Architektur Frontend (kolla_vue)

Stand: 07.01.2026

1. Zweck & Kontext

Dieses Repository enthält das Frontend der Web-Applikation „Kolla“. Ziel ist Workflow-/Aufgabenmanagement mit sequenziellen Arbeitsschritten, Rollen-/Berechtigungslogik, persönlicher Aufgabenliste, Priorisierung und einer Workflow-Manager-Sicht.

Wichtige Domänenbegriffe (wie sie im Code benutzt werden):

- **Workflow**: Im Backend als **Objective** modelliert (`/Objective/*`), im Frontend als `Workflow` in `src/types/domain.ts`.
- **WorkStep/Arbeitsschritt**: Im Backend als **Assignment** modelliert (`/Assignment/*`), im Frontend als `WorkStep`.
- **Actor**: Backend-Entität für Benutzer (und perspektivisch Systemakteure). Wird im Frontend häufig zu `User` gemappt.
- **Role**: Backend-Rolle (`RoleDto`), im Frontend zusätzlich ein vereinfachtes Enum `Role`.

Quelle fachlicher Anforderungen: `anforderungen.md`.

2. Tech-Stack & Tooling

Runtime / Frameworks

- **Vue 3** (`vue`) mit **Composition API** und `<script setup lang="ts">`
- **Vue Router 4** (`vue-router`) für SPA-Routing
- **Pinia** (`pinia`) als globales State-Management
- **@vuepic/vue-datepicker** für Datumseingaben

Build / Dev

- **Vite** als Dev-Server und Bundler
- TypeScript via **vue-tsc** (Type-Checking) und `tsconfig.*`
- Node Engine: `^20.19.0 || >=22.12.0` (`package.json`)

Qualitätssicherung

- **ESLint 9** (Flat Config) + `eslint-plugin-vue` + `@vue/eslint-config-typescript`
- **Prettier** (Formatierung)
- **Vitest** + **@vue/test-utils** + **jsdom** (Unit-Tests)

3. Architektur-Überblick

Das Frontend folgt einer klaren **Schichten-/Layering**-Struktur, kombiniert mit einem **MVVM-ähnlichen Ansatz** in Vue:

- **View/UI Layer**: `src/views/*` und Vue-Komponenten in `src/components/*`

- **ViewModel/Use-Case Layer**: `src/composables/*` (Business-Logik, Orchestrierung, Ladezustände)
- **Model/State Layer**: `src/stores/*` (Pinia Stores, reaktive Domänenzustände, Ableitungen)
- **Data Access Layer**: `src/services/api/*` (API-Client + Services pro Backend-Ressource + Mapper)
- **Domain/Types**: `src/types/*` (Domain-Modelle, DTOs, Autorisierungstypen)

Zusätzlich:

- **Design System** über CSS-Variablen (`src/assets/design-system.css`) und Theme-Prototypen (`src/composables/useThemePrototype.ts`).

3.1 Zentrale Architekturentscheidungen (ADR-Style)

1. **Vue 3 + Composition API**

- Entscheidung: Komponentenlogik in Composables (`use*`) kapseln.
- Motivation: Testbarkeit, Wiederverwendung, klare Trennung UI/Logik.

2. **Pinia als Single Source of Truth**

- Entscheidung: Domänenzustand in Stores (`user`, `workflow`, `workStep`, `notification`).
- Motivation: globale Reaktivität, einfache Ableitungen (computed), konsistente Datenflüsse.

3. **API-Layer als Services + Mapper**

- Entscheidung: API-Aufrufe über `ApiClient` + Service-Klassen (z. B. `WorkflowApiService`, `WorkStepApiService`).
- Motivation: klare Abgrenzung von HTTP/DTOs, zentrale Fehlerbehandlung, Testbarkeit.

4. **Dependency Injection (DI) für API-Services**

- Entscheidung: `ApiServicesKey` (`src/composables/useApi.ts`) + `app.provide()` in `src/main.ts`.
- Motivation: Austauschbarkeit/Mocking (Tests, alternative Backends), entkoppelte Composables.

