Nrwl NX - Siemens

1. What is Nrwl NX 1
2. Workspace 2
3. Applications 3
4. Libraries 5
    - Backend library 6
    - Shell library 7
    - Feature library 8
    - Data Access library 9
    - Dto library 10
    - Ui library 11
    - Util library 12
    - Api library 13
5. Development 14
    - Setup Workspace 14
    - View Dep Graph 16
    - View affected Dep Graph 16
    - Affected test, lint, build, ... 17
    - Define tags for libraries and applications 17
    - Lint dependencies between libraries and applications 18
    - Format 19

1. What is Nrwl NX

Nx is a set of extensible dev tools for monorepos, which helps you develop like Google, Facebook, and Microsoft. It has first-class support for many frontend and backend technologies, so its documentation comes in multiple flavors.

- https://nx.dev/web
- https://nx.dev/web/getting-started/what-is-nx

2. Workspace

Generate

```
npx create-nx-workspace@latest
```

Structure

- apps
  Contains all frontend (also frontend e2e) and backend applications
  - <app-name>
    - ...
- libs
  Contains all libraries for frontend and backend applications
  - <domain-name>
    - backend
      Contains all libraries of type backend for a specific domain
      - <backend-feature-name>
        - …
      - …
    - shell
      Library of type shell for a specific domain (entry point of domain)
      - ...
    - feature
      Contains all libraries of type feature for a specific domain
      - <feature-name>
        - ...
      - ...
    - data-access
      Library of type data-access for a specific domain
      - ...
    - dto
      Library of type dto for a specific domain
      - ...
    - ui
      Contains all libraries of type ui for a specific domain
      - <ui-name>
        - ...
      - ...
    - util
      Contains all libraries of type util for a specific domain
      - <util-name>
        - ...
      - ...
    - api
      Library of type api for a specific domain (export functionality to another domain)
      - ...
  - ...
- angular.json
  Contains build, serve, lint and test definitions for all applications and libraries
- workspace.json (optional)
  Contains build, serve, lint and test definitions for all applications and libraries if workspace has no Angular application (e.g. a React and Node application)
- nx.json
  Contains tags definitions used at linting (ensures relations between libraries and applications)
- tsconfig.js
  Contains the "paths" section where each library gets a "path mapping" to keep imports clean and avoid relative imports in applications ("../../../..."). This "path mapping" normally points to an "index.ts" file which is the entry point of each library.
- …

3. Applications

An application inside a Nrwl Nx Workspace can be a Web, React, NextJS, Angular, Node or NestJS project. The Nrwl Nx Workspace can consist out of different types of applications (e.g. a workspace can contain multiple Angular, React and Node projects at the same time). The only important point here is that if you are using different types of applications inside your workspace the "angular.json" file is replace by a Nrwl Nx specific "workspace.json" file. This will then contain all definitions for build, serve, lint and test similar to the "angular.json".

Generate

Application	Command
Angular	`nx g @nrwl/angular:application`
NestJS	`nx g @nrwl/nest:application`

Note: Run the above CLI commands with the flag `--dry-run` to show what will be generated without writing to disk.

Structure (Angular)

- src
  - app
    - components
      - app
        - app.component.html
        - app.component.scss
        - app.component.ts
      - ...
      - components.ts
    - routes
      - routes.ts
    - app.module.ts
  - assets
    - …
  - environments
    - …
  - index.html
  - main.ts
  - polyfills.ts
  - styles.scss
  - test-setup.ts
- jest.config.js
- proxy.conf.json
- tsconfig.app.json
- tsconfig.json
- tsconfig.spec.json
- tslint.json

Structure (NestJS)

- src
  - app
    - ...
    - app.module.ts
  - assets
    - …
  - environments
    - …
  - migrations
    - …
  - ormconfig.ts
  - ormconfig-cli.ts
  - main.ts
- jest.config.js
- tsconfig.app.json
- tsconfig.json
- tsconfig.spec.json
- tslint.json

4. Libraries

A library inside a Nrwl Nx Workspace can be used to encapsulate specific functionality which than can be reused by multiple applications or other libraries.

