Angular Dev

- HowTos
  - How to Override Styles of Third-Party Components
  - How to Override Property Type In Derived Interfaces
  - How to Set Properties of a Web Component
  - Angular: Sorting Arrays

- Unsorted
- Angular Monorepo Setup
- Developing in Multiple Angular Versions
- NVM - Node.js Version Manager
- Building for Multiple Environments
- Angular Material Dark Mode Theming
- CSS Reset
- V19 Sharing DI Services
- HttpClientModule Deprecation
- Angular: App Startup (pre-V19)
- Angular: How to Return a Promise
- Angular: App Startup (V19 and later)
- Angular Workspace Additional Setup
- Common Imports
- Angular Provider Usage
- VSCode CORS
- Angular: Add Query Parameters without Router
- Angular Dynamic Routing
- Angular: Get Button Name
- Angular Context Menu
- Typescript This

- HTML CSS Prevent Text Selection
- Swapping Side-Nav Groups
- Angular: Access CSS Variables
- Web Push Notifications

HowTos

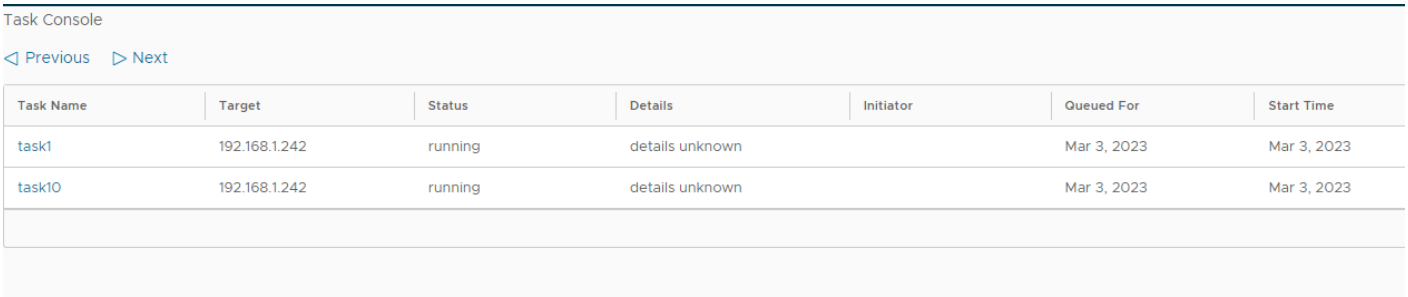
HowTos

How to Override Styles of Third-Party Components

This is a quick how to, for overriding the styles used inside third party angular components.

For example: The Clarity angular library includes a datagrid (clr-datagrid), which has an obligatory margin at its top and bottom, which creates too much whitespace when attempting to visually add controls to the datagrid.

Note how the Previous and Next buttons sit too high off the top of the datagrid to visually belong to it.



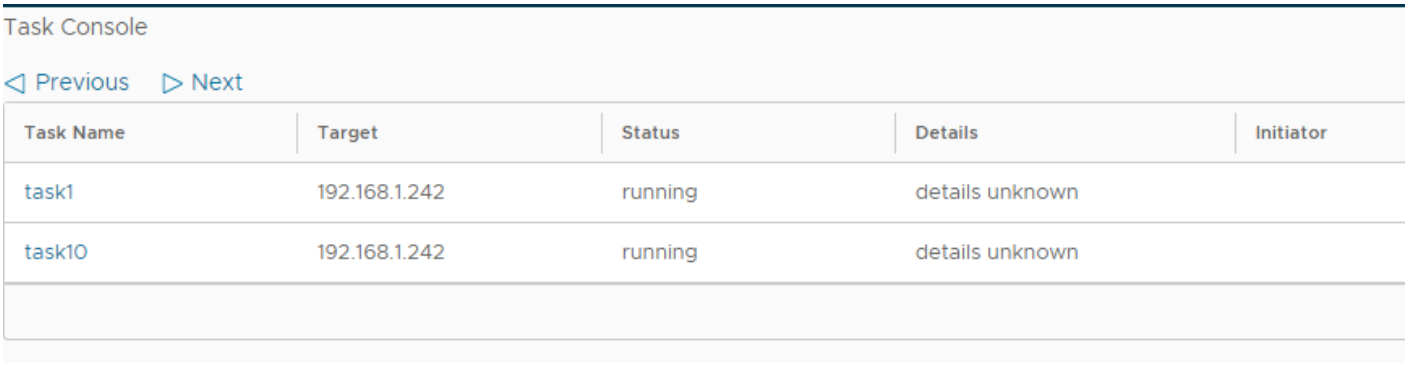
But, adding a global style (in the global styles.scss file) can override the class style of a third party library.

Here's a style in the global style file that gets rid of the margin above the clarity datagrid, fixing the above problem:

```
// This is a style override used by the tasks-page componentt.
// It overrides a style of the Clarity datagrid class so that the task page component can place its Previous and Next buttons directly on top of the datagrid.
.task-page-datagrid .datagrid {
    margin-top: 0px;
}
```

NOTE: an additional class name was added, so that our global style override would only affect the desired datagrid instance in our application.

Adding this global style got rid of the top margin of the above datagrid instance, making it look better... like this:



This override method was taken from here: Overriding CSS properties of third-party components in Angular

How to Override Property Type In Derived Interfaces

Sometimes, an interface is created that the wrong datatype for a property.
This can be because the interface is from a third-party or because you are composing a more specific interface type and need to swap out more generic properties of the base type.

Taken from here: https://bobbyhadz.com/blog/typescript-override-interface-property

Since TypeScript is compiled to javascript, it includes a utility, called Omit, that can do the type swapping work during compilation. An Omit declaration replaces the standard extends declaration of the interface by including a list of properties to be left out.

The Omit utility type constructs a new type by removing the specified keys (as a pipe-delimited list) from the existing type.

Here is an example base interface, and a derived interface, using Omit, to change the data types of the base:

index.ts

```ts
interface Location {
  address: string;
  x: number;
  y: number;
}

interface SpecificLocation extends Omit<Location, 'address'> {
  address: {
    country: string;
    city: string;
    street: string;
  };
}

const example: SpecificLocation = {
  x: 5,
  y: 10,
  address: {
    country: 'Chile',
    city: 'Santiago',
    street: 'Example street 123',
  },
};
```

How to Set Properties of a Web Component

Attribute binding can get complex. Here’s a running list of examples of how to do it.

To set properties on a Web Component use the Angular `[property]` binding syntax. To listen to events use the Angular `(event)` binding syntax.

```
<!--
- status - attribute style hook
- [closable] - setting the 'closable' property on the element
- (closeChange) - listen for the 'closeChange' custom event
-->

<cds-alert status="info" [closable]="true" (closeChange)="log($event.detail)">
  Hello World
</cds-alert>
```

Angular: Sorting Arrays

Here’s a simple technique for sorting an array of objects.

It accepts a simple function that returns a 1 or -1 result.

```
// The questions property is an array of objects with a display order property.
// We want them to be ordered ascending.
// NOTE: The sort method returns the sorted array.
let ql = this.questions.sort((a, b) => (a.displayOrder < b.displayOrder) ? -1 : 1);
```

This will sort descending:

```
// The questions property is an array of objects with a display order property.
// We want them to be ordered descending.
// NOTE: The sort method returns the sorted array.
let ql = this.questions.sort((a, b) => (a.displayOrder > b.displayOrder) ? -1 : 1);
```

Unsorted

Angular Monorepo Setup

Here's some notes on how to develop Angular in a Monorepo, with a specific (non-global) Angular version.

What this means is, we will install the desired version of Angular, specifically for the application, so that multiple versions can exist on our development machine, without interference.

What is A MonoRepo?

A monorepo provides the ability to contain multiple projects in one workspace.

Good references for this pattern: Monorepo Pattern: Setting up Angular workspace for multiple applications in one single repository

Angular Workspaces: Multi-Application Projects

Benefits

Several aspects of a monorepo are different from a single-project structure. Specifically, lots of elements are now shared. This means there are common config files, such as: tsconfig.json, package.json, angular.json, etc. These files would normally sit at the project level, but are now shared by all projects in a monorepo.

The monorepo gets created with a shared node_modules folder. This saves space by not requiring several gigs of disk for each project’s copy of referenced libraries. This has the benefit of saving disk space, AND the greater benefit of ensuring consistent dependency versioning when using components between projects.

The monorepo has a shared dist folder, where all library and app builds are published. No direct benefit comes from this. It’s just nice to have things in one spot.

A good benefit for a monorepo is that it can leverage a shared assets folder. This is good for projects that use the same icons and images, allowing them all to use a single set of icon and image files, preventing duplication and folder sync problems (for traditional project structures).

Benefits to Library Development

The best benefit for a monorepo is being able to create multiple projects for developing a component library, who all share the same dependencies and rigid pathing. Specifically, this allows you to create multiple projects for a single purpose, like the following:

- Dev Project - A project for developing new components that will be part of a published library
- Library Project - to hold the pristine components. This project is the one that gets directly built and published
- Testing Project - a project that contains all the logic for unit or integration testing of the library

Doing this for each library or app you create, allows you to minimize pollution of your main app or library.

Downside

The downside to using a monorepo, is a little extra complexity in setup (covered by this guide) and the need to explicitly “enter” a project: specifically, navigating into a project folder for editing, calling it by name during builds, etc…

Creating a MonoRepo

Here are steps to create a MonoRepo that uses a local Angular CLI version.

There’s a little bit of indirection with first creating a monorepo. And, that has to do with the creation of the root workspace.
The root workspace is the container of the shared node_modules, projects, Dist, and common config.

Considerations for New Projects

1. See this page for Angular version compatibility, before choosing the version you will use:

Developing in Multiple Angular Versions

2. Once you decide on the version of Angular, you need to ensure your development host includes NVM, to manage your Node.js version.

> Follow the instructions, here, to setup NVM: Node.js Version Support

The above link includes instructions for how to setup and swap in the desired version of Node.js.

3. Once NVM is installed and you've had it install the Angular-compatible Node.js version, you can continue.

4. Tell NVM to swap in the compatible version of Node.js.
For example, Angular 17 works with Node.js v20.18.3:

```
nvm use 20.18.3
```

Angular 19 works with this Node.js version:

```
nvm use 22.14.0
```

5. Before creating your project, make sure your development host is using the correct NPM repository.
Currently, we have a NPM repository running on the house build server, that holds all build packages.
As well, it proxies all package requests to the public NPM repository.
So, be sure that your dev host is configured to use it.

> NOTE: See this page for details: Private NPM Repository

If needed, use this command to point all projects to it:

```
npm set registry https://npmrepo.ogsofttech.com:4873
```

To confirm that the house NPM repository is configured, use this:

```
npm config get registry
```

If setup, it will respond with this:

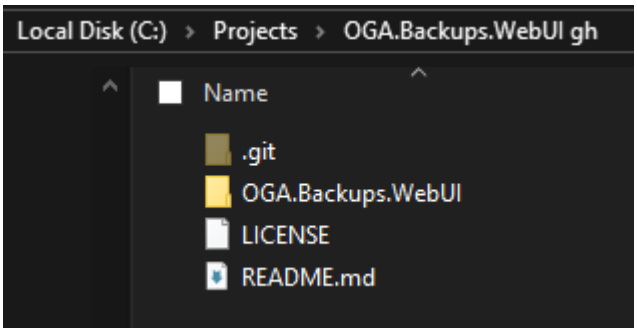
6. Create a baseline GitHub project, and note its url

For example, we have a GH project at: LeeWhite187/OGA.Backups.WebUI

7. Create a folder in your dev host and checkout the GH project to it.

8. Create a subfolder inside the checkout root with the same name as the GH project.

NOTE: This folder will be adjacent to the checked out LICENSE and README.md, like this:



The created folder will hold the dev workspace, assets folder, library projects, etc.

