Entain Nx architecture

29-06-2023

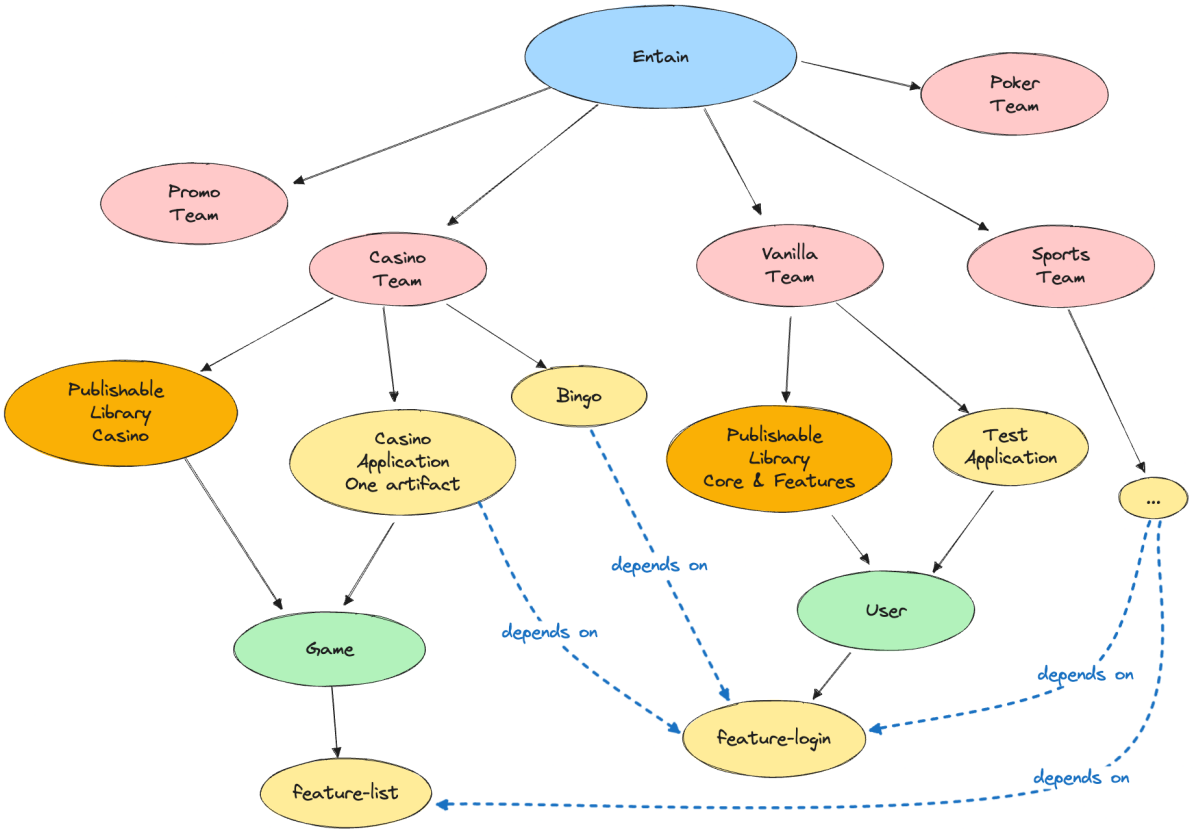
This document outlines the target architecture of the Entain Nx monorepository. The Entain Nx migration plan document describes the steps needed to reach it and their order based on the order and priority of the goals described in the Entain Nx migration goals document.

All Nx projects will be categorized by domain and types. Thus will allow us to configure some dependency constraints and enforce our architectural guidelines.

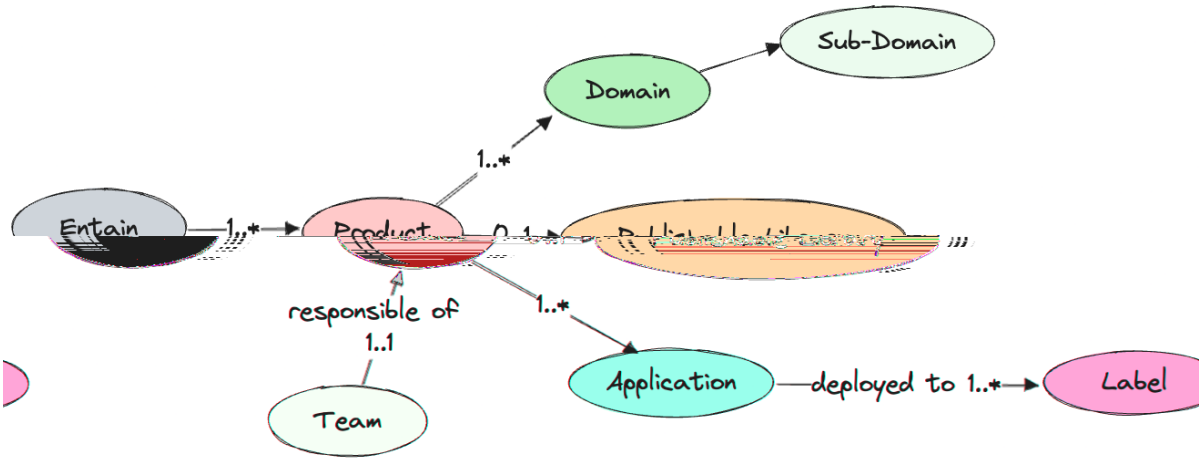
Entain Code Organization

The Nx architecture should help to organize the code by representing the way the organization works.

