WEBINAR

Easy State Management in React

Mark Volkmann, Partner and Principal Software Engineer
mark@objectcomputing.com

GOALS

1. Store state in one place, outside any component
2. Allow components to access any state
3. Re-render components if any state they depend on changes
4. Make it easy for components to modify any state
5. Do not require new code to support new state

EXISTING STATE MANAGEMENT APPROACHES

- There are many popular approaches to managing state in React applications
- Reviewed on following slides

PROP DRILLING

- Hold application state in the topmost component
  and pass it down to descendant components using props
- Downside:
  Many components will accept and pass props they do not actually use
- When component tree becomes deep, this does not scale well

REDUX ...

- Uses a single "store" that can be thought of as a client-side database
- Components
  - obtain state data through functions like **`mapStateToProps`**
  - request state changes using functions like **`mapStateToDispatch`**
  - create action objects and dispatch them
- Reducer functions
  - are given current state and an action
  - responsible for returning new state without mutating current state

... REDUX

- There are many sub-approaches to using Redux
- Involve concepts like “thunks” and “sagas”
- Introduce unnecessary complexity

CONTEXT API ...

- Built into React
- An application can create any number of **`Context`** objects by calling **`React.createContext`**
- A **`Context`** can include both data and methods to update the data
- Each **`Context`** has a **`Provider`** and a **`Consumer`**

... CONTEXT API

- **`Provider`** elements are often rendered
  at the top of the component tree to
  make the context available to the entire application
- **`Consumer`** elements wrap around
  the rendered JSX of components that
  need to access and update context data

ISSUES WITH REDUX AND CONTEXT API

- Both have a learning curve
- Both require new code to support new state data
  - in Redux this involves modifying a reducer function
  - in the Context API this involves adding methods to the `Context`

SEGREGATION OF APP STATE ...

- Redux always uses a single store,
  but parts of the store can be managed by different reducer functions
  that are combined into a single reducer function
- Context API supports creating multiple **`Context`** instances
  that each manage subsets of the application data
- Benefit of segregating application state is that
  it's possible to know that only a subset of components
  use and have the ability to modify a particular piece of state data

... SEGREGATION OF APP STATE

- Segregating application state becomes messy
  if some components need data from multiple subsets
- Another way to think about this segregation
  is to consider REST service implementation
  - often operate on relational database tables
  - not typical to restrict a REST service to a subset of database tables
- Attempts to segregate application state data
  make code more complex for little benefit

context-easy

- How can we achieve the goals described earlier in the simplest possible way?
- context-easy is an open source library in npm
- Builds on Context API
- Implements a **`Provider`** that can manage all state for a React application
- Highly generic, making it suitable for any application

HOOKS

- Easiest way for components to use context-easy **`Provider`** is through **`useContext`** hook
- If not yet familiar with React hooks, check out my video at https://bit.ly/2Tez5V1

CONFIGURING USE ...

- Done in topmost component, often **`src/App.js`**
- Three steps are required

1. Import Easy Provider

```
import {EasyProvider} from 'context-easy';
```

... CONFIGURING USE ...

OBJECT COMPUTING

2. Define initial state

```
const initialState = {
  count: 0,
  person: {
    name: 'Mark',
    occupation: 'software developer'
  },
  size: 'medium'
};
```

... CONFIGURING USE

3. Wrap top-most component in **`EasyProvider`**

```
export default function App() {
  return (
    <EasyProvider initialState={initialState} log validate>
      {/* top-most components go here */}
    </EasyProvider>
  );
}
```

USING IN COMPONENTS ...

- In function components that access/modify state

1. Import **`useContext`** hook and **`EasyContext`**

```
import React, {useContext} from 'react';
import {EasyContext} from 'context-easy';
```

2. Get context object inside function component

```
const context = useContext(EasyContext);
```

... USING IN COMPONENTS

3. Access state from **`context`** object

```
const {name} = context.person;
```

4. Update state properties at specific paths by calling methods on **`context`**

```
context.set('person.name', 'Mark');
```

CONTEXT METHODS ...

- The context object implements ten methods
- Most take a path argument that is a string representing a dot-separated path into state
- Let's review these in alphabetical order

... CONTEXT METHODS ...

- **`context.decrement(path)`**
  - decrements number at given path
  - optional second argument specifies amount by which to decrement and defaults to one
- **`context.delete(path)`**
  - deletes property at given path

... CONTEXT METHODS ...