5. **SPA Deploy auf GitHub Pages**

- Entscheidung: Vite `base` abhängig von `NODE_ENV` (`vite.config.ts`) und `postbuild`-Script `scripts/copy-404.js`.
- Motivation: Direktnavigation/Reload auf Subrouten via `404.html` Fallback.

4. Verzeichnisstruktur (relevant)

- `src/main.ts`: App-Bootstrap (Pinia, Router, DI, Restore-User, Theme)
- `src/App.vue`: Shell/Layout + Top-Navigation + Notification UI + Theme Toggle
- `src/router/index.ts`: Routen + Auth-/RBAC-Guard + User-Restore auf Reload
- `src/views/*`: Seiten (Login, Dashboards, Management)
- `src/components/containers/*`: „Smart/Container"-Komponenten (Daten laden, State halten, Modals/Forms)
- `src/components/presenters/*`: „Dumb/Presenter"-Komponenten (UI-only, Props/Events)

- `src/composables/*`: ViewModels/Use-Cases (load/create/update, Orchestrierung)
- `src/stores/*`: Pinia Stores (State + Getter + Actions)
- `src/services/api/*`: API-Client + Services + Mapper
- `src/services/authorization/*`: lokale Autorisierungslogik
- `src/types/*`: Domain & DTO Typen

5. UI-Architektur: Komponenten & Patterns

5.1 Container/Presenter Pattern

Die Komponenten sind in zwei Rollen unterteilt:

- **Container** (`src/components/containers/*`):
  - kümmern sich um Laden/Mutationen (via Composables), Dialogsteuerung, Aggregationen
  - reichen Daten als Props an Presenters
  - emittieren Events nach oben
- **Presenter** (`src/components/presenters/*`):
  - reine Darstellung
  - Props-in, Events-out

Beispiele:

- `WorkflowListContainer.vue` (Container) + `WorkflowCard.vue` (Presenter)
- `ObjectiveListContainer.vue` (Container) + `ObjectiveList.vue`/`ObjectiveCard.vue` (Presenter)
- `WorkStepBoard.vue`/`WorkStepCard.vue` als UI-Bausteine im Actor Dashboard

5.2 Styling & Design System

- Kein Tailwind; Styling erfolgt über **CSS Custom Properties** und BEM-ähnliche Klassennamen.
- Zentrale Tokens/Theme-Variablen in `src/assets/design-system.css`.
- Theme-Switching über `data-theme` und „Theme Prototypes“ (`useThemePrototype.ts`) + Persistenz in `localStorage`.

6. Routing, Navigation & Zugriffskontrolle

6.1 Routen

Definiert in `src/router/index.ts` (Lazy Loading für viele Views):

- `/login` (public)
- `/` (Home, requiresAuth)
- `/my-work-steps` (Actor Dashboard, requiresAuth)
- `/workflow-manager` (Workflow Manager Dashboard, requiresAuth)
- `/workflows` (requiresAuth)
- `/about` (requiresAuth)
- `/roles` (requiresAuth + requiresAdmin)
- `/users` (requiresAuth + requiresAdmin)
- `/users/:id` (requiresAuth, zusätzliche Einschränkung für Non-Admin)

6.2 Router Guard (Auth & RBAC)

- Guard versucht bei Reload **User aus `localStorage` wiederherzustellen**, falls Store noch nicht authentifiziert ist.
- RBAC:
  - `requiresAuth`: Redirect nach `/login` wenn nicht authentifiziert.
  - `requiresAdmin`: Non-Admin wird auf `/my-work-steps` umgeleitet.
  - Route `actor-details`: Non-Admin darf nur die eigene `:id` öffnen.

Persistenz:

- `useUserStore().setCurrentUser()` schreibt `currentUser` nach `localStorage`.

7. State Management (Pinia Stores)

7.1 `user` Store (`src/stores/user.ts`)

- Hält `currentUser` + `users`.
- Computed: `isAuthenticated`, `isAdmin`, `isWorkflowManager`, `isTeamMember`.
- Persistenz: `currentUser` in `localStorage` (primär `actorGuid`).

7.2 `workflow` Store (`src/stores/workflow.ts`)

- Hält `workflows` + `currentWorkflowId`.
- Getter: `currentWorkflow`, `workflowCount`.

