|  |
| --- |
|  |
|  |
|  |

|  |  |
| --- | --- |
| Future of Data Collections, UI Architecture Document |  |
|  |  |
| Nilanjan Chaudhuri |  |
| 20th September 2019 |  |
| FCA Restricted |  |

Table of Contents

[1 Background 4](#_Toc23343080)

[2 Objective 5](#_Toc23343081)

[3 Business and Architectural Drivers 6](#_Toc23343082)

[3.1 Drivers 6](#_Toc23343083)

[4 Framework and Technology Stack 7](#_Toc23343084)

[4.1 The Single Page Application paradigm 7](#_Toc23343085)

[4.2 Advantages of a SPA 7](#_Toc23343086)

[4.3 SPA Framework of Choice 9](#_Toc23343087)

[5 FDC UI Architecture – Foundation 10](#_Toc23343088)

[6 High level Design & Packaging 11](#_Toc23343089)

[7 Angular Project Structure 13](#_Toc23343090)

[7.1 Drivers 13](#_Toc23343091)

[7.2 Project Structure 14](#_Toc23343092)

[7.3 Sample Expanded Project Structure 18](#_Toc23343093)

[7.4 Working Example 20](#_Toc23343094)

[7.4.1 PSD Single Page Application 20](#_Toc23343095)

[7.4.2 RMA Single Page Application 20](#_Toc23343096)

[7.5 Shared Services Single Page Application 21](#_Toc23343097)

[8 Accessibility 26](#_Toc23343098)

[8.1 Overview 26](#_Toc23343099)

[8.2 Semantics 27](#_Toc23343100)

[8.3 Accessibility Attributes 28](#_Toc23343101)

[8.4 Routing and Focus Management 29](#_Toc23343102)

[8.5 Angular UI Components 29](#_Toc23343103)

[9 API based Solution for Dynamic Form Rendering and Multilanguage 31](#_Toc23343104)

[9.1 Solution Overview 31](#_Toc23343105)

[9.2 Solution Details 32](#_Toc23343106)

[10 Client Side Validation Framework 36](#_Toc23343107)

[11 Error Handling 38](#_Toc23343108)

[12 Best Practices 39](#_Toc23343109)

[12.1 Injectable Singleton Services – Single Responsibility 39](#_Toc23343110)

[12.2 TypeScript Clean Code 39](#_Toc23343111)

[12.2.1 Variables and Functions 39](#_Toc23343112)

[12.2.2 Objects and Data Structures 40](#_Toc23343113)

[12.2.3 Classes 40](#_Toc23343114)

[12.2.4 SOLID 41](#_Toc23343115)

[13 Appendix 42](#_Toc23343116)

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Issue Status** | **Author** | **Description** | **Date** |
| 0.1 | Nilanjan Chaudhuri | Initial Draft | 03.07.2019 |
| 0.2 | Nilanjan Chaudhuri | Updated Angular Packaging | 06.08.2019 |
| 0.3 | Nilanjan Chaudhuri | Updated Angular project structure | 19.08.2019 |
| 0,4 | Nilanjan Chaudhuri | Firm major version with updated section for Multilanguage implementation m Accessibility, Project structure | 02.09.2019 |
| 0.5 | Nishi Ajmera, Dinesh Aswale | Validation framework and Error Handling | 14.10.2019 |
| 1.0 | Nilanjan Chaudhuri | Updated all sections, updated diagrams, aggregated content | 19.10.2019 |

**Distribution and History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Issue Status** | **Issued To** | **Role** | **Date** |
| **1.0** | **Swapnonil Mukherjee** | **Enterprise Architect -DAPG** |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Document Reference**

|  |  |  |
| --- | --- | --- |
| **Document Name** | **Location** | **Version** |
| AIA Document | <http://recmgmt.is.fsa.gov.uk/livelink/livelink/overview/57047469> | 1.0 |
| Solution Outline Document | <http://recmgmt.is.fsa.gov.uk/livelink/livelink/overview/60217859> | 1.0 |

**Abbreviations**

| **Abbreviation** | **Definition** |
| --- | --- |
| FDC | Future Data Collection |
| AIM | Analytics & Information Management |
| UDOP | Unified Digital Operating Platform |
| XBRL | eXtensible Business Reporting Language |
| DAPG | Data and Analytics Product Group |
| IPG | Infrastructure Product Group |
| CDC | Cloud Data Centre |
| DADA | Data and Analytics Design Authority |
| TSG | Technical Steering Group |

# Background

Future of Data Collections is a large program being initiated by the FCA to replace the ageing GABRIEL solution.

The scope for Future data collection (FDC) is to, “Deliver a stable, bespoke, flexible platform that meets **the scheduled and ad hoc data collection needs** of the FCA and PRA”. The critical path for Future of Data Collection is to replace Gabriel by March 2021 in advance of the data centre closure and to mitigate risks due to reliance on aged components.

# Objective

The purpose of this document is to establish the drivers, architectural approach and principles that drive the design and build of the FDC User Interface and covers the various aspects of the components that play together to render the UI application.

Most of the inherent principles governing the overall FDC platform, as laid out in the Solution Outline document (provided in references section) still apply to the UI Architecture. However, this document specifically dwells into the architecture and design of the User Interface application

# Business and Architectural Drivers

This section specifies the drivers (business and architectural) which drive the technology stack, architecture, and design of the FDC Front-End User Interface application.

## Drivers

1. Fast and responsive - The FDC front end UI application would strive to ensure that there is a negligible wait time for the user in terms of application response, aligning with the NFR specifications.
2. Decouple frontend from the backend – The developer should not be writing code to render pages on the server. Once the skeleton page loads, the server acts purely as a data provider.
3. Easy to deploy, support for versioning and rollback
4. Offline support and caching – Ability to cache local data which would ensure a smoother user experience if the device loses connectivity.
5. Smooth future transition to mobile development – It should be easy to build a future mobile interface from the UI application.

# Framework and Technology Stack

## The Single Page Application paradigm

A SPA is a web application that has one page — a single HTML page that is loaded in the browser and is not reloaded during use. When users interact with the page, the application dynamically rewrites/updates the current page rather than loading a new page from the server. Since the HTML is rendered and manipulated on the client-side, the size of the payload is decreased because the server returns only JSON, not HTML. Many popular JavaScript frameworks, including Vue.js, Angular, Ember.js, Meteor.js, and React, have adopted SPA principles.

## Advantages of a SPA

1. Fast and fluid response time - SPAs request the mark-up (HTML) and the data (JSON) independently and renders pages directly in the browser. All the necessary code (HTML, CSS, JavaScript) is received in a single page load or the appropriate code is dynamically loaded as needed in response to user actions. However, while a SPA is running, only data is sent over the wire, which takes a lot less time and bandwidth than constantly sending HTML.



