The Journal of Open Source Education

react-simplified: an easy to use JavaScript UI library with automatic state and property management

Ole Christian Eidheim[1]

1 Department of Computer Science, Norwegian University of Science and Technology

DOI: 10.21105/jose.00137

Software

- Review
- Repository
- Archive

Submitted: 01 October 2021
Published: 28 October 2021

Summary

Many applications today are web applications where the frontend is typically written in HTML, CSS, and JavaScript or TypeScript. Interactive user interfaces (UI) were initially written using the web browser APIs, until JavaScript libraries became available and helped the programmers writing larger applications targeting multiple web browsers more easily. One such library is React (Walke et al., 2013) that has become popular since its initial release in 2013.

User interfaces have traditionally been written in an imperative style, where widget instances are managed by the developer. Having access to the widget instances enables the programmer to set properties on the instances directly, and programmatically place widgets in a specific container widgets that influence how the widgets will be presented. Examples of UI libraries using an imperative paradigm are Qt (Nord et al., 1995) and GTK (Kimball et al., 1998), both popular libraries for creating platform independent desktop applications.

In declarative programming, in contrast to the imperative style, the user interface component instances are instead handled by the UI library. In general, the focus in declarative programming is more on *what* is to be computed, and less on *how* it is computed (Lloyd, 1994). This might ease the burden of the programmer, but on the other hand implies additional complexity to the user interface libraries. Especially how to automatically update the component tree structure in an effective manner is challenging since the decision might depend on mutable states in the application. Further advantages of the declarative style are improved compatibility across software and API releases, and increased robustness to platform and other software specific variations (Wilde, 2007). For instance, declarative programming enables HTML written following a newer specification to be rendered in an older browser. Although there might be defects, most of the information will be available to the reader.

React is a JavaScript library that enables performant declarative UI programs (Staff, 2016) through a heuristic algorithm that works around the complexity of tree transformations (Bille, 2005). React supports both functional components and class components, but the focus of this article are class components, which are more general, that is fewer specialized functions are required, and also easier to learn for developers with a background in object oriented programming.

Components extended from the React component are drawn and redrawn depending on internal component states and component properties. Properties are handed down from the parent component, and a component is redrawn whenever one of its properties changes. State changes, on the other hand, require manual calls to the component to initiate a redraw. The main contribution of the `react-simplified` library is to also redraw components upon state changes automatically, through the use of JavaScript Proxy objects.

Proxy objects make it possible to intercept reads on an object acting as a state during rendering, and thus identify which member variable modifications would require a component to be redraw. The result is further ease of burden on the developer who no longer needs to track which member variables are used as render states, and furthermore makes it unnecessary to manually initiate redraw of components. The use of member variables as internal component states also does not require special handling of how types of states are added for static type checking, which in regular React components must be added as a generic parameter when declaring the component. See section Examples for code examples comparing use of React to `react-simplified` components.

In addition to internal component states, that is member variables used during rendering, global state objects can also more easily be used if created through the function `sharedComponentData()`. These global objects are similarly tracked during component render methods, and changes to these objects will initiate redraw on all components that utilize the objects during rendering. Although the consensus might be that global state should be avoided, it gives additional flexibility to the programmer implementing for instance store type objects, containing both member functions to fetch data from a server and the data itself, of which the data is possibly used by several concurrently visible components.

Another challenge is when component properties change that should cause new data to be fetched from a database or a web server, and the fetched data might affect rendering of the component. When a component relies on data fetches, these fetches are normally handled in a member function that is called when the component is inserted into the visible component tree. In the React library, this function is named `ComponentDidMount()`. In `react-simplified`, the properties used in this member function are also detected through the use of JavaScript Proxy objects, and changes to these properties will cause the `ComponentDidMount()` member function to be called again. Additionally, if the member function which will handle the removal of the component from the visible component tree, named `componentWillUnmount()` in React, is defined, this function will be called prior to `ComponentDidMount()`.

