TAS Core Development Standards



–

July 2018

Department of Veterans Affairs

Office of Information and Technology (OI&T)

### Revision History

| **Date** | **Revision** | **Description** | **Author** |
| --- | --- | --- | --- |
| 5/15/2018 | 0.01 | Add Tableau section | Patrick Whalen - Halfaker |
| 7/30/2018 | 0.0.2 | Add API comments | David Betz |
| 8/17/2018 | 0.0.3 | Add Treeview section | Patrick Whalen |
| 11/7/2018 | 0.0.4 | Add Datatable section | Patrick Whalen |
|  |  |  |  |

Table of Contents

[Revision History 2](#_Toc521405478)

[Who it’s for 5](#_Toc521405479)

[Skills Required 5](#_Toc521405480)

[Useful Links 5](#_Toc521405481)

[Development Setup 5](#_Toc521405482)

[Install 5](#_Toc521405483)

[Running 6](#_Toc521405484)

[Unit Testing 6](#_Toc521405485)

[Directory Structure 6](#_Toc521405486)

[Application Directory Structure 7](#_Toc521405487)

[Where to Put Stuff 8](#_Toc521405488)

[Module 8](#_Toc521405489)

[Component is 8](#_Toc521405490)

[API running 8](#_Toc521405491)

[API testing 8](#_Toc521405492)

[API directory structure 9](#_Toc521405493)

[API programming model 9](#_Toc521405494)

[Environment Variables 10](#_Toc521405495)

[Adding NPM module 10](#_Toc521405496)

[Adding/Modifying CSS 11](#_Toc521405497)

[Adding/Modifying JavaScript 11](#_Toc521405498)

[Form Components 11](#_Toc521405499)

[Adding Tooltips 11](#_Toc521405500)

[Adding Breadcrumbs 12](#_Toc521405501)

[Notifications 12](#_Toc521405502)

[Unit Tests 13](#_Toc521405503)

[Accessibility Compliance 13](#_Toc521405504)

[UI Widget Library 13](#_Toc521405505)

[Encryption 13](#_Toc521405506)

[Tableau API 14](#_Toc521405507)

# Who it’s for

This Developer Manual provides software developers the information necessary to modify and add-to the Medical Care Collection Fund System (MCCF) TAS application.

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 v8

CSS

HTML

# Useful Links

<https://angular.io>

<https://nodejs.org>

# Development Setup

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

## Install

The code base must 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 immediately see changes to code.

Start the development environment.

**npm start**

When a TypeScript file is edited, the Angular development environment automatically detects the change and updates 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 is 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 describing 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 each component’s functionality is required and where it is used. To reduce browser memory usage and to make the system faster, components and Modules should use Angular “lazy loading” and should not be loaded until they are needed by the app.

## API running

Reference: Installing in TAS API Readme

TAS API functionality is designed to be run via Docker in both development and production.

docker build . -f Dockerfile.dev -t tas/node8

To setup prerequisites for the TAS API, run the following from the ./src folder:

docker run -v $PWD:/var/app tas/node8 npm install

Run with:

docker run -it -e "DEBUG=api,elastic,azure,fs,core,memory,webclient,errors" -v $PWD:/var/app -p 3000:3000 tas/node8

## API testing

Reference: Tests in TAS API Readme

Run tests with mocha:

docker run -it -v $PWD:/var/app tas/node8 mocha

## API directory structure

Reference: Structural Overview of APIs with TAS API in TAS API Readme

The internal mechanics of this project are contained in the tas/ project. Do not edit files in this folder.

Product development is done in the app/ and test/ folders

* app/
  + example/bindings/
    - azure.js
    - memory.js
    - mongodb.js
  + example/namespaces/
    - data.js
    - modular.js
    - v1.js
  + example/endpoint.js
* config.js
* endpoints.js
  + test/

## API programming model

The primary way to work with the TAS API is via commands. These abstract much of the underlying plumbing for both HTTP endpoints and AMQP. They're your primary way of creating functionality with the TAS API.

Refer to the data file of the example endpoint for a full in-line command example (a better, more modular approach is discussed later).

Consider the following snippet from increment.js:

class extends $tas.command.Command {

async run(securityContext, values, adapter) {

return adapter.createResponse($tas.constants.status.SUCCESS, values.number + 1)

}

}

Consider also using this command in an endpoint namespace:

const commands = new $tas.command.Registry()

//+ pre-defined, general use commands

const { IncrementCommand } = require('../commands')

commands

.start('increment')

.next(new IncrementCommand())

.end()

This is an example of a predefined command registered to a namespace on an endpoint. This would be accessed via the following path:

/api/core/example/data/increment

See the TAS API Readme for details.

# Environment Variables

Environment variables are global application-level variables that are specific to each server. For instance, one web server might point to a back-end API server URL while another web server may point to a different back-end API server URL. Because of MCCF deployment requirements, environment variables can be changed without a rebuild. For this reason they are stored in the /assets/config folder, which is not bundled and can be seen/edited on the webserver at any time.

The environment variables are located in:

**/assets/config/environment.json**

Environment variables include:

**production: Boolean, hides development pages**

**coreAPI: string, business server URL**

# 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 using the VA’s TRM process. The NPM repository contains many useful modules. Installing modules requires 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 in different places. CSS can be in the 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. 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 are 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)"

## Adding Breadcrumbs

Breadcrumbs show the navigation path and allow the user to return to previously visited pages. Follow these steps to add breadcrumbs to a page.

1. Add the Breadcrumb Service to the home page component. Call the reset() function in the ngOnInit. This only needs to be done once.

ngOnInit() {  
 this.breadcrumbService.reset()  
}

2. Add HTML tag to the top of the page.

<app-tas-breadcrumb></app-tas-breadcrumb>

3. Add the label to the path in the module routing.ts file. Inside the data key, add the breadcrumb key with value, which is the label that will be shown on the page.

{ path: 'home',  
 component: EbillingHomeComponent,  
 data: { breadcrumb: "eBilling Home" }  
}

## Treeview Widget

A tree view widget presents a hierarchical list. Items in the hierarchy can have child items, and items that have children may be expanded or collapsed to show or hide the children.

Adding Treeview to component page:

1. Add treeview element into HTML page.

<tas-treeview [config]="config” [items]="items" (filterChange)="onFilterChange($event)" (selectedChange)="values = $event">

</tas-treeview>

2. Include treeview libraries.

import { TreeviewItem } from '../../mccf-common/treeview/treeview-item'

import { TreeviewConfig } from '../../mccf-common/treeview/treeview-config'

3. Define minimal required variables and functions in component.

values: number[];  
items: TreeviewItem[];  
config: TreeviewConfig;  
this.config = TreeviewConfig.create({  
 showCheckBox: true,  
 hasAllCheckBox: true,  
 hasFilter: false,  
 hasCollapseExpand: false,  
 decoupleChildFromParent: false,  
 maxHeight: 1200  
 });

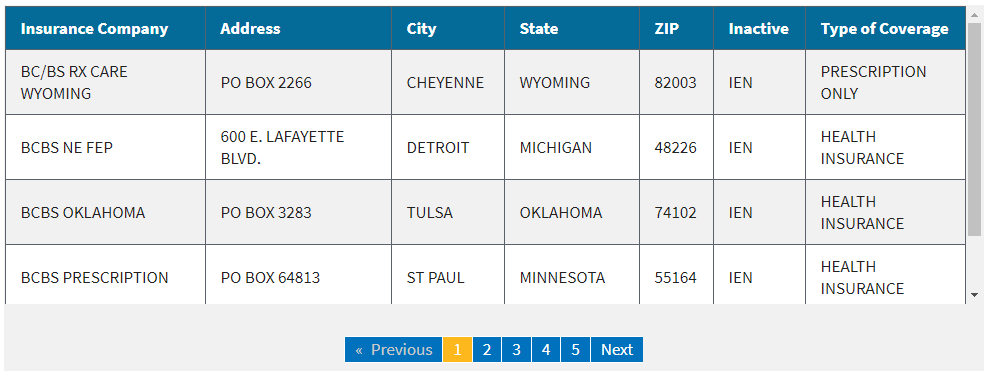
4. Add data inside ngOnInit function or after page loaded. Notice the JSON text-value structure for data along with a “children” field. The second TreeviewItem Boolean parameter tells the component to show a checkbox.

const states = new TreeviewItem({  
 text: 'States', value: 'States', collapsed: true, children: [  
 {  
 text: 'Texas', value: 'Texas', children: [{  
 text: 'DIV1', value: 'DIV1', children: [  
 { text: 'DIV2', value: 'DIV2' },  
 { text: 'DIV3', value: 'DIV3' }  
 ]  
 }]  
 }  
 ]}, true);

this.items = [states];

## Data Table

TAS Core Data Table allows the app to present large sets of data in a lightweight and 508 compliant widget. It contains basic functionality such as pagination, row selection, and column sort.



Adding Data-table into HTML:

1. Add selector item into HTML.

<tas-data-table (rowClicked)="rowClicked($event)"

[config]="myConfig"

[data]="myData">

</tas-data-table>

2. Include data-table libraries in component

import { DataTableConfig } from '../../mccf-common/data-table/data-table-config'

import { DataTableData } from '../../mccf-common/data-table/data-table-data'

3. Add bound variables and functions in component.

myConfig = new DataTableConfig()

myData = new DataTableData()

4. Load data inside ngOnInit function or after page loaded. Data-table headers are defined in a DataTableConfig variable called “headers”, which is an array of string. The rows are defined in DataTableData variable called “row”, which is an array of string arrays. Assure proper JSON text-value structure.

this.myConfig.headers = ["Insurance Company", "Address", "City", "State", "ZIP", "Inactive", "Type of Coverage"]

this.myData.row = [

["BC\/BS RX CARE WYOMING", "PO BOX 2266", "CHEYENNE", "WYOMING", "82003", "IEN", "PRESCRIPTION ONLY"],

["BCBS NE FEP", "600 E. LAFAYETTE BLVD.", "DETROIT", "MICHIGAN", "48226", "IEN", "HEALTH INSURANCE"],

["ZZBLUE CROSS\/BS NE", "MAIN PO STATION\/PO BOX 3248", "OMAHA", "NEBRASKA", "56890", "IEN", "HEALTH INSURANCE"],

["ZZBLUE CROSS\/BS GA", "2357 WARM SPRING RD", "COLUMBUS", "GEORGIA", "71703", "IEN", "HEALTH INSURANCE"]

]

# Notifications

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

[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()

# Tableau API

Tableau is a powerful tool for creating interactive, rich-content reports. The TAS App can include the JavaScript-based Tableau report client into TAS App pages. Reports developed using Tableau report creation application can be shown simple by calling them from TAS App components. To include the Tableau client into a page, follow these steps:

1. Add the div element with ID to the html page code. The code will inject the report into this div element.  
   <div id="tableauViz"></div>
2. Declare a variable of type any named “tableau” in the component code file. Put this below the import and above the component (see below example). This will be set by the Tableau global script.  
   declare var tableau: any;
3. Declare a component variable of type any named “tableauViz” for callback info.  
   tableauViz: any
4. Inside ngOnInit function, add code to initialize and add report. Change url, options, and parameters as required and for the report. The url can be found on a Tableau report share link.

let placeholderDiv = document.getElementById('tableauViz');

let url = 'https://public.tableau.com/views/USTreasuryInterestRate/Sheet1?:embed=y&:display\_count=yes';  
let options = {  
 hideTabs: true,  
 width: '800px',  
 height: '700px',  
 onFirstInteractive: function() {  
 // The viz is now ready and can be safely used.  
 }  
 };  
 this.tableauViz = new tableau.Viz(placeholderDiv, url, options)

Example code changes for adding a Tableau client to angular component