9. Open a command window, and navigate to the created subfolder.

10. Execute this command in the folder, to create monorepo workspace, with an Angular 17 intent:

```
npx @angular/cli@17 new dev-workspace --create-application=false
```

> NOTE: If the above hangs, you may need to update your NPM version, with this:
> `npm install -g npm`

> NOTE: Also, if the above hangs, you can run the command with the '--skip-install' switch, and execute the 'npm install' command separately, to troubleshoot what's wrong.
> Here's what the same command looks like with the skip install switch:
> `npx @angular/cli@17 new dev-workspace --create-application=false --skip-install`
>
> Or, for Angular 19.2.3:
> `npx @angular/cli@19 new dev-workspace --create-application=false --skip-install`

> NOTE: If the above command continues to fail, at the npm install, then the house NPM package service may be down.
>
> You can check on the NPM package repository, here: https://npmrepo.ogsofttech.com:4873/

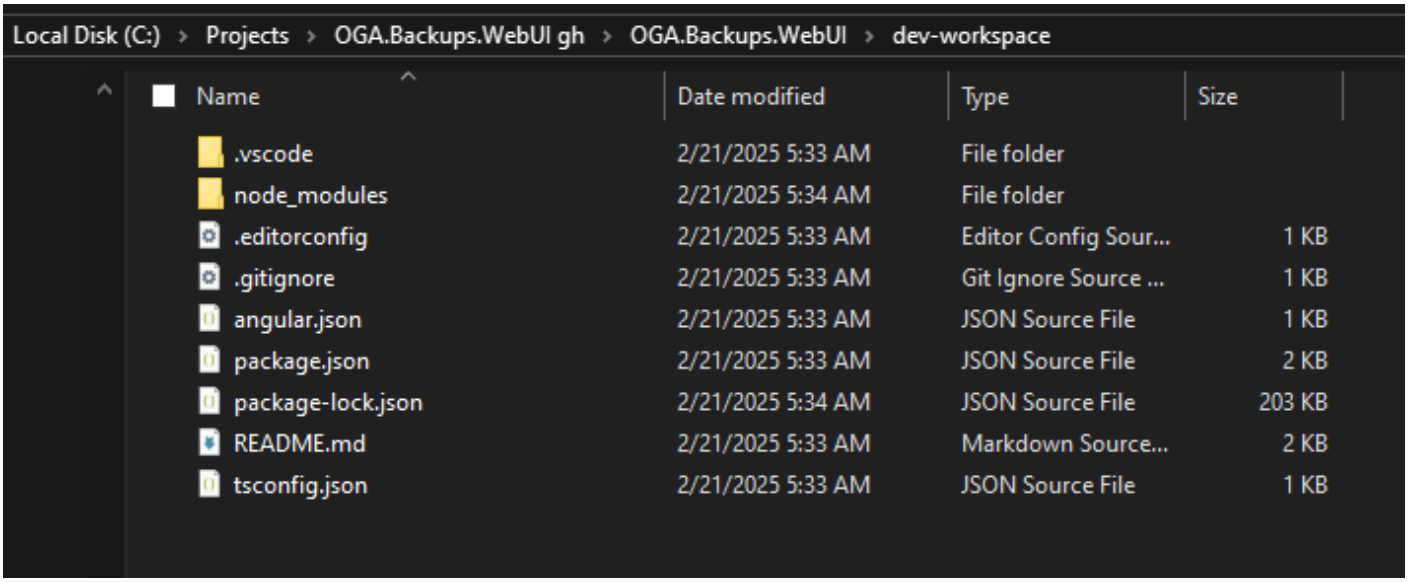
The above creates a root monorepo workspace named, 'dev-workspace', without an application, and preps it for a local Angular 17 (or 19) CLI.

> NOTE: The above command creates a subfolder inside the working directory, with the workspace name.

> NOTE: Specify a different angular cli version if needed.

NPX will churn for a bit, downloading packages and populating the workspace.

11. Once finished, change to the workspace folder (with a cd command).
The workspace will look something like this:



12. Once your command prompt is changed to the workspace folder, you can install the desired Angular version for the monorepo, with this:

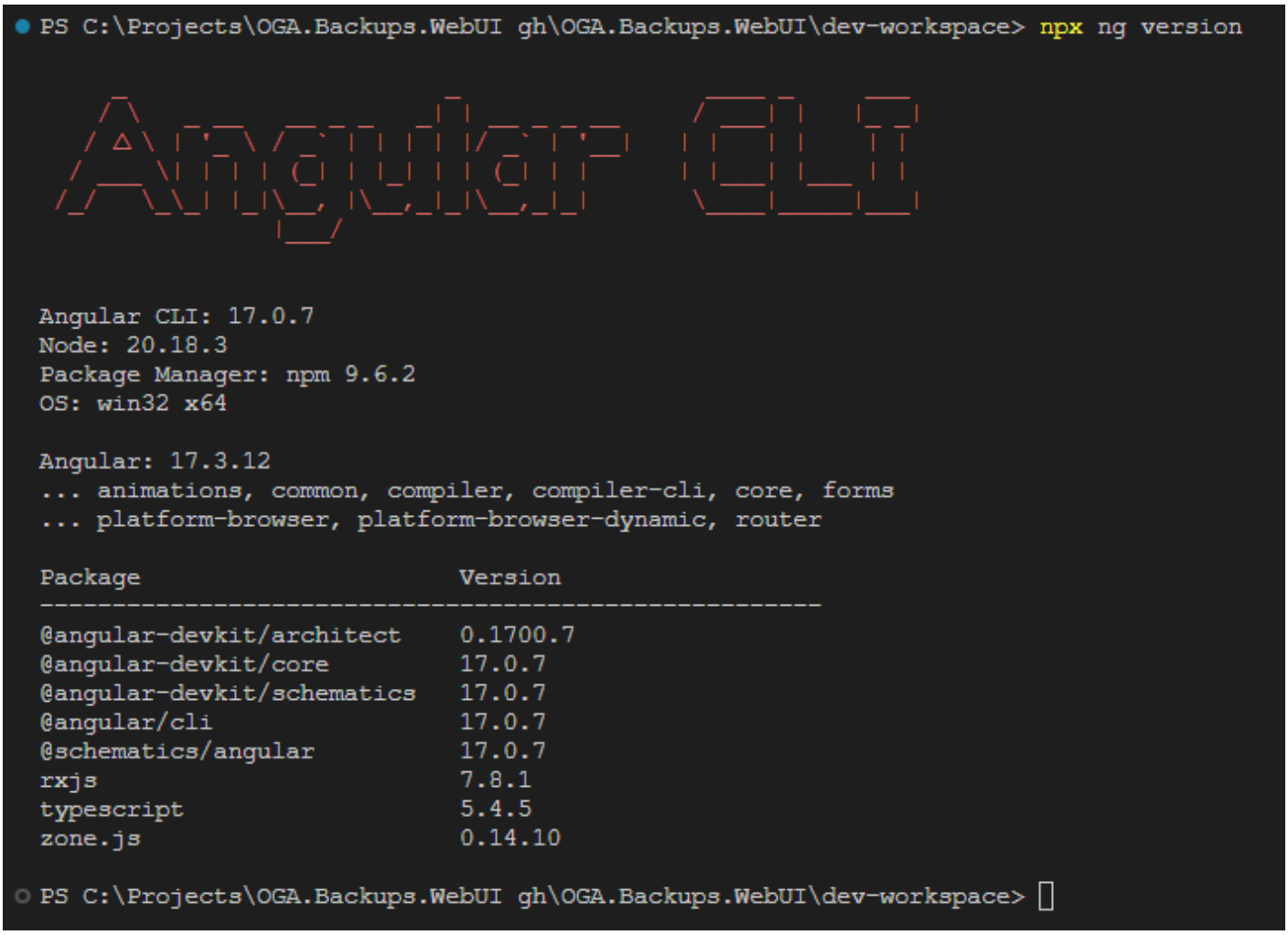
`npm install --save-dev @angular/cli@17.3.10`

For a new Angular 19 project, use this:
`npm install --save-dev @angular/cli@19.2.3`

You can open the workspace folder in VSCode, and continue.

With the workspace is open in a command line, you can verify the local Angular version of the monorepo, with this:

```
npx ng version
```



13. Open the README.md file in the workspace root, and add these lines to it, as a reminder of things you must do each time it is checked out:

> On checkout, you will need to do the following things:
>
> Tell NVM to swap in the needed Node.js version for the project:
>
> nvm use 20.18.3
>
> Pull down any packages needed for building, with this:
>
> npm install

The above includes the command to swap in the Node.js version, compatible with the project's Angular version.
And, the other downloads packages for building.

14. Update the README.md to list usage of npx ng, instead of ng.
This is required, since we are using a local Angular version.

Doing so, means the following changes to ng command usage:

- 'ng serve' becomes 'npx ng serve'
- 'ng generate' becomes 'npx ng generate'
- 'ng build' becomes 'npx ng build'

- 'ng test' becomes 'npx ng test'
- 'ng e2e' becomes 'npx ng e2e'
- 'ng help' becomes 'npx ng help'

15. Enforce local CLI usage, by adding a file to the workspace root, called: .npmrc.
Give it this content:

```
engine-strict=true
```

This will force npm to use the locally installed Angular CLI.

16. Open package.json, and update the scripts to use 'npx ng', instead of 'ng'.
Like this:

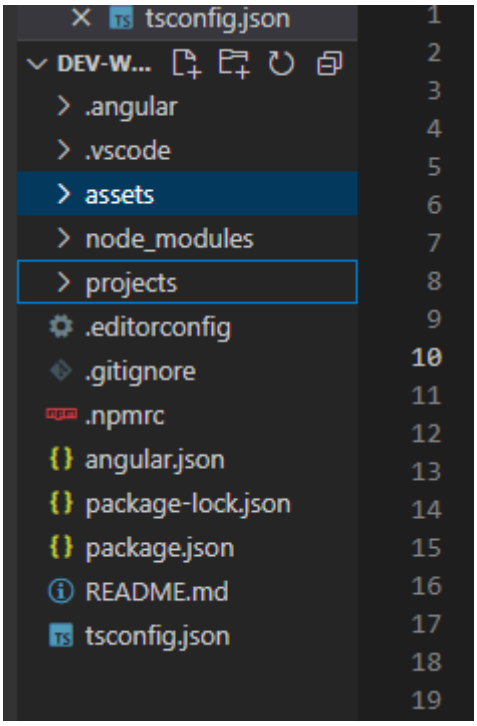
```
  "scripts": {
    "ng": "ng",
    "start": "npx ng serve",
    "build": "npx ng build",
    "watch": "npx ng build --watch --configuration development",
    "test": "npx ng test"
  },
```

17. In order for the projects of your workspace to reference any shared assets or libraries, you must set the baseUrl in the tsconfig.json file, like this:

```
/* To learn more about this file see: https://angular.io/config/tsconfig. */
{
  "compileOnSave": false,
  "compilerOptions": {
    "baseUrl": "./",
    "outDir": "./dist/out-tsc",
    "forceConsistentCasingInFileNames": true,
  ...
```

NOTE: We set the 'baseUrl' property to "./".

18. Read this page to create a shared asset folder in your workspace: Angular Shared Asset Library
But, the asset folder, itself can be created in the workspace root, like this:



The assets folder will hold all shared assets, such as icons, images, and such.

19. It's a good idea to create the projects folder, as well. This is hold all of the applications and libraries of the monorepo.
Create it inside the workspace root, same as the assets folder, above.

Adding Projects

Here's now to create apps and libraries in the monorepo.

Create an App

You can generate applications in the monorepo with this:

> NOTE: Make sure the command window is at the workspace root, when executing this.

```
npx ng generate application admin-app --no-standalone
```

> NOTE: We are using 'npx' instead of 'ng'. This is required

> NOTE: If you are working in Angular 17 or later, and don’t want standalone components, add this: '`--no-standalone`'

The generate command will prompt for the normal application creation things.
Say NO to routing, and choose the SCSS style sheet type.

Common Config

When using a monorepo, there is a common angular.json file. It retains much of the normal content of a single project angular.json file, but instead, has an array for all project entries.

When using a monorepo, there will be cascaded tsconfig.json files. A common tsconfig.json for the workspace, that sets workspace-global things such as the dist folder path (outDir). And, a tsconfig.json in each project that “extends” the common tsconfig, adding paths and shortcuts that are unique to each project.

App Updates for MonoRepo Workspace

For each app created in a monorepo, there are several things that have to be done, so it plays nicely with other apps, across the shared workspace.

- We have to increase the app’s build budget values in the angular.json file.
- We need to promote the app’s tsconfig file to be a “tsconfig.json” file.
- We need to update the paths in the app’s tsconfig.json, because the root tsconfig doesn’t list them.
- We need to update the application’s assets property in the angular.json file.
- We need to copy over in app files and app config from another app project.
- Each app needs its own app library folder, which contains commonly named components that are used by the Common-Library.

Refer to this page for how to do these things: Current Angular App Development

Once your app is setup, you can run it with this: `npx ng serve -- open`

Create a Library

A Monorepo is an ideal pattern for library development, in that the workspace can contain a project for the library, another project for testing the library, and any scratch work projects, and usage demonstrations.

Here’s the corresponding article for standing up a library project: Angular Library Development

> Sidebar: See this Tutorial for Angular Library development: Step-by-step guide to creating your first library in Angular

Creating a library project in your monrepo is slightly different, with this command:

```
npx ng generate library some-new-library --no-standalone
```

> NOTE: Execute the above while at the workspace root.

Once created, the library can be built with: `npx ng build`

Additional Considerations

Here are some special things done to projects in a monorepo so that all apps share components and libraries: Current Angular App Development

See this page for additional workspace setup: Angular Workspace Additional Setup

Initial Checkout

NOTE: Each time you checkout this project from GH, you will need to run the following to set the Node.js version and download packages needed for build and testing:

```
nvm use 20.18.3
npm install
```

The above command swaps in the Node.js version, and reads the `package.json` and `package-lock.json` files and installs everything needed, including the locally-specified Angular CLI version.

> NOTE: If you follows the above setup instructions, these are also in the workspace README.md.

Developing in Multiple Angular Versions

When developing Angular applications and libraries on a host, you have to pay attention to what version of Angular you're targeting, to determine what version of Node.js, Typescript, and RxJS that your host needs installed.

Here's the source matrix: https://angular.dev/reference/versions

And, here's a list of versions that we use for development:

NOTE: The Angular version is in the header row.

Package	14.2.0	15.2.10	15.1.0	13.3.0	17.0.7	17.3.12	19.2
Angular	^14.2.0	^15.2.10	^15.1.0	~13.3.0	^17.0.7	^17.3.0	*19.2
@angular/cli	~14.2.4		~15.1.3	~13.3.3	~17.0.7	^17.0.7	
Angular/material	~7.0.0		^15.1.2		^17.0.4		
Node.js		^18.10.0			20.18.3	20.18.3	
Typescript	~4.7.2	~4.9.5	~4.9.4	~4.6.2	5.2		
rxjs	~7.5.0	~7.5.0	~7.8.0	~7.5.0	^7.5.0		
tslib	^2.3.0	~2.3.0	^2.3.0	^2.3.0	^2.6.2		
@clr/angular	^13.8.3			^13.1.0			
@cds/angular	^6.1.5			^5.7.0			
@clr/icons	^13.8.3			^13.0.2			

NVM - Node.js Version Manager

When maintaining Angular apps and other Javascript-based libraries, you will come across the need to have a specific version of Node.js running on your development host.

Node Version Manager, is a Windows based package, that will swap in desired versions of Node.js as needed.

> NOTE: If you are installing NVM and Node on a Jenkins server, you will need to run all of the following under the jenkins user context.
>
> See this page for how to impersonate the Jenkins user: Linux: Impersonating Users
>
> As a quick answer, run this to become the Jenkins user, for the purpose of the steps in this article, you can use:
> `sudo su jenkins`
> Or:
> `sudo -u jenkins bash`
> And, be sure to exit from the jenkins user context, when finished.

Installing NVM

For Windows

Download and install the latest version of nvm for Windows.

For Ubuntu

From a terminal, navigate to your home folder with: `cd ~/`

Run the following:

```
curl -fsSL https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.5/install.sh | bash
```

If the above gives you any warning that it had trouble editing your shell profile file, try running it as sudo.

>

> NOTE: If you are installing NVM for a system account, such as Jenkins, follow the steps in the Jenkins section, first.

If the above, failed to edit your profile file, you can always edit your profile file manually, like this:

Your shell profile file will be one of the following: `~/.bashrc`, `~/.profile`, or `~/.bash_profile`

Navigate to your HOME folder, and open your profile file, and make sure the following is appended to its bottom:

```
export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"  # This loads nvm
[ -s "$NVM_DIR/bash_completion" ] && \. "$NVM_DIR/bash_completion"  # This loads nvm bash_completion
```

Once appended, save and close.

If you have an open shell session as the system account user, execute this to apply the changes you just made:

```
source ~/.bashrc
```

Jenkins-Specific Problems

For a Jenkins install on an Ubuntu build server, the above curl command gives this warning:

```
jenkins@blissbuildvm:~$ curl -fsSL https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.5/install.sh | bash
=> Downloading nvm from git to '/var/lib/jenkins/.nvm'
=> Cloning into '/var/lib/jenkins/.nvm'...
remote: Enumerating objects: 381, done.
remote: Counting objects: 100% (381/381), done.
remote: Compressing objects: 100% (324/324), done.
remote: Total 381 (delta 43), reused 175 (delta 29), pack-reused 0 (from 0)
Receiving objects: 100% (381/381), 383.82 KiB | 4.13 MiB/s, done.
Resolving deltas: 100% (43/43), done.
* (HEAD detached at FETCH_HEAD)
  master
=> Compressing and cleaning up git repository

=> Profile not found. Tried ~/.bashrc, ~/.bash_profile, ~/.zprofile, ~/.zshrc, and ~/.profile.
=> Create one of them and run this script again
   OR
=> Append the following lines to the correct file yourself:

export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"  # This loads nvm

=> Close and reopen your terminal to start using nvm or run the following to use it now:

export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"  # This loads nvm
jenkins@blissbuildvm:~$ ls -lha
```

This is because the Jenkins user is a system account, and the NVM installer cannot determine a profile file to update.

Follow the steps on this page, to add a shell and profile file for the Jenkins user: Ubuntu: Converting a System Account to Interactive

Once complete, you can rerun the NVM installer, and it should complete without issue.

> NOTE: If you are installing NVM on a Jenkins build server, the nvm.sh script that gets installed does not have execute permission by default. This will cause an error during a Jenkins build job, when using NVM to change or install a version of Node.js.
> To fix this, allow execute permission for the script with this:
> `sudo chmod 774 ~/.nvm/nvm.sh`

Verify NVM Install

Run the following to confirm that NVM is installed.

> NOTE: NVM is installed on a per-user basis. So, you will need to be in a shell session of the specific user.

```
nvm --version
```

Once confirmed installed, you can install the desired Node.js versions that NVM will manage for you.

Usage

Installing Versions

Once installed, you need to tell NVM to install the desired Node.js versions that it will manage for you:

`nvm install 16`

`nvm install 20`

`nvm install 22`

Switching Versions

With the desired versions of Node.js installed, you can use these commands to switch between them:

To run Node.js v16 (for Angular v14):

`nvm use 16.20.2`

To run Node.js v20 (for Angular v17):

`nvm use 20.18.3`

To run Node.js v22 (for Angular v19):

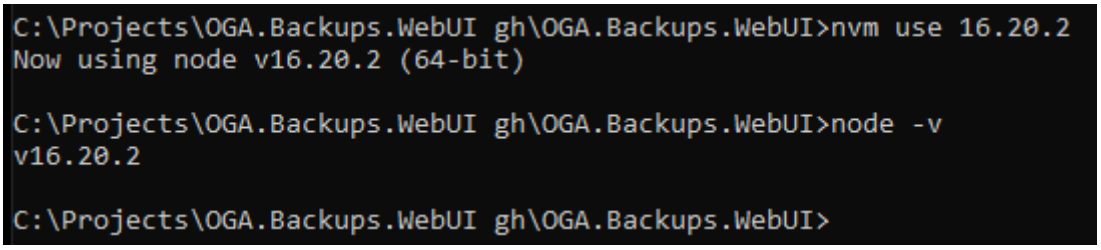
`nvm use 22.14.0`

> NOTE: Each time you tell NVM to switch versions, it may require answering a UAC popup.

Verifying Active Version

And once NVM has switched in the desired Node.js version, you can call this, to verify:

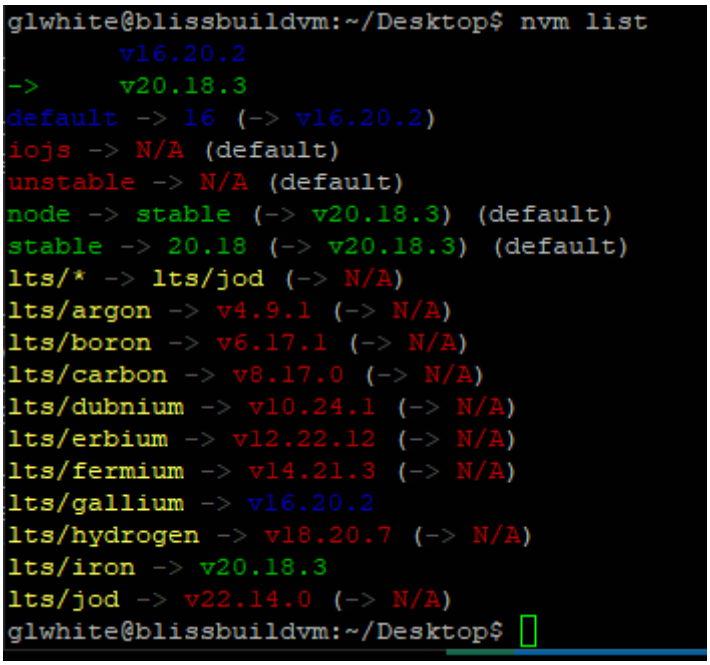
`node -v`



Listing Installed Versions

Use this command to see what versions of Node.js are installed and managed by NVM:

```
nvm list
```


glwhite@blissbuildvm:~/Desktop$ nvm list
v16.20.2
-> v20.18.3
default -> 16 (-> v16.20.2)
iojs -> N/A (default)
unstable -> N/A (default)
node -> stable (-> v20.18.3) (default)
stable -> 20.18 (-> v20.18.3) (default)
lts/* -> lts/jod (-> N/A)
lts/argon -> v4.9.1 (-> N/A)
lts/boron -> v6.17.1 (-> N/A)
lts/carbon -> v8.17.0 (-> N/A)
lts/dubnium -> v10.24.1 (-> N/A)
lts/erbium -> v12.22.12 (-> N/A)
lts/fermium -> v14.21.3 (-> N/A)
lts/gallium -> v16.20.2
lts/hydrogen -> v18.20.7 (-> N/A)
lts/iron -> v20.18.3
lts/jod -> v22.14.0 (-> N/A)
glwhite@blissbuildvm:~/Desktop$

Building for Multiple Environments

You will encounter the need to have different settings when building for dev, prod, test, etc.
This is easy to do, and only requires modifying your workspace's angular.json.

There's a couple of use cases for this:

- Using different environment.ts configuration when building for dev vs prod.
- When whitelabeling an app (same project for different customers).

This page is written as if you're wanting to swap in different environment.ts config files for dev and prod.
But, you can use the same technique to swap in other config, assets, and such, to build an app for different customers.

Environment.ts

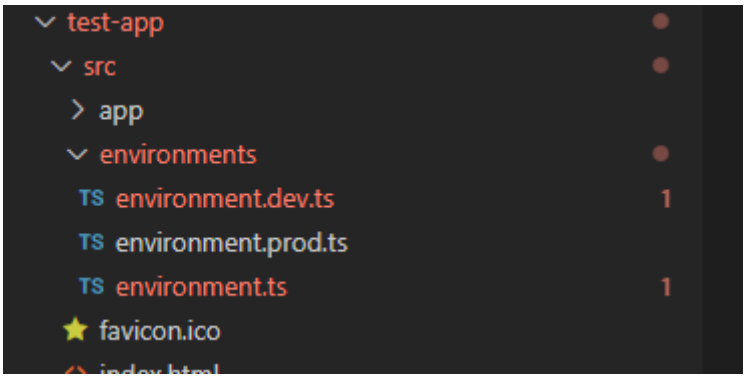
1. Make a folder where the environment.ts file lives in your app project.
This will let you keep the source tree organized.

