Thunk Layer Discussion: Size, Helpers, and Organization

Question: "Won't moving thunks to Presentation make that layer too large? Should we add a helper or another layer?"

Short Answer: No, it's not too large. Start explicit. Add helpers only if real duplication emerges.

The Concern

After moving Redux thunks from Use Cases to Presentation layer, you'll have:

```
src/presentation/state/thunks/
├── lifeContext.thunks.ts    (~150 lines, 7 thunks)
├── metricLog.thunks.ts      (~100 lines, 4 thunks)
├── myHealth.thunks.ts       (~120 lines, 5 thunks)
├── carefeed.thunks.ts       (~60 lines, 2 thunks)
├── insight.thunks.ts        (~40 lines, 1 thunk)
└── onboarding.thunks.ts     (~130 lines, 5 thunks)

Total: ~600 lines across 6 files, 24 thunks
```

Developer concern: "That feels large. Should we abstract this?"

Three Possible Solutions

Option 1: Keep It Explicit (Recommended ⭐)

Just write the thunks out:

```typescript
// src/presentation/state/thunks/metricLog.thunks.ts

export const saveMetricLogs = createAsyncThunk<
  SuccessMessage,
  SaveMetricLogsParams,
  { rejectValue: string }
>("metricLog/save", async (params, { rejectWithValue }) => {
  try {
    const useCase = container.resolve(SaveMetricLogsUseCase);
    return await useCase.execute(params);
  } catch (error) {
    return rejectWithValue(
      error instanceof Error ? error.message : "Failed to save metric logs",
    );
  }
});

export const listMetricLogs = createAsyncThunk<
  MetricLog[],
  ListMetricLogsParams,
  { rejectValue: string }
>("metricLog/list", async (params, { rejectWithValue }) => {
  try {
    const useCase = container.resolve(ListMetricLogsUseCase);
    return await useCase.execute(params);
  } catch (error) {
    return rejectWithValue(
      error instanceof Error ? error.message : "Failed to list metric logs",
    );
  }
});

// ... 2 more thunks
```

Option 2: Create a Helper (Middle Ground 👍)

If you notice real duplication, create a helper:

```
// src/presentation/state/helpers/createUseCaseThunk.ts

import { createAsyncThunk } from "@reduxjs/toolkit";
import { container } from "tsyringe";

/**
 * Helper to create a thunk that wraps a use case.
 * Use for simple cases. Write custom thunks for complex logic.
 */
export function createUseCaseThunk<
  TResult,
  TParams,
  TUseCase extends { execute: (params: TParams) => Promise<TResult> },
>(actionName: string, UseCaseClass: new (...args: any[]) => TUseCase) {
  return createAsyncThunk<TResult, TParams, { rejectValue: string }>(
    actionName,
    async (params, { rejectWithValue }) => {
      try {
        const useCase = container.resolve(UseCaseClass);
        return await useCase.execute(params);
      } catch (error) {
        const message =
          error instanceof Error ? error.message : "Operation failed";
        return rejectWithValue(message);
      }
    },
  );
}
```

Usage:

```
// src/presentation/state/thunks/metricLog.thunks.ts

import { createUseCaseThunk } from "../helpers/createUseCaseThunk";
import { SaveMetricLogsUseCase } from "@/application/useCases/metricLog/SaveMetricLogs.useCase";
import { ListMetricLogsUseCase } from "@/application/useCases/metricLog/ListMetricLogs.useCase";
```

Option 3: Add Another Layer (Not Recommended ❌)

Create "Interface Adapters" layer:

```
src/
├── container.ts        ← DI container (bootstrap)
├── domain/
├── application/
├── adapters/           ← NEW LAYER
│   └── redux/
│       └── thunks/     ← Thunks here
├── infrastructure/
└── presentation/
    └── ui/             ← Only components
```

Arguments for:

- Follows "textbook" Clean Architecture (5 layers)
- Separates "state" from "UI components"
- Thunks aren't "mixed" with UI

Arguments against:

- ❌ **Adds complexity** - 5 layers is overkill for most apps
- ❌ **Unclear boundaries** - What's "adapter" vs "presentation"?
- ❌ **Not industry standard** - React community considers Redux part of presentation
- ❌ **Solves wrong problem** - The "size" issue is perception, not architecture
- ❌ **More navigation** - Developers jump between more folders

When this WOULD make sense:

- Multi-platform app (Web + Mobile sharing business logic)
- Multiple UI frameworks (React + Vue + Angular)

Reality Check: Is 600 Lines Actually Large?

