zur Todo-Ansicht und ruft **TodosComponent.loadTodosByBereichId()** auf.
- editCategories() / handleBereichEdit(): Verwaltet Edit-Workflow via **PopupComponent** und **PUT**-Request.
- **Karussell**: **prevSlide(), nextSlide()** navigieren in 3-er-Schritten.
- **deletedCategories():** Prüft vorhandene Todos, zeigt ggf. Popover zur Bestätigung oder löscht direkt. **showDeletePopover()** implementiert die Popover-Logik inkl. Event-Handler.

2.3.1 UI & Template (.html)

- Eingabezeile mit Kategorie-Textfeld + Buttons
- Karussell mit ***ngFor** zeigt max. 3 Kategorien
- Edit-/Delete-Buttons mit **$event.stopPropagation()**
- **<app-popup>** zur Übergabe von Bearbeitungsergebnissen

2.3.2 Styling (.css)

- Flexbox-Karussell: **.carousel-container, .carousel-track, .slide**
- Transition-Effekte, responsive Layouts
- Button-Designs: **.btn2, .back-btn, .carousel-button** mit Hover-State
- Harmonische Farbgestaltung via CSS-Variablen

3. Frontend – CategoriesService

```typescript
export class CategoriesService {
  private http = inject(HttpClient);
  private apiUrl = 'https://todobackend-dupl0s-janniks-projects-e7141841.vercel.app/sections';

  constructor(private todoService: TodoService) { }

  getBereiche(userid: string): Observable<Bereich[]> {
    return this.http.get<{ sections: Bereich[] }>(this.apiUrl,
      {
        params: { userid: userid }
      })
      .pipe(
        map(response => response.sections || [])
      );
  }
  addBereich(name: string, userid: string): Observable<Bereich> {
    const body = { name, userid };
    return this.http.post<{ section: Bereich }>(this.apiUrl, body)
      .pipe(
        map(response => response.section)
      );
    }
  deleteBereich(id: number): Observable<void> {
    return this.http.delete<void>(`${this.apiUrl}/${id}`);
  }
  handleUpdate(bereich: Bereich): Observable<Bereich> {
    return this.http.put<{ section: Bereich }>(this.apiUrl + '/' + bereich.id, bereich)
      .pipe(
        map(response => response.section)
      );
  }
  todosInBereich(bereichId: number): boolean {
    const todos = this.todoService.loadTodos();
    return todos.some(todo => todo.bereichsID === bereichId);
  }
  todosInBerech(bereichId: number): boolean {
    const todos = this.todoService.loadTodos();
    return todos.some(todo => todo.bereichsID === bereichId); // checks if there is at least 1 element
  }
}
```

Funktionen:

- Verbindet die Komponente mit dem Backend via **HTTP-Aufrufen** (GET, POST, PUT, DELETE).
- t**odosInBereich()**: lokale Prüfung der Existenz von Todos für Löschlogik.

4. Backend – Express-Endpunkte

```
185  app.get("/sections", async (_req, res) => {
186    const userid = _req.query.userid as string;
187    try {
188      const allSections = await db.select().from(sections).where(eq(sections.userid, userid));
189      res.json({ sections: allSections });
190    } catch (error) {
191      res.status(500).json({ message: "DB error", error: error.message });
192    }
193  });
194
195  app.post("/sections", async (req: Request, res: Response) => {
196    const { name, userid } = req.body;
197    console.log("Name:", name);
198    console.log("UserID:", userid);
199    if (!userid) {
200      res.status(400).json({ error: "userid fehlt im Body" });
201    }
202    try {
203      const inserted = await db.insert(sections).values({ name, userid }).returning();
204      res.status(201).json({ section: inserted[0] });
205    } catch (error) {
206      res.status(500).json({ message: "DB error", error: error.message });
207    }
208  });
209
210  app.put("/sections/:id", async (req: Request, res: Response) => {
211    const { id } = req.params;
212    const { name } = req.body;
213    try {
214      const updated = await db.update(sections)
215        .set({ name })
216        .where(eq(sections.id, Number(id)))
217        .returning();
218      res.status(200).json({ section: updated[0] });
219    } catch (error) {
220      res.status(500).json({ message: "DB error", error: error.message });
221    }
222  });
```

```
224  app.delete("/sections/:id", async (req: Request, res: Response) => {
225    const { id } = req.params;
226    try {
227      const deleted = await db.delete(sections).where(eq(sections.id, Number(id))).returning();
228      res.status(200).json({ message: "Section deleted", section: deleted[0] });
229    } catch (error) {
230      res.status(500).json({ message: "DB error", error: error.message });
231    }
232  });
233
234  // Export für Vercel
235  export default app;
```

Prinzipien:

- REST-konform mit klaren Statuscodes (200, 201, 400, 500)
- JSON-Antworten (`{ sections: […] }`, `{ section: … }`)
- Parametrisierung via `userid` bzw. `id`
- Fehlerbehandlung über Status und informative JSON-Nachrichten

5. Technische & Architekturentscheidungen

1. Stand-alone Komponenten: Verbessern Testbarkeit, Modularität und vermeiden Module-Bloat.
2. Popovers statt Alerts: Für bessere Nutzererfahrung durch kontextuelle Feedbacks.
3. Popup-Komponente für Edit-Funktion: Separiert UI-Logik von component logic.
4. Optimistische UI-Updates: Änderungen am UI ohne komplettes Nachladen.
5. Löschschutz bei vorhandenen Todos: Sicherheitsmechanismus via Confirmation-Popover.
6. Slider mit manueller Fensterbewegung: UX-Optimierung bei vielen Kategorien.
7. **stopPropagation** bei Buttons: verhindert unerwünschtes Navigieren beim Klick auf Edit/Delete.
8. Datenkapselung via Service: Klare Trennung zwischen UI und Datenzugriff.
9. Fehlermeldungen für Entwickler: Alle Fehlerfälle sind protokolliert und sichtbar.

7. Fazit

Diese Dokumentation bietet:

- Eine vollständige Beschreibung der Funktionalität frontendseitig und backendseitig.
- Technische Einbettung ins Gesamtprojekt („Einbindung“).
- Code-Darstellung für wichtige Methoden & Endpunkte.
- Hintergründe zu Entscheidungen & Architektur.