**SNIATM Swordfish Basic Web Client Developer Guide**

|  |  |  |
| --- | --- | --- |
| **Author** | **Date** | **Version** |
| **Sravanthi kalluri** | **26/04/2018** | **0.1** |
|  | **6/6/2018** | **0.2** |
|  | **07/06/2018** | **0.2.1** |

Contents

[1. Introduction of SNIA Basic Web Client 2](#_Toc516124076)

[2. Overview 2](#_Toc516124077)

[2.1 CORS (Cross-origin resource sharing) 2](#_Toc516124078)

[2.2 URI Request and Response model 2](#_Toc516124079)

[2.3 Implications of Node 3](#_Toc516124080)

[2.4 Express 3](#_Toc516124081)

[2.5 Angular-cli 4](#_Toc516124082)

[2.6 overview of Redfish/Swordfish schema usage 4](#_Toc516124083)

[2.7 Redfish/Swordfish schema usage using web client 5](#_Toc516124084)

[3. Project Structure of Swordfish Basic Web Client 7](#_Toc516124085)

[4. Data Flow 11](#_Toc516124086)

[4.2 Views folder 11](#_Toc516124087)

[5. Components Used for Swordfish Basic Web Client 12](#_Toc516124088)

# Overview of SNIA Basic Web Client

The Swordfish Basic Web Client gives a clear demonstration of collections and its sub collections data in a blade interface.

A detailed view of real time system’s data is organized in a hierarchical manner showing the kind of relation (members, links, properties etc) between them using blade interface design.

The Swordfish Basic Web Client can connect to one or more Swordfish services (including the Swordfish emulator, redfish emulator), and present in a web UI frame the entire Swordfish hierarchy. The basic web client also provides basic capabilities to modify configurable Properties (as specified in the schema), as well as providing a basic interface to add or remove elements from the service through a Web UI.

# Technology Stack

## 2.1 CORS (Cross-origin resource sharing)

* It is a mechanism that allows restricted resources (e.g. fonts) on a web page to be requested from another domain outside the domain from which the first resource was served.
* A web page may freely embed cross-origin images, stylesheets, scripts, frames, and videos.
* Certain "cross-domain" requests, notably Ajax requests, are forbidden by default by the same-origin security policy.
* For Ajax and HTTP request methods that can modify data (usually HTTP methods other than GET, or for POST usage with certain MIME types), the specification mandates that browsers "preflight" the request, soliciting supported methods from the server with an HTTP OPTIONS request method, and then, upon "approval" from the server, sending the actual request with the actual HTTP request method.
* Servers can also notify clients whether "credentials" (including Cookies and HTTP Authentication data) should be sent with requests

## URI Request and Response model

* The Web Request and Web Response classes contain methods and properties that apply generically across the different supported protocols and enable applications to download and upload data given a specific URI and to specify authentication details, proxy information, and content details such as the type and size of the content.
* An asynchronous pattern is also provided for any operation that could be blocking on network activity.
* In angular, Request instances are typically created by higher-level classes, like Http and Jsonp, but it may occasionally be useful to explicitly create Request instance

**Ex**: this.http.request(new Request ({

method: RequestMethod.Get, // could be POST,PUT,DELETE

url: URL, // Endpoint to communicate to server

body:{} // optional

}));

* **Response Model :** Though this object isn't usually instantiated by end-users, it is the primary object interacted with when it comes time to add data to a view

## Implications of Node

* Node (or more formally Node.js) is an open-source, cross-platform, runtime environment that allows developers to create all kinds of server-side tools and applications in JavaScript.
* The runtime is intended for use outside of a browser context (i.e. running directly on a computer or server OS).
* As such, the environment omits browser-specific JavaScript APIs and adds support for more traditional OS APIs including HTTP and file system libraries.
* The node package manager (NPM) provides access to hundreds of thousands of reusable packages.

