Angular Component Book

Describes reusable components for app development.

- Runtime Security Service
- Generic Web Reply
- REST Base Service
- Layout Service
- App Local Storage Service

Runtime Security Service

Add the Library

Add the library to your app.

App.Config.ts (formerly App.Module.ts)

Import the service and injection token:

```
import { RUNTIMESECURITYSERVICE_INJTOKEN } from 'lib-oga-webui-sharedkernel';
import { RuntimeSecurityService } from 'lib-oga-webui-runtimesecurity';
```

If you are using the mock runtime security service, use this, instead:

```
import { RUNTIMESECURITYSERVICE_V2_INJTOKEN } from 'lib-oga-webui-sharedkernel';
import { RuntimeSecurityService_v2_MOCK } from 'lib-oga-webui-runtimesecurity';
```

Register it with DI, in a provider entry:

```
providers:
[
  // Register the Runtime Security Service for DI consumption...
  { provide: RUNTIMESECURITYSERVICE_INJTOKEN, useExisting: RuntimeSecurityService },
],
```

If you are using the mock runtime security service, use this, instead:

```
providers:
[
  // Register the Runtime Security Service for DI consumption...
  { provide: RUNTIMESECURITYSERVICE_V2_INJTOKEN, useExisting: RuntimeSecurityService_v2 },
],
```

Generic Web Reply

This is a class that is returned by the REST Base Service: Runtime Security Service

It is a generic struct, returned by a web call, that contains the status of the call, any error that occurred, and the data if received.

It has the following properties:

- data - Contains the typed response body from the call.
- err - If an error occurred, this may contain the error message.
- res - Contains the method call return value. 1 for success.
- statuscode - Contains the Http Status code of the response.
- timedout - Will be set if the call timed out, waiting for a reply.

It also has these methods:

- IsReturnGood() - Checks that the return code is success and the Http Status code is 200-299.

- IsReturnGoodDataExists() - Same as previous, but also checks that data is not null.

REST Base Service

We have a base REST API service that we can leverage in the entire app stack.

It encapsulates the ceremony of dealing with backend calls, and provides us with a clean set of generic, async methods.

It's located in: lib-oga-webui-sharedkernel.

To use in a component or service, import it with this:

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

Add it to your constructor or a private injection:

```
constructor(private _restsvc: RestBaseService)
{
  ...
}
```

Now, you can use its various calls.

Each call has these properties:

- Uses one of the four verbs: GET, POST, PUT, DELETE.
- Accepts the endpoint path fragment as string.
  Composes the url from the app origin and the api_BasePath property from environment.ts.
- Accepts an optional header instance, of type HttpHeaders.
- Returns a Promise.
  Use await to call the method.
- Returns a generic web response object. See this for details: Generic Web Reply).
  The concrete type must be specified when calling the method.

API URL

Each method composes the url based on the app's origin and the api_BasePath property from environment.ts.

The url is composed of this form:

`${origin}${this.env.api_BasePath}/${urlpath}`

The origin will be the app's origin, unless overridden.

> You can override the origin used by setting the public property, OriginOverride, of the REST service instance before making the call.

The api_BasePath is defined in your app's environment.ts file.

The urlpath is what your calling method passes in.
It should include any query parameters, already urlencoded.

Example Usage

> NOTE: The endpoint passed to the method does NOT need to include a leading '/'. This is appended for you.

GET - Returning a void:

```
let res = await this._restsvc.GetRequest<void>("VoidTests_V1/Get/AcceptVoid/ReturnVoid");
if(!res || res === null || !res.IsReturnGood())
{
  // Call failed.
}
// Call returned success.
```

GET - Returning a type:

```
let res = await this._restsvc.GetRequest<sometype>("API_endpoint/Get/Somecall");
if(!res || res === null || !res.IsReturnGood())
{
  // Call returned an error or failed.
}
// Call returned success.
```

POST - Accepting a type and returning a type:

```
let res = await this._restsvc.PostRequest<requesttype,responsetype>("API/ENDPOINT", requestdata);
if(!res || res === null || !res.IsReturnGoodDataExists())
{
  // The call failed, or returned no data.
```

```
}
// It succeeded, and we have data.
```

PUT - returning a type:

```
let res = await this._restsvc.PutRequest<requesttype,responsetype>("API/ENDPOINT", requestdata);
if(!res || res === null || !res.IsReturnGoodDataExists())
{
  // The call failed, or returned no data.
}
// It succeeded, and we have data.
```

DELETE - returning a void:

```
let res = await this._restsvc.DeleteRequest<responsetype>("API/ENDPOINT");
if(!res || res === null || !res.IsReturnGood())
{
  // The call failed, or returned no data.
}
// It succeeded, and we have data.
```

Layout Service

The layout service provides the means to display and swap in outer-frames of an app.

This service is given two things:

- Layout Anchor - that tells where to inject outer frame layouts into the DOM
- Layout Choices - an array list of available outer frames, organized by name

Code Location

This service, its directive and model, are currently located in the common-lib library of the Backups UI workspace project.

That library will eventually get moved to its own monorepo, and independently built, for app consumption.

Usage

Here's steps to use the layout service.

App.Config Updates

In the app.config.ts component of your app, you need to do the following:

Import the layout service and choices array type:

```
import { LayoutChoices } from './layout-control/model/ILayoutChoice';
import { LayoutService } from './layout-control/services/layout.service';
```

Inject the layout service into your app.config.ts, like this:

```
const _layoutsvc = inject(LayoutService);
```

If your app uses an initialization function, provided by provideAppInitializer, update your `initializeApp` function to create a list of layouts and send them to the layout service for use.

Here's a snippet that will do so, and can be added to your `initializeApp` function:

```
// This method is called when the app first starts up.
// It will load any initial config, and setup anything required.
export function initializeApp(): Promise<boolean>
//export function initializeApp(): () => Promise<boolean>
{
  const _appconfig = inject(AppconfigService);
  const _ctsvc = inject(ColorThemingService);
  const _layoutsvc = inject(LayoutService);
  const env = inject(ENVIRONMENT_INJTOKEN);

  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 other startup things...

    // Define layouts...
    {
      console.log("AppModule-" + "_" + ":initializeApp - Defining layout choices...");

      // Create the layout choice listing...
      let lc:LayoutChoices =
      [
        {
          name: 'main',
          component: AppLayoutComponent,
          requireLogin: false
        },
        {
          name: 'error',
          component: Error404PageComponent,
          requireLogin: false
        },
        {
```

```
          name: 'test',
          component: TestPageComponent,
          requireLogin: false
        },
        {
          name: 'login',
          component: LoginPageComponent,
          requireLogin: false
        }
      ];

      // Pass it to the layout service...
      if(_layoutsvc.DefineLayoutChoices(lc) === false)
      {
        console.error("AppModule-" + "_" + ":initializeApp - Failed to set layout choices.");

        return false;
      }
    }

    // Report success...
    console.log("AppModule-" + "_" + ":initializeApp - returned.");
    return true;
  })(); // Notice the () at the end - this immediately invokes the async lambda
}
```

App.Component Updates

For the layout service to swap outer frames of your app, it must have an anchor point for insertion.

For an app with multiple outer frames (for login, app-layout, testing, error, etc), the ideal spot for the outer frame anchor is in App.Component.

Here are changes to incorporate in the pieces of App.Component:

app.component.html:

```
<div class="fill-parent">
  <!-- Below is the a dynamic insertion point for anchoring components to the empty page -->
  <ng-template EmptyFrameAnchor></ng-template>
</div>
```

The above html template include the anchor directive on an ng-template element.
And, it contains a div that fills the parent width and height.

app.component.ts:

```
import { Component, Inject, ViewChild } from '@angular/core';
import { OnInit, AfterViewInit, AfterContentInit, OnDestroy } from '@angular/core';

import { mergeMap, takeUntil } from 'rxjs/operators';
import { Subject } from 'rxjs';

// Import things we need, for swapping layouts...
import { EmptyFrameAnchorDirective } from './layout-control/directives/emptyframe-anchor.directive';
import { LayoutService } from './layout-control/services/layout.service';

// Import security service.
// This module subscribes to the global login state, so that it can flip to the Login page if no one is logged in.
import { IRuntimeSecurityService } from 'lib-oga-webui-sharedkernel';
import { RUNTIMESECURITYSERVICE_INJTOKEN } from 'lib-oga-webui-sharedkernel';



@Component({
  selector: 'app-root',
  imports:
  [
    EmptyFrameAnchorDirective
  ],
  templateUrl: './app.component.html',
  styleUrl: './app.component.scss'
})
export class AppComponent implements OnInit, AfterViewInit {
  // Define an anchor reference for where layouts will be pushed...
  @ViewChild(EmptyFrameAnchorDirective, {static: true}) domanchor!: EmptyFrameAnchorDirective;

  constructor(private _layoutsvc:LayoutService,
              @Inject(RUNTIMESECURITYSERVICE_INJTOKEN) private _securityService: IRuntimeSecurityService)
  {
  }

  ngOnInit(): void
```

```
  {
    this.#_Initialize_DOMAnchor();

    // Check if the security service can notify us of login events...
    if(this._securityService.Has_Privilege_to_Open_App$ == null)
    {
      console.error("AppComponent-" + this.instanceId + ":ngOnInit - security service instance is null.");
      throw new Error("domanchor is undefined");
    }

    // Setup an observable that will take action on login event changes...
    // This observable will tell the profile service to load the appropriate component based on user login state.
    // It will remain active until this component is destroyed.
    this._securityService.Has_Privilege_to_Open_App$
      .pipe(
        takeUntil(this.destroySubject),
        mergeMap(has_privilege_to_open_app => Promise.resolve(null).then(() =>
        {
          console.log("AppComponent-" + this.instanceId + ":ngOnInit - observed has_privilege_to_open_app
change state to: " +
                    has_privilege_to_open_app);

          console.log("AppComponent-" + this.instanceId + ":ngOnInit - " +
                    "telling the layout service to swap outer frames for the user change event...");
          this.SwapBasePageforLoginEvent(has_privilege_to_open_app);

          console.log("AppComponent-" + this.instanceId + ":ngOnInit - " +
                    "SwapBasePageforLoginEvent swapped in the desired frame for the user security event.");
        })
        )
      )
      .subscribe();
  }

  ngAfterViewInit() {
    console.log("AppComponent-" + this.instanceId + ":ngAfterViewInit - triggered.");
  }

  // Private method that locates the outer frame anchor and sends it to the layout service.
  #_Initialize_DOMAnchor()
```

```
  {
    // Throw an exception of the layout dom anchor can't be found...
    if (!this.domanchor) {
      console.error("AppComponent-" + this.instanceId + ":_Initialize_DOMAnchor - Cannot locate layout DOM
anchor.");
      throw new Error("domanchor is undefined");
    }

    // Give the layout anchor point to the layout service...
    this._layoutsvc.SetLayoutAnchor(this.domanchor);
  }

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

    // Unhook and close out subscriptions to the log in observable...
    this.destroySubject.next();
    this.destroySubject.complete();
  }



}
```

The above logic does several things:

- It locates the anchor point, where outer frames are swapped into the DOM.
- Sends the outer frame anchor point to the layout service, for use.
- Subscribes to user login events, to be notified when a user logs in and out.
-

App Local Storage Service

This service allows an app to store data in the browser's localstore.

The local store is treated as a key-value store, with generic method calls, that will serialize and deserialize data as needed.

Code Location

This service is located in library: lib-oga-webui-sharedkernel.

Usage

Below are usage example for the call surface of this service.

Saving Data

Storing a value to the local store, like this:

```
const _als = inject(AppLocalStorageService);
const keyname = 'auth-token';

const tokenvalue = 'ssdfjhsldfjhlsakjdfslakjdfhslkdjfhszdlkjfhszdjkl';
this._als.SaveValue(keyname, tokenvalue);
```

The above injects the local store service, and saves a string to it.

Loading Data

Retrieving a value from the store, like this:

```
const _als = inject(AppLocalStorageService);

const keyname = 'auth-token';

let rr = this._als.GetValue<string>(keyname);

if(!rr || rr === null)
```

```
{
  console.log("TokenStorageService-" + this.instanceId + ":getRefreshToken - no token stored.");
}
```

Deleting Data

Entries are deleted by key name.

```
const _als = inject(AppLocalStorageService);

const keyname = 'auth-token';

this._als.RemoveKey(keyname);
```

Get Key List

This will retrieve the list of keys of stored things.

```
const _als = inject(AppLocalStorageService);

let kl = service.GetKeys();

console.log("Key list is: " + JSON.stringify(kl));
```