Contents

[Project Structure 2](#_Toc64390490)

[Common Module 2](#_Toc64390491)

[Feature Module 2](#_Toc64390492)

[Component 3](#_Toc64390493)

[Aliases for imports 3](#_Toc64390494)

[Using SASS 4](#_Toc64390495)

[Unit Tests 6](#_Toc64390496)

[NGRX 6](#_Toc64390497)

[Naming conventions 7](#_Toc64390498)

[File Naming 7](#_Toc64390499)

[Class Naming 7](#_Toc64390500)

[Variable Naming 7](#_Toc64390501)

[Selector Naming 7](#_Toc64390502)

[Git branch 8](#_Toc64390503)

[Coding best practices 8](#_Toc64390504)

[Subscribe in Template 8](#_Toc64390505)

[Unsubscribe observables in component/ Service 9](#_Toc64390506)

[trackBy with \*ngFor 9](#_Toc64390507)

[Avoid having subscriptions inside subscriptions 10](#_Toc64390508)

[Avoid any; type everything. 10](#_Toc64390509)

[Avoid logic in templates 10](#_Toc64390510)

[Add caching mechanisms 10](#_Toc64390511)

[Avoid long methods 10](#_Toc64390512)

[Small reusable components 11](#_Toc64390513)

[Use appropriate RXJS operators 11](#_Toc64390514)

[ES6 Feature 11](#_Toc64390515)

[Uglification/Minification 11](#_Toc64390516)

[Security 11](#_Toc64390517)

[Best Practices 11](#_Toc64390518)

[Trusting safe values 11](#_Toc64390519)

[Use of Renderer2 12](#_Toc64390520)

[Logging Standards 12](#_Toc64390521)

[Logging Levels 12](#_Toc64390522)

[Environment based Logging 12](#_Toc64390523)

[Add Context to Log Messages 13](#_Toc64390524)

[Log in Machine Parseable Format 13](#_Toc64390525)

[Don’t Log Sensitive Information 13](#_Toc64390526)

[Logging Implementation 13](#_Toc64390527)

[Telemetry/Application Insights 13](#_Toc64390528)

[Implementation Approach 14](#_Toc64390529)

[**Step 1 – Add Application Insights to your Azure subscription** 14](#_Toc64390530)

[**Step 2 – Configure Angular application and include dependencies** 14](#_Toc64390531)

[**Step 3 – Consume the Application Insights SDK in TypeScript** 15](#_Toc64390532)

[**Step 4 – Add proper error handling** 16](#_Toc64390533)

[VSCode Extensions 16](#_Toc64390534)

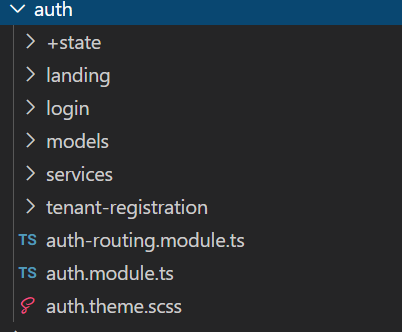
[External References 17](#_Toc64390535)

# Project Structure

## Common Module

* Build a CommonModule with the components, directives, and pipes that you use throughout your app. This module should consist completely of declarations, most of them exported.
* Import the CommonModule in your feature modules, both those loaded when the app starts and those you lazy load later

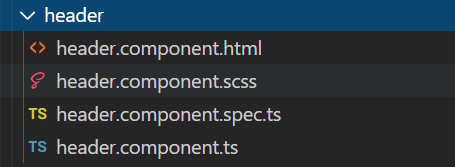
## Feature Module



1. +**state** folder contains files related to ngrx.
2. Should have **<feature>.theme.scss** file which contains color styling related mixins and styles.
3. If it contains route then create **<feature>-routing.module.ts** file.

## Component

1. Component should follow below file structure.
2. Avoid manually component creation instead use angular cli commands.



## Aliases for imports

Aliasing our app and environments folders will enable us to implement clean imports which will be consistent throughout our application.