To further ease the developer experience, methods defined in `react-simplified` components are binded automatically, meaning their `this` objects are always correctly set within the methods. Automatically binded methods moreover eliminate the need for creating class constructors in most cases.

The major disadvantage of `react-simplified` components compared to regular React components, apart from the added complexity and resource use, is lack of support for Internet Explorer due to its missing implementation of Proxy objects. However, Internet Explorer is no longer in active development, and relinquishing Internet Explorer support is acceptable for most new development projects.

The library `react-simplified` has been used in several student projects at the Norwegian University of Science and Technology. In the thesis (Andersson & Roll, 2020), `react-simplified` was compared to several other similar frameworks such as regular React, Vue.js (You & Community, 2014) and Angular (Google & Community, 2016), and in the end, due to its ease of use, `react-simplified` was chosen to develop a web application for managing exercise approvals for various courses in the department.

There are a large number of libraries that extend React, such as (Abramov et al., 2015) and (Weststrate & Community, 2015), but they have in general a steeper learning curve than `react-simplified` due to stricter compatibility and resource use requirements to stay competitive for general use. These requirements can, however, be relaxed in an educational setting.

Statement of Need

Lecturers are often limited by the currently available libraries that are not necessarily a good fit for the chosen curriculum. The library `react-simplified` was created to workaround such limitations when creating UI applications, be it in web browsers, as desktop applications using the Electron framework (GitHub & Community, 2013), or as mobile applications through React Native (Facebook & Community, 2013). This has enabled students to develop UI applications more easily and with less specialized functions that must be learned beforehand or discovered during development. As application development in general is taught worldwide, it is likely that other lecturers will find the library `react-simplified` useful. Furthermore, this library is not limited to educational use, the students can continue to use `react-simplified` in their professional careers.

Usage in Teaching and Learning

The library `react-simplified` is currently used in subsequent JavaScript/TypeScript programming courses spanning three semesters. The content of the first course is JavaScript foundations, including functional array algorithms such as `map()`, `filter()`, and `reduce()` that will be used when implementing `react-simplified` components in the following semester. In addition to programming JavaScript for web browser pages, the students are taught to create desktop applications using the Electron framework (GitHub & Community, 2013) to perform file and database operations, which are not allowed in web browser applications directly.

In the second semester, the students learn object oriented programming and to program desktop and mobile applications using `react-simplified`. The students learn multiple design patterns, including how to develop and make use of reusable components, also called widgets, using a CSS framework. Static type checking through TypeScript is also introduced in this semester.

Web applications are revisited in the third semester, by introducing server side programming to support data fetches and manipulations by web or mobile client applications through the HTTP protocol. The library `react-simplified` is used to create the application clients in this semester as well. Additionally, the students are thought both client and server side testing, continuous integration, and full-duplex client-server communication through WebSocket.

Teaching and Learning Experience

Prior to `react-simplified` we made use of available web technologies such as Angular (Google & Community, 2016), Mithril (Horie & Community, 2017), and regular React. These libraries were difficult to grasp fully, and much focus had to be placed on how to solve various challenges with these libraries in particular. Additionally, some design patterns were unavailable, and extra libraries were needed to implement for instance shared states between components.

Libraries such as Angular (Google & Community, 2016) and Vue.js (You & Community, 2014) have custom component string properties to for instance render multiple component children from a component state. In React, however, students can instead take advantage of functional algorithms such as `map`, `filter`, and `reduce` that are common to many programming languages.

By utilizing `react-simplified` we can focus more on programming and design patterns in general, and the techniques learned are less tied to a particular library. Components extended from `react-simplified` are also backward compatible with regular React components, and if needed the students can write parts of the source code using regular React code, for instance influenced by examples discovered on the Internet.

Examples

The examples in this section are written in TypeScript to also show the additional generic parameters that are needed in regular React components for internal component state types.

The first example demonstrates the different use of React and `react-simplified` components with an internal component state. Binding of class methods is also demonstrated, where `increaseCounter()` is automatically bound in `ReactSimplifiedComponent` and there is thus no need to implement a constructor for this component.

```
import * as React from 'react';
import * as ReactSimplified from 'react-simplified';
import ReactDOM from 'react-dom';

// Component extending a regular React component.
class ReactComponent extends React.Component<{}, { counter: number }> {
  state = { counter: 0 };

  constructor(props: {}) {
    super(props);

    // Must bind the class method `increaseCounter` so that the `this` keyword
    // is correctly set inside the method body when called from the button
    // onclick event.
    this.increaseCounter = this.increaseCounter.bind(this);
  }

  increaseCounter() {
    this.setState({ counter: this.state.counter + 1 });
  }

  render() {
    return (
      <>
        Counter: {this.state.counter}
        <button onClick={this.increaseCounter}>Increate counter</button>
      </>
    );
  }
}

// Component extending a react-simplified component.
class ReactSimplifiedComponent extends ReactSimplified.Component {
  // Member variable `counter` can be used as an internal component state
  // directly, and the component will schedule rerender automatically when the
  // variable is modified.
  counter: number = 0;
```

```
  // Class methods like `increaseCounter()` are automatically bound in classes
  // extending a react-simplified component.
  increaseCounter() {
    this.counter++;
  }

  render() {
    return (
      <>
        Counter: {this.counter}
        <button onClick={this.increaseCounter}>Increate counter</button>
      </>
    );
  }
}

ReactDOM.render(
  <>
    <ReactComponent />
    <ReactSimplifiedComponent />
  </>,
  document.getElementById('root')
);
```

The next example will demonstrate automatic property management in `react-simplified` components. A common design pattern is to use service classes, sometimes called data access objects (DAO), to fetch data from a server. Such service class methods are often called when a component is first made visible, in the `componentDidMount()` method in React components, where the result is stored as an internal state and made visible by the `render()` method by scheduling a redraw of the component. If a component property is used as a parameter for the service method call, the service method has to be called again if the component property changes. In regular React components, this logic has to be added to the member function `componentDidUpdate()`, however, `react-simplified` components will track properties used in the `componentDidMount()` call, and call the `componentDidMount()` method again whenever these properties change.

```
import * as React from 'react';
import * as ReactSimplified from 'react-simplified';
import ReactDOM from 'react-dom';

class Service {
  // Method simulating a data fetch from server given argument `id`.
  get(id: number) {
    return Promise.resolve('Fetch result from id=' + id);
  }
}
const service = new Service();

class ReactComponent extends React.Component<{}, { id: number }> {
  state = { id: 0 };

  render() {
    return (
      <>
```

```
        <button onClick={() => this.setState({ id: this.state.id + 1 })}>
          Increase id
        </button>
        <ReactChildComponent id={this.state.id} />
      </>
    );
  }
}

class ReactChildComponent extends React.Component<
  { id: number },
  { result: string }
> {
  state = { result: '' };

  render() {
    return <>{this.state.result}</>;
  }

  componentDidMount() {
    service
      .get(this.props.id)
      .then((result) => this.setState({ result: result }));
  }

  // `componentDidUpdate()` has to be added in regular React components to
  // handle cases when a property change and new data has to be fetched from
  // the server.
  componentDidUpdate(prevProps: { id: number }) {
    if (this.props.id !== prevProps.id) {
      service
        .get(this.props.id)
        .then((result) => this.setState({ result: result }));
    }
  }
}

class ReactSimplifiedComponent extends ReactSimplified.Component {
  id = 0;

  render() {
    return (
      <>
        <button onClick={() => this.id++}>Increase id</button>
        <ReactSimplifiedChildComponent id={this.id} />
      </>
    );
  }
}

class ReactSimplifiedChildComponent extends ReactSimplified.Component<{
  id: number,
}> {
  result = '';
```

```
  render() {
    return <>{this.result}</>;
  }

  // Since the property `id` is used in `componentDidMount()`,
  // `componentDidMount()` will be called again if the property `id` changes.
  componentDidMount() {
    service.get(this.props.id).then((result) => (this.result = result));
  }
}

ReactDOM.render(
  <>
    <ReactComponent />
    <ReactSimplifiedComponent />
  </>,
  document.getElementById('root')
);
```

