CONTRIBUTING

============

<!-- START doctoc generated TOC please keep comment here to allow auto update -->

<!-- DON'T EDIT THIS SECTION, INSTEAD RE-RUN doctoc TO UPDATE -->

- [Getting Started](#getting-started)

- [Fork, Clone & Install](#fork-clone--install)

- [Commit Messages](#commit-messages)

- [Commands](#commands)

- [Workflow](#workflow)

- [Create a Component](#create-a-component)

- [Using propTypes](#using-proptypes)

- [Conformance Test](#conformance-test)

- [Open A PR](#open-a-pr)

- [Spec out the API](#spec-out-the-api)

- [API](#api)

- [SUI HTML Classes](#sui-html-classes)

- [API Patterns](#api-patterns)

- [Building className](#building-classname)

- [Testing className](#testing-classname)

- [SUI HTML Markup](#sui-html-markup)

- [SUI Components vs Component Parts](#sui-components-vs-component-parts)

- [React Components & Sub Components](#react-components--sub-components)

- [Component Part Props](#component-part-props)

- [Testing](#testing)

- [Coverage](#coverage)

- [Common Tests](#common-tests)

- [Usage](#usage)

- [isConformant (required)](#isconformant-required)

- [State](#state)

- [AutoControlledComponent](#autocontrolledcomponent)

- [Documentation](#documentation)

- [Website](#website)

- [Components](#components)

- [Props](#props)

- [Examples](#examples)

- [Releasing](#releasing)

<!-- END doctoc generated TOC please keep comment here to allow auto update -->

## Getting Started

Make sure you have at least [Node.js v6][11]:

```sh

node -v

v6.2.1

```

### Fork, Clone & Install

Start by [forking Semantic UI React][12] to your GitHub account. Then clone your fork and install dependencies:

```sh

git clone git@github.com:<your-user>/Semantic-UI-React.git

cd Semantic-UI-React

yarn

```

>Note: we use `yarn` and advise you do too while contributing. Get it [here](https://yarnpkg.com/). You can use `npm install / npm ci` but we don't include a `package-lock.json` in the repository, so you may end up with slightly out of sync dependencies.

Add our repo as a git remote so you can pull/rebase your fork with our latest updates:

```

git remote add upstream git@github.com:Semantic-Org/Semantic-UI-React.git

```

### Commit Messages

Please follow the [Angular Git Commit Guidelines][8] format.

### Commands

>This list is not updated, you should run `yarn run` to see all scripts.

```sh

yarn start // run doc site

yarn ci // run all checks CI runs

yarn test // test once

yarn test:watch // test on file change

yarn build // build everything

yarn build:dist // build dist

yarn build:docs // build docs

yarn build:docs-toc // build toc for markdown files

yarn deploy:docs // deploy gh-pages doc site

yarn lint // lint once

yarn lint:fix // lint and attempt to fix

yarn lint:watch // lint on file change

```

## Workflow

- [Create a Component](#create-a-component)

- [Conformance Test](#conformance-test)

- [Open A PR](#open-a-pr)

- [Spec out the API](#spec-out-the-api)

### Create a Component

Create components in `src`. The directory structure follows SUI naming conventions. If you're updating a component, push a small change so you can open a PR early.

Stateless components should be written as a `function`:

```js

function Button(props) {

// ...

}

```

Stateful components should be classes:

```js

import { AutoControlledComponent as Component } from '../../lib'

class Dropdown extends Component {

// ...

}

```

>You probably need to extend our [`AutoControlledComponent`](#autocontrolledcomponent) to support both [controlled][2] and [uncontrolled][3] component patterns.

### Using propTypes

Every component must have fully described `propTypes`.

```js

import React, { PropTypes } from 'react'

function MyComponent(props) {

return <div className={props.position}>{props.children}</div>

}

MyComponent.propTypes = {

children: PropTypes.node,

position: PropTypes.oneOf(['left', 'right']),

}

```

### Conformance Test

Review [common tests](#common-tests) below. You should now add the [`isConformant()`](#isconformant-required) common test and get it to pass. This will validate the `\_meta` and help you get your component off the ground.

### Open A PR

This is a good time to open your PR. The component has been created, but the API and internals are not yet coded. We prefer to collaborate on these things to minimize rework.

This will also help with getting early feedback and smaller faster iterations on your component.

### Spec out the API

Review the SUI documentation for the component. Spec out the component's proposed API. The spec should demonstrate how your component's API will support all the native SUI features. You can reference this [API proposal][7] for the Input.

Once we have solidified the component spec, it's time to write some code. The following sections cover everything you'll need to spec and build your awesome component.

## API

The primary areas of focus when designing a component API are:

1. [SUI HTML Classes](#sui-html-classes)

1. [SUI HTML Markup](#sui-html-markup)

Our goal is to map these to a declarative component API. We map HTML classes to component props. We map markup to sub components (and sometimes props).

