Fazit React-Prototyp

Einleitung

Im April 2019 wurde ein Prototyp mit React entwickelt. Eine lauffähige Version befindet sich auf meiner Github Seite und implementiert verschiedene UI Frameworks (unter anderem Elements UI, Ant Design, React-Virtualized). Auf dieser Seite befinden sich einige Anmerkungen die man während der Entwicklung mit React berücksichtigen sollte und die mir persönlich insbesondere aufgefallen sind. Hierbei wird ein direkter Vergleich zum Vue-Test Prototypen gemacht.

Allgemeines

1. Anders als Vue besitzt React "nur" One-way data binding. Das bedeutet, dass zum Beispiel eine Eingabe in ein Input-Feld nicht direkt in den State der Komponente gelangt, sondern zunächst mit einer Handler-Funktion bearbeitet werden muss:

   Input for React

   ```
   <input type="text" value={this.state.myinput} onChange={(key) => this.setState({ this.state.myinput : key
   })} />
   ```

2. Man soll den State in React nie direkt verändern (z.B ist `this.state.myinput = "Test";` nicht gut). Stattdessen verwendet man die React interne `this.setState()` Methode. Jedoch rendert React bei jedem State-change die komplette Komponente neu, was zu Performance Problemen führen kann. Dies kann allerdings verhindert werden, wenn man die `shouldComponentUpdate` Methode implementiert oder von Anfang an eine `PureComponent` baut, welche shallow Vergleiche des States durchführt. Somit wird die Virtual DOM nur dann re-rendert wenn sich der State wirklich verändert.
3. Inline Conditional Rendering ist in React ein wenig umständiger als in Vue. Nehmen wir zum Beispiel folgende Komponente um eine Liste von Item anzuzeigen:

   v-for

   ```
   <ul>
           <li
                   v-for = "(item, index) in items"
                   v-bind:item = "item"
                   v-bind:index = "index"
                   v-bind:key = "item.id"
           ></li>
   </ul>
   ```

   Diese Komponente würde eine einfache Liste von Items anzeigen. Wie man sehen kann ist sie sehr kompakt, einfach & sehr gut leserlich. Das gleiche würde man in React mithilfe von maps lösen:

   react map

   ```
   render() {
           return(
                   <ul>
                           { this.props.items.map(
                                   item => <li key=item.id> {item.label} </li>)
                           }
                   </ul>
                   );
           }
   ```

   Beide Komponenten bewirken fast das selbe jedoch unterscheiden Sie sich sehr von der Vorgehensweise. Des Weiteren kann man in React in der `render()` Funktion JSX Expressions verwenden um somit Inline Conditional Rendering zu ermöglichen. Alles nach einem { (Curly Bracket) ist für React eine JSX Expression:

cond. rendering

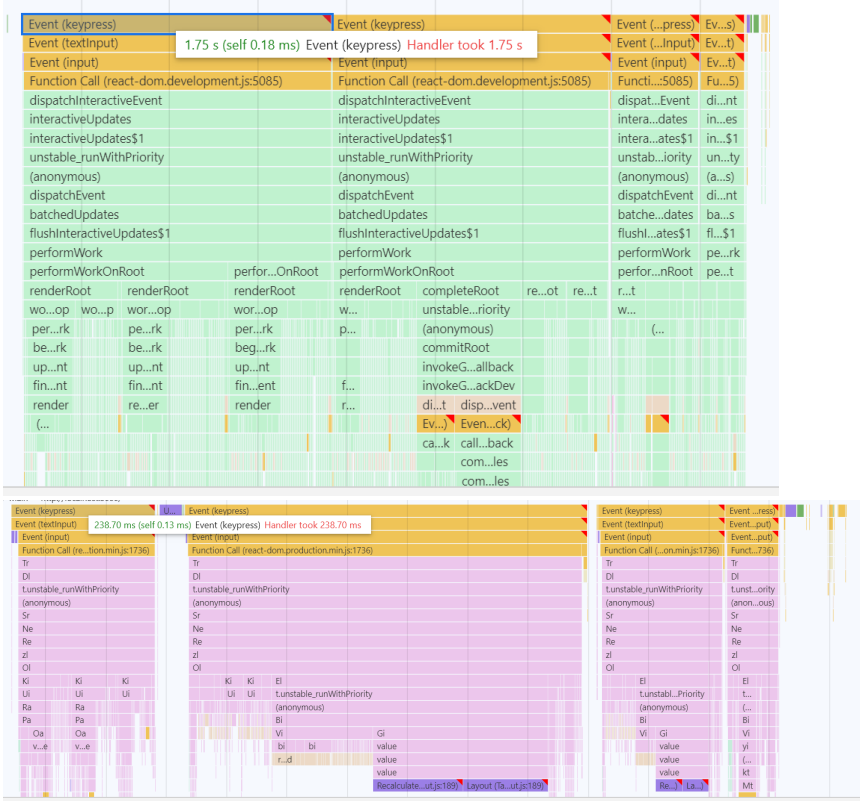
```
render() {
        const isVIP = this.state.isVIP;
        return (
                <div>
                        Dein VIP Status ist <b>{isVIP ? 'aktiv' : 'nicht aktiv'}</b>.
                </div>
        );
}

render() {
        const notifications = this.state.notifications;
        return (
                <div>
                        { notifications > 0 &&
                                <p> Du hast neue Benachrichtigungen! </p>
                        }
                </div>
        );
```

