

**NeMo**

**Technical Governance Document V1.0**

****

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Prepared By** | **Reviewed By** | **Approved By** |
| **Name** | Anil Krishna Madhira |  |  |
| **Role** |  |  |  |
| **Signature** |  |  |  |
| **Date** |  |  |  |

Table of Contents

[1.0 Introduction 4](#_Toc534212196)

[1.1 Purpose of this document 4](#_Toc534212197)

[1.2 Scope 4](#_Toc534212198)

[1.3 Intended Audience 4](#_Toc534212199)

[1.4 Acronyms 4](#_Toc534212200)

[2.0 User Stories Implementation 5](#_Toc534212201)

[2.1.1 Definition of Ready 5](#_Toc534212202)

[2.1.1 Definition of Done 6](#_Toc534212203)

[3.0 Application type and design 7](#_Toc534212204)

[3.1 OneApp 7](#_Toc534212205)

[3.1.1 Design 7](#_Toc534212206)

[3.2 Web Application Desgin 7](#_Toc534212207)

[3.2.1 Purpose of the Design 7](#_Toc534212208)

[3.2.2 Design 7](#_Toc534212209)

[4.0 Application Flow (With API) 8](#_Toc534212210)

[4.1 Modules 10](#_Toc534212211)

[4.2 Components 10](#_Toc534212212)

[4.3 Component Life cycle 11](#_Toc534212213)

[4.4 Forms 13](#_Toc534212214)

[4.5 Redux 14](#_Toc534212215)

[4.6 Services 16](#_Toc534212216)

[4.7 Routing 16](#_Toc534212217)

[5.0 Front End Design 16](#_Toc534212218)

[5.1 User Interface Design 16](#_Toc534212219)

[5.2 Project Structure Definition 16](#_Toc534212220)

[6.0 Consolidated API View 18](#_Toc534212221)

[6.1 Request – Response Mapping 18](#_Toc534212222)

[6.2 Requirement – Solution Mapping 18](#_Toc534212223)

[7.0 Coding Check list 19](#_Toc534212224)

[7.1 Code Review & Traceability Template 20](#_Toc534212225)

[8.0 Testing 20](#_Toc534212226)

[9.0 Branching and Merging Strategy 20](#_Toc534212227)

[10.0 Build & Deployment Strategy 21](#_Toc534212228)

[11.0 References 21](#_Toc534212229)

[12.0 Change History 22](#_Toc534212230)

# Introduction

## Purpose of this document

The purpose of this document to provide development governance for Amex Nemo Project.

## Scope

The document covers the following:

1. Project execution model
2. Definition of Ready for the User Stories
3. Definition of Done for the User Stories
4. System architecture (Need to get it from Amex)
5. Application architecture
6. Tools, Technologies and Framework
7. Assumptions and dependencies
8. Approach towards Non Functional Requirement
9. Design Guidelines with patterns and approaches
10. Coding Guidelines with checklist
11. Review process with templates
12. Unit Testing strategy
13. Code Promotion: Branching and Merging Strategy
14. Build and deployment strategy

## Intended Audience

* Development Team
* Architect
* Scrum Master
* Product Owner

## Acronyms

|  |  |
| --- | --- |
| NFR | Non Functional Requirement |
| DOF | Definition of Ready |
| DOD | Definition of Done |
| PO | Product Owner |
| SM | Scrum Master |
| SLA | Service Level Agreement |
| CR | Change Request |
| CI | Continues Integration |
| CD | Continues Development |

Project Execution Process

### Scrum activities

|  |  |  |
| --- | --- | --- |
| **Activities** | **Participants** | **Frequency** |
| Technical Scrum | Scrum Team, SM(optional) | Daily |
| Daily Standup Meeting | SM, Scrum team | Daily |
| Sprint Planning | PO, SM, Scrum team | Once in Sprint |
| Sprint Review | SM, Scrum team | Once in Sprint |
| Sprint Retrospective | SM, Scrum team | Once in Sprint |