2. Move the existing environment.ts into it.

3. Make a copy of the existing environment.ts and name it: environment.prod.ts

4. Rename environment.ts to environment.dev.ts.

Once done, your project folder will look like this:



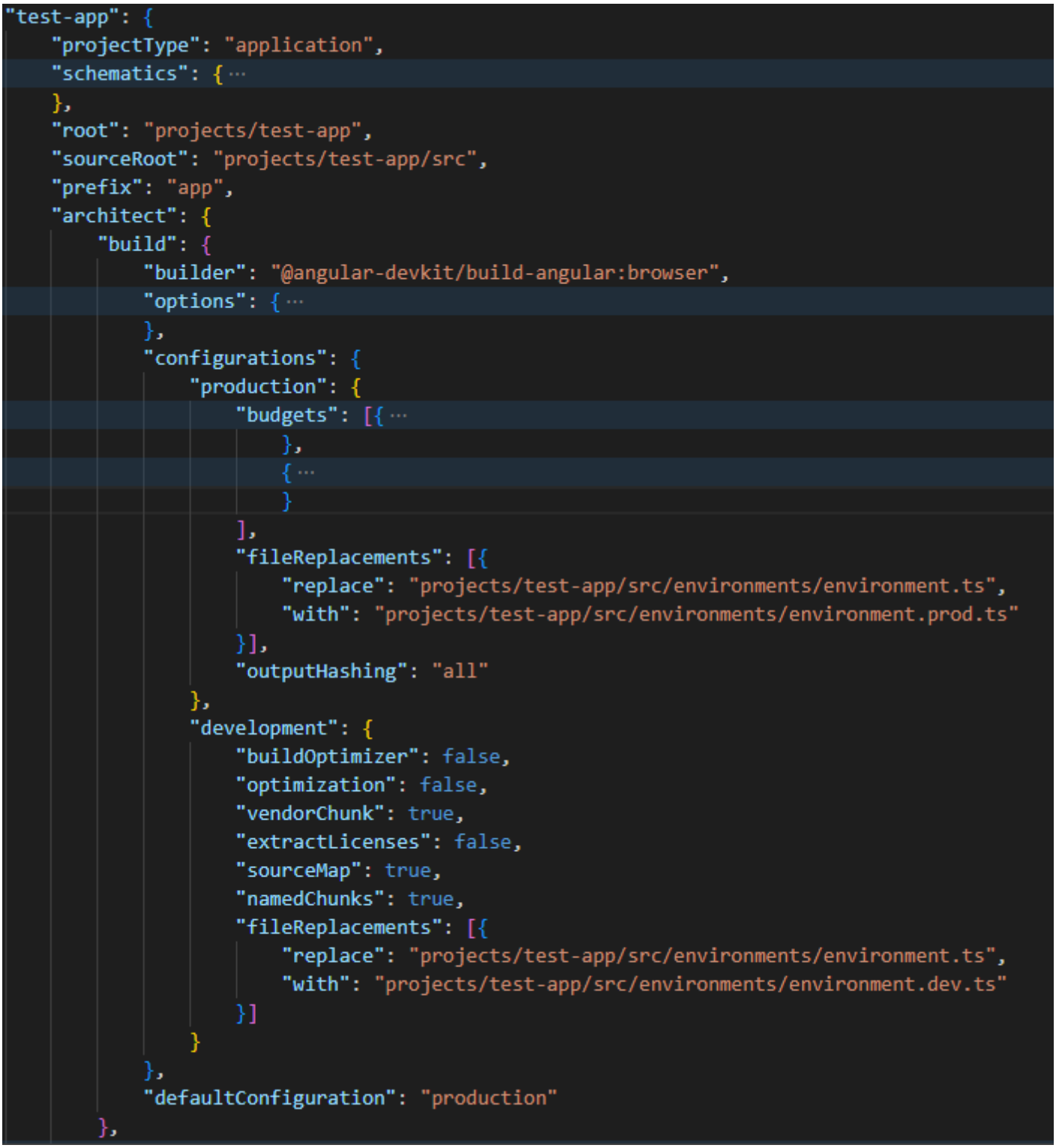
Angular.json

Now, open your angular.json file and locate the build configurations section for your app.
It will be in this path: projects -> appname -> architect -> build -> configurations

We will paste in this block into the development and production configuration nodes:

```
"fileReplacements": [{
    "replace": "projects/test-app/src/environments/environment.ts",
    "with": "projects/test-app/src/environments/environment.prod.ts"
}],
```

Making the configurations block look something like this:



Referenced Config

Do a quick search across your codebase for code that imports the environment class (from environment.ts), and update the path to include the environments folder that you created, earlier.

Below is an example, where it is imported in main.ts.

Found in main.ts

The below main.ts takes action on config from the environment.ts. We must update the import statement, to point to the path of the base environment class, in environments/environment:

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

It may also be imported in App.Module.ts.
Be sure to check, there.

Angular Material Dark Mode Theming

Here are the steps required to implement light/dark mode in Angular 17 with Angular Material.

Once done, your app will be able to toggle between light and dark modes.

Locked Theme

If you want your app to define a theme, and prevent user changes, set the theme name in the environment.ts to one of these:

- lock-light
- lock-dark

Styles.scss

Update the styles.scss file to add this at the top:

```
/* Start of light-dark mode styling */
@use 'assets/themes.scss' as themes;
/* End of light-dark modestyling */
```

Themes.scss

Create a themes.scss file in your app's assets folder.

Give it this content:

```
/* This file includes light/dark mode theming for Angular Material and non-Material components.
    It is referenced by an include in styles.scss.

    It includes some custom coloring for Material card and dialog components.
    And, it includes custom coloring for non-Material elements.
*/

@use '@angular/material' as mat;
```

```
@include mat.core();

// Define the default (light) theme
$light-theme: mat.define-light-theme((
  color: (
    primary: mat.define-palette(mat.$indigo-palette),
    accent: mat.define-palette(mat.$pink-palette),
  ),
  typography: mat.define-typography-config(),
  density: 0
));

// Define the dark theme
$dark-theme: mat.define-dark-theme((
  color:
  (
    primary: mat.define-palette(mat.$blue-gray-palette),
    accent: mat.define-palette(mat.$deep-orange-palette),
    background:
    (
      background: #29414e, // Custom dark background color
      card: #252525,       // Background for Material cards
      dialog: #2A2A2A,     // Background for dialogs/modals
      hover: #333333,      // Background when hovering Material elements
    )
  )
));

// Apply the light theme globally by default
@include mat.all-component-themes($light-theme);

// Manually set background colors
body {
  background-color: #ffffff; // Light theme background
  color: rgba(0, 0, 0, 0.87); // Light theme text color
  transition: background 0.3s ease-in-out, color 0.3s ease-in-out;
}

// Dark theme styles
.dark-theme {
```

```
  @include mat.all-component-colors($dark-theme);

  // #29414e
  background-color: #29414e; // Dark theme background
  color: rgba(255, 255, 255, 0.87); // Dark theme text color
}

/* The following will assign light/dark mode theme colors to non-Material elements */

// Style additional elements
.dark-theme h1,
.dark-theme h2,
.dark-theme h3,
.dark-theme p,
.dark-theme a {
  color: rgba(255, 255, 255, 0.87); // Adjust text color for dark mode
}

.dark-theme a:hover {
  color: #bb86fc; // Optional: Change hover color for dark mode
}

.dark-theme button {
  background-color: #333; // Dark theme for non-Material buttons
  color: white;
  border: 1px solid #555;
}

.dark-theme input {
  background-color: #222; // Dark theme for input fields
  color: white;
  border: 1px solid #555;
}
```

Color Theme Service

We need a service to centrally manage color theming.

Here's what it looks like:

```
/*
  This service provides a central point for color theme changes to the app.
  Callers can reference this service, and call setTheme() or toggleTheme().

*/

import { Injectable, Inject, OnDestroy } from '@angular/core';

import { BehaviorSubject } from 'rxjs';

import { IEnvironment, ENVIRONMENT_INJTOKEN } from 'lib-oga-webui-sharedkernel';


@Injectable({
  providedIn: 'root'
})
export class ColorThemingService implements OnDestroy
{
  //#region Private Fields

  static lastusedInstanceId: number = 0;
  private instanceId: number = 0;

  private isDarkThemeSubject = new BehaviorSubject<boolean>(false);
  isDarkTheme$ = this.isDarkThemeSubject.asObservable();

  public lockedtheme: boolean = false;

  //#endregion


  //#region ctor / dtor

  constructor(
            // Injecting app-level configuration for use...
            // See this:
https://oga.atlassian.net/wiki/spaces/~311198967/pages/131760134/Angular+Expose+Environment+Config+to
+DI
            @Inject(ENVIRONMENT_INJTOKEN) private env: IEnvironment)
  {
```

```
  ColorThemingService.lastusedInstanceId++;
  this.instanceId = ColorThemingService.lastusedInstanceId;
  console.log("ColorThemingService-" + this.instanceId + ":constructor - triggered.");

  // Automatically load the previous theme when the service is created...
  this._loadTheme();

  console.log("ColorThemingService-" + this.instanceId + ":constructor - ended.");
}

ngOnDestroy()
{
  console.log("ColorThemingService-" + this.instanceId + ":ngOnDestroy - triggered.");
}

//#endregion


//#region Private Methods

// Private method used to initialize the theme, on startup, from stored preferences or platform defaults.
private _loadTheme()
{
  console.log("ColorThemingService-" + this.instanceId + ":_loadTheme - triggered.");

  // Attempt to load theme from localStorage...
  const storedTheme = localStorage.getItem('theme');

  // Attempt to get the platform preference...
  const prefersDark = window.matchMedia('(prefers-color-scheme: dark)').matches;

  // And, get the default theme from the environment...
  const envtheme = this.env.default_ThemeName;

  // Check if the environment allows user theme choice...
  if(envtheme === 'light')
  {
    // The environment specifies a light theme.

    // Lock the theme...
```

```
    this.lockedtheme = true;

    // Set the light theme...
    this.setTheme(false);
  }
  else if(envtheme === 'dark')
  {
    // The environment specifies a dark theme.

    // Set the dark theme...
    this.setTheme(true);

    // Lock the theme...
    this.lockedtheme = true;
  }
  else
  {
    // Environment doesn't specify a known theme name.
    // So, the user is allowed to choose their theme.

    // Set the theme based on saved preference, system setting, or environment...
    const isDark = storedTheme ? storedTheme === 'dark' : prefersDark;
    this.setTheme(isDark);

    // Unlock the theme...
    this.lockedtheme = false;
  }

  console.log("ColorThemingService-" + this.instanceId + ":_loadTheme - ended.");
}

//#endregion



//#region Public Methods

// Public method to initialize color theme service.
// This method created to preload the service and baseline color theme.
Init()
{
```

```
  console.log("ColorThemingService-" + this.instanceId + ":Init - triggered.");

  // We don't need to actually call the _loadTheme() method, as the constructor already did.
}

// Public method to toggle the current theme light->dark or dark->light.
toggleTheme()
{
  console.log("ColorThemingService-" + this.instanceId + ":toggleTheme - triggered.");

  // See if the theme is locked...
  if(this.lockedtheme === true)
  {
    // Theme is locked by environment.
    // Don't allow user to change it.

    console.log("ColorThemingService-" + this.instanceId + ":toggleTheme - theme locked. Cannot change.");

    return;
  }

  this.setTheme(!this.isDarkThemeSubject.value);
}

// Public method call to set the desired theme.
setTheme(isDark: boolean)
{
  console.log("ColorThemingService-" + this.instanceId + ":setTheme - triggered.");

  // See if the theme is locked...
  if(this.lockedtheme === true)
    {
      // Theme is locked by environment.
      // Don't allow user to change it.

      console.log("ColorThemingService-" + this.instanceId + ":setTheme - theme locked. Cannot change.");

      return;
    }
```

```
    this.isDarkThemeSubject.next(isDark);
    localStorage.setItem('theme', isDark ? 'dark' : 'light');

    if (isDark)
    {
      document.body.classList.add('dark-theme');

      console.log("ColorThemingService-" + this.instanceId + ":setTheme - theme changed to dark.");
    }
    else
    {
      document.body.classList.remove('dark-theme');

      console.log("ColorThemingService-" + this.instanceId + ":setTheme - theme changed to light.");
    }
  }

  //#endregion
}
```

The above service is a production-ready implementation that provides public methods to set the theme or toggle it.
It also includes an Init() method that needs to be called on app startup.

See the next section for how to do this.

