RealWorld example app

This project is already migrated to Angular v17 and to the new Control Flow and Deferred Loading. I'm also migrating it from NgRx Store to NgRx Signal Store. The Auth library, the Article library and the Profile library have already migrated. You can already have a look.

netlify success

> **Angular, ngrx/platform, nrwl/nx codebase containing real world examples (CRUD, auth, advanced patterns, etc) that adheres to the RealWorld spec and API.**

Demo RealWorld

This codebase was created to demonstrate a fully fledged fullstack application built with Angular, ngrx/platform, nrwl/nx including CRUD operations, authentication, routing, pagination, and more.

We've gone to great lengths to adhere to the Angular community styleguides & best practices.

For more information on how to this works with other frontends/backends, head over to the RealWorld repo.

Functionality overview

The example application is a social blogging site (i.e. a Medium.com clone) called "Conduit". It uses a custom API for all requests, including authentication.

General functionality:

- Authenticate users via JWT (login/signup pages + logout button on settings page)
- CRU* users (sign up & settings page - no deleting required)
- CRUD Articles
- CR*D Comments on articles (no updating required)
- GET and display paginated lists of articles
- Favorite articles
- Follow other users

The general page breakdown looks like this:

- Home page (URL: /#/)
  - List of tags
  - List of articles pulled from either Feed, Global, or by Tag
  - Pagination for list of articles
- Sign in/Sign up pages (URL: /#/login, /#/register)
  - Uses JWT (store the token in localStorage)
  - Authentication can be easily switched to session/cookie based
- Settings page (URL: /#/settings)
- Editor page to create/edit articles (URL: /#/editor, /#/editor/article-slug-here)
- Article page (URL: /#/article/article-slug-here)
  - Delete article button (only shown to article's author)
  - Render markdown from server client side
  - Comments section at bottom of page
  - Delete comment button (only shown to comment's author)
- Profile page (URL: /#/profile/:username, /#/profile/:username/favorites)
  - Show basic user info
  - List of articles populated from author's created articles or author's favorited articles

Commands

Run the application

```
npm run start
```

Unit tests

Run all the tests: `nx run-many -t test`

Lint

```
nx run-many -t lint
```

Architecture

Folders/Libs structure For this project I created a monorepo. There is one app for the moment (conduit) which consumes the libraries under the libs folder.

The folder structure is:

```
├──
 libs│├──
    articles││├──
       data-access││├──
       feature-article-edit││├──
       feature-article││├──
       feature-articles-list│├──
    auth││├──
       data-access││├──
       feature-auth│├──
    core││├──
       api-types││├──
       error-handler││├──
       http-client││├──
       forms│├──
    profile││├──
       data-access││├──
       feature-profile│├──
    ui││├──
       components
```

I used two classifiers to name my libraries. The first classifier is the `scope` and the second the `type` . The main reason is that I want every developer when he looks a library to understand where this library can be used and which kind of services/components/etc contains.

The `scope` is the section (domain) of the app the library can be used. It gives a clear indication that a feature belongs to a specific domain. For example the libraries under `users` scope, are used in the users and favourite users pages. The ibraries under the `core` scope can be reused between all the sections of the app. ***The `core` scope will be rename soon to `shared`***.

The `type` indicates the purpose of a library. I have used a number of different types (feature, data-access, ui, api-types) The `feature-...` type contains smart components. These are components which enable the communication with the data-sources (most likely they inject api services). The `data-access` type contains code for interacting with the server. The `ui` type contains dumb (presentational) components. These components are reusable in the scope of this library

Standalone components I have used only standalone components. You won't see any modules in the app.

Lazy loaded components As you can see from the route configuration, the two main pages in the app are loaded lazily. This will make the initial loading time of the app faster.

```
 {
  path: '',
  redirectTo: 'home',
  pathMatch: 'full',
},
{
  path: 'home',
  loadChildren: () => import('@realworld/home/src/lib/home.routes').
      then((home) => home.HOME_ROUTES),
},
{
  path: 'login',
  loadComponent: () => import('@realworld/auth/feature-auth').then((m)
      => m.LoginComponent),
},
{
  path: 'register',
  loadComponent: () => import('@realworld/auth/feature-auth').then((m)
      => m.RegisterComponent),
},
{
  path: 'article',
  loadChildren: () => import('@realworld/articles/article').then((m) =>
       m.ARTICLE_ROUTES),
},
{
  path: 'settings',
  loadComponent: () =>
    import('@realworld/settings/feature-settings').then((settings) =>
       settings.SettingsComponent),
},
{
  path: 'editor',
  loadChildren: () => import('@realworld/articles/article-edit').then((
      article) => article.ARTICLE_EDIT_ROUTES),
  canActivate: [authGuard],
},
{
  path: 'profile',
  loadChildren: () => import('@realworld/profile/feature-profile').then
      ((profile) => profile.PROFILE_ROUTES),
},
```

State management **TBD**

The smart-dumb components design pattern for the components: There is a clear distinction in the codebase between the smart and dumb components. The main reason behind this decision is that I want most of my components to be reusable and easier to be tested. That means that they should not have dependencies and they just consume the data they get from the smart component. Also it makes clearer a compoenent's responsibility.

Avoid using external dependencies As you can see in the package.json, we didn't include external libraries, like `angular-material`, libs for the ui components, state management,etc. The reason is that it might be tempting to use a library like this in the short term to develop something fast, but in the long term they can introduce many problems:

- opinionated styles
- make the migration to newer versions of Angular more difficult
- not full control on them

Testing **TBD**