TAS CORE Development Standards

-- DRAFT --

# Who it’s for

This Developer Manual is to provide software developers with information necessary to modify and add-to the Medical Care Collection Fund System (MCCF).

MCCF is a web-based system built on the Angular 2(Version 4) framework and Node.js. Angular is a TypeScript-based open-source front-end web application platform for building complex web applications. Node.js is a JavaScript run-time environment for executing JavaScript code server-side.

# Skills Required

Git Version Control System

Software Development

JavaScript (version 1.6)

Angular 2 (version 4) TypeScript

Node.js v6

CSS

HTML

# Useful Links

<https://angular.io>

<https://nodejs.org>

# Development Setup

The developer should have access to the internet and have installed git and NPM clients. Also, using a good code editor, or IDE, tool is advisable.

## Install

The code base should be copied into the developer’s local environment. Usually this is done with **git clone** from a git repository.

**git clone [repository URL]**

**Next install the required support libraries from NPM.**

**npm install**

## Running

Software developers should run the built-in webserver and open a browser to see changes to code immediately.

Start the development environment.

**npm start**

When a TypeScript file is edited, the Angular development environment automatically sees the change and updated the local webpage. Open a browser (Chrome, Internet Explorer) and enter the following URL:

**http://localhost:4200/**

## Unit Testing

Automated unit testing is very important to help assure nothing was broken after changes are made to the system.

Run the automated unit tests after modifying the system.

**npm test**

# Directory Structure

**/mccf\_tas\_core  
/src  
 /app  
 /assets  
 /environments  
 index.html  
 polyfills.ts  
 test.ts  
 tsconfig.spec.json  
 favicon.ico  
 main.ts  
 styles.css  
 tsconfig.app.json  
 typings.d.ts  
/scripts  
.angular-cli.json  
.bowerrc  
.jazzignore  
inventory  
protractor.conf.js  
tsconfig.json  
.jshintrc  
Jenkinsfile  
README.md  
tslint.json  
.travis.yml  
/karma  
requirements.yml**

## Application Directory Structure

**app/**

**app.component.css**

**app.component.html**

**app.component.spec.ts**

**app.component.ts**

**app.module.ts**

**app.routing.ts**

**core/**

**global.error-handler.ts**

**mccf-common/**

**mccf-eadmin/**

**mccf-ebilling/**

**mccf-einsurance/**

**mccf-epayment/**

**mccf-epharmacy/**

**mccf-non-mccf/**

**mccf-npi/**

**mccf-search/**

**mccf-system/**

## Where to put stuff

The Angular framework uses modules to organize components of the application (app). The main modules are directories under the **app** directory. Below are the general rules for where components should be placed.

|  |  |
| --- | --- |
| Module | Component is |
| Core | * always loaded * vital functions of app * used throughout the app |
| Mccf-common | * not loaded immediately * used by many other modules |
| Mccf-[product name] | * used by specific product * not loaded immediately |
| Mccf-system | * not loaded immediately * used by app admin only |
| Mccf-search | * search functions * loaded if needed |
|  |  |

When developing components, the developer should consider when the functionality is required and where it is used. To reduce browser memory usage and make the system faster, Components and Modules should use Angular “lazy loading” and not be loaded until they are needed by the app.

# Adding NPM module

Most of the time, it is better to use a module that is already developed when it has the desired functionality and has been approved by the VA’s TRM process. The NPM repository has many useful modules. Installing modules require different steps, so developers should read the install instructions carefully.

Install a NPM module (generally)

**npm install –save-exact [module name]**

Make sure the new line in package.json does not contain the “^” character, telling NPM to update with higher versions.

Note: When a library is added, the file package-lock.json is updated and should be saved.

# Adding/Modifying CSS

Be aware that the CSS code used to control a component’s html style can be located in different places. CSS can be put in same directory as a component if only that component uses its CSS elements. CSS code can also be in the parent app component **app.component.css**, which is used by all components in the app. And finally, a CSS file can be placed in the **src/assets** directory.

# Adding/Modifying JavaScript

JavaScript code should be placed in a file inside the **src/assets** directory. The JavaScript file needs to be registered for deployment. To register a JavaScript file, add its path to the **scripts** property in **angular-cli.json**.

Examples of registered JavaScript files:

**"scripts": [**

**"assets/js/datetimeclock.js",**

**"assets/uswds/js/uswds.min.js",**

**"assets/js/switch-sub.js"**

**],**

# Form Components

Form Components should extend MccfFormComponent, located in **mccf-common/mccf-form/mccf-form.component.ts.** This parent component has functions used by all forms pages (see Adding Tooltips).

## Adding Tooltips

Tooltip text is configurable content. This means HTML element tooltips are not hard-coded in the HTML but loaded when a mouse hovers over an element. To add tooltips to an element with the **title** property, add the following code inside the element:

**(mouseover)="tooltip('**[tooltip\_key]**',$event)"**

Note that [tooltip\_key] is the ID in the database for this tooltip. Example: **(mouseover)="tooltip('search',$event)"**

# Notifications

“Toaster pop-up messages” or Growl messages are useful for giving notifications for events to app users. Follow the below steps to add a notification to a component.

1. Import Primeng’s message service in component   
   import {MessageService} from "primeng/components/common/messageservice"
2. Inject message service into component  
   constructor(private messageService: MessageService) {}
3. Call message service in code where an event occurs  
   this.messageService.addAll([  
    {severity: 'success', summary: 'Success', detail: 'Success Message'},  
    {severity: 'warn', summary: 'Warning', detail: 'warning message'},  
    {severity: 'error', summary: 'Error', detail: 'There is an error'}  
   ])

# Unit Tests

Components should always have associated unit tests. Unit tests help assure that a component continues to work as designed even after changes. Unit tests are kept in files that end with “spec.ts”.

The project uses Angular’s advanced unit test features, also called “Component Testing”. This is a step up from “Unit Testing” because it allows for loading component HTML pages, setting variables, and simulating a database service call.

## Accessibility Compliance

All components with HTML must run the Axe-core library against the component HTML to check for basic accessibility issues. Initializing and configuring the component for testing can be complicated. The best way to setup an accessibility test is to study and use (copy) one of the many existing unit tests that check for accessibility using the axe-core library.

# UI Widget Library

PrimeNG is a collection of rich UI components for Angular. When complex widgets are required such as graphs and image viewers, PrimeNG elements should be considered.

[www.primefaces.org/primeng](http://www.primefaces.org/primeng)

Keep in mind that a simple system is often more robust; so, for common elements like buttons and forms, stick with the less complicated HTML and USWDS CSS elements.

# Encryption

crypto-js is the javascript library used for encryption. There are three steps to using it.

1. Import the library into the component that requires encryption.   
   import \* as CryptoJS from 'crypto-js'
2. Declare two variables for the key and initialization vector.  
   // Encrypt with Base64  
   key = CryptoJS.enc.Base64.parse("#base64Key#")  
   iv = CryptoJS.enc.Base64.parse("#base64IV#")
3. And encrypt.  
   this.crypto\_password = CryptoJS.AES.encrypt(password\_from\_ui, this.key, {iv: this.iv}).toString()