Avoiding Redux-Saga: Zustand, SWR, Redux Toolkit, TanStack Query

This guide shows how to handle saga-style async problems while trying to avoid redux-saga.

Preferred order:

1. zustand
2. swr
3. redux toolkit
4. tanstack query
5. redux-saga

Use redux-saga only when orchestration itself becomes complex enough to justify a dedicated side-effect runtime.

Quick Mapping

```
| Need | Prefer |
| Simple async actions | ‘zustand‘ |
| Caching and revalidation | ‘swr‘ |
| Sequential API flow | ‘zustand‘ async action |
| Advanced retry/backoff | custom helper, or ‘tanstack query‘ |
| Search cancellation | ‘tanstack query‘, or ‘zustand‘ + ‘AbortController‘ |
| Parallel dashboard loading | ‘swr‘, or ‘Promise.allSettled‘ |
| Checkout workflow/state machine | ‘zustand‘ |
| Redux app orchestration | Redux Toolkit ‘listenerMiddleware‘ |
| True long-running task orchestration | ‘redux-saga‘ |
```

1. Complex Sequential Flows With Conditional Logic

Problem: execute a series of API calls where each step depends on the previous result and decisions depend on responses.

Example: user onboarding flow.

```
import { create } from "zustand";

type OnboardingStep =
  | "idle"
  | "creating-user"
  | "creating-profile"
  | "checking-eligibility"
  | "completed"
  | "failed";

type OnboardingState = {
  step: OnboardingStep;
  error?: string;
  onboardUser: (input: {
    email: string;
```

```
    name: string;
    planId?: string;
  }) => Promise<void>;
};

export const useOnboardingStore = create<OnboardingState>((set) => ({
  step: "idle",

  onboardUser: async (input) => {
    try {
      set({ step: "creating-user", error: undefined });

      const user = await api.createUser({
        email: input.email,
      });

      set({ step: "creating-profile" });

      const profile = await api.createProfile({
        userId: user.id,
        name: input.name,
      });

      set({ step: "checking-eligibility" });

      const eligibility = await api.checkEligibility({
        userId: user.id,
        profileId: profile.id,
      });

      if (!eligibility.allowed) {
        await api.markUserForReview(user.id);
        set({ step: "completed" });
        return;
      }

      if (input.planId) {
        await api.assignPlan({
          userId: user.id,
          planId: input.planId,
        });
      }

      await api.sendWelcomeEmail(user.id);

      set({ step: "completed" });
    } catch (error) {
      set({
        step: "failed",
        error: error instanceof Error ? error.message : "Onboarding failed",
      });
    }
  },
}));
```

Why this avoids saga:

- async/await replaces yield call(...).
- set(...) replaces yield put(...).
- Branching is plain TypeScript.

- The flow is local and easy to test as a function.

2. Complex Retry Logic With Exponential Backoff

Problem: sophisticated retry logic with different strategies for different failure types.

Example: payment processing with intelligent retries.

```
type RetryOptions = {
  maxAttempts: number;
  baseDelayMs: number;
  shouldRetry: (error: unknown, attempt: number) => boolean;
};

const sleep = (ms: number) =>
  new Promise((resolve) => setTimeout(resolve, ms));

async function retryWithBackoff<T>(
  task: () => Promise<T>,
  options: RetryOptions
): Promise<T> {
  let lastError: unknown;

  for (let attempt = 1; attempt <= options.maxAttempts; attempt += 1) {
    try {
      return await task();
    } catch (error) {
      lastError = error;

      if (!options.shouldRetry(error, attempt)) {
        throw error;
      }

      const delay = options.baseDelayMs * 2 ** (attempt - 1);
      await sleep(delay);
    }
  }

  throw lastError;
}
```

Use it in a zustand store:

```
import { create } from "zustand";

type PaymentStatus = "idle" | "processing" | "succeeded" | "failed";

export const usePaymentStore = create<{
  status: PaymentStatus;
  error?: string;
  processPayment: (orderId: string) => Promise<void>;
}>((set) => ({
  status: "idle",

  processPayment: async (orderId) => {
    set({ status: "processing", error: undefined });

    try {
      await retryWithBackoff(() => api.chargePayment(orderId), {
```

```
        maxAttempts: 4,
        baseDelayMs: 500,
        shouldRetry: (error, attempt) => {
          if (isCardDeclined(error)) return false;
          if (isValidationError(error)) return false;
          if (isPaymentGatewayTimeout(error)) return attempt < 4;
          if (isNetworkError(error)) return attempt < 3;

          return false;
        },
      });

      set({ status: "succeeded" });
    } catch (error) {
      set({
        status: "failed",
        error: getErrorMessage(error),
      });
    }
  },
}));
```

Why this avoids saga:

- Saga retry(...) is replaced by a reusable helper.
- Domain rules stay close to payment logic.
- Different error classes can have different retry policies.

3. Race Conditions And Cancellation

Problem: cancel in-progress requests when the user navigates away or triggers a new action.

Example: search autocomplete.