After discussions, here a first subset of the way the code is organized:



From the example above, we can define the global relationship graph between the Entain elements:



- ➔ The **E** organization is **d**ivided in multiple **P d c**
- ➔ An Entain **Te** is responsible f**o**r one **P d c**
- ➔ The business context of a **P d c** can be divided in multiple **D** and **d a S a b**
- ➔ A **P d c** will expose multiple **A c** related to on**e** business context
- ➔ A **P d c** can expose one **P e L e** to b**e** able to s**h**ar**e b b b** some **Fe e** with the oth**e**r **P d c**
- ➔ An **A c** can be deplo**y**ed to multiple **L e a**

The structure of the domains will depend on the complexity of the product.

Folder structure

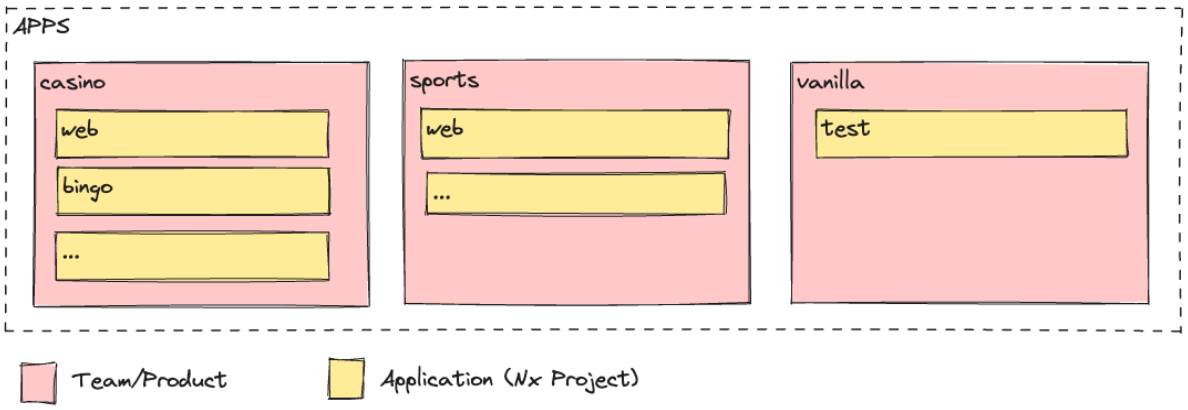
Based on the code organization, we can extract a structure of the code that can be used in the Nx architecture.

In Nx, we have 2 defaults folders: and DOCS **b**

Applications

The folder will co**n**tain all **c** that w**i**ll be **d**eployed to **e** .

An **c** doesn'**t** conta**i**n a lot. Just the main configurations and the root component that are needed to bundle it. DOCS