## Express

* Express.js, a Sinatra-inspired web development framework for Node.js, and the de-facto standard for the majority of Node.js applications out there today.
* Express is the most popular Node web framework, and is the underlying library for a number of other popular Node web frameworks. It provides mechanisms to:

1. Write handlers for requests with different HTTP verbs at different URL paths (routes).
2. Integrate with "view" rendering engines in order to generate responses by inserting data into templates.
3. Set common web application settings like the port to use for connecting, and the location of templates that are used for rendering the response.
4. Add additional request processing "middleware" at any point within the request handling pipeline.

## Angular-cli

* There are two main ways to compile and serve an Angular 2 applications and that is via web pack or systemjs.
* For Swordfish Basic Web Client web pack is used
* Any dependency added in the package.json need to be installed
* run npm install to install the package dependencies

1. **tsconfig.json :** tells compiler how we want to compile TypeScript files
2. the two most important properties and those are

{

“target”: “ES5” ,

“module”: “commonjs”

}

1. Here we are setting ECMAScript version to ES5 and modules that we want to generate should be in the commonjs format.
2. **package.json :** tells how to start the applications
3. **Bootstrap :** To tell the compiler which module and component must be run first, we need to specify the bootstrap component
4. To Bootstrap the module, include it in main.ts file
5. To bootstrap our application, need to include the necessary resources into our index.html file

Ex: <body>

<app-home>loading….</app-home>

</body>

1. Here, Angular has instantiated AppHome Component and loaded its template into the app-home element.
2. Inside app.module.ts, inject the AppHome Component into bootstrap section

## Overview of Redfish/Swordfish schema usage

* The Redfish/Swordfish scheme (can be of any version, currently using static and dynamic(**v1.0.6**) can be split into various sections based on the properties they possess in XML Schema definitions
* In Swordfish basic web client, based on the navigation property from XML Schema definitions, links are enabled to click
* The response model is segregated into 4 entities

1. Object-Object Entity

2. Object-Array

3. Array-Object

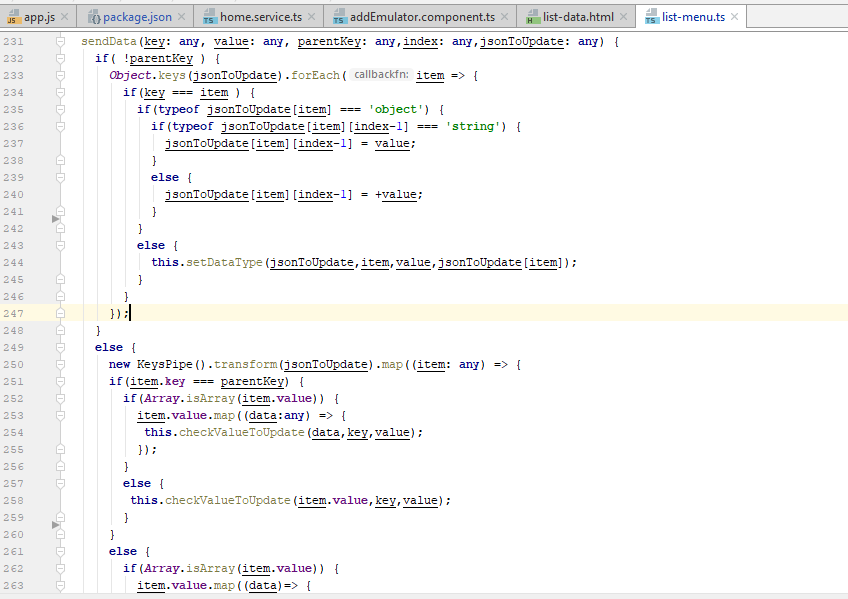
4. Array-Array

* The resultant data after segregation is shown to user under properties, OData and links sections
* Properties can be editable if the read-only property is set to false in the schema definitions
* User can extend the collection in the service by adding a new collection to the existing one using add collection functionality