### SUI HTML Classes

SUI component definitions (style and behavior) are defined by HTML classes. These classes can be split into 4 groups:

1. Standalone &mdash; `basic` `compact` `fluid`

1. Pairs &mdash; `left floated` `right floated`

1. Mixed &mdash; `corner` `top corner`, `padded` `very padded`

1. Groups &mdash; sizes: `tiny` `small` `big`, colors: `red` `green` `blue`

Each group has an API pattern and prop util for building up the `className` and a [Common test](#commont-tests).

#### API Patterns

```js

<Segment basic /> // standalone

<Segment floated='left' /> // pairs

<Segment padded /> // mixed

<Segment padded='very' />

<Segment size='small' color='red' /> // groups

```

```html

<div class="ui basic segment"></div>

<div class="ui left floated segment"></div>

<div class="ui padded segment"></div>

<div class="ui very padded segment"></div>

<div class="ui small red segment"></div>

```

#### Building className

Use [`classNameBuilders`][4] to extract the prop values and build up the `className`. Grouped classes like `color` and `size` simply use the prop value as the `className`.

```js

import cx from 'clsx'

import { useKeyOnly, useValueAndKey, useKeyOrValueAndKey } from '../../lib'

function Segment({ size, color, basic, floated, padded }) {

const classes = cx(

'ui',

size,

color,

useKeyOnly(basic, 'basic'),

useValueAndKey(floated, 'floated'),

useKeyOrValueAndKey(padded, 'padded'),

'segment'

)

return <div className={classes} />

}

```

#### Testing className

Use [`commonTests`](#common-tests) to test the `className` build up for each prop. These tests will run your component through all the possible usage permutations:

```js

import \* as common from 'test/specs/commonTests'

import Segment from 'src/elements/Segment/Segment'

describe('Segment', () => {

common.propValueOnlyToClassName(Segment, 'size')

common.propValueOnlyToClassName(Segment, 'color')

common.propKeyOnlyToClassName(Segment, 'basic')

common.propKeyAndValueToClassName(Segment, 'floated')

common.propKeyOrValueAndKeyToClassName(Segment, 'padded')

})

```

### SUI HTML Markup

#### SUI Components vs Component Parts

It is important to first differentiate between \*components\* and \*component parts\* in SUI. Per the [SUI Glossary][9] for `ui`:

>`ui` is a special class name used to distinguish parts of components from components.

>

>For example, a list will receive the class `ui list` because it has a corresponding definition, however a list item, will receive just the class `item`.

The `ui header` \*component\* is not the same as a `header` \*component part\*. They share the same name but do not support the same features.

A [`ui header`][5] accepts a size class. The `ui modal` has a \*component part\* called `header`. However, the size class is not valid on the `header` \*component part\*. You size the `ui modal` \*component\* instead.

\*\*Header Component\*\*

```html

<div class="ui small header">...</div>

```

\*\*Modal Component (with header \*component part\*)\*\*

```html

<div class="ui small modal">

<div class="header">...</div>

</div>

```

#### React Components & Sub Components

Top level Semantic UI React components correspond to SUI \*components\*. Stardust sub components correspond to SUI \*component parts\*.

This allows us to provide accurate `propTypes` validation. It also separates concerns, isolating features and tests.

Use sub components to design \*component part\* markup.

```js

<List>

<List.Item>Apples</List.Item>

<List.Item>Oranges</List.Item>

<List.Item>Pears</List.Item>

</List>

```

Create the sub component as a separate component in the parent component's directory:

```js

function ListItem() {

// ...

}

```

Attach it to the parent via static properties:

```js

import ListItem from './ListItem'

function List() {

// ...

}

List.Item = ListItem

```

```js

import ListItem from './ListItem'

class List {

static Item = ListItem

}

```

#### Component Part Props

Sometimes it is convenient to use props to generate markup. Example, the [Label][10] markup is minimal. One configuration includes an image and detail:

```html

<a class="ui image label">

<img src="veronika.jpg">

Veronika

<div class="detail">Friend</div>

</a>

```

We allow props to define these minimal \*component parts\*:

```jsx

<Label

image='veronika.jpg'

detail='Friend'

text='Veronica'

/>

```

When props are used for component markup generation, children are not allowed in order to prevent conflicts. See [this response][14] for more.

See [`src/factories`][13] for special methods to convert props values into ReactElements for this purpose.

## Testing

Run tests during development with `yarn test:watch` to re-run tests on file changes.

### Coverage

All PRs must meet or exceed test coverage limits before they can be merged.

Every time tests run, `/coverage` information is updated. Open `coverage/lcov/index.html` to inspect test coverage. This interactive report will reveal areas lacking test coverage. You can then write tests for these areas and increase coverage.

### Common Tests

There are many common things to test for. Because of this, we have [`test/specs/commonTests.js`][1].