Vue hat somit bei Inline Conditional Rendering in Hinsicht der Leserlichkeit (mit v-for / v-if) des Codes klar den Vorteil.

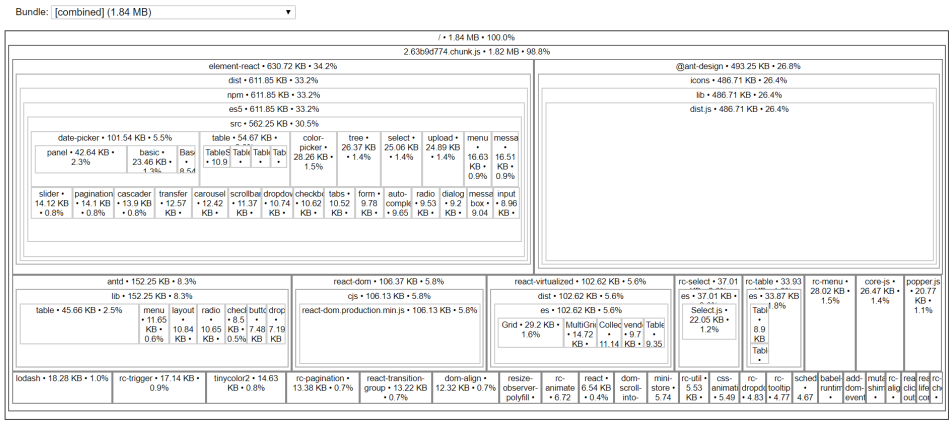
Performance

1. Die Performance in React hängt zunächst davon ab, ob man im Production oder im Development Modus ist. In der Folgenden Grafik wurde die selbe Aktion auf einer Tabelle mit 500 Einträgen (in Elements UI) ausgeführt und die Response time gemessen:



Wie man sehen kann, brauchte die selbe Aktion im Development Modus ganze 1.75 Sekunden, während im Produktions Modus nur 238ms gebraucht wurden. Allerdings sind 238ms immernoch nicht gut, jedoch liegt das mehr an Elements UI und an der relativ großen Tabelle (500 Einträge). Eine UI Response-time von ~60ms sollte angestrebt werden und wird auch mit anderen UI Frameworks, welche für große Tabellen angepasst sind (AntD, React-Virtualized), erreicht.

2. Des Weiteren hängt die Performance von der bundle-size des Projektes ab. Ein >1mb großes Bundle sorgt dafür, dass die initial lade Phase sehr groß ist und insbesondere bei älteren Maschinen / Smartphones zu Problemen führen kann. Im folgenden ist die bundle-size vom Prototypen zu sehen. Man sieht, dass die UI Frameworks den größten Teil des Bundles ausmachen:



3. Wie oben schon beschrieben rerendert React bei jedem State-change die komplette Komponente. `React.PureComponent` wirkt diesem hingegen, jedoch macht sie nur shallow-comparisons. Das heißt wenn der State der Komponente komplexe Datenstrukturen enthält, kann dieses zu False-negatives führen. Dann muss man selber eine `shouldComponentUpdate()` Methode implementieren. Eine Beispiel Implementierung würde zum Beispiel so aussehen:

shouldComponentUpdate()

```
shouldComponentUpdate(nextProps, nextState) {
        // Component should not rerender if myTest gets updated since its not part of the User Interface and
not seen by the User
        if (this.state.myTest !== nextState.myTest) {
                return false;
        }
        return true;
}
```