## Redfish/Swordfish schema usage using web client

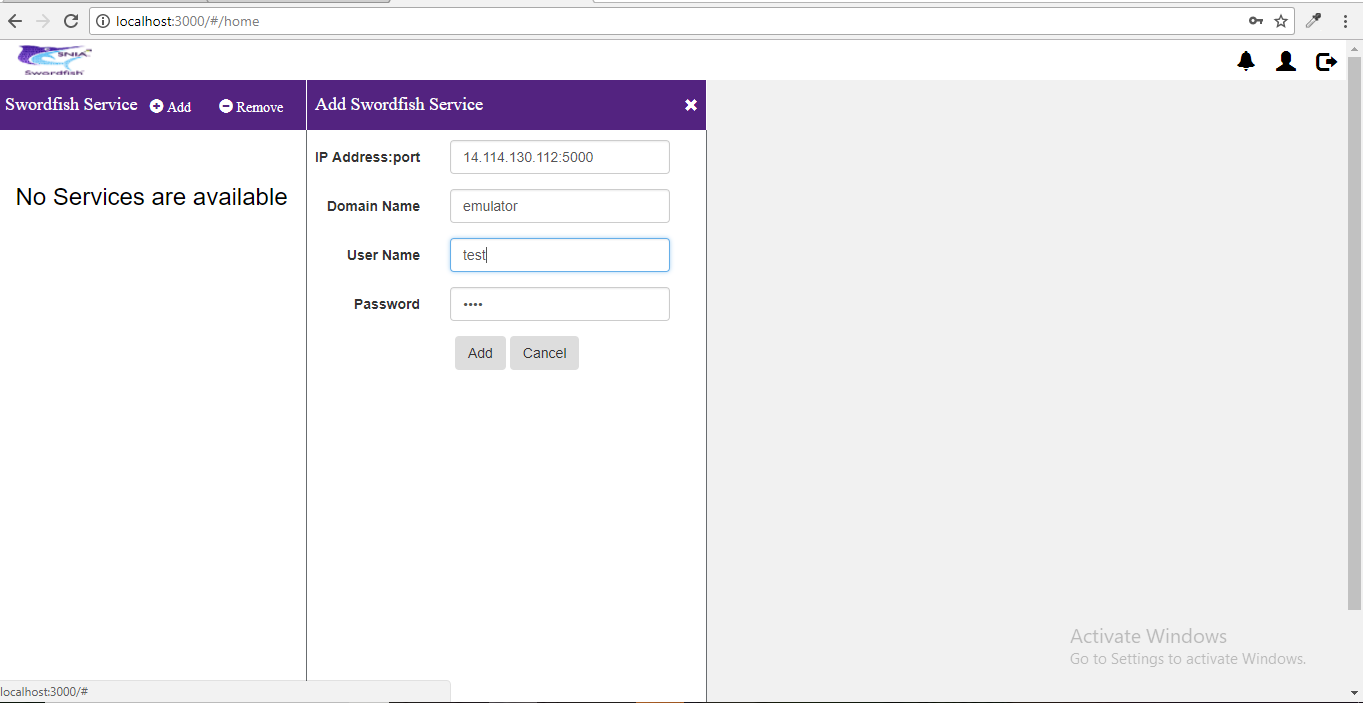
1. **Support to new property**: The newly added property is appended as a part of request body to the service that talks with the schema

Below code block is implemented in such a way that it appends the new property to the existing collection properties



1. Basic web client is designed and implemented in such a way that it supports for both static and dynamic schemas.
2. A complete IP Address is fetched and framed to make a service call to Schema to retrieve the information

PS: Below image allows user to provide mandatory fields data that are used to talk with the schema



1. Programmatical code:

add-emulator.component.html:



1. Web client has a feasibility to add multiple services in a single session.



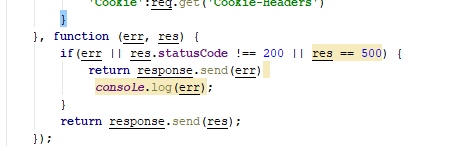
Following methods in [home.service.ts](http://home.service.ts) service allows user to add multiple services by storing the IP address of respective service in local storage.

