CODING STANDARDS
Javascript and Typescript

Contents

Introduction

Orbis *Technologies* **develops** and maintains ***enterprise*** CCMS software, solutions, and services to organizations worldwide. Orbis Orbis provide solutions to clients across various industries that including manufacturing, healthcare, publishing, healthcare insurance, education industries and the federal government. Table 1

Table 1.

Thead	Thead	Thead	Thead
Well written software offers many advantages.	Well www.google.com written software offers many advantages.	Well written software offers many advantages.	Well written software offers many advantages.

Well written software offers many advantages. It will contain fewer bugs and will run more efficiently than poorly written programs. Since software has a life cycle and much of which revolves around maintenance, it will be easier for the original developer(s) and future keepers of the code to maintain and modify the software as needed. This will lead to increased productivity of the developer(s). The overall cost of the software is greatly reduced when the code is developed and maintained according to software standards. Table 1

The goal of these guidelines is to create uniform coding habits among software personnel in the engineering department so that reading, checking, and maintaining code written by different persons becomes easier. The intent of these standards is to define a natural style and consistency yet leave to the authors of the engineering department source code, the freedom to practice their craft without unnecessary burden.

Use Angular CLI

Angular CLI is one of the most powerful accessibility tools available when developing apps with Angular. Angular CLI makes it easy to create an application and follows best practices. Angular CLI is a command-line interface tool that is used to initialize, develop, scaffold, maintain, and test and debug Angular applications.

Instead of creating files and folders manually, use Angular CLI to generate new components, directives, modules, services, and pipes.

 # Install Angular CLI

```
npm install -g @angular/cli
```

 # Check Angular CLI version

```
ng version
```

Maintain proper folder structure

Folder structure is an important factor to consider before initiating a project. The following folder structure will easily adapt to new changes during development.

-- app

|-- modules

|-- home

|-- [+] components

|-- [+] pages

|-- home-routing.module.ts

|-- home.module.ts

|-- core

|-- [+] authentication

|-- [+] footer

|-- [+] guards

|-- [+] http

|-- [+] interceptors

|-- [+] mocks

|-- [+] services

|-- [+] header

|-- core.module.ts

|-- ensureModuleLoadedOnceGuard.ts

|-- logger.service.ts

|

|-- shared

|-- [+] components

|-- [+] directives

|-- [+] pipes

|-- [+] models

|

|-- [+] configs

|-- assets

|-- scss

|-- [+] partials

|-- _base.scss

|-- styles.scss

Follow consistent Angular coding styles

Here are some rules to follow to ensure that your project follows the proper coding standard.

- **Limit files to 400 Lines of code.**
- **Define small functions and limit them to no more than 75 lines.**
- **Have consistent names for all symbols. The recommended pattern is feature.type.ts.**

- **If the values of the variables are intact, then declare it with ‘const’.**
- **Use dashes to separate words in the descriptive name and use dots to separate the descriptive name from the type.**
- **Names of properties and methods should always be in lower camel case.**
- **Always leave one empty line between imports and modules, such as third party and application imports and third-party modules and custom modules.**

Use ES6 features

ECMAScript is constantly updated with new features and functionalities. The current version is ES6, which has lots of new features and functionalities that can be utilized in Angular.

Here are a few ES6 Features:

- **Arrow Functions**
- **String interpolation**
- **Object Literals**
- **Let and Const**
- **Destructuring**
- **Default**

Use trackBy along with ngFor

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

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

Use ngFor

```
<tr *ngFor="let item of listComponent;">{{item.id}}</li>
```

Now each time the changes occur, the whole DOM tree will be re-rendered.

Using trackBy function

```
<tr *ngFor="let item of listComponent ">
<td>{{ item.fullPath }}</td>
<td>{{ item.id && item.id.split('-').pop() }}</td>
<td>{{ item.userId }}</td>
</tr>
```

Load items

```
export class listComponent {
lockedItems =[];
getItems() {
this.lockedItems = items;
}
trackByFn(index, item) {
return index; // or item.id
}
}
```

Now it returns as a unique identifier for each item so only updated items will be re-rendered.