Wenn man nicht jedes mal eine `shouldComponentUpdate()` implementieren möchte, da es bei sehr vielen Props und States zu Verwirrung kommen könnte, kann man die Komponenten aufteilen und die states verteilen, sodass die Komponenten sich eigenständig updaten. Eine alternative dazu wäre sogenannte Uncontrolled Compontents zu implementieren. Anders als Controlled Components werden die Daten dabei nicht von React bearbeitet sondern von der DOM selber. Dabei benutzt man `refs` um die Daten von der DOM zu bekommen. Ein Beispiel einer unkontrollierten Komponente würde so aussehen:

uncontrolled component

```
class simpleUC extends React.Component {
        constructor(props) {
                super(props);
                this.handleSubmit = this.handleSubmit.bind(this); // Form binden
                this.input = React.createRef(); // Ref hier erstellen
        }

        handleSubmit(event) {
                alert('Dein Name lautet: '+ this.input.current.value); // Ref Daten hier aus der DOM bekommen
                event.preventDefault();
        }

        render() {
                return(
                        <form onSubmit={this.handleSubmit}>
                                <label>
                                        Name:
                                        <input type="text" ref={this.input} /> // Ref hier anbinden
                                </label>
                                <input type="submit" value="Submit" />
                        </form>
                );
        }
}
```

4. Für dbh Zwecke empfiehlt sich `react-virtualized` zu benutzen, da dieses UI Framework für größere Tabellen optimiert ist. Dieses Framework rendert nur die DOM-Elemente die auch wirklich vom User gesehen werden. Es handelt sich auch hierbei wie Elements UI um ein Framework, welches die meisten Funktionen schon selber implementiert. Eine Demo kann hier ausprobiert werden und die Dokumentation befindet sich hier.

Zusätzliche Tools

1. Für den Prototypen wurde als Entwicklungsumgebung Visual Studio Code verwendet. VSCode bringt neben einem großen Erweiterungsmarketplace, einen nativen Debugging Tool für React & Vue. Dieser kann dann zusätzlich durch Plugins wie "Debugger for Chrome" im Browser direkt debugged werden. Zusätzlich ist die Einbindung von ESLint und Babel relativ einfach. Im Browser kann zusätzlich durch die Entwicklertools (F12 in Chrome) debugged werden. Diese Tools bringen neben einem Inspect-Tool ein Performance profiling tool, womit die Performance der UI-Elemente gemessen werden kann.
2. Als Testing Tool empfiehlt sich das von Facebook selbst entwickelte Tool Jest. Zum Beispiel können Oberflächen Tests mit Snapshot testing durchgeführt werden. Außerdem empfiehlt sich `Enzyme` oder `react-test-renderer` , um den Output von React-Komponenten zu testen. Für Komponententests welche einen Enduser emulieren bzw. simulieren empfiehlt sich `react-testing-library` . Des Weiteren gibt es ein allgemeines Test Utility Tool welches `ReactTestUtils` heißt. Ein Leitfaden kann man sich zum Beispiel hier und hier angucken.

Anmerkung zu Snapshot testing

1. Jest Hauptaugenmerk liegt auf Snapshot testing. Snapshot testing geht mehr in Richtung UI-Komponent testing und testet nicht den Code selbst. Dies empfiehlt sich allerdings nur bei Komponenten die sich nicht oft ändern, da der bei jeder Änderung der Komponente ein neuer Snapshot angelegt werden muss und der alte gelöscht werden muss. Das ist bei dauerhafter Änderung nicht ratsam. Also sollte Snapshot testing nur bei Komponenten durchgeführt werden die soweit stabilized und fertig sind. Zum testen der Methoden und Logik empfiehlt sich wie oben schon erwähnt `react-test-render` .

Evaluation Enzyme + Jest

1. Ein allgemeiner Test ist im Ordner `__tests__` vorhanden und testet die Funktionalitäten von Enzyme in Verbindung mit Jest matchern. Eine genaue Erläuterung des Tests wird im nachfolgenden Abschnitt bereitgestellt.