>This list is not updated, check the [source][1] for current tests and inline documentation.

```js

common.isConformant()

common.hasUIClassName()

common.hasSubcomponents()

common.isTabbable()

common.rendersChildren()

common.implementsIconProp()

common.implementsImageProp()

common.implementsTextAlignProp()

common.implementsVerticalAlignProp()

common.implementsWidthProp()

common.propKeyOnlyToClassName()

common.propValueOnlyToClassName()

common.propKeyAndValueToClassName()

common.propKeyOrValueAndKeyToClassName()

```

#### Usage

Every common test receives your component as its first argument.

```js

import React from 'react'

import \* as common from 'test/specs/commonTests'

import Menu from 'src/collections/Menu/Menu'

import MenuItem from 'src/collections/Menu/MenuItem'

describe('Menu', () => {

common.isConformant(Menu)

common.hasUIClassName(Menu)

common.rendersChildren(Menu)

common.hasSubcomponents(Menu, [MenuItem]) // some take additional arguments

})

```

The last argument to a common test is always `options`. You can configure the test here. Example, if your component requires certain props to render, you can pass in `requiredProps`:

```js

import \* as common from 'test/specs/commonTests'

import Select from 'src/addons/Select/Select'

describe('Select', () => {

common.isConformant(Select, {

requiredProps: {

options: [],

},

})

})

```

#### isConformant (required)

This is the only required test. It ensures a consistent baseline for the framework. It also helps you get your component off the ground. You should add this test to new components right away.

>This list is not updated, check the [source][1] for the latest assertions.

1. Component and filename are correct

1. Events are properly handled

1. Extra `props` are spread

1. Base `className`s are applied

1. Component is exported if public / hidden if private

## State

Strive to use stateless functional components when possible:

```js

function MyComponent(props) {

return <div {...props} />

}

```

If you're component requires event handlers, it is a stateful class component. Want to know [why][15]?

```js

class MyComponent extends Component {

handleClick = (e) => {

console.log('Clicked my component!')

}

render() {

return <div onClick={this.handleClick} />

}

}

```

### AutoControlledComponent

TODO

>For now, you should reference Dropdown as an example implementation. You can also consult the comments in AutoControlledComponent.js for more background.

## Documentation

- [Website](#website)

- [Components](#components)

- [Props](#props)

- [Examples](#examples)

Our docs are generated from docblock comments, `propTypes`, and hand written examples.

### Website

Developing against the doc site is a good way to try your component as you build it. Run the doc site with:

```sh

yarn start

```

### Components

A docblock should appear above a component class or function to describe it:

```js

/\*\*

\* A <Select /> is sugar for <Dropdown selection />.

\* @see Dropdown

\*/

function Select(props) {

return <Dropdown {...props} selection />

}

```

### Props

A docblock should appear above each prop in `propTypes` to describe them:

>Limited props shown for brevity.

```js

Label.propTypes = {

/\*\* An element type to render as (string or function). \*/

as: PropTypes.elementType,

/\*\* A label can reduce its complexity. \*/

basic: PropTypes.bool,

/\*\* Primary content. \*/

children: PropTypes.node,

/\*\* Additional classes. \*/

className: PropTypes.string,

/\*\* Color of the label. \*/

color: PropTypes.oneOf(Label.\_meta.props.color),

/\*\* Place the label in one of the upper corners . \*/

corner: PropTypes.oneOfType([

PropTypes.bool,

PropTypes.oneOf(['left', 'right']),

]),

/\*\* Add an icon by icon className or pass an <Icon /> \*/

icon: PropTypes.oneOfType([

PropTypes.string,

PropTypes.element,

]),

}

```

### Examples

>This section is lacking in instruction as the docs are set to be overhauled (PRs welcome!).

Usage examples for a component live in `docs/src/examples`. The examples follow the SUI doc site examples.

Adding documentation for new components is a bit tedious. The best way to do this (for now) is to copy an existing component's and update them.

## Releasing

On the latest clean `master`:

```sh

npm run release:<major|minor|patch>

```

> :warning: `npm` must be used. At the time of writing`yarn` does not properly handle the credentials.

Releasing will update the changelog which requires [github\_changelog\_generator][15].

[1]: https://github.com/Semantic-Org/Semantic-UI-React/blob/master/test/specs/commonTests.js

[2]: https://facebook.github.io/react/docs/forms.html#controlled-components

[3]: https://facebook.github.io/react/docs/forms.html#uncontrolled-components

[4]: https://github.com/Semantic-Org/Semantic-UI-React/blob/master/src/lib/classNameBuilders.js

[5]: https://semantic-ui.com/elements/header

[6]: https://semantic-ui.com/views/item

[7]: https://github.com/Semantic-Org/Semantic-UI-React/pull/281#issuecomment-228663527

[8]: https://github.com/angular/angular/blob/master/CONTRIBUTING.md#commit

[9]: https://semantic-ui.com/introduction/glossary.html

[10]: https://semantic-ui.com/elements/label.html

[11]: https://nodejs.org/

[12]: https://github.com/Semantic-Org/Semantic-UI-React#fork-destination-box

[13]: https://github.com/Semantic-Org/Semantic-UI-React/blob/master/src/factories

[14]: https://github.com/Semantic-Org/Semantic-UI-React/pull/335#issuecomment-238960895

[15]: https://github.com/Semantic-Org/Semantic-UI-React/issues/607

[16]: https://yarnpkg.com/en/docs/getting-started