Import statements looking something like

import { AuthService } from '../../auth/services/auth.service'

should be used as below

import { AuthService } from '@app/auth/services/auth.service'

make sure you add configuration in tsconfig.json as below

"paths": {

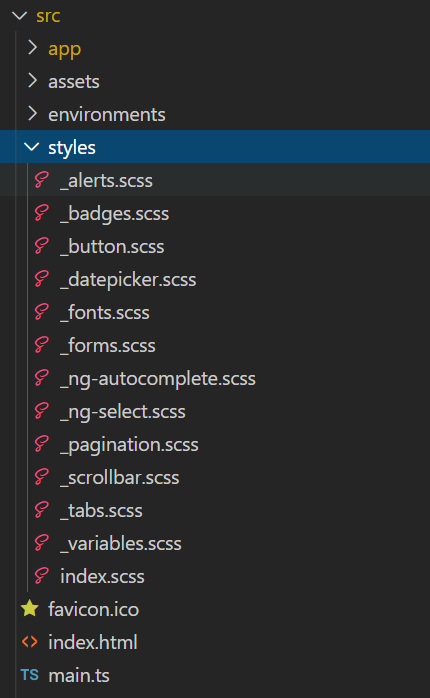
      "@app/\*": ["src/app/\*"]

    }

## Using SASS

SASS is a styles preprocessor which brings support for variables, functions, mixins etc.

The global styles for the project are placed in a styles folder. Any new component styles can be added under this folder.



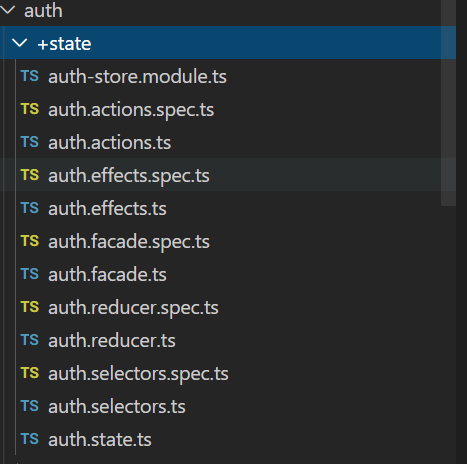
styles.scss file imports index.scss file from style folder.

## Unit Tests

* Name test specification files the same as the component they test.
* Name test specification files with a suffix of .spec. Example user.component.spec.ts
* If your mocks/stubs are being used multiple places then create separate folder mocks and keep all mocks there.

## NGRX

To keep state management for projects, we can go with NGRX. It varies based upon schematics used.



Below schematics are suggested to generate NGRX feature module

1. @ngrx/schematics
2. @nrwl/angular

# Naming conventions

## File Naming

|  |  |  |
| --- | --- | --- |
| Type | Convention | Example |
| Component | <kebab-case>.component.ts | user.component.ts |
| Service | <kebab-case>.service.ts | user.service.ts, user-info.service.ts |
| Pipe | <kebab-case>.pipe.ts | truncate.pipe.ts |
| Module | <kebab-case>.module.ts | user.module.ts |
| Directive | <kebab-case>.directive.ts | order.directive.ts |

## Class Naming

|  |  |  |
| --- | --- | --- |
| Type | Convention | Example |
| Component | <UpperCamelCase>Component | UserComponent |
| Service | <UpperCamelCase>Service | UserService |
| Pipe | <UpperCamelCase>Pipe | TruncatePipe |
| Module | <UpperCamelCase>Module | UserModule |
| Directive | <UpperCamelCase>Directive | OrderDirective |
| Interface | UpperCamelCase | User |

## Variable Naming

|  |  |  |
| --- | --- | --- |
| Type | Convention | Example |
| Property | lowerCamelCase | userInfo |
| Const, let | lowerCamelCase | isLoggedIn |
| CONSTANT Application level | UPPER\_SNAKE\_CASE | USER\_TOKEN |

## Selector Naming

|  |  |  |
| --- | --- | --- |
| Type | Convention | Example |
| Component | kebab-case | app-user |
| Directive | lowerCamelCase | appUser |
| Pipe | lowerCamelCase | truncate |

## Git branch

|  |  |
| --- | --- |
| Type | Example |
| Feature | feature/ticketId-description |
| Bugfix | bugfix/ticketId-description |
| Test Cases | tests/ticketId-description |

# Coding best practices

## Subscribe in Template

With the async pipe, we can subscribe to the stream directly inside our template, without having to store the result in an intermediate property. The subscription will terminate when the component gets destroyed, which makes subscription-handling easier and contributes to cleaner code.