Types

- **backend**
  Implements backend module (controller, service, …)
- **shell**
  Implements a shell to group features and provide an entry point to the domain (routing, …)
- **feature**
  Implements a use case (view) using smart components and pipes
- **data-access**
  Implements data access (calls to BE, ngrx store, resolvers, effects, ...)
- **dto**
  Provides domain specific dto's (models, enums, …)
- **ui**
  Provides use case agnostic and thus reusable components (dumb components) and pipes
- **util**
  Provides helper functions
- **api**
  Provides functionalities (services, models, enums, …) exposed for other domains

Please note the separation between smart and dumb components here. Smart components within feature libraries are use case specific. An example is a component which allows to search for products. On contrary, dumb components don't know the current use case at all. They just receive data via inputs, display it in a specific way and emit events.

To generate a new library, use the Nrwl Nx CLI as following:

Application	Command
Angular	`nx g @nrwl/angular:library`
Node	`nx g @nrwl/node:library`

Note: Run the above CLI commands with the flag `--dry-run` to show what will be generated without writing to disk.

Backend library

Generate

```
nx g @nrwl/nest:library --name=<library-name> --directory=<domain-name>/backend
--controller --service --tags="scope:<domain-name>, type:backend"
```

Example

```
nx g @nrwl/nest:library --name=User --directory=user/backend --controller
--service --tags="scope:user, type:backend"
```

Structure

- src
  - lib
    - controllers
      - specs
        - <controller-name>.controller.spec.ts
        - …
      - < controller-name>.controller.ts
      - …
    - services
      - specs
        - <service-name>.service.spec.ts
        - …
      - <service-name>.service.ts
      - ...
    - entities
      - …
    - <domain-name>-backend-<backend-feature-name>.module.ts
  - index.ts
- jest.config.js
- tsconfig.json
- tsconfig.lib.json
- tsconfig.spec.json
- tslint.json

Shell library

Generate

```
nx g @nrwl/angular:library --name=Shell --directory=<domain-name> --prefix=lib
--style=scss --tags="scope:<domain-name>, type:shell"
```

Example

```
nx g @nrwl/angular:library --name=Shell --directory=user --prefix=lib --style=scss
--tags="scope:user, type:shell"
```

Structure

- src
  - lib
    - routes
      - routes.ts
    - <domain-name>-shell.module.ts
  - index.ts
  - test-setup.ts
- jest.config.js
- tsconfig.json
- tsconfig.lib.json
- tsconfig.spec.json
- tslint.json

Feature library

Generate

```
nx g @nrwl/angular:library --name=<library-name> --directory=<domain-name>/feature
--prefix=lib --style=scss --tags="scope:<domain-name>, type:feature"
```

Example

```
nx g @nrwl/angular:library --name=UserProfile --directory=user/feature
--prefix=lib --style=scss --tags="scope:user, type:feature"
```

Structure

Note: Inside a Feature library there is always one component that is named equally to the feature. This component is the main smart entry component.

- src
  - lib
    - components
      - <feature-name>
        - specs
          - <feature-name>.component.spec.ts
          - <feature-name>.component.comp.spec.ts
        - <feature-name>.component.html
        - <feature-name>.component.scss
        - <feature-name>.component.ts
      - <feature-name>-<component-name>
        - specs
          - <feature-name>-<component-name>.component.spec.ts
          - <feature-name>-<component-name>.component.comp.spec.ts
        - <feature-name>-<component-name>.component.html
        - <feature-name>-<component-name>.component.scss
        - <feature-name>-<component-name>.component.ts
      - ...
      - components.ts
    - pipes
      - <pipe-name>
        - specs
          - <pipe-name>.pipe.spec.ts
        - <pipe-name>.pipe.ts
      - ...
      - pipes.ts
    - constants
      - constants.ts
    - routes
      - routes.ts
    - services
      - <service-name>.service.ts
      - ...
    - models
      - <model-name>.model.ts
      - ...
    - enums
      - <enum-name>.enum.ts
      - ...
    - <domain-name>-feature-<feature-name>.module.ts
  - index.ts
  - test-setup.ts
- jest.config.js
- tsconfig.json
- tsconfig.lib.json
- tsconfig.spec.json
- tslint.json

