javivelasco / react-css-themr

Watch 16 | Star 296 | Fork 24

Code | Issues 3 | Pull requests 1 | Projects 0 | Wiki | Pulse | Graphs

Easy theming and composition for CSS Modules.

theming react css-modules themr react-toolbox

104 commits | 1 branch | 15 releases | 11 contributors | MIT

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

javivelasco 2.0.0 release — Latest commit a54d42c 8 days ago

src	Fix linting errors	8 days ago
test	Deprecate withRef for innerRef (#46)	8 days ago
.babelrc	Fix IE10	2 months ago
.bumpedrc	Add bumped	10 months ago
.editorconfig	Add editorconfig	10 months ago
.eslintrc	Update linting	10 months ago
.gitignore	Initial commit	10 months ago
.travis.yml	Rename travis config file	10 months ago
LICENSE	Initial commit	10 months ago
README.md	Deprecate withRef for innerRef (#46)	8 days ago
index.d.ts	Deprecate withRef for innerRef (#46)	8 days ago
package.json	2.0.0 release	8 days ago
yarn.lock	Update dependencies	2 months ago

README.md

React CSS Themr

Easy theming and composition for CSS Modules.

```
$ npm install --save react-css-themr
```

Note: Feedback and contributions on the docs are highly appreciated.

Why?

When you use CSS Modules to style your components, a classnames object is usually imported from the same component. Since css classes are scoped by default, there is no easy way to make your component customizable for the outside world.

The approach

Taking ideas from future-react-ui and react-themeable, a component should be shipped **without** styles. This means we can consider the styles as an **injectable dependency**. In CSS Modules you can consider the imported classnames object as a **theme** for a component. Therefore, every styled component should define a *classname API* to be used in the rendering function.

The most immediate way of providing a classname object is via *props*. In case you want to import a component with a theme already injected, you have to write a higher order component that does the job. This is ok for your own components, but for ui-kits like React Toolbox or Belle, you'd have to write a wrapper for every single component you want to use. In this fancy, you can understand the theme as a **set** of related classname objects for different components. It makes sense to group them together in a single object and move it through the component tree using a context. This way, you can provide a theme either via **context**, **hoc** or **props**.

The approach of react-css-themr consists of a *provider* and a *decorator*. The provider sets a context theme. The decorator adds to your components the logic to figure out which theme should be used or how should it be composed, depending on configuration, context and props.

Combining CSS modules

There are three possible sources for your component. Sorted by priority: **context**, **configuration** and **props**. Any of them can be missing. In case multiple themes are present, you may want to compose the final classnames object in three different ways:

- *Override*: the theme object with the highest priority is the one used.
- *Softly merging*: theme objects are merged but if a key is present in more than one object, the final value corresponds to the theme with highest priority.
- *Deeply merging*: theme objects are merged and if a key is present in more than one object, the values for each objects are concatenated.

You can choose whatever you want. We consider the last one as the most flexible so it's selected *by default*.

How does it work?

Say you have a `Button` component you want to make themeable. You should pass a unique name identifier that will be used to retrieve its theme from context in case it is present.

```
// Button.js
import React, { Component } from 'react';
import { themr } from 'react-css-themr';

@themr('MyThemedButton')
class Button extends Component {
  render() {
    const { theme, icon, children } = this.props;
    return (
      <button className={theme.button}>
        { icon ? <i className={theme.icon}>{icon}</i> : null}
        <span className={theme.content}>{children}</span>
      </button>
    )
  }
}

export default Button;
```

The component is defining an API for theming that consists of three classnames: button, icon and content. Now, a component can use a button with a success theme like:

```
import Button from './Button';
import successTheme from './SuccessButton.css';

export default (props) => (
  <div {...props}>
    <p>Do you like it?</p>
    <Button theme={successTheme}>Yeah!</Button>
  </div>
);
```

Default theming

If you use a component with a base theme, you may want to import the component with the theme already injected. Then you can compose its style via props with another theme object. In this case the base css will **always** be bundled:

```
// SuccessButton.js
import React, { Component } from 'react';
import { themr } from 'react-css-themr';
import successTheme from './SuccessButton.css';

@themr('MySuccessButton', successTheme)
class Button extends Component {
  render() {
    const { theme, icon, children } = this.props;
    return (
      <button className={theme.button}>
        { icon ? <i className={theme.icon}>{icon}</i> : null}
        <span className={theme.content}>{children}</span>
      </button>
    )
  }
}

export default Button;
```

Imagine you want to make the success button uppercase for a specific case. You can include the classname mixed with other classnames:

```
import React from 'react';
import SuccessButton from 'SuccessButon';
import style from './Section.css';

export default () => (
  <section className={style.section}>
    <SuccessButton theme={style.button}>Yai!</SuccessButton>
  </section>
);
```

And being `Section.css` something like:

```
.section { border: 1px solid red; }
.button  { text-transform: uppercase; }
```

The final classnames object for the `Button` component would include class values from `SuccessButton.css` and `Section.css` so it would be uppercase!

Context theming

Although context theming is not limited to ui-kits, it's very useful to avoid declaring hoc for every component. For example, in react-toolbox, you can define a context theme like:

```
import React from 'react';
import { render } from 'react-dom';
import { ThemeProvider } from 'react-css-themr';
import App from './app'

const contextTheme = {
  RTButton: require('react-toolbox/lib/button/style.scss'),
  RTDialog: require('react-toolbox/lib/dialog/style.scss')
};

const content = (
  <ThemeProvider theme={contextTheme}>
    <App />
  </ThemeProvider>
);

render(content, document.getElementById('app'));
```

The main idea is to inject classnames objects for each component via context. This way you can have the whole theme in a single place and forget about including styles in every require. Any component `Button` or `Dialog` from will use the provided styles in the context.

API

`<ThemeProvider theme>`

Makes available a `theme` context to use in styled components. The shape of the theme object consists of an object whose keys are identifiers for styled components provided with the `themr` function with each theme as the corresponding value. Useful for ui-kits.

`themr(Identifier, [defaultTheme], [options])`

Returns a `function` to wrap a component and make it themeable.

The returned component accepts a `theme`, `composeTheme` and `innerRef` props apart from the props of the original component. They former two are used to provide a `theme` to the component and to configure the style composition, which can be configured via options too, while the latter is used to pass a ref callback to the decorated component. The function arguments are:

- `Identifier` *(String)* used to provide a unique identifier to the component that will be used to get a theme from context.
- `[defaultTheme]` (*Object*) is classname object resolved from CSS modules. It will be used as the default theme to calculate a new theme that will be passed to the component.
- `[options]` (*Object*) If specified it allows to customize the behavior:
  - [`composeTheme = 'deeply'`] *(String)* allows to customize the way themes are merged or to disable merging completely. The accepted values are `deeply` to deeply merge themes, `softly` to softly merge themes and `false` to disable theme merging.

About

The project is originally authored by Javi Velasco as an effort of providing a better customization experience for React Toolbox. Any comments, improvements or feedback is highly appreciated.

Thanks to Nik Graf and Mark Dalgleish for their thoughts about theming and customization for React components.

License