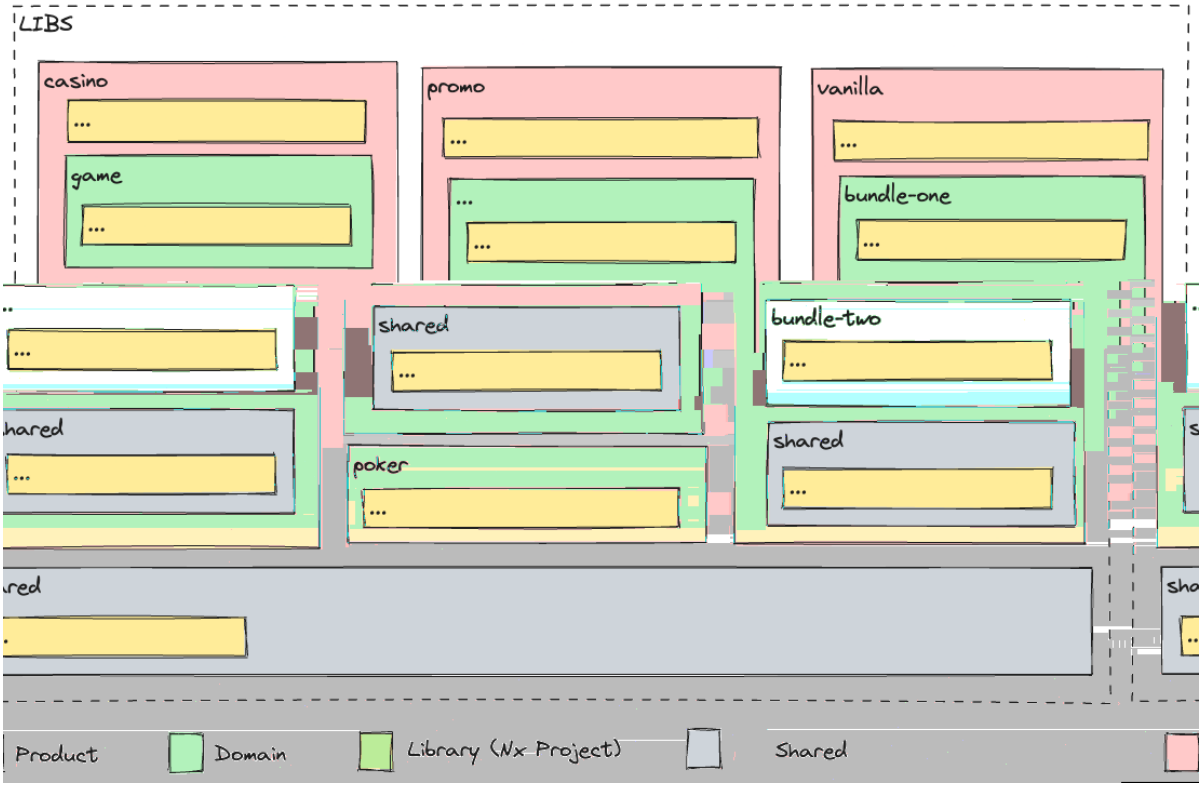


The c will be a grouped in a folder representing the d c / e . a

Libraries

Domain

The folder will be structured by d b to allow a better distribution and isolation of the code. DOCS



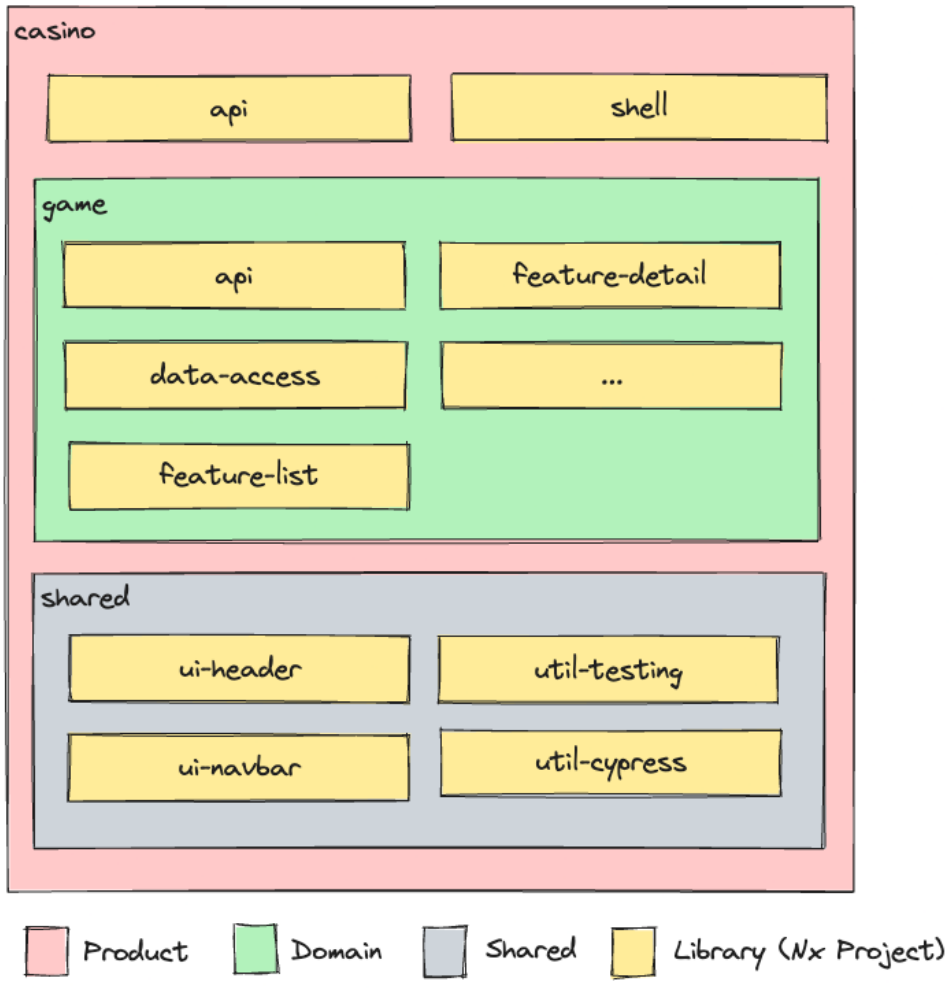
- The first level is the d c which is related to the e Global a libraries related to one product will be located at d c e e a a b
- The second level will be the business d related to the a d c Libraries specific to one domain will be located at d c e d ae a e a a b

- ➔ Extra **d** can be used if **a**needed **b**
- ➔ At each level, a **ed** folder can be **a**reated to group libraries containing code shared at the same level:
  - libs/shared/*
  - libs/casino/shared/*
  - libs/promo/shared/*
  - libs/casino/game/shared/*

Types

Each library context is defined by the folder structure which is representing the domain related (see above).

Each library will also be defined by **e**, the role that the library is representing. To do so, some conventions will be used in library name.



