



**UTICA – Frontend Development**

**Standards**

Contents

[1 Frameworks / Libraries / Plugins 3](#_Toc535425797)

[2 Coding & Naming Conventions 4](#_Toc535425798)

[2.1 Using const instead of let 4](#_Toc535425799)

[2.2 File structure conventions 4](#_Toc535425800)

[2.3 Separate file names with dots and dashes 4](#_Toc535425801)

[2.4 Component Selectors 5](#_Toc535425802)

[2.5 Angular NgModule names 5](#_Toc535425803)

[2.6 Using Upper Camel cases 6](#_Toc535425804)

[2.7 Constants 6](#_Toc535425805)

[2.8 Using Sass pre-processor 6](#_Toc535425806)

[3 Programming Practices 7](#_Toc535425807)

[3.1 Rule of One 7](#_Toc535425808)

[3.2 Use of Reactive Forms 9](#_Toc535425809)

[3.3 Use of dependency injection 9](#_Toc535425810)

[3.4 Using shared or core modules 9](#_Toc535425811)

[4 Error handling 11](#_Toc535425812)

[5 References 12](#_Toc535425813)

# Frameworks / Libraries / Plugins

Following are the list of frameworks & libraries that would be used as a part of the development:

**1. Node JS**

Node.js is an open-source, cross-platform JavaScript run-time environment that executes JavaScript code outside of a browser.

**2. npm**

The Angular Framework, Angular CLI, and components used by Angular applications are packaged as [npm packages](https://docs.npmjs.com/getting-started/what-is-npm" \o "What is npm?) and distributed via the [npm registry](https://docs.npmjs.com/).

You can download and install these npm packages by using the [npm CLI client](https://docs.npmjs.com/cli/install), which is installed with and runs as a [Node.js®](https://nodejs.org/) application. By default, the Angular CLI uses the npm client.

**3. Bootstrap**

Bootstrap is a free and open-source front-end framework for developing websites.

**4. Node - sass**

Node-sass is a library allows you to natively compile .scss files to css.

# Coding & Naming Conventions

Following are the general coding guidelines that we have followed.

## Using const instead of let

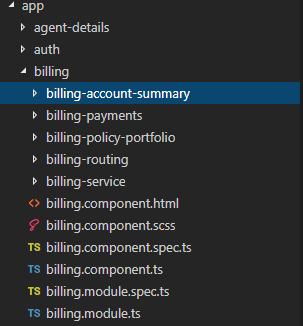
When declaring variables, use **const** when the value is not going to be reassigned. Using **let** and **const** appropriately to make the intensions of declarations cleaner.

## File structure conventions

Angular prefers "package by feature". In package-by-feature, the package names correspond to important, high-level aspects of the problem domain.

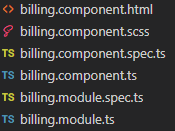
For example, in billing feature we have the following packages.

* Billing-account-summary
* Billing-payments
* Billing-policy-portfolio
* Billing-routing
* Billing-service



## Separate file names with dots and dashes

* Do use dashes to separate words in the descriptive name
* Do use dots to separate the descriptive name from the type
* Do use consistent type names for all components following a pattern that describes the component's feature then its type. A recommended pattern is feature.type.ts



## Component Selectors

* Do use dashed-case or kebab-case for naming the element selectors of components.

**Bad practice**

@Component({

selector: 'tohHeroButton',

templateUrl: './hero-button.component.html'

})

export class HeroButtonComponent {}

**Good practice**

@Component({

selector: 'toh-hero-button',

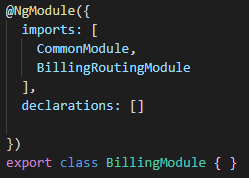
templateUrl: './hero-button.component.html'

})

export class HeroButtonComponent {}

## Angular NgModule names

* Do append the symbol name with the suffix Module
* Do give the file name the .module.ts extension
* Do name the module after the feature and folder it resides in



## Using Upper Camel cases

* Do use upper camel case when naming classes

**Bad practice**

export class exceptionService {

constructor() { }

}

**Good practice**

export class ExceptionService {

constructor() { }

}

## Constants

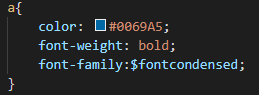
* Do declare variables with **const** if their values should not change during the application lifetime.

