

**NeMo**

**Technical Governance Document V1.0**

****

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Prepared By** | **Reviewed By** | **Approved By** |
| **Name** | Anil Krishna & Prince Antony |  |  |
| **Role** |  |  |  |
| **Signature** |  |  |  |
| **Date** |  |  |  |

Table of Contents

[1.0 Introduction 4](#_Toc534811430)

[1.1 Purpose of this document 4](#_Toc534811431)

[1.2 Scope 4](#_Toc534811432)

[1.3 Intended Audience 4](#_Toc534811433)

[1.4 Acronyms 4](#_Toc534811434)

[1.5 Project Execution Process 5](#_Toc534811435)

[1.5.1 Scrum activities 5](#_Toc534811436)

[1.5.1 Definition of Ready – User Story 6](#_Toc534811437)

[1.5.1 Definition of Done – User Story 7](#_Toc534811438)

[2.0 Application type and design 8](#_Toc534811439)

[2.1 Amex Framework – OneApp 8](#_Toc534811440)

[2.2 Web Application design 8](#_Toc534811441)

[2.2.1 Purpose of the Design 8](#_Toc534811442)

[2.3 Modules 9](#_Toc534811443)

[2.4 Components 9](#_Toc534811444)

[2.4.1 Presentational and Container Components 9](#_Toc534811445)

[2.5 Data binding - Lazy loading approach 10](#_Toc534811446)

[2.6 Component Life cycle 11](#_Toc534811447)

[2.6.1 Mounting 11](#_Toc534811448)

[2.6.2 Updating 11](#_Toc534811449)

[2.6.3 Unmounting 11](#_Toc534811450)

[2.6.4 ErrorHandling 11](#_Toc534811451)

[2.6.5 componentWillUnmount() 11](#_Toc534811452)

[2.6.6 componentDidMount() (ES5 and ES6) 12](#_Toc534811453)

[2.7 Forms 12](#_Toc534811454)

[2.8 Redux 13](#_Toc534811455)

[2.9 Services 15](#_Toc534811456)

[2.10 Routing 15](#_Toc534811457)

[3.0 Amex NeMo 15](#_Toc534811458)

[3.1 User Interface Design 15](#_Toc534811459)

[3.2 Development specification 15](#_Toc534811460)

[3.3 Project Structure 16](#_Toc534811461)

[4.0 Consolidated API View 16](#_Toc534811462)

[4.1 Request – Response Mapping 16](#_Toc534811463)

[4.2 Requirement – Solution Mapping 17](#_Toc534811464)

[5.0 Coding Check list 17](#_Toc534811465)

[5.1 Code Review & Traceability Template 18](#_Toc534811466)

[6.0 Testing 18](#_Toc534811467)

[6.1 Unit Testing 18](#_Toc534811468)

[6.2 Important References for Unit Testing 18](#_Toc534811469)

[7.0 Code Coverage 19](#_Toc534811470)

[8.0 Code Quality 19](#_Toc534811471)

[8.1 Code Analysis tools 19](#_Toc534811472)

[8.2 ESLint 19](#_Toc534811473)

[8.3 source-map-explorer 20](#_Toc534811474)

[8.4 Coding Standards 20](#_Toc534811475)

[9.0 Branching and Merging Strategy 21](#_Toc534811476)

[10.0 Build & Deployment Strategy 21](#_Toc534811477)

[11.0 References 21](#_Toc534811478)

[12.0 Change History 21](#_Toc534811479)

# 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 |

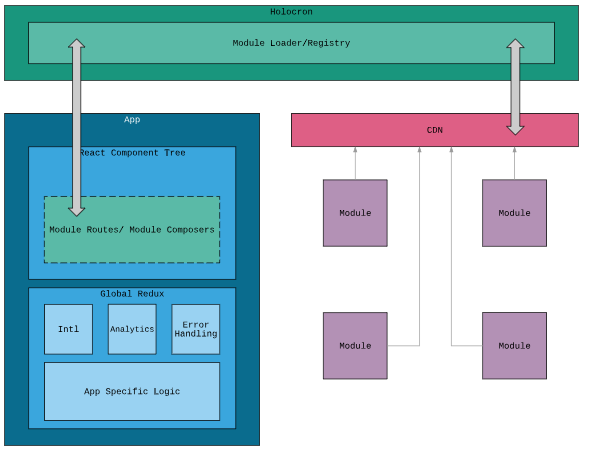
### Definition of Done – User Story

|  |  |
| --- | --- |
| **#** | **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 showstopper defects. |
| 4 | Peer code review using  Bit Bucket 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 [OneApp](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 design