We use the following conventional library types:

- ➔ **e** : will be related to an application and will be responsible for the orchestration.
- ➔ : will expose all **s**hared code to be used by another domain

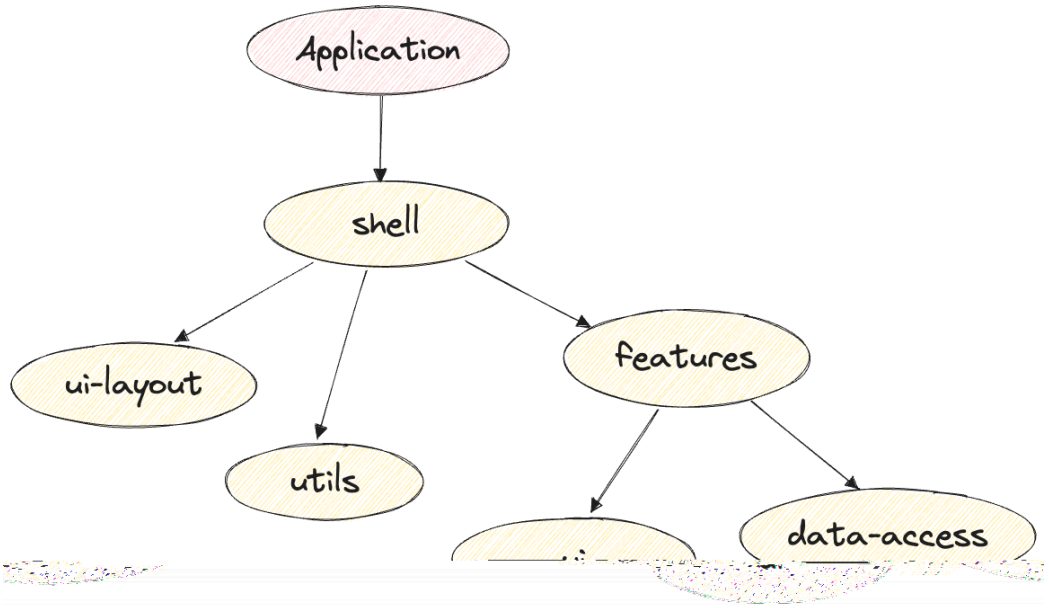
- **d cce** : will cota ia the client services and the entities related to the domain
- **e e**: will contain the implementation of the feature relate to the domain
- : will contain all dumb components usually shared between features
- : will contain all code usually shared

Extra library types can be used to share static files:

- **e** : will contain some shared assets such as images, ...
- **e e** : will contains some shared Angular environment files
- **e** : will contain some shared styles and fonts

Interaction Graph

Based on the folder structure above, we can see already how an **c** will interact with the **e** : **a** **b**



Import paths

Local libraries and buildable libraries

Every local or buildable library has a path mapping prefixed with the npm scope that follows the workspace folder structure described in the *Entain Nx architecture* document, for example `@entain/bwin/mobile-sports/sports-web/basket-ball/calendar/feature-matches` and `@entain/shared/basketball/ui-matches` which would have the following path mapping.

```
{
    "compilerOptions": {
        "//": "Other settings omitted for brevity",
        "paths": {
```

```
"@entain/bwin/mobile-sports/sports-web/basket-ball/calendar/feature-matches": [

"libs/bwin/mobile-sports/sports-web/basket-ball/calendar/feature-matches/src/index.ts"
            ],
            "@entain/shared/basketball/ui-matches": [
                "libs/shared/basketball/ui-matches/src/index.ts"
            ]
        }
    }
}
```

Path mappings in the `tsconfig.base.json` TypeScript configuration file.

Publishable libraries

Publishable libraries use the npm scope that is registered in JFrog Artifactory.

No nested re-exports

1. Each library project has at most one barrel file (`index.ts`/`public_api.ts`) per entry point.
    1. All libraries containing TypeScript have a primary entry point (`index.ts`/`public_api.ts`).
    2. Publishable libraries can have secondary entry points, adding an additional `index.ts`/`public_api.ts` file each.
2. Except for library entry points, no ECMAScript module has `export` statements that re-export software artifacts from other ECMAScript modules.

Project tags

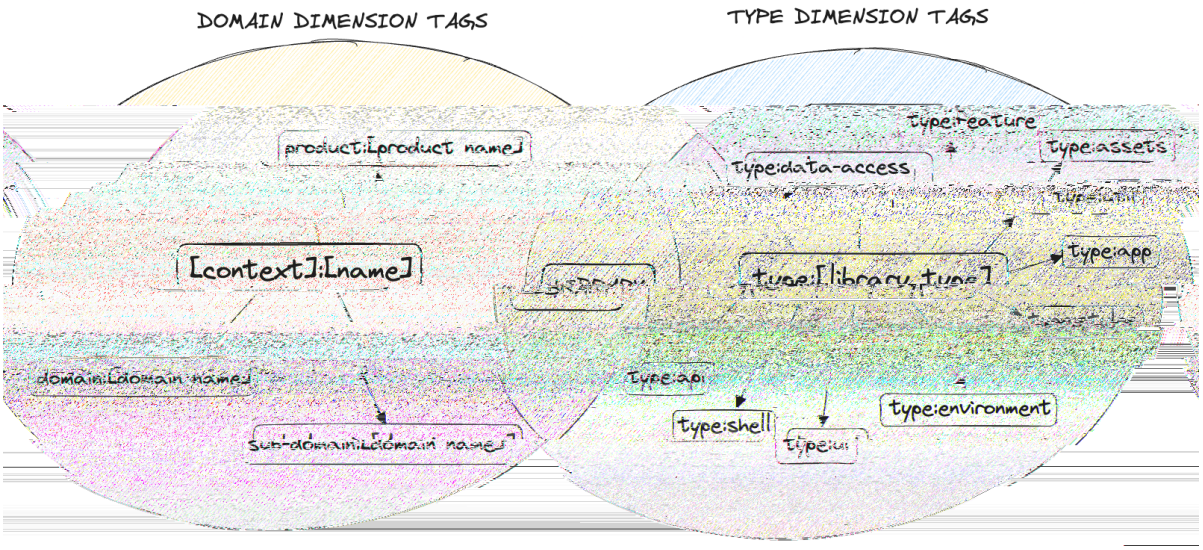
The in Nx is a concept that allows us to describe a library in multiple dimensions.