1. Simple to Build – SPAs decouple the front end from the back end and the UI developer does not have to write code to render pages on server. The entire backend can be replaced as long as the APIs are the same and vice versa.
2. Simple to Debug – Debugging of SPAs in Chrome is easy and straightforward and it is easy to investigate page elements and HTTP requests.
3. Simple to Deploy in production – SPAs load just one index file with a CSS bundle and a JavaScript bundle. These static files can then be uploaded to any static content server like Amazon S3 , Apache Nginx or Firebase Hosting. Please refer to the logical infrastructure architecture for more visibility on this.
4. SPA versioning in production – Another advantage of deploying the front-end application as a SPA is versioning and rollback. The build output (CSS and JS bundles) can be versioned. The server that is serving the SPA can be configured with a parameter which specified the version of the front-end application to build.
5. Mobile-friendliness – SPAs can easily adapt to mobile device browsers as long as the APIs remain the same and transitioning to a mobile experience for the FDC application would not involve complete new development of the UI.
6. Caching capabilities – SPAs can effectively cache local data. They send a single request to the web server and after that saves all the data it gets. Since the apps have ongoing access to this data, they let the users to keep on working even if they lack a strong network connection.

## SPA Framework of Choice

FDC UI will use Angular as the SPA framework of choice. The SPA frameworks in contention were React and Angular though in essence they are quite different in their technical contexts and their use cases are also different.

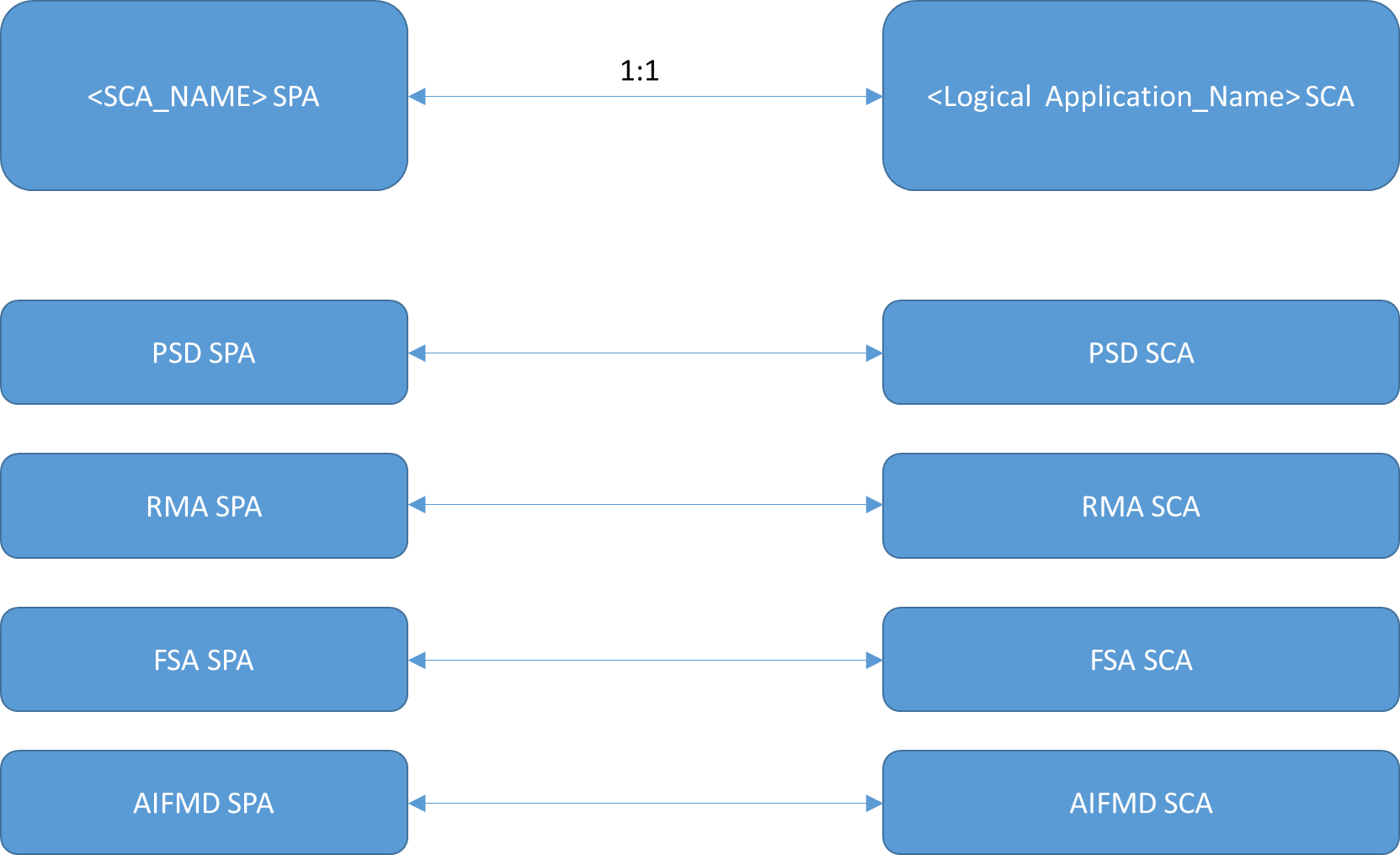
Choosing Angular over React is a bet on the jockey and not necessarily on the horse as Google have been much better custodians of open source software.

|  |  |
| --- | --- |
| **Angular** | **React** |
| A JavaScript Framework | A JavaScript Library |
| Works with Regular DOM | Works with virtual DOM |
| Two-way binding | One way binding |
| Has a CLI – ng cli | Does not have a proper CLI |
| Best fit for web applications which update one view at a time | Best fit for web applications which update many views at a time |