Sprint duration: 2 weeks (10 Working days)

Sprint start day: Alternate Tuesday

### Definition of Ready – User Story

|  |  |
| --- | --- |
| **#** | **Criteria** |
| 1 | User Story clearly defined in Rally or/and in Enterprise Confluence tool |
| 2 | User Story dependencies should be identified – Like inter story dependency |
| 3 | Effort required to implement User Story should be defined by Nemo Development team. |
| 4 | All NFR should be defined before we start the user story |
| 5 | The API from the backend Server are available and Service mapping for the API fields and User Interface fields is completed. |
| 6 | If the API is not available, Swagger yaml file should be available as part of the User Story and if mock data to be used it has to be agreed with PO. |
| 7 | Need to define for how we will accommodate the mandatory CR, design changes during the sprint which will impact the on-going sprint |
| 8 | Clarifications/Query SLA should be defined in advance to avoid impact on the sprint |

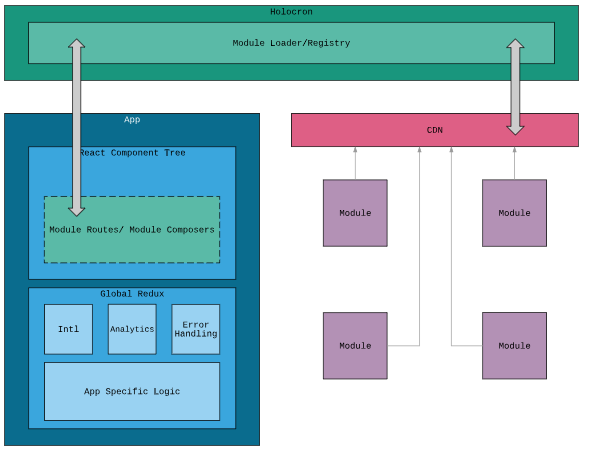
### Definition of Done – User Story

|  |  |
| --- | --- |
| **#** | **Criteria** |
| 1 | Completed User Story meets all of the acceptance criteria described in Rally and Confluence. |
| 2 | 100% Unit Test coverage.  Visible to everyone via dashboard or Jenkins |
| 3 | No open P1/P2/S1/S2 defects. |
| 4 | Peer code review using  Git Pull Requests is completed. |
| 5 | The code is executable based on the e1 environment data or based on the mock data, whichever is available at the ‘Ready’ state of the Sprint. |
| 6 | Integrated functional testing should be done from the e1 environment for the quality approval |
| 7 | End of each sprint, we should have lesson learned/retrospective meeting with entire team |
| 8 | Show and Tell of the sprint(s) : There should be a demo session for each two sprints to Business/Customer from dev team |

# Application type and design

## Amex Framework – One App

One App is a platform for American Express React components. It will provide us an established pattern, development approach, approved tools and 3rd party libraries, deployment process and testing strategy. As per the One App framework, all ReactJs based project UI component should follow the below architecture

Please refer to [One App](https://one-dev.aexp.com/v2/guide/#One-App) Amex application details in the URL below:

<https://one-dev.aexp.com/v2/guide/#One-App>

## Web Application Desgin

### Purpose of the Design

The purpose of the design document is to have uniform flow regardless of the User Story and Sprint across all the developers.

Without a proper guide and design document, there is a chance that each individual developer may implement the User Story in his own way and this may lead to accumulating technical debt.

The objectives here are:

1. Separate the Business Logic and User Interface
2. Uniform way of implementing the Web Application features across the sprints and User Stories.

.

## Modules

The application should have a separation of concern with clear division of business logic and view logic.

React Components: We can divide as many no of component to improve our application performance. Build encapsulated components that manage their own state, then compose them to make complex UIs. It helps to create reusable components.

It is always advisable to use functional components, where the main purpose of the Component is to take the props and convert it into UI.

Redux: Redux stores all the global data and data is sent to React Components as props, so the data in the View layer is displayed using the React Components.

## Components

Common Components are placed under .src/common Components folder and required to include index.js file so we can export all components available under one file.

Notes:

* Please do not include component state management and manage through props only
* Please do not include redux functionality
* Please do not use redux state directly
* Please include prop-types clearly and do not use module specific names.
* Please export component everything through index.js page.

------------------------------------

import React from 'react';

import ReactDOM from 'react-dom';

import PropTypes from 'prop-types';

class SimpleComponent extends React.Component {

render() {

      return (

        <div key={`${key}-${disable}`} id={id}>

         {/\*Key Logic here\*/}

      </div>

      );

   }

}

SimpleComponent.propTypes = {

   type: PropTypes.string.isRequired,

   image: PropTypes.string.isRequired,

   disable: PropTypes.bool.isRequired,

   id: PropTypes.string.isRequired,

   backGroundColor: PropTypes.string,

   onHover: PropTypes.func,

};

export default SimpleComponent;

--------------------------------------

Higher Order Component: If one more components do the similar activity and following the same code pattern, then there could be a chance to create a Higher Order Component

Higher Order Components take a component and enhances and add a new functionality and return a new component.

## Component Life cycle

Mounting

* [constructor()](https://reactjs.org/docs/react-component.html#constructor)
* [static getDerivedStateFromProps()](https://reactjs.org/docs/react-component.html#static-getderivedstatefromprops)
* [render()](https://reactjs.org/docs/react-component.html#render)
* [componentDidMount()](https://reactjs.org/docs/react-component.html#componentdidmount)

Note: These methods are considered legacy and you should [avoid them](https://reactjs.org/blog/2018/03/27/update-on-async-rendering.html) in new code:

[UNSAFE\_componentWillMount()](https://reactjs.org/docs/react-component.html#unsafe_componentwillmount)

Updating

* [static getDerivedStateFromProps()](https://reactjs.org/docs/react-component.html#static-getderivedstatefromprops)
* [shouldComponentUpdate()](https://reactjs.org/docs/react-component.html#shouldcomponentupdate)
* [render()](https://reactjs.org/docs/react-component.html#render)
* [getSnapshotBeforeUpdate()](https://reactjs.org/docs/react-component.html#getsnapshotbeforeupdate)
* [componentDidUpdate()](https://reactjs.org/docs/react-component.html#componentdidupdate)

Note:

These methods are considered legacy and you should [avoid them](https://reactjs.org/blog/2018/03/27/update-on-async-rendering.html) in new code:

[UNSAFE\_componentWillUpdate()](https://reactjs.org/docs/react-component.html#unsafe_componentwillupdate)

[UNSAFE\_componentWillReceiveProps()](https://reactjs.org/docs/react-component.html#unsafe_componentwillreceiveprops)

Unmounting

* [componentWillUnmount()](https://reactjs.org/docs/react-component.html#componentwillunmount)

Error Handling

* [componentDidCatch()](https://reactjs.org/docs/react-component.html#componentdidcatch)

A few things generally follow in component lifecycle

componentWillUnmount()

* This method is called before a component is unmounted from the DOM.
* It is a good place to perform cleaning operations like
* Removing event listeners.
* Clearing timers.
* Stopping sockets.
* Cleaning up redux states.

componentDidMount() (ES5 and ES6)

* The component has been mounted and you are now able to access the component's DOM nodes, e.g. via refs.
* This method should be used for
* Preparing timers
* Fetching data
* Adding event listeners
* Manipulating DOM elements

## Forms

Based on the requirement we can define forms/components in to

* Stateless components 🡪 Preferred way where we don’t have to track any user/state information
* State full components 🡪 Scenarios where we need to track user/state information
* HOC Components 🡪 For common functionality for multiple components like showing loader symbol.
* Pure Components 🡪 We can use to reduce extra re-renders

Every form component which need to be designed/build, need to be viewed from the below guidelines:

* 🚀 Lightweight and fast.
* 🔥 Built-in input primitives for building quickly
* 🚚 Easily integrate with 3rd party components or build your own!
* ✍️ Nested Fields and ultra-composable syntax for complex form shapes.
* ⏲ Asynchronous validation
* 😂 Works in IE (with a polyfill or two) and other browsers

class CustomForm extends React.Component {

constructor() {

super();

this.state = {

person: {

firstName: '',

lastName: ''

}

}

}

handleChange(event) {

let person = this.state.person;

person[event.target.name] = event.target.value;

this.setState({person});

}

render() {

return (

<form>

<input

type="text"

name="firstName"

value={this.state.firstName}

onChange={this.handleChange.bind(this)} />

<input

type="text"

name="lastName"

value={this.state.lastName}

onChange={this.handleChange.bind(this)} />

</form>

)

}}

In this above example,

* we initialize state with an empty person object.
* We then bind the values of the 2 inputs to the individual keys of the person object.
* Then as the user types, we capture each value in the handleChange function.
* Since the values of the components are bound to state we can rerender as the user types by calling setState().

NOTE: Not calling setState() when dealing with controlled components, will cause the user to type, but not see

* the input because React only renders changes when it is told to do so.
* It's also important to note that the names of the inputs are same as the names of the keys in the person object.
* This allows us to capture the value in dictionary form as seen here.

handleChange(event) {

let person = this.state.person;

person[event.target.name] = event.target.value;

this.setState({person});

}

## Redux

We can maintain redux in two ways

1. Global state used for across all modules
2. Module or component based state.

### 

Action

React JS

Components

Redux-thunk Middleware

Store

Reducers

Network Util

Api

Updated State

Updated State

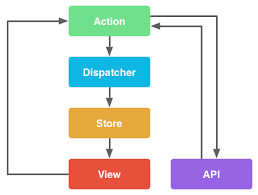
View : The View components are React functional Components.

Reducers: Reducers take the actions are update the application state.

Actions: Actions provides a way to provide the type of action and payload to the Store.

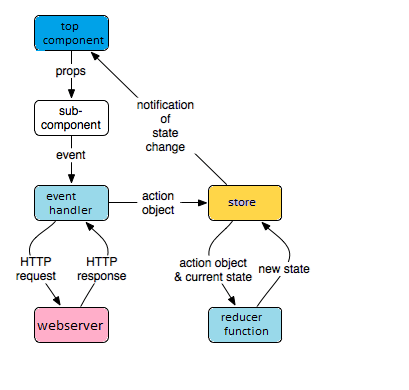
Dispatch: All actions are dispatched to the redux store using the dispatch function.

Async Operations: All the async operations like API, are handled by Redux-thunk Middleware



Network Util

For handling asynchronous data ‘redux-saga’ library can be used. This library solely responsible for handling side effects. This is a redux middleware, which means this can be started, paused, cancelled from the normal application using generator functions provided by redux



Actions are payloads of information that send data from your application to your store. They are the only source of information for the store. You send them to the store using [store.dispatch()](https://redux.js.org/api-reference/store#dispatch).

Here's an example actions.

**Actions.js**

import { HOME\_PAGE\_LOADED\_REQUEST, HOME\_PAGE\_LOADED\_SUCESS } from '../constants/home-constants';

function loadhomepageSucess(subreddit) {

return {

type: HOME\_PAGE\_LOADED\_SUCESS,

subreddit,

data:[],

}

}

export default loadhomepageSucess ;

**Reducers .js file**

import {

HOME\_PAGE\_LOADED\_REQUEST,

HOME\_PAGE\_LOADED\_SUCESS,

HOME\_PAGE\_LOADED\_FAILURE

} from '../constants/home-constants';

export default (state = {

isRequest: false,

isSuccess: false,

isError: false,

data: [],

errorCode: ''

}, action) => {

switch (action.type) {

case HOME\_PAGE\_LOADED\_SUCESS:

return {

...state,

isSuccess: true,

isRequest: false,

data: action.data

};

case HOME\_PAGE\_LOADED\_REQUEST:

return {

...state,

isRequest: true,

data: []

};

case HOME\_PAGE\_LOADED\_FAILURE:

return {

...state,

isRequest: false,

errorCode: action.errorCode

};

default:

return state;

}

};

## Services

The API services are provided with separate folder structure and components may call or trigger the corresponding event to load the data.

* Axios is a JavaScript library used to make http requests from node.js or XMLHttpRequests from the browser and it supports the Promise API that is native to JS ES6
* Second feature that it has over fetch() is that it performs automatic transforms of JSON data

We need a API layer for making service call and this need to be generic component which need to be followed by everyone.

## Routing

Routing in Amex Framework is achieved using HoloCron Module.

# Amex NeMo Front End Design

## User Interface Design

For each layout there will be Visual Design diagram and to provide the responsive design for any other screen resolutions specific breakpoints should be provided. These breakpoints will be used as part of media queries in the CSS.

SCSS should be used as CSS preprocessor.

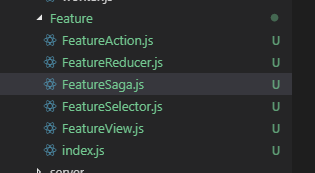
Reusable CSS class should be moved to global CSS file and that must be part of “styles”.

## Development specification

|  |  |
| --- | --- |
| Development environment editor | Visual Studio Code |
| Development library | ReactJS |
| UI unit testing framework | Jest |
| Module bundler | Webpack 4.8.3 |
| HTTP client api | fetch |
| CSS pre-processor | SAAS |
| Scripting standard/type | ES6 |
| State management | Aurelia-redux |
| styles | Css3 |
| Validation | Aurelia-validation |
| Responsive page | bootstrapper |

## Project Structure Definition

The project structure is based on the Feature. For each Feature, we can have a Reducer, Selector, View and actions. This way folder structure can be scalable and each feature may contain a similar folder structure, which brings the uniformity.



* FeatureAction.js : All the Action functions for reducer.
* FeatureReducer.js: Reducer for Feature
* FeatureSaga.js : Sagas for handling async operations
* FeatureSelector: Selector functions part for reducer
* FeatureView: Only View or part of container with props and dispatch functions
* Index.js : Grouping all the files and giving a feature structure to be independent module

//For feature index.js

// Actions

const LOAD = 'my-app/featureAction/LOAD';

const CREATE = 'my-app/featureAction /CREATE';

const UPDATE = 'my-app/featureAction /UPDATE';

const REMOVE = 'my-app/featureAction /REMOVE';

// Reducer

export default function reducer(state = {}, action = {}) {

switch (action.type) {

// do reducer stuff

default: return state;

}

}

// Action Creators

export function loadUserData() {

return { type: LOAD };

}

export function createUserProfile(userData) {

return { type: CREATE, userData };

}

export function update(userData) {

return { type: UPDATE, userData };

}

export function remove(userData) {

return { type: REMOVE, userData };

}

// side effects, only as applicable

// e.g. saga

export function getUsers () {

return dispatch => get('/users).then(users => dispatch(updateWidget(userData)))

}

# Consolidated API View

## Request – Response Mapping

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Screen Name** | **Screen Element** | **Service Name** | **Type of Service Operation** | **Request Field** | **Response Field** | **Remarks** |
| <An unique name / identifier for the screen> | <Identify an element on the screen which gets mapped to a particular service response field / an element that triggers a service invocation> | <Specify the name of the service> | <Specify if is a GET operation or POST operation> | <For POST operation, map the screen element with service element> | <For GET operation, map the screen element with service element> | <Optionally, add any relevant comments> |

## Requirement – Solution Mapping

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Requirement Id** | **Requirement Description** | **High Level Design Approach** | **Associated Files** | **Remarks** |
| <Refer to the requirement id from requirement document> | <A high level description of the requirement> | <A high level overview of the solution approach> | <Refer to files from both front end / back end> | <Optionally, add any relevant comments> |

# Coding Check list

|  |  |  |
| --- | --- | --- |
| 1 | All violations coming out of esLint have been fixed | Yes |
| 2 | All Components must have proptypes | Yes |
| 3 | All the li components are with a key value | Yes |
| 4 | Minimum code in the render method and logic should be in helper methods. | Yes |
| 5 | Split the React Component till it handles single responsibility | Yes |
| 6 | No editing of props within the React Component | Yes |
| 7 | Name the boolean variables and functios starting with "is", "has", "should" etc.. | Yes |
| 8 | Use the default values in the function. Use these default values , whereever applicable. Set defaultprops, where for props of component, if applicable. | Yes |
| 9 | Use spread operators instead of Object.assign | Yes |
| 10 | Change all var and let to const, whereever possible. | Yes |
| 11 | No mutation of state in case of redux reducer | Yes |
| 12 | Remove commented code. No code should exist which is commented in the file. | Yes |
| 13 | For each file corresponding test file exists. | Yes |
| 14 | Each component should be a separate file. | Yes |
| 15 | Use ErrorBoundary component where applicable and show corresponding error message. | Yes |
| 16 | Use of constants instead of hard coded values. | Yes |
| 17 | NamingConventions for Classes, filenames are followed | Yes |
| 18 | Proper folder structure is followed. | Yes |
| 19 | Don't use this.state, inside a setState function Call setState, with a function and previousState as a varialbe. this.setState( (prevState,props) => { return ( {value: prevSate.value + props.value } )} | Yes |
| 20 | Intialize the state in the component with expected default values. | Yes |
| 21 | Don't use ComponentWillMount, instead of use ComponentDidMount.  "ComponentWillMount" will be deprecated. | Yes |
| 22 | Don't call, setState inside a "shouldComponentUpdate" "componentWillUpdate", it will cause infinite loop. | Yes |
| 23 | Don't call setState inside a render method, as setState triggers render and render is updating the state. | Yes |
| 24 | Don't call setState in componentWillUnmount, as the component is unmounting anyway. | Yes |
| 25 | The best place for setState, ComponentDidMount and in the event actions based on the trigger. | Yes |

## Code Review & Traceability Template





# Testing

Tools used: Jest, for unit tests for end 2 end test

Type of test: For each sprint testing is performed on the e1 environment.

# Branching and Merging Strategy

-----------------------------------------------------------(master) -🡪

-----------------------------------------------------------(release branch) -🡪

-----------------------------------------------------------(develop branch) -🡪

The branching is created based on the features in the current sprint, so it may or may not completely align with the user stories. So if one or more team members are working on a single feature they will create a single feature branch and work on the particular branch.

The feature branch is created using the develop branch as the base. If during the sprint the develop branch is updated and with different commits, then the feature branch is rebaselined using “git rebase” command.

During the end of the sprint the feature branch is merged with the develop branch.

# Build & Deployment Strategy

Map it to Amex documentation

# References

**https://reactjs.org/**

**https://github.com/reactjs**

[**https://github.com/facebook/react**](https://github.com/facebook/react)

[**https://redux-saga.js.org/**](https://redux-saga.js.org/)

[**https://redux.js.org/**](https://redux.js.org/)

**add testing**

# Change History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Version Number | Changes Made | | | |
| V1.0 | <<First version>> | | | |
| V1.1 | <<If the change details are not explicitly documented in the table below, reference should be provided here>> | | | |
| Page No. | Changed by | Effective Date | Changes Effected |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| V1.2 | <<If the change details are not explicitly documented in the table below, reference should be provided here>> | | | |
| Page No. | Changed by | Effective Date | Changes Effected |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |