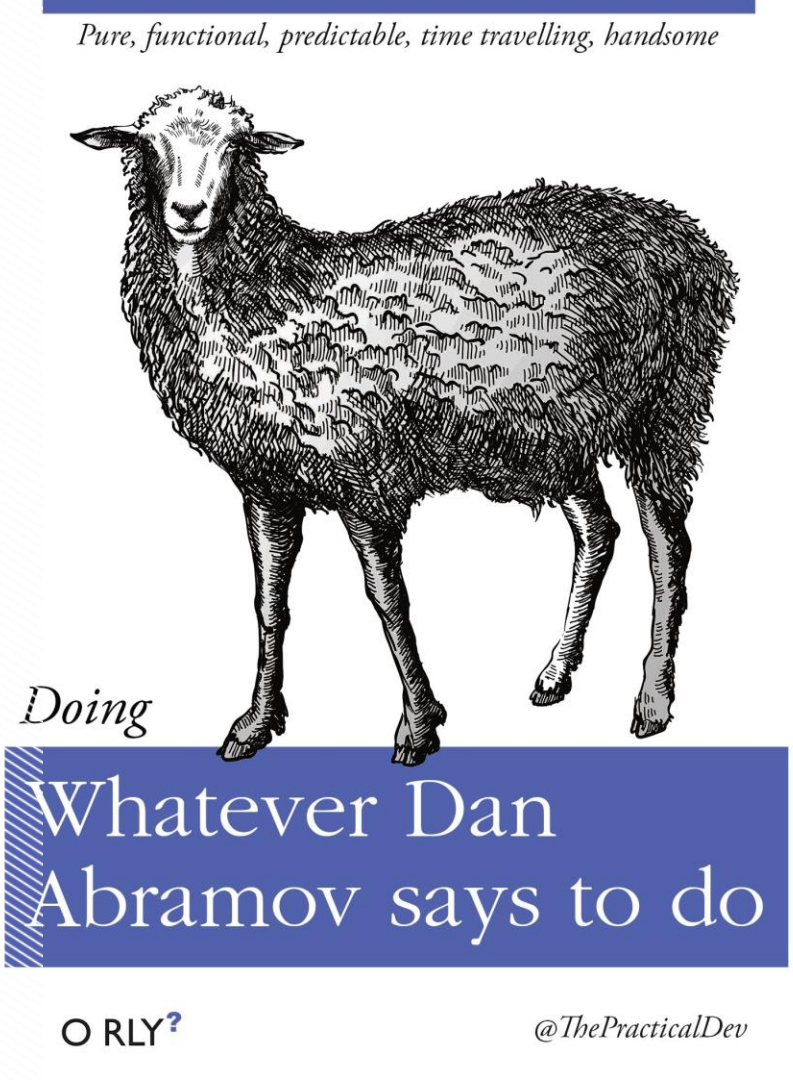
React

or reinventing the wheel



Frontend Junior Program - 2022

“I've seen things you people wouldn't believe. Attack ships on fire off the shoulder of Orion. I watched C-beams glitter in the dark near the Tannhäuser Gate. All those moments will be lost in time, like tears in rain.”

Of course, we are talking about the frameworks…

Refactoring sites into XHTML and developing AngularJS applications with TypeScript are our *C-beams* moments. And while AngularJS and XHTML are long gone, you may think that React Hooks is the way to go for many years. Think twice.

Even replicants do have a life-span: 4 years. React Hooks are 3 years old now.

So, you still could have some years left with Hooks

And that is pretty reasonable. *Hooks* and the *React* are just tools. You learn these not just to use them, but to give birth to beautiful, complex things with full of color and life.

You want to learn React or web-development? You'd probably need to *learn how to create*.

Let's create then!

Preliminary steps

The implementation, however, should be done very carefully

We have 3 important elements to consider:

- *the starting point*
- *the goal*
- *and the way*

start: it is easy, we have nothing

goal: we'd like to create web sites on the client-side, exclusively – no backend rendering is involved

way: we don't know yet, but we'll figure it out – step by step

before any move, we need a plan: we need to understand the actual status and the result we'd like achieve

let's formalize it!

a role: who? → **As a** developer,

the goal: what? → **I want** to have a tool for web-development

a reason: why? → **so that** I can develop web sites on the client-side only

Creating an element

```
<html>
  <head></head>
▼ <body>
  ▼ <div id="root">
      <span>You've done a man's job, sir.</span> == $0
    </div>
  </body>
</html>
```

You've done a man's job, sir.

we append the component to the container

creating an element is simply…

…appending the content to the element…

…and returning the result

we only need an element with id: “root” to start

Welcome | Elements | Memory | Console | Sources | Network | Performa

top | Filter | Default levels | No Issues

```
> function render(component, container) {
    container.append(component);
  }

  function createElement(tagName, content) {
    const element = document.createElement(tagName);

    element.append(content);

    return element;
  }

  let rootComponent = createElement('span', 'You\'ve done a man\'s job, sir.');

  render(rootComponent, document.getElementById('root'));
```

More + nested elements

You've done a man's job, sir.
I guess you're through, huh?

```
▼<span>
   "You've done a man's job, sir."
   <br>
   "I guess you're through, huh?"
 </span>
```

Welcome | Elements | Memory | Console | Sources | Netwo

top | Filter | Default levels

we can add more elements in one step

concat works nicely with an array or a single value as well

declaring many elements

works recursively as well

```
/* Library code */

function render(component, container) {
    container.append(component);
}

function createElement(tagName, ...children) {
    const element = document.createElement(tagName);

    [].concat(children).forEach(child => element.append(child));

    return element;
}

/* Component code */

let rootComponent = createElement('span',
    'You\'ve done a man\'s job, sir.',
    createElement('br'),
    'I guess you\'re through, huh?'
);

render(rootComponent, document.getElementById('root'));
```

Analyze this!

```
/* Library code */

function render(component, container) {
    container.append(component);
}

function createElement(tagName, ...children) {
    const element = document.createElement(tagName);

    [].concat(children).forEach(child => element.append(child));

    return element;
}

/* Component code */

let rootComponent = createElement('span',
    'You\'ve done a man\'s job, sir.',
    createElement('br'),
    'I guess you\'re through, huh?'
);

render(rootComponent, document.getElementById('root'));
```

we were able to separate the library code (remember: this is our goal), and the component code nicely

append is a code duplication, we could use the general *render* here

this is not enough! we want to have life in our components: we need a place for code, that can run *when* the component…

we don't know yet *when*: when the component has been created? or when the component will be attached to the DOM?

We are using Class based components now

We could use simply functions, but an object not just provides code to run and variables, but it can:

- *run code when initializing*
- *have internal state*
- *have methods*

```
/* Library code */

class Component {
}

function render(element, container) {
    let isCustomElement = Component.prototype.isPrototypeOf(element);

    const node = isCustomElement ? element.render() : element;

    container.append(node);
}

function createElement(type, ...children) {
    const element = typeof type === 'string'
        ? document.createElement(type)
        : new type();

    [].concat(children).forEach(child => render(child, element));

    return element;
}

/* Component code */

class Root extends Component {
    render() {
        return createElement('span',
            'You\'ve done a man\'s job, sir.',
            createElement('br'),
            'I guess you\'re through, huh?'
        );
    }
};

render(createElement(Root), document.getElementById('root'));
```

we need a base class

we have to handle our custom components specially, because we need a DOM node to attach at the end, so we'd need to render that

now it is not *tagName* but type, because it can be a custom component as well

instead of directly appending, now we have to render before that,

another render: a component method

now we can use the same *createElement* in both renders (one is the global render, one is the method

```
/* Library code */

class Component {
}

function render(element, container) {
    let isCustomElement = Component.prototype.isPrototypeOf(element);

    const node = isCustomElement ? element.render() : element;

    container.append(node);
}

function createElement(type, ...children) {
    const element = typeof type === 'string'
        ? document.createElement(type)
        : new type();

    [].concat(children).forEach(child => render(child, element));

    return element;
}

/* Component code */

class Root extends Component {
    render() {
        return createElement('span',
            'You\'ve done a man\'s job, sir.',
            createElement('br'),
            'I guess you\'re through, huh?'
        );
    }
};

render(createElement(Root), document.getElementById('root'));
```

```
/* Library code */

class Component {
    constructor(props) {
        for (let prop in props) {
            this[prop] = props[prop];
        }
    }

    render() {
    }
}

function render(element, container) {
    let isCustomElement = Component.prototype.isPrototypeOf(element);
    const node = isCustomElement ? element.render() : element;

    container.append(node);
}

function createElement(type, props, ...children) {
    const element = typeof type === 'string'
        ? document.createElement(type)
        : new type(props);

    [].concat(children).forEach(child => render(child, element));

    return element;
}
```