# FDC UI Architecture – Foundation

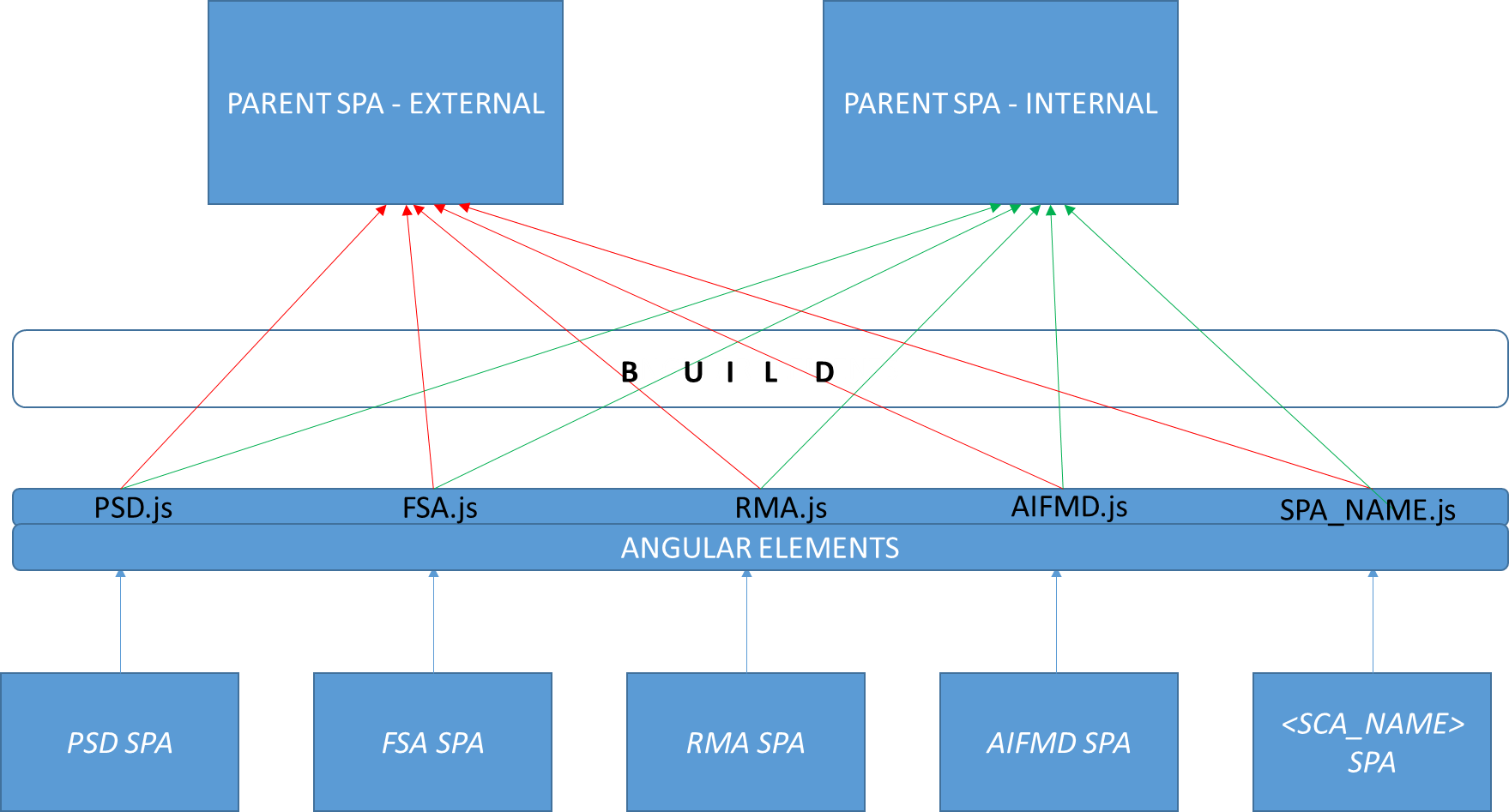
The foundation of the UI architecture is based on a ***one to one*** linear relationship between the logical Self-Contained Applications (SCA) and the corresponding Single Page Application UIs (SPA).

In other words, every logical Self-Contained Application (SCA) will have its own UI application or Single Page application.



# High level Design & Packaging

This is the tentative approach and is awaiting a POC completion for technical feasibility on a few aspects. Certain aspects of the feasibility have already been proven.



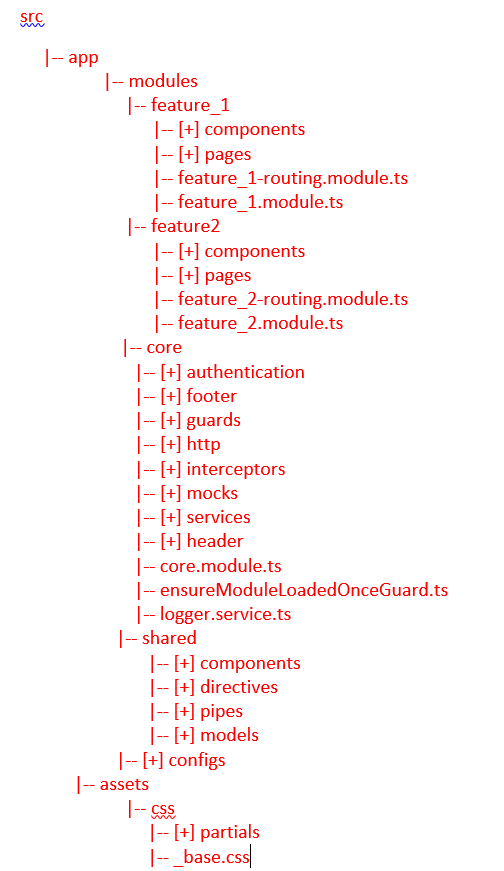
* The salient features of the FDC Front End Application Design and packaging are as follows
* There will be two parent Single Page Applications (SPA) – one for the external users and one for the internal users
* The parent SPAs are the final runtime artefacts, which will render the UI for the overall application.
* Each parent SPA will be built of child SPAs – where each such child SPA will represent a functional module in the FCA application and will have a one to one linear relationship with the corresponding Self-contained Java application.
* Thus, both the External and Internal SPAs will be built of child SPAs that are common to both and will be plugged into them, alongside the core and utility modules.
* Each such child SPA will represent an independent UI application with its own structure and packaging.
* Using Angular elements each such child pluggable SPA will be converted into its specific js files
* We will use Angular dependency injection to pass in the reference in the current module’s injector. The injector lets us share the context in multiple applications as shared services
* During build, each such .js file (representing each child SPA) will be plugged into the parent SPA
* In this way, the code is not duplicated. Each child SPA is plugged into the External and the internal application but is an entity in itself.
* This also helps parallel development and testing of each such SPA without impacting the other SPA.
* Each such child SPA will have its own master and develop branches.

# Angular Project Structure

## Drivers

* Locate code quickly
* Identify the code at a glance
* Keep the Flattest structure you can
* Try to be DRY
* Have a near term view of implementation and long-term vision

## Project Structure



We will use the following project structuring approach to meet the above LIFT principles

The main tenets of the approach include –

1. Have all the code for the application under a *“src”* folder
2. Create a ngmodule in the app’s root folder – ( *src/app*/*app.module.ts )*
3. The core module is designed to contain singleton services, global components and features that have only one instance per application. The idea is to make sure that only instance of these services is created in the entire app. The core module is included in the entire app only once in AppModule (in the import property of the @NgModule () decorator inside the main app.module.ts). To prevent reimporting the core module elsewhere, we may add a guard in the core module constructor.

|-- core

|-- [+] authentication

|-- [+] footer

|-- [+] guards

|-- [+] http

|-- [+] interceptors

|-- [+] mocks

|-- [+] services

|-- [+] header

|-- core.module.ts

|-- ensureModuleLoadedOnceGuard.ts

|-- logger.service.ts

The authentication handles the authentication user flow

|-- authentication

|-- authentication.service.ts|spec.ts

The header and footer contains the global component files used across the entire application

|-- header

|-- header.component.ts|html|css|spec.ts

|-- footer

|-- footer.component.ts|html|css|spec.ts

The http folder handles all http calls from the application. Additionally, it will also contain one single api.service.ts file to maintain all http calls from the application in one single place. This is optional.

|-- http

|-- user

|-- authorization.service.ts|spec.ts

|-- authentication.service.ts|spec.ts

|-- api.service.ts|spec.ts

The interceptor folder is a collection of useful interceptors.

