micro-libraries

A discourse on separation of concerns, DDD, and Strategic Design

Sections

DDD Strategic Design
Common Closure Principle
Domain
Data-Access
Feature
Shared
Shell
UI
Util
Feature-API
Modularized Development
Q&A

DDD Strategic Design

What is Strategic Design

Anti-Corruption Layer

Glossary

Additional Reading

Anti-Corruption Layer

An Anti-Corruption Layer (ACL) is a set of patterns placed between the domain model and other bounded contexts or third party dependencies. The intent of this layer is to prevent the intrusion of foreign concepts and models into the domain model. This layer is typically made up of several well-known design patterns such as Facade and Adapter. The patterns are used to map between foreign domain models and APIs to types and interfaces that are part of the domain model.

CCP: Common Closure Principle Definition

Whether we group things by micro-libraries or by libraries with folders, the same paradigm remains true in the fact that we are grouping things together based on the underlying commonality.

The Reuse/Release Equivalence Principle (REP): THE GRANULE OF REUSE IS THE GRANULE OF RELEASE.

Additional Read

Library Types (can also be folders)

Agnostic*	APP	API	Serverless**
Domain	Shell		
Data-Access	UI		
Feature			
Feature-API			
Util			
Shared			

* each agnostic library type works for any swimlane
** since serverless is mainly just functions, it does not constitute swimlanes, but should likely be condensed into libraries folder and functions folder be moved up under only serverless just like apps/ apis/

Library Naming Convention & Encapsulating boundaries

Two Different Variations (We need to pick one)

NESTED

```
libs/
−| maps/
−−| api/
−−−| domain/
−−−| data-access/
−−| app/
−−−| domain/
−−−| feature/
−−−−−| spotlight/
−−−−−| locate-me/
−−−| util/
−−−| app-api/
```

VS

FLAT

```
libs/
−| maps/
−−| api-domain/
−−| api-data-access/
−−| api-util/
−−| app-domain/
−−| app-data-access/
−−| app-feature-spotlight/
−−| app-feature-locate-me/
−−| app-util/
−−| app-api/
```

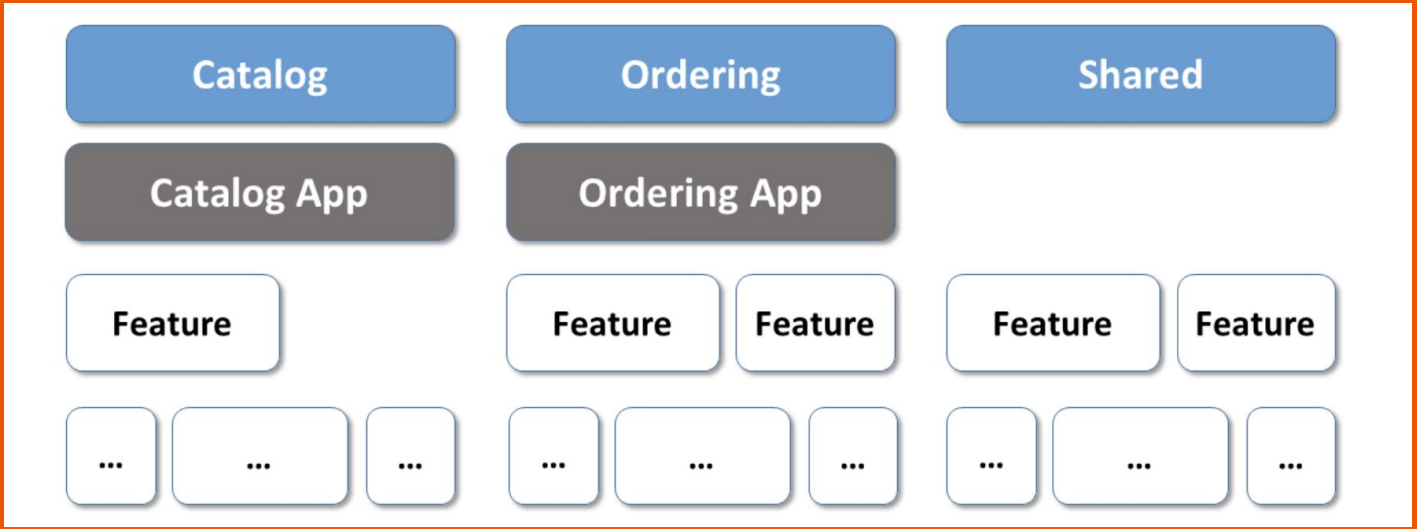
`Example options: feature` (if nested) or `feature-*` (e.g., `feature-home`).

@rahrah/maps-api-domain

@rahrah/maps-app-feature-spotlight

@rahrah/maps-app-util

Legos at a High Level



Key Points on micro-libraries

1. Swimlanes are broken down by Domains into subdomains; subdomains are children of a higher-level domain but are domains in and of itself. Not to be confused with the Domain Logic Layer.

2. Each library may only access micro-libraries from the same domain or shared libraries; access to APIs such as `maps-api` must be explicitly granted to individual swimlanes. Ideally we utilize available tools to enforce this.

3. Reusable micro libraries across the monorepo that are generic without business logic belong to the shared domain and a subdomain of app or api or serverless or global.

