Date: 29.07.20

Version: 0.1

DEVELOPMENT GUIDELINES

# Prerequisites

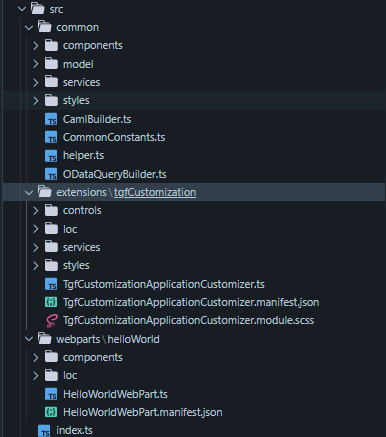
* Good understanding of React.
* Basic knowledge in Sass, in using UI library.
* Familiar with ES6.
* Basic knowledge in RxJs or observable pattern (not required but some of the codebase will use RxJs)

# General Guidelines

*The following guidelines are opinionated and are not forced to apply to every project. But in the author’s opinion, these guidelines help build clean, concise, and maintainable code.*

## Files structure

Suggestive files structure:



- **common** folder contains reusable code:

* **components** for reusable components
* **model** for type declaration, interface definition, class implementation,…
* **services** for business logic
* **styles** for global style, theming, and constants for the SCSS module.

- **extensions** for spfx extensions (automatically generated when creating spfx extension)

- **webparts** for spfx webparts (automatically generated when creating spfx webpart)

* **components** for webpart child components
* **loc** for localization files

## Every webpart should use Office UI Fabric library’s components

SharePoint modern experience is built on React and Office UI Fabric library, by using the library, we have similar UI with native Sharepoint components and provide a clean interface for the end-user.

In case of building a component from scratch, at least follow UI Fabric styling by using UI Fabric Core SCSS.

Reference:  
- Components and controls: <https://developer.microsoft.com/en-us/fluentui#/controls/>  
- Style: <https://developer.microsoft.com/en-us/fluentui#/styles>  
- Icons: <https://developer.microsoft.com/en-us/fluentui#/styles/web/icons>

## Separation of concerns (SoC) in React components

Avoid writing too much business logic inside a component: for simple components, it is ok to do so but in other cases, it will make the component bloated, decrease its readability and maintainability.

Business logic should be extracted to a common service file so we can reuse it in other components and make the component itself cleaner. Additionally, by using this pattern, we can easily implement in-memory caching for data that fetched from web requests.

For further refactoring, we can apply concepts such as [container-presentational component](https://medium.com/@alialhaddad/separation-of-concerns-in-react-d4f74aaf3800), [higher-order component (HoC)](https://reactjs.org/docs/higher-order-components.html), [render-props](https://reactjs.org/docs/render-props.html).

From React 16.8, we can increase reusability with [Hooks](https://reactjs.org/docs/hooks-intro.html).

## Use ES6 whenever possible

Using ES6 features help our code more readable and cleaner, also in some cases it helps improve performance.

Most frequently used ES6 features are:

* Array higher-order functions: forEach, map, reduce,…
* Spread and rest operator (…)
* Object destructuring
* String interpolation
* Arrow functions

## Making HTTP requests in SPFx

**Always use built-in HTTP clients** (*context.httpClient* for general HTTP, *context.spHttpClient* for SharePoint REST API, *context.graphClient* for Graph API) to make requests. To use it, we have to pass the webpart context from the top-level component of each webpart.

Reference: <https://docs.microsoft.com/en-us/sharepoint/dev/spfx/connect-to-sharepoint>

## Avoid open/close ClientContext too many times in CSOM code

* Opening ClientContext too frequently will affect the performance of the code and may cause throttling issues with SharePoint Server.
* Thoroughly consider whenever open or close ClientContext.
* Plan the strategy to get the data efficiently in one ClientContext session instead of opening multiple sessions to get data from only one site collection

# SPFx Codebase

*This section explains each file in* ***common*** *folder responsibility and functionality*

## CamlBuilder.ts

* A wrapper class for the camljs library.
* **CamlBuilder** class is really helpful to create a Caml query when calling SharePoint REST API.

## ODataQueryBuilder.ts

* A wrapper class for the odata-query-builder library.
* **ODataQueryBuilder** class helps us easily create an OData query when calling SharePoint Rest API.

## CommonConstants.ts

* Its name is pretty self-explanatory, use to store common constants of the project.

## Utils.tsx

* Contains utility functions.
* **getParent** & **getParentElement**: get the parent element of the provided DOM node.
* **getUserImage**: get the avatar of the user that has the provided email.
* **getReactFromDOM**: get the React instance of the provided DOM node.
* **log**: use for logging.
* **globalCache** functions: utility functions for making/creating/storing cache key.

## Model folder

* Contains basic models for SharePoint ListItem

## Styles/variable.scss

* Widely used variables for the SCSS modules should be put here.
* Currently, this file has variables and mixins used for The Global Fund project.

## Services/BaseListService.ts

* **BatchBodyBuilder**: used when calling batch requests to SharePoint API.
* **BaseListService**: the abstract class that contains basic functionality to interact with SharePoint List API. Future implementation of any service that requires interaction with SharePoint List should extend this class.

## Services/ConfigurationService.ts

* Contains methods that help get ListItems (or Configuration Value) from the ‘Configurations’ list from the config site.
* Both list name and site URL are configurable.

## Services/NavService.ts

* Navigation service.
* Some methods help to execute callbacks when page navigated (detailed in SharePoint extension section).
* Methods to interact with the current URL address.
* Declaration of the environment variables and configurations.

## Services/PeoplePickerService.ts

* Used for any SharePoint group/user related API.
* **searchPeople**: is used for searching people and integrate with **PeoplePicker** control of UI Fabric.

## Services/ServiceFactory.ts

* Implementation of Service Factory Pattern.
* Any usage of any service instance should be gotten from here instead of initializing a new instance of that service.
* It is not required when using *NavService* since it is static.

# Example implementation of SPFx webpart/extension

*This section only shows notable points of the example. For detailed implementation, please read the code in sp-training folder.*

## Extension: Project Information App Customizer

* Example implementation in folder src/extensions/projectInformation.
* **ProjectInformation** component subscribes to **currentProject$** subject in **ProjectService** to get the project data. The reason that a subscription is needed is when we navigate to other page, App Customizer is not re-rendered => project data is not reloaded with new information.
* **ProjectInformation** component extends **BaseReactiveComponents** to handle subscriptions easily.
* Quicklaunch (left navigation) is also customized using CSS in file **global.module.scss**. Because fonts in SPFx is partially loaded (only load what you render) so in **ProjectInformationApplicationCustomizer** there is some elements are rendered to load font-face.

## Webpart Project Team Members

* Example implementation in folder src/webparts/projectTeamMembers.
* This webpart will reuse the project information that loaded on the extension (ProjectInformation).
* **TeamMember** component is a functional component, which without using hooks, does not have state.

# PnP Provisioning

Some useful cmdlets:

* [Get-PnPProvisioningTemplate](https://docs.microsoft.com/en-us/powershell/module/sharepoint-pnp/get-pnpprovisioningtemplate?view=sharepoint-ps): get pnp provisioning template of the current connected site.
* [Read-PnPProvisioningTemplate](https://docs.microsoft.com/en-us/powershell/module/sharepoint-pnp/read-pnpprovisioningtemplate?view=sharepoint-ps): read the pnp provisioning template, can be used to verify if the template has any syntax error.
* [Apply-PnPProvisioningTemplate](https://docs.microsoft.com/en-us/powershell/module/sharepoint-pnp/apply-pnpprovisioningtemplate?view=sharepoint-ps): apply pnp provisioning template to the current connected site.

Tips:

* When get/modify template, remember to remove ‘Version’ attributes in all field definitions since it may cause issues when we re-apply the template to a site.
* There are a lot of versions for the template schema, we can find the version at the beginning of the template:  
    
  <pnp:Provisioning xmlns:pnp="http://schemas.dev.office.com/PnP/2020/02/ProvisioningSchema">   
    
  And review if the properties we used in the template are available [here](https://github.com/pnp/PnP-Provisioning-Schema).
* We can also use tokens to represent some values in pnp template such as: site collection id, generate guid, current user id, associated member group… List of available tokens is provided [here](https://github.com/pnp/PnP-Sites-Core/blob/master/Core/ProvisioningEngineTokens.md).

# Advanced concepts

## CSS Module with SCSS

SPFx uses CSS Module with SCSS for stylesheet implementation. The purpose of using it is to encapsulate the style of the module, ensure it does not affect other modules, and is not affected by other modules. When bundling, SCSS modules are compiled and mapping files will be generated. Mapping files contain Javascript objects which contain the css classes that are appended by auto-generated hashes.

The concept is quite hard to explain in words. Let’s see an example:

Button.module.css

.error {

background-color: red;

}

another-stylesheet.css

.error {

color: red;

}

Button.js

*import* React, { Component } *from* 'react';

*import* styles *from* './Button.module.css'; *// Import css modules stylesheet as styles*

*import* './another-stylesheet.css'; *// Import regular stylesheet*

*class* Button *extends* Component {

render() {

*// reference as a js object*

*return* <button className={styles.error}>Error Button</button>;

}

}

Result  
No clashes from other .error class names

*<!-- This button has red background but not red text -->*

<button class="Button\_error\_ax7yz">Error Button</button>

Button\_error \_ax7yz: the class that is generated.

Reference: <https://github.com/css-modules/css-modules>

## Gulp and Webpack

SPFx uses Gulp as a build tool. Depend on the use cases, we can add our custom build task by writing it in the **gulpfile.js** file. We can furtherly extend the SPFx build rig by adding/merging webpack configure.

Example of some custom build tasks:

* <https://thechriskent.com/2017/06/21/extending-sharepoint-framework-build-tasks/>
* <https://docs.microsoft.com/en-us/sharepoint/dev/spfx/toolchain/extending-webpack-in-build-pipeline>
* The Global Fund project gulpfile: **tgfgulpfile.js**