The guard’s folder contains all the guards to protect different routes in the application.

|-- guards

|-- auth.guard.ts

|-- no-auth-guard.ts

|-- admin-guard.ts

Mock folder contains all the mock files for app and is specifically useful for testing

|-- mocks

|-- psd007transaction.mock.ts

The services folder consists of the Services. The services should be singleton services and should have single responsibility. These would be injected into the components as pluggable services.

1. Structure the project by feature – Create folders by feature areas they represent – *app/feature\_name*. This will help to locate the code quickly and identify the code at a glance.
2. In the folder for the specific feature – place the corresponding feature module and name it as per the feature area that is being represented - *(feature\_name.module.ts)*
3. Create an NgModule for each distinct feature area so that routable features can be lazy loaded. It also helps to isolate, reuse and test features
4. Within a feature folder, keep the structuring flat – Have subfolders when the number of files exceed 7 (may vary)
5. The Shared Module contains all the components, pipes/filters and directives, which are referenced across the application - *app/shared/shared.module.ts.*

|-- shared

|-- [+] components

|-- [+] directives

|-- [+] pipes

|-- [+] models

The components folder contains all shared components like buttons etc., which are in turn used, by multiple components.

|-- components

|-- loader

|-- loader.component.ts|html|scss|spec.ts

|-- buttons

|-- submit-button

|-- submit-button.component.ts|html|fdc|spec.ts

|-- collapse-button

|-- collapse-button.component.ts|html|fdc|spec.ts

The directives, pipes, models and filters folder contain those which are used all across the application.

1. Features, which need to lazy-loaded need to be placed in a lazy-loaded folder and such features should not be imported directly by other modules. (as otherwise, they will be loaded on demand).
2. The configs folder contains application configuration settings and other predefined settings like language specific configuration.

|-- configs

|-- fdc-app-settings.config.ts

|-- fdc-endpoint-settings.config.ts

|-- fdc-app-settings.config.GB.ts

|-- fdc-app-settings.config.CY.ts

1. The assets folder will contain all common assets for the project including the global styling elements. The global styles are placed in a folder under assets.

## Sample Expanded Project Structure

src

|-- app

|-- app.module.ts

|-- modules

**|-- feature\_1**

|-- [+] components

|-- [+] pages

|-- feature\_1-routing.module.ts

|-- feature\_1.module.ts

**|-- feature2**

|-- [+] components

|-- [+] pages

|-- feature\_2-routing.module.ts

|-- feature\_2.module.ts

**|-- core**

|-- authentication

|-- authentication.service.ts|spec.ts

|-- footer

|-- footer.component.ts|html|css|spec.ts

|-- guards

|-- auth.guard.ts

|-- no-auth-guard.ts

|-- admin-guard.ts

|-- http

|-- user

|-- authorization.service.ts|spec.ts

|-- authentication.service.ts|spec.ts

|-- api.service.ts|spec.ts

|-- [+] interceptors

|-- [+] mocks

|-- [+] services

|-- header

|-- header.component.ts|html|css|spec.ts

|-- core.module.ts

|-- ensureModuleLoadedOnceGuard.ts

|-- logger.service.ts

**|-- shared**

|-- components

|-- loader

|-- loader.component.ts|html|scss|spec.ts

|-- buttons

|-- submit-button

|-- submit-button.component.ts|html|fdc|spec.ts

|-- collapse-button

|-- collapse-button.component.ts|html|fdc|spec.ts

|-- [+] directives

|-- [+] pipes

|-- [+] models

**|-- configs**

|-- fdc-app-settings.config.ts

|-- fdc-endpoint-settings.config.ts

|-- fdc-app-settings.config.GB.ts

|-- fdc-app-settings.config.CY.ts

|-- assets

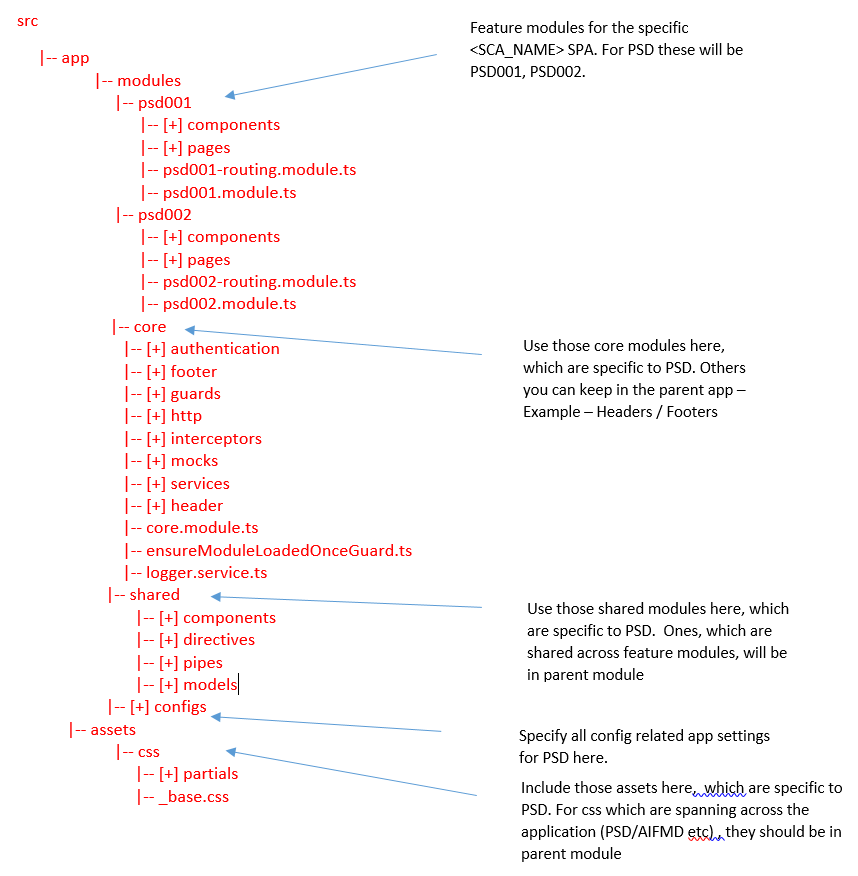
|-- css

|-- [+] partials

|-- \_base.css

## Working Example

### PSD Single Page Application



### RMA Single Page Application

src

Feature modules for the specific <SCA\_NAME> SPA. For RMA, these will be RMAA, RMAD1 and so on

|-- app

|-- modules

|-- rmaa

|-- rma\_a\_v1

Provision for version information should be in place for SPAs, which require version information