App.Module.ts Changes

Here's what app.module.ts will look like with the necessary pieces in it:

```
import { NgModule } from '@angular/core';

import { APP_INITIALIZER } from '@angular/core';

// Import app-level libraries...
import { LibOgaWebuiSharedkernelModule, Runtime_Sleep } from 'lib-oga-webui-sharedkernel';

import { AppComponent } from './app.component';

import { ColorThemingService } from './color-theming/services/color-theming.service';
```

```
// Import environment variable elements...
import { ENVIRONMENT_INJTOKEN, IEnvironment } from 'lib-oga-webui-sharedkernel';
import { environment } from '../environments/environment';
import { provideAnimationsAsync } from '@angular/platform-browser/animations/async';



@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    AppRoutingModule,
    LibOgaWebuiSharedkernelModule,
  ],
  providers:
  [
    // Here, we register a provider that makes our app-level environment config available for consumption.
    // See this:
https://oga.atlassian.net/wiki/spaces/~311198967/pages/131760134/Angular+Expose+Environment+Config+to
+DI
    // Register our injection token for the environment class data of our app, so libraries can use it through DI...
    { provide: ENVIRONMENT_INJTOKEN, useValue: environment },
    {
      provide: APP_INITIALIZER,
      useFactory: initializeApp,
      multi: true,
      deps: [ColorThemingService, ENVIRONMENT_INJTOKEN]
    },
    provideAnimationsAsync(),
  ],
  bootstrap: [AppComponent]
})
export class AppModule {

  //#region Private Fields

  static lastusedInstanceId: number = 0;
  private instanceId: number = 0;

  //#endregion
```

```
  //#region ctor / dtor

  constructor()
  {
    AppModule.lastusedInstanceId++;
    this.instanceId = AppModule.lastusedInstanceId;
    console.log("AppModule-" + this.instanceId + ":constructor - triggered.");
  }

  //#endregion
}



// This method is called when the app first starts up.
// It will load any initial config, and setup anything required.
export function initializeApp(_ctsvc:ColorThemingService,
                              env: IEnvironment): () => Promise<boolean>
{
  console.log("AppModule-" + "_" + ":initializeApp - triggered.");

  // To ensure that dependencies are stood up and ready, we must wrap our logic in a lambda, instead of naked
execution...
  let lambda = async (): Promise<boolean> => {

    console.log("AppModule-" + "_" + ":initializeApp - inside startup promise...");

    // Pause a bit...
    {
      console.log("AppModule-" + "_" + ":initializeApp - Sleeping...");

      await Runtime_Sleep(1000);

      console.log("AppModule-" + "_" + ":initializeApp - Sleep done.");
    }

    // Preload the color theme service...
    {
      console.log("AppModule-" + "_" + ":initializeApp - Standup the color theme service...");
```

```
    // Verify the color theme service exists...
    if(_ctsvc == null)
    {
      console.error("AppModule-" + "_" + ":initializeApp - color theme service instance is null.");

      // Cannot load initial color theme.
      return false;
    }

    let val2 = await _ctsvc.Init();

    console.log("AppModule-" + "_" + ":initializeApp - Color Theme service preloaded.");
  }

  // Register any singletons...


  // Stand up any local config...


  // Report success...
  console.log("AppModule-" + "_" + ":initializeApp - returned.");
  return true;
 };

 return lambda;
}
```

The initializeApp() method is what initializes up the color theme service.

This method is called by an APP_INITIALIZER provider.
You can follow the above provider declaration for how to include it.

User Theme Changes

To allow a user to change light/dark mode, requires the following.

1. Import the color theme service to the component.
2. Add the color theme service to the component's constructor.
3. Call the toggleTheme() method of the service.

Here's what a minimal component would look like:

```
import { Component } from '@angular/core';

import { ColorThemingService } from '../../color-theming/services/color-theming.service';

@Component({
  selector: 'app-layout',
  templateUrl: './app-layout.component.html',
  styleUrl: './app-layout.component.scss'
})
export class AppLayoutComponent {

  constructor(private ctsvc: ColorThemingService)
  {
  }

  toggleTheme()
  {
    this.ctsvc.toggleTheme();
  }
}
```

The above component includes a toggle method that a button can call.
When pressed, it will tell the color theme service to swap modes.

CSS Reset

Some browsers impose a margin and padding to elements, that will create inconsistent rendering.

To prevent this, you can perform a CSS reset, by adding the following to your css:

```
/* Reset */
* {
    box-sizing: border-box;
    margin: 0;
    padding: 0;
}
```

But, the above will affect spacing in some libraries, like Angular Material.

So, a better approach, when using Angular Material is to use this reset:

```
/* Start of CSS Reset
    Perform a CSS Reset that doesn't affect Angular Material
    The following two blocks margin and spacing for naked elements.
    It will not affect Angular Material elements.
*/

// This Ensures any padding and borders don't affect an element's width/height.
// Specifically, it prevents layout issues when adding padding to elements.
* {
    box-sizing: border-box;
}

// Clear any spacing for naked elements...
// Specifically, this removes default spacing applied by browsers.
body, div, p, h1, h2, h3, h4, h5, h6 {
    margin: 0;
    padding: 0;
}
/* End of CSS Reset */
```

> NOTE: If you're working in Angular, place the above block near the top of the styles.scss file of your project.

The above will set border-box sizing, so any border and padding don't cause elements to be larger.

As well, it clears spacing for naked elements.

And, any spacing in Angular Material should be unaffected.

V19 Sharing DI Services

Here are some steps to include, when creating a public library that uses services from DI, such as HttpClient.

We cater the example on this page to using HttpClient.
But, they apply, equally, to any library that consumes from DI.

Library Service

Our example is a library service that uses HttpClient.
It will inject HttpClient from DI.

NOTE: Our service example will require that something registers HttpClient with DI, so that no error occurs at runtime.
So, we have a providers instance, below, that needs to be run by any consuming app.
Think of it as an initialization for the library.

Here's what the library service looks like:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class MyLibraryService {
  private http = inject(HttpClient);

  constructor() {}

  getData() {
    return this.http.get('https://api.example.com/data');
  }
}
```

Note that the above example injects the HttpClient, directly, into a private variable, instead of in the constructor.
This is a new injection method, allowed in V18.

Library Providers

Next, our library needs to include a providers instance that any consuming library or app can add to its providers array.

For this, we create a file, lib-providers.ts, in the src folder of our library, that looks like this:

```
import { ApplicationConfig } from '@angular/core';
import { provideHttpClient, withFetch } from '@angular/common/http';

export const LIB_OGAWEBUISHAREDKERNEL_PROVIDERS: ApplicationConfig =
{
  providers: [provideHttpClient(withFetch())]
};
```

Note: We've set the provider name to be unique to our library.
You, as well, will want to do this, to prevent colliding with the providers entry of another library.

Library Export

The public-api.ts file of our library needs to include an export statement for the providers instance, above:

```
export * from './lib-providers';
```

Consuming Library

Any library that consumes our library needs to add our provider to its providers array, like this lib-providers.ts:

```
import { ApplicationConfig } from '@angular/core';
import { LIB_OGAWEBUISHAREDKERNEL_PROVIDERS } from 'my-library';

export const CONSUMER_LIB_PROVIDERS: ApplicationConfig = {
  providers: [
    ...LIB_OGAWEBUISHAREDKERNEL_PROVIDERS.providers // □ Inherit HttpClient from `my-library`
  ]
};
```

The above will ensure that any consumer of our consuming library, will also include the providers from our library.

Consuming Service

Any service in our consuming library or app that consumes the service from our library, will inject it, like this:

```
import { Injectable, inject } from '@angular/core';
import { MyLibraryService } from 'my-library';

@Injectable({
  providedIn: 'root'
})
export class ConsumerLibraryService {
  private myLibraryService = inject(MyLibraryService);

  fetchData() {
    return this.myLibraryService.getData(); // □ Calls the service from `my-library`
  }
}
```

Consuming Library Public-API.ts

The public-api.ts file of the consuming library, will include a similar export of its providers instance, so that any registrations are performed by consumers of it.

That public-api.ts includes these elements:

```
export * from './consumer-service';
export * from './providers'; // □ Expose CONSUMER_LIB_PROVIDERS for easy consumption
```

Consuming Application

And, any application that consumes our library, or the intermediate consuming library, needs to import it and add its providers to the app's bootstrap statement, like this:

```
import { bootstrapApplication } from '@angular/platform-browser';
import { AppComponent } from './app/app.component';
```

```
import { CONSUMER_LIB_PROVIDERS } from 'consumer-library';

bootstrapApplication(AppComponent, CONSUMER_LIB_PROVIDERS);
```

> NOTE: If your consuming app does NOT register the DI provider of the library, you will see a runtime error, that the injector cannot resolve the HttpClient.

Consuming Component

This leaves our last example of a component of our app, that consumes the service from our library.

For a pre-V19 app, it looks like this:

```
import { Component, inject } from '@angular/core';
import { ConsumerLibraryService } from 'consumer-library';

@Component({
  selector: 'app-root',
  template: `<button (click)="loadData()">Fetch Data</button>`,
})
export class AppComponent {
  private consumerService = inject(ConsumerLibraryService);

  loadData() {
    this.consumerService.fetchData().subscribe(data => {
      console.log('Received Data:', data);
    });
  }
}
```

For a v19 and later app, the provider gets referenced in the app.config.ts file, like this:

```
import { ApplicationConfig, provideZoneChangeDetection } from '@angular/core';
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';

import { LIB_OGAWEBUISHAREDKERNEL_PROVIDERS } from '../../../lib-oga-webui-sharedkernel/src/lib-providers';
```

```
export const appConfig: ApplicationConfig = {
  providers:
  [
    provideZoneChangeDetection({ eventCoalescing: true }), provideRouter(routes),
    LIB_OGAWEBUISHAREDKERNEL_PROVIDERS.providers
  ]
};
```

HttpClientModule Deprecation

The HttpClientModule has been deprecated as of Angular 18.
So, here's what it looks like, to use Http in v18 and forward.

HttpClient Migration

First. The HttpClientModule is deprecated.
So, we will now be injecting HttpClient into components, via provider.

Add this line to components that will need Http:

```
import { HttpClient } from '@angular/common/http';
```

Instead of the class constructor injector, your component/service will simply use the above import, and have a private variable with the injected HttpClient, like the below service example:

```
import { Injectable, inject } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable({
  providedIn: 'root' // Or use `providedIn: MyLibraryModule` if needed
})
export class MyHttpService {
  private http = inject(HttpClient);

  getData() {
    return this.http.get('https://api.example.com/data');
  }
}
```

Angular: App Startup (pre-V19)

During application startup, you may have activities that need to occur before the first page opens. One example is a splash screen.

> NOTE: This page applies to Angular v18 and earlier, as APP_INITIALIZER is deprecated in V19.
>
> See this page for how to do the same in v19: Angular: App Startup (V19 and later)

To make one occur, do these things:

1. Add an import of `APP_INITIALIZER` to your app.module.ts, like this:

   ```
   // This import added to hook into the initialization so we can delay opening pages for the splash container to display.
   import { APP_INITIALIZER } from '@angular/core';
   ```

2. The App Initalizer can act as a DI provider that will call any method you like.
   We will add a startup method call to the providers block, like this:

   ```
   providers:
   [
     {
       provide: APP_INITIALIZER,
       useFactory: initializeApp,
       multi: true,
       deps: [AppconfigService]
     },
   ```

   The above provider calls a method, named, `initializeApp`. This method must return a Promise<boolean>.

   > NOTE: The above call has a DI dependency. So, we add a deps list that includes it.

3. Still inside app.module.ts, define the `initializeApp` method, like this:

```
// This method is called when the app first starts up.
// It will load any initial config, and setup anything required.
export function initializeApp(_appsvcconfig: AppconfigService): () => Promise<boolean>
{
  console.log("AppModule-" + "_" + ":initializeApp - triggered.");

  // To ensure that dependencies are stood up and ready, we must wrap our logic in a lambda, instead of naked execution...
  let lambda = async (): Promise<boolean> => {

    console.log("AppModule-" + "_" + ":initializeApp - inside startup promise...");

    if(_appsvcconfig == null)
    {
      console.error("AppModule-" + "_" + ":initializeApp - appconfig service instance is null.");

      // Cannot startup app.
      return false;
    }

    console.log("AppModule-" + "_" + ":initializeApp - Sleeping...");

    await Runtime_Sleep(1000);

    console.log("AppModule-" + "_" + ":initializeApp - Sleep done. Calling LoadConfig...");

    let val = await _appsvcconfig.LoadConfig();

    console.log("AppModule-" + "_" + ":initializeApp - LoadConfig returned.");

    // Register any singletons...

    // Stand up any local config...