props are immutable, we have to copy them to *this*

now we have props!
with props we can pass data to components

passing props via constructor

```
/* Component code */

class Sir extends Component {
    constructor(props) {
        super(props);
    }

    render() {
        return createElement('i', null, this.str );
    }
};

class Root extends Component {
    render() {
        return createElement('span', null,
            'You\'ve done a man\'s job, ',
            createElement(Sir, {
                str: 'sir:',
            }),
            createElement('br'),
            'I guess you\'re through, huh?'
        );
    }
};

render(createElement(Root), document.getElementById('root'));
```

null, if no props

passing data

Let’s have a state!

We have a state now

… and a Clock component, but the time is static at this point.

a new component

we use the *time* from props

we have now a state

props is an object

You've done a man's job, *sir: 16:13:26*
I guess you're through, huh?

Welcome | Elements | Memory | Console | Sources | Network | Performance

top | Filter | Default levels | No Issues

```
class Clock extends Component {
    constructor(props) {
        super(props);
    }

    render() {
        return createElement('b', null,
            this.time
        );
    }
}

class Sir extends Component {
    constructor(props) {
        super(props);
        this.state = {date: new Date()};
    }

    render() {
        return createElement('i', null,
            this.str + ' ',
            createElement(Clock, {time: this.state.date.toLocaleTimeString()})
        );
    }
};
```

```
/* Framework code */

class Component {
    constructor(props) {
        this.current = {};
        for (let prop in props) {
            this[prop] = props[prop];
        }
    }

    render() {
    }

    setState(state) {
        this.state = {
            ...this.state,
            ...state
        };

        const node = this.render();
        this.current.replaceWith(node);
        this.current = node;
    }
}

function render(element, container) {
    let isCustomElement = Component.prototype.isPrototypeOf(element);
    const node = isCustomElement ? element.render() : element;

    container.append(node);

    if (isCustomElement) {
        element.current = node;
    }
}
```

merging the state

updating the DOM

we need to store the current node to be able to replace with the new one

```
class Sir extends Component {
    constructor(props) {
        super(props);
        this.state = {date: new Date()};
        setInterval(
            () => this.tick(),
            1000
        );
    }

    tick() {
        this.setState({date: new Date()});
    }

    render() {
        return createElement('i', null,
            this.str + ' ',
            createElement(Clock, {time: this.state.date.toLocaleTimeString()})
        );
    }
};
```

the clock is really ticking now, we just need to tell the component to update the DOM somehow, for that, we call the *setState*

```
/* Framework code */

class Component {
    constructor(props) {
        this.current = {};
        for (let prop in props) {
            this[prop] = props[prop];
        }
    }

    componentDidMount() {
    }

    render() {
    }

    setState(state) {
        this.state = {
            ...this.state,
            ...state
        };

        const node = this.render();
        this.current.replaceWith(node);
        this.current = node;
    }
}

function render(element, container) {
    let isCustomElement = Component.prototype.isPrototypeOf(element);
    const node = isCustomElement ? element.render() : element;

    container.append(node);

    if (isCustomElement) {
        element.current = node;
        element.componentDidMount();
    }
}
```

now we have a life-cycle method!

componentDidMount really runs after the component has been mounted to DOM

```
class Sir extends Component {
    constructor(props) {
        super(props);
        this.state = {date: new Date()};
    }

    componentDidMount() {
        setInterval(
            () => this.tick(),
            1000
        );
    }

    tick() {
        this.setState({date: new Date()});
    }

    render() {
        return createElement('i', null,
            this.str + ' ',
            createElement(Clock, {time: this.state.date.toLocaleTimeString()})
        );
    }
};
```

it is the proper place of initiating something once

We are done!

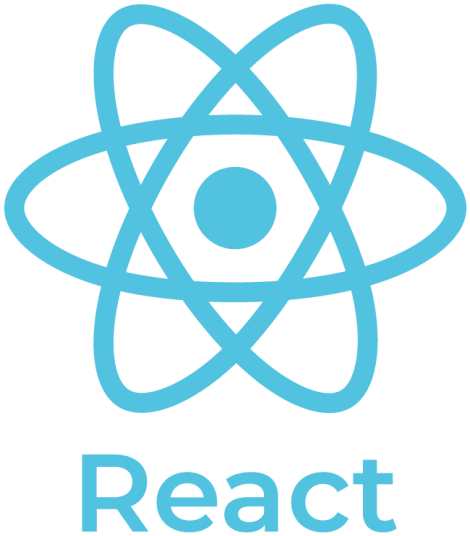
Now we have a library, which

- *is reactive, able to react to prop or state changes*
  (it can be optimized, though, DOM manipulation is expensive, so we can add an internal (shadow) DOM and diff later to change only what is needed)
- *is component-based, every module is encapsulated, and concerns are separated*
  (functional components can be added easily)
- *does have internal state management*
- *uses immutable data flow, via props*
- *has a life-cycle method,*
  (other life-cycle methods can be added later)
- *is less than 50 LOC (lines of code)* ¯_(ツ)_/¯

```
/* Framework code */

class Component {
    constructor(props) {
        this.current = {};
        for (let prop in props) {
            this[prop] = props[prop];
        }
    }

    componentDidMount() {
    }

    render() {
    }

    setState(state) {
        this.state = {
            ...this.state,
            ...state
        };

        const node = this.render();
        this.current.replaceWith(node);
        this.current = node;
    }
}

function render(element, container) {
    let isCustomElement = Component.prototype.isPrototypeOf(element);
    const node = isCustomElement ? element.render() : element;

    container.append(node);

    if (isCustomElement) {
        element.current = node;
        element.componentDidMount();
    }
}

function createElement(type, props, ...children) {
    const element = typeof type === 'string'
        ? document.createElement(type)
        : new type(props);

    [].concat(children).forEach(child => render(child, element));

    return element;
}
```

We just need a proper name…

If that is reactive, what if we just call it…



Ahh, somebody has invented our library, already…

Well, our library does not exactly equal to React

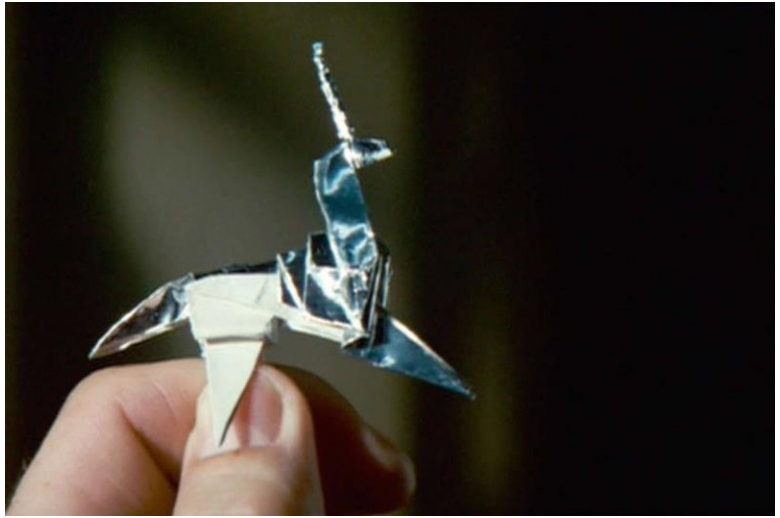
but as you can see, is very similar. Even the basic interfaces are the same.

The most problematic part of understanding an SPA library / framework is to get a grasp on why the actual elements are needed and how they are integrated together.

React, in its core, is very simple. Maybe not that *50 lines simple*, but the concepts are.

If you understand these, then you don't really need to be afraid about what is coming after the Hooks.

You will be prepared, and the Hooks will be your C-beams moment forever.



It's to bad our library won't live, but then again who does?

Q&A

edu_hu@epam.com