|-- [+] components

|-- [+] pages

|-- rmaa-routing.module.ts

|-- rmaa.module.ts

|-- rma\_a\_v2

|-- [+] components

|-- [+] pages

|-- rmaa\_v2-routing.module.ts

|-- rmaa\_v2.module.ts

|-- rma\_d1

|-- rma\_d1\_v1

|-- [+] components

|-- [+] pages

|-- rma\_d1\_v1-routing.module.ts

|-- rma\_d1\_v1.module.ts

|-- core

Use those core modules here, which are specific to RMA. Others you can keep in the parent app – Example – Headers / Footers

|-- [+] authentication

|-- [+] footer

|-- [+] guards

|-- [+] http

|-- [+] interceptors

|-- [+] mocks

|-- [+] services

|-- [+] header

|-- core.module.ts

|-- ensureModuleLoadedOnceGuard.ts

|-- logger.service.ts

|-- shared

Use those shared modules here, which are specific to RMA. Ones, which are shared. across feature modules, will be in parent module

|-- [+] components

|-- [+] directives

|-- [+] pipes

|-- [+] models

|-- [+] configs

Specify all config related app settings for RMA here.

|-- assets

|-- css

|-- [+] partials

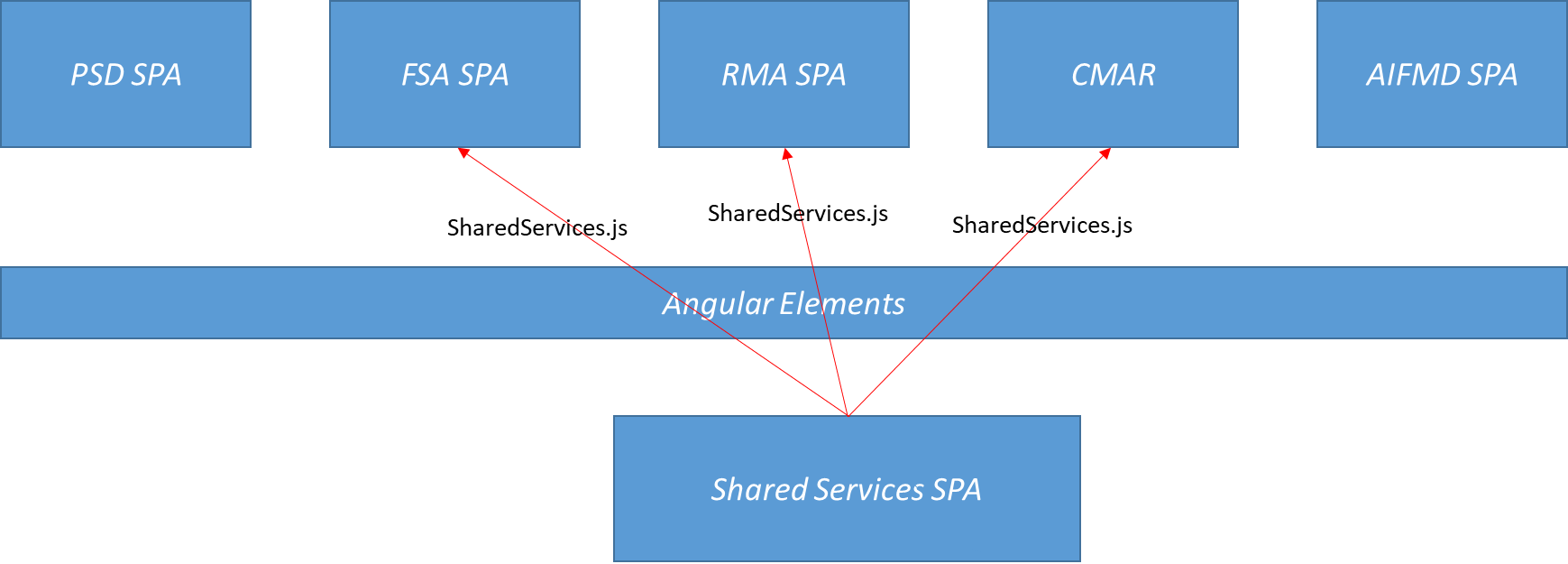
|-- \_base.css

Include those assets here, which are specific to RMA. For css which are spanning across the application (PSD/AIFMD etc), they should be in parent module

## Shared Services Single Page Application

We will also have shared SPA’s if there are shared services and utilities across the child SPA’s or some of them as the driver is to avoid duplication and enhance maintainability as much as possible.

For example, the Online SPA’s like RMA, FSA, CMAR etc may have a lot of common services, shared services, shared utilities, configs etc which may be common to each of them. Instead of bundling them as part of each separate SPA, we can have a shared Services SPA, which will be injected using Angular elements in a similar manner to these child SPAs. There can be one more of these shared SPA’s as the case maybe.