1. When a web client tries to talk to any schema it encounters with 2 scenarios’

1. Success: Schema replies with a 200-status code along with the schema body as apart of response to the web client

2. Error: Schema replies with a status code of 500 if the request made by the web client is inappropriate

1. Error responses may vary from versions to versions. Some may indicate the code as part of statusCode or append as part of body If the web client encountered any issue while talking to a service implemented with a newer version of a redfish or swordfish schema, check the error response and add the status code in the below lines of app.js file

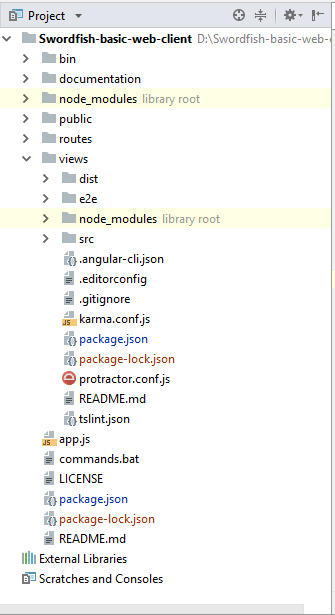


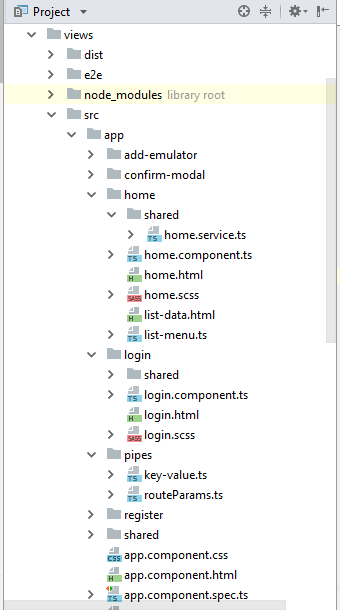
\*\* Please refer to Swordfish Basic Web Client User Guide on how to add/remove collections/properties

.

# Project Structure of Swordfish Basic Web Client

1. The Application is build using node, express and angular-cli
2. **Package.json:** this file contains list of all the dependencies that are required to run and angular-cli
3. Run ‘npm install’ to install all the specified dependencies into local project
4. **node\_modules** will be created which contains the copy of the libraries/dependencies.





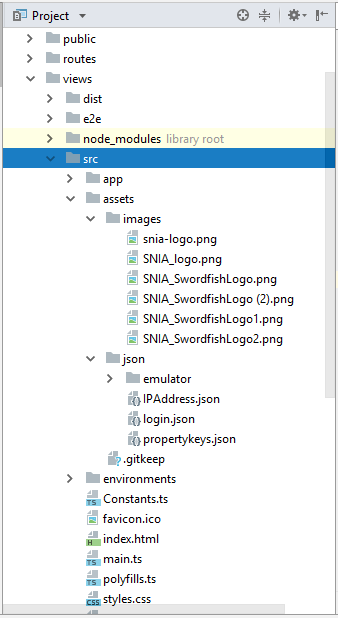


Fig: Project Architeture of basic web client

1. **views** folder contains the modules and components that are used to build the basic web client
2. **views/src** – This folder contains all the modules and components that aer used to build basic web client
3. **views/src/app –** This folder contains all the building blocks of basic web client application
4. **views/src/assets** – This folder contains all the static files,json and images that are used across the application
5. **.angular-cli.json** - contains the configurations that are required to load the angular-cli
6. **views/src/environments -** let you specify settings to customize your application behaviour

* You can define your own environments in the .angular-cli.json file.
* The default ones are:

1. **source**: use settings defined in environments/environment.ts
2. **dev**: use settings defined in environments/environment.ts
3. **prod**: use settings define in environments/environment.prod.ts

.

