by

RAILS

meets

REACT.JS

by **Marcin Grzywaczewski**
and **Robert Pankowecki**

Rails Meets React JSX

Arkency

© 2016 Arkency

Contents

Anatomy of the React component **1**
- State 1
- Properties 5
- Component children 14
- `render` method 17
- Mixins 20
- Context 21
- References (refs) to components 24
- Desugaring JSX 27

Anatomy of the React component

Let's see a really simple example of a React component:

```
class Greeter extends React.Component {
  render() {
    return (
      <span className="greeter-message">
        {`Hello, ${this.state.name}!`}
      </span>
    );
  }
}
```

To master React, you need to know the component's structure. Basically, components in React are made of three basic parts:

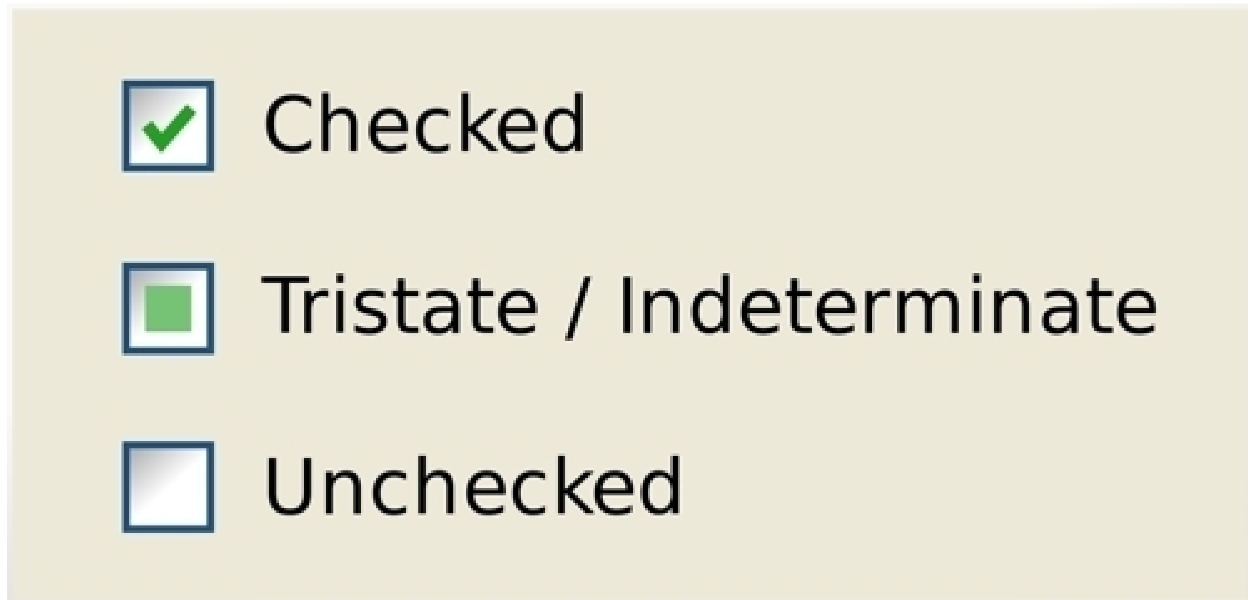
- **state** - React components maintain their dynamic part of data separated from the static data for performance reasons. Manipulating components is done by changing the state — doing so forces React to re-render a component.
- **properties** - React components get properties in the moment they are instantiated. You can use them to create the initial state and **set values that need to be accessible and are constant through the whole lifecycle of a component**.
- **`render` method** - The render method should always return a React component. It has access to properties and state and they (and only they) decide what will be the outcome of the render method. The render method should only generate HTML and assign key/refs/event handlers to it. If you generate it using only state and properties, you achieve what React developers call "side-effect free render method". You should aim to do so every time — it makes React faster, makes your development and testing easier.

That's it. These three parts need to be present in all React components you create. Of course it is not all - React components come with some additional features that will be discussed later.

State

In the world of dynamic user interfaces the "user interface is stateless" statement (which *template libraries* like Handlebars or Mustache repeat like a mantra) is hard to defend — the more complex

UI, the harder it may be to defend it. React developers know it — that's why they made component's state one of the most important concepts in the whole library.



Even the simplest input mechanism in browsers has state.

Changing state should be the only way to re-render your component, resulting in a visible change — state is available during rendering of the component and in lifecycle methods of it (discussed later). You may think of it as 'dynamic data' of a component — with an additional side-effect: **Changing state forces re-rendering of component**.