Break down into small reusable components

This is an extension of the single responsibility principle. Large components are very difficult to debug, manage, and test. If a component becomes large, break it down into more reusable smaller components to reduce duplication of the code, so that we can easily manage, maintain, and debug with less effort.

Angular best practices - parent component

Use Lazy Loading

Try to lazy load the modules in an Angular application whenever possible. Lazy loading loads something only when it is used. This reduces the size of the application load initial time and improves the application boot time by not loading unused modules.

Without lazy loading

// app.routing.ts

```
import { WithoutLazyLoadedComponent } from './without-lazy-loaded.component';
{
path: 'without-lazy-loaded',
component: WithoutLazyLoadedComponent
}
```

With lazy loading

//app.routing.ts

```
{
path: 'lazy-load',
loadChildren: 'lazy-load.module#LazyLoadModule'
}
```

// lazy-load.module.ts

```
import { LazyLoadComponent } from './lazy-load.component';
@NgModule({
imports: [
RouterModule.forChild([
{
path: '',
component: LazyLoadComponent
}
])
],
declarations: [
LazyLoadComponent
]
})
export class LazyModule {
}
```

Use Index.ts

index.ts helps us keep all related things together so that we don't have to be concerned about the source file name. This helps reduce the size of the import statement.

```
For example, we have webapp/src/app/store/actions/index.ts as
export * from './favorites.actions';
export * from './briefcase.actions';
export * from './clipboard.actions';
export * from './content.actions';
export * from './most-recents.actions';
export * from './node.actions';
export * from './repository.actions';
export * from './trashcan.actions';
import { Hero, HeroService } from '../heroes'; // index is implied
```

Avoid logic in templates

All template logic will be extracted into a component. This helps to cover that case in a unit test and reduce bugs when there is a template change.

Logic in templates

```
// template
Status: Unavailable// component
ngOnInit (): void {
this.aliases=resp.value;
}
```

Logic in component

```
// template
Status: Unavailable// component
ngOnInit (): void {
this.aliases=resp.value;
this.isUnavailable = this.aliases === 'inActive' || 'hold';
}
```

Cache API calls

Responses from some API calls do not change frequently. In those cases, we can add a caching mechanism and store the value from an API. When another request to the same API is made, we get a response from the check. If there is no value available in the cache then we make an API call and store the result. Table 1

We can introduce a cache time for some API calls when the value changes infrequently. Caching API calls and avoiding unwanted duplicate API calls improves speed of the application and ensures we do not download the same information repeatedly.

Use async pipe in templates

Observables can be directly used in templates with the async pipe instead of using plain JS values. Under the hood, an observable would be unwrapped into plain value. When a component using the template is destroyed, the observable is automatically unsubscribed.

Without Using async pipe

```
//template
{{ text }}
```

Using async pipe

```
// template
{{ text | async }}
// component
this.text = observable.pipe(
map(value => value.item)
);
```

Declare safe strings

The variable of type string has only some set of values and we can declare the list of possible values as the type. So, the variable will accept only the possible values. We can avoid bugs while writing the code during compile time itself.

Normal String declaration

```
private taskName: string;
// We can assign any string to taskName.
this.taskName = 'Quick task;
```

Safe string declaration

```
private taskName: 'Quick task' | 'Simple editorial';
this. taskName = "Quick task ';
this.taskName = 'Simple editorial';
this. taskName = 'Workflow'; // This will give the below error
Type ""Workflow"" is not assignable to type '"Quick task" | "Simple editorial"'
```

Avoid any type when declaring constants and variables

Declare variables or constants with proper types other than any. This will reduce unintended problems. Another advantage of having good typing in our application is that it makes refactoring easier. If we had a proper typing, we would get a compile-time error as shown below:

```
export interface HotFolder {
id: string;
folderName: string;
enabled: string;
processDefinitionName: string;
params: { [key: string]: string };
}
```

```
export class HotFoldersAddComponent implements OnInit
hotFolder: HotFolder;
constructor() {
}
ngOnInit() {
this.hotFolder = {
id: 12345,
folderName: ' /home/rsuite/test ',
enabled: true,
path: ‘test’
}
}
}
```

// Error

Type ‘{ id: number; foldername: string; path: string; }’ is not assignable to type ‘hotFolder.