Let's compare to other parts of your codebase:

Layer/Folder	Files	Lines	Purpose
Domain	38 files	~2,000 lines	Entities, types
Application	109 files	~6,000 lines	Use cases, services
Infrastructure	51 files	~3,000 lines	Firebase, Sanity
Presentation (current)	59 files	~3,500 lines	ViewModels, components
Presentation (after)	**~70 files**	**~4,100 lines**	**+ 600 lines of thunks**

Adding 600 lines to a layer that already has 3,500 lines = 17% increase

That's not "too large." That's normal growth from putting things in the right place.

Comparison to Alternatives

What if you used different approaches?

Redux Toolkit Query (RTK Query)

```
// Generated code per endpoint
const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: "/api" }),
  endpoints: (builder) => ({
    getLifeContext: builder.query<LifeContext[], string>({
      query: (uid) => `lifeContext/${uid}`,
    }),
    saveMetricLogs: builder.mutation<SuccessMessage, SaveParams>({
      query: (params) => ({
        url: "metricLogs",
        method: "POST",
        body: params,
      }),
    }),
    // ... 22 more endpoints
  }),
});
```

Generated code size: 1,000+ lines (hooks, selectors, cache management)

Your manual thunks: 600 lines

Winner: Your approach is actually leaner!

Apollo Client (GraphQL)

```
// Queries, mutations, and generated hooks
const GET_LIFE_CONTEXT = gql`...`;
const SAVE_METRIC_LOGS = gql`...`;
// + Generated TypeScript types
```

The "Explicitness Tax" is Worth Paying

Scenario 1: Bug Investigation

With helper (implicit):

```
export const saveMetricLogs = createUseCaseThunk(
  "metricLog/save",
  SaveMetricLogsUseCase,
);
// Where does error handling happen? 🤔
// What's the error message format? 🤔
// Can I customize the loading state? 🤔
```

Without helper (explicit):

```
export const saveMetricLogs = createAsyncThunk(
  "metricLog/save",
  async (params, { rejectWithValue }) => {
    try {
      const useCase = container.resolve(SaveMetricLogsUseCase);
      return await useCase.execute(params);
    } catch (error) {
      return rejectWithValue(error.message); // ← Clear error handling
    }
  },
);
```

When there's a bug, explicit code is your friend.

Scenario 2: Adding Custom Logic

Need to add analytics tracking to one thunk?

Organization Strategy

The key is organization, not abstraction:

✅ Good Organization

```
src/presentation/state/
├── store.ts                 (1 file, 100 lines)
├── slices/
│   ├── lifeContext/
│   │   └── lifeContext.slice.ts
│   ├── metricLog/
│   │   └── metricLog.slice.ts
│   └── ...                  (13 slice files, ~1,500 lines total)
├── thunks/
│   ├── lifeContext.thunks.ts    (7 thunks, ~150 lines)
│   ├── metricLog.thunks.ts      (4 thunks, ~100 lines)
│   ├── myHealth.thunks.ts       (5 thunks, ~120 lines)
│   ├── carefeed.thunks.ts       (2 thunks, ~60 lines)
│   ├── insight.thunks.ts        (1 thunk, ~40 lines)
│   └── onboarding.thunks.ts     (5 thunks, ~130 lines)
├── listeners/               (4 files, ~400 lines)
└── selectors/               (Optional: ~200 lines)

Total: ~2,600 lines across ~30 files
```

Well organized by domain = Easy to navigate

❌ Bad Organization

```
src/presentation/state/
├── store.ts
├── slices.ts               (ALL slices in one file, 2,000 lines!)
├── thunks.ts               (ALL thunks in one file, 600 lines!)
└── listeners.ts            (ALL listeners in one file, 400 lines!)

Total: ~3,000 lines across 4 files
```

Recommended Approach

Step 1: Start Explicit (Week 1)

Write your first 5-10 thunks manually:

```
// metricLog.thunks.ts
export const saveMetricLogs = createAsyncThunk(/* ... */);
export const listMetricLogs = createAsyncThunk(/* ... */);
export const deleteMetricLog = createAsyncThunk(/* ... */);

// lifeContext.thunks.ts
export const fetchLifeContext = createAsyncThunk(/* ... */);
export const initiateMenstruation = createAsyncThunk(/* ... */);
```

You'll quickly notice:

- Which thunks are simple (just call use case)
- Which thunks need custom logic (auth, analytics, multi-step)
- If a pattern emerges

Step 2: Assess After 10+ Thunks (Week 2)

Ask yourself:

- Do 80%+ of thunks look identical?
- Is the repetition actually painful?
- Would a helper make things clearer or more magical?