Another common design pattern is using store classes instead of service classes. Compared to service classes, store classes also contain for instance the server fetch result as a global state that can be used by several components. The final example demonstrates how store classes can be implemented and used by react-simplified components. Note that regular React components do not support this design pattern.

```
import * as React from 'react';
import * as ReactSimplified from 'react-simplified';
import ReactDOM from 'react-dom';

class Store {
  // The member variable `counter` can be used in several react-simplified
  // components, and these components will schedule redraw whenever `counter`
  // changes.
  counter: number = 0;

  increaseCounter() {
    this.counter++;
  }
}
const store = ReactSimplified.sharedComponentData(new Store());

class ReactSimplifiedComponent1 extends ReactSimplified.Component {
  render() {
    return (
      <>
        Counter: {store.counter}
        <button onClick={store.increaseCounter}>Increate counter</button>
      </>
    );
  }
}

class ReactSimplifiedComponent2 extends ReactSimplified.Component {
  render() {
    return (
```

```
      <>
        Counter: {store.counter}
        <button onClick={store.increaseCounter}>Increate counter</button>
      </>
    );
  }
}

ReactDOM.render(
  <>
    <ReactSimplifiedComponent1 />
    <ReactSimplifiedComponent2 />
  </>,
  document.getElementById('root')
);
```

Acknowledgments

I would like to thank all the students and other developers who have given their valuable feedback on this project.

References

Abramov, D., Clark, A., & Community. (2015). Redux. In *GitHub repository*. GitHub. https://github.com/reduxjs/redux

Andersson, V., & Roll, E. (2020). *Front-end study and application of modern web-app technologies with the aim of improving an existing system* [B.S. thesis, NTNU]. https://hdl.handle.net/11250/2672185

Bille, P. (2005). A survey on tree edit distance and related problems. *Theoretical Computer Science*, *337*(1-3), 217–239. https://doi.org/10.1016/j.tcs.2004.12.030

Facebook, & Community. (2013). React native. In *GitHub repository*. GitHub. https://github.com/facebook/react-native

GitHub, & Community. (2013). Electron. In *GitHub repository*. GitHub. https://github.com/electron/electron

Google, & Community. (2016). Angular. In *GitHub repository*. GitHub. https://github.com/angular/angular

Horie, L., & Community. (2017). Mithril. In *GitHub repository*. GitHub. https://github.com/MithrilJS/mithril.js

Kimball, S., Mattis, P., Project, T. G., Computing Facility, eXperimental, & Community. (1998). GTK. In *The GNOME Project Git Repository*. The GNOME Project. https://gitlab.gnome.org/GNOME/gtk

Lloyd, J. W. (1994). Practical advtanages of declarative programming. *GULP-PRODE'94, 1994 Joint Conference on Declarative Programming*, 3–17.

Nord, H., Chambe-Eng, E., Trolltech, Nokia, Project, Q., Digia, Company, T. Q., & Community. (1995). Qt. In *Qt Project Git Repository*. Qt Project. https://code.qt.io/cgit/qt/qtbase.git/

Staff, C. (2016). React: Facebook's functional turn on writing javascript. *Commun. ACM*, *59*(12), 56–62. https://doi.org/10.1145/2980991

Walke, J., Facebook, & Community. (2013). React. In *GitHub repository*. GitHub. https://github.com/facebook/react

Weststrate, M., & Community. (2015). MobX. In *GitHub repository*. GitHub. https://github.com/mobxjs/mobx

Wilde, E. (2007). Declarative web 2.0. *2007 IEEE International Conference on Information Reuse and Integration*, 612–617. https://doi.org/10.1109/IRI.2007.4296688

You, E., & Community. (2014). Vue.js. In *GitHub repository*. GitHub. https://github.com/vuejs/vue-next