Best fit: tanstack query, because the query function receives an AbortSignal.

```
import { useQuery } from "@tanstack/react-query";

export function useSearchUsers(query: string) {
  return useQuery({
    queryKey: ["users", "search", query],
    enabled: query.trim().length >= 2,
    staleTime: 15_000,
    queryFn: async ({ signal }) => {
      const response = await fetch(
        `/api/users/search?q=${encodeURIComponent(query)}`,
        { signal }
      );

      if (!response.ok) {
        throw new Error("Search failed");
      }

      return response.json() as Promise<User[]>;
    },
  });
}
```

Component:

```
function UserAutocomplete() {
  const [query, setQuery] = useState("");
  const { data = [], isFetching } = useSearchUsers(query);

  return (
    <>
      <input value={query} onChange={(event) => setQuery(event.target.value)} />

      {isFetching && <span>Searching...</span>}

      {data.map((user) => (
        <button key={user.id}>{user.name}</button>
      ))}
    </>
  );
}
```

Why this avoids saga:

- Saga takeLatest(...) is replaced by query-key changes and request cancellation.
- Saga cancel(...) is replaced by AbortSignal.
- Older results do not need to win over newer input.

4. Parallel Execution With Coordination

Problem: execute multiple async tasks in parallel, coordinate results, and handle partial failures.

Example: dashboard data loading.

```
import { create } from "zustand";

type DashboardState = {
  loading: boolean;
  widgets: {
    profile?: Profile;
    stats?: Stats;
    notifications?: Notification[];
  };
  errors: Record<string, string>;
  loadDashboard: () => Promise<void>;
};

export const useDashboardStore = create<DashboardState>((set) => ({
  loading: false,
  widgets: {},
  errors: {},

  loadDashboard: async () => {
    set({ loading: true, errors: {} });

    const results = await Promise.allSettled([
      api.getProfile(),
      api.getStats(),
      api.getNotifications(),
```

```
    ]);

    const errors: Record<string, string> = {};
    const widgets: DashboardState["widgets"] = {};

    if (results[0].status === "fulfilled") {
      widgets.profile = results[0].value;
    } else {
      errors.profile = "Could not load profile";
    }

    if (results[1].status === "fulfilled") {
      widgets.stats = results[1].value;
    } else {
      errors.stats = "Could not load stats";
    }

    if (results[2].status === "fulfilled") {
      widgets.notifications = results[2].value;
    } else {
      errors.notifications = "Could not load notifications";
    }

    set({
      loading: false,
      widgets,
      errors,
    });
  },
}));
```

If the dashboard data is cacheable, prefer swr:

```
import useSWR from "swr";

const fetcher = (url: string) => fetch(url).then((res) => res.json());

export function useDashboardData() {
  const profile = useSWR("/api/profile", fetcher);
  const stats = useSWR("/api/stats", fetcher);
  const notifications = useSWR("/api/notifications", fetcher);

  return {
    data: {
      profile: profile.data,
      stats: stats.data,
      notifications: notifications.data,
    },
    loading: profile.isLoading || stats.isLoading || notifications.isLoading,
    errors: {
      profile: profile.error,
      stats: stats.error,
      notifications: notifications.error,
    },
  };
}
```

Why this avoids saga:

- Saga all(...) is replaced by Promise.allSettled(...).
- Partial failures are modeled explicitly.

- Cacheable data is better handled by a server-state tool.

5. Complex State Machine / Workflow

Problem: implement a workflow that reacts to multiple events in a specific order.

Example: shopping cart checkout.

```
import { create } from "zustand";

type CheckoutState =
  | "cart"
  | "validating"
  | "address"
  | "payment"
  | "confirming"
  | "placing-order"
  | "success"
  | "failed";

type CheckoutStore = {
  state: CheckoutState;
  error?: string;
  startCheckout: () => Promise<void>;
  submitAddress: (address: Address) => Promise<void>;
  submitPayment: (payment: PaymentInput) => Promise<void>;
  placeOrder: () => Promise<void>;
};

export const useCheckoutStore = create<CheckoutStore>((set, get) => ({
  state: "cart",

  startCheckout: async () => {
    set({ state: "validating", error: undefined });

    try {
      const result = await api.validateCart();

      if (!result.valid) {
        set({ state: "cart", error: result.reason });
        return;
      }

      set({ state: "address" });
    } catch {
      set({ state: "failed", error: "Could not validate cart" });
    }
  },

  submitAddress: async (address) => {
    if (get().state !== "address") return;

    await api.saveAddress(address);
    set({ state: "payment" });
  },

  submitPayment: async (payment) => {
    if (get().state !== "payment") return;
```

```
      await api.savePaymentMethod(payment);
      set({ state: "confirming" });
    },

    placeOrder: async () => {
      if (get().state !== "confirming") return;

      set({ state: "placing-order" });

      try {
        await api.placeOrder();
        set({ state: "success" });
      } catch {
        set({ state: "failed", error: "Could not place order" });
      }
    },
  }));
```

Why this avoids saga:

- The workflow is represented by explicit states.
- Invalid transitions are guarded.
- UI can render directly from state.

6. Caching And Revalidation

Problem: keep remote data fresh, dedupe requests, and revalidate on focus or reconnect.

Best fit: swr.

```
import useSWR, { mutate } from "swr";

const fetcher = async (url: string) => {
  const response = await fetch(url);

  if (!response.ok) {
    throw new Error("Request failed");
  }

  return response.json();
};

export function useUserProfile(userId: string) {
  return useSWR(`/api/users/${userId}`, fetcher, {
    revalidateOnFocus: true,
    revalidateOnReconnect: true,
    dedupingInterval: 2_000,
  });
}

export async function updateUserProfile(
  userId: string,
  input: UpdateUserInput
) {
  await mutate(
    `/api/users/${userId}`,
    async (currentUser?: User) => {
      const updatedUser = await api.updateUser(userId, input);
```

```
      return { ...currentUser, ...updatedUser };
    },
    {
      optimisticData: (currentUser?: User) => ({
        ...currentUser,
        ...input,
      }),
      rollbackOnError: true,
      revalidate: true,
    }
  );
}
```

Why this avoids saga:

- Cache and revalidation should not be hand-built in Redux.
- SWR handles focus revalidation, reconnect revalidation, deduping, optimistic updates, and rollback.

7. Simple Async Actions

Problem: load or mutate data with simple loading/error state.

Best fit: zustand.

```
import { create } from "zustand";

type UserStore = {
  user?: User;
  loading: boolean;
  error?: string;
  loadUser: (id: string) => Promise<void>;
};

export const useUserStore = create<UserStore>((set) => ({
  loading: false,

  loadUser: async (id) => {
    set({ loading: true, error: undefined });

    try {
      const user = await api.getUser(id);
      set({ user, loading: false });
    } catch (error) {
      set({
        loading: false,
        error: getErrorMessage(error),
      });
    }
  },
}));
```

Why this avoids saga:

- No action ceremony.
- No separate thunk or saga layer.
- Component state stays readable.

Redux Toolkit Listener Middleware As A Saga Alternative

If the app already uses Redux, Redux Toolkit listenerMiddleware can cover many saga-like needs.

Example: takeLatest style search.

```
import {
  createAction,
  createListenerMiddleware,
  createSlice,
} from "@reduxjs/toolkit";

export const searchChanged = createAction<string>("search/changed");

const searchSlice = createSlice({
  name: "search",
  initialState: {
    query: "",
    results: [] as User[],
    loading: false,
  },
  reducers: {
    searchStarted: (state) => {
      state.loading = true;
    },
    searchSucceeded: (state, action) => {
      state.loading = false;
      state.results = action.payload;
    },
    searchFailed: (state) => {
      state.loading = false;
    },
  },
});

export const searchListener = createListenerMiddleware();

searchListener.startListening({
  actionCreator: searchChanged,
  effect: async (action, listenerApi) => {
    listenerApi.cancelActiveListeners();

    await listenerApi.delay(300);

    listenerApi.dispatch(searchSlice.actions.searchStarted());

    try {
      const results = await api.searchUsers(action.payload, {
        signal: listenerApi.signal,
      });

      listenerApi.dispatch(searchSlice.actions.searchSucceeded(results));
    } catch (error) {
      if (listenerApi.signal.aborted) return;

      listenerApi.dispatch(searchSlice.actions.searchFailed());
    }
  },
});
```

Saga concepts replaced:

```
| Redux-Saga | Alternative |
| ‘takeLatest(...)‘ | ‘listenerApi.cancelActiveListeners()‘ |
| ‘delay(...)‘ | ‘listenerApi.delay(...)‘ |
| ‘cancel(...)‘ | ‘AbortSignal‘ / listener cancellation |
| ‘call(...)‘ | direct async function call |
| ‘put(...)‘ | ‘dispatch(...)‘ |
```

When Redux-Saga Is Still Reasonable

Use redux-saga only when several of these are true:

- You have long-running background processes.
- You need to coordinate many independent Redux actions over time.
- You need advanced event channels, websockets, or streams.
- You need cancellation trees or task supervision.
- You need transactional workflows spanning many screens and many domains.
- Your team is already fluent in saga effects and tests.

Otherwise:

- Use zustand for client-side workflow state.
- Use swr for cached server reads and revalidation.
- Use tanstack query when request cancellation, retries, mutations, or query coordination become important.
- Use Redux Toolkit listenerMiddleware when you are already in Redux and need saga-like orchestration.

Final Recommendation

Start with this order:

1. zustand for local app state, async actions, and explicit workflows.
2. swr for cache, revalidation, deduping, optimistic updates, and rollback.
3. Redux Toolkit listenerMiddleware if Redux is already part of the app.
4. tanstack query for stronger server-state orchestration, cancellation, retries, and mutations.
5. redux-saga only for true orchestration-heavy applications.

In most modern React apps, the combination of zustand + swr or zustand + tanstack query handles the practical benefits people used to reach for in redux-saga, while keeping the code easier to read and maintain.