The list of the tags will be specified in the **ec** of the project.

DOCS

Dimensions

As described in the previous sections, we have 2 dimensions:



➔ The **d** dimension will **a**se the tags format: **c** **e** **e** **a**

➔ The **e** dimension will use the tags format: **e** **e** **a**

Examples

Application Projects

➔ apps/casino/web/project.json

Unset

```
{
  "tags": [
    "product:casino",
    "type:app"
  ]
}
```

Libraries Projects

➔ libs/casino/api/project.json

Unset

```
{
  "tags": [
    "product:casino",
    "type:api"
  ]
}
```

➔ libs/casino/shell/project.json

```
Unset
{
  "tags": [
    "product:casino",
    "type:shell"
  ]
}
```

➔ libs/casino/game/feature-list/project.json

```
Unset
{
  "tags": [
    "product:casino",
    "domain:game",
    "type:feature"
  ]
}
```

➔ libs/casino/shared/ui-header/project.json

```
Unset
{
  "tags": [
    "product:casino",
    "domain:shared",
    "type:ui"
  ]
}
```

➔ libs/shared/assets/project.json

```
Unset
{
  "tags": [
    "product:shared",
    "domain:shared",
    "type:assets"
  ]
}
```

Boundaries

We configure Nx lint checks to enforce our architectural boundaries as described in this section. DOCS

Application projects are not meant to contain reusable or composable code. Reusable and composable code is instead placed in library projects.

1. No project must import from an application project.
   This is enforced when Nx lint checks are enabled.
2. They must not have TypeScript path mappings.
3. They must not have an entry point (`index.ts`/`public_api.ts`).

End-to-end testing projects are not meant to contain cross-project code. Reusable end-to-end testing code is instead placed in end-to-end testing utility libraries.

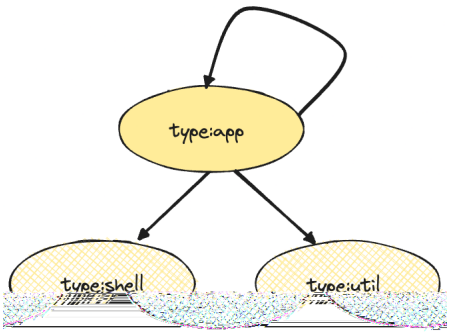
1. No project must import from an end-to-end testing project.
   This is enforced when Nx lint checks are enabled.
2. They must not have TypeScript path mappings.
3. They must not have an entry point (`index.ts`/`public_api.ts`).

Other dependency constraints are configured as described in the *Entain Nx architecture* document.

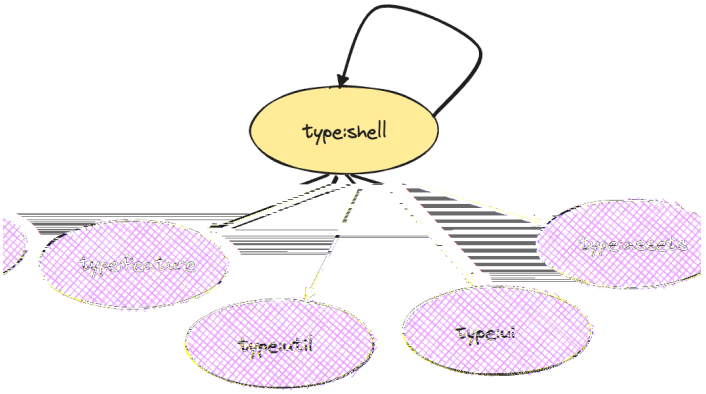
By Type

Configuring by **N ec e** will allow us to create a logic in the way projects are interacting between each other. For example, it doesn't make sense that an utility library can access a feature.
Below, a list of access that will be configured for each project type in the **e ce d e d e** linting configa**a**ration: **b**

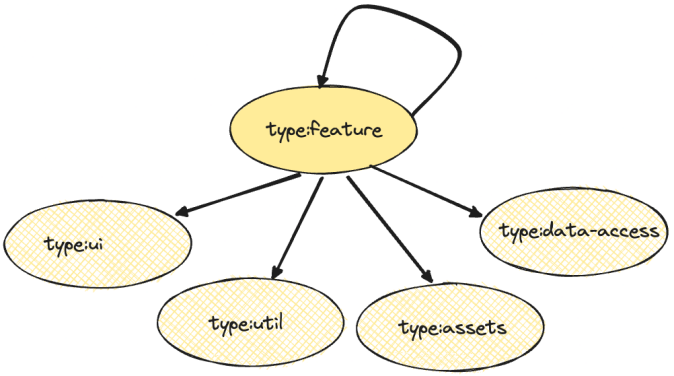
➔ The **c e** can only**a** access to the shell orchestrator and the utilities:



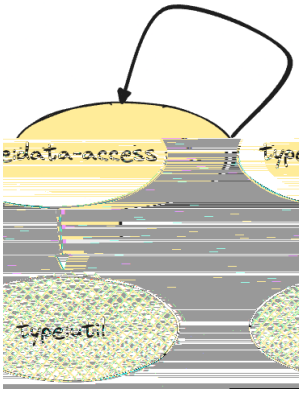
➔ The **e e** can access to all types needed for the application initialization:



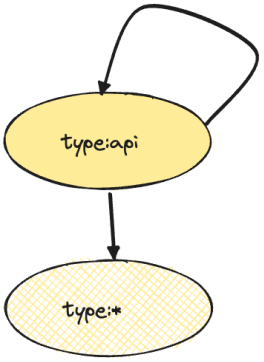
➔ The **e** **e** **e** can acc**a**ss to all types needed to build the pages:



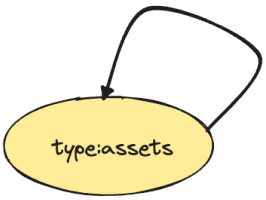
➔ The **d** **cce** **e** ca**a** **a**r**a**dy access to the utilities:

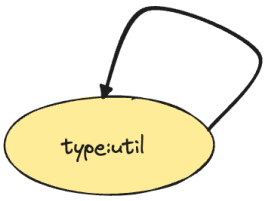


➔ The **e** can accesa to all types that contains code that you want to expose:

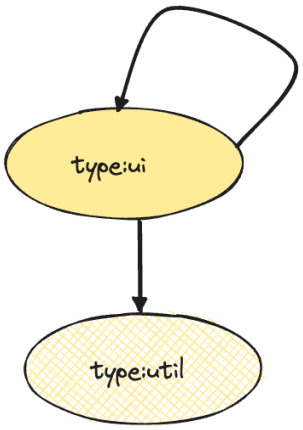


➔ The **e** and the **e** **e** can only access to themself:





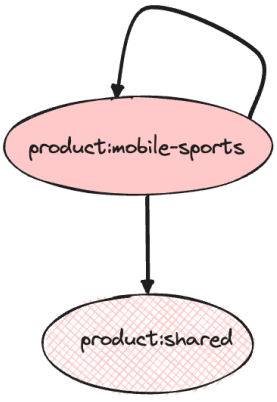
➔ The **e** can only access to the utilities:



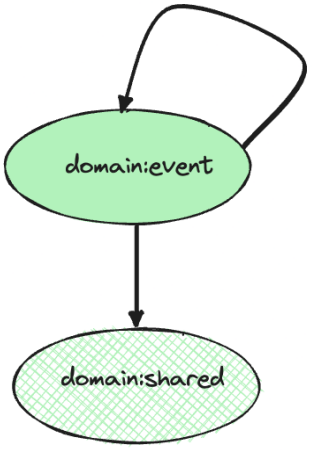
By Domain

Configuring boundaries by **B** **e** **D** will ensure that the code is well organized. All dependencies should be explicitly declared and will be exposed in the dependency graph.
Below, a list of accesses that will be configured for each project type in the **e** **ce** **d** **e** **d** **e** linting configuration: **b**

➔ The **P** **d** **c** domain can access to all domain in his scope + the shared domain:



➔ The **D** and **D** **a** domain can access to all domain in his scope + the shared domain:



Configuration

Initial global Nx lint checks enabled with the `error` severity.

```
"@nx/enforce-module-boundaries": [
  "error",
  {
    "allow": [],
    "allowCircularSelfDependency": false,
    "banTransitiveDependencies": true,
    "checkDynamicDependenciesExceptions": [],
    "checkNestedExternalImports": true,
    "enforceBuildableLibDependency": true,
    "depConstraints": [
      {
        "sourceTag": "*",
        "onlyDependOnLibsWithTags": ["*"]
      }
    ]
  }
]
```

Initial Nx lint checks in the `.eslintrc.json` workspace-wide ESLint configuration file located in the root of the Nx workspace.

As soon as any other `depConstraints` lint setting is added, remove the wildcard `depConstraints` lint setting.

1. External dependency constraints must be respected
   We enable the `banTransitiveDependencies` and `checkNestedExternalImports` Nx lint checks.
2. Project boundaries must be respected
   - Imports within an ECMAScript module's own project must be relative. We disallow circular self-dependencies by setting `allowCircularSelfDependency` to `false`.
   - Workspace projects are imported by their npm-scoped alias, not through relative or absolute imports. See the *npm scope* section. Deep imports are disallowed. Only the npm-scoped

alias is used for imports and wildcard TypeScript path mappings are banned.

3. Dynamic imports are enforced
   We keep the `checkDynamicDependenciesExceptions` list empty to prevent lazy-loaded projects from being statically loaded.
4. No exceptions
   The global `allow` list is kept empty.

Workspace toolchain

To ease migration to the Nx monorepository workspace, the toolchain used in an application repository will be supported initially. However, the target workspace toolchain is fast, modern, and based on stable first-class Nx plugins complemented by bespoke local Nx plugins.

Current toolchain

The current toolchain includes the following technologies.

- ➔ Angular
- ➔ TypeScript
- ➔ Zone.js
- ➔ Node.js
- ➔ Webpack
- ➔ Gulp
- ➔ Karma
- ➔ Playwright
- ➔ Moxxi
- ➔ Hammer.js
- ➔ Lodash

Except for documentation updates, Gulp has not been maintained since 2019H1. Karma has officially been deprecated by Google. Moxxi is an internal library.

Target toolchain

The following technologies are included in the target toolchain.

- ➔ Angular
- ➔ TypeScript
- ➔ Zone.js
- ➔ ESBuild
- ➔ First-class Nx plugins
- ➔ Local Nx plugins
- ➔ Nx custom commands
- ➔ Nx custom scripts
- ➔ Cypress feature tests
- ➔ Jest
- ➔ Prettier

- ESLint
- Angular ESLint
- user-flow
- Hammer.js
- Lodash

Workspace layout

The Nx workspace is aligned with an integrated Nx Angular workspace layout.

npm scope

The npm scope of the workspace is `@entain`.

The npm scope is configured in `nx.json` as follows.

```
{
    "//": "Other settings omitted for brevity",
    "npmScope": "entain",
}
```

Workspace npm scope configured in the `nx.json` Nx configuration file.

Nx targets

All workspace tasks are wrapped in an Nx target using a first-class, third-party, or local Nx executor, the `command` Nx target option, the `nx:run-commands` executor, or the `nx:run-script` executor. Cacheable tasks are listed in the `cacheableOperations` task runner option in the `nx.json` configuration file.

Buildable and publishable libraries

To support incremental build and serve, every library is buildable or publishable so that each application project can use incremental build and serve to speed up the build and re-build process. Buildable Angular libraries use the `@nx/angular:ng-packagr-lite` Nx executor for their `build` target to output a minimal number of bundle formats that support incremental build and serve. Publishable Angular libraries use the `@nx/angular:package` Nx executor

Buildable and publishable libraries must declare required `peerDependencies` and `dependencies` in their output `package.json` file. They can be updated programmatically but not by using the `updateBuildableProjectDepsInPackageJson` and `buildableProjectDepsInPackageJsonType` Nx generator options for the `@nx/angular:package` generator as they are in the process of being deprecated as of June 2023..

Angular applications

Angular applications use the `@nx/angular:webpack-browser` Nx executor for their `build` target with the `--parallel` option enabled to support incremental builds. Additionally, they use the `@nx/web:file-server` Nx executor for their `serve` target with the `--parallel` option enabled for incremental serve.

Inputs, outputs, and dependencies

Target *inputs* are maintained through a combination of the `targetDefaults` and `namedInputs` properties of the `nx.json` configuration file as well as the `inputs` and `namedInputs` property of the target definition in its containing `project.json` file.

Target *outputs* use `minimatch`-compatible glob patterns as needed to include only specific files for caching. When specifying the `outputs` Nx project property, take Nx's default `outputs` patterns into account. As of Nx 16.0, they are as follows.

- `{workspaceRoot}/dist/{projectRoot}`
- `{projectRoot}/build`
- `{projectRoot}/dist`
- `{projectRoot}/public`

We use the `dependsOn` Nx target property and/or the `implicitDependencies` Nx project property in a `project.json` to describe non-static dependencies between Nx targets and Nx projects, respectively.

The `{projectRoot}` and `{workspaceRoot}` placeholders are used for target inputs and outputs as appropriate.

Refer to Project configuration, Customizing Inputs and Named Inputs, and How Caching Works in the Nx documentation.

Lint rules

Lint rules recommended by Nx

We use lint rules recommended by Nx as seen in the following configuration.

```
{
  "root": true,
  "ignorePatterns": ["**/*"],
  "plugins": ["@nx"],
  "overrides": [
    {
      "files": ["*.ts", "*.tsx", "*.js", "*.jsx"],
      "rules": {
        "@nx/enforce-module-boundaries": [
          "error",
```

```
          {
            "//": "See the Module boundaries section and the Entain Nx
architecture document"
          }
        ]
      }
    },
    {
      "files": ["*.ts", "*.tsx"],
      "extends": ["plugin:@nx/typescript"],
      "rules": {}
    },
    {
      "files": ["*.js", "*.jsx"],
      "extends": ["plugin:@nx/javascript"],
      "rules": {}
    },
    {
      "files": ["*.spec.ts", "*.spec.tsx", "*.spec.js", "*.spec.jsx"],
      "env": {
        "jest": true
      },
      "rules": {}
    }
  ]
}
```

The `.eslintrc.json` workspace-wide ESLint configuration file located in the root of the Nx workspace.

Additionally, each project has the following lint rules.

```
{
  "//": "Other settings omitted for brevity",
  "overrides": [
    {
      "files": ["*.ts"],
      "rules": {
        "@angular-eslint/directive-selector": [
          "error",
          {
            "type": "attribute",
            "prefix": "<prefix>",
            "style": "camelCase"
          }
        ],
        "@angular-eslint/component-selector": [
          "error",
          {
            "type": "element",
            "prefix": "<prefix>",
            "style": "kebab-case"
          }
        ]
```

```
      },
      "extends": [
        "plugin:@nx/angular",
        "plugin:@angular-eslint/template/process-inline-templates"
      ]
    },
    {
      "files": ["*.html"],
      "extends": ["plugin:@nx/angular-template"],
      "rules": {}
    }
  ]
}
```

An `.eslintrc.json` project-specific ESLint configuration file located in the root of each Nx project.

The preceding `<prefix>` placeholder is replaced in each Nx project depending on its product dimension, for example the `vn` which is shorthand for Vanilla.

Severity level

Lint rules must either have the `"error"` severity level or the `"off"` severity level. For example, the following lint rules must be readjusted from the `"warn"` to the `"error"` severity level because they are set to the `"warn"` severity level by the `@typescript-eslint/eslint-recommended` ESLint plugin.

- `@typescript-eslint/no-explicit-any`
- `@typescript-eslint/no-non-null-assertion`
- `@typescript-eslint/no-unused-vars`

Alternatively, ESLint must be run with the `--max-warnings=0` parameter to treat lint warnings as lint errors during CI pipelines.

Compiler settings

Strict compiler settings

The following TypeScript compiler settings are applied to every project to enforce TypeScript and Angular *Strict mode*.

```
{
  "//": "Other settings omitted for brevity",
  "angularCompilerOptions": {
    "strictInjectionParameters": true,
    "strictInputAccessModifiers": true,
    "strictTemplates": true
  },
  "compilerOptions": {
    "strict": true
  }
}
```

A TypeScript configuration file enabling TypeScript and Angular Strict mode.

Consider enabling strict compiler settings one at a time. `compilerOptions.strict` and `angularCompilerOptions.strictTemplates` are both shorthands for groups of compiler settings.

> **N e**
>
> Enabling the `strict` and `strictTemplates` shorthand compiler settings enable the current compiler setting groups that they represent as well as any future compiler settings added to their respective groups.

Recommended compiler settings

The following TypeScript compiler settings are applied to every project as recommended by Angular and Nx.

```
{
  "//": "Other settings omitted for brevity",
  "angularCompilerOptions": {
    "enableI18nLegacyMessageIdFormat": false,
  }
  "compilerOptions": {
    "forceConsistentCasingInFileNames": true,
    "importHelpers": true,
    "noImplicitOverride": true,
    "noPropertyAccessFromIndexSignature": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "skipLibCheck": true,
    "skipDefaultLibCheck": true,
  }
}
```

A TypeScript configuration file set up to Nx recommendations.

ECMAScript compatibility compiler settings

The following TypeScript compiler settings are applied to every project to enforce ECMAScript compatibility.

```
{
  "//": "Other settings omitted for brevity",
  "compilerOptions": {
    "experimentalDecorators": false,
    "useDefineForClassFields": true
  }
}
```

A TypeScript configuration file enforcing ECMAScript compatibility.

To disable `experimentalDecorators`, we must not use constructor parameter decorators like Angular's `@Inject`, `@Attribute`, `@Host`, or `@Self` decorators.

To enable `useDefineForClassFields`, we avoid TypeScript constructor parameter properties as the order of their property initialization is switched from *before* ECMAScript property initializers (`"useDefineForClassFields": "false"`) to *after* ECMAScript property initializers (`"useDefineForClassFields": "true"`). Where an ECMAScript property initializer could previously refer to a TypeScript constructor parameter property (`"useDefineForClassFields": "false"`), all TypeScript constructor parameters are now `undefined` at the time of ECMAScript property initialization (`"useDefineForClassFields": "true"`).

N e

Both ECMAScript property initializers and TypeScript constructor parameter property initializers are executed before a constructor body.

Runtime environment

Starting with Angular version 15, the `target` TypeScript compiler setting of an Angular application project, a buildable Angular library, or a publishable Angular library overrides the Angular compiler.

A TypeScript configuration file adjusted to target an ECMAScript 2021 runtime environment.

Additionally, we install the `eslint-plugin-es-x` package and configure ESLint as follows.

```
{
  "//": "Other settings omitted for brevity"
  "plugins" [
    "es-x"
  ],
  "overrides": [
    {
      "files": ["*.ts", "*.js"],
      "extends": [
          "plugin:es-x/restrict-to-es2021",
          "plugin:es-x/restrict-to-es2021-intl-api"
      ],
      "rules": {
        "es-x/no-regexp-lookbehind-assertions": "error"
      }
    }
  }
}
```

An `.eslintrc.json` ESLint configuration file adjusted to support iOS Safari 16.3.

Resources

- ➔ DOCS
- ➔ The Angular Developer's Nx Handbook for definitions of conventional and common library types.
- ➔ Micro Frontends and Moduliths with Angular for definitions of conventional and common library types.
- ➔ Tiny Angular application projects in Nx workspace for assets, environments, and styles libraries.
- ➔ Shell Library patterns with Nx and Monorepo Architectures for shell library variants.