By default, when instantiating a component, the state is empty — it is a `null` value. You can change it using the `setState` method which every component has. **Note that you cannot manipulate the state of unmounted components**. All components start as unmounted — you must explicitly call the `ReactDOM.render` method to make it **mounted** which allows you to use `setState` and other component's instance methods.

State is accessible in most component methods under `this.state` instance variable. In the `Greeter` component you directly access the `name` field of the `this.state` inside `render` method.

Common mistakes

Consider this small snippet of code.

```
class Greeter extends React.Component {
  render() {
    return (
      <span className="greeter-message">
        {`Hello, ${this.state.name}!`}
      </span>
    );
  }
}

$(() => {
  const greeter = <Greeter />;
  greeter.setState({ name: 'Johny' }); // ?
});
```

In this case the `setState` invocation results in an error since elements do not have state. Only the rendered components have state. Calling `setState` in such situation will result in an error.

Here is another, more tricky example.

```
class Greeter extends React.Component {
  render() {
    return (
      <span className="greeter-message">
        {`Hello, ${this.state.name}!`}
      </span>
    );
  }
}

$(() => {
  const component = ReactDOM.render(
    <Greeter />,
    document.getElementById('greeter-placeholder')
  );
  component.setState({ name: 'Johny' }); // ?
});
```

`ReactDOM.render` will throw the `null is not an object` exception. Why is that? Because `ReactDOM.render` calls the `render` method from the component that is being mounted. Remember that `this.state` by default is `null`. So calling `this.state.name` inside the `render` method raises an error in such situation.

If `render` finished successfully it'd be absolutely fine to ask for the state change. To fix this code, you need to know how to provide the initial state to our components. Let's see how to do this.

Setting initial state

Since state in components is `null` by default, it'd be hard to make something meaningful with React if there was no way to have control the initial state. In fact, without such control you wouldn't be able to create any dynamic components at all. Since there will be only properties available and properties are static, they shouldn't change. You can achieve it by setting initial state in components constructor, this way it is used before any `setState` happens.

Setting initial state in a constructor allows you to use Greeter component

```
class Greeter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: 'World'
    };
  }

  render() {
    return (
      <span className="greeter-message">
        {`Hello, ${this.state.name}!`}
      </span>
    );
  }
}
```

After this change to the `Greeter` component our last example is going to work as expected:

```
const greeter = <Greeter />;

const component = ReactDOM.render(
  greeter,
  document.getElementById('greeter-placeholder')
);
// <span class='greeter-message'>Hello, World!</span> will be
// placed inside #greeter-placeholder element, overriding previous content.

component.setState({ name: 'My first component' });
// Component is re-rendered, so it results in "Hello, my first React component!"
// as opposed to "Hello, World!" you had before.
```

The important part here is that you can use properties to instantiate the initial state. That means you can easily make defaults overridden.

Maintaining good state

What data should be stored inside the state? There are a few rules of thumb:

1. Data which we'd like to change in order to force re-render the component. That's the idea behind state after all!
2. You should rather store 'raw data' which allows you to compute all needed further data representations in the `render` method. State is not a place to duplicate data since it's our main source of truth you want to keep it as consistent as possible. The same applies to properties. If you are thinking about which of the two things you should put into the state — A or B and B can be computed from A, you should definitely store A in the state and compute B in `render` method.
3. **No components in state**. Components should be created in the `render` method based on the current state and properties provided. Storing components in state is discouraged and considered as an anti-pattern.
4. **Things that you need to dispose when component dies**. Especially important in integration with external libraries.

If your component is keeping reference to 3rd party library object (for example carousel, popup, graph, autocomplete) you need to call `remove()` (or some kind of equivalent of it) on it when the React component is unmounted (removed from a page). You can define your *clean-up* algorithm, overriding the default `componentWillUnmount` method. This method will be called just before unmounting your component.

Properties

You have seen that you can go quite deep even without knowing what properties are. But properties are a very helpful part of all React components you are going to create. The basic idea is that they are data which have to be unchanged during the whole lifecycle of the component — whether to instantiate a default state or to reference this data while defining component's behaviour and the render outcome. You may think about them as 'constants' in your component — you may reference them during rendering or setting component's behaviour, but you shouldn't change them.

React developers often say that properties are **immutable**. That means that after construction of the component they stay the unchanged. Forever. If it's a complex object you can of course invoke methods on it — but it is only because it's hard to ensure for React that something changed. React assumes that changes to properties never happen. Changing properties, even by invoking methods on it is considered an anti-pattern.