- `context.filter(path, fn)`
  - replaces array at given path with new array that is the result of filtering current elements
  - function provided as second argument is called on each array element
  - should return true for elements to be retained and false for elements to be filtered out

... CONTEXT METHODS ...

- **`context.increment(path)`**
  - increments number at given path
  - optional second argument specifies amount by which to increment and defaults to one

... CONTEXT METHODS ...

- **`context.log(label)`**
  - writes current state to devtools console
  - outputs **`context-easy:`**, followed by an optional label (defaults to `''`), **`state =`**, and the state object
  - object starts in collapsed view; click disclosure triangles to expand

... CONTEXT METHODS ...

- **`context.map(path, fn)`**
  - replaces array at given path with new array
  - function provided as second argument
    is passed each array element one at a time
  - new array will contain return values of these calls

... CONTEXT METHODS ...

- **`context.push(path, newValue1, newValue2, ...)`**
  - replaces array at given path with new array
  - new array starts with all existing elements and ends with all specified new values
- **`context.set(path, value)`**
  - sets value at given path to given value

... CONTEXT METHODS

- `context.toggle(path)`
  - toggles boolean value at given path
- `context.transform(path, fn)`
  - sets value at given path to
    value returned by passing current value
    to function provided as second argument

COMPONENT RE-RENDERING ...

- **`useContext`** hook subscribes components that call it to context state updates
- Means components will be re-rendered on every context state change
- To only re-render when specific context state properties are changed, wrap component JSX in call to **`useCallback`**

... COMPONENT RE-RENDERING

OBJECT COMPUTING

- Example: Suppose a component only depends on state properties **`count`** and **`person.name`**

```
import React, {useCallback, useContext} from 'react';

export default SomeComponent() {
  const context = useContext(EasyContext);
  const {count, person} = context;
  const {name} = person;
  return useCallback(
    <div>
      ...component JSX goes here...
    </div>,
    [count, name] // key part!
  );
}
```

FORM ELEMENTS

- It is common to have **`input`**, **`select`**, and **`textarea`** elements with **`onChange`** handlers that get their value from **`event.target.value`** and update a specific state path
- Alternative is to use provided components **`Input`**, **`Select`**, **`TextArea`**, **`RadioButtons`**, and **`Checkboxes`**
- All these take a **`path`** prop which is used to get the current value for the component and update the value at that path

Two-way
data binding!

INPUT COMPONENT

- HTML **`input`** elements can be replaced by **`Input`** component
- **`type`** property defaults to **`'text'`**,
  but can be set to any valid value including **`'checkbox'`**
- Value used by **`Input`** is state value at specified path
- When user changes value, this component updates value at path
- To perform additional processing of changes such as validation,
  supply an **`onChange`** prop whose value is a function

```
<Input path="user.firstName" />
```

TEXTAREA COMPONENT

- HTML **`textarea`** elements can be replaced by **`TextArea`** component

```
<TextArea path="feedback.comment" />
```

SELECT COMPONENT

- HTML **`select`** elements can be replaced by **`Select`** component

```
<Select path="favorite.color">
  <option>red</option>
  <option>green</option>
  <option>blue</option>
</Select>
```

- If **`option`** elements have **`value`** attribute,
  that value will be used instead of text inside **`option`**

RADIOBUTTONS COMPONENT

OBJECT COMPUTING

- For set of radio buttons, use **`RadioButtons`** component

```
<RadioButtons
  className="flavor"
  list={radioButtonList}
  path="favorite.flavor"
/>
```

```
const radioButtonList = [
  {text: 'Chocolate', value: 'choc'},
  {text: 'Strawberry', value: 'straw'},
  {text: 'Vanilla', value: 'van'}
];
```

- When a radio button is clicked, state property **`flavor`** will be set to the value of that radio button

CHECKBOXES COMPONENT

OBJECT COMPUTING

- For set of checkboxes, use **`Checkboxes`** components

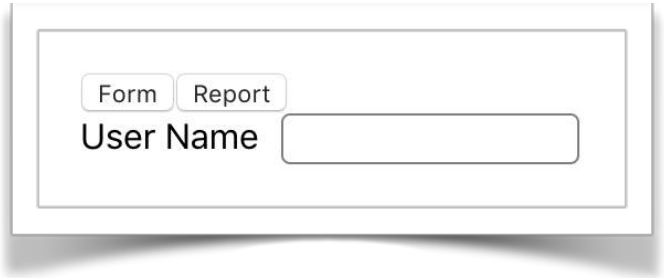
```
<Checkboxes className="colors" list={checkboxList} />
```

```
const checkboxList = [
  {text: 'Red', path: 'color.red'},
  {text: 'Green', path: 'color.green'},
  {text: 'Blue', path: 'color.blue'}
];
```

- When a checkbox is clicked, boolean value at corresponding path will be toggled between false and true

DEMO

- Demonstrates using context-easy in an app with multiple views



DEMO SETUP ...

- Open terminal window
- Enter **`create-react-app demo`**
- Enter **`cd demo`**
- Enter **`npm run start`**
- Verify that initial app is running in default browser

... DEMO SETUP

- Open new terminal window
- Enter **`npm install context-easy`**

DEMO - FORM COMPONENT

OBJECT COMPUTING

- Create **`src/Form.js`**
  - use an **`Input`** component for path **`user.name`**

```
import React from 'react';
import {Input} from 'context-easy';

export default function Form() {
  return (
    <form>
      <label>User Name</label>
      <Input path="user.name" />
    </form>
  );
}
```

Two-way data binding!

DEMO - REPORT COMPONENT

OBJECT COMPUTING

- Create **`src/Report.js`**
  - render **`context.user.name`**

```
import React, {useContext} from 'react';
import {EasyContext} from 'context-easy';

export default function Report() {
  const context = useContext(EasyContext);
  return <div>The user name is {context.user.name}.</div>;
}
```

DEMO - TOP COMPONENT

- Create **`src/Top.js`**
  - use value of **`context.route`** to decide whether a **`Form`** or **`Report`** component should be rendered
  - provides very simple routing for apps where the URL does not need to change when the view changes
  - see code on next slide

DEMO - TOP COMPONENT

OBJECT COMPUTING

```
import React, {useContext} from 'react';
import {EasyContext} from 'context-easy';
import Form from './Form';
import Report from './Report';

export default function Top() {
  const context = useContext(EasyContext);
  const page = context.route === 'report' ? <Report /> : <Form />;
  return (
    <div>
      <header>
        <button onClick={() => context.set('route', 'form')}>Form</button>
        <button onClick={() => context.set('route', 'report')}>Report</button>
      </header>
      {page}
    </div>
  );
}
```

DEMO - APP COMPONENT

OBJECT COMPUTING

- Modify **`src/App.js`**
  - render **`Top`** component inside an **`EasyProvider`**

```
import React from 'react';
import {EasyProvider} from 'context-easy';
import Top from './Top';
import './App.css';

const initialState = {
  route: '',
  user: {name: ''}
};

export default function App() {
  return (
    <EasyProvider
      initialState={initialState}
      log validate>
      <Top />
    </EasyProvider>
  );
}
```

DEMO - CSS

OBJECT COMPUTING

- Modify **`src/App.css`**
  - remove unused rules
  - add rules for **`body`** and **`input`** elements

```
body {
  padding: 20px;
}

input {
  border: solid gray 1px;
  border-radius: 4px;
  margin-left: 10px;
  padding: 4px;
}
```

DEMO OPERATION

- Return to default web browser
- Enter a user name
- Click “Report” button
- Verify that user name is displayed
- Click “Form” button
- Change name
- Verify that report is updated

DEMO DEBUGGING

- Open browser devtools console
- Note messages that are output when context is modified and how entire state can be viewed
- Select React tab to view the react-devtools (assumes it is installed)
- Select **`EasyProvider`** element inside the **`App`** element
- Note how all context state can be viewed here also

DEMO EXTRA CREDIT

- Implement same app using Redux
  or using Context API without context-easy
- The code will be much longer!

OPTIONS ...

- **`EasyProvider`** component accepts props that specify options
- To log all state changes in devtools console, include **`log`** prop with no value
- To validate all method calls made on context object
  and throw an error when they are called incorrectly,
  include **`validate`** prop with no value
- Useful in development, but typically should not be used in production
- If **`NODE_ENV`** environment variable is set to **`'production'`**,
  `log` and **`validate`** options are ignored

... OPTIONS

- Other options are specified in **`options`** prop
  whose value is an object that specifies their values
- **`persist`** option is described on "SessionStorage" slide later
- **`version`** option is described on "Versions" slide later
- **`replacerFn`** and **`reviverFn`** options
  are described on "Sensitive Data" slide later

PATH CONCERNS ...

- When layout of state changes,
  it is necessary to change state paths throughout code
- For apps that use a small number of state paths
  this is likely not a concern
- For apps that use a large number of state paths,
  consider creating a source file that exports
  constants for state paths, perhaps named **`path-constants.js`**,
  and use those when calling **`context`** methods that requires a path

... PATH CONCERNS

- Example:

```
// In path-constants.js ...
const GAME_HIGH_SCORE = 'game.statistics.highScore';
const USER_CITY = 'user.address.city';
```

```
// In the source file for a component ...
import {GAME_HIGH_SCORE, USER_CITY} from './path-constants';
...
context.set(USER_CITY, 'St. Louis');
context.transform(GAME_HIGH_SCORE, score => score + 1);
```

- With this approach, if layout of state changes it is only necessary to update these constants

SESSIONSTORAGE ...

- Typically React state is lost when users refresh the browser
- To avoid this, **`sessionStorage`** is used to save context state on every state change
- Throttled to not updated more than once per second
- **`sessionStorage`** state is automatically reloaded into context state when browser is refreshed

... SESSIONSTORAGE

- To opt out of this behavior ...

```
const options = {persist: false}; // defaults to true
...
return (
  <EasyProvider initialState={initialState} options={options}>
    ...
  </EasyProvider>
);
```

VERSIONS ...

- During development, when the shape of the initial state changes it is desirable to replace what is in **`sessionStorage`** with new initial state
- One way is to close browser tab and open new one
- Otherwise the application may not work properly because it will expect different data than what is in **`sessionStorage`**

... VERSIONS

- To force new initial state to be used, supply a version property in options object passed to **`EasyProvider`**
- When a new version is seen, data in **`sessionStorage`** replaced with **`initialState`** value passed to **`EasyProvider`**

SENSITIVE DATA

- Can prevent sensitive data such as passwords and credit card numbers from being written to **`sessionStorage`**
- Add **`replacerFn`** and **`reviverFn`** functions to options object passed to **`EasyProvider`**
  - similar to optional **`replacer`** and **`reviver`** parameters used by **`JSON.stringify`** and **`JSON.parse`**
  - both are passed a state object
- To change data in any way, including deleting, modifying, and adding properties, make a copy of state object, modify the copy, and return it

Consider using the lodash function **`deepClone`** to create the copy.

EXAMPLE APP

- A larger example application that uses context-easy can be found in the GitHub repository at https://github.com/mvolkmann/context-easy-demo

BROWSER DEVTOOLS ...

- Nice feature of Redux is ability to use redux-devtools
- Supports viewing all actions that have been dispatched and state after each action has been processed
- Also supports "time travel debugging" which shows state of UI after a selected action

... BROWSER DEVTOOLS

- **`log`** feature of context-easy outputs a description of each context method call and the state after the call
- Somewhat of a replacement for what redux-devtools provides
- react-devtools displays data in a context when its **`Provider`** element is selected
- Updated dynamically when context data changes

GOAL ASSESSMENT ...

1. Store state in one place, outside any component
   - context-easy uses a single Context
2. Allow components to access any state
   - **`context`** object holds all application state
     and is directly accessible in components
     using references like **`context.user.address.city`**

... GOAL ASSESSMENT ...

3. Re-render components if any state they depend on changes

- components that call **`useContext(EasyContext)`**
  are subscribed to changes in the context
  and re-render when the context state changes
- to make this more efficient, add use of **`useCallback`** hook
  so components only re-render when there are changes they care about

... GOAL ASSESSMENT ...

4. Make it easy for components to modify any state

- **`context`** object has methods that support this such as **`context.set(path, value)`**

... GOAL ASSESSMENT

5. Do not require new code to support new state

- when new application state data is needed, only change is to add initial value where initial state is defined
- when acceptable to begin with **`undefined`** value, no changes are needed

Wrap Up

- I believe context-easy provides the easiest way to manage state in a React application
- Much easier than using Redux or using the Context API directly

Other Options

- If you would rather continue using Redux, see **redux-easy** in npm
  - supports mostly same API
- To use this approach in a Vue application, see **vuex-easy** in npm
  - also supports mostly same API

LEARN MORE ABOUT OCI EVENTS AND TRAINING

Events:

- objectcomputing.com/events

Training:

- objectcomputing.com/training
- grailstraining.com
- micronauttraining.com

Or email info@ocitraining.com to schedule a custom training program for your team online, on site, or in our state-of-the-art, Midwest training lab.

OBJECT COMPUTING

CONNECT WITH US

1+ (314) 579-0066

@objectcomputing

objectcomputing.com