# Data Flow

The below diagram illustrates the data flow between components

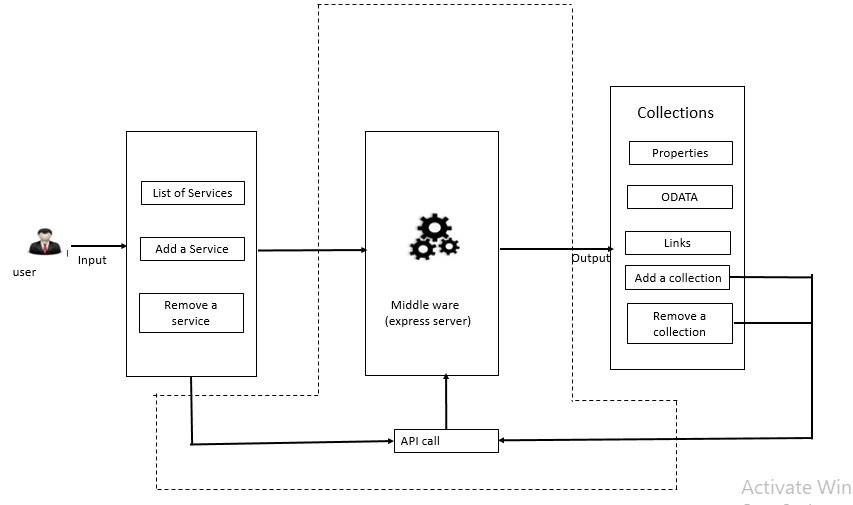


Fig: Data flow of Swordfish basic web client

## Views folder

1. Any Modifications/addition of new files that relates to functional change or enhancement of the basic web client are to be done to files in this folder
2. Middle Ware(app.js): This file is used as middle ware between basic web client and the backend server
3. **Views/src/app/home** : This folder contains the logic that is used to consume the input and produce the output

# Components Used for Swordfish Basic Web Client

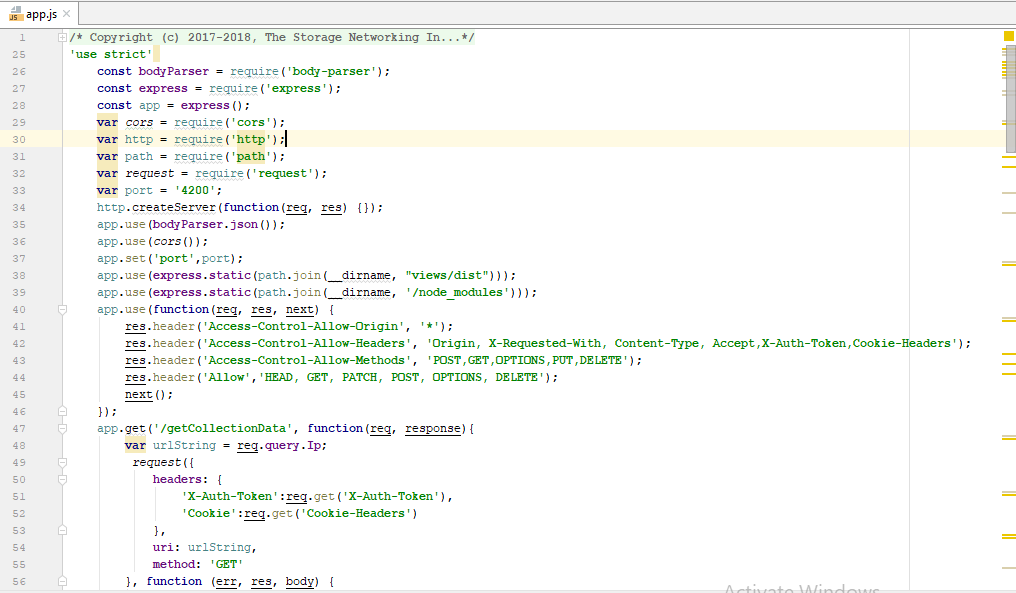
**App.js**

* This acts as a middle ware between angular components(basic web client ) and server(backend)
* All the required libraries are injected in to the app.js file using **require** method
* Example:

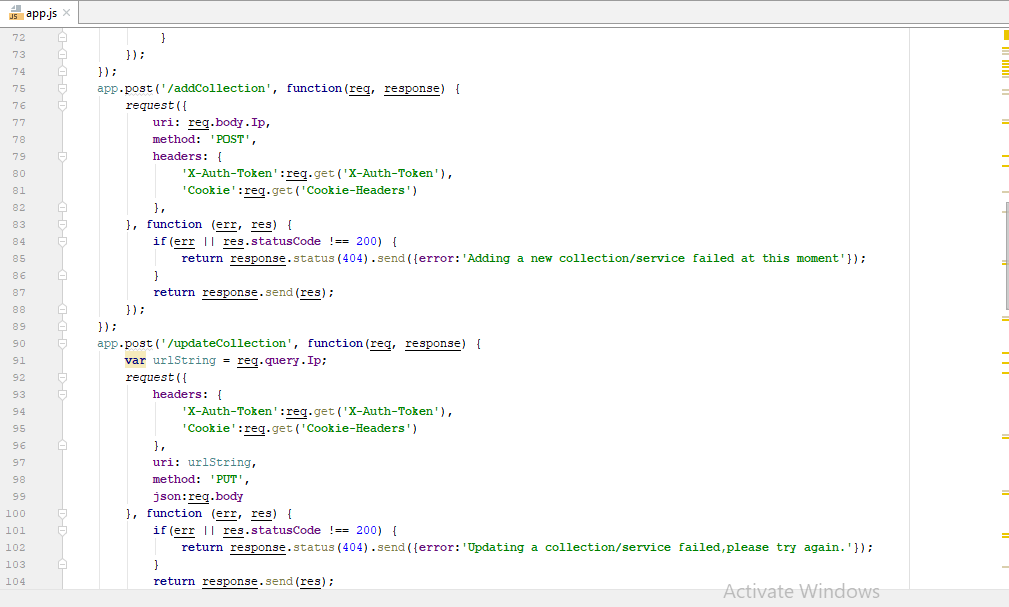
**var** *cors* = require(**'cors'**);  
 **var** http = require(**'http'**);  
 **var** path = require(**'path'**);  
 **var** *request* = require(**'request'**);

* To handle CORS issue, added required headers using app.use()

app.use(**function**(req, res, next) {  
 res.header(**'Access-Control-Allow-Origin'**, **'\*'**);  
 res.header(**'Access-Control-Allow-Headers'**, **'Origin, X-Requested-With, Content-Type, Accept,X-Auth-Token,Cookie-Headers'**);  
 res.header(**'Access-Control-Allow-Methods'**, **'POST,GET,OPTIONS,PUT,DELETE'**);  
 res.header(**'Allow'**,**'HEAD, GET, PATCH, POST, OPTIONS, DELETE'**);  
 next();  
});



* All methods that are used to communicate the view (basic web client) and backend(server) resides in this file
* CORS dependency is used to handle all the domain related issues and to transfer data from one domain to another
* **app.get ():** This method is used to fetch data from the endpoint.
* **app.post ():** This method is used to add data to the endpoint
* headers can be added and body is sent along with request



* **app.delete():** This method is used to perform any delete operations



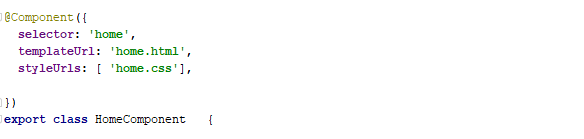
1. **app.module.ts** : This file organizes and separates the code

* imports: You can import any other modules into the imports section.
* declarations: You declare any components in your declarations. Any components used in the routing of that module, must be declared in that module. If components are used in another module, then you only list them in that other module.
* Pipes created in the application need to declare in the declarations section
* Providers: you can inject all the services into providers section
* Bootstrap: you can specify which component should bootstrap/load first into the browser in the bootstrap section



1. **home.component.ts**:

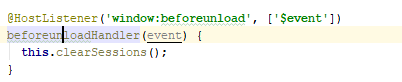
Component: It holds the configuration properties of the component like selector, names of properties, names of events and list of the injectables into the component. The value of the selector can be the same as the string to be used on HTML markup, it doesn’t need to be camel-cased



* To create a component (collection blade) dynamically ,a reference to the parent div is created with the help of @ViewChild interface



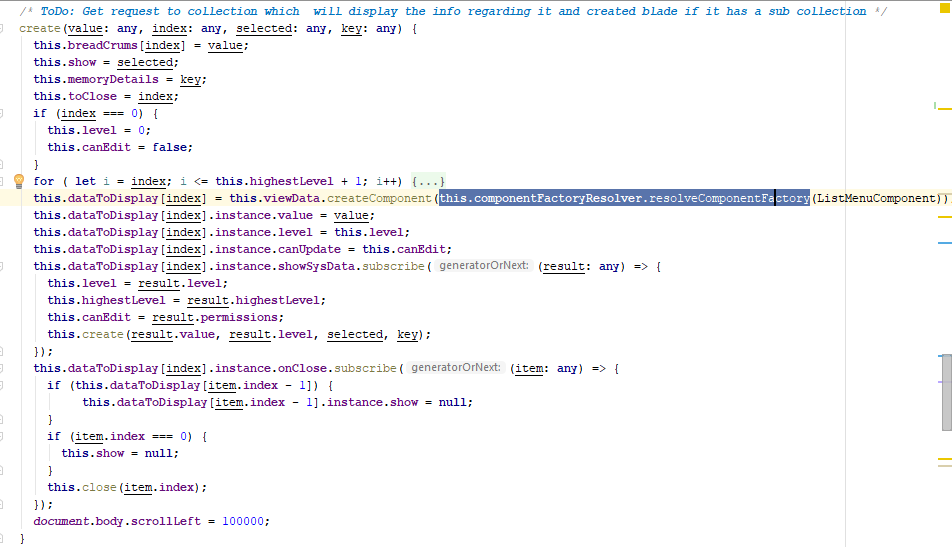
* To handle sessions and to maintain track of cookie id received while adding a service @HostListener and window events are used



* A constructor() is used to load data for html pages when component is instantiated and to inject dependencies of services

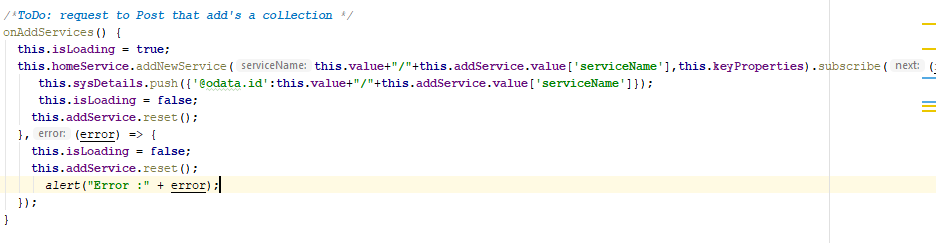


* The logic that identify the links present ,properties that exist for a collection is implemented in the create() method.
* componentFactoryResolver.resolveComponentFactory() is used to create a new blade(collection blade) if the links do exist for that collection

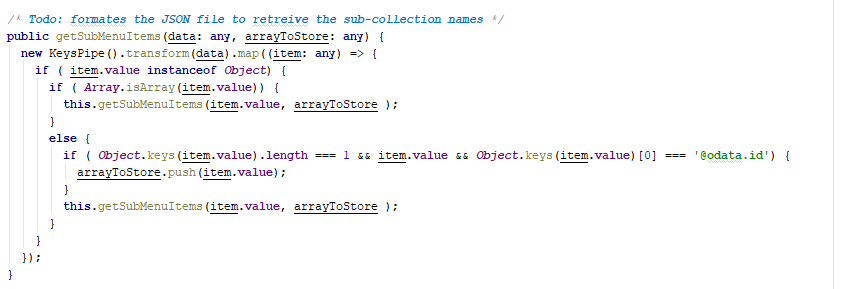


1. **list.component.ts :** The segregation of collection information runs in this component

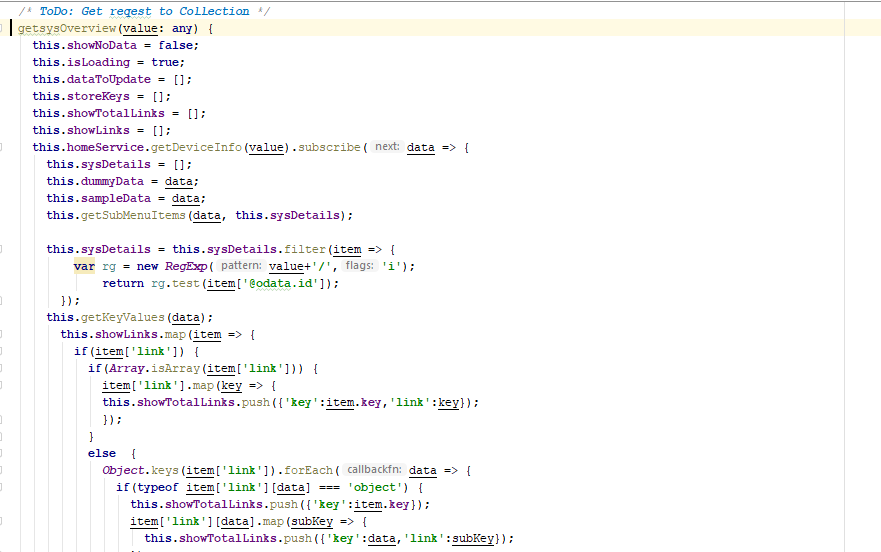
* Whenever a new service is added to the collection the response from the observable is fetched using subscribe and displays in view



* The navigable properties of collections are extracted from the @Odata string using the below method

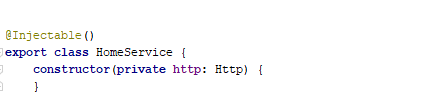


* All the properties of the collection is fetched and read entity either object/Array using getSysOverview()

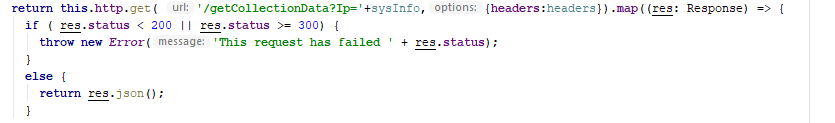


1. **home.service.ts :** The service layer from the client side is implemented in home.service.ts file

* An instance of http is created to make http calls across the browser



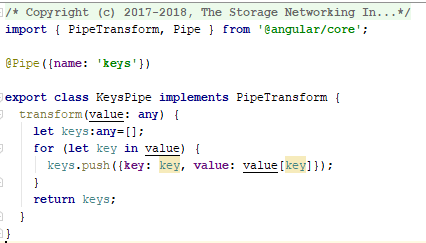
* http.get is used to make a request to fetch data from the server



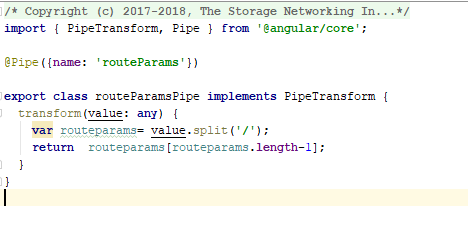
* mock API’s are built using express to avoid CORS. So, any request made from the [home.service.ts](http://home.service.ts) file will first hit the app.js file
* from app.js file the request are forwarded to the server

1. **pipes/key-value.ts** : This file is used to process and format the desired data

* The object/array entities are dived into key-value pairs where the property is stored into key and the value of the property is stored into value



1. **pipes/routeParams.ts** : To get the name of the current active collection from the link this pipe is used



1. **dist folder:** This folder contains thecompiled code of the angular application

\*\*\* build the angular application after each change/new pull request using the command cd views && ng build