You pass properties during construction of an element. With JSX it's done this way:

```
<Component properties>children</Component>
```

Properties are accessible even in elements via the `this.props` field. As I mentioned before, you can also use them in constructor to parametrize default state creation:

Example: Passing a parameter to instantiate default state

```
class Greeter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: this.props.initialName || 'World'
    };
  }

  render() {
    return (
      <span className="greeter-message">
        {`Hello, ${this.state.name}!`}
      </span>
    );
  }
}

$(() => {
  let component;
  const rootNode = document.getElementById('greeter-box');

  component = ReactDOM.render(<Greeter />, rootNode);
  // <span class='greeter-message'>Hello, World!</span>

  component = ReactDOM.render(<Greeter initialName="Andy" />, rootNode);
  // <span class='greeter-message'>Hello, Andy!</span>

  component.setState({ name: 'Marcin' });
  // <span class='greeter-message'>Hello, Marcin!</span>
});
```

This use case is a quite common usage of properties. Another common usage is to pass dependencies to components:

Example: Metrics tracking tool needs to be notified about user decision of reading full content of blogpost.

```
class BlogPost extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      fullContentVisible: false
    };
  }

  continueReadingClicked() {
    if (!this.state.fullContentVisible) {
      this.props.metricsAdapter.track('full-content-hit');
    }
    this.setState({ fullContentVisible: true });
  }

  render() {
    return (
      <div className="blog-content">
        <h1 className="title" key="blog-title">
          {this.props.blogpost.title}
        </h1>
        <p className="lead" key="blog-lead">
          {this.props.blogpost.lead}
        </p>
        <a href="#!/more" key="blog-more" onClick={this.continueReadingClicked}>
          Continue reading ->
        </a>
        {this.state.fullContentVisible ?
          <div key="blog-full-content">
            {this.props.blogpost.fullContent}
          </div> : null
        }
      </div>
    );
  }
}


const post = <BlogPost metricsAdapter={this.googleAnalyticsAdapter} />;
const componentInstance = ReactDOM.render(post, document.getElementById('blog'));
```

Properties allow you to store all references that are important for our component, but do not change over time.

You can use properties alone to create React components for your static views. Such components are called **stateless components** — and should be used as often as possible.

Stateless components

In our previous examples you were using components from the React library which are provided to create basic HTML tags you know and use in static pages. If you look at them closely you may notice that they are **stateless**. You pass only properties to elements of their type and they maintain no state. State is an inherent part of more dynamic parts of your code. However it is advisable to keep *'display data only'* components stateless. It makes code more understandable and testable and it's generally a good practice.

Displaying person information does not need state at all

```
1  class PersonInformation extends React.Component {
2    person() {
3      return this.props.person;
4    }
5
6    render() {
7      return (
8        <div className="person-info">
9          <header key="info-header">
10           <img
11             alt=""
12             className="person-avatar"
13             key="avatar"
14             src={this.person().avatarUrl}
15           />
16           <h2 className="person-name" key="full-name">
17             {this.person().fullName}
18           </h2>
19         </header>
20       </div>
21     );
22   }
23 }
```

The rule of thumb here - when possible, create stateless components.

Stateless functional components in React 0.14

React 0.14 now supports stateless components as simple functions[1]. Which is especially useful with ES2015 (ES6) and JSX syntax.

```
// A functional component using an ES2015 (ES6) arrow function:
var Header = (props) => {
  return <h2 className="header">{props.text}</h2>;
};

// Or with destructuring and an implicit return, simply:
var Header = ({text}) => (
  <h2 className="header">
    {text}
  </h2>
);
```

Setting default properties

It is often helpful to create some kind of default state — while it is not necessary, it may reduce conditionals in your code, making it easier to read and maintain. Default properties can be set with `defaultProps` as in next example.

Unnecessary or statement can be avoided here with setting default state.

```
class Greeter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: this.props.initialName || 'World'
    };
  }

  render() {
    // ...
```

[1] https://facebook.github.io/react/blog/2015/10/07/react-v0.14.html#stateless-functional-components

Fixed version of the code above.

```
class Greeter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: this.props.initialName
    };
  }

  render() {
    // ...
  }
}
Greeter.defaultProps = { initialName: 'World' };
```

You may also find it useful to use this method to improve declarativeness of your `render` method with this neat trick:

Avoiding conditionals in render method would make it shorter and easier to comprehend.

```
class OnOffCheckboxWithLabel extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      toggled: props.initiallyToggled
    };
    this.toggle = this.toggle.bind(this);
  }

  toggle() {
    this.setState({ toggled: !this.state.toggled });
  }

  render() {
    return (
      <div className="on-off-checkbox">
        <label htmlFor={this.props.id} key="label">
          {this.state.toggled ?
            this.props.onLabel
            :
            this.props.offLabel
          }
```

```
23          </label>
24          <input
25            key="checkbox"
26            type="checkbox"
27            id={this.props.id}
28            checked={this.state.toggled}
29            onChange={this.toggle}
30          />
31        </div>
32      );
33    }
34  }
35  OnOffCheckboxWithLabel.defaultProps = {
36    onLabel: 'On',
37    offLabel: 'Off',
38    id: 'toggle',
39    initiallyToggled: false
40  };
```

This neat trick allows you to avoid conditional in label.

```
1   class OnOffCheckboxWithLabel extends React.Component {
2     // ...
3
4     render() {
5         // ...
6           <label htmlFor={this.props.id} key="label">
7             {this.props.labels[this.state.toggled]}
8           </label>
9         // ...
10
11    }
12  }
13  OnOffCheckboxWithLabel.defaultProps = {
14    onLabel: 'On',
15    offLabel: 'Off',
16    id: 'toggle',
17    labels: {
18      true: 'On',
19      false: 'Off'
20    },
21    initiallyToggled: false
22  };
```

Properties defined in `defaultProps` are overridden by properties passed to an element. That means setting messages or strings in default properties makes our components powerfully configurable out-of-the-box. It greatly improves reusability.

You can use our on/off toggle with more sophisticated labels for free with default properties approach.

```
1 const coolFeatureToggle = <OnOffCheckboxWithLabel
2   id="cool-feature-toggle"
3   labels={{ true: 'Cool feature enabled', false: 'Cool feature disabled' }}
4 />;
```

As you can see, relying on default properties can have many advantages for your component. If you do not want to have configurable messages, you can create a method inside the component and put the `labels` object there. I like to have sensible defaults and to provide ability to change it to match my future needs.

Validating properties

You may validate what kind of properties were sent to the component. Which can be beneficial, especially in a case when the component is reused in many places.

All you have to do is define a value `propTypes` (being a plain object) in your component. Every key in `propTypes` represents a property to validate, while a value says what kind of validation it is. Effect? When a property is invalid, a warning shows in the JavaScript console.

That's a very powerful mechanism, but appears only in development mode.

React comes with built-in validators such as:

- JS primitive type, e.g. string:

  ```
  property: React.PropTypes.string
  ```

- Custom type:

  ```
  property: React.PropTypes.instanceOf(MyClass)
  ```

- Enum:

  ```
  property: React.PropTypes.oneOf(['Foo', 'Bar']),
  ```

- Presence:

  ```
  property: React.PropTypes.any.isRequired
  ```

  (or simply `isRequired` chained to any of above)

You can also define your custom validators. The point is: it should return an `Error` object if the validation fails.

An example of custom validator.

```
1 const customProp = (props, propName, componentName) => {
2   if (!/matchme/.test(props[propName])) {
3     return new Error('Validation failed!');
4   }
5 };
```

After defining such, you can use it instead of functions from `React.PropTypes`:

Usage of custom validator.

```
 1 const customProp = (props, propName, componentName) => {
 2   if (!/matchme/.test(props[propName])) {
 3     return new Error('Validation failed!');
 4   }
 5 };
 6
 7 Blogpost.propTypes = {
 8   // name: React.PropTypes.string.isRequired
 9   name: customProp
10 };
```

To see more of the possible validations, visit Prop Validation section on the official docs[2].

[2]https://facebook.github.io/react/docs/reusable-components.html#prop-validation

Validations in action.

```
class Blogpost extends React.Component {
  render() {
    return (
      <div>
        <div>
          {this.props.name}
        </div>
        <div>
          {`Type: ${this.props.type}`}
        </div>
        <div>
          {this.props.date.toDateString()}
        </div>
        <div>
          {this.props.content}
        </div>
      </div>
    );
  }
}

Blogpost.propTypes = {
  name: React.PropTypes.string.isRequired,
  date: React.PropTypes.instanceOf(Date),
  type: React.PropTypes.oneOf(['Post', 'Ad']),
  content: React.PropTypes.any.isRequired
};

$(() => {
  ReactDOM.render(<Blogpost
    content="This is a short blogpost about the props validation in React"
    date={new Date(2015, 9, 30)}
    name="John Doe"
    type="Post"
  />, document.getElementById('blog'));
});
```

Component children

Let’s look at the React component instance creation with JSX.

```
<Component properties>children</Component>
```

While you already have detailed knowledge about properties, the `children` argument remains a mystery until now. As you may know, HTML of your web page forms a tree structure — you have HTML tags nested in another tags, which nest another tags and so on. React components can have exactly the same structure — and that's what the `children` attribute is for. You can include children simple between the opening and closing tags — it's up to you what you'll do with it. Typically you pass an array of components or a single component there. Such children components are available in a special property called `children`:

Using children in HTML5 React components

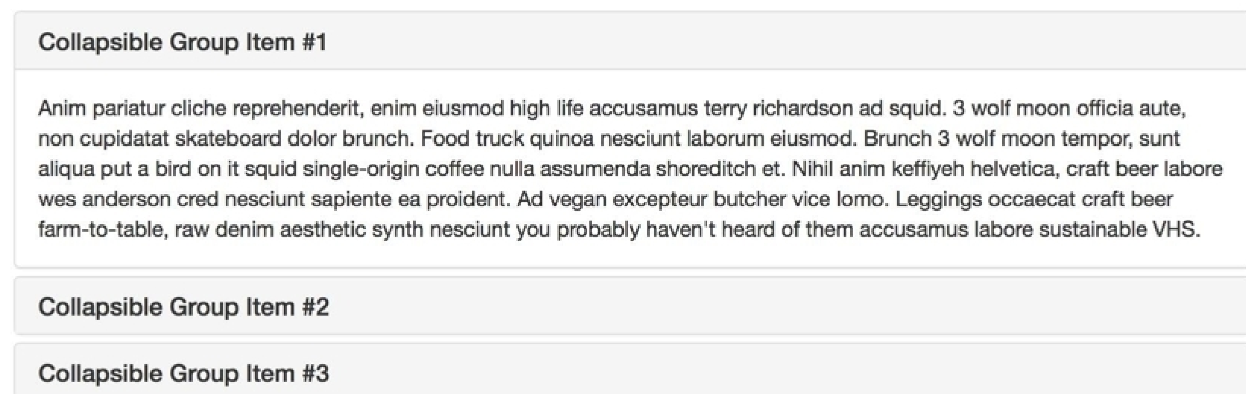
```
return (
  <div className="on-off-checkbox">
    <label htmlFor={this.props.id} key="label">
      {this.props.labels[this.state.toggled]}
    </label>
    <input
      key="checkbox"
      type="checkbox"
      id={this.props.id}
      checked={this.state.toggled}
      onChange={this.toggle}
    />
  </div>
);
```

You can access children using a special `children` field inside properties.

```
class WhoIsInvited extends React.Component {
  render() {
    return (
      <div className="who-is-invited-box">
        <h3 className="who-is-invited-header" key="header">
          You"ve invited this people to tonight"s pajama party:
        </h3>
        <ul className="who-is-invited-list" key="invited-list">
          {this.props.children.map(function (personInformationComponent) {
            return (
              <li
                className="who-is-invited-list-item"
                key={`person-${personInformationComponent.person().id}}`}
              >
```

```
15                {personInformationComponent}
16              </li>
17            );
18          })}
19        </ul>
20      </div>
21    );
22  }
23  }
24
25  invitations = (
26    <WhoIsInvited>
27      <KidInformation />
28      <AdultInformation />
29      <AdultInformation />
30    </WhoIsInvited>
31  )
32
33  // this.props.children of invitations component will contain
34  // [KidInformation(...), AdultInformation(...), AdultInformation(...)] array.
```

This feature allows you to easily create some kind of *wrapper components* (I call it *open components*) like a list above. It's quite common to have these kind of user interface elements — many well-established generic UI solutions can be implemented this way. For example — accordion boxes, drop-down menus, modals… all of these generic solutions you know and use can follow this pattern, since they are generally 'containers' to our content.



Accordion boxes are great candidates to be implemented as open components in React.

Properties and state are a way of React component to handle data which can be a result of user actions or events from the external world (messaging mechanism like Pusher, AJAX calls, websockets, etc.). **While state is all about dynamism of components (and that's why you have switched from**

Rails views to frontend applications after all), properties allow you to provide data for more 'static' purposes in our component.

`render` method

Last, but not least, all React components must have the `render` method. **This method always returns a React component**. Since you compose your component from smaller components (with HTML5 components e.g. `div`, `span` as leafs of your *'components tree'*) it is an understandable constraint here.

It can return `false` or `null` as well. In such case React will render an invisible `<noscript>` tag.

What is important, you should never call the `render` method explicitly. This method is called automatically by React in these two situations:

- If the state changes. It also calls `componentDidUpdate` method.
- If you mount component using `ReactDOM.render`. It also calls the `componentDidMount` method.

The lifecycle methods such as the mentioned `componentDidMount` and `componentDidUpdate` will be discussed later in the book.

An important assumption that React makes about your `render` method is that it is **idempotent** - that means, calling it multiple times with the same properties and state will always result in the same outcome.

React components style (especially with JSX syntax extension) resembles HTML builders a bit.

What you achieve with state and properties is dynamic nature of this HTML - you can click on something to make an action, change the rendering of this HTML and so on. It looks really familiar to what you may find in `.html.erb` view files:

Example of ERB view

```
1 <section class='container'>
2   <h1><%= post.title %></h1>
3   <h2><%= post.subtitle %></h1>
4   <div class='content'>
5     <%= post.content %>
6   </div>
7 </section>
```

React counterpart

```
<section className="container">
  <h1>{this.state.post.title}</h1>
  <h2>{this.state.post.subtitle}</h2>
  <div className="content">
    {this.state.post.content}
  </div>
</section>
```

What you should do in the `render` method is:

- **Computing the auxiliary data** you may need in your user interface from state and properties. You should not cache this result in those places. I'd recommend to only call computation methods and never inline this logic inside `render`.

Example of precomputing values in render method.

```
class PersonGreeter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      person: new Person('Bruce', 'Wayne')
    };
  }

  personFullName(person) {
    return [person.firstName, person.lastName].join(' ');
  }

  greeterSpan(children) {
    return (
      <div className="greeter-text">
        {children}
      </div>
    );
  }

  greeterBox(children) {
    return (
      <div className="greeter-box">
        {children}
      </div>
```

```
26      );
27    }
28
29    render() {
30      // Computing full name of person
31      const fullName = this.personFullName(this.state.person);
32
33      return (
34        <div>{this.greeterSpan(`Hello, ${fullName}!`)}</div>
35      );
36    }
37  }
```

- **Creating components based on properties and state**. Since you should never store components in state or properties, the React way is to construct components inside the `render` method.

Creating components based on properties and state.

```
 1  class BooksListItem extends React.Component {
 2    render() {
 3      const name = this.props.book.name;
 4      return (
 5        <li key={name} >{name}</li>
 6      );
 7    }
 8  }
 9
10  class BooksList extends React.Component {
11    render() {
12      return (
13        <ul className="book-list">
14          {this.props.books.map(function (book) {
15            // You create components from books in our properties.
16            return (<BooksListItem book={book} />);
17          })}
18        </ul>
19      );
20    }
21  }
```

- **Define the HTML structure of a user interface part**. That means you can compose the return value from HTML5 components directly or with higher level components. Eventually it is transformed into HTML.

Do not try to memoize data in the `render` method. Especially components. React is managing the component's lifecycle using its internal algorithm. Trying to re-use component by hand is asking for trouble, since it is easy to violate React rules for re-rendering here.

Mixins

Mixins are not supported for ES6 classes in React.

Basically, mixins are the snippets of code that can be shared by many components. The mixin objects can contain both variables and functions (which can be included to a component as, respectively, properties and methods).

Mixins can define *lifecycle methods* (if you are not familiar, jump to Lifecycle methods), and they do it smartly. As the Mixins section on official docs[3] says:

> A nice feature of mixins is that if a component is using multiple mixins and several mixins define the same lifecycle method (i.e. several mixins want to do some cleanup when the component is destroyed), all of the lifecycle methods are guaranteed to be called. Methods defined on mixins run in the order mixins were listed, followed by a method call on the component.

See this on an example:

A mixin defining lifecycle methods

```
1  const loggingAtMountingMixin = {
2    componentWillMount() {
3      console.log('Component about to be mounted...');
4    },
5    componentDidMount() {
6      console.log('Component mounted');
7    }
8  };
9
10 const MyComponent = React.createClass({
11   mixins: [loggingAtMountingMixin],
12   render() {
13     return null;
14   }
15 });
```

There are two mixins predefined in `React.addons`:

[3] https://facebook.github.io/react/docs/reusable-components.html#mixins

- PureRenderMixin[4] - to boost performance under certain circumstances
- LinkedStateMixin[5] - to express two-way binding

Context

Context[6] is another way (beside properties) to pass a value to a child component. The difference between `context` and `props` is: *once set, context is being passed all the way down, to all the descendants.*

When you want the whole subtree to have an access to the same "attribute", you can use context - instead of passing the same property again and again.

For example, consider a single component that renders a `user`. This component has many sub-components as its children - usually, having many small components instead of a huge one is a good practice. Unfortunatelly, the small sub-components don't have an access to the `user` variable. To pass it, you can use context:

```
1   class UserComponent extends React.Component {
2     getChildContext() {
3       return { user: this.props.user };
4     }
5
6     render() {
7       return (
8         <div>
9           <Header />
10          <Avatar />
11          <Ranking />
12          <Stats />
13        </div>
14      );
15    }
16  }
17  UserComponent.childContextTypes = {
18    user: React.PropTypes.object
19  };
20
21  class Header extends React.Component {
22    render() {
```

[4] https://facebook.github.io/react/docs/pure-render-mixin.html
[5] https://facebook.github.io/react/docs/two-way-binding-helpers.html
[6] https://facebook.github.io/react/docs/context.html

```
23      return (
24        <h1>{this.context.user.name}</h1>
25      );
26    }
27  }
28  Header.contextTypes = {
29    user: React.PropTypes.object
30  };
31
32  class Avatar extends React.Component {
33    render() {
34      return (
35        <a href=`${"http://www.example.org/profile/"}${this.context.user.id}">
36          <img src={this.context.user.avatarUrl} alt={this.context.user.name} />
37        </a>
38      );
39    }
40  }
41  Avatar.contextTypes = {
42    user: React.PropTypes.object
43  };
44
45  class Stats extends React.Component {
46    render() {
47      return (
48        <div>
49          Won {this.context.user.won}
50          <br />
51          Lost {this.context.user.lost}
52          <br />
53          <a href={`${'http://www.example.org/stats/'}${this.context.user.id}`} >
54            (see the whole stats of {this.context.user.name})
55          </a>
56        </div>
57      );
58    }
59  }
60  Stats.contextTypes = {
61    user: React.PropTypes.object
62  };
63
64  class Ranking extends React.Component {
```

```
65    render() {
66      return (
67        <h2>
68          <a href={`${'http://www.example.org/ranking/'}${this.context.user.id}`} >
69            Ranking
70          </a>
71        </h2>
72      );
73    }
74  }
75  Ranking.contextTypes = {
76    user: React.PropTypes.object
77  };
78
79  const amalie = {
80    id: 1,
81    name: 'Amalie',
82    avatarUrl: 'http://www.example.org/avatar/123',
83    won: 13,
84    lost: 7,
85    ranking: 14,
86  };
87
88  $(() => {
89    ReactDOM.render(<UserComponent user={amalie} />, document.body)
90  });
```

The `UserComponent` declares the values in the context for its descendants via `getChildContext` object. Also the types of each value must be specified in child `contextTypes`. The mechanism for definig the types is the same as for validating properties - `React.PropTypes`.

Components have `this.context` being an empty object - `{}` - by default. Sub-components must explicitly declare (in `contextTypes`) which parts of the context they will access, otherwise they won't be able to use it.

It's OK *not* to declare `contextTypes` if a component doesn't need any value from the context. If you refer to `this.context.attribute` in a child `render` method, without specifying the `attribute` type in `contextTypes`, you will get an error.

The parent component has no access to its own context - it is meant for the child components only. If you need the context from the outermost component, you have to wrap it within another component, which defines `getChildContext` and `childContextTypes`.

A context can be extended by new values in a sub-component, and existing values can be overwriten.

References (refs) to components

Every component can have a `ref` attribute set. Its value can be one of two types, each having a different purpose:

`ref` as a string

Every component has `this.refs` property which contains references to mounted child components with `ref` attribute set. The value of `ref` identifies the component in parent's `this.refs`. For example, `ref: 'anInput'` converts to `this.refs.anInput`.

So `this.refs` are a shoutcut to some of child components. Often they are used for finding the DOM node from a child component, like this:

```
ReactDOM.findDOMNode(this.refs.focusableContainer)
```

As said before, you can only access the components that are already mounted. If you want to access a `ref` as soon as possible, you should use `componentDidMount` lifecycle method (if you are not familiar, jump to Lifecycle methods).

See the whole example, where exactly this mechanism is used. This is a wrapper for a casual HTML `input` tag. The wrapper makes sure that after the input is rendered the browser's focus will be on it.

```
1  class FocusableWrapper extends React.Component {
2    componentDidMount() {
3      ReactDOM.findDOMNode(this.refs.focusableContainer).children[0].focus();
4    }
5
6    render() {
7      return (
8        <div
9          ref="focusableContainer"
10       >
11         {this.props.children}
12       </div>
13     );
14   }
15 }
```

In most cases using `this.refs` can be avoided by using natural data-flow mechanisms like callbacks, props or events. This is why Refs to Components section of the React docs[7] comments string-refs as *mostly legacy at this point.*

`ref` as a string may be also useful in testing - see Testing with references.

[7] https://facebook.github.io/react/docs/more-about-refs.html#the-ref-string-attribute

DOM node refs

Since React 0.14 refs to DOM (HTML5) components are exposed as the browser DOM node itself. Note that refs to custom (user-defined) components work exactly as before; only the built-in DOM components are affected by this change.

Our `focusableContainer` is a `div` tag, a *built-in DOM component*, not a *custom* one. We don't need to use `ReactDOM.findDOMNode(this.refs.focusableContainer)`, altough it won't hurt. Instead, `this.refs.focusableContainer` will suffice.

```
1  class FocusableWrapper extends React.Component {
2    componentDidMount() {
3      this.refs.focusableContainer.children[0].focus();
4    }
5
6    render() {
7      return (
8        <div
9          ref="focusableContainer"
10       >
11         {this.props.children}
12       </div>
13     );
14   }
15 }
```

`ref` as a function

In particular - a callback function. It will be called immediately after the component is mounted. The callback can take an argument - the referenced component.

```
1  class FocusableWrapper extends React.Component {
2    render() {
3      return (
4        <div
5          ref={function (div) {
6            if (div) { ReactDOM.findDOMNode(div).children[0].focus(); }
7          }}
8        >
9          {this.props.children}
10       </div>
11     );
12   }
13 }
```

One thing to remember is (after the Refs to Components section of the React docs[8]): *when the referenced component is unmounted and whenever the ref changes, the old ref will be called with `null` as an argument.*

So if we define a callback like that, in the `render` method, the callback will be a new function in every re-render. It means that when the component is re-rendered, the `ref` callback function defined in previous `render` would be called with a `null` argument, and the ref callback function defined in current call of render would be called with non-null argument.

This is why we have the condition (`if div`). You might think we could avoid it by defining the callback once, in the component - instead of inside the `render` method.

```
1  class FocusableWrapper extends React.Component {
2    focusCallback(div) {
3      ReactDOM.findDOMNode(div).children[0].focus();
4    }
5
6    render() {
7      return (
8        <div
9          ref={this.focusCallback}
10       >
11         {this.props.children}
12       </div>
13     );
14   }
15 }
```

This is beneficial because re-render does not define a new function so it avoids problems mentioned in previous paragraph. But `focusCallback` can be called with `null` when `FocusableWrapper` component is unmounted. So in the end we need that `if` anyway.

```
1  class FocusableWrapper extends React.Component {
2    focusCallback(div) {
3      if (div) {
4        ReactDOM.findDOMNode(div).children[0].focus();
5      }
6    }
7
8    render() {
9      return (
10       <div
```

[8] https://facebook.github.io/react/docs/more-about-refs.html#the-ref-callback-attribute

```
11          ref={this.focusCallback}
12        >
13          {this.props.children}
14        </div>
15      );
16    }
17  }
```

Desugaring JSX

`React.DOM.tagName(properties, children)`

Instead of using JSX syntax we could write `React.DOM.div({}, "Hello world!")`. `React.DOM` is a namespace for components representing HTML5 tags. The most obvious one is `React.DOM.div`. **They are not real HTML5 tags. They are just their representation**. We can use them to describe what actual tags should be rendered by React in the browser.

Example

Desugaring JSX is quite easy. Consider the following JSX example (taken from the official documentation):

JSX example to desugar

```
1   <div>
2     <ProfilePic key="pic" username={this.props.username} />
3     <ProfileLink key="link" username={this.props.username} />
4   </div>
```

For each component, JSX is composed this way:

JSX is composed like this.

```
1 <ComponentName property1=value1 property2=value2 ...>[children]</ComponentName>
2 or:
3 <ComponentName property1=value1 property2=value2 ... /> // children set to null
```

There is a special notation inside curly brackets for a runtime evaluation.

That means the previous JSX desugars to this:

Desugared JSX example

```
const profilePic = React.createFactory(ProfilePic);
const profileLink = React.createFactory(ProfileLink);

React.DOM.div({}, [
  profilePic({key: 'pic', username: this.props.username)},
  profileLink({key: 'link', username: this.props.username})
])
```

Desugared JSX example (inlined)

```
React.DOM.div({}, [
  React.createElement(ProfilePic, {key: 'pic', username: this.props.username }),
  React.createElement(ProfileLink,{key: 'link', username: this.props.username})
])
```

You can also use JSX compiler[9] from the official site.

Summary

In this chapter you learned about the three main building blocks of React components — state, properties and `render` method. This alone is a very complete knowledge that you can use straight away. Since you know how to create a component, feed it with data and manipulate it, you can make fairly complex components even now. What you lack is mastering how to handle actions to user and the external world. The next chapter is all about it.

[9] http://facebook.github.io/react/jsx-compiler.html