## Using Sass pre-processor

* Sass is a styles pre-processor which brings support for fancy things like variables, functions, mixins.

Use variables in scss:





# Programming Practices

Angular is a platform and framework for building client applications in HTML and TypeScript. Angular is written in TypeScript. It implements core and optional functionality as a set of TypeScript libraries that you import into your apps.

Below are some of the good programming practices to be followed for developing Angular applications using TypeScript.

## Rule of One

This rule states that one must define one thing, such as a service or component per file. One should consider limiting files to maximum 400 lines of code. This is done because:

* One component per file makes it far easier to read, maintain, and avoid collisions with teams in source control.
* One component per file avoids hidden bugs that often arise when combining components in a file where they may share variables, create unwanted closures, or unwanted coupling with dependencies.
* A single component can be the default export for its file which facilitates lazy loading with the router.

**Bad practice**

/\* avoid \*/

import { Component, NgModule, OnInit } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

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

class Hero {

id: number;

name: string;

}

@Component({

selector: 'my-app',

template: `

<h1>{{title}}</h1>

<pre>{{heroes | json}}</pre>

`,

styleUrls: ['app/app.component.css']

})

class AppComponent implements OnInit {

title = 'Tour of Heroes';

heroes: Hero[] = [];

ngOnInit() {

getHeroes().then(heroes => (this.heroes = heroes));

}

}

@NgModule({

imports: [BrowserModule],

declarations: [AppComponent],

exports: [AppComponent],

bootstrap: [AppComponent]

})

export class AppModule {}

platformBrowserDynamic().bootstrapModule(AppModule);

const HEROES: Hero[] = [

{ id: 1, name: 'Bombasto' },

{ id: 2, name: 'Tornado' },

{ id: 3, name: 'Magneta' }

];

function getHeroes(): Promise<Hero[]> {

return Promise.resolve(HEROES); // TODO: get hero data from the server;

}

It is a better practice to redistribute the component and its supporting classes into their own, dedicated files. As the app grows, this rule becomes even more important.

**Good practice**

1. **app.module.ts**

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

import { BrowserModule } from '@angular/platform-browser';

import { RouterModule } from '@angular/router';

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

import { HeroesComponent } from './heroes/heroes.component';

@NgModule({

imports: [

BrowserModule,

],

declarations: [

AppComponent,

HeroesComponent

],

exports: [ AppComponent ],

bootstrap: [ AppComponent ]

})

export class AppModule { }

1. **app.component.ts**

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

import { HeroService } from './heroes';

@Component({

selector: 'toh-app',

template: `

<toh-heroes></toh-heroes>

`,

styleUrls: ['./app.component.css'],

providers: [HeroService]

})

export class AppComponent {}

## Use of Reactive Forms

Reactive forms provide a model-driven approach to handling form inputs whose values change over time. **Reactive forms** are more robust: they're more scalable, reusable, and testable.

## Use of dependency injection

Dependency injection (DI), is an important application design pattern. Angular has its own DI framework, which is typically used in the design of Angular applications to increase their efficiency and modularity.

@Injectable({

providedIn: 'root'

})

export class BillingPolicydocumentResolverService implements Resolve<any> {

## Using shared or core modules

Splitting the app into core and shared modules will help make it reusable and modular. Core module should be created of components (i.e. header, main navigation, footer) that will be used across the entire app.

Shared module can have components, directives and pipes that will be shared across multiple modules and components, but not the entire app necessarily.

For example, we have a created a header component that will be common across all modules.

@Component({

selector: "app-header",

templateUrl: "./header.component.html",

styleUrls: ["./header.component.scss"],

providers: [DateSuffix]

})

export class HeaderComponent implements OnInit,OnDestroy {

# Error handling

Error handling for network calls have been implemented in project.

* Success/Error callbacks - A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete network calls.

this.genservice.getPersonalizeData("billingfaqs").toPromise().then(data => {

//success code goes here

}, error => {

this.spinner.hide();

if (error.status === 407) {

//error handled

else {

this.sharedService.ErrorPopTrigger(Constants.TOPFIVEFAQ, false);

}

});

# References

* <https://angular.io/guide/styleguide>
* <https://angular.io/guide/reactive-forms>