Instead using below code in component TS file

this.loadItems()

  .pipe(

    map(items => this.items = items;

  ).subscribe();

Use below code

 @Component({

    ...

    template: `<items [items]="items$ | async"></items>`

})

class AppComponent {

  items$: Observable<Item[]>;

  constructor(private itemService: ItemService) {}

  ngOnInit() {

    this.items = this.loadItems();

  }

  loadItems(): Observable<Item[]> {

    return this.itemService.findItems();

  }

}

## Unsubscribe observables in component/ Service

Sometimes due to complex logic we may need to subscribe observables inside component and services. In that case make sure we unsubscribe it as well.

this.authFacade.error$

      .pipe(takeUntil(this.unsubscribe))

      .subscribe((error: any) => {}

…

ngOnDestroy(): void {

    this.unsubscribe.next();

    this.unsubscribe.complete();

 }

## trackBy with \*ngFor

When using ngFor to loop over an array in templates, use it with a **trackBy** function which will return an unique identifier for each item.

When an array changes, Angular re-renders the whole DOM tree. But if you use **trackBy**, Angular will know which element has changed and will only make DOM changes for that element.

// in the template

<li \*ngFor="let item of items; trackBy: trackByFn">{{ item }}</li>

// in the component

trackByFn(index, item) {

   return item.id; // unique id corresponding to the item

}

## Avoid having subscriptions inside subscriptions

Sometimes you may want values from more than one observable to perform an action. In this case, avoid subscribing to one observable in the subscribe block of another observable. Instead, use appropriate chaining operators. Chaining operators run on observables from the operator before them. Some chaining operators are: **withLatestFrom**, **combineLatest**, etc.

## Avoid any; type everything.

Always declare variables or constants with a type other than **any**.

## Avoid logic in templates

If you have any sort of logic in your templates, even if it is a simple **&&** clause, it is good to extract it out into its component.

## Add caching mechanisms

When making API calls, responses from some of them do not change often. In those cases, you can add a caching mechanism and store the value from the API. When another request to the same API is made, check if there is a value for it in the cache and if so, use it. Otherwise, make the API call and cache the result.

If the values change but not frequently, you can introduce a cache time where you can check when it was last cached and decide whether to call the API.

## Avoid long methods

Long methods generally indicate that they are doing too many things. Try to use the Single Responsibility Principle. The method itself as a whole might be doing one thing, but inside it, there are a few other operations that could be happening. We can extract those methods into their own method and make them do one thing each and use them instead.

## Small reusable components

Extract the pieces that can be reused in a component and make it a new one. Make the component as dumb as possible, as this will make it work in more scenarios. Making a component dumb means that the component does not have any special logic in it and operates purely based on the inputs and outputs provided to it.

## Use appropriate RXJS operators

When using flattening operators with your observables, use the appropriate operator for the situation.

*switchMap:* when you want to ignore the previous emissions when there is a new emission

*mergeMap:* when you want to concurrently handle all the emissions

*concatMap:* when you want to handle the emissions one after the other as they are emitted

## ES6 Feature

**var** versus **let** versus **const**

* Prefer const. Use it for all variables whose values never change.
* Otherwise, use let – for variables whose values do change.
* Avoid var.

## Uglification/Minification

For production deployment use **ng build --prod** command which handles uglification and minification as well.

