REACT IN PATTERNS

written by

KRASIMIR TSONEV

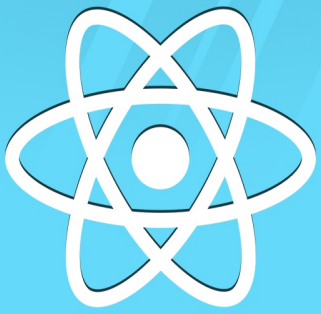


Table of Contents

MISC

React in patterns

A book about common design patterns used while developing with React. It includes techniques for composition, data flow, dependency management and more.

- Web
- PDF
- Mobi
- ePub

REACT IN PATTERNS

written by

KRASIMIR TSONEV

In brief

This cookbook is targeting developers that already have basic understanding of what React is and how it works. It's not meant to be used as a complete how-to guide but as an introduction to popular concepts/design patterns. Paradigms that more or less are introduced by the community. It points you to an abstract thinking. For example, instead of talking about Flux, it talks about data flow. Instead of talking about higher-order components it talks about composition.

The book is highly opinioned. It represents my own understanding of the described patterns and it is possible tha they have a different interpretation around the web. Have this in mind when arguing with someone and using this book as an argument.

Communication

Every React component is like a small system that operates on its own. It has its own state, input and output. In the following section we will explore these characteristics.

Input

The input of a React component is its props. That's how we pass data to it:

```
// Title.jsx
function Title(props) {
  return <h1>{ props.text }</h1>;
}
Title.propTypes = {
  text: PropTypes.string
};
Title.defaultProps = {
  text: 'Hello world'
};

// App.jsx
function App() {
  return <Title text='Hello React' />;
}
```

The `Title` component has only one input (prop) - `text`. The parent component (`App`) should provide it as an attribute while using the `<Title>` tag. Alongside the component definition we also have to define at least `propTypes`. In there we define the type of every property and React hints us in the console if something unexpected gets send. `defaultProps` is another useful option. We may use it to set a default value of component's props so that if the developer forgets to pass them we have a meaningful values.

React is not defining strictly what should be passed as a prop. It may be whatever we want. It could even be another component:

```
function SomethingElse({ answer }) {
  return <div>The answer is { answer }</div>;
}
function Answer() {
  return <span>42</span>;
}

// later somewhere in our application
<SomethingElse answer={ <Answer /> } />
```

There is also a `props.children` property that gives us an access to the child components passed by the owner of the component. For example:

```
function Title({ text, children }) {
  return (
    <h1>
      { text }
      { children }
    </h1>
  );
}
function App() {
  return (
    <Title text='Hello React'>
      <span>community</span>
    </Title>
  );
```

```
  }
```

In this example `<span>community</span>` in `App` component is `children` in `Title` component. Notice that if we don't return `{ children }` as part of the `Title` 's body the `<span>` tag will not be rendered.

(prior v16.3) An indirect input to a component may be also the so called `context` . The whole React tree may have a `context` object which is accessible by every component. More about that in the dependency injection section.

Output

The first obvious output of a React component is the rendered HTML. Visually that is what we get. However, because the prop may be everything including a function we could also send out data or trigger a process.

In the following example we have a component that accepts the user's input and sends it out (`<NameField />`).

```
function NameField({ valueUpdated }) {
  return (
    <input
      onChange={event => valueUpdated(event.target.value) } />
  );
};
class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = { name: '' };
  }
  render() {
    return (
      <div>
        <NameField
          valueUpdated={ name => this.setState({ name }) } />
        Name: { this.state.name }
      </div>
    );
  }
};
```

Very often we need an entry point of our logic. React comes with some handy lifecycle methods that may be used to trigger a process. For example we have an external resource that needs to be fetched on a specific page.

```
class ResultsPage extends React.Component {
  componentDidMount() {
    this.props.getResults();
  }
  render() {
    if (this.props.results) {
      return <List results={ this.props.results } />;
    } else {
      return <LoadingScreen />
    }
  }
}
```

Let's say that we are building a search-results experience. We have a search page and we enter our criteria there. We click submit and the user goes to `/results` where we have to display the result of the search. Once we land on the results page we render some sort of a loading screen and trigger a request for fetching the results in `componentDidMount` lifecycle hook. When the data comes back we pass it to a `<List>` component.

Final thoughts

It is nice that we may think about every React component as a black box. It has its own input, lifecycle and output. It is up to us to compose these boxes. And maybe that is one of the advantages that React offers. Easy to abstract and easy to compose.

Event handlers

React provides a series of attributes for handling events. The solution is almost the same as the one used in the standard DOM. There are some differences like using camel case or the fact that we pass a function but overall it is pretty similar.

```
const theLogoIsClicked = () => alert('Clicked');

<Logo onClick={ theLogoIsClicked } />
<input
  type='text'
  onChange={event => theInputIsChanged(event.target.value) } />
```

Usually we handle events in the component that contains the elements dispatching the events. Like in the example below, we have a click handler and we want to run a function or a method of the same component:

```
class Switcher extends React.Component {
  render() {
    return (
      <button onClick={ this._handleButtonClick }>
        click me
      </button>
    );
  }
  _handleButtonClick() {
    console.log('Button is clicked');
  }
};
```

That's all fine because `_handleButtonClick` is a function and we indeed pass a function to the `onClick` attribute. The problem is that as it is the code doesn't keep the same context. So, if we have to use `this` inside

`_handleButtonClick` to refer the current `Switcher` component we will get an error.

```
class Switcher extends React.Component {
  constructor(props) {
    super(props);
    this.state = { name: 'React in patterns' };
  }
  render() {
    return (
      <button onClick={ this._handleButtonClick }>
        click me
      </button>
    );
  }
  _handleButtonClick() {
    console.log(`Button is clicked inside ${ this.state.name }`);
    // leads to
    // Uncaught TypeError: Cannot read property 'state' of null
  }
};
```

What we normally do is to use `bind` :

```
<button onClick={ this._handleButtonClick.bind(this) }>
  click me
</button>
```

However, this means that the `bind` function is called again and again because we may render the button many times. A better approach would be to create the bindings in the constructor of the component:

```
class Switcher extends React.Component {
  constructor(props) {
    super(props);
    this.state = { name: 'React in patterns' };
    this._buttonClick = this._handleButtonClick.bind(this);
  }
  render() {
    return (
      <button onClick={ this._buttonClick }>
        click me
      </button>
    );
  }
  _handleButtonClick() {
    console.log(`Button is clicked inside ${ this.state.name }`);
  }
};
```

Facebook by the way recommend the same technique while dealing with functions that need the context of the same component.

The constructor is also a nice place for partially executing our handlers. For example, we have a form but want to handle every input in a single function.

```
class Form extends React.Component {
  constructor(props) {
    super(props);
    this._onNameChanged = this._onFieldChange.bind(this, 'name');
    this._onPasswordChanged =
      this._onFieldChange.bind(this, 'password');
  }
  render() {
    return (
      <form>
        <input onChange={ this._onNameChanged } />
        <input onChange={ this._onPasswordChanged } />
      </form>
    );
  }
  _onFieldChange(field, event) {
    console.log(`${ field } changed to ${ event.target.value }`);
  }
};
```

Final thoughts

There is not much to learn about event handling in React. The authors of the library did a good job in keeping what's already there. Since we are using HTML-like syntax it makes total sense that we have also a DOM-like event handling.

Composition

One of the biggest benefits of React is composability. I personally don't know a framework that offers such an easy way to create and combine components. In this section we will explore few composition techniques which proved to work well.

Let's get a simple example. Let's say that we have an application with a header and we want to place a navigation inside. We have three React components - `App`, `Header` and `Navigation`. They have to be nested into each other so we end up with the following dependencies:

```
<App> -> <Header> -> <Navigation>
```

The trivial approach for combining these components is to reference them in the places where we need them.

```
// app.jsx
import Header from './Header.jsx';

export default function App() {
  return <Header />;
}

// Header.jsx
import Navigation from './Navigation.jsx';

export default function Header() {
  return <header><Navigation /></header>;
}

// Navigation.jsx
export default function Navigation() {
  return (<nav> ... </nav>);
}
```

However, by following this approach we introduced couple of problems:

- We may consider the `App` as a place where we do our main composition. The `Header` though may have other elements like a logo, search field or a slogan. It will be nice if they are passed somehow from the `App` component so we don't create a hard-coded dependencies. What if we need the same `Header` component but without the `Navigation`. We can't easily achieve that because we have the two bound tightly together.
- It's difficult to test. We may have some business logic in the `Header` and in order to test it we have to create an instance of the component. However, because it imports other components we will probably create instances of those components too and it becomes heavy to test. We may break our `Header` test by doing something wrong in the `Navigation` component which is totally misleading. *(Note: to some extent shallow rendering solves this problem by rendering only the `Header` without its nested children.)*

Using React's children API

In React we have the handy `children` prop. That's how the parent reads/accesses its children. This API will make our Header agnostic and dependency-free:

```
// App.jsx
export default function App() {
  return (
    <Header>
      <Navigation />
    </Header>
  );
}

// Header.jsx
export default function Header({ children }) {
  return <header>{ children }</header>;
};
```

Notice also that if we don't use `{ children }` in `Header`, the `Navigation` component will never be rendered.

It now becomes easier to test because we may render the `Header` with an empty `<div>`. This will isolate the component and will let us focus on one piece of our application.

Passing a child as a prop

Every React component receives props. As we mentioned already there is no any strict rule about what these props are. We may even pass other components.

```
const Title = function () {
  return <h1>Hello there!</h1>;
}
const Header = function ({ title, children }) {
  return (
    <header>
      { title }
      { children }
    </header>
  );
}
function App() {
  return (
    <Header title={ <Title /> }>
      <Navigation />
    </Header>
  );
};
```

This technique is useful when a component like `Header` needs to take decisions about its children but don't bother about what they actually are. A simple example is a visibility component that hides its children based on a specific condition.

Higher-order component

For a long period of time higher-order components were the most popular way to enhance and compose React elements. They look really similar to the decorator design pattern because we have component wrapping and enhancing.

On the technical side the higher-order component is usually a function that accepts our original component and returns an enhanced/populated version of it. The most trivial example is as follows:

```
var enhanceComponent = (Component) =>
  class Enhance extends React.Component {
    render() {
      return (
        <Component {...this.props} />
      )
    }
  };

var OriginalTitle = () => <h1>Hello world</h1>;
var EnhancedTitle = enhanceComponent(OriginalTitle);

class App extends React.Component {
  render() {
    return <EnhancedTitle />;
  }
};
```

The very first thing that the higher-order component does is to render the original component. It's a good practice to proxy pass the `props` to it. This way we will keep the input of our original component. And here it comes the first big benefit of this pattern - because we control the input

of the component we may send something that the component usually has no access to. Let's say that we have a configuration setting that `OriginalTitle` needs:

```
var config = require('path/to/configuration');

var enhanceComponent = (Component) =>
  class Enhance extends React.Component {
    render() {
      return (
        <Component
          {...this.props}
          title={ config.appTitle }
        />
      )
    }
  };

var OriginalTitle  = ({ title }) => <h1>{ title }</h1>;
var EnhancedTitle = enhanceComponent(OriginalTitle);
```

The knowledge for the `appTitle` is hidden into the higher-order component. `OriginalTitle` knows only that it receives a `prop` called `title` . It has no idea that this is coming from a configuration file. That's a huge advantage because it allows us to isolate blocks. It also helps with the testing of the component because we can create mocks easily.

Another characteristic of this pattern is that we have a nice buffer for additional logic. For example, if our `OriginalTitle` needs data also from a remote server. We may query this data in the higher-order component and again send it as a prop.

```
var enhanceComponent = (Component) =>
  class Enhance extends React.Component {
    constructor(props) {
      super(props);

      this.state = { remoteTitle: null };
    }
    componentDidMount() {
      fetchRemoteData('path/to/endpoint').then(data => {
        this.setState({ remoteTitle: data.title });
      });
    }
    render() {
      return (
        <Component
          {...this.props}
          title={ config.appTitle }
          remoteTitle={ this.state.remoteTitle }
        />
      )
    }
  };

var OriginalTitle  = ({ title, remoteTitle }) =>
  <h1>{ title }{ remoteTitle }</h1>;
var EnhancedTitle = enhanceComponent(OriginalTitle);
```

Again, the `OriginalTitle` knows that it receives two props and has to render them next to each other. Its only concern is how the data looks like not where it comes from and how.

Dan Abramov made a really good point that the actual creation of the higher-order component (i.e. calling a function like `enhanceComponent`) should happen at a component definition level. Or in other words, it's a bad practice to do it inside another React component because it may be slow and lead to performance issues.

Function as a children, render prop

Last couple of months the React community started shifting in an interesting direction. So far in our examples the `children` prop was a React component. There is however a new pattern gaining popularity in which the same `children` prop is a JSX expression. Let's start by passing a simple object.

```
function UserName({ children ) {
  return (
    <div>
      <b>{ children.lastName }</b>,
      { children.firstName }
    </div>
  );
}

function App() {
  const user = {
    firstName: 'Krasimir',
    lastName: 'Tsonev'
  };
  return (
    <UserName>{ user }</UserName>
  );
}
```

This may look weird but in fact is really powerful. Like for example when we have some knowledge in the parent component and don't necessary want to send it down to children. The example below prints a list of TODOs. The `App` component has all the data and knows how to determine whether a TODO is completed or not. The `TodoList` component simply encapsulate the needed HTML markup.

```
function TodoList({ todos, children }) {
  return (
    <section className='main-section'>
      <ul className='todo-list'>{
        todos.map((todo, i) => (
          <li key={ i }>{ children(todo) }</li>
        ))
      }</ul>
    </section>
  );
}

function App() {
  const todos = [
    { label: 'Write tests', status: 'done' },
    { label: 'Sent report', status: 'progress' },
    { label: 'Answer emails', status: 'done' }
  ];
  const isCompleted = todo => todo.status === 'done';
  return (
    <TodoList todos={ todos }>
      {
        todo => isCompleted(todo) ?
          <b>{ todo.label }</b> :
          todo.label
      }
    </TodoList>
  );
}
```

Notice how the `App` component doesn't expose the structure of the data. `TodoList` has no idea that there is `label` or `status` properties.

The so called *render prop* pattern is almost the same except that we use a prop and not `children` for rendering the todo.

```
function TodoList({ todos, render }) {
  return (
    <section className='main-section'>
      <ul className='todo-list'>{
        todos.map((todo, i) => (
          <li key={ i }>{ render(todo) }</li>
        ))
      }</ul>
    </section>
  );
}

return (
  <TodoList
    todos={ todos }
    render={
      todo => isCompleted(todo) ?
        <b>{ todo.label }</b> : todo.label
    } />
);
```

These two patterns, *function as children* and *render prop* are probably one of my favorite ones recently. They provide flexibility and help when we want to reuse code. They are also a powerful way to abstract imperative code.

```
class DataProvider extends React.Component {
  constructor(props) {
    super(props);

    this.state = { data: null };
    setTimeout(() => this.setState({ data: 'Hey there!' }), 5000);
  }
  render() {
    if (this.state.data === null) return null;
    return (
      <section>{ this.props.render(this.state.data) }</section>
    );
  }
}
```

`DataProvider` renders nothing when first gets mounted. Five seconds later we update the state of the component and we render a `<section>` followed by what is `render` prop returning. Imagine that this same component fetches data from a remote server and we want to display it only when it is available.

```
<DataProvider render={ data => <p>The data is here!</p> } />
```

We do say what we want to happen but not how. That is hidden inside the `DataProvider`. These days we used this pattern at work where we had to restrict some UI to certain users having `read:products` permissions. And we used the *render prop* pattern.

```
<Authorize
  permissionsInclude={[ 'read:products' ]}
  render={ () => <ProductsList /> } />
```

Pretty nice and self-explanatory in a declarative fashion. `Authorize` goes to our identity provider and checks what are the permissions of the current user. If he/she is allowed to read our products we render the `ProductList`.

Final thoughts

Did you wonder why HTML is still here. It was created in the dawn of the internet and we still use it. That is because is highly composable. React and its JSX looks like HTML on steroids and as such it comes with the same capabilities. So, make sure that you master the composition because that is one of the biggest benefits of React.

Controlled and uncontrolled inputs

These two terms *controlled* and *uncontrolled* are very often used in the context of forms management. *controlled* input is an input that gets its value from a single source of truth. For example the `App` component below has a single `<input>` field which is *controlled*:

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { value: 'hello' };
  }
  render() {
    return <input type='text' value={ this.state.value } />;
  }
};
```

The result of this code is an input element that we can focus but can't change. It is never updated because we have a single source of truth - the `App` 's component state. To make the input works as expected we have to add an `onChange` handler and update the state (the single source of truth). Which will trigger a new rendering cycle and we will see what we typed.

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { value: 'hello' };
    this._change = this._handleInputChange.bind(this);
  }
  render() {
    return (
      <input
        type='text'
        value={ this.state.value }
        onChange={ this._change } />
    );
  }
  _handleInputChange(e) {
    this.setState({ value: e.target.value });
  }
};
```

On the opposite side is the *uncontrolled* input where we let the browser handles the user's updates. We may still provide an initial value by using the `defaultValue` prop but after that the browser is responsible for keeping the state of the input.

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { value: 'hello' };
  }
  render() {
    return <input type='text' defaultValue={ this.state.value } />
  }
};
```

That `<input>` element above is a little bit useless because the user updates the value but our component has no idea about that. We then have to use `Refs` to get an access to the actual element.

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { value: 'hello' };
    this._change = this._handleInputChange.bind(this);
  }
  render() {
    return (
      <input
        type='text'
        defaultValue={ this.state.value }
        onChange={ this._change }
        ref={ input => this.input = input }/>
    );
  }
  _handleInputChange() {
    this.setState({ value: this.input.value });
  }
};
```

The `ref` prop receives a string or a callback. The code above uses a callback and stores the DOM element into a *local* variable called `input`. Later when the `onChange` handler is fired we get the new value and send it to the `App`'s state.

Using a lot of `refs` is not a good idea. If it happens in your app consider using `controlled` inputs and re-think your components.

Final thoughts

controlled versus *uncontrolled* inputs is very often underrated. However I believe that it is a fundamental decision because it dictates the data flow in the React component. I personally think that *uncontrolled* inputs are kind of an anti-pattern and I'm trying to avoid them when possible.

Presentational and container components

Every beginning is difficult. React is no exception and as beginners we also have lots of questions. Where I'm supposed to put my data, how to communicate changes or how to manage state? The answers of these questions are very often a matter of context and sometimes just practice and experience with the library. However, there is a pattern which is used widely and helps organizing React based applications - splitting the component into presentation and container.

Let's start with a simple example that illustrates the problem and then split the component into container and presentation. We will use a `Clock` component. It accepts a Date object as a prop and displays the time in real time.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = { time: this.props.time };
    this._update = this._updateTime.bind(this);
  }
  render() {
    const time = this._formatTime(this.state.time);
    return (
      <h1>
        { time.hours } : { time.minutes } : { time.seconds }
      </h1>
    );
  }
  componentDidMount() {
    this._interval = setInterval(this._update, 1000);
  }
  componentWillUnmount() {
    clearInterval(this._interval);
  }
```

```
  _formatTime(time) {
    var [ hours, minutes, seconds ] = [
      time.getHours(),
      time.getMinutes(),
      time.getSeconds()
    ].map(num => num < 10 ? '0' + num : num);

    return { hours, minutes, seconds };
  }
  _updateTime() {
    this.setState({
      time: new Date(this.state.time.getTime() + 1000)
    });
  }
};

ReactDOM.render(<Clock time={ new Date() }/>, ...);
```

In the constructor of the component we initialize the component's state which in our case is just the current `time` value. By using `setInterval` we update the state every second and the component is re-rendered. To make it looks like a real clock we use two helper methods - `_formatTime` and `_updateTime`. The first one is about extracting hours, minutes and seconds and making sure that they are following the two-digits format. `_updateTime` is mutating the current `time` object by one second.

The problems

There are couple of things happening in our component. It looks like it has too many responsibilities.

- It mutates the state by itself. Changing the time inside the component may not be a good idea because then only `Clock` knows the current value. If there is another part of the system that depends on this data it will be difficult to share it.

- `_formatTime` is actually doing two things - it extracts the needed information from the date object and makes sure that the values are always presented by two digits. That's fine but it will be nice if the extracting is not part of the function because then it is bound to the type of the `time` object (coming as a prop). I.e. knows specifics about the shape of the data and at the same time deals with the visualization of it.

Extracting the container

Containers know about data, its shape and where it comes from. They know details about how the things work or the so called *business logic*. They receive information and format it so like is easy to be used by the presentational component. Very often we use higher-order components to create containers because they provide a buffer space where we can insert custom logic.

Here's how our `ClockContainer` looks like:

```
// Clock/index.js
import Clock from './Clock.jsx'; // <-- that's the presentational
component

export default class ClockContainer extends React.Component {
  constructor(props) {
    super(props);
    this.state = { time: props.time };
    this._update = this._updateTime.bind(this);
  }
  render() {
    return <Clock { ...this._extract(this.state.time) }/>;
  }
  componentDidMount() {
    this._interval = setInterval(this._update, 1000);
  }
  componentWillUnmount() {
    clearInterval(this._interval);
```

```
  }
  _extract(time) {
    return {
      hours: time.getHours(),
      minutes: time.getMinutes(),
      seconds: time.getSeconds()
    };
  }
  _updateTime() {
    this.setState({
      time: new Date(this.state.time.getTime() + 1000)
    });
  }
};
```

It still accepts `time` (a date object), does the `setInterval` loop and knows details about the data (`getHours` , `getMinutes` and `getSeconds`). In the end renders the presentational component and passes three numbers for hours, minutes and seconds. There is nothing about how the things look like. Only *business logic*.

Presentational component

Presentational components are concerned with how the things look. They have the additional markup needed for making the page pretty. Such components are not bound to anything and have no dependencies. Very often implemented as a stateless functional components they don't have internal state.

In our case the presentational component contains only the two-digits check and returns the `<h1>` tag:

```
// Clock/Clock.jsx
export default function Clock(props) {
  var [ hours, minutes, seconds ] = [
    props.hours,
    props.minutes,
    props.seconds
  ].map(num => num < 10 ? '0' + num : num);

  return <h1>{ hours } : { minutes } : { seconds }</h1>;
};
```

Benefits

Splitting the components in containers and presentation increases the reusability of the components. Our `Clock` function/component may exist in application that doesn't change the time or it's not working with JavaScript's Date object. That's because it is pretty *dummy*. No details about the data are required.

The containers encapsulate logic and we may use them together with different renderers because they don't leak information about the visual part. The approach that we took above is a good example of how the container doesn't care about how the things look like. We may easily switch from digital to analog clock and the only one change will be to replace the `<Clock>` component in the `render` method.

Even the testing becomes easier because the components have less responsibilities. Containers are not concern with UI. Presentational components are pure renderers and it is enough to run expectations on the resulted markup.

Final thoughts

The concept of container and presentation is not new at all but it fits really nicely with React. It makes our applications better structured, easy to manage and scale.

One-way direction data flow

One-way direction data flow is a pattern that works nicely with React. It is around the idea that the components do not modify the data that they receive. They only listen for changes of this data and maybe provide the new value but they do not update the actual data. This update happens following another mechanism in another place and the component just gets re-rendered with the new value.

Let's for example get a simple `Switcher` component that contains a button. If we click it we have to enable a flag in the system.

```
class Switcher extends React.Component {
  constructor(props) {
    super(props);
    this.state = { flag: false };
    this._onButtonClick = e => this.setState({
      flag: !this.state.flag
    });
  }
  render() {
    return (
      <button onClick={ this._onButtonClick }>
        { this.state.flag ? 'lights on' : 'lights off' }
      </button>
    );
  }
};

// ... and we render it
function App() {
  return <Switcher />;
};
```

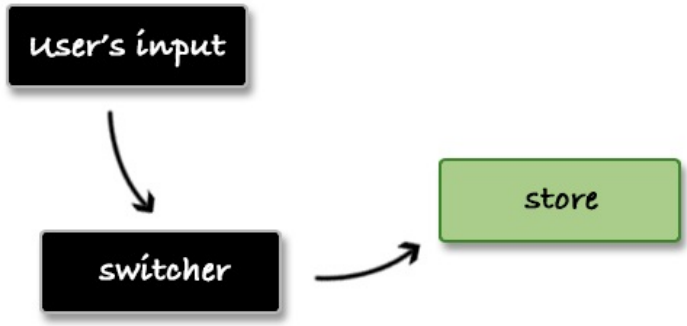
At this moment we have the data inside our component. Or in other words, `Switcher` is the only one place that knows about our `flag`. Let's send it out to some kind of a store:

```
var Store = {
  _flag: false,
  set: function(value) {
    this._flag = value;
  },
  get: function() {
    return this._flag;
  }
};

class Switcher extends React.Component {
  constructor(props) {
    super(props);
    this.state = { flag: false };
    this._onButtonClick = e => {
      this.setState({ flag: !this.state.flag }, () => {
        this.props.onChange(this.state.flag);
      });
    }
  }
  render() {
    return (
      <button onClick={ this._onButtonClick }>
        { this.state.flag ? 'lights on' : 'lights off' }
      </button>
    );
  }
};

function App() {
  return <Switcher onChange={ Store.set.bind(Store) } />;
};
```

Our `Store` object is a singleton where we have helpers for setting and getting the value of the `_flag` property. By passing the getter to the `Switcher` we are able to update the data externally. More or less our application workflow looks like that:

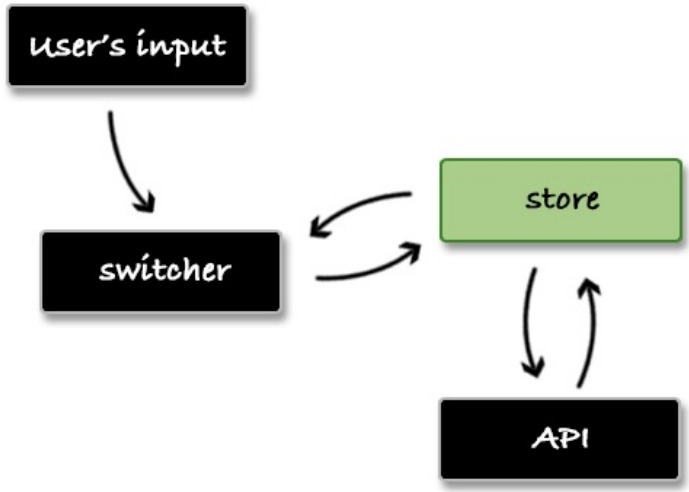


Let's assume that we are saving the flag value to a back-end service via the `Store`. When the user comes back we have to set a proper initial state. If the user left the flag as `true` we have to show *"lights on"* and not the default *"lights off"*. Now it gets tricky because we have the data in two places. The UI and the `Store` have their own states. We have to communicate in both directions from the store to the switcher and from the switcher to the store.

```
// ... in App component
<Switcher
  value={ Store.get() }
  onChange={ Store.set.bind(Store) } />

// ... in Switcher component
constructor(props) {
  super(props);
  this.state = { flag: this.props.value };
  ...
```

Our workflow changes to the following:



All this leads to managing two states instead of one. What if the `Store` changes its value based on other actions in the system. We have to propagate that change to the `Switcher` and we increase the complexity of our app.

One-way direction data flow solves this problem. It eliminates the multiple places where we manage states and deals with only one which is usually the store. To achieve that we have to tweak our `Store` object a little bit. We need logic that allows us to subscribe for changes:

```
var Store = {
  _handlers: [],
  _flag: '',
  subscribe: function(handler) {
    this._handlers.push(handler);
  },
  set: function(value) {
    this._flag = value;
    this._handlers.forEach(handler => handler(value))
  },
  get: function() {
    return this._flag;
  }
};
```

Then we will hook our main `App` component and we'll re-render it every time when the `Store` changes its value:

```
class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = { value: Store.get() };
    Store.subscribe(value => this.setState({ value }));
  }
  render() {
    return (
      <div>
        <Switcher
          value={ this.state.value }
          onChange={ Store.set.bind(Store) } />
      </div>
    );
  }
};
```

Because of this change the `Switcher` becomes really simple. We don't need the internal state and the component may be written as a stateless function.

```
function Switcher({ value, onChange }) {
  return (
    <button onClick={ e => onChange(!value) }>
      { value ? 'lights on' : 'lights off' }
    </button>
  );
};

<Switcher
  value={ Store.get() }
  onChange={ Store.set.bind(Store) } />
```

Final thoughts

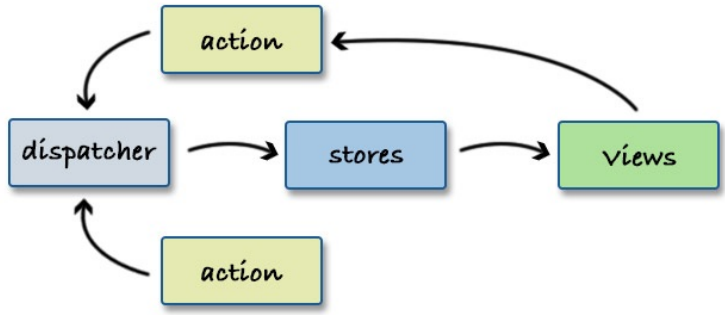
The benefit that comes with this pattern is that our components become dummy representation of the store's data. There is only one source of truth and this makes the development easier. If you are going to take one thing from this book I would prefer to be this chapter. The one-direction data flow drastically changed the way of how I think when designing a feature so I believe it will have the same effect on you.

Flux

I'm obsessed by making my code simpler. I didn't say *smaller* because having less code doesn't mean that is simple and easy to work with. I believe that big part of the problems in the software industry come from the unnecessary complexity. Complexity which is a result of our own abstractions. You know, we (the programmers) like to abstract. We like placing things in black boxes and hope that these boxes work together.

Flux is an architectural design pattern for building user interfaces. It was introduced by Facebook at their F8 conference. Since then, lots of companies adopted the idea and it seems like a good pattern for building front-end apps. Flux is very often used with React. Another library released by Facebook. I myself use React+Flux/Redux in my daily job and I could say that it is simple and really flexible. The pattern helps creating apps faster and at the same time keeps the code well organized.

Flux architecture and its main characteristics



The main actor in this pattern is the *dispatcher*. It acts as a hub for all the events in the system. Its job is to receive notifications that we call *actions* and pass them to all the *stores*. The store decides if it is interested or not and reacts by changing its internal state/data. That change is triggering re-rendering of the *views* which are (in our case) React components. If we have to compare Flux to the well known MVC we may say that the store is similar to the model. It keeps the data and its mutations.

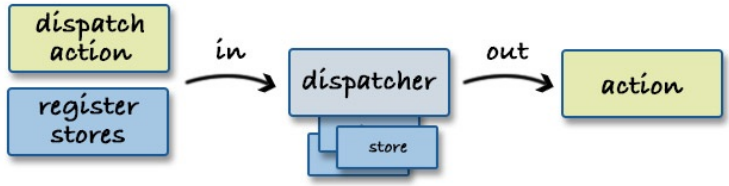
The actions are coming to the dispatcher either from the views or from other parts of the system, like services. For example a module that performs a HTTP request. When it receives the result it may fire an action saying that the request was successful.

Implementing a Flux architecture

As every other popular concept Flux also has some variations. Very often to understand something we have to implement it. In the next few sections we will create a library that provides helpers for building the Flux pattern.

The dispatcher

In most of the cases we need a single dispatcher. Because it acts as a glue for the rest of the parts it makes sense that we have only one. The dispatcher needs to know about two things - actions and stores. The actions are simply forwarded to the stores so we don't necessary have to keep them. The stores however should be tracked inside the dispatcher so we can loop through them:



That's what I started with:

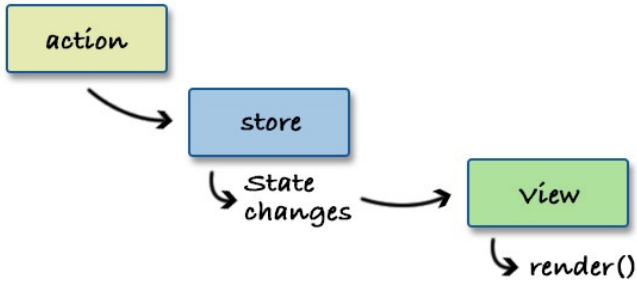
```
var Dispatcher = function () {
  return {
    _stores: [],
    register: function (store) {
      this._stores.push({ store: store });
    },
    dispatch: function (action) {
      if (this._stores.length > 0) {
        this._stores.forEach(function (entry) {
          entry.store.update(action);
        });
      }
    }
  }
};
```

The first thing that we notice is that we *expect* to see an `update` method in the passed stores. It will be nice to throw an error if such method is not there:

```
register: function (store) {
  if (!store || !store.update) {
    throw new Error('You should provide a store that has an `updat
e` method.');
  } else {
    this._stores.push({ store: store });
  }
}
```

Bounding the views and the stores

The next logical step is to connect our views to the stores so we re-render when the state in the stores is changed.



Using a helper

Some of the flux implementations available out there provide a helper function that does the job. For example:

```
Framework.attachToStore(view, store);
```

However, I don't quite like this approach. To make `attachStore` works we expect to see a specific API in the view and in the store. We kind of strictly define new public methods. Or in other words we say "Your views and store should have such APIs so we are able to wire them together". If we go down this road then we will probably define our own base classes which could be extended so we don't bother the developer with Flux details. Then we say "All your classes should extend our classes". This doesn't sound good either because the developer may decide to switch to another Flux provider and has to amend everything.

With a mixin

What if we use React's mixins.

```
var View = React.createClass({
  mixins: [Framework.attachToStore(store)]
  ...
});
```

That's a "nice" way to define behavior of existing React component. So, in theory we may create a mixin that does the bounding for us. To be honest, I don't think that this is a good idea. And it looks like it's not only me. My reason of not liking mixins is that they modify the components in a non-predictable way. I have no idea what is going on behind the scenes. So I'm crossing this option.

Using a context

Another technique that may answer the question is React's context. It is a way to pass props to child components without the need to specify them in every level of the tree. Facebook suggests context in the cases where we have data that has to reach deeply nested components.

> Occasionally, you want to pass data through the component tree without having to pass the props down manually at every level. React's "context" feature lets you do this.

I see similarity with the mixins here. The context is defined somewhere at the top and magically serves props for all the children below. It's not immediately clear where the data comes from.

Higher-Order components concept

Higher-Order components pattern is introduced by Sebastian Markbåge and it's about creating a wrapper component that returns ours. While doing it it has the opportunity to send properties or apply additional logic. For example:

```
function attachToStore(Component, store, consumer) {
  const Wrapper = React.createClass({
    getInitialState() {
      return consumer(this.props, store);
    },
    componentDidMount() {
      store.onChangeEvent(this._handleStoreChange);
    },
    componentWillUnmount() {
      store.offChangeEvent(this._handleStoreChange);
    },
    _handleStoreChange() {
      if (this.isMounted()) {
        this.setState(consumer(this.props, store));
      }
    },
    render() {
      return <Component {...this.props} {...this.state} />;
    }
  });
  return Wrapper;
};
```

`Component` is the view that we want attached to the `store`. The `consumer` function says what part of the store's state should be fetched and send to the view. A simple usage of the above function could be:

```
class MyView extends React.Component {
  ...
}

ProfilePage = connectToStores(MyView, store, (props, store) => ({
  data: store.get('key')
}));
```

That is an interesting pattern because it shifts the responsibilities. It is the view fetching data from the store and not the store pushing something to the view. This of course has its own pros and cons. It is nice because it makes the store dummy. A store that only mutates the data and says "Hey, my state is changed". It is not responsible for sending anything to anyone. The downside of this approach is maybe the fact that we have one more component (the wrapper) involved. We also need the three things - view, store and consumer to be in one place so we can establish the connection.

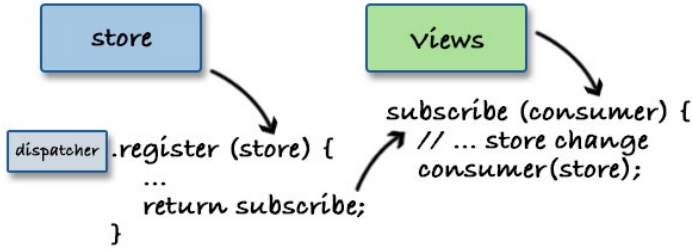
My choice

The last option above, higher-order components, is really close to what I'm searching for. I like the fact that the view decides what it needs. That *knowledge* anyway exists in the component so it makes sense to keep it there. That's also why the functions that generate higher-order components are usually kept in the same file as the view. What if we can use similar approach but not passing the store at all. Or in other words, a function that accepts only the consumer. And that function is called every time when there is a change in the store.

So far our implementation interacts with the store only in the `register` method.

```
register: function (store) {
  if (!store || !store.update) {
    throw new Error('You should provide a store that has an `updat
e` method.');
  } else {
    this._stores.push({ store: store });
  }
}
```

By using `register` we keep a reference to the store inside the dispatcher. However, `register` returns nothing. And instead of nothing it may return a **subscriber** that will accept our consumer functions.



I decided to send the whole store to the consumer function and not the data that the store keeps. Like in the higher-order components pattern the view should say what it needs by using store's getters. This makes the store really simple and there is no trace of presentational logic.

Here is how the register method looks like after the changes:

```
register: function (store) {
  if (!store || !store.update) {
    throw new Error(
      'You should provide a store that has an `update` method.'
    );
  } else {
    var consumers = [];
    var subscribe = function (consumer) {
      consumers.push(consumer);
    };

    this._stores.push({ store: store });
    return subscribe;
  }
  return false;
}
```

The last bit in the story is how the store says that its internal state is changed. It's nice that we collect the consumer functions but right now there is no code that execute them.

According to the basic principles of the flux architecture the stores change their state in response of actions. In the `update` method we send the `action` but we could also send a function `change`. Calling that function should trigger the consumers:

```
register: function (store) {
  if (!store || !store.update) {
    throw new Error(
      'You should provide a store that has an `update` method.'
    );
  } else {
    var consumers = [];
    var change = function () {
      consumers.forEach(function (l) {
        l(store);
      });
    };
    var subscribe = function (consumer) {
      consumers.push(consumer);
    };

    this._stores.push({ store: store, change: change });
    return subscribe;
  }
  return false;
},
dispatch: function (action) {
  if (this._stores.length > 0) {
    this._stores.forEach(function (entry) {
      entry.store.update(action, entry.change);
    });
  }
}
```

Notice how we push `change` together with `store` inside the `_stores` array. Later in the `dispatch` method we call `update` by passing the `action` and the `change` function.

A common use case is to render the view with the initial state of the store. In the context of our implementation this means firing all the consumers at least once when they land in the library. This could be easily done in the `subscribe` method:

```
var subscribe = function (consumer, noInit) {
  consumers.push(consumer);
  !noInit ? consumer(store) : null;
};
```

Of course sometimes this is not needed so we added a flag which is by default falsy. Here is the final version of our dispatcher:

```
var Dispatcher = function () {
  return {
    _stores: [],
    register: function (store) {
      if (!store || !store.update) {
        throw new Error(
          'You should provide a store that has an `update` method'
        );
      } else {
        var consumers = [];
        var change = function () {
          consumers.forEach(function (l) {
            l(store);
          });
        };
        var subscribe = function (consumer, noInit) {
          consumers.push(consumer);
          !noInit ? consumer(store) : null;
        };

        this._stores.push({ store: store, change: change });
        return subscribe;
      }
      return false;
    },
    dispatch: function (action) {
      if (this._stores.length > 0) {
        this._stores.forEach(function (entry) {
          entry.store.update(action, entry.change);
        });
      }
    }
  }
};
```

The actions

You probably noticed that we didn't talk about the actions. What are they? The convention is that they should be simple objects having two properties - `type` and `payload` :

```
{
  type: 'USER_LOGIN_REQUEST',
  payload: {
    username: '...',
    password: '...'
  }
}
```

The `type` says what exactly the action is and the `payload` contains the information associated with the event. And in some cases we may leave the `payload` empty.

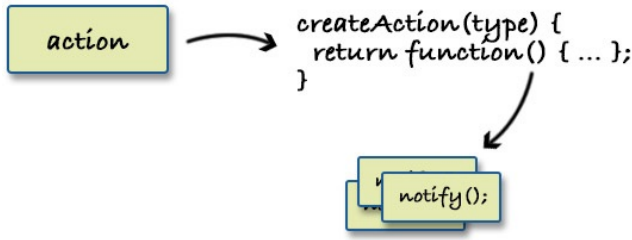
It's interesting that the `type` is well known in the beginning. We know what type of actions should be floating in our app, who is dispatching them and which of the stores are interested. Thus, we can apply partial application and avoid passing the action object here and there. For example:

```
var createAction = function (type) {
  if (!type) {
    throw new Error('Please, provide action\'s type.');
  } else {
    return function (payload) {
      return dispatcher.dispatch({
        type: type,
        payload: payload
      });
    }
  }
}
```

`createAction` leads to the following benefits:

- We no more need to remember the exact type of the action. We now have a function which we call passing only the payload.
- We no more need an access to the dispatcher which is a huge benefit. Otherwise, think about how we have to pass it to every

single place where we need to dispatch an action.
- In the end we don't have to deal with objects but with functions which is much nicer. The objects are *static* while the functions describe a *process*.



This approach for creating actions is actually really popular and functions like the one above are usually called *action creators*.

The final code

In the section above we successfully hide the dispatcher while submitting actions. We may do it again for the store's registration:

```
var createSubscriber = function (store) {
  return dispatcher.register(store);
}
```

And instead of exporting the dispatcher we may export only these two functions `createAction` and `createSubscriber`. Here is how the final code looks like:

```
var Dispatcher = function () {
  return {
    _stores: [],
    register: function (store) {
      if (!store || !store.update) {
        throw new Error(
          'You should provide a store that has an `update` method'
        );
      } else {
        var consumers = [];
        var change = function () {
          consumers.forEach(function (l) {
            l(store);
          });
        };
        var subscribe = function (consumer, noInit) {
          consumers.push(consumer);
          !noInit ? consumer(store) : null;
        };

        this._stores.push({ store: store, change: change });
        return subscribe;
      }
      return false;
    },
    dispatch: function (action) {
      if (this._stores.length > 0) {
        this._stores.forEach(function (entry) {
          entry.store.update(action, entry.change);
        });
      }
    }
  }
};

module.exports = {
  create: function () {
    var dispatcher = Dispatcher();

    return {
      createAction: function (type) {
        if (!type) {
          throw new Error('Please, provide action\'s type.');
        } else {
```

```
          return function (payload) {
            return dispatcher.dispatch({
              type: type,
              payload: payload
            });
          }
        }
      },
      createSubscriber: function (store) {
        return dispatcher.register(store);
      }
    }
  }
};
```

If we add the support of AMD, CommonJS and global usage we end up with 66 lines of code, 1.7KB plain or 795 bytes after minification JavaScript.

Wrapping up

We have a module that provides two helpers for building a Flux project. Let's write a simple counter app that doesn't involve React so we see the pattern in action.

The markup

We'll need some UI to interact with it so:

```
<div id="counter">
  <span></span>
  <button>increase</button>
  <button>decrease</button>
</div>
```

The `span` will be used for displaying the current value of our counter. The buttons will change that value.

The view

```
const View = function (subscribeToStore, increase, decrease) {
  var value = null;
  var el = document.querySelector('#counter');
  var display = el.querySelector('span');
  var [ increaseBtn, decreaseBtn ] =
    Array.from(el.querySelectorAll('button'));

  var render = () => display.innerHTML = value;
  var updateState = (store) => value = store.getValue();

  subscribeToStore([updateState, render]);

  increaseBtn.addEventListener('click', increase);
  decreaseBtn.addEventListener('click', decrease);
};
```

It accepts a store subscriber function and two action function for increasing and decreasing the value. The first few lines of the view are just fetching the DOM elements.

After that we define a `render` function which puts the value inside the `span` tag. `updateState` will be called every time when the store changes. So, we pass these two functions to `subscribeToStore` because we want to get the view updated and we want to get an initial rendering. Remember how our consumers are called at least once by default.

The last bit is calling the action functions when we press the buttons.

The store

Every action has a type. It's a good practice to create constants for these types so we don't deal with raw strings.

```
const INCREASE = 'INCREASE';
const DECREASE = 'DECREASE';
```

Very often we have only one instance of every store. For the sake of simplicity we'll create ours as a singleton.

```
const CounterStore = {
  _data: { value: 0 },
  getValue: function () {
    return this._data.value;
  },
  update: function (action, change) {
    if (action.type === INCREASE) {
      this._data.value += 1;
    } else if (action.type === DECREASE) {
      this._data.value -= 1;
    }
    change();
  }
};
```

`_data` is the internal state of the store. `update` is the well known method that our dispatcher calls. We process the action inside and say `change()` when we are done. `getValue` is a public method used by the view so it reaches the needed info. In our case this is just the value of the counter.

Wiring all the pieces

So, we have the store waiting for actions from the dispatcher. We have the view defined. Let's create the store subscriber, the actions and run everything.

```
const { createAction, createSubscriber } = Fluxiny.create();
const counterStoreSubscriber = createSubscriber(CounterStore);
const actions = {
  increase: createAction(INCREASE),
  decrease: createAction(DECREASE)
};

View(counterStoreSubscriber, actions.increase, actions.decrease);
```

And that's it. Our view is subscribed to the store and it renders by default because one of our consumers is actually the `render` method.

A live demo

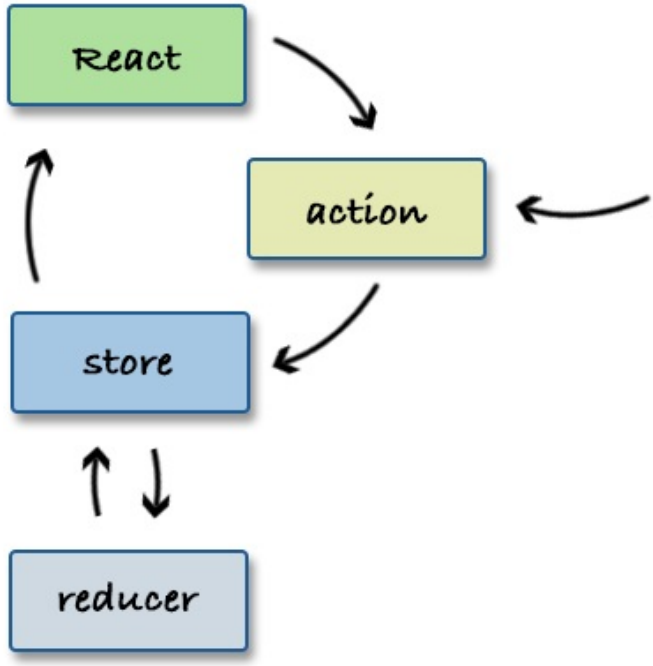
A live demo could be seen in the following JSBin http://jsbin.com/koxidu/embed?js,output. If that's not enough and it seems too simple for you please checkout the example in Fluxiny repository https://github.com/krasimir/fluxiny/tree/master/example. It uses React as a view layer.

The Flux implementation discussed in this section is available here github.com/krasimir/fluxiny. Feel free to use it in a browser directly or as a npm dependency.

Redux

Redux is a library that acts as a state container and helps managing your application data flow. It was introduced back in 2015 at ReactEurope conference (video) by Dan Abramov. It is similar to Flux architecture and has a lot in common with it. In this section we will create a small counter app using Redux alongside React.

Redux architecture and its main characteristics



Similarly to Flux architecture we have the view components (React) dispatching an action. Same action may be dispatched by another part of our system. Like a bootstrap logic for example. This action is dispatched not to a central hub but directly to the store. We are saying "store" not "stores" because there is only one in Redux. That is one of the big differences between Flux and Redux. The logic that decided how

our data changes lives in pure functions called reducers. Once the store receives an action it asks the reducers about the new version of the state by sending the current state and the given action. Then in immutable fashion the reducer needs to return the new state. The store continues from there and updates its internal state. As a final step, the wired to the store React component gets re-rendered.

The concept is pretty linear and again follows the one-direction data flow. Let's talk about all these pieces and introduce a couple of new terms that support the work of the Redux pattern.

Actions

The typical action in Redux (same as Flux) is just an object with a `type` property. Everything else in that object is considered a context specific data and it is not related to the pattern but to your application logic. For example:

```
const CHANGE_VISIBILITY = 'CHANGE_VISIBILITY';
const action = {
  type: CHANGE_VISIBILITY,
  visible: false
}
```

It is a good practice that we create constants like `CHANGE_VISIBILITY` for our action types. It happens that there are lots of tools/libraries that support Redux and solve different problems which do require the type of the action only. So it is just a convenient way to transfer this information.

The `visible` property is the meta data that we mentioned above. It has nothing to do with Redux. It means something in the context of the application.

Every time when we want to dispatch a method we have to use such objects. However, it becomes too noisy to write them over and over again. That is why there is the concept of *action creators*. An action creator is a function that returns an object and may or may not accept an argument which directly relates to the action properties. For example the action creator for the above action looks like this:

```
const changeVisibility = visible => ({
  type: CHANGE_VISIBILITY,
  visible
});

changeVisibility(false);
// { type: CHANGE_VISIBILITY, visible: false }
```

Notice that we pass the value of the `visible` as an argument and we don't have to remember (or import) the exact type of the action. Using such helpers makes the code compact and easy to read.

Store

Redux provides a helper `createStore` for creating a store. Its signature is as follows:

```
import { createStore } from 'redux';

createStore([reducer], [initial state], [enhancer]);
```

We already mentioned that the reducer is a function that accepts the current state and action and returns the new state. More about that in a bit. The second argument is the initial state of the store. This comes as a handy instrument to initialize our application with data that we already have. This feature is the essence of processes like server-side rendering or persistent experience. The third parameter, enhancer,

provides an API for extending Redux with third party middlewares and basically plug some functionally which is not baked-in. Like for example an instrument for handling async processes.

Once created the store has four methods - `getState`, `dispatch`, `subscribe` and `replaceReducer`. Probably the most important one is `dispatch`:

```
store.dispatch(changeVisibility(false));
```

That is the place where we use our action creators. We pass the result of them or in other words action objects to this `dispatch` method. It then gets spread to the reducers in our application.

In the typical React application we usually don't use `getState` and `subscribe` directly because there is a helper (we will see it in the next sections) that wires our components with the store and effectively `subscribe`s for changes. As part of this subscription we also receive the current state so we don't have to call `getState` ourself. `replaceReducer` is kind of an advanced API and it swaps the reducer currently used by the store. I personally never used this method.

Reducer

The reducer function is probably the most *beautiful* part of Redux. Even before that I was interested in writing pure functions with an immutability in mind but Redux forced me to do it. There are two characteristics of the reducer that are quite important and without them we basically have a broken pattern.

(1) It must be a pure function - it means that the function should return the exact same output evert time when the same input is given.

(2) It should have no side effects - stuff like accessing a global variable, making an async call or waiting for a promise to resolve have no place in here.

Here is a simple counter reducer:

```
const counterReducer = function (state, action) {
  if (action.type === ADD) {
    return { value: state.value + 1 };
  } else if (action.type === SUBTRACT) {
    return { value: state.value - 1 };
  }
  return { value: 0 };
};
```

There are no side effects and we return a brand new object every time. We accumulate the new value based on the previous state and the incoming action type.

Wiring to React components

If we talk about Redux in the context of React we almost always mean react-redux module. It provides two things that help connecting Redux to our components.

(1) `<Provider>` component - it's a component that accepts our store and makes it available for the children down the React tree via the React's context API. For example:

```
<Provider store={ myStore }>
  <MyApp />
</Provider>
```

We usually have a single place in our app where we use it.

(2) `connect` function - it is a function that does the subscribing for updates in the store and re-renders our component. It implements a higher-order component. Here is its signature:

```
connect(
  [mapStateToProps],
  [mapDispatchToProps],
  [mergeProps],
  [options]
)
```

`mapStateToProps` parameter is a function that accepts the current state and must return a set of key-value pairs (an object) that are getting send as props to our React component. For example:

```
const mapStateToProps = state => ({
  visible: state.visible
});
```

`mapDispatchToProps` is a similar one but instead of the `state` receives a `dispatch` function. Here is the place where we can define a prop for dispatching actions.

```
const mapDispatchToProps = dispatch => ({
  changeVisibility: value => dispatch(changeVisibility(value))
});
```

`mergeProps` combines both `mapStateToProps` and `mapDispatchToProps` and the props send to the component and gives us the opportunity to accumulate better props. Like for example if we need to fire two actions we may combine them to a single prop and send that to React. `options` accepts couple of settings that control how how the connection works.

Simple counter app using Redux

Let's create a simple counter app that uses all the APIs above.

The "Add" and "Subtract" buttons will simply change a value in our store. "Visible" and "Hidden" will control its visibility.

Modeling the actions

For me, every Redux feature starts with modeling the action types and defining what state we want to keep. In our case we have three operations going on - adding, subtracting and managing visibility. So we will go with the following:

```
const ADD = 'ADD';
const SUBTRACT = 'SUBTRACT';
const CHANGE_VISIBILITY = 'CHANGE_VISIBILITY';

const add = () => ({ type: ADD });
const subtract = () => ({ type: SUBTRACT });
const changeVisibility = visible => ({
  type: CHANGE_VISIBILITY,
  visible
});
```

Store and its reducers

There is something that we didn't talk about while explaining the store and reducers. We usually have more then one reducer because we want to manage multiple things. The store is one though and we in theory have only one state object. What happens in most of the apps running in production is that the application state is a composition of slices. Every slice represents a part of our system. In this very small example we have counting and visibility slices. So our initial state looks like that:

```
const initialState = {
  counter: {
    value: 0
  },
  visible: true
};
```

We must define separate reducers for both parts. This is to introduce some flexibility and to improve the readability of our code. Imagine if we have a giant app with ten or more state slices and we keep working within a single function. It will be too difficult to manage.

Redux comes with a helper that allows us to target a specific part of the state and assign a reducer to it. It is called `combineReducers`:

```
import { createStore, combineReducers } from 'redux';

const rootReducer = combineReducers({
  counter: function A() { ... },
  visible: function B() { ... }
});
const store = createStore(rootReducer);
```

Function `A` receives only the `counter` slice as a state and needs to return only that part. Same for `B`. Accepts a boolean (the value of `visible`) and must return a boolean.

The reducer for our counter slice should take into account both actions `ADD` and `SUBTRACT` and based on them calculates the new `counter` state.

```
const counterReducer = function (state, action) {
  if (action.type === ADD) {
    return { value: state.value + 1 };
  } else if (action.type === SUBTRACT) {
    return { value: state.value - 1 };
  }
  return state || { value: 0 };
};
```

Every reducer is fired at least once when the store is initialized. In that very first run the `state` is `undefined` and the `action` is `{ type: "@@redux/INIT"}`. In this case our reducer should return the initial value of our data - `{ value: 0 }`.

The reducer for the visibility is pretty similar except that it waits for `CHANGE_VISIBILITY` action:

```
const visibilityReducer = function (state, action) {
  if (action.type === CHANGE_VISIBILITY) {
    return action.visible;
  }
  return true;
};
```

And at the end we have to pass both reducers to `combineReducers` so we create our `rootReducer`.

```
const rootReducer = combineReducers({
  counter: counterReducer,
  visible: visibilityReducer
});
```

Selectors

Before moving to the React components we have to mention the concept of a *selector*. From the previous section we know that our state is usually divided into different parts. We have dedicated reducers to update the data but when it comes to fetching it we still have a single object. Here is the place where the selectors come in handy. The selector is a function that accepts the whole state object and extracts only the information that we need. For example in our small app we need two of those:

```
const getCounterValue = state => state.counter.value;
const getVisibility = state => state.visible;
```

A counter app is too small to see the real power of writing such helpers. However, in a big project is quite different. And it is not just about saving a few lines of code. Neither is about readability. Selectors come with these stuff but they are also contextual and may contain logic. Since they have access to the whole state they are able to answer business logic related questions. Like for example "Is the user authorize to do X while being on page Y". This may be done in a single selector.

React components

Let's first deal with the UI that manages the visibility of the counter.

```
function Visibility({ changeVisibility }) {
  return (
    <div>
      <button onClick={ () => changeVisibility(true) }>
        Visible
      </button>
      <button onClick={ () => changeVisibility(false) }>
        Hidden
      </button>
    </div>
  );
}

const VisibilityConnected = connect(
  null,
  dispatch => ({
    changeVisibility: value => dispatch(changeVisibility(value))
  })
)(Visibility);
```

We have two buttons `Visible` and `Hidden`. They both fire `CHANGE_VISIBILITY` action but the first one passes `true` as a value while the second one `false`. The `VisibilityConnected` component class gets created as a result of the wiring done via Redux's `connect`. Notice that we pass `null` as `mapStateToProps` because we don't need any of the data in the store. We just need to `dispatch` an action.

The second component is slightly more complicated. It is named `Counter` and renders two buttons and the counter value.

```
function Counter({ value, add, subtract }) {
  return (
    <div>
      <p>Value: { value }</p>
      <button onClick={ add }>Add</button>
      <button onClick={ subtract }>Subtract</button>
    </div>
  );
}

const CounterConnected = connect(
  state => ({
    value: getCounterValue(state)
  }),
  dispatch => ({
    add: () => dispatch(add()),
    subtract: () => dispatch(subtract())
  })
)(Counter);
```

We now need both `mapStateToProps` and `mapDispatchToProps` because we want to read data from the store and dispatch actions. Our component receives three props - `value`, `add` and `subtract`.

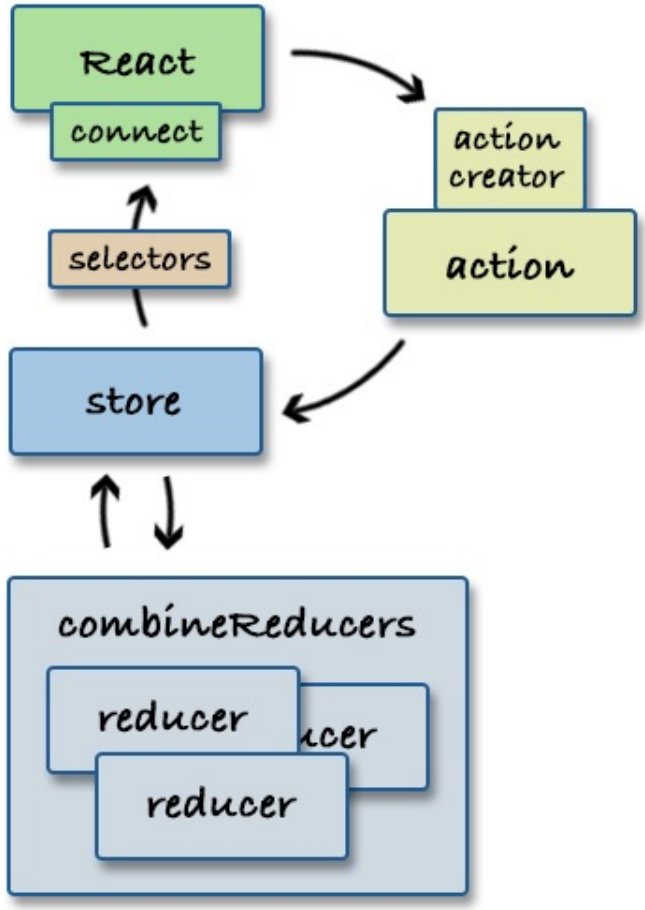
The very last bit is an `App` component where we compose the application.

```
function App({ visible }) {
  return (
    <div>
      <VisibilityConnected />
      { visible && <CounterConnected /> }
    </div>
  );
}
const AppConnected = connect(
  state => ({
    visible: getVisibility(state)
  })
)(App);
```

We again need to `connect` our component because we want to control the visibility of the counter. The `getVisibility` selector returns a boolean that indicates weather `CounterConnected` will be rendered or not.

Final thoughts

Redux is a wonderful pattern. Over the years the JavaScript community developed the idea and enhanced it with couple of new terms. I think a typical redux application looks more like this:



By the way we didn't mention the side effects management. It is a whole new story with its own ideas and solutions.

We can conclude that Redux itself is a pretty simple pattern. It teaches very useful techniques but unfortunately it is very often not enough. Sooner or later we have to introduce more concepts/patterns. Which of course is not that bad. We just have to plan for it.

Dependency injection

Big part of the modules/components that we write have dependencies. A proper management of these dependencies is critical for the success of the project. There is a technique (most people consider it a *pattern*) called *dependency injection* that helps solving the problem.

In React the need of dependency injector is easily visible. Let's consider the following application tree:

```
// Title.jsx
export default function Title(props) {
  return <h1>{ props.title }</h1>;
}

// Header.jsx
import Title from './Title.jsx';

export default function Header() {
  return (
    <header>
      <Title />
    </header>
  );
}

// App.jsx
import Header from './Header.jsx';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { title: 'React in patterns' };
  }
  render() {
    return <Header />;
  }
};
```

The string "React in patterns" should somehow reach the `Title` component. The direct way of doing this is to pass it from `App` to `Header` and then `Header` pass it down to `Title`. However, this may work for these three components but what happens if there are multiple properties and deeper nesting. Lots of components will act as proxy passing properties to their children.

We already saw how the higher-order component may be used to inject data. Let's use the same technique to inject the `title` variable:

```
// inject.jsx
const title = 'React in patterns';

export default function inject(Component) {
  return class Injector extends React.Component {
    render() {
      return (
        <Component
          {...this.props}
          title={ title }
        />
      )
    }
  };
}

// ----------------------------------
// Header.jsx
import inject from './inject.jsx';
import Title from './Title.jsx';

var EnhancedTitle = inject(Title);
export default function Header() {
  return (
    <header>
      <EnhancedTitle />
    </header>
  );
}
```

The `title` is hidden in a middle layer (higher-order component) where we pass it as a prop to the original `Title` component. That's all nice but it solves only half of the problem. Now we don't have to pass the `title` down the tree but how this data reaches the `inject.jsx` helper.

Using React's context (prior v. 16.3)

React has the concept of *context*. The *context* is something that every React component has access to. It's something like an event bus but for data. A single *store* which we access from everywhere.

```
// a place where we will define the context
var context = { title: 'React in patterns' };

class App extends React.Component {
  getChildContext() {
    return context;
  }
  ...
};
App.childContextTypes = {
  title: React.PropTypes.string
};

// a place where we use the context
class Inject extends React.Component {
  render() {
    var title = this.context.title;
    ...
  }
}
Inject.contextTypes = {
  title: React.PropTypes.string
};
```

Notice that we have to specify the exact signature of the context object. With `childContextTypes` and `contextTypes`. If those are not specified then the `context` object will be empty. That may be a little bit frustrating

because we may have lots of stuff to put there. That is why it is a good practice that our `context` is not just a plain object but it has an interface that allows us to store and retrieve data. For example:

```
// dependencies.js
export default {
  data: {},
  get(key) {
    return this.data[key];
  },
  register(key, value) {
    this.data[key] = value;
  }
}
```

Then, if we go back to our example, the `App` component may look like that:

```
import dependencies from './dependencies';

dependencies.register('title', 'React in patterns');

class App extends React.Component {
  getChildContext() {
    return dependencies;
  }
  render() {
    return <Header />;
  }
};
App.childContextTypes = {
  data: React.PropTypes.object,
  get: React.PropTypes.func,
  register: React.PropTypes.func
};
```

And our `Title` component gets it's data through the context:

```
// Title.jsx
```

```
export default class Title extends React.Component {
  render() {
    return <h1>{ this.context.get('title') }</h1>
  }
}
Title.contextTypes = {
  data: React.PropTypes.object,
  get: React.PropTypes.func,
  register: React.PropTypes.func
};
```

Ideally we don't want to specify the `contextTypes` every time when we need an access to the context. This detail may be wrapped again in a higher-order component. And even better, we may write an utility function that is more descriptive and helps us declare the exact wiring. I.e instead of accessing the context directly with `this.context.get('title')` we ask the higher-order component to get what we need and pass it as props to our component. For example:

```
// Title.jsx
import wire from './wire';

function Title(props) {
  return <h1>{ props.title }</h1>;
}

export default wire(Title, ['title'], function resolve(title) {
  return { title };
});
```

The `wire` function accepts a React component, then an array with all the needed dependencies (which are `register` ed already) and then a function which I like to call `mapper` . It receives what is stored in the context as a raw data and returns an object which is later used as props for our component (`Title`). In this example we just pass what we get - a `title` string variable. However, in a real app this could be a collection of data stores, configuration or something else.

Here is how the `wire` function looks like:

```
export default function wire(Component, dependencies, mapper) {
  class Inject extends React.Component {
    render() {
      var resolved = dependencies.map(
        this.context.get.bind(this.context)
      );
      var props = mapper(...resolved);

      return React.createElement(Component, props);
    }
  }
  Inject.contextTypes = {
    data: React.PropTypes.object,
    get: React.PropTypes.func,
    register: React.PropTypes.func
  };
  return Inject;
};
```

`Inject` is a higher-order component that gets access to the context and retrieves all the items listed under `dependencies` array. The `mapper` is a function receiving the `context` data and transforms it to props for our component.

Using React's context (v. 16.3 and above)

For years the context API was not really recommended by Facebook. They mentioned in the official docs that the API is not stable and may change. And that is exactly what happened. In the version 16.3 we got a new one which I think is more natural and easy to work with.

Let's use the same example with the string that needs to reach a `<Title>` component.

We will start by defining a file that will contain our context initialization:

```
// context.js
import { createContext } from 'react';

const Context = createContext({});

export const Provider = Context.Provider;
export const Consumer = Context.Consumer;
```

`createContext` returns an object that has `.Provider` and `.Consumer` properties. Those are actually valid React classes. The `Provider` accepts our context in the form of a `value` prop. The consumer is used to access the context and basically read data from it. And because they usually live in different files it is a good idea to create a single place for their initialization.

Let's say that our `App` component is the root of our tree. At that place we have to pass the context.

```
import { Provider } from './context';

const context = { title: 'React In Patterns' };

class App extends React.Component {
  render() {
    return (
      <Provider value={ context }>
        <Header />
      </Provider>
    );
  }
};
```

The wrapped components and their children now share the same context. The `<Title>` component is the one that needs the `title` string so that is the place where we use the `<Consumer>`.

```
import { Consumer } from './context';

function Title() {
  return (
    <Consumer>{
      ({ title }) => <h1>Title: { title }</h1>
    }</Consumer>
  );
}
```

Notice that the `Consumer` class uses the function as children (render prop) pattern to deliver the context.

The new API feels easier to understand and eliminates the boilerplate. It is still pretty new but looks promising. It opens a whole new range of possibilities.

Using the module system

If we don't want to use the context there are a couple of other ways to achieve the injection. They are not exactly React specific but worth mentioning. One of them is using the module system.

As we know the typical module system in JavaScript has a caching mechanism. It's nicely noted in the Node's documentation:

> Modules are cached after the first time they are loaded. This means (among other things) that every call to require('foo') will get exactly the same object returned, if it would resolve to the same file.
>
> Multiple calls to require('foo') may not cause the module code to be executed multiple times. This is an important feature. With it, "partially done" objects can be returned, thus allowing transitive dependencies to be loaded even when they would cause cycles.

How is that helping for our injection? Well, if we export an object we are actually exporting a singleton and every other module that imports the file will get the same object. This allows us to `register` our dependencies and later `fetch` them in another file.

Let's create a new file called `di.jsx` with the following content:

```
var dependencies = {};

export function register(key, dependency) {
  dependencies[key] = dependency;
}

export function fetch(key) {
  if (dependencies[key]) return dependencies[key];
  throw new Error(`"${ key } is not registered as dependency.`);
}

export function wire(Component, deps, mapper) {
  return class Injector extends React.Component {
    constructor(props) {
      super(props);
      this._resolvedDependencies = mapper(...deps.map(fetch));
    }
    render() {
      return (
        <Component
          {...this.state}
          {...this.props}
          {...this._resolvedDependencies}
        />
      );
    }
  };
}
```

We'll store the dependencies in `dependencies` global variable (it's global for our module, not for the whole application). We then export two functions `register` and `fetch` that write and read entries. It looks a little bit like implementing setter and getter against a simple JavaScript

object. Then we have the `wire` function that accepts our React component and returns a higher-order component. In the constructor of that component we are resolving the dependencies and later while rendering the original component we pass them as props. We follow the same pattern where we describe what we need (`deps` argument) and extract the needed props with a `mapper` function.

Having the `di.jsx` helper we are again able to register our dependencies at the entry point of our application (`app.jsx`) and inject them wherever (`Title.jsx`) we need.

```
// app.jsx
import Header from './Header.jsx';
import { register } from './di.jsx';

register('my-awesome-title', 'React in patterns');

class App extends React.Component {
  render() {
    return <Header />;
  }
};

// -----------------------------------
// Header.jsx
import Title from './Title.jsx';

export default function Header() {
  return (
    <header>
      <Title />
    </header>
  );
}

// -----------------------------------
// Title.jsx
import { wire } from './di.jsx';

var Title = function(props) {
  return <h1>{ props.title }</h1>;
```

```
};

export default wire(
  Title,
  ['my-awesome-title'],
  title => ({ title })
);
```

If we look at the `Title.jsx` file we'll see that the actual component and the wiring may live in different files. That way the component and the mapper function become easily unit testable.

Final thoughts

Dependency injection is a tough problem. Especially in JavaScript. Lots of people didn't realize that but putting a proper dependency management is a key process of every development cycle. JavaScript ecosystem offers different tools and we as developers should pick the one that fits in our needs.

Styling React components

React is a view layer. As such it kind of controls the markup rendered in the browser. And we know that the styling with CSS is tightly connected to the markup on the page. There are couple of approaches for styling React applications and in this section we will go through the most popular ones.

The good old CSS class

JSX syntax is pretty close to HTML syntax. As such we have almost the same tag attributes and we may still style using CSS classes. Classes which are defined in an external `.css` file. The only caveat is using `className` and not `class`. For example:

```
<h1 className='title'>Styling</h1>
```

Inline styling

The inline styling works just fine. Similarly to HTML we are free to pass styles directly via a `style` attribute. However, while in HTML the value is a string in JSX must be an object.

```
const inlineStyles = {
  color: 'red',
  fontSize: '10px',
  marginTop: '2em',
  'border-top': 'solid 1px #000'
};

<h2 style={ inlineStyles }>Inline styling</h2>
```

Because we write the styles in JavaScript we have some limitations from a syntax point of view. If we want to keep the original CSS property names we have to put them in quotes. If not then we have to follow the camel case convention. However, writing styles in JavaScript is quite interesting and may be a lot more flexible then the plain CSS. Like for example inheriting of styles:

```
const theme = {
  fontFamily: 'Georgia',
  color: 'blue'
};
const paragraphText = {
  ...theme,
  fontSize: '20px'
};
```

We have some basic styles in `theme` and with mix them with what is in `paragraphText`. Shortly, we are able to use the whole power of JavaScript to organize our CSS. What it matters at the end is that we generate an object that goes to the `style` attribute.

CSS modules

CSS modules is building on top of what we said so far. If we don't like the JavaScript syntax then we may use CSS modules and we will be able to write plain CSS. Usually this library plays its role at bundling time. It is possible to hook it as part of the transpilation step but normally is distributed as a build system plugin.

Here is a quick example to get an idea how it works:

```
/* style.css */
.title {
  color: green;
}

// App.jsx
import styles from "./style.css";

function App() {
  return <h1 style={ styles.title }>Hello world</h1>;
}
```

That is not possible by default but with CSS modules we may import directly a plain CSS file and use the classes inside.

And when we say *plain CSS* we don't mean that it is exactly like the normal CSS. It supports some really helpful composition techniques. For example:

```
.title {
  composes: mainColor from "./brand-colors.css";
}
```

Styled-components

Styled-components took another direction. Instead of inlining styles the library provides a React component. We then use this component to represent a specific look and feel. For example, we may create a `Link` component that has certain styling and use that instead of the `<a>` tag.

```
const Link = styled.a`
  text-decoration: none;
  padding: 4px;
  border: solid 1px #999;
  color: black;
`;
```

```
<Link href='http://google.com'>Google</Link>
```

There is again a mechanism for extending classes. We may still use the `Link` component but change the text color like so:

```
const AnotherLink = styled(Link)`
  color: blue;
`;

<AnotherLink href='http://facebook.com'>Facebook</AnotherLink>
```

By far for me styled-components are probably the most interesting approach for styling in React. It is quite easy to create components for everything and forget about the styling. If your company has the capacity to create a design system and building a product with it then this option is probably the most suitable one.

Final thoughts

There are multiple ways to style your React application. I did experienced all of them in production and I would say that there is no right or wrong. As most of the stuff in JavaScript today you have to pick the one that fits better in your context.

Third-party integration

React is probably one of the best choices for building UI. Good design, support and community. However, there are cases where we want to use an external service or we want to integrate something completely different. We all know that React works heavily with the actual DOM and basically controls what's rendered on the screen. That's why integrating of third-party components may be tricky. In this section we will see how to mix React and jQuery's UI plugin and do it safely.

The example

I picked *tag-it* jQuery plugin for my example. It transforms an unordered list to input field for managing tags:

```
<ul>
  <li>JavaScript</li>
  <li>CSS</li>
</ul>
```

to:

To make it work we have to include jQuery, jQuery UI and the *tag-it* plugin code. It works like that:

```
$('<dom element selector>').tagit();
```

We select a DOM element and call `tagit()`.

Now, let's create a simple React app that will use the plugin:

```
// Tags.jsx
class Tags extends React.Component {
  render() {
    return (
      <ul>
      {
        this.props.tags.map(
          (tag, i) => <li key={ i }>{ tag } </li>
        )
      }
      </ul>
    );
  }
};

// App.jsx
class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = { tags: ['JavaScript', 'CSS' ] };
  }
  render() {
    return (
      <div>
        <Tags tags={ this.state.tags } />
      </div>
    );
  }
}

ReactDOM.render(<App />, document.querySelector('#container'));
```

The entry point is our `App` class. It uses the `Tags` component that displays an unordered list based on the passed `tags` prop. When React renders the list on the screen we know that we have a `<ul>` tag so we can hook it up to the jQuery plugin.

Force a single-render

The very first thing that we have to do is to force a single-render of the `Tags` component. That is because when React adds the elements in the actual DOM we want to pass the control of them to jQuery. If we skip this both React and jQuery will work on same DOM elements without knowing for each other. To achieve a single-render we have to use the lifecycle method `shouldComponentUpdate` like so:

```
class Tags extends React.Component {
  shouldComponentUpdate() {
    return false;
  }
  ...
```

By always returning `false` here we are saying that our component will never re-render. If defined `shouldComponentUpdate` is used by React to understand whether to trigger `render` or not. That is ideal for our case because we want to place the markup on the page using React but we don't want to rely on it after that.

Initializing the plugin

React gives us an API for accessing actual DOM nodes. We have to use the `ref` attribute on a node and later reach that node via `this.refs`. `componentDidMount` is the proper lifecycle method for initializing the *tag-it* plugin. That's because we get it called when React mounts the result of the `render` method.

```
class Tags extends React.Component {
  ...
  componentDidMount() {
    this.list = $(this.refs.list);
    this.list.tagit();
  }
  render() {
    return (
      <ul ref='list'>
      {
        this.props.tags.map(
          (tag, i) => <li key={ i }>{ tag } </li>
        )
      }
      </ul>
    );
  }
  ...
```

The code above together with `shouldComponentUpdate` leads to React rendering the `<ul>` with two items and then *tag-it* transforms it to a working tag editing widget.

Controlling the plugin using React

Let's say that we want to programmatically add a new tag to the already running *tag-it* field. Such action will be triggered by the React component and needs to use the jQuery API. We have to find a way to communicate data to `Tags` component but still keep the single-render approach.

To illustrate the whole process we will add an input field to the `App` class and a button which if clicked will pass a string to `Tags` component.

```
class App extends React.Component {
  constructor(props) {
    super(props);

    this._addNewTag = this._addNewTag.bind(this);
    this.state = {
      tags: ['JavaScript', 'CSS' ],
      newTag: null
    };
  }
  _addNewTag() {
    this.setState({ newTag: this.refs.field.value });
  }
  render() {
    return (
      <div>
        <p>Add new tag:</p>
        <div>
          <input type='text' ref='field' />
          <button onClick={ this._addNewTag }>Add</button>
        </div>
        <Tags
          tags={ this.state.tags }
          newTag={ this.state.newTag } />
      </div>
    );
  }
}
```

We use the internal state as a data storage for the value of the newly added field. Every time when we click the button we update the state and trigger re-rendering of `Tags` component. However, because of `shouldComponentUpdate` we have no any updates on the screen. The only one change is that we get a new value of the `newTag` prop which may be captured via another lifecycle method - `componentWillReceiveProps`:

```
class Tags extends React.Component {
  ...
  componentWillReceiveProps(newProps) {
    this.list.tagit('createTag', newProps.newTag);
  }
  ...
```

`.tagit('createTag', newProps.newTag)` is a pure jQuery code. `componentWillReceiveProps` is a nice place for calling methods of the third-party library.

Here is the full code of the `Tags` component:

```
class Tags extends React.Component {
  componentDidMount() {
    this.list = $(this.refs.list);
    this.list.tagit();
  }
  shouldComponentUpdate() {
    return false;
  }
  componentWillReceiveProps(newProps) {
    this.list.tagit('createTag', newProps.newTag);
  }
  render() {
    return (
      <ul ref='list'>
      {
        this.props.tags.map(
          (tag, i) => <li key={ i }>{ tag } </li>
        )
      }
      </ul>
    );
  }
};
```

Final thoughts

Even though React is manipulating the DOM tree we are able to integrate third-party libraries and services. The available lifecycle methods give us enough control on the rendering process so they are the perfect bridge between React and non-React code.