Data Access library

Generate

```
nx g @nrwl/angular:library --name=DataAccess --directory=<domain-name>
--prefix=lib --style=scss --tags="scope:<domain-name>, type:data-access"
```

Example

```
nx g @nrwl/angular:library --name=DataAccess --directory=user --prefix=lib
--style=scss --tags="scope:user, type:data-access"
```

Structure

- src
  - lib
    - state
      - <state-name>
        - mocks
          - <state-name>.facade.mock.ts
        - <state-name>.actions.ts
        - <state-name>.effects.ts
        - <state-name>.facade.ts
        - <state-name>.reducer.ts
        - <state-name>.selectors.ts
      - ...
      - <module-name>.module.state.ts
    - services
      - resolvers
        - <service-name>.resolver.service.ts
        - ...
      - <service-name>.service.ts
      - ...
    - <domain-name>-data-access.module.ts
  - index.ts
  - test-setup.ts
- jest.config.js
- tsconfig.json
- tsconfig.lib.json
- tsconfig.spec.json
- tslint.json

Dto library

Generate

```
nx g @nrwl/node:library --name=Dto --directory=<domain-name>
--tags="scope:<domain-name>, type:dto"
```

Example

```
nx g @nrwl/node:library --name=Dto --directory=user --tags="scope:user, type:dto"
```

Structure

- src
  - lib
    - constants
      - constants.ts
    - models
      - <model-name>.model.ts
      - ...
    - enums
      - <enum-name>.enum.ts
      - ...
  - index.ts
  - test-setup.ts
- jest.config.js
- tsconfig.json
- tsconfig.lib.json
- tsconfig.spec.json
- tslint.json

Ui library

Generate

```
nx g @nrwl/angular:library --name=<library-name> --directory=<domain-name>/ui
--prefix=lib --style=scss --tags="scope:<domain-name>, type:ui"
```

Example

```
nx g @nrwl/angular:library --name=UserDetailsTable --directory=user/ui
--prefix=lib --style=scss --tags="scope:user, type:ui"
```

Structure

- src
  - lib
    - components
      - <component-name>
        - specs
          - <component-name>.component.spec.ts
          - <component-name>.component.comp.spec.ts
        - <component-name>.component.html
        - <component-name>.component.scss
        - <component-name>.component.ts
      - ...
      - components.ts
    - pipes
      - <pipe-name>
        - specs
          - <pipe-name>.pipe.spec.ts
        - <pipe-name>.pipe.ts
      - ...
      - pipes.ts
    - models
      - <model-name>.model.ts
      - ...
    - enums
      - <enum-name>.enum.ts
      - ...
    - <domain-name>-ui-<ui-name>.module.ts
  - index.ts
  - test-setup.ts
- jest.config.js
- tsconfig.json
- tsconfig.lib.json
- tsconfig.spec.jso
- tslint.json

Util library

Generate (typescript only)

```
nx g @nrwl/node:library --name=<library-name> --directory=<domain-name>/util
--tags="scope:<domain-name>, type:util"
```

Example

```
nx g @nrwl/node:library --name=UserNameFormat --directory=user/util
--tags="scope:user, type:util"
```

Structure (typescript only)

- src
  - lib
    - ...
      - ...
    - ...
  - index.ts
  - test-setup.ts
- jest.config.js
- tsconfig.json
- tsconfig.lib.json
- tsconfig.spec.json
- tslint.json

Generate (with Angular module)

```
nx g @nrwl/angular:library --name=<library-name> --directory=<domain-name>/util --
prefix=lib --style=scss --tags="scope:<domain-name>, type:util"
```

Example

```
nx g @nrwl/angular:library --name=UserNameFormat --directory=user/util --
prefix=lib --style=scss --tags="scope:user, type:util"
```

Structure (with Angular module)

- src
  - lib
    - services
      - <service-name>.service.ts
      - ...
    - <domain-name>-util-<util-name>.module.ts
  - index.ts
  - test-setup.ts
- jest.config.js
- tsconfig.json
- tsconfig.lib.json
- tsconfig.spec.json
- tslint.json

Api library

Generate

```
nx g @nrwl/angular:library --name=Api --directory=<domain-name> --prefix=lib --
style=scss --tags="scope:<domain-name>, scope:<domain-name>/api, type:api"
```

Example

```
nx g @nrwl/angular:library --name=Api --directory=user --prefix=lib --style=scss -
-tags="scope:user, scope:user/api, type:api"
```

Structure

- src
  - lib
    - <domain-name>-api.module.ts
  - index.ts
  - test-setup.ts
- jest.config.js
- tsconfig.json
- tsconfig.lib.json
- tsconfig.spec.json
- tslint.json

5. Development

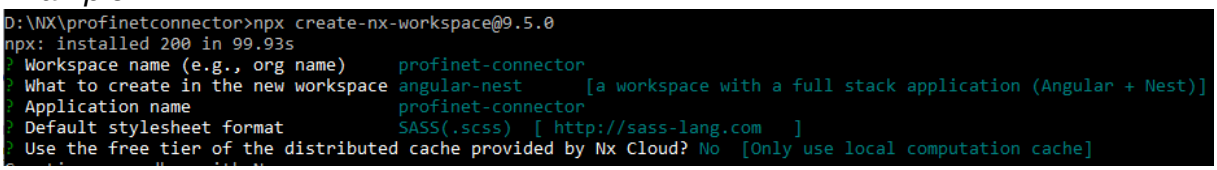
Setup Workspace

The following steps should give a short overview about the setup of a new workspace. A well setup workspace as reference can be found here: https://code.siemens.com/reusable-frontend-components/sample-workspace

1. **Run `npx create-nx-workspace@<version>` to create a new workspace**

   During the setup you get asked several questions like *"Workspace name?", "What to create in the new workspace?", …*

   Example

   In our cases we most common need a preset Angular application (+ optional Nest backend) with "SASS(.scss)" as *"Default stylesheet format"*.

2. **Check packages in package.json**

   Make sure to remove all **^** and **~** at the package versions (necessary because we need explicit versions for the *Siemens Clearing process*).

   Additionally add needed packages like (ngrx, …) and ensure consistent version usage across the apps (maybe copy from other apps package.json to speed up *Siemens Clearing process* because those versions are already "cleared").

3. **Adjust scripts in package.json**

   For our daily development we add the following scripts usually:

   - `"affected:lint-fix": "nx affected:lint --parallel --fix "`
   - `"affected:test-failed": "nx affected:test --parallel --only-failed"`

   Furthermore, adding the "`--parallel`" flag to the existing "affected:lint" and "affected:test" scripts is highly recommended!

4. **Set the default affected base to "development" at "nx.json"**

```
"affected": {
    "defaultBase": "development"
}
```

5. **Setup prettier**

   Prettier is used to format the files correctly and so we must apply the same settings as at our other workspaces. (Just copy settings from an existing projects “.prettierrc” file)

6. **Setup TSLint**

   TSLint is used to lint the files and so we must apply the same rules as at our other workspaces. (Just copy rules from an existing projects “tslint.json” file)

7. **Remove boilerplate generated by Nrwl Nx**

   Nrwl Nx provides us a demo setup with some components and so on which needs to be removed or adapted by our needs.

8. **Done**

   The basic workspace setup is done, and the next steps depend on your needs.

   Most common steps which will follow are:

   Backend
   - Setup DB
   - Setup TypeORM
   - …

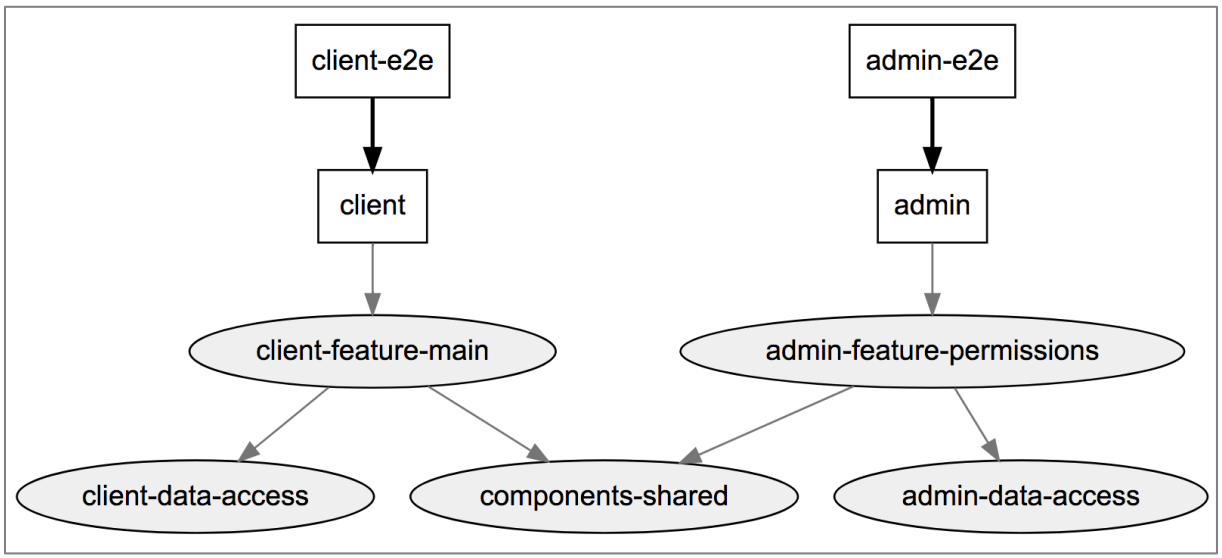
   Frontend:
   - Add Reusable Components Libraries
   - Setup Routing
   - Setup App Layout
   - Setup Translation Handling
   - …

View Dep Graph

To get an overview on how the libraries and applications are depending on each other Nrwl Nx can show you a so called "Dep Graph" to visualize the dependencies.

Run `nx dep-graph` to see the dependency graph.

Example

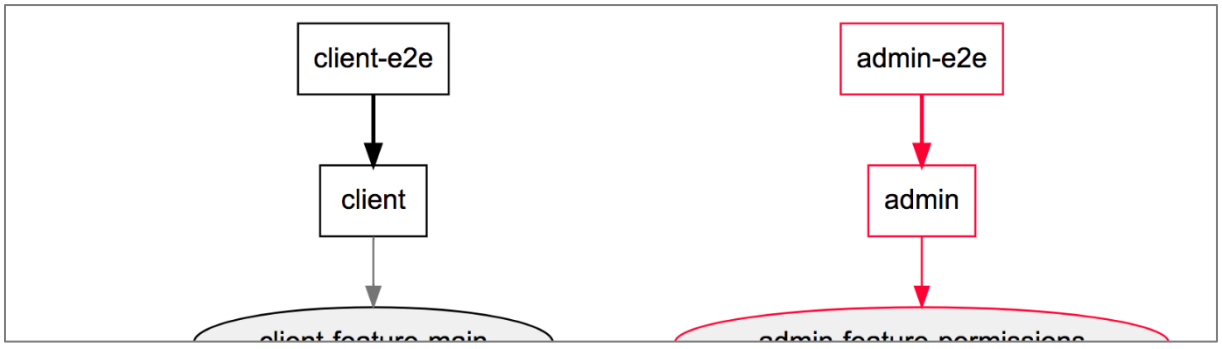


View affected Dep Graph

To get an overview on which libraries and applications are affected by a change you made Nrwl Nx can determine what files have changed based on GIT.

Run `nx affected:dep-graph --base=master --head=HEAD` to see the dependency graph with highlighted affected libraries and applications.
The `--base` defaults to `master` and `--head` defaults to `HEAD`, so when running it locally you can usually omit it.
Nx will find the most common ancestor of the base and head SHAs and will use it to determine what has changed between it and head.