# Security

## Best Practices

* Cross-Site Scripting (XSS) Prevention-
* Regular Expression Attributes
* Regular Expression Object Model
* HTML Encoding
* URL Encoding
* Keep application with the latest library releases.
* Avoid using APIs marked in the documentation as “*Security Risk*.”
* Avoid using deprecated API’s.
* Custom Error Page for Error Handling. Do not disclose sensitive information in error responses, including system details, session identifiers or account information
* Proper Authentication and Session Management.
* Role based Security to protect client-side resources.
* Logging and Auditing- Do not log sensitive data.
* Avoid direct DOM manipulation.
* Specify proper character sets, such as UTF-8, for all sources of input.
* Proper input data validation for length, type of data, data range & allowed characters wherever applicable.
* Use proper HTTP methods to pass data to server for example-Use only HTTP POST requests to transmit authentication credentials.
* On Logout functionality all the session related information including local, or session storage should be completely removed.
* Use of encryption/decryption technique.
* Disable client-side caching on pages containing sensitive information. Cache-Control: no-store, may be used in conjunction with the HTTP header control "Pragma: no-cache", which is less effective, but is HTTP/1.0 backward compatible

## Trusting safe values

To mark a value as trusted, inject DomSanitizer and call one of the following methods:

* bypassSecurityTrustHtml
* bypassSecurityTrustScript
* bypassSecurityTrustStyle
* bypassSecurityTrustUrl
* bypassSecurityTrustResourceUrl

## Use of Renderer2

The **Renderer2** class is an out of the box feature from Angular which provides an abstraction for manipulating UI rendering elements of your app at runtime without the need to access the DOM directly.

We can use a custom renderer to bypass Angular's templating and make custom UI changes that can't be expressed declaratively. For example if you need to set a property or an attribute whose name is not statically known, use the setProperty() or setAttribute() method.

# Logging Standards

## Logging Levels

|  |
| --- |
|  |
|  |

* ***TRACE level***: This should be used during development to track bugs.
* ***DEBUG level***: This is mostly used during debugging and trimming down the number of debug statement before entering the production stage, so that only the most meaningful entries are left, and can be activated during troubleshooting.
* ***INFO level***: Log at this level all actions that are user-driven, or system specific (ie regularly scheduled operations).
* ***WARN level***: log at this level all events that could potentially become an error. For instance if one database call took more than a predefined time, or if an in-memory cache is near capacity. This will allow proper automated alerting, and during troubleshooting will allow to better understand how the system was behaving before the failure.
* ***ERROR level***: log every error condition at this level. That can be API calls that return errors or internal error conditions.
* ***FATAL level***: Use this very scarcely, this shouldn’t happen a lot in a real program. Usually logging at this level signifies the end of the program. For instance, if a network daemon can’t bind a network socket, log at this level and exit is the only sensible thing to do.

## Environment based Logging

Logging will be configurable environment wise. Like on production and similar higher environment we can log only few log levels like Errors while for dev environment all logs can be enabled.

## Add Context to Log Messages

* 1. [Method Name] : Message
  2. [Action]: Message

## Log in Machine Parseable Format

Log message which can be parsed by another programs as well.

Example: Instead logging message in below format

UserRequest User 1334563 plays 4 of spades in game 23425656

Use Below format

UserRequest User plays {'user':1334563, 'card':'4 of spade', 'game':23425656}

## Don’t Log Sensitive Information

We should not log below information

* Passwords
* Credit card numbers
* Social security numbers
* Session identifiers Information the user has opted out of
* Authorization tokens
* PII (Personal Identifiable Information, such as personal names)

# Logging Implementation

Logging is something which is not needed to reinvent the implementation.

There are open source libraries/ documentation for logging implementation in angular.

1. Use existing Sky1.0 Logging implementation
2. Create custom logging service
3. Angular libraries for logging
   1. [Ngrx-logger](https://www.npmjs.com/package/ngx-logger)

# Telemetry/Application Insights

## Implementation Approach

Application Insights is an Azure service for monitoring web applications. It includes analytics tools to help you understand application usage and to diagnose issues with the goal of continuously improving performance and usability. In addition to its integration with a variety of backend web platforms, Application Insights also includes browser analytics.