### 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:

* Separate the Business Logic and User Interface
* 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.

### Presentational and Container Components

One of the most common patterns to organize our React code is to [split them into presentational and container components](https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0), where containers provide the data needed to render the presentational components. This means that the presentational components don’t have direct access to your Redux store or actions. Instead, they are passed down as props from their containers.

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

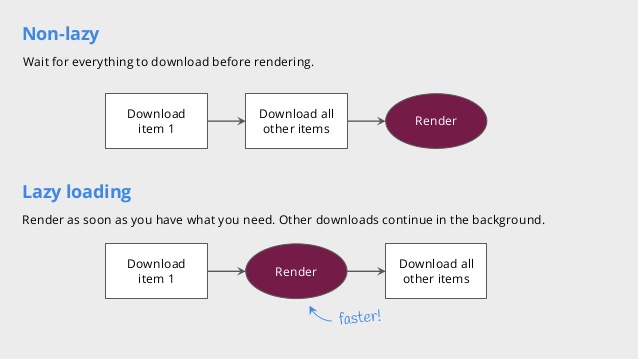
Higher Order Components take a component and enhances and add a new functionality and return a new component.

## Data binding - Lazy loading approach

The main aim of lazy loading, render downloaded items later download or fetch remaining items, this loop will be continuing till the end. For example, below diagram we have two variations

* Non-Lazy
* Lazy loading

In Lazy loading, first required components/modules will be downloaded and corresponding render function will be invoked. Later same cycle will be repeated.



## 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)

### ErrorHandling

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

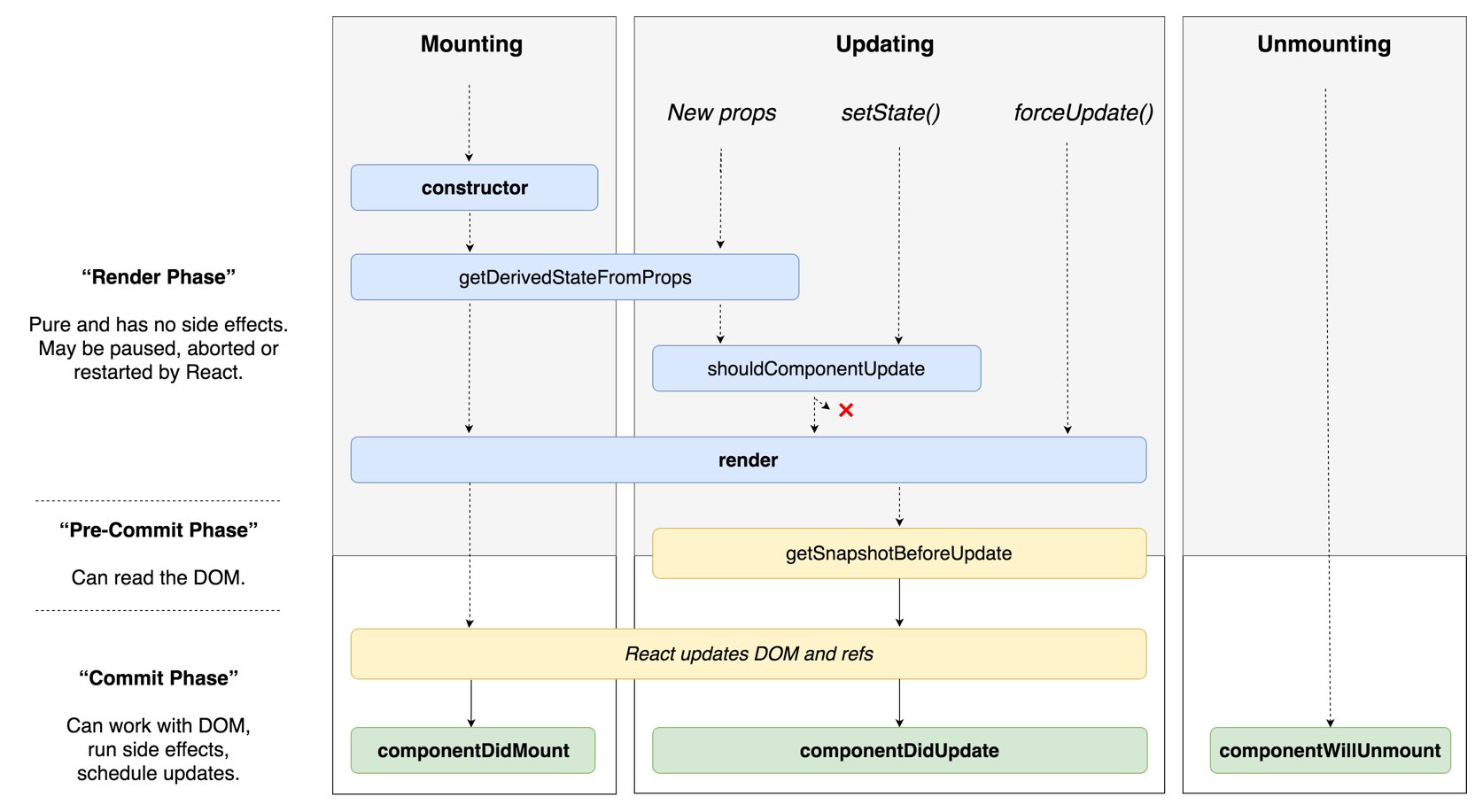
A few things generally follow in component lifecycle