src

|-- app

|-- modules

|-- utility\_modules

|-- [+] components

|-- [+] pages

|-- utility.module.ts

|-- utility.module.ts

|-- core

|-- [+] guards

|-- [+] http

|-- [+] interceptors

|-- [+] mocks

|-- [+] services

|-- core.module.ts

|-- ensureModuleLoadedOnceGuard.ts

|-- logger.service.ts

|

|-- shared

|-- [+] shared\_components

|-- [+] shared\_directives

|-- [+] shared\_pipes

|-- [+] shared\_models

|

|-- [+] configs\_shared

|-- assets

|-- css

|-- [+] partials

|-- sharedCss

Injection of the Shared Services module into the RMA and FSA

src

|-- app

|-- modules

|-- rmaa\_a

|-- rmaa\_a\_v1

|-- [+] components

|-- [+] pages

|-- utility.module.ts

|-- utility.module.ts

|-- core

|-- [+] guards

|-- [+] http

|-- [+] interceptors

|-- [+] mocks

|-- [+] services

|-- core.module.ts

|-- ensureModuleLoadedOnceGuard.ts

|-- logger.service.ts

|

|-- shared

|-- [+] shared\_components

|-- [+] shared\_directives

|-- [+] shared\_pipes

|-- [+] shared\_models

|

|-- [+] configs\_shared

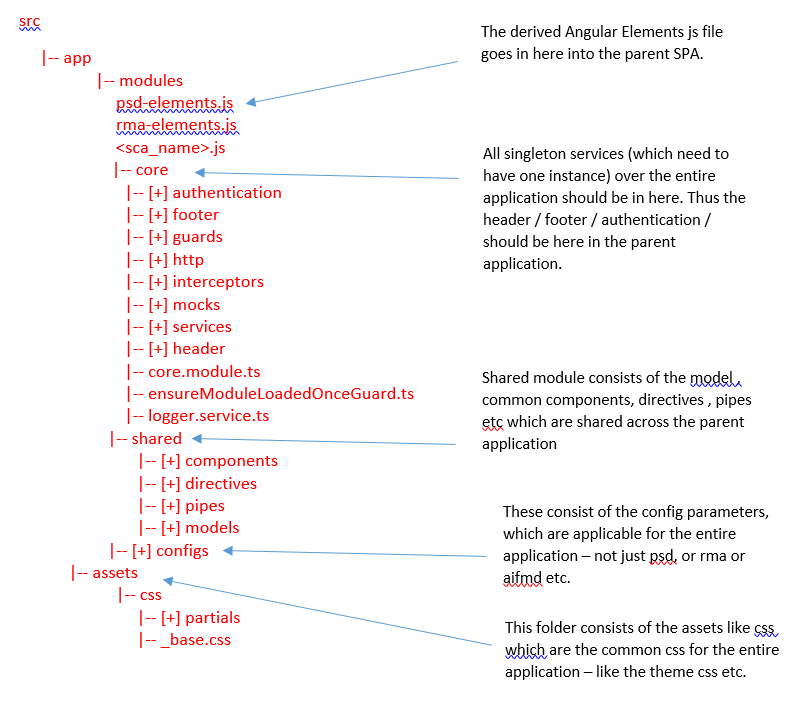
|-- assets

|-- css

|-- [+] partials

|-- sharedCss

Parent Single Page Application



*Hence there will be two copies of the Parent SPA – An External SPA (for external users) and an Internal SPA (for internal users) and the structure will be the same for both as specified above.*

# Accessibility

Operational Acceptance Testing strategy is yet to be developed and would be driven by signed-off NFRs.

## Overview

Some people cannot

* + Use a mouse
  + View a screen
  + See low contrast text
  + Hear dialogue or music
  + Understand complex language

Some people need

* + Keyboard support
  + Screen reader support
  + High contrast text
  + Captions and transcripts
  + Plain language

The FDC accessibility framework would strive to adhere to the main Accessibility principles –

* Perceivable – All users with varying degrees of sensory abilities should be able to perceive the content. If something is perceivable with one sense , such as sight does not mean that all users can see it.
* Operable – Users should be able to use UI components and navigate content. A component that requires a hover interaction cannot be operated by someone who does not use a mouse or touch screen.
* Understandable – The User interface should be consistent enough to avoid confusion.
* Robust – The content can be rendered by a wide variety of user agents or browsers including with assistive technology.

Accessibility Design Overview for FDC

To adhere to the above principles of accessibility The FDC UI accessibility framework uses various techniques which can be broadly categorised into three categories –

* Semantics
* Focus
* Styling

## Semantics

If we assume that we are building a UI for a screen reader, we would not be required to create a visual UI at all, instead we just need to provide just about enough information for the screen reader to understand. This is the accessibility tree. A browser can transform the DOM tree of a accessible visual web application into an accessibility tree as the DOM has implicit semantic meaning. In other words , for this , the DOM needs to use native html elements that are recognised by all browsers.

* Use of native HTML elements – Native HTML elements capture many important interaction patterns that are important to accessibility and the FDC UI application would strive to reuse these native elements when possible rather than re-implementing well-supported behaviours.
  + Use html button element for ‘click’ events which describes actionable area. This enables the button to be easily be focused, outlines the button currently focused and enables traversing to the button by keyboard tabbing.
  + Use html link for navigation instead of using a div with same class as button.
  + Use html form tag for forms. This enables keyboard tabbing, navigation and indicates an input making the application easily accessible.
  + For all input elements, either use a <label> tag or use aria-label along with placeholder. This describes what the input element does.
* Use of proper HTML tags to ensure proper page structuring and hierarchy

1. The UI pages would have proper structuring through use of nav, header,footer,section,article and aside to describe the layout of the page
2. Proper use of h1, h2 tags to ensure proper page hierarchy

* Containers for native elements – The FDC UI will use container components that use content projection to include native control in the components API

## Accessibility Attributes

The FDC UI Application will use aria attributes to provide semantic meaning where it might otherise be missing. For custom components we will use necessary aria attributes as it helps screen readers to understand when dynamic components have entered in a view.

* aria-label - It will use static aria attributes like
* <button aria-label = “Validate and Save”> .. </button>
* It will also use attribute binding to control values of accessibility related attributes.
* <button [attr.aria-label]=Validate and Save> .. </button>
* aria-owns – We will use aria-owns to inform screen readers that an element that is separate in the DOM should be treated as a child of the current element.
* aria-landmark - It would also enable navigation by landmark using ARIA landmarks – in order to allow users to navigate between page landmarks like headers. The FDC UI pages would aim to have atleast 3 ARIA landmarks – main , banner, contentinfo. Additoinally we will also use the navigation landmark to allow for pages with navigation. The corresponding HTML elements will have a role attribute with role = “<landmark>”.
* aria-describedby / aria-labelledby – to descrive elements that are otherwise not visible but provides some extra explanatory texts which assistive users might need.

## Routing and Focus Management

Tracking and controlling focus in the FDC UI is a very important factor in making the UI accessible. Input focus that follows the order of the visual layout usually flows from the top to the bottom of the screen.

* The FDC UI will ensure that all fields, buttons and any other interactive elements such as tool tips are tab friendly so that keyboard users can easily navigate through the form without the use of a mouse.
* Radio buttons and checkboxes will use legends and fieldsets
* Labels for drop down and text fields
* Navigation to validation error messages
* Error messages with alert tone
* The FDC UI will use the NavigationEnd element from the Router service to know when to update focus and will find and focus the main content header in the DOM after navigation.
* *router.events.pipe(filter(e => e instanceof NavigationEnd)).subscribe(() => {*
* *const mainHeader = document.querySelector('#main-content-header') if (mainHeader) {*
* *mainHeader.focus(); }*
* *});*
* The FDC UI will ensure that offscreen elements which are part of the DOM but not currently visible should not receive focus and hence maintain the taborder. We can use display:none or visibility: hidden to hide the elements and not allow it to receive focus.
* Use the default focus styles for all input elements or add one.
* Notify the screen reader of the siplay changes by setting the focus on the heading of the area that is changing.

## Angular UI Components

The core of the FDC UI Accessibility framework is based on creating a suite of reusable UI components that aim to be fully reusable and accessible by leveraging the Angular material library- a11y package (*Angular CDK a11y package -* import {A11yModule} from '@angular/cdk/a11y'), which provides a number of tools to improve accessibility in the application.

[*https://material.angular.io/cdk/a11y/overview*](https://material.angular.io/cdk/a11y/overview)

These components in turn use many aria-attributes within them. Broadly they are classified as

* Form Controls
* Navigation
* Layout
* Buttons and Indicators
* Popups and Modals
* Data Table

# API based Solution for Dynamic Form Rendering and Multilanguage



## Solution Overview

The FDC Application will provide the capability of dynamic form rendering and Multilanguage implementation in the following manner

1. On load of a form or any html view in the FDC angular application, the page labels are not specified within the angular view in a static manner, instead they are retrieved dynamically from a custom API based content management service.
2. The Angular page view calls the Outer Service by passing the formId, the version of the form if any (in order to support backward compatibility) and the language in which the application needs to be rendered.
3. The Outer Service acts as a bypass service and in turn invokes the Content Management Platform Service (CMS Service) with the same request parameters.
4. For each unique combination of formId, version and language we keep an entry in a DynamoDb table. Thus, for a single version, there are at least 2 entries for each form – one for language = English and one for language = Welsh. Each such entry has a specific json with the form labels (ID and value) with naming convention as <formId>\_<version>\_<language>. Each such json is stored in S3 bucket and the table contains the file path to the S3 bucket.
5. The CMS Platform service on receiving the request from the Outer service, retrieves the corresponding item in the DynamoDB table for the specific combination of - <formId>, <versionId> and <language> and gets the corresponding file from the S3 bucker associated with the row item and returns the json. This json essentially contains the forms labels for the requested version of the form and for the requested language.
6. The CMS Service returns the response to the Outer Service which in turn returns it to the Angular View which looks into the keys of the json and has a binding on the keys through which it renders the corresponding labels.
7. Thus the screen labels are displayed in the corresponding language and with the labels for the specific version. Please note the data would always be displayed in English.
8. There will also be a admin screen for the content management service which will allow the admin users to store these form label jsons and modify it dynamically without necessitating any impact on the FDC Angular UI application. This will be part of the API contract for the CMS Service and will be discussed in the corresponding design.

## Solution Details

1. A table need to be created in DynamoDB called ***fdc-forms*** – with following attributes

formId String (Sort Key)

language String (Partition Key)

version String

lastUpdatedTime String

filePath String

filePath is the path of the file in the S3 bucket.

1. Create a S3 bucket – ex - ***fdc-forms***
2. The JSONs to be stored in the bucket should be labelled as – formID\_language\_version.json
3. For each view / form of the FDC application, there should be a separate and unique entry in the Dynamo DB table for every unqiue combination of <formId>, <version>, <language> which means for one version, there are at least 2 entries to provide for support for English and Welsh. This also implies an equal number of jsons in the S3 bucket. Thus, every unique combination of <formId><version><language> - there will be a separate entry in the DynamoDB table and a separate json in the S3 bucket.

The form label json stored in S3 should have the fully qualified name of every attribute in the view page / form page as the key. The value should be the corresponding label texts (in English for the \_en.json OR in welsh for the \_cy.json). The fully qualified name of an attribute is as specified in the XSD of the form or the page.

Hence if there are two labels of the same name (say X) in the form (say Form F1) in two section, say A and B then the key should be be fully qualified in the form

<Form\_name>. <Section\_Name>.<Subsection\_Name>.labelKey

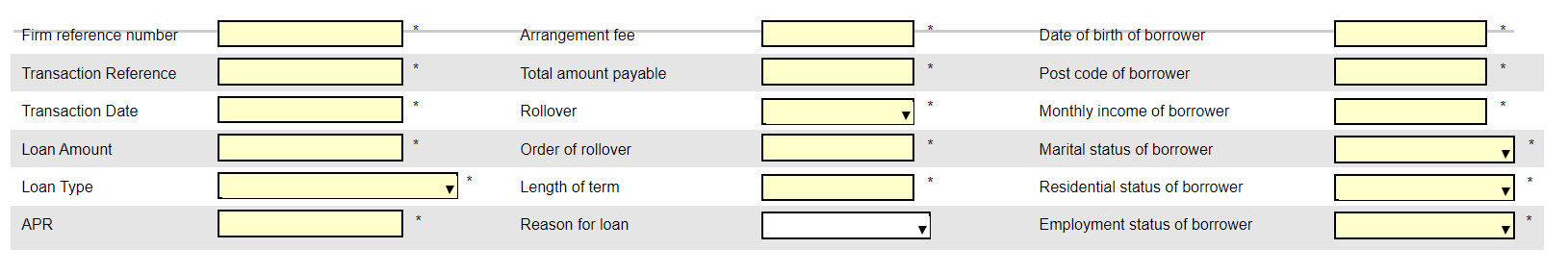
In this case it would be

F1.A.X = X

F1.B.X = X

This should be true for each json file for a specific form

1. Consider Short Term loans



* 1. We need to create two jsons in this format for this above table
     + One for English

<complete\_hierarchy\_to\_attribute>\_<XML\_ElementName>: field-label\_text

Example

{

<complete\_hierarchy\_to\_attribute>\_ firmReferenceNumber: Firm reference number,

<complete\_hierarchy\_to\_attribute>\_transRef: Transaction Reference,

<complete\_hierarchy\_to\_attribute>\_transactionDate: Transaction date,

<complete\_hierarchy\_to\_attribute>\_loanAmount: Loan Amount

<complete\_hierarchy\_to\_attribute>\_loanType: Loan Type

<complete\_hierarchy\_to\_attribute>\_apr: APR

<complete\_hierarchy\_to\_attribute>\_arrangementFee: APR: Arrangement fee

--

--

}

* One for welsh

{

<complete\_hierarchy\_to\_attribute>\_ firmReferenceNumber: lorem ipsum,

<complete\_hierarchy\_to\_attribute>\_transRef: lorem ipsum,

<complete\_hierarchy\_to\_attribute>\_transactionDate: lorem ipsum,

<complete\_hierarchy\_to\_attribute>\_loanAmount: lorem ipsum,

<complete\_hierarchy\_to\_attribute>\_loanType: lorem ipsum,

<complete\_hierarchy\_to\_attribute>\_apr: lorem ipsum,

<complete\_hierarchy\_to\_attribute>\_arrangementFee: lorem ipsum,

--

--

}

1. The Service to be created would be a Platform Service and would be a Spring Boot project and named as DCP-SCS-CMS-Service which would be responsible for all Content Management specific operations. API method to be created in CMS Platform Service (DCP-SCS-CMS-Service) which has the following input parameters – **formId, language, version**. It produces a response json as specified above.

It should have two APIs for this purpose

/{formId}/{language} – GET (version as request parameter)

/{formId}/{language} – POST (version as request parameter and formData

as request body)

1. The Angular form view should use the binding on the json keys and display the corresponding values. So, for the first field in the form – it should bind the firmReferenceNumber key such that it displays the value of the firmReferenceNumber attribute as defined in the json. If the backend administrator changes the value of the firmReferenceNumber attribute to “XXX” it will display “XXX” instead of “Firm reference number”.
2. This also means that Multilanguage will be implemented via this API service. Thus, for every form we will have two JSON defined – one for English and one for Welsh.

# Client Side Validation Framework

For implementation of client side validations, a validation rules ts file is created which has the client side validation logic for all data types present inside it. A validation service is created which reads the json file created from the data items xsd, creates an array of form control validators and adds it to the form group. Below steps are needed to be incorporated to the component.ts file for each data-item.

Once the form group for any form is created, the ‘setFormValidators()’ method of validation service should be called using following syntax:

Component.ts File:

constructor (

    private validationService: ValidationService

) { }

…

ngOnInit() {

…

    this.testRMAEForm = this.fb.group({

        // Your form group created in previous section.

    });

    this.validationService.setFormValidators('RMA-E-Schema', 'v8', 'RMA-E-PIISelfCertification', this. testRMAEForm);    // Arguments – Schema name, Version, Root schema element, Form Group variable.

…

}

Also, in case of dynamic control creation, the same method should be called again using same syntax. The root schema element should be replaced with the absolute dot-separated path to the root control. For example, the above mentioned form has a sub-group named as “PIIPolicies -> PIIPolicy” which allows dynamic control addition. Then, the validations for these dynamically added controls should be added as follows:

…

this.validationService.setFormValidators('RMA-E-Schema', 'v8', 'RMA-E-PIISelfCertification.PIIPolicies.PIIPolicy ', this. testRMAEForm, false, true);    // Arguments – Schema name, Version, Root schema element, Form Group variable, self, arrayChild

…

Note, here the arguments ‘self’ and ‘arrayChild’ specify whether the validations are to be applied to only the sepecified element or the children as well and whether the specified element is an entity within the form array or not, respectively.

# Error Handling

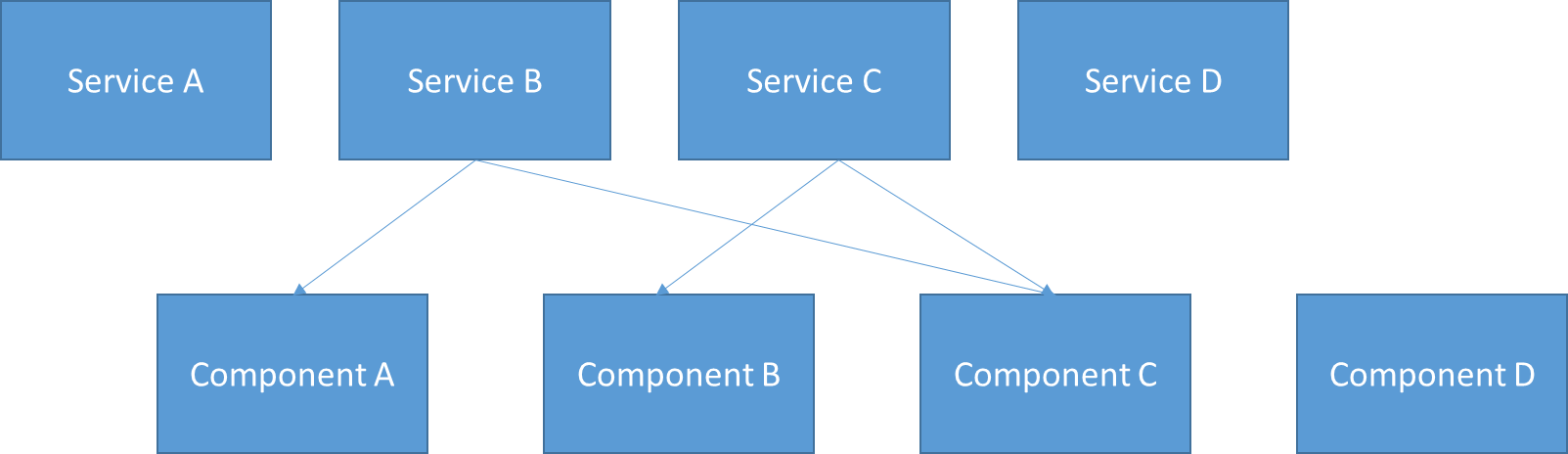
Error handling is implemented using http interceptor interface provided by Angular. An error handling service is created which has the interceptor logic implemented in it. All the http requests goes through the interceptor and in case of any errors it is routed to the error page which displays the error code and the error message.

# Best Practices

## Injectable Singleton Services – Single Responsibility

All services in Angular should be Singleton services and should have single responsibility. The services should be as pluggable entities into components (views).

The services should be placed in the core modules as there can be only one instance of the services across the application with guards placed.



So the Angular components, which are primarily used to render the views, can be injected with these services as per the functionality and depending upon the service to be called and the data to be retrieved. The services can be thus used as reusable services across components, but all components must use the single instance of a specific service. We use guards to ensure the single instance of these services.

## TypeScript Clean Code

### Variables and Functions

* 1. Use meaningful, pronounceable, searchable, and explanatory variables and variable names
  2. Same function name for same type of variable
  3. Do not add unnecessary context
  4. Use enums instead of const to document the code intent
  5. Function names should not have more than 2 parameters. If more than 2 parameters are required, used object literals instead. This improves readability of code.
  6. A function should be designed to perform a single functionality. Split functions if required but ensure you do not entrust more than one functionality to a function. Along with this , a function should have one level of abstraction. These two go hand in hand.
  7. Function names should specify the intent
  8. Remove code duplication but do the abstraction right.
  9. Use Object.assign for setting default objects or use destructuring
  10. Split the function if it has different code paths based on conditions (boolean)
  11. Use polymorphism to avoid conditions as a class or a function should do only one thing
  12. Avoid type checking using instanceOf and specify types of variables, parameters and return types
  13. Use generators and iterables while working with data collections used a stream.

### Objects and Data Structures

* Use getters and setters to access data from objects that encapsulate data, as typescript supports getter/setter syntax
* Typescript supports public, protected and private access specifiers on class members and so make objects use accessors.
* Use Typescripts immutability to mark individual properties on an interface / class as immutable using the Readonly built-in-type. For Arrays – use ReadOnlyArrays.
* Use type when you want a union or intersection and use interface when you want to extend or implement.

### Classes

* A class should be small
* Use High cohesion and low coupling in a class
* Use composition instead of inheritance when applicable.
* Use method chaining pattern

### SOLID

* Single Responsibility Principle – A class must have a single responsibility so that there is only reason for the class to change. In other words m we should avoid jam-packing multiple responsibilities to a single class. In such case, split it into multiple classes.
* Open/Closed Principle – Classes/modules/functions should open for addition of new functionalities, but closed for modification of existing functioanlity.
* Liskov Substituion Principle – Objects of a parent and child class can be used interchangeably without getting incorrect results.
* Interface Segregation Principle – Clients should not be burdened with implementing methods of an interface that they do not use. In such case , segregate the interface. This is to do with the way you design your abstractions and is tied to single responsibility principle.
* Dependency Inversion Principle – achieved through Dependency injection in Angular.

Please refer to TypeScript best practices list as in url mentioned in appendix.

# Appendix

| **SN** | **Description** | **Link** |
| --- | --- | --- |
| 1 | TypeScript Best practices | <https://github.com/labs42io/clean-code-typescript> |
| 2 |  |  |
|  |  |  |