With Application Insights, you have full observability into your application across all components and dependencies of your complex distributed architecture, as well as:

* Availability and performance monitoring
* Deep code diagnostics with profiler and debugger
* Users and usage insights
* Out of the box performance and failures triage with end-to-end transaction diagnostics
* Seamless integration with Microsoft Azure and Visual Studio
* Supports ASP.NET, .NET Core, Java, Node.js applications and websites on Azure or on-premises

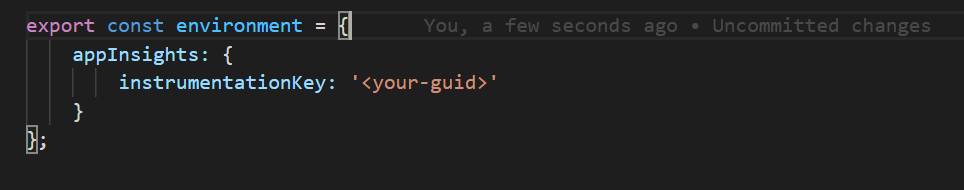
**The following steps will be taken as implementation approach for application insights:**

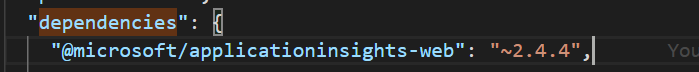
## **Step 1 – Add Application Insights to your Azure subscription**

Using this step, we will receive an Instrumentation key that can be used in step2.

## **Step 2 – Configure Angular application and include dependencies**

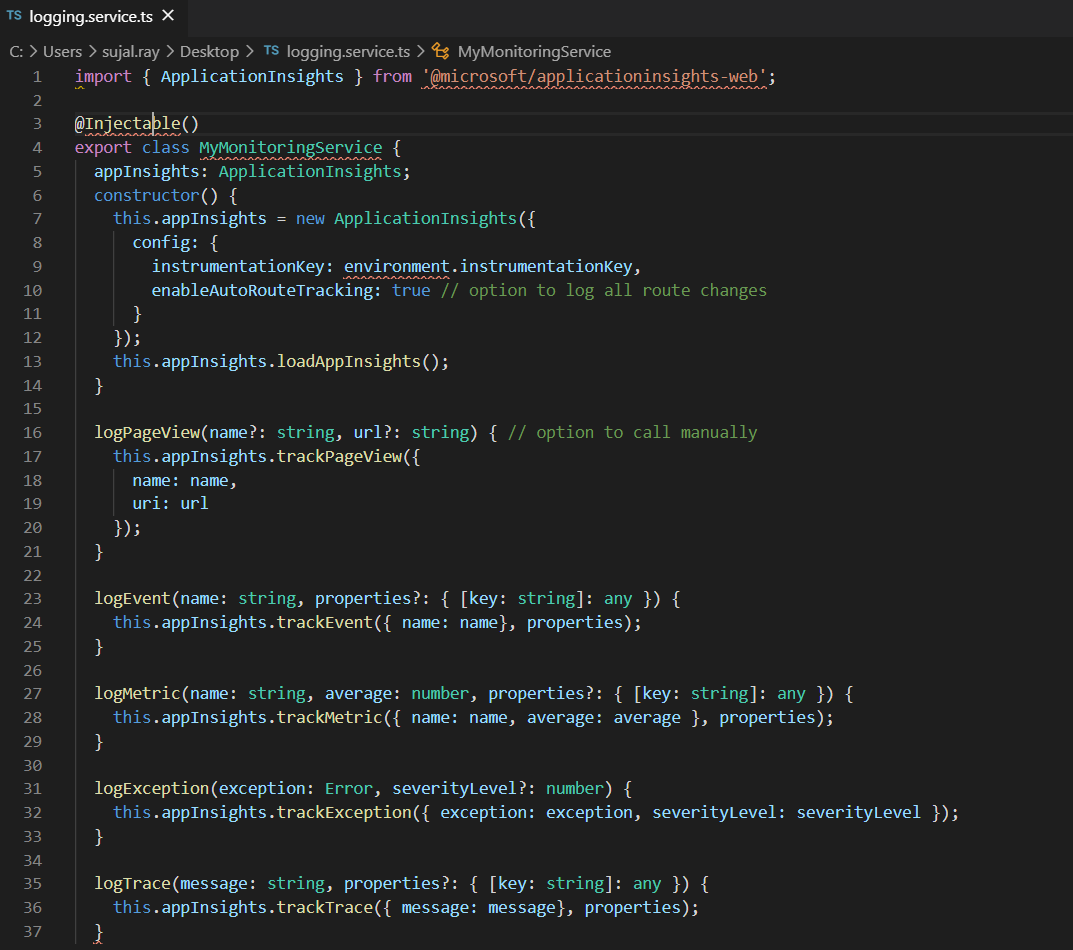
1. Add InstrumentationKey inside environment file or make it part of configuration.