    // Report success...
    return true;
  };

  return lambda;
```

```
}
```

The above method returns a Promise (to the provider factory) that is actually a lambda method with an async body.

4. Inside the initializer’s lambda, we call a sleep method to slow down presentation, so a splash page can linger.
5. We can add any startup activities to this method as needed for initialization.

Angular: How to Return a Promise

You will come across the need to return a promise from a function.

Here’s an example of a promise that resolves itself immediately:

```
private getStudyPeriods(): Promise<CurrentPeriod> {
    let data = [];

    return new Promise(resolve => {
        resolve(data);
    });
}
```

Angular: App Startup (V19 and later)

During application startup, you may have activities that need to occur before the first page opens. One example is a splash screen.

> NOTE: This page applies to Angular v19 and later, as APP_INITIALIZER was deprecated in V19.
>
> See this page for how to do the same, before v19: https://wiki.galaxydump.com/link/130

To make one occur, do these things:

1. Open the app.config.ts of your app.
2. Add an import for provideAppInitializer:

```
import { provideAppInitializer } from '@angular/core';
```

3. Create an initializeApp method in the file (or, somewhere else that you can import).
   It must return a Promise<boolean>. So, make sure its signature looks like this:

```
export function initializeApp(): Promise<boolean>
```

> NOTE: The old way of doing this, used an initializer method that returned a () => Promise<boolean>
> So, be sure to update the return when migrating older code.

4. Add a provider entry for the initializer:
   It looks like this:

```
export const appConfig: ApplicationConfig = {
  providers:
  [
    provideAppInitializer(initializeApp)
  ]
};
```

5. If your initializeApp function needs to run some async logic, structure it like this:

```
export function initializeApp(): Promise<boolean>
{
  const _appsvcconfig = inject(AppconfigService);

  console.log("AppModule-" + "_" + ":initializeApp - triggered.");

  // To ensure that dependencies are stood up and ready, we must wrap our logic in a lambda, instead
of naked execution...
  return (async (): Promise<boolean> =>
  {
    console.log("AppModule-" + "_" + ":initializeApp - inside startup promise...");

    ... DO SOME THINGS...

    // Report success or failure...
    return true;
  })(); // Notice the () at the end - this immediately invokes the async lambda
}
```

Angular Workspace Additional Setup

Here's some additional things that must be done to tidy up an Angular monorepo workspace.

Environment.ts Files

See this page for how to setup multiple environment config files for dev and prod: Building for Multiple Environments

Karma test.ts

Create a test.ts file in the /src folder of each project in the monorepo workspace.

Give it this content:

```
// This file is required by karma.conf.js and loads recursively all the .spec and framework files

import 'zone.js';
import 'zone.js/testing';
import { getTestBed } from '@angular/core/testing';
import {
  BrowserDynamicTestingModule,
  platformBrowserDynamicTesting
} from '@angular/platform-browser-dynamic/testing';

// First, initialize the Angular testing environment.
getTestBed().initTestEnvironment(
  BrowserDynamicTestingModule,
  platformBrowserDynamicTesting(),
);
```

Open the workspace angular.ts file, and edit the test/options block to add a 'main' key that looks like this:

> NOTE: The full path in angular.ts is: <project-name>/architect/test/options

```
"test": {
  "builder": "@angular-devkit/build-angular:karma",
  "options": {
        "main": "projects/lib-oga-webui-runtimesecurity/src/test.ts",
    "tsConfig": "projects/lib-oga-webui-runtimesecurity/tsconfig.spec.json",
    "karmaConfig": "projects/lib-oga-webui-runtimesecurity/karma.conf.js",
    "polyfills": [
      "zone.js",
      "zone.js/testing"
    ]
  }
}
```

The main key needs to point to the test.ts file that you created, above.

Repeat the above for each project in the workspace.

Test.ts Usage

Test.ts Include

For tests to be discovered, you need to make sure that each tsconfig.spec.ts file (one in each project) includes a files reference to the test.ts file.

Here's an example of what a tsconfig.spec.json file looks like with the files block added, and having a reference to the test.ts file:

```
/* To learn more about Typescript configuration file: https://www.typescriptlang.org/docs/handbook/tsconfig-json.html. */
/* To learn more about Angular compiler options: https://angular.dev/reference/configs/angular-compiler-options. */
{
  "extends": "../../tsconfig.json",
  "compilerOptions": {
    "outDir": "../../out-tsc/spec",
```

```
    "types": [
      "jasmine"
    ]
  },
  "files": [
    "src/test.ts"
  ],
  "include": [
    "**/*.spec.ts",
    "**/*.d.ts"
  ]
}
```

Test.ts Exclude

Since libraries get consumed by other developers, we need to make sure that the test.ts file doesn't get included in the packaged library.

To prevent this, we need to add an exclude to the tsconfig.lib.ts of each library project.

Open each tsconfig.lib.ts file and add an exclude for the test.ts file, like the following:

```
/* To learn more about this file see: https://angular.io/config/tsconfig. */
{
  "extends": "../../tsconfig.json",
  "compilerOptions": {
    "outDir": "../../out-tsc/lib",
    "declaration": true,
    "declarationMap": true,
    "inlineSources": true,
    "types": []
  },
  "exclude": [
    "src/test.ts",
    "**/*.spec.ts"
  ]
}
```

NOTE: The above exclude block points to the src/test.ts in that project.

Karma.Conf.ts File

Create a karma.conf.ts file in the base of each project of the monorepo workspace.
This will be just inside the project folder, adjacent to the src folder.

Open the file, and add the content, below.

> NOTE: each karma.conf.ts has a unique path in the coverageReporter block.

Here's what the content should look like:

```
// Karma configuration file, see link for more information
// https://karma-runner.github.io/1.0/config/configuration-file.html

module.exports = function (config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine', '@angular-devkit/build-angular'],
    plugins: [
      require('karma-jasmine'),
      require('karma-chrome-launcher'),
      require('karma-jasmine-html-reporter'),
      require('karma-coverage'),
      require('@angular-devkit/build-angular/plugins/karma')
    ],
    client: {
      jasmine: {
        // you can add configuration options for Jasmine here
        // the possible options are listed at https://jasmine.github.io/api/edge/Configuration.html
        // for example, you can disable the random execution with `random: false`
        // or set a specific seed with `seed: 4321`
      },
      clearContext: false // leave Jasmine Spec Runner output visible in browser
    },
    jasmineHtmlReporter: {
      suppressAll: true // removes the duplicated traces
    },
    coverageReporter: {
      dir: require('path').join(__dirname, '../../coverage/middle2-lib'),
      subdir: '.',
      reporters: [
        { type: 'html' },
```

```
      { type: 'text-summary' }
    ]
  },
  reporters: ['progress', 'kjhtml'],
  port: 9876,
  colors: true,
  logLevel: config.LOG_INFO,
  autoWatch: true,
  browsers: ['Chrome'],
  singleRun: false,
  restartOnFileChange: true
 });
};
```

> NOTE: Be sure to edit the path in coverageReporter/dir, to match the name of the containing project.

Open the angular.ts file, and add a karmaConfig key in each project.

The karmaConfig key will be put in the same test options block that you put the test.ts entry in the previous section.

Here's an example of what that options block will look like:

```
"test": {
 "builder": "@angular-devkit/build-angular:karma",
 "options": {
       "main": "projects/lib-oga-webui-runtimesecurity/src/test.ts",
  "tsConfig": "projects/lib-oga-webui-runtimesecurity/tsconfig.spec.json",
  "karmaConfig": "projects/lib-oga-webui-runtimesecurity/karma.conf.js",
  "polyfills": [
    "zone.js",
    "zone.js/testing"
  ]
 }
}
```

The karmaConfig key needs to point to the karma.conf.js file that you created, above.

Repeat the above for each project in the workspace.

Common Imports

Here's a list of common imports, used in Angular components.

NOTE: Usage requires an import statement and adding the module to the Imports block of your component.

Common Module

Provides core directives like `*ngIf`, `*ngFor`, `ngClass`, `ngStyle`, etc.

Imported as:

`import { CommonModule } from '@angular/common';`

Forms Module

If your component uses template-driven forms (`ngModel`), import `FormsModule`.

Imported with this:

`import { FormsModule } from '@angular/forms';`

Reactive Forms Module

If using reactive forms (`FormGroup`, `FormControl`), import `ReactiveFormsModule`

Imported with this:

`import { ReactiveFormsModule } from '@angular/forms';`

Angular Material

There are several imports for this, depending on what elements you use.

```
import {MatCardModule} from '@angular/material/card';
import {MatDividerModule} from '@angular/material/divider';
import {MatListModule} from '@angular/material/list';
import {MatIconModule} from '@angular/material/icon';
```

CDK Drag and Drop

Imported with this:

```
import {DragDropModule} from '@angular/cdk/drag-drop';
```

Angular Provider Usage

When developing libraries, you will come across the need for your library to somehow, convey or pass on, any registration needs it has to the consuming application, or consuming library.

Simple Example

The typical way to do this, is to include a lib-providers file with a function that returns a providers array.

Here's a simple example of a lib-providers.ts:

```
import { ApplicationConfig } from '@angular/core';
import { provideHttpClient, withFetch } from '@angular/common/http';

export const LIB_OGAWEBUISHAREDKERNEL_PROVIDERS: ApplicationConfig =
{
  providers: [provideHttpClient(withFetch())]
};
```

The above is a simple lib-providers.ts file that a library would contain.
It includes a provider call to register the HttpClient, as the library uses it.

And, the consuming application would simply add that provider call to its providers block, like this:

```
import { LIB_OGAWEBUISHAREDKERNEL_PROVIDERS } from 'lib-oga-webui-sharedkernel';

export const appConfig: ApplicationConfig = {
  providers:
  [
    provideZoneChangeDetection({ eventCoalescing: true }), provideRouter(routes),
    ...LIB_OGAWEBUISHAREDKERNEL_PROVIDERS.providers
  ]
};
```

The above example is how an app would include a library's needed providers, for DI registration.

>

NOTE: The usage of the spread operator (...) on the providers line.
This flattens the received array to one-dimension.

VSCode CORS

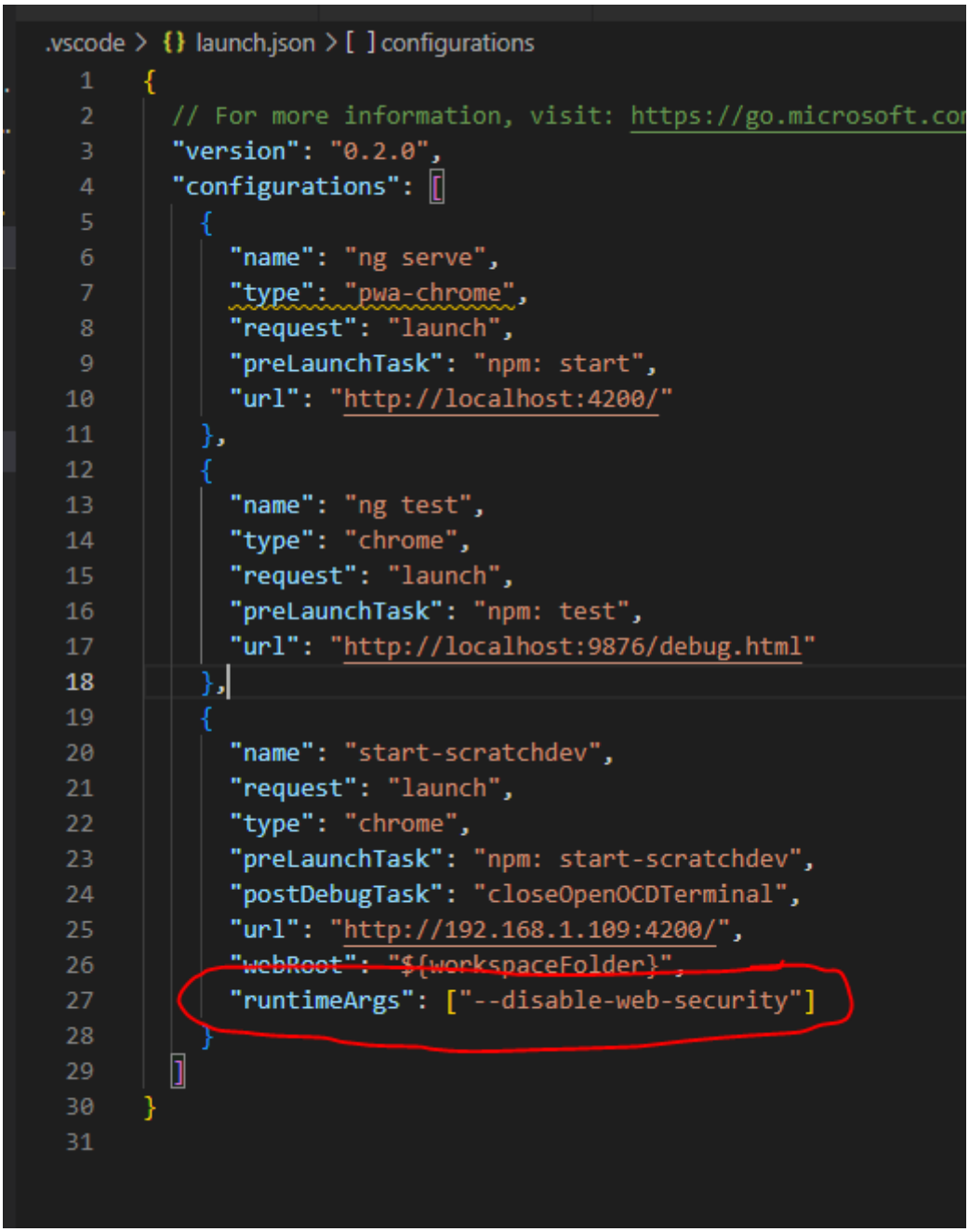
When debugging webUI projects in VSCode, it can be necessary to disable any CORS checking, so that VSCode can serve your app, while your backend API calls point elswhere.

Aside: For more information on how to debug Angular applications in VSCode, see this: Debugging Angular in VSCode

Any easy way to accomplish this, is to disable web security of the node runtime that is hosting the app (for VSCode).

To do this, add a --disable-web-security switch to the runtime arguments of your debug startup.

Below, is a launch.json file of an Angular project in VSCode. It contains a debug startup called, `start-scratchdev.` We add this line to it: `"runtimeArgs": ["--disable-web-security"]`



The above runtime argument addition allows the node hosting process of our debugger session, to let the app make API calls to a different origin.

Angular: Add Query Parameters without Router

Sometimes it is necessary to add query parameters to a REST API url without going through the Angular router logic, such as when doing paginated lists on large tables.

To make this easy to do, just use the HttpParams module.

Import it with:

```
import { HttpParams } from '@angular/common/http';
```

And, add the query parameters block to your http client call, like this:

```
return this.http.get('/api/v1/products',
{
        params: new HttpParams()
        .set('pageNumber', pageNumber.toString())
        .set('pageSize', pageSize.toString())
        }).pipe(products =>  console.log(products));
}
```

> NOTE: Lines 2 through 6 (stopping after the curly) are the query parameter block.

Here’s a source reference:

https://fireflysemantics.medium.com/passing-url-parameters-with-the-angular-http-client-821d1a097f96

Angular Dynamic Routing

Here’s a technique for implementing dynamic routing, where we want a common page or component to serve multiple route paths.

If you’re interested in passing query parameters through Angular routes, see this: Angular: Add Query Parameters without Router

Implementation

The technique, below, is from here: Angular Basics: Dynamic Activated Route Snapshots

We will use an Activated Route, in the target component, so it can recover path fragments from its route.

First. We will create a route to our component in the routing class. But, we will include a variable in the route string.

Here’s what ours will look like:

```
const routes: Routes = [
  {
    path: 'home',
    component: HomePageComponent,
  },
  // Below, is our dynamic route, with a dynamic suffix of 'dbname'.
  // Any calls to 'liveview/***', will open the LiveViewComponent.
  {
    path: 'liveview/:dbname',
    component: LiveViewComponent,
  },
  // redirect to `home` if there is no path
  {
    path: '',
    redirectTo: 'home',
    pathMatch: 'full',
  },
```

```
];
```

In the above route list, the `LiveViewComponent` will be called for any path that begins with ‘liveview’.

Second. We will update `LiveViewComponent`, to retrieve the ‘dbname’ portion of the route, so it can display the appropriate data.

To do this, we add ActivatedRoute to the constructor of LiveViewComponent, like this:

```
export class DashbLayout4Component {
  constructor(private route: ActivatedRoute)
  {
    ...
  }
}
```

With the ActivatedRoute in the constructor, our component can now retrieve the ‘dbname’ fragment from the url route, like this:

```
let dbname = this.route.snapshot.params['dbname'] ?? "";
```

If we add the above line to the ngOnInit or ngAfterContentInit, it will be able to load any special content, based on the ‘dbname’ value.

Angular: Get Button Name

When building navigation, or handling button clicks, it’s sometimes necessary to retrieve the name of a button (or other control) from the click event. This is done by drilling into the MouseEvent that is passed into the handler.

Basically, we get the target property of the MouseEvent, cast it as an HTMLElement, and get the innerHTML string, which should be the button name.

For example, the following menu selections use the same click handler method:

```
<clr-vertical-nav [clrVerticalNavCollapsible]="true">
    <a clrVerticalNavLink (click)="onSideMenuClick($event)" routerLinkActive="active">
        <cds-icon clrVerticalNavIcon shape="bolt"></cds-icon>
        RFI
    </a>
    <a clrVerticalNavLink (click)="onSideMenuClick($event)" routerLinkActive="active">
        <cds-icon clrVerticalNavIcon shape="sad-face"></cds-icon>
        Action Items
    </a>
    <a clrVerticalNavLink (click)="onSideMenuClick($event)" routerLinkActive="active">
        <cds-icon clrVerticalNavIcon shape="bug"></cds-icon>
        Issues
    </a>
</clr-vertical-nav>
```

The above three menu selections all have a click event tied to the same handler method (`onSideMenuClick`).
This method call, determines which item was clicked, by interrogating the click event. See below:

```
onSideMenuClick(event: MouseEvent) {
  // Get the name of the button that was clicked...
  var sss = (event.target as HTMLElement).innerHTML;

  // Open the desired page...
  switch (sss.trim()) {
    case "Action Items":
      this.openActionItemsPage();
      break;
```

```
      case "Return to Projects":
        this.NavigateToProjectsListing();
        break;
      case "RFI":
        this.openRFIPage();
        break;
        default:
        console.error("ProjectFrameComponent-" + this.instanceId + ":onMenuDropdownClick - encountered
unknown dropdown button: " + sss);
        break;
    }
  }
```

The above click handler, accepts a MouseEvent, gets the target from it, casts it to an HTMLElement, and gets the innerHTML text of the button.

Then, it decides which action based on the text of the menu button.

This same technique can be used for dynamically generated menu items, where the menu name is not hardcoded, but is set by variable.

Angular Context Menu

Here’s some notes on context menus in Angular.

This article was used as basis: Context Menus Made Easy with Angular CDK

The above article reference included a working StackBlitz, downloaded, here: angular-yd6ay3.zip

It uses the overlay functionality from Angular CDK, to present the menu, and access it.

This has been incorporated into the dashboard project, as a context menu for adding divs at runtime.

Typescript This

Taken from an older version of the Typescript handbook, as the latest does not include this, but is still relevant.

This.

Learning how to use this in JavaScript is something of a rite of passage. Since TypeScript is a superset of JavaScript, TypeScript developers also need to learn how to use this and how to spot when it's not being used correctly. Fortunately, TypeScript lets you catch incorrect uses of this with a couple of techniques. If you need to learn how this works in JavaScript, though, first read Yehuda Katz's Understanding JavaScript Function Invocation and "this". Yehuda's article explains the inner workings of this very well, so we'll just cover the basics here.

this and arrow functions

In JavaScript, this is a variable that's set when a function is called. This makes it a very powerful and flexible feature, but it comes at the cost of always having to know about the context that a function is executing in. This is notoriously confusing, especially when returning a function or passing a function as an argument.

Let's look at an example:

```
let deck = {
  suits: ["hearts", "spades", "clubs", "diamonds"],
  cards: Array(52),
  createCardPicker: function () {
    return function () {
      let pickedCard = Math.floor(Math.random() * 52);
      let pickedSuit = Math.floor(pickedCard / 13);

      return { suit: this.suits[pickedSuit], card: pickedCard % 13 };
    };
  },
};

let cardPicker = deck.createCardPicker();
```

```
let pickedCard = cardPicker();


alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

Notice that createCardPicker is a function that itself returns a function. If we tried to run the example, we would get an error instead of the expected alert box. This is because the this being used in the function created by createCardPicker will be set to window instead of our deck object. That’s because we call cardPicker() on its own. A top-level non-method syntax call like this will use window for this. (Note: under strict mode, this will be undefined rather than window).

We can fix this by making sure the function is bound to the correct this before we return the function to be used later. This way, regardless of how it’s later used, it will still be able to see the original deck object. To do this, we change the function expression to use the ECMAScript 6 arrow syntax. Arrow functions capture the this where the function is created rather than where it is invoked:

```
let deck = {
  suits: ["hearts", "spades", "clubs", "diamonds"],
  cards: Array(52),
  createCardPicker: function () {
    // NOTE: the line below is now an arrow function, allowing us to capture 'this' right here
    return () => {
      let pickedCard = Math.floor(Math.random() * 52);
      let pickedSuit = Math.floor(pickedCard / 13);


      return { suit: this.suits[pickedSuit], card: pickedCard % 13 };
    };
  },
};

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();


alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

Notice. In the above example, the function defined inside `createCardPicker` was changed to a lambda, giving it this syntax: `return () => {}`

This subtle change, tells the compiler to inject the ‘this’ context into the lambda.

Takeaway

So. Whenever you come across a spot in your code, where you have a callback method that uses ‘this’, but the this is not the same as the object where the callback resides, then the this context has inadvertently been swapped out. You can fix the problem with the same technique as above.

In addition, TypeScript will warn you when you make this mistake if you pass the noImplicitThis flag to the compiler. It will point out that this in this.suits[pickedSuit] is of type any.

HTML CSS Prevent Text Selection

Here’s a simple css style you can add to your styles.scss of any project, to prevent user selection of text.

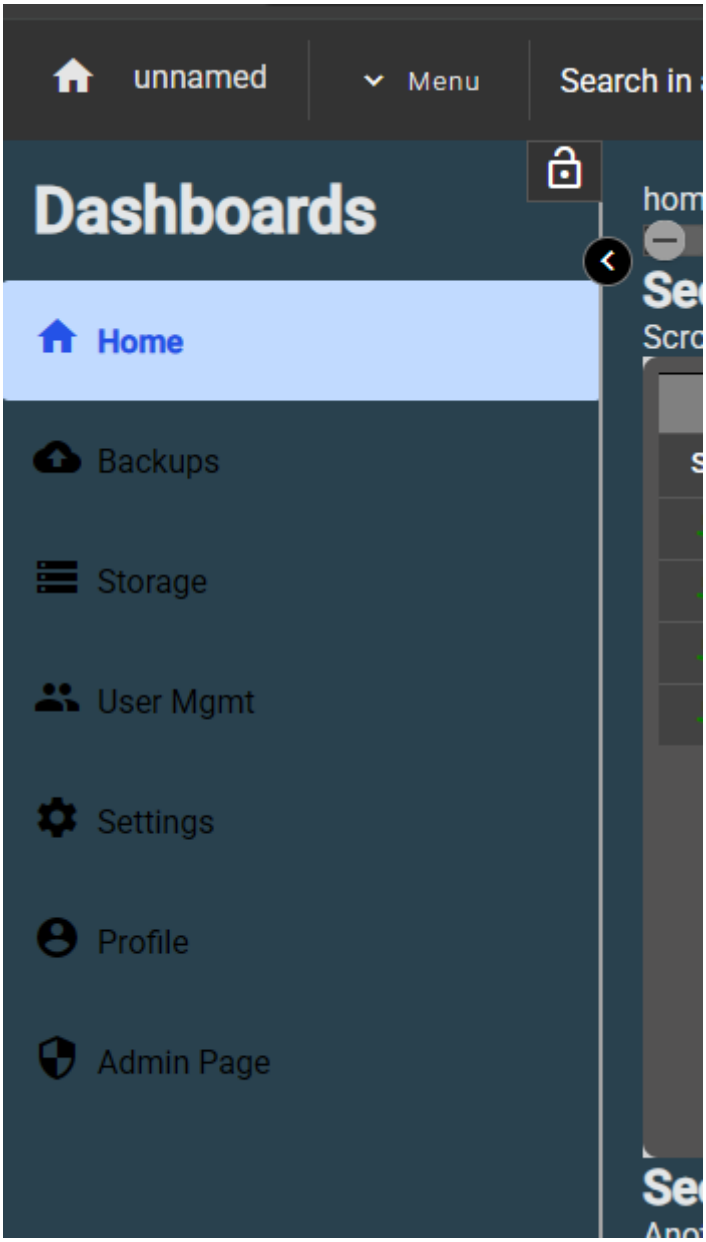
```
.prevent-select {
  -webkit-user-select: none; /* Safari */
  -ms-user-select: none; /* IE 10 and IE 11 */
  user-select: none; /* Standard syntax */
}
```

Swapping Side-Nav Groups

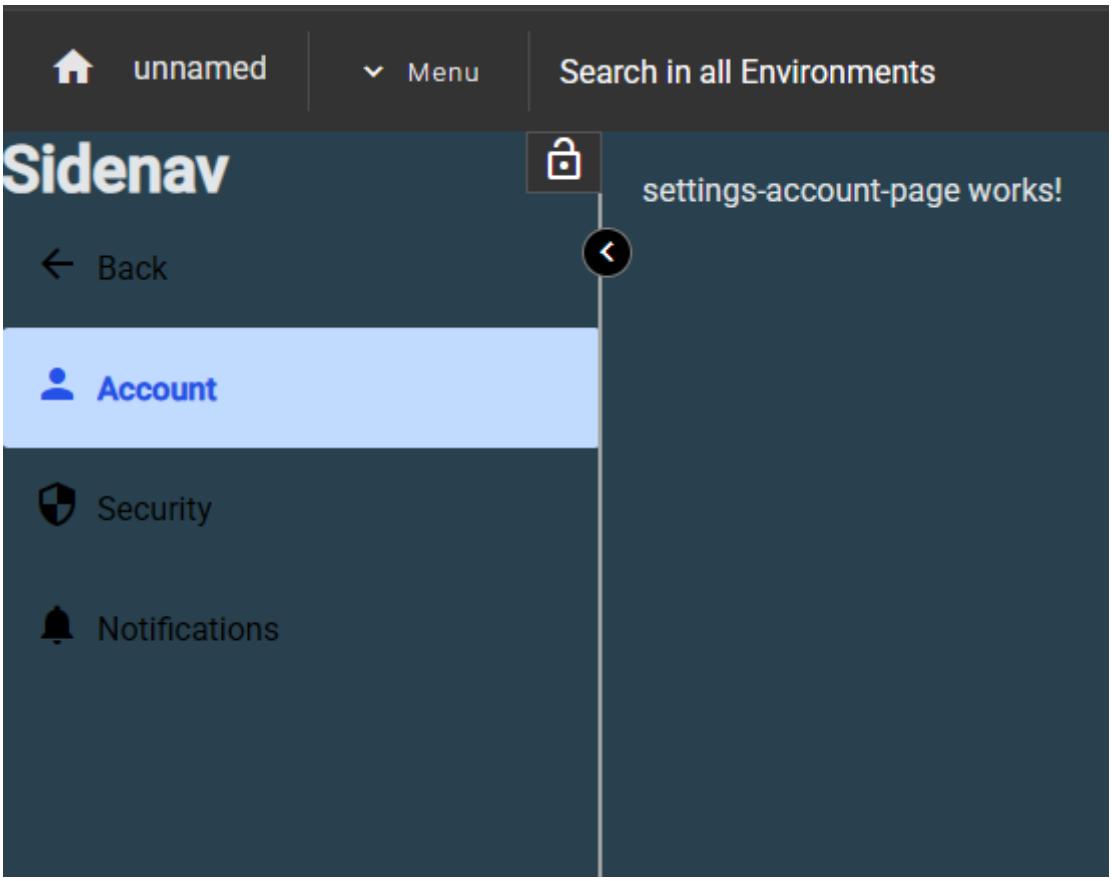
As your application grows in complexity, it will be come necessary to add layers to your side navigation menus.

One way to do this, is to swap in the different layers, based on context.

For example: Below is a side-nav menu with a top-level entry list:



If the user presses "Settings", a secondary list is swapped in, like this:



The user can navigate the settings related pages.

And, they can go back to the top-level list by pressing 'Back'.

The side nav service does the dirty work of tracking a stack of menus as they are drilled down and drilled up (pushed and popped).

All the consuming logic has to do is, push() a new menu, to 'drill down', and pop() the menu when 'drilling up'.

Library Usage

Containing Component Needs

Whatever you define that will hold the side nav element (Likely an app-layout component), will need to:

- Push the initial menu content to the side nav service, when opening.
- Pop the menu set, when closing.

Here's what a partial layout component would need, to do the above:

```
import { Component, ChangeDetectorRef } from '@angular/core';
import { AfterViewInit, OnDestroy } from '@angular/core';

import { DefaultSidenavComponent } from 'somewhere';

export class AppLayoutComponent implements AfterViewInit, OnDestroy {

  constructor(public sidenavService: SidenavService,
              private cdr: ChangeDetectorRef)
  {
  }

  ngAfterViewInit() {
    this.sidenavService.push(DefaultSidenavComponent);
    this.sidenavService.CollapseSidenav();
    this.cdr.detectChanges();
  }

  ngOnDestroy() {
    this.sidenavService.pop();
    this.cdr.detectChanges();
  }
```

Routing Side Nav Service

To automate menu content changes, your app can include a routing side nav service, that will hook into the routing flow, to watch for navigation changes that require a different menu list.

Here's a working routing side nav service that handles two menu sets: default and settings.

```
import { Injectable, OnDestroy } from '@angular/core';

import { Router } from '@angular/router';
import { NavigationEnd } from '@angular/router';

import { filter, map } from 'rxjs';

// Need a reference to the side nav service...
import { SidenavService } from 'side-nav';
```

```
// Import the side nav entry components...
import { SettingsSidenavComponent } from './settings-sidenav/settings-sidenav.component';
import { DefaultSidenavComponent } from './default-sidenav/default-sidenav.component';



@Injectable({
  providedIn: 'root'
})
export class RoutingSidenavService implements OnDestroy
{
  //#region Private Fields

  static lastusedInstanceId: number = 0;
  private instanceId: number = 0;

  // Store the last known side-nav key that is being presented...
  private currentSidenavKey: string | null = null;

  //#endregion



  //#region ctor / dtor

  constructor(private _snsvc:SidenavService,
              private router: Router)
  {
    RoutingSidenavService.lastusedInstanceId++;
    this.instanceId = RoutingSidenavService.lastusedInstanceId;
    console.log("RoutingSidenavService-" + this.instanceId + ":constructor - triggered.");
    // Subscribe to router events...
    this.router.events
      .pipe(
        // Only respond to completed navigation events (i.e., when a route is fully activated)...
        filter(event => event instanceof NavigationEnd),
        // Start at the root route...
        map(() => this.router.routerState.root),
        // Traverse down to the deepest active route (where the sidenav data is most likely defined)...
        map(route => {
          while (route.firstChild) route = route.firstChild;
          return route;
```

```
      }),
      // Extract the `sidenav` metadata from the route's snapshot (defined in the route config `data`)...
      map(route => route.snapshot.data['sidenav']),
      // Filter for only events, where the sidenav key is different than what we have...
      // This prevents swapping in side-nav entries when the key hasn't changed.
      filter(sidenavKey => sidenavKey !== this.currentSidenavKey)
    )
    // This is our callback logic that will make the request to swap in the needed side nav entries,
    //  based on the side-nav key.
    .subscribe((sidenavKey) => {
      // Store the new side nav key...
      this.currentSidenavKey = sidenavKey;

      // Perform the necessary swap...
      switch (sidenavKey) {
        case 'settings':
          this._snsvc.push(SettingsSidenavComponent);
          break;
        case 'default':
        default:
          this._snsvc.pop();
          break;
      }
    });
  }

  ngOnDestroy()
  {
    console.log("RoutingSidenavService-" + this.instanceId + ":ngOnDestroy - triggered.");
    // Drill up as far as we can nest...
    this._snsvc.pop();
    this._snsvc.pop();
  }

  //#endregion
}
```

Menu Content Implementations

And, you need to generate the menu entry content for a default menu, and any other menu sets you need.

Default Menu Content

Here's what a default menu looks like:

```
<app-sidenav-link routerLink="/home">
  <mat-icon icon>home</mat-icon>
  Home
</app-sidenav-link>

<app-sidenav-link routerLink="/backups">
  <mat-icon icon>backup</mat-icon>
  Backups
</app-sidenav-link>

<app-sidenav-link routerLink="/storage">
  <mat-icon icon>storage</mat-icon>
  Storage
</app-sidenav-link>

<app-sidenav-link routerLink="/users">
  <mat-icon icon>people</mat-icon>
  User Mgmt
</app-sidenav-link>



<app-sidenav-link routerLink="/settings">
  <mat-icon icon>settings</mat-icon>
  Settings
</app-sidenav-link>

<app-sidenav-link routerLink="/profile">
  <mat-icon icon>account_circle</mat-icon>
  Profile
</app-sidenav-link>

<app-sidenav-link routerLink="/admin">
  <mat-icon icon>security</mat-icon>
  Admin Page
```

```
</app-sidenav-link>
```

And, it's component definition:

```
import { Component } from '@angular/core';

import { MatIconModule } from '@angular/material/icon';

import { SidenavLinkComponent } from 'side-nav';



@Component({
  // selector: 'default-sidenav',
  imports:
  [
    MatIconModule,
    SidenavLinkComponent
  ],
  templateUrl: './default-sidenav.component.html',
  styleUrl: './default-sidenav.component.scss'
})
export class DefaultSidenavComponent {

}
```

And, here's what a secondary menu set looks like.
Our example is a second-level Settings menu.

```
<app-sidenav-link
  routerLink="/"
  [routerLinkActiveOptions]="{ exact: true }">
  <mat-icon icon> arrow_back </mat-icon>
  Back
</app-sidenav-link>

<app-sidenav-link routerLink="/settings">
  <mat-icon icon> settings </mat-icon>
  Settings Home
</app-sidenav-link>
```

```
<app-sidenav-link routerLink="/settings/account">
  <mat-icon icon> person </mat-icon>
  Account
</app-sidenav-link>

<app-sidenav-link routerLink="/settings/security">
  <mat-icon icon> security </mat-icon>
  Security
</app-sidenav-link>

<app-sidenav-link routerLink="/settings/notifications">
  <mat-icon icon> notifications </mat-icon>
  Notifications
</app-sidenav-link>
```

And, its corresponding component definition:

```
import { Component } from '@angular/core';

import { MatIconModule } from '@angular/material/icon';

import { SidenavLinkComponent } from 'side-nav';



@Component({
  // selector: 'settings-sidenav',
  imports:
  [
    MatIconModule,
    SidenavLinkComponent
  ],
  templateUrl: './settings-sidenav.component.html',
  styleUrl: './settings-sidenav.component.scss'
})
export class SettingsSidenavComponent {

}
```

Conclusion

With the above elements defined in your app, you should have a working side nav implementation.

Angular: Access CSS Variables

CSS Variables can be defined in css (or SCSS) like this:

> NOTE: These are defines in the root element, in the styles.scss file of an app.

```
:root {
    --sidenav-width: 300px;
    --sidenav-collapsed-width: 50px;
    --sidenav-background-color: var(--app-background-color);
    --sidenav-text-color: var(--app-text-color);
    --sidenav-transition-duration: 400ms;
    --app-background-color: #b6b9bd;
    --app-text-color: #282828;
}
```

The above defines several global CSS variables (root scoped).
It as well, shows how to define a variable that takes the value of another.

Reading

Here's how we can access these global CSS variables from an Angular component:

```
const sidenavTransitionDurationFromCssVariable =
        getComputedStyle(document.body)
        .getPropertyValue('--sidenav-transition-duration');


let dur = parseInt(sidenavTransitionDurationFromCssVariable, 10);
```

The above code snippet retrieves a global CSS variable called, sidenav-transition-duration, from the DOM, and parses it into an integer (with a default value, if not found).

This allows an Angular component to use data stored in CSS.

Writing

Here's how to write a new value to the same global CSS variable from an Angular component:

```
document.documentElement.style
    .setProperty('--sidenav-transition-duration', '300');
```

The above statement writes a '300' to the transition duration variable that is at :root scope of the DOM.

> NOTE: All writes to variables are as strings, so you will have to convert to string before the setProperty call.

Web Push Notifications

Here’s some initial references for setting up Push Notifications in web pages.

Service Worker API - Web APIs | MDN

Push API - Web APIs | MDN

Push notifications overview | Articles | web.dev

Push notifications are now supported cross-browser | Blog | web.dev

Web Push for Web Apps on iOS and iPadOS