Object literal may only specify known properties, and ‘path does not exist in type ‘hotFolder.

State management

One of the most challenging things in software development is state management. State management in Angular helps manage state transitions by storing the state of any form of data. There are several state management libraries for Angular (including NGRX, NGXS, Akita, etc.), all with different uses and purposes.

We can choose suitable state management for our application before we implement it.

Some benefits of using state management.

- **It enables sharing data between different components.**
- **It provides centralized control for state transition.**
- **The code is cleaner and more readable.**
- **The code is easier to debug when something goes wrong.**
- **Dev tools are available for tracing and debugging state management libraries.**

Use CDK Virtual Scroll

Loading hundreds of elements can be slow in any browser, but CDK virtual scroll support displays large lists of elements more efficiently. Virtual scrolling enables a performant way to simulate all items being rendered by making the height of the container element the same as the height of the total number of elements to be rendered, and then only rendering the items in view.

//Template

```
<ul>
<cdk-virtual-scroll-viewport itemSize="100">
<ng-container *cdkVirtualFor="let item of items">
<li> {{item}}</li>
</ng-container>
</cdk-virtual-scroll-viewport>
</ul>
```

// Component

```
export class CdkVirtualScroll {
items = [];
constructor() {
this.items = Array.from({length: 100000}).map((_, i) => `scroll list ${i}`);
}
}
```

Use environment variables

Angular provides environment configurations to declare variables unique for each environment. The default environments are development and production. We can even add more environments or add new variables in existing environment files.

Dev environment

```
// environment.ts environment variables
const v1 = '/api/v1';
const v2 = '/api/v2';
const v3 = '/api/v3';
export const environment: { [key: string]: any } = {
production: false,
isEnterprise: true,
name: 'enterprise',
features: [
'custom-columns',
'custom-menu'
],
navigationPanel: [
'content',
'search',
'favorites',
'mostRecents',
'clipboard',
'briefcase',
'trashcan',
'workflow',
'workflowSearch',
'briefcase',
'task'
],
logo: './assets/logo/rsuite-enterprise.png',
favicon: './assets/favicon.ico',
v3: v3,
v1: v1,
v2: v2
sessionUrl: `${v2}/user/session`
}
```

During the application build, the environment variables are applied automatically from the specified environment file.

Use lint rules for Typescript and SCSS

tslint and stylelint have various inbuilt options that force the program to be cleaner and more consistent. It is widely supported across all modern editors and can be customized with your own lint rules and configurations. This will ensure consistency and readability of the code.

Some inbuilt options in tslint: no-any, no-magic-numbers, no-debugger, no-console, etc.

Some inbuilt options in stylelint: color-no-invalid-hex, selector-max-specificity, function-comma-space-after, declaration-colon-space-after, etc.

Always document the code

Always document the code as much as possible. It will improve readability and help a new developer involved in a project to understand its logic.

It is a good practice to document each variable and method. Define methods using multi-line comments on what task the method performs. All parameters should be explained.

```
/**
* Returns the managed object content children
* @param id - managed object's canonical id
* @param uri - managed object's canonical uri
* @returns Observable the emits the contents of the tree node
*/
browseTreeNode(id: string, uri: string): Observable<ServerResponse> {
return this.appService.browseTreeNode(id, uri);
}
```

Write readable code

If you want refactoring to be as efficient as possible, it is important to have readable code. Readable code is easier to understand, which makes it easy to debug, maintain, and extend.

