**Lab 5 — API Visualization App**

Rick & Morty API (No Key) • Next.js + MUI + TanStack Query

*October 14, 2025*

**1) Links**

GitHub repository: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Live demo (Vercel) or Demo video link: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Discord post (API claim): https://rickandmortyapi.com/

**2) Overview**

We built a small, app-like UI that visualizes public data from the Rick & Morty API. The app uses three independent interactions so each teammate can own a feature: Characters (search + filters + images), Episodes (search + pagination), and Locations (autocomplete + residents). We used Material UI (MUI) for fast, accessible components and TanStack Query for robust fetching/caching and simple loading/error states.

**3) Features / Interactions**

• Characters: name search, status filter (Alive/Dead/Unknown), species filter, image cards, pagination, skeleton loaders.

• Episodes: search by name, paginated list, inline loading indicator.

• Locations: autocomplete search, location details, residents list (nested fetch).

• Dark Mode toggle, themed colors, sticky AppBar, tabs navigation.

**4) Component Ownership (Team)**

Member A: Characters — components/CharacterSearch.tsx

Member B: Episodes — components/EpisodeList.tsx

Member C: Locations — components/LocationExplorer.tsx

All: Theme, AppBar/Tabs, polishing, bug fixes.

**5) Tech Stack & Rationale**

• Next.js (App Router): simple file-based routing and local dev server.

• MUI: production-grade components (Tabs, Cards, Pagination, Autocomplete).

• TanStack Query: declarative data fetching with caching and race-free updates.

• TypeScript: better DX and fewer runtime errors.

**6) How to Run Locally**

npm i  
npm run dev  
# open http://localhost:3000

**7) Deployment (Optional)**

We deployed to Vercel: https://\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (fill after deploy).

**8) Group Reflection (short)**

We chose the Rick & Morty API because it requires no key, supports CORS, and exposes rich, related resources (characters, episodes, locations). MUI helped us move quickly with a clean, responsive design. React Query simplified fetching and provided caching/loading/error states out of the box. The trickiest part was designing filters and nested fetches for residents without overfetching; Query’s queryKey patterns and keepPreviousData solved most of this cleanly.

**9) Individual Reflections**

Member A — Characters: I implemented debounced search, status/species filters, and image cards. I kept derived UI state out of useEffect and used keepPreviousData to avoid flicker. Challenge: handling 'no results' vs 'error'.

Member B — Episodes: I added search + pagination with optimistic UI. I replaced ad-hoc effects with useQuery; the API’s paging info guided the Pagination count. Challenge: preserving scroll/position between pages.

Member C — Locations: I built an autocomplete that fetches options and then fetches residents for the selected location. Challenge: coordinating queries; solution: enable flags tied to selected IDs and stable queryKeys.

**10) useEffect Replacement Challenges (from React docs)**

We completed five replacements. For each, we explain why the original was suboptimal and show our optimized code.

A — Derived value (Full Name)

Why the original was suboptimal: stored a pure derivation in state and synced with useEffect, causing extra renders and potential stale state.

// before  
const [fullName, setFullName] = useState("");  
useEffect(() => { setFullName(`${first} ${last}`); }, [first, last]);  
  
// after  
const fullName = `${first} ${last}`;

B — Filter/sort list without effects

Why: filtering is render-only work; no side effect. Keeping it in state adds sync complexity.

// before  
const [filtered, setFiltered] = useState(items);  
useEffect(() => { setFiltered(items.filter(i => i.done)); }, [items]);  
  
// after  
const filtered = items.filter(i => i.done);  
// or  
const filtered = useMemo(() => items.filter(i => i.done), [items]);

C — Don’t mirror props into state

Why: two sources of truth. If you need a reset when identity changes, use a key instead of syncing effects.

// before  
const [title, setTitle] = useState(propTitle);  
useEffect(() => setTitle(propTitle), [propTitle]);  
  
// after (simple)  
const title = propTitle;  
  
// after (reset local state on identity change)  
<Editor key={docId} initial={doc.title} />

D — Data fetching with React Query (not ad-hoc effects)

Why: React Query provides caching, retries, race-free updates; less boilerplate and fewer bugs.

// before  
useEffect(() => {  
 let cancel = false;  
 setLoading(true);  
 fetch(url).then(r => r.json())  
 .then(d => { if (!cancel) setData(d); })  
 .catch(e => !cancel && setError(e))  
 .finally(() => !cancel && setLoading(false));  
 return () => { cancel = true; };  
}, [url]);  
  
// after  
const { data, isLoading, error } = useQuery({  
 queryKey: ['data', url],  
 queryFn: () => fetch(url).then(r => r.json()),  
});

E — Heavy computation without effects

Why: sorting/formatting are pure computations; useMemo keeps results stable without state+effect churn.

// before  
const [sorted, setSorted] = useState<Item[]>([]);  
useEffect(() => { setSorted(heavySort(items)); }, [items]);  
  
// after  
const sorted = useMemo(() => heavySort(items), [items]);

**11) AI Attestation**

I acknowledge using ChatGPT for brainstorming APIs, scaffolding React components, and debugging. Benefits: speed, cleaner patterns, and catching edge cases. Challenges: verifying types, ensuring examples matched our project structure, and adapting suggestions to course requirements.

**12) Appendix — Future Improvements**

• Add infinite scroll for characters and episodes.

• Offline cache (persist Query cache).

• Compare two locations side-by-side (residents overlap).

• Add tests for hooks and data formatters.