If YES to all three → Create helper
If NO to any → Keep explicit

Counter-Arguments

"But it's so much typing!"

Response:

- 10 lines per thunk × 24 thunks = 240 lines of "boilerplate"
- That's 0.4% of your total codebase
- Not actually a significant amount

Trade-off:

- Explicit = More typing, clearer intent
- Helper = Less typing, more magic

Industry leans toward: Explicit for thunks (RTK doesn't provide helpers either)

"Other developers will copy-paste and make mistakes!"

Response:

- Good! They'll see the full pattern and understand it
- VS: They use helper wrong and don't understand what it does

Better solution:

- Code review catches copy-paste errors
- Snippet/template reduces typing (see below)

"Can't we just generate these?"

IDE Snippet Solution

Instead of a helper, create an IDE snippet:

VS Code Snippet

```
// .vscode/thunk.code-snippets
{
  "Create Use Case Thunk": {
    "prefix": "usecasethunk",
    "body": [
      "export const ${1:functionName} = createAsyncThunk<",
      "  ${2:ResultType},",
      "  ${3:ParamsType},",
      "  { rejectValue: string }",
      ">(",
      "  '${4:domain}/${5:action}',",
      "  async (params, { rejectWithValue }) => {",
      "    try {",
      "      const useCase = container.resolve(${6:UseCaseClass});",
      "      return await useCase.execute(params);",
      "    } catch (error) {",
      "      return rejectWithValue(",
      "        error instanceof Error ? error.message : 'Operation failed'",
      "      );",
      "    }",
      "  }",
      ");"
    ],
    "description": "Create a thunk that wraps a use case"
  }
}
```

Usage: Type `usecasethunk` + Tab, fill in the blanks

Benefits:

- ✅ Fast typing (5 seconds per thunk)

Final Recommendation

For Your Team

Option 1: Start Explicit ⭐⭐⭐⭐⭐

- Write thunks manually
- Organize by domain (6 files)
- Create IDE snippet for speed
- Re-evaluate after 15+ thunks

Option 2: Helper After Proof ⭐⭐⭐

- Write 10 thunks manually first
- If clear pattern + real pain → create helper
- Use helper for simple cases only
- Keep complex thunks explicit

Option 3: Add Layer ⭐

- Only if multi-platform or multi-framework
- Not for single React Native app

My Strong Opinion

"Explicit is better than implicit. DRY is good, but clarity is better."

The "cost" of writing 10 lines per thunk is tiny compared to the benefit of:

- Immediate understanding when debugging

Decision Matrix

Use this to decide:

Scenario	Recommendation
Just starting migration	⭐ Write explicit thunks
Have 5 thunks, all similar	⭐ Keep explicit (too early to tell)
Have 15 thunks, 80% identical	⭐⭐ Consider helper
Have 15 thunks, 50% need customization	⭐ Keep explicit
Multi-platform app	⭐⭐ Maybe add adapter layer
Single React Native app	⭐ Stay with 4 layers
Team loves abstractions	⭐⭐ Helper might be okay
Team is new to Redux	⭐ Explicit helps learning

Conclusion

The developer's concern is understandable but misplaced.

The "largeness" is not a problem to solve—it's the natural size of making things explicit and properly organized.

Start explicit. Add helpers only if real pain emerges.

Don't add layers. Organize what you have.

Quick Reference

✅ DO

- Write explicit thunks initially
- Organize by domain (one file per domain)
- Keep files under 200 lines
- Allow customization per-thunk
- Create IDE snippets for speed

❌ DON'T

- Create helpers before seeing patterns
- Put all thunks in one file
- Add a 5th architectural layer
- Optimize for "less typing" over clarity
- Abstract just because it "feels large"

See Also:

- Architecture Analysis Report - Full details on refactoring
- Migration Example - Step-by-step thunk creation
- Quick Reference - Daily development guide

Last Updated: 2025-11-19