File names

While adding new files, pay attention to the file names you decide to use. File names should be consistent and describe the feature by dot separation.

```
|-- content-inspect-addedto.component.ts
or
|-- managed-object.service.ts
```

Variable and function names

Name the variables and functions so the next developer understands them. Be descriptive and use meaningful names — clarity over brevity.

This will help us avoid writing functions like this:

```
function div(x, y)) { const val = x / y; return val;}
```

And hopefully more like this:

```
function divide(divident, divisor) {
const quotient = divident / divisor;
return quotient;
}
```

Write small pure functions

When we write functions to execute some business logic, we should keep them small and clean. Small functions are easier to test and maintain. When you start noticing that your function is getting long and cluttered, it’s probably a good idea to abstract some of the logic to a new one.

This avoids functions like this:

```
addOrUpdateData(data: Data, status: boolean) {
if (status) {
return this.http.post<Data>(url, data)
.pipe(this.catchHttpErrors());
}
return this.http.put<Data>(`${url}/${data.id}`, data)
.pipe(this.catchHttpErrors());
}
}
```

And hopefully more like this:

```
removeManagedObject(canonicalId: string): Observable<ServerResponse> {
const formData = new FormData();
formData.append('id', canonicalId);
return this.http.request<ServerResponse>('delete', `${endpoint.contentUrl}`, {
body: formData,
headers: this.jsonHeaders
});
}
```

Remove unused code

It is extremely valuable to know if a piece of code is being used or not. If you let unused code stay, then in the future, it can be hard or almost impossible to be certain if it's actually used or not. Therefore, you should make it a high priority to remove unused code.

Avoid code comments

Although there are cases for comments, you should really try to avoid them. You don't want your comments to compensate for your failure to express the message in your code. Comments should be updated as code is updated, but a better use of time would be to write code that explains itself. Inaccurate comments are worse than no comments at all.

Best practices for Angular unit test cases

- **Make sure unit tests run in an isolated environment so that they run fast.**
- **Make sure we have at least 80% of the code coverage.**
- **When it comes to testing services, always use spies from the jasmine framework for the dependencies this makes testing a lot faster.**
- **When we subscribe to observables while testing, make sure we provide both success and failure callbacks. This will remove unnecessary errors.**
- **When testing components with service dependencies, always use mock services instead of real services.**
- **All the TestBedConfiguration should be placed in beforeEach not to repeat the same code for every test.**
- **While testing components, always access the DOM with debugElement instead of directly calling nativeElement. This is because debugElement provides an abstraction for the underlying runtime environment. This will reduce unnecessary errors.**
- **Prefer By.css instead of queryselector if you are running the app on the server as well. This is because the queryselector works only in the browser. if we are running the app on the server, this will fail. But we have to unwrap the result to get the actual value.**
- **Always use a fixture.detectChanges() instead of ComponentFixtureAutoDetect. ComponentFixtureAutoDetect doesn't know the synchronous update to the components.**
- **Call compileComponents() if we are running the tests in the non-CLI environment.**
- **Use RXJS marble testing whenever we are testing observables.**
- **Use PageObject model for reusable functions across components.**

- **Don’t overuse NO_ERRORS_SCHEMA. This tells angular to ignore the attributes and unrecognized tags, prefer to use component stubs.**
- **Another advantage of using component stubs instead of NO_ERRORS_SCHEMA is that we can interact with these components if necessary. It’s better to use both techniques together**
- **Never do any setup after calling compileComponents(). compileComponents is asynchronous and should be executed in async beforeEach. Maintain the second beforeEach to do the synchronous setup.**

Error handling

Handling errors properly is essential in building a robust application in Angular. Error handlers provide an opportunity to present friendly information to the user and collect important data for development. In today's age of advanced front-end websites, it's more important than ever to have an effective client-side solution for error handling.

An application that does not handle errors gracefully leaves its users confused and frustrated when the app suddenly breaks without explanation. Handling these errors correctly across an application greatly improves user experience. Collected data from the error handling can inform the development team about important issues that slipped past testing. This is why monitoring tools like Rollbar are so important.