1. Add a dependency to *package.json* and restore using **npm install**. 

## **Step 3 – Consume the Application Insights SDK in TypeScript**

1. Create a TypeScript class as a wrapper around the Application Insights JavaScript API and import the AppInsights class using the module loading system.

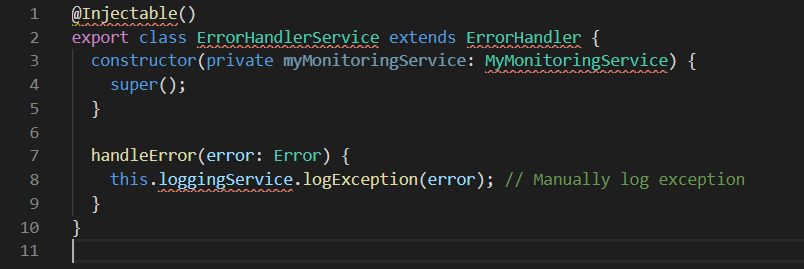


1. Create an instance of the class defined in step 5 as soon as the application starts.

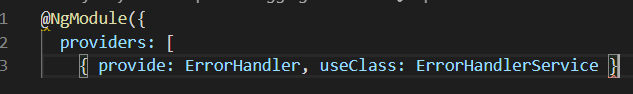
## **Step 4 – Add proper error handling**

Although the Application Insights JS code will log all uncaught exceptions it is not a good experience for users to have uncaught exceptions occurring on a page. It is better to catch all exceptions in a global error handler and manually call the trackException() method.

1. Add a custom error handler.



1. Instruct the module to use the custom error handler.



# VSCode Extensions

Install these extensions

1. Prettier
2. Eslint/TSLint
3. SimonTest
4. NX Console

Add below setting in your vscode setting.json file

  "[html]": {

        "editor.defaultFormatter": "esbenp.prettier-vscode"

    },

    "[typescript]": {

        "editor.defaultFormatter": "esbenp.prettier-vscode"

    },

    "typescript.updateImportsOnFileMove.enabled": "always",

    "javascript.updateImportsOnFileMove.enabled": "always",

    "diffEditor.ignoreTrimWhitespace": false,

    "editor.formatOnSave": true,

    "[json]": {

        "editor.defaultFormatter": "esbenp.prettier-vscode"

    },

    "[javascript]": {

        "editor.defaultFormatter": "esbenp.prettier-vscode"

    },

    "typescriptHero.imports.grouping": [

        "Plains",

        "Modules",

        "Workspace"

    ],

    "editor.codeActionsOnSave": {​​

        "source.fixAll": true,

        "source.organizeImports": true ,

    }

# External References

<https://angular.io/guide/styleguide>

<https://www.freecodecamp.org/news/best-practices-for-a-clean-and-performant-angular-application-288e7b39eb6f/>

<https://itnext.io/clean-code-checklist-in-angular-%EF%B8%8F-10d4db877f74>

<https://devblogs.microsoft.com/premier-developer/angular-how-to-add-application-insights-to-an-angular-spa/>