7.3 `workStep` Store (`src/stores/workStep.ts`)

- Hält `workSteps` + `currentWorkStepId`.
- Getter: `prioritizedWorkSteps` (Sortierung nach Priority + Sequenz).
- Enthält eine **Prioritätslogik** auf Basis Workflow-Deadline und *Summe verbleibender Dauer* (Remaining Duration) – inkl. manual override (`manualPriority`).

7.4 `notification` Store (`src/stores/notification.ts`)

- In-Memory Notifications mit `unreadCount`, `markAsRead`, `markAllAsRead` etc.
- Hinweis: Aktuell keine Backend-Persistenz (nur Frontend-State).

8. Composables (ViewModel/Use-Case Layer)

Composables bündeln Business-Logik und orchestrieren API + Stores.

Wichtige Beispiele:

- `useApi()`: Liefert DI-injizierte API-Services (Fallback: default).
- `useWorkflow()`: load/create/update/delete Workflows, schreibt in `workflow` Store.
- `useWorkStep()`: load/create/update/complete WorkSteps (inkl. Benachrichtigungen, Auto-Assign).
- `useWorkflowManager()`: Fortschritt/Deadline-Tracking (`WorkflowProgress`), manuelles Setzen von Priorität.
- `useAuthorization()`: Zugriffskontrolle über `AuthorizationService`.
- `useNotification()`: Filtert Notifications pro User.

- `useTheme()`/`useThemePrototype()`: Theme Umschaltung + Persistenz.

Konvention:

- Composables exposen i. d. R. `loading` + `error` und Actions.

9. API-Integration & Abhängigkeiten

9.1 API Base URL & Proxy

- Dev: `baseURL = '/api'` (Proxy via Vite `server.proxy` → `https://kolla-cdb6b0d315ac.herokuapp.com`)
- Prod: `import.meta.env.VITE_API_BASE_URL` oder Fallback Heroku.
- Relevante Dateien: `vite.config.ts`, `src/services/api/index.ts`.

9.2 API Client

`src/services/api/client.ts`:

- Wrapper um `fetch()`
- Einheitliche Header (`Content-Type: application/json`)
- Timeout via `AbortSignal.timeout()`
- Fehlerbehandlung inkl. CORS/Network Messaging
- Unterstützt JSON und GUID-Responses als Text

9.3 Service-Klassen pro Ressource

`src/services/api/*`:

- `WorkflowApiService`: nutzt `/Objective/*` (Backend-Term „Objective“)
- `WorkStepApiService`: nutzt `/Assignment/*` + z. T. `/Actor/*` und `/Objective/GetAllAssignments/{workflowId}`
- `AssignmentApiService`: kapselt Assignment Endpoints und Client-Side Validierung (Datum, Basic Constraints)
- `ActorApiService`: `/Actor/*`
- `RoleApiService`: `/Role/*` (inkl. Robustheit gegen Response-Format-Varianten)
- `ObjectiveApiService`: `/Objective/*`

9.4 DTO ↔ Domain Mapping

`src/services/api/mappers.ts`:

- `mapAssignmentToWorkStep()` und `mapWorkStepToAssignment()`
- `mapActorToUser()` und `mapRoleDtoToRole()`
- Mapping von backend enums (Priority/status als Zahl) zu frontend enums.

Wichtige Entscheidung: Backend-Felder sind teils nicht vorhanden (z. B. Workflow-Deadline im ObjectiveDto), daher existieren Frontend-Fallbacks/Defaults.

10. Datenflüsse (End-to-End)

10.1 Grundlegender Request-Fluss

1. View/Component triggert Action
2. Composable (`useX`) ruft API-Service (`useApi().x`) auf
3. API-Service nutzt `ApiClient` → HTTP
4. Response wird gemappt (DTO → Domain)
5. Store wird aktualisiert (Pinia)
6. UI aktualisiert automatisch via Vue-Reaktivität

10.2 Login & Session Restore

- Login (`LoginView.vue`):
  - `useActor().loadActors()` lädt Actors
  - Auswahl via DisplayName, Mapping `mapActorToUser()`
  - `userStore.setCurrentUser()` + `localStorage` Persistenz
- Restore:
  - In `src/main.ts` vor `app.mount()` wird `restoreUser()` ausgeführt.
  - Zusätzlich im Router Guard: Restore auf Reload, damit Route-Guard korrekt arbeitet.