4. Stating the obvious: The stronger the enforcement of bounded contexts, the less of an ability to mix Concerns. Micro-libraries are stronger enforcements than folder separation.

Domain

```
domain: Domain logic like calculating additional expenses (not used here), validations or
facades for use cases and state management.

├── domain*
|   ├── configs**
|   ├── entities
|   ├── store***
|   ├── validations
|   ├── ...

Speaking Point: Value Objects vs Entities vs Aggregates vs Root Aggregate
```

*. Domain contains all the business logic for the application; this includes any calculations, etc.

**. configs in this case is not to be confused with environments/configs; any going out of the application and into the underlying infrastructure for environment variables should not be done in these configs, but rather in the environments/configs grouping folder (this includes the shared swimlane).

***. The store is the state management; this is only applicable when the state management is internally within the application and there is no need to go externally to say redis or memcached. In those instances, it would be part of the data-acess layer. Direct access to the statement management only occurs through the api (facades) and the inner workings of the store are kept internal to these swimlanes allowing easy swapping out of the implementation of the underlining store of choice.

Data Access

Implements data accesses, e.g. via HTTP or WebSockets

Feature

Implements a use case with smart components

Feature-API

Provides functionalities exposed to other domains

Shell

shell: For an application that has multiple domains, a shell provides the entry point for a domain

Types of Shell Patterns and Use Cases

UI

Provides use case-agnostic and thus reusable components (dumb components)

Util

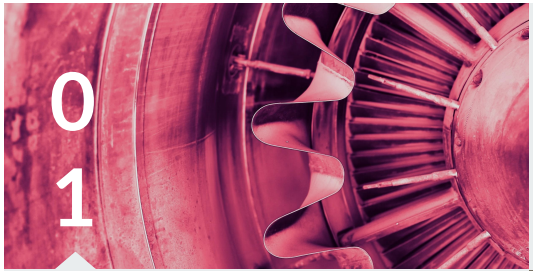
Provides helper functions

Shared

Micro-libraries based on very specific use cases


Tools & Frameworks

Nrwl NX



Node.js

@nrwl/node

React

@nrwl/react

Tags: Enforcing Bounded Context

Tags

Incremental Adoption of NX

Why NX?

In a nutshell, Nx helps to: speed up your computation (e.g. builds, tests etc), locally and on CI and to integrate and automate your tooling via its plugins. All of this can be adopted incrementally. You can use plugins, but you don't have to. Look at the Nx architecture in the next section to learn more.

TAGS: LIBRARY BOUNDARIES

```
"projects": {
 "ui": { "tags": ["scope:app"] },
 "ui-e2e": {"tags": ["scope:e2e"] },
 "catalog-shell": { "tags": ["scope:catalog",
"type:shell"] },
 "catalog-feature-request-product": { "tags":
["scope:catalog", "type:feature"] },
 "catalog-feature-browse-products": { "tags":
["scope:catalog", "type:feature"] },
 "catalog-api": { "tags": ["scope:catalog",
"type:api", "name:catalog-api"] },
 "catalog-data-access": { "tags":
["scope:catalog", "type:data-access"] },
 "shared-util-auth": { "tags": ["scope:shared",
"type:util"] } }
```

Eslint rules for Boundary Enforcement

```
"@nrwl/nx/nx-enforce-module-boundaries": [
  "error",
  {
    "allow": [],
    "depConstraints": [
      { "sourceTag": "scope:app",
        "onlyDependOnLibsWithTags": ["type:shell"] },
      { "sourceTag": "scope:catalog",
        "onlyDependOnLibsWithTags": ["scope:catalog", "scope:shared"] },
      { "sourceTag": "scope:shared",
        "onlyDependOnLibsWithTags": ["scope:shared"] },
      { "sourceTag": "scope:booking",
        "onlyDependOnLibsWithTags":
          ["scope:booking", "scope:shared", "name:catalog-api"] },

      { "sourceTag": "type:shell",
        "onlyDependOnLibsWithTags": ["type:feature", "type:util"] },
      { "sourceTag": "type:feature",
        "onlyDependOnLibsWithTags": ["type:data-access", "type:util"] },
      { "sourceTag": "type:api",
        "onlyDependOnLibsWithTags": ["type:data-access", "type:util"] },
      { "sourceTag": "type:util",
        "onlyDependOnLibsWithTags": ["type:util"] }
    ]
  }
]
```

Distributed Caching

https://nx.dev/core-features/share-your-cache

The computation cache provided by Nx can be distributed across multiple machines. You can either build an implementation of the cache or use Nx Cloud. Nx Cloud is an app that provides a fast and zero-config implementation of distributed caching. It's completely free for OSS projects and for most closed-sourced projects (read more here).

React Demo

https://nx.dev/react-tutorial/1-code-generation

OUT OF THE BOX SUPPORT FOR SWC

One package.json (but yet ...)

generatePackageJson*
boolean
Default: `false`

Generates a `package.json` and pruned lock file with the project's `node_module` dependencies populated for installing in a container.

If a `package.json` exists in the project's directory, it will be reused with dependencies populated

* this is executors/webpack, but other plugins or custom executors can provide the same thing

Thank you.