### 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
* 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-compostable syntax for complex form shapes.
* Asynchronous validation
* Works in IE (with a Polyfill or two) and other browsers

## Redux

We can maintain redux in two ways

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

**

React Components : The functional , typed and presentations components(Components & Containers)

**Redux**: An architectural predictable for the application state management.

**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.

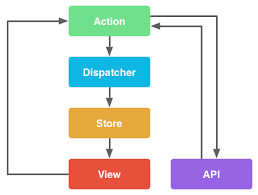
**Reducers**: Specify how the application's state changes in response to actions sent to the store

**Store**: To hold application state

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

**Network Util** : It is the wrapper node.js script to connect APIs from React application

**Exception/Error** : All errors will push to local couch database for further verification



For handling asynchronous data ‘redux-thunk’ 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.

## 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 generic API layer for making service call(Network Util) from the react application which will avoid redundant code in the service layer

## Routing

Routing in Amex Framework is achieved using HoloCronModule.

# Amex NeMo

## 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

|  |  |
| --- | --- |
| **Open Source** | |
| Development environment editor | Visual Studio Code |
| Development library | ReactJS |
| UI unit testing framework | Jest |
| Module bundler | Webpack |
|  |  |
| HTTP client api | fetch |
| CSS pre-processor | SAAS |
| Scripting standard/type | ES6 |
| Styles | Css3 |
| Responsive page | Bootstrapper |
| Utility library | Lodash |
| Software Configuration tool | Bitbucket |

|  |  |
| --- | --- |
| **Amex In House** | |
| Client State management | axp-global-ducks, Holocron |
| Reusable components | axp-base |
| Module bundler | one-amex-bundler |
| Router | [holocron-module-route](https://stash.aexp.com/stash/projects/ONE/repos/holocron/browse/packages/holocron-module-route) |
|  |  |

## Project Structure

To be updated with proposed project structure

# 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> | GET/ POST/ PUT/ DELETE | <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 list 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 starts with "is", "has", "should" etc.. | Yes |
| 8 | Use the default values in the function. Use these default values , wherever 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 variable. 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 useComponentDidMount.  "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

## Unit Testing

We need to write at least basic unit tests to make sure regressions don't spring up because some new developer doesn't understand what's going on. The core units of the application should be verified with accompanying unit tests. In JavaScript apps, the smallest units of code we can test are usually individual functions.