Example



Affected test, lint, build, ...

Since Nrwl Nx can determine the files changed based on GIT and therefore knows which libraries and applications have changed, it can also run testing, linting and building on those affected libraries and applications only.

Run `nx affected:apps` to print the affected applications.
Run `nx affected:libs` to print the affected libraries.
Run `nx affected:build` to build the affected libraries and applications.
Run `nx affected:lint` to lint the affected libraries and applications.
Run `nx affected:test` to test the affected libraries and applications.
Run `nx affected:e2e` to e2e test the affected libraries and applications.

Note: Run the above CLI commands with the flag `--base=<branch-name> (e.g. --base=origin/xyz)` to determine files that have changed between the current and the defined branch.

Define tags for libraries and applications

As already mentioned in the libraries section, Nrwl Nx suggests different categories for libraries. To define the category and corresponding domain, the global "nx.json" file contains a section for each application and library (project) where an array of tags can be defined.

Types

- type:app-backend
- type:app-e2e
- type:app
- type:backend
- type:shell
- type:feature
- type:data-access
- type:dto
- type:ui
- type:util
- type:api

Scopes

For scopes the format of the tag should be "scope:<scope-name>".

Ideally each library and application does have at least two tags. One which defines the type and another one which defines the scope.

Lint dependencies between libraries and applications

To ensure the right correlation between a library and another library or between a library and an application Nrwl Nx provides a linting rule called "nx-enforce-module-boundaries". This rule is placed in the global "tslint.json". It contains so called "depConstraints" where the dependency constraint is based on the "tags" of each library or application defined at "nx.json" file.

Example

```
"nx-enforce-module-boundaries": [
      true,
      {
        "depConstraints": [
          {
            "sourceTag": "type:app-backend",
            "onlyDependOnLibsWithTags": ["type:backend"]
          },
          {
            "sourceTag": "type:app-e2e",
            "onlyDependOnLibsWithTags": ["type:app"]
          },
          {
            "sourceTag": "type:app",
            "onlyDependOnLibsWithTags": ["type:shell", "type:feature",
"type:util"]
          },
          {
            "sourceTag": "type:backend",
            "onlyDependOnLibsWithTags": ["type:dto", "type:util"]
          },
          {
            "sourceTag": "type:shell",
            "onlyDependOnLibsWithTags": ["type:feature", "type:data-
access", "type:dto", "type:util", "type:api"]
          },
          {
            "sourceTag": "type:feature",
            "onlyDependOnLibsWithTags": ["type:data-
access", "type:dto", "type:ui", "type:util", "type:api"]
          },
          {
            "sourceTag": "type:data-access",
            "onlyDependOnLibsWithTags": ["type:data-
access", "type:dto", "type:util"]
          },
          {
            "sourceTag": "type:dto",
            "onlyDependOnLibsWithTags": ["nothing"]
          },
          {
            "sourceTag": "type:ui",
```

```
            "onlyDependOnLibsWithTags": ["type:dto", "type:ui", "type:util", "type
:api"]
          },
          {
            "sourceTag": "type:util",
            "onlyDependOnLibsWithTags": ["type:dto", "type:util"]
          },
          {
            "sourceTag": "type:api",
            "onlyDependOnLibsWithTags": ["type:data-
access", "type:dto", "type:util"]
          },
          {
            "sourceTag": "scope:<domain-name>",
            "onlyDependOnLibsWithTags": ["scope:<domain-name>", "scope:shared"]
          },
          {
            "sourceTag": "scope:shared",
            "onlyDependOnLibsWithTags": ["scope:shared"]
          },
        ],
        "enforceBuildableLibDependency": true
      }
    ]
```

Format

To format files the same way across all IDEs and personalized setups Nrwl NX does support Prettier out of the box. Each workspace contains a ".prettierrc" file to define general code style settings.

Run `nx format:write` to update the format of all files or `nx format:check` to check which files aren't formatted correctly.

It's highly recommended to setup your IDE to format files on saving directly. Checkout Prettier documentation on how to setup:
https://prettier.io/docs/en/editors.html