10.3 WorkStep Completion → Auto-Assign (Prototype)

- `useWorkStep.completeWorkStep(workStepId)`:
  - PATCH Status auf Completed (`/Assignment/SetStatus`)
  - Store-Update (`workStepStore.updateWorkStep`)
  - Next Step via `sequenceNumber + 1`
  - `getAvailableActors(requiredRole)` + Assign (`/Assignment/SetAssignee`)
  - Notifications an Actor und Workflow Manager (aktuell lokal)

10.4 Priorisierung

- Automatisch:
  - Store `prioritizedWorkSteps` nutzt Deadline + Remaining Duration.
- Manuell:
  - Workflow Manager setzt `manualPriority` via `useWorkflowManager.setManualPriority()` → `/Assignment/SetPriority`.

10.5 Benachrichtigungen

- `useNotification()` filtert pro current user.
- Auslöser: aktuell v. a. Frontend-seitig bei Create/Assign/Reassign/Complete.
- Persistenz: aktuell nicht vorhanden.

11. Deployment & Betrieb

11.1 GitHub Pages

- Vite `base`:
  - Prod: `/kolla_vue/`
  - Dev: `/`

- `postbuild`: `scripts/copy-404.js` kopiert `dist/index.html` → `dist/404.html`.
- Zusätzlich existiert `public/404.html` als SPA-Redirect-Helper.

11.2 CORS

- Dokumentiert in `README.md`: Backend muss CORS für `https://zaynrix.github.io` erlauben.

12. Konventionen & Guidelines

- Imports nutzen Alias `@` → `src` (Vite + TS paths).
- Domain-Typen strikt in `src/types/domain.ts`.
- Backend DTOs strikt in `src/types/api.ts`.
- API-Aufrufe ausschließlich über `src/services/api/*`.
- UI-Zustände (`loading`, `error`) in Composables.

13. Bekannte Tradeoffs / Risiken (aus Code ersichtlich)

- **N+1 Calls**: Mehrere Stellen laden GUID-Liste → dann `Promise.all` Detail-Requests (Actors, Workflows/Objectives, Assignments).
- **Deadline/Dauer Inkonsistenzen**: Teile der Daten sind „Frontend-only" oder haben Defaults (z. B. `duration: 8` im Mapper).
- **Notification Persistenz**: Notifications sind aktuell nur im Frontend-State.
- **RBAC-Feingranularität**: UI blendet via `isAdmin` Navigation aus, Router Guard schützt Admin-Routen; einige Routen haben `requiresAdmin: false` (Workflow-Manager-Route ist für alle Authentifizierten).
- **WorkStep Sequenzierung**: `sequenceNumber` wird teils aus Index abgeleitet; kann bei Reordering/Deletes inkonsistent werden.

14. Erweiterungspunkte (empfohlen)

- **Backend-Unterstützung für Workflow-Deadline** (ObjectiveDto erweitern) und konsistentes Mapping.
- **Persistente Notifications** (Backend-Events / Polling / WebSocket) statt rein lokal.
- **Batch Endpoints** zur Reduzierung von N+1 (z. B. `GET /Actors?ids=...`).
- **Zentrale Error/Toast Strategie** (UI Feedback konsistent, statt `alert()`/`console.*`).

15. Quick Reference: Abhängigkeiten

Aus `package.json` (prod):

- `vue`, `vue-router`, `pinia`, `@vuepic/vue-datepicker`

Dev/Tooling:

- `vite`, `@vitejs/plugin-vue`, `vite-plugin-vue-devtools`
- `typescript`, `vue-tsc`, `@vue/tsconfig`
- `eslint`, `eslint-plugin-vue`, `@vue/eslint-config-typescript`, `prettier`
- `vitest`, `@vue/test-utils`, `jsdom`

Wenn du möchtest, kann ich als nächsten Schritt zusätzlich:

- eine **kleine ADR-Sammlung** als einzelne Dateien (`docs/adr/0001-...md`) erzeugen, oder
- ein **C4-ähnliches Diagramm** (PlantUML/Mermaid) ergänzen (ohne neue Features im Code).