Why Frameworks matter - RWieruch

There are many people entering the field of web development right now. It can be an overwhelming experience for beginners to get to know all the tools which are used in modern web development. The historical gap between running HTML in Netscape and today widens with each of these new tools which are put on top of the tech stack. At some point, it makes no sense anymore for beginners to learn jQuery. Students will jump straight on their favorite framework after learning vanilla JavaScript. What's missing for these people is all the historical knowledge from the gap in between.

In this article, I want to focus on the leap from vanilla JavaScript to a modern library like React. When people are going to use such a library, they most often never experienced the struggle from the past which led to these solutions. The question to be asked: Why did we end up with these libraries? I want to showcase why a library like React matters and why you wouldn't want to implement applications in vanilla JavaScript anymore. The whole story can be applied in analogy to any other library/framework such as Vue, Angular or Ember.

I want to showcase how a small application can be build in vanilla JavaScript and React. If you are new to web development, it should give you a clear comparison why you would want to use a library to build a larger application in JavaScript. The following small application has just the right size for vanilla JavaScript, but it shows clear trends why you would choose a library once you are going to scale it. You can checkout the finished applications in this GitHub repository. It would be great to find contributors to add implementations for other libraries/frameworks as well.

Solving a problem in vanilla JavaScript

Let's build together an application in vanilla JavaScript. The problem: Search stories from Hacker News and show the result in a list in your browser. The application only needs an input field for the search request and a list to show the result. If a new search request is made, the list has to be updated in the browser.

In a folder of your choice, create a *index.html* file. Let's write a couple of lines of HTML in this file. First, there has to be HTML boilerplate to render the content to the browser.

```
<!DOCTYPE html>
<html>
```

```
  <head>
    <title>Vanilla JavaScript</title>
  </head>
  <body>
  </body>
  <script src="index.js"></script>
</html>
```

The important part is the imported *index.js* file. That's the file where the vanilla JavaScript code will end up. You can create it next to your *index.html* file. But before you start to write JavaScript, let's add some more HTML. The application should show an input field and a button to request data based on a search query from the input field.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Vanilla JavaScript</title>
  </head>
  <body>
    <div id="app">
      <h1>Search Hacker News with vanilla
JavaScript</h1>
      <input id="searchInput" />
      <button id="searchButton">Search</button>
    </div>
  </body>
  <script src="index.js"></script>
</html>
```

You might have noticed that there is no container to show the requested content yet. In a perfect world, there would be some kind of element which has multiple elements itself to show the requested stories from Hacker News. Since this content is unknown before the request happens, it's a better approach to render it dynamically after the request is made. You will do this in JavaScript by using the DOM API for HTML manipulations in the next part.

The HTML element with the id `app` can be used to hook JavaScript into the DOM later on. In addition, the button element can be used to assign a click event listener. That's the perfect place to start the JavaScript code. Let's start with the *index.js* file.

```
function addButtonEvent() {
```

```
  document.getElementById('searchButton')
    .addEventListener('click', function () {
      // (4) remove old list if there is already a
list
      // (1) get value from input field
      // (2) search list from API with value
      // (3) append list to DOM
    });
};
addButtonEvent();
```

That's basically everything needed for the application. Once the *index.js* file runs, there will be an event listener added to the button element with the id `searchButton`. You can find the button element in your *index.html* file.

The last line is important because someone has to call the function in the first place. The function itself is only the declaration but not the execution of it. All the following implementations will be just a couple of more functions which are executed once someone clicks the button.

The comments in the code show you the business logic which will be implemented step by step. Let's try to keep the code concise here. You can extract the function which is called on a button click event.

```
function addButtonEvent() {
  document.getElementById('searchButton')
    .addEventListener('click', onSearch);
};
function onSearch() {
};
```

Now let's implement the business logic once the button is clicked. There are three things which need to happen. First, you need to retrieve the value from the HTML input field which is used for the search request. Second, you have to make the asynchronous search request. And third, you need to append the result from the search request to the DOM.

```
function addButtonEvent() {
  document.getElementById('searchButton')
    .addEventListener('click', onSearch);
};
function onSearch() {
  doSearch(getValueFromElementById('searchInput'))
    .then(appendList);
```

```
};
```

There are three functions now which you will implement in the following steps. First, let's retrieve the value from the input element with the id `searchInput`.

```
function onSearch() {
  doSearch(getValueFromElementById('searchInput'))
    .then(appendList);
};
function getValueFromElementById(id) {
  return document.getElementById(id).value;
};
```

If you type something in the rendered HTML input field in your browser, it should be picked up once you click the button. Now this value should be used in the `doSearch()` function which you will implement in the second part. The function returns a Promise and thus the `then()` method can be used to append the result (list) in the third step.

```
var BASE_URL = 'https://hn.algolia.com/api/v1/';
function doSearch(query) {
  var url = BASE_URL + 'search?query=' + query +
'&hitsPerPage=200';
  return fetch(url)
    .then(function (response) {
      return response.json();
    })
    .then(function (result) {
      return result.hits;
    });
}
function onSearch() {
  doSearch(getValueFromElementById('searchInput'))
    .then(appendList);
};
```

The function uses the native fetch API which returns a promise. For the sake of simplification, I kept out the error handling in this scenario which could be implemented in a `catch()` block. The request is made to the Hacker News API and the value from the input field is inserted by using string concatenation. Afterward, the response is transformed and only the `hits`

(list) are returned from the result. The third step is to append the list to the DOM.

```
function onSearch() {
  doSearch(getValueFromElementById('searchInput'))
    .then(appendList);
};
function appendList(list) {
  var listNode = document.createElement('div');
  listNode.setAttribute('id', 'list');

document.getElementById('app').appendChild(listNode)
;
  // append items to list
};
```

First, you create a new HTML element and second you give the element an id attribute to check. This id can be used later on to check whether there is already a list in the DOM once a second request is made. Third, you append the new element to your DOM by using the HTML element with the id `app` which you can find in the *index.html* file. What's missing is appending the list of items.

```
function onSearch() {
  doSearch(getValueFromElementById('searchInput'))
    .then(appendList);
};
function appendList(list) {
  var listNode = document.createElement('div');
  listNode.setAttribute('id', 'list');

document.getElementById('app').appendChild(listNode)
;
  list.forEach(function (item) {
    var itemNode = document.createElement('div');

itemNode.appendChild(document.createTextNode(item.ti
tle));
    listNode.appendChild(itemNode);
  });
};
```

For each item in the list, you create a new HTML element, append text to the element and append the element to the list HTML element. You can extract the function to make it concise again. Therefore you have to use a higher order function to pass the list element to the function.

```
function onSearch() {
  doSearch(getValueFromElementById('searchInput'))
    .then(appendList);
};
function appendList(list) {
  var listNode = document.createElement('div');
  listNode.setAttribute('id', 'list');

document.getElementById('app').appendChild(listNode)
;
  list.forEach(appendItem(listNode));
};
function appendItem(listNode) {
  return function (item) {
    var itemNode = document.createElement('div');

itemNode.appendChild(document.createTextNode(item.ti
tle));
    listNode.appendChild(itemNode);
  };
};
```

That's it for the implementation of the three steps. First, retrieve the value from the input field, Second, perform an asynchronous request with the value to retrieve the list from a result from the Hacker News API. And third, append the list and item elements to your DOM.

Last but not least, there is one crucial part missing. You shouldn't forget to remove the list from the DOM when requesting a new list from the API. Otherwise, the new result from the search request will just be appended to your previous result in the DOM.

```
function onSearch() {
  removeList();
  doSearch(getValueFromElementById('searchInput'))
    .then(appendList);
};
function removeList() {
  var listNode = document.getElementById('list');
  if (listNode) {
```

```
    listNode.parentNode.removeChild(listNode);
  }
}
```

You can see that it was quite some work to solve the defined problem from the article. There needs to be someone in charge of the DOM. The DOM update happens in a very naive way here, because it just removes a previous result if there is already one and appends the new result to the DOM again. Everything works just fine to solve the defined problem, but the code gets complex once you add functionality or extend the features of the application.

If you haven't npm installed, install it first via node. Finally, you can test your two files as application in your local browser by installing a HTTP server on the command line with npm.

```
npm install http-server -g
```

Afterwards, you can start the HTTP server from the command line in the directory where you have created your index.html and index.js files:

```
http-server
```

The output should give you a URL where you can find your application in the browser.

Solving the same problem in React

In this part of the article, you are going to solve the same problem with React. It should give you a way to compare both solutions and maybe convince you why a library such as React is a fitting tool to solve such problems.

The project will consist again of a *index.html* and *index.js* file. Our implementation starts again with the HTML boilerplate in the *index.html* file. It requires the two necessary React and ReactDOM libraries. The latter is used to hook React into the DOM and the former for React itself. In addition, the *index.js* is included too.

```
<!DOCTYPE html>
<html>
  <head>
    <title>React</title>
```

```
    <script
src="https://unpkg.com/react@16/umd/react.developmen
t.js"></script>
    <script src="https://unpkg.com/react-
dom@16/umd/react-dom.development.js"></script>
  </head>
  <body>
    <script src="index.js"></script>
  </body>
</html>
```

Second, add Babel to transpile your JavaScript code to vanilla JavaScript, because the following code in your *index.js* file will use non vanilla JavaScript functionalities such as JavaScript ES6 classes. Thus you have to add Babel to transpile it to vanilla JavaScript to make it work in all browsers.

```
<!DOCTYPE html>
<html>
  <head>
    <title>React</title>
    <script
src="https://unpkg.com/react@16/umd/react.developmen
t.js"></script>
    <script src="https://unpkg.com/react-
dom@16/umd/react-dom.development.js"></script>
    <script src="https://unpkg.com/babel-
standalone@6.26.0/babel.min.js"></script>
  </head>
  <body>
    <script type="text/babel" src="index.js">
</script>
  </body>
</html>
```

Third, you have to define an element with an id. That's the crucial place where React can hook into the DOM. There is no need to define further HTML elements in your *index.html* file, because everything else will be defined in your React code in the *index.js* file.

```
<!DOCTYPE html>
<html>
  <head>
    <title>React</title>
```

```
    <script
src="https://unpkg.com/react@16/umd/react.developmen
t.js"></script>
    <script src="https://unpkg.com/react-
dom@16/umd/react-dom.development.js"></script>
    <script src="https://unpkg.com/babel-
standalone@6.26.0/babel.min.js"></script>
  </head>
  <body>
    <div id="app" />
    <script type="text/babel" src="index.js">
</script>
  </body>
</html>
```

Let's jump into the implementation in the *index.js* file. First, you can define the search request at the top of your file as you have done before in vanilla JavaScript.

```
var BASE_URL = 'https://hn.algolia.com/api/v1/';
function doSearch(query) {
  var url = BASE_URL + 'search?query=' + query +
'&hitsPerPage=200';
  return fetch(url)
    .then(function (response) {
      return response.json();
    })
    .then(function (result) {
      return result.hits;
    });
}
```

Since you have included Babel in your *index.html* file, you can refactor the last piece of code to JavaScript ES6 by using arrow functions and template literals.

```
const BASE_URL = 'https://hn.algolia.com/api/v1/';
function doSearch(query) {
  const url = `${BASE_URL}search?
query=${query}&hitsPerPage=200`;
  return fetch(url)
    .then(response => response.json())
    .then(result => result.hits);
}
```

In the next part, let's hook a React component in your HTML by using ReactDOM. The HTML element with the id `app` is used to render your first root component with the name App.

```
class App extends React.Component {
  render() {
    return <h1>Hello React</h1>;
  }
}
ReactDOM.render(
  <App />,
  document.getElementById('app')
);
```

The App component uses React's JSX syntax to display HTML. In JSX you can use JavaScript as well. Let's extend the rendered output to solve the defined problem in this article.

```
class App extends React.Component {
  render() {
    return (
      <div>
        <h1>Search Hacker News with React</h1>
        <form type="submit" onSubmit={}>
          <input type="text" onChange={} />
          <button type="text">Search</button>
        </form>
        {/* show the list of items */}
      </div>
    );
  }
}
```

The component renders a form with an input element and a button element. In addition, there is a placeholder to render the list from the search request in the end. The two handlers for the input element and the form submit are missing. In the next step, you can define the handlers in a declarative way in your component as class methods.

```
class App extends React.Component {
  constructor() {
```

```
    super();
    this.onChange = this.onChange.bind(this);
    this.onSubmit = this.onSubmit.bind(this);
  }
  onSubmit(e) {
    e.preventDefault();
  }
  onChange(e) {
  }
  render() {
    return (
      <div>
        <h1>Search Hacker News with React</h1>
        <form type="submit" onSubmit=
{this.onSubmit}>
          <input type="text" onChange=
{this.onChange} />
          <button type="text">Search</button>
        </form>
        {/* show the list of items */}
      </div>
    );
  }
}
```

The last code shows the declarative power of React. You can implement what every handler in your HTML is doing based on well defined class methods. These can be used as callbacks for your handlers.

Each handler has access to React's synthetic event. For instance, it can be used to retrieve the value from the input element in the `onChange()` handler when someone types into the field. You will do this in the next step.

Note that the event is already used in the 'onSubmit()' class method to prevent the native browser behavior. Normally the browser would refresh the page after a submit event. But in React you don't want to refresh the page, you just want to let React deal with it.

Let's enter state handling in React. Your component has to manage state: the value in the input field and the list of items which is retrieved from the API eventually. It needs to know about those state in order to retrieve the value from the input field for the search request and in order to render the list eventually. Thus, you can define an initial state for the component in its constructor.

```
class App extends React.Component {
```

```
  constructor() {
    super();
    this.state = {
      input: '',
      list: [],
    };
    this.onChange = this.onChange.bind(this);
    this.onSubmit = this.onSubmit.bind(this);
  }
  ...
}
```

Now, you can update the state for the value of the input field by using React's local state management. In a React component, you have access to the `setState()` class method to update the local state. It uses a shallow merge and thus you don't need to worry about the list state when you update the input state.

```
class App extends React.Component {
  constructor() {
    super();
    this.state = {
      input: '',
      list: [],
    };
    this.onChange = this.onChange.bind(this);
    this.onSubmit = this.onSubmit.bind(this);
  }
  ...
  onChange(e) {
    this.setState({ input: e.target.value });
  }
  ...
}
```

By using `this.state` in your component you can access the state from the component again. You should provide the updated input state to your input element. This way, you take over controlling the state of the element and not the element doesn't do it itself. It becomes a so called controlled component which is a best practice in React.

```
class App extends React.Component {
  constructor() {
```

```
    super();
    this.state = {
      input: '',
      list: [],
    };
    this.onChange = this.onChange.bind(this);
    this.onSubmit = this.onSubmit.bind(this);
  }
  ...
  onChange(e) {
    this.setState({ input: e.target.value });
  }
  render() {
    return (
      <div>
        <h1>Search Hacker News with React</h1>
        <form type="submit" onSubmit=
{this.onSubmit}>
          <input type="text" onChange=
{this.onChange} value={this.state.input} />
          <button type="text">Search</button>
        </form>
        {/* show the list of items */}
      </div>
    );
  }
}
```

Once the local state of a component updates in React, the `render()` method of the component runs again. Thus you have always the correct state available when rendering your elements. If you change the state again, for instance by typing something in the input field, the `render()` method will run for you again. You don't have to worry about creating or removing DOM elements when something changes.

In the next step, you will call the defined `doSearch()` function to make the request to the Hacker News API. It should happen in the `onSubmit()` class method. Once a request resolved successfully, you can set the new state for the list property.

```
class App extends React.Component {
  constructor() {
    super();
    this.state = {
      input: '',
```

```
      list: [],
    };
    this.onChange = this.onChange.bind(this);
    this.onSubmit = this.onSubmit.bind(this);
  }
  onSubmit(e) {
    e.preventDefault();
    doSearch(this.state.input)
      .then((hits) => this.setState({ list: hits
}));
  }
  ...
  render() {
    return (
      <div>
        <h1>Search Hacker News with React</h1>
        <form type="submit" onSubmit=
{this.onSubmit}>
          <input type="text" onChange=
{this.onChange} value={this.state.input} />
          <button type="text">Search</button>
        </form>
        {/* show the list of items */}
      </div>
    );
  }
}
```

The state gets updated once the request fulfils successfully. Once the state is updated, the `render()` method runs again and you can use the list in your state to render your elements by using JavaScript's built-in map functionality.

That's the power of JSX in React, because you can use vanilla JavaScript to render multiple elements.

```
class App extends React.Component {
  constructor() {
    super();
    this.state = {
      input: '',
      list: [],
    };
    this.onChange = this.onChange.bind(this);
    this.onSubmit = this.onSubmit.bind(this);
  }
```

```
  onSubmit(e) {
    e.preventDefault();
    doSearch(this.state.input)
      .then((hits) => this.setState({ list: hits
}));
  }
  ...
  render() {
    return (
      <div>
        <h1>Search Hacker News with React</h1>
        <form type="submit" onSubmit=
{this.onSubmit}>
          <input type="text" onChange=
{this.onChange} value={this.state.input} />
          <button type="text">Search</button>
        </form>
        {this.state.list.map(item => <div key=
{item.objectID}>{item.title}</div>)}
      </div>
    );
  }
}
```

That's it. Both class methods update the state in synchronous or asynchronous way. After the state updated eventually, the `render()` method runs again and displays all the HTML elements by using the current state. There is no need for you to remove or append DOM elements in an imperative way. You can define in a declarative way what you want to display with your component.

You can try out the application the same way as the vanilla JavaScript application. On the command line navigate into your folder and use the http-server to serve the application.

Overall both scenarios which are using vanilla JavaScript and React should have shown you a great comparison of imperative and declarative code. In imperative programming, you describe with your code *how to do something*. That's what you have done in the vanilla JavaScript scenario. In contrast, in declarative programming, you describe with your code *what you want to do*. That's the power of React and of using a library over vanilla JavaScript.

The implementation of both examples is quite small and should show you that the problem can be solved by both approaches. I would argue that the vanilla JavaScript solution is even better suited for this problem. However, once you scale your application, it becomes more complex in vanilla

JavaScript to manage the DOM, DOM manipulations and the application state. There would come a point in time where you would end up with the infamous spaghetti code like it happened for lots of jQuery applications in the past. In React, you keep your code declarative and can describe a whole HTML hierarchy with components. These components manage their own state, can be reused and composed into each other. You can describe a whole component tree with them. React keeps your application readable, maintainable and scalable. It's fairly simple to split up a component into multiple components.

```
class App extends React.Component {
  ...
  render() {
    return (
      <div>
        <h1>Search Hacker News with React</h1>
        <form type="submit" onSubmit=
{this.onSubmit}>
          <input type="text" onChange=
{this.onChange} value={this.state.input} />
          <button type="text">Search</button>
        </form>
        {this.state.list.map(item =>
          <Item key={item.objectID} item={item} />
        )}
      </div>
    );
  }
}
const Item = ({ item }) =>
  <div>{item.title}</div>
```

The last code snippet shows how you can extract another component from the App component. This way, you can scale your component hierarchy and maintain business logic colocated to components. It would be way more difficult in vanilla JavaScript to maintain such code.

You can find all the solutions in this GitHub repository. There is also a solution for JavaScript ES6 which can be used in between of the vanilla JavaScript and React approaches. It would be great to find contributors for implementing examples for Angular, Ember and other solutions too. Feel free to contribute to it :)

If you enjoyed this journey from vanilla JavaScript to React and you decided to learn React, checkout The Road to learn React as your next journey to learn

React. Along the way, you will transition smoothly from vanilla JavaScript to JavaScript ES6 and beyond.

In the end, always remember that there are people working behind the curtains to enable these solutions for you. You can do the contributors a huge favor by cheering them up on Twitter once in a while or by getting involved in open source. After all, nobody wants to build larger applications in vanilla JavaScript anymore. So cherish your library or framework that you are using every day :)

Show Comments