main.test.js

```
/* eslint-disable */

import React from 'react';
import { mount } from 'enzyme';
import LocalePerson from '../LocalePerson';

describe ('<LocalePerson />', () => {
    it ('General Test', () => {
        // Full mount
        const wrapper = mount(<LocalePerson />);
        // Copy initial state for comparisons later
        const initState = wrapper.state();
        // Copy child state for comparisons later
        const listPaneState = wrapper.find('ListPane').instance().state;
        // Check if the states has been defined
        expect(initState).toBeDefined();
        expect(listPaneState).toBeDefined();
        // See if 4 Buttons have been rendered (Force Rerender, Copy, Delete, New)
        expect(wrapper.find('Button')).toHaveLength(4);
        // Select the copyButton
        const copyBtn = wrapper.find('Button').at(1);
        // Simulate onClick without having anything selected
        copyBtn.simulate('click');
        // Checking if state has been modified => it should not have been modified (testing internal
implementation is a bad idea: https://www.valentinog.com/blog/testing-react/)
        expect(wrapper.state().table).toEqual(initState.table);
        // Checking if table still has x rows => it should still have x entries since no selection have been
made (Testing the visible UI is better)
        expect(wrapper.find('.el-table__row')).toHaveLength(initState.table.length);
        // Simulate a selection in the table row 2 cell 2
        wrapper.find('.el-table_1_column_2').at(2).simulate('click');
        // Since we made a selection the state's myPerson should be changed
        expect(wrapper.state().myPerson).not.toEqual(initState.myPerson);
        // Simulate onClick again while having something selected
        copyBtn.simulate('click');
        // Table state should now be modified and not be equal to the initial state anymore
        expect(wrapper.state().table).not.toEqual(initState.table);
        // Table state should now have x+1 Elements since we copied something (Again testing the visible UI
instead of comparing state.table lenghts)
        expect(wrapper.find('.el-table__row')).toHaveLength(initState.table.length+1);
```

```
        // Since we copied row 2 we're gonna check if it has been copied successfully
        // Again we could just test the internal implementation i.e the table state but testing the visible
UI is better since changes in the
        // implementation wont affect this test.
        const copiedPerson = wrapper.find('.el-table_1_column_2').at(2);
        const personUnderCopied = wrapper.find('.el-table_1_column_2').at(3);
        const personAboveCopied = wrapper.find('.el-table_1_column_2').at(1);
        // .html to compare the rendered HTML markup
        expect(copiedPerson.html()).toEqual(personUnderCopied.html());
        expect(copiedPerson.html()).not.toEqual(personAboveCopied.html());
    })
})
```

Zunächst habe ich mich dazu entschieden die Main-Komponente vollständig zu mounten, da man so direkt Zugriff auf die Child-Komponenten hat, ohne sie mocken zu müssen. Außerdem können wir so direkt Änderungen an der DOM feststellen. Eine Dokumentation zur Enzyme mounting-API befindet sich hier. Nachdem wir die Main-Komponente gemountet haben, haben wir auch direkt Zugriff auf ihren React-State sowie auf die States von ihren Child-Komponenten wie in Zeile 12 bis 17 zu sehen ist. Wir können jegliche DOM Elemente mit Enzymes find Funktion finden und sie mit Jest-matchern vergleichen. Zum Beispiel rendert unser React-prototyp insgesamt 4 Buttons (Force Rerender zu debug zwecken, Copy, Delete und New). Mit Jest's Expect können wir testen, ob wirklich 4 Buttons auf der DOM gerendert wurden oder nicht. Des Weiteren können wir mit Enzyme auf jedem Wrapper Events simulieren. So können wir zum Beispiel testen, ob onClick Methoden wirklich ausgeführt werden oder nicht.

2. Da wir hier eine Web UI bauen ist es öfter mehr sinnvoll, die UI bzw. die gerenderten DOM-Elemente selber zu testen, anstatt die Implementierung zu testen (eine Erklärung dazu befindet sich hier). Die sichtbare UI zu testen ist meistens besser, da Implementierungsänderungen die Tests nicht beeinflussen und somit unabhängig von der Implementierung geschrieben werden können. Allerdings bietet Enzyme bei gemounteten Komponenten natürlich auch die Möglichkeit an, deren interne Implementierung zu vergleichen (wie zum Beispiel in Zeile 25 zu sehen ist).
3. Allgemein lässt sich sagen, dass mit Enzyme und Jest komplexe Unit-Tests möglich sind, welche auch unabhängig von der internen Implementierung funktionieren, da wir direkt die gerenderte Oberfläche testen bzw. miteinander vergleichen können.
4. Der Jest-Matcher ist ebenfalls für Vue-Anwendungen verfügbar, jedoch muss zum mounten bei Vue ein anderes Framework als Enzyme verwendet werden. (zum Beispiel Vue Test Utils).
5. Die Debugging-Möglichkeiten unter Visual Studio Code für Jest sind hervorragend, da Jest selber ein Plugin dafür anbietet: Jest Plugin für Visual Studio Code