In this section, we will compare several solutions for error handling in Angular apps. First we will describe the traditional approaches using ErrorHandler and HttpClient. Then we will show you a better solution using HttpInterceptor. We'll also show you how to use this interceptor to monitor and track errors centrally in https://rollbar.com/error-tracking/angular/ Rollbar.

One traditional way of handling errors in Angular is to provide an ErrorHandler class. This class can be extended to create your own global error handler. This is also a useful way to handle all errors that occur, but is mostly useful for tracking error logs. For reference, you can check our tutorial on how to use HYPERLINK "https://rollbar.com/blog/client-side-angular-error-handling/" ErrorHandler in Angular 2+.

By implementing error handling in HttpClient or HttpInterceptor, you can work directly with all HTTP requests in your application, providing you with the ability to transform the request, retry it, and more. ErrorHandler is useful for more generic error handling, however, HttpInterceptor provides a much more robust way to handle errors related to the server and network.

Handling Errors with HttpClient

By using Angular's HttpClient along with catchError from RxJS, we can easily write a function to handle errors within each service. HttpClient will also conveniently parse JSON responses and return a JavaScript object in the observable. There are two categories of errors which need to be handled differently:

Client-side: Network problems and front-end code errors. With HttpClient, these errors return ErrorEvent instances. Server-side: AJAX errors, user errors, back-end code errors, database errors, file system errors. With HttpClient, these errors return HTTP Error Responses.

By verifying if an error is an instance of ErrorEvent, we can figure out which type of error we have and handle it accordingly.

This is a good solution for just one service, but a real app contains numerous services that can all potentially throw errors. Unfortunately, this solution requires copying the handleError function across all services, which is a very serious anti-pattern in Angular development. If the way we handle errors needs to change, then we have to update every single handleError function across every service. This is counter-productive and can easily lead to more bugs. We need an efficient way to handle errors globally across the entire application. Fortunately, Angular supports this using HttpInterceptor.

Here is an example service which uses this basic form of error handling:

```
import { HttpClient } from '@angular/common/http';
@Injectable()
export class PublishService {
constructor( private http: HttpClient ) {
}
publish(data: any): Observable<ServerResponse> {
const formData = new FormData();
Object.keys(data).forEach(key => formData.append(key, data[key]));
return this.http.post<ServerResponse>(`${endpoint.publishUrl}`, formData).
pipe(catchError(this.handleError));
}
handleError(error){
return throwError(error.message || "server Error"); }
```

While ErrorHandler is a useful way to handle errors across an app, HttpInterceptor provides a much more robust solution for handling server and connection-related errors giving the ability to retry or return a richer error response to the client.

Handling Errors with HttpInterceptor

HttpInterceptor provides a way to intercept HTTP requests and responses to transform or handle them before passing them along.

```
import { Injectable } from '@angular/core';
import {
HttpEvent ,  HttpRequest ,  HttpHandler ,
HttpInterceptor ,  HttpErrorResponse
} from '@angular/common/http';
import { Observable,  throwError  } from ' rxjs ';
import { retry,  catchError  } from ' rxjs /operators';
@Injectable()
export class  ServerErrorInterceptor  implements  HttpInterceptor  {
intercept(request:  HttpRequest <any>, next:  HttpHandler ): Observable< HttpEvent <any>> {
return  next.handle (request).pipe(
retry(1),
catchError ((error:  HttpErrorResponse ) => {
if ( error.status  === 401) {
// refresh token
} else {
return  throwError (error);
}
})
);
}
}
```

Heaging 3

HttpInterceptor provides a way to intercept HTTP requests and responses to transform or handle them before passing them along.

Part1_Heaging 3

HttpInterceptor provides a way to intercept HTTP requests and responses to transform or handle them before passing them along.

Part2_Heaging 3

HttpInterceptor provides a way to intercept HTTP requests and responses to transform or handle them before passing them along.

Note:

Note: This is note Para