Each developer has to verify and make sure his/her chunk of code works perfectly as intended. So, when all the small chunks of code are joined together, they have a good chance of working as a whole.

**Jest** is preferred to do the Unit Testing.[Facebook's Jest](https://facebook.github.io/jest/docs/en/tutorial-react.html) hand walks the developer by explaining its implementation.

## Important References for Unit Testing

|  |  |
| --- | --- |
| * Jest Unit Testing | * <https://facebook.github.io/jest/docs/en/tutorial-react.html> |
| * Jest Unit Testing * Best Practices | * <https://facebook.github.io/jest/docs/en/tutorial-react.html> |

# Code Coverage

One of the important things in the phase of Unit Testing is Code Coverage. It is a measure used to describe the degree to which the source code of a program is executed when a particular test suite runs. A program with high code coverage, measured as a percentage, has had more of its source code executed during testing which suggests it has a lower chance of containing undetected software bugs compared to a program with low code coverage.

Jest’s Built-in code coverage reports: Easily create code coverage reports using [--coverage](https://facebook.github.io/jest/docs/en/cli.html). No additional setup or libraries needed! Jest can collect code coverage information from entire projects, including untested files.

# Code Quality

## Code Analysis tools

Static Code Analysis is a method of computer program debugging that is done by examining the code without executing the program. It is a collection of algorithms and techniques used to analyze source code in order to automatically find potential errors or poor coding practices. The process provides an understanding of the code structure, and can help to ensure that the code adheres to industry standards.

It commonly refers to the running of Static Code Analysis tools that attempt to highlight possible vulnerabilities like syntax errors within static source code.

It is performed as part of a Code Review that happens without the manual intervention. It is advised to carry out continuously thorough out the development of the project

## ESLint

ESLint is an open source JavaScript linting utility originally created by Nicholas C. Zakas in June 2013. Code [linting](https://en.wikipedia.org/wiki/Lint_(software)) is a type of static analysis that is frequently used to find problematic patterns or code that doesn’t adhere to certain style guidelines. There are code linters for most programming languages, and compilers sometimes incorporate linting into the compilation process.

JavaScript, being a dynamic and loosely-typed language, is especially prone to developer error. Without the benefit of a compilation process, JavaScript code is typically executed in order to find syntax or other errors. Linting tools like ESLint allow developers to discover problems with their JavaScript code without executing it.

|  |  |
| --- | --- |
| * ESLint | * [https://eslint.org](https://eslint.org/docs/rules/) |
| * ESLint Rules Explained | * <https://eslint.org/docs/rules/> |
| * ESLint can be configured as a plugin in Microsoft Visual Code, WebStorm, Sublime, and Spring Eclipse IDEs to get the errors during the development. | |

Please do configure all the rules that are recommended in the ESLint website for the better quality of the static code.

## [source-map-explorer](https://www.npmjs.com/package/source-map-explorer)

[source-map-explorer](https://www.npmjs.com/package/source-map-explorer) is great. It shows you a treemap visualization to help you debug where all the code is coming from. Thus it’s easy to analyze and debug JavaScript (or Sass or LESS) code bloat through source maps.

## Coding Standards

A good piece of code should be easy to understand and tell us what is going on — not more and not less. This can be achieved by writing sensible code by following the simple Universal Standards.

Each developer should feel accountable for his/her piece of code and deliver a good chunk of code on each day of the project that ultimately leads to a Quality Product at the end.

Coding Standards benefit both the developers and the application in the following ways.

* Suggests Syntax, Naming Conventions, Definition & Declaration of the Variables and Methods, Application Structure of the project, etc.
* Guides all the developers to follow the same set of standards across the application that keeps the code more maintainable and consistent throughout Development, Testing and Maintenance phases.
* Increases Readability, Maintainability and Quality of the code.

# 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 rebase lined 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/)

# 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 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |