# **Component Class Naming**

* **root** should be the class attached to the wrapper div of the whole component
* If a class is being used for a state - should start with **is**
  + EXAMPLE: isEmpty, isError
* div that wraps a larger chunk of component (but not the root) should be named **section**
  + EXAMPLE:
* <div className={classes.root}>  
   <div className={classes.section}>  
   <Component />  
   <Component />  
   <Component />  
   </div>  
  </div>
* Think of components like how you would layout a page: if the component has a "header" name it with a **header** class, same with footer, think of **section**  like body etc
* Wrap individual elements with a **container**
  + **EXAMPLE:**<div className={classes.root}>  
     <div className={classes.section}>  
     <div className={classes.header>  
     <HeaderComponent />  
     </div>  
     <div className={classes.nameContainer>  
     <NameComponent />  
     </div>  
     </div>  
    </div>
* For other class names try to follow BEM principles. BEM stands for Block, Element, Modifier
  + Block would be the component name but (For JSS users) you don't need to specify this because this is generated by JSS
  + Elements are things like "image", "name", "address"
  + Modifiers would be anything that would change the element: "imageLarge", "imageSmall", "buttonPrimary", "agentName" , "customerName" etc
* No abbreviations! so spell everything out img should be image, num should be number etc

# **Prop Naming**

* If it’s related to a boolean should start with "is" or "has"
  + ex: "isRequired", "isSelected", "hasElement"
* No abbreviations. spell everything out
* Avoid being too general. "text doesn't explain what is going on especially with a lot of props
* Anything related to a text should end with "text"
  + "titleText", "labelText" etc
* Try to alphabetize the prop list
* Avoid using prop names that are too similar within the same component

# **Other Prop Related Things**

* Properties that are not required should be created to be optional
  + I.e prop?: any
* Type properties as best as possible
  + If a prop should only have a few states don’t cast as it as a string create a property with those options
  + Avoid the type of any (Exception classes)
  + Ex:

export type GridXs = false| 'auto'| true| 1| 2| 3| 4| 5| 6| 7| 8| 9| 10| 11| 12;

# **Component Styling**

* All components should be functional components
  + I.e they should not extend the pure component established from react
  + Ex below:

// @flow

import React from 'react';

import cn from 'classnames';

import { ButtonColor, ButtonSize, ButtonVariant } from '../../config/prop-types';

export type Props = {

classes?: any,

color?: ButtonColor,

isDisabled?: boolean,

label: string,

size: ButtonSize,

variant: ButtonVariant,

onClick: (any) => {},

};

const Button = (props: Props) => {

const {

classes,

color,

isDisabled,

variant,

label,

size,

onClick,

...other

} = props;

return (

<button

className={cn(classes.root, className, classes[color], classes[size], classes[variant])}

type="button"

disabled={isDisabled}

onClick={onClick}

{...other}

>

{label}

</button>

);

};

Button.defaultProps = {

color: 'primary',

isDisabled: false,

};

export { Button };

* Components should follow the enhancer pattern
  + I.E each component should have a normal js file, an enhancer, a style sheet, and an index js file. This enables easier debugging of components and less coupling between components and style sheets.
    - Sub Note the enhancer file is used to inject the style sheet into a component
    - The Style sheet should export styles
    - Index.js file should export the enhanced component

# **Testing**

## **Storybook Testing**

* All components should have one or many corresponding story book tests
* All possible states of the component should be demonstrated in the test
* All props that appear in multiple test should be added as a const above
  + Ex below:

const requiredProps = () => ({

label: 'NEXT',

color: 'primary',

size: 'large',

variant: 'outline',

});

/\*\* How to import into test \*\*/

stories.add('Button with primary', () => (

<Button {...requiredProps()} />

));

* Ensure names of stories accurately describe what the story is doing
  + Sub Note whenever a component is built, make sure it doesn’t break storybook

## **Unit Testing**

* All utility functions must have corresponding jest tests with them
* Each Function should have one and only one describe
  + Each test should be represented with an it call (Make sure to utilize arrow functions for these)
  + Ex:

describe('Format Phone Numbers', () => {

it('Should handle number clumps', () => {

const mockPhone = '6667778888';

expect(formatPhoneNumber(mockPhone)).toBe('(666) 777-8888');

});

it('Should handle truncated numbers', () => {

const mockPhone = '666777888';

expect(formatPhoneNumber(mockPhone)).toBe(null);

});

it('Should handle long numbers', () => {

const mockPhone = '66677788889';

expect(formatPhoneNumber(mockPhone)).toBe(null);

});

it('Should handle null case', () => {

const mockPhone = '';

expect(formatPhoneNumber(mockPhone)).toBe(null);

});

});

* All Calls done utilizing redux should have a corresponding thunk test
* These tests should mock whatever api it is calling (Consider using axios-mock-adapter)
* After each test the mock should be reset
* Each test should validate proper response as well as actions generated by redux
* All Testing variables should be set inside of a before each
* Ex:

import configureMockStore from 'redux-mock-store';

import thunk from 'redux-thunk';

import { genericAction } from '../actions/submit-lead';

import mockSubmitLeadResponse from './context/mock-submit-lead';

import CONFIG from './context/config';

const axios = require('axios');

const MockAdapter = require('axios-mock-adapter');

const mock = new MockAdapter(axios);

const middlewares = [thunk];

mock.onGet('config.json').reply(200, {

...CONFIG,

});

describe('Generic Test', () => {

let mockStore;

let body;

let store;

afterAll(() => {

mock.reset();

});

describe('Generic Test', () => {

beforeEach(() => {

body = {

autoAssign: false,

};

mockStore = configureMockStore(middlewares);

store = mockStore({});

});

it('Should successfully do something', async() => {

mock.onPost(`${Generic\_Endpoint}`).reply(200, {

...genericResponse,

});

const expectedActions = ['GENERIC\_INITIATED', 'SET\_LOADING', 'GENERIC\_SUCCESS', 'SET\_LOADING'];

await store.dispatch(genericAction(body));

const actualActions = store.getActions().map(action => action.type);

expect(actualActions).toStrictEqual(expectedActions);

});

it('Should be able to handle if agent api is down', async() => {

mock.onPost(`${generic-endpoint`).reply(404, {

});

const expectedActions = ['Generic\_INITIATED', 'SET\_LOADING', 'Generic\_FAILURE', 'SET\_LOADING'];

try {

await store.dispatch(genericAction(body));

expect(true).toStrictEqual(false);

} catch (err) {

const actualActions = store.getActions().map(action => action.type);

expect(actualActions).toStrictEqual(expectedActions);

}

});

});

});