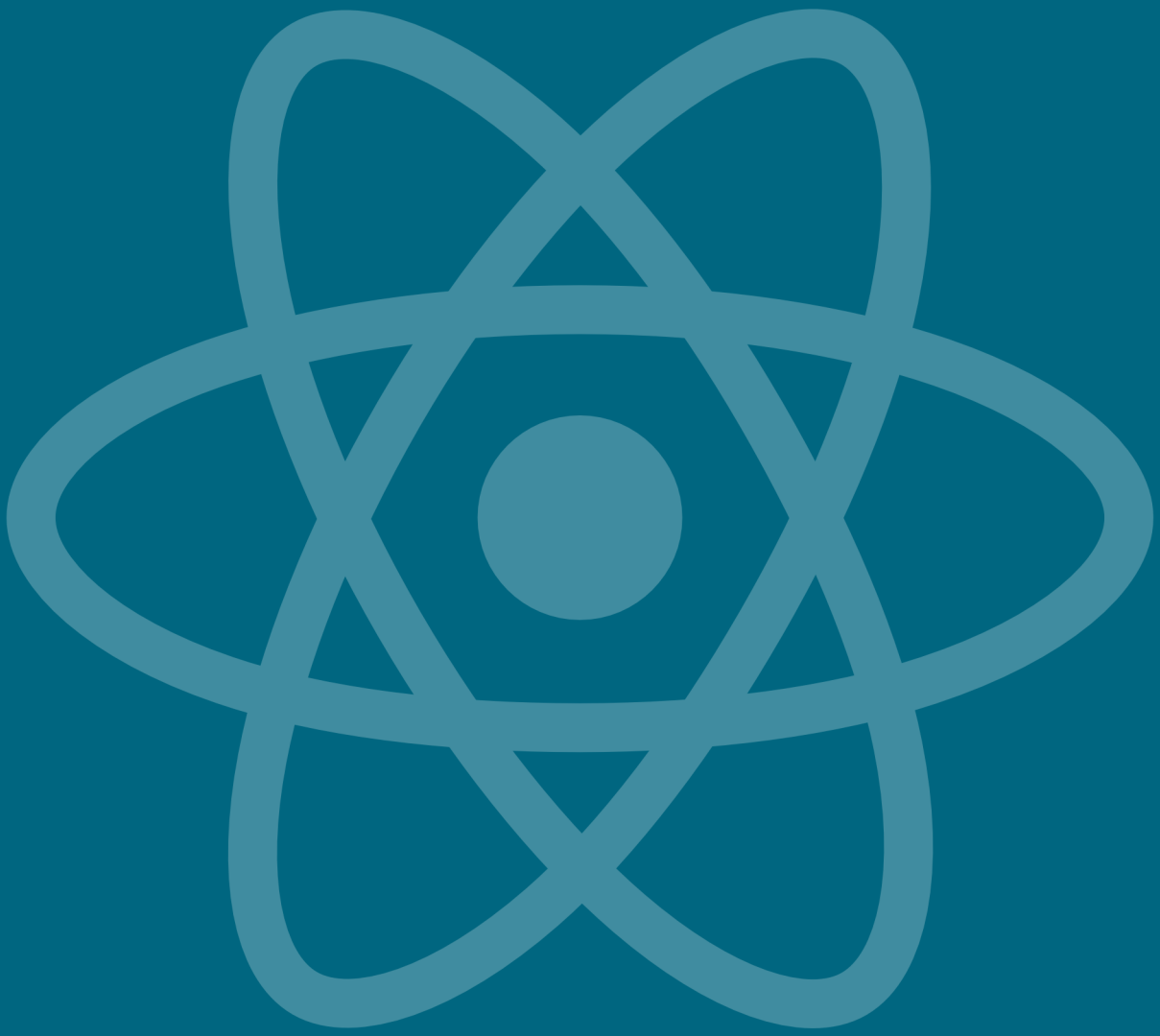
# react.js

## by example



# Password Strength Meter


**Password Strength Meter**

Registration form is like the first step that user needs to take to use your web application. It's interesting how often it is not optimal part of the app. Having an unfriendly registration form may hurt (and usually hurts) the conversion rate of your service badly.

That's why dynamic features are often starting with forms. On-the-fly validations, popovers and so on - all of these are common in the modern web. All to increase chance of signing up by an user.

Apart from the sole signing up, a good registration form needs to make sure that an user does not do anything wrong - like setting too simple password. Password strength meters are a great way to show an user how his password should be constructed to be secure.

## Requirements

This example will use React-Bootstrap[1] components. **Remember that React-Bootstrap must be installed separately - visit the main page of the project for installation details**. Using React Bootstrap simplifies the example because common UI elements like progress bars don't need to be created from scratch.

Apart from this, a tiny utility called classnames[2] will be used. It allows you to express CSS class set with conditionals in an easy way.

Of course the last element is the React library itself.

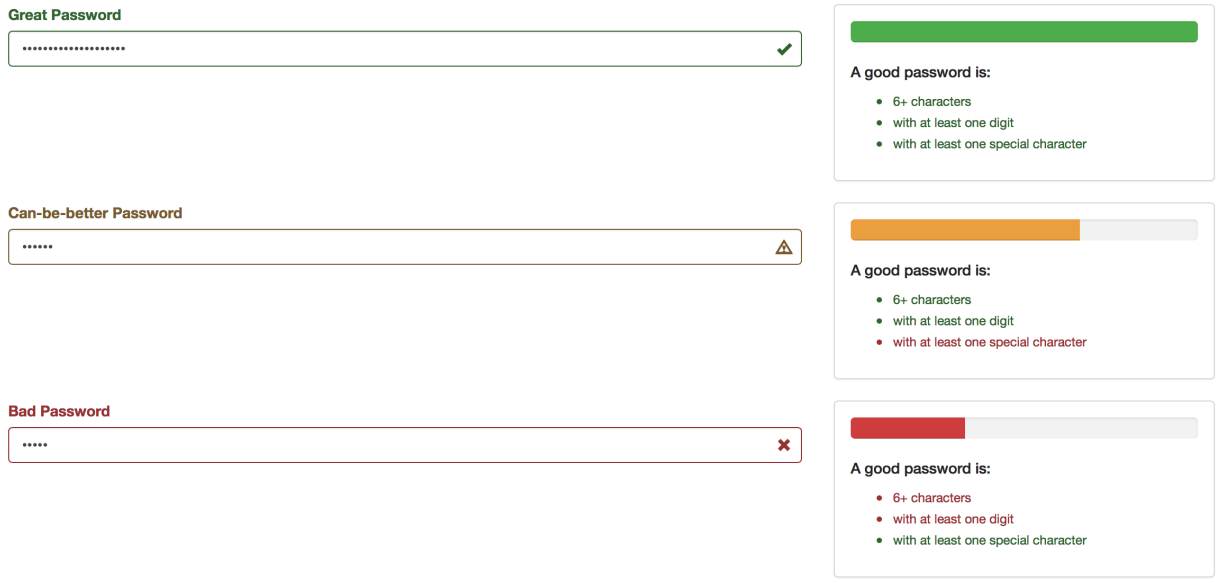
## Recipe

In this example you don't need to make any assumptions about how the password strength meter will work. There is the static HTML mockup ready to reference how the strength meter will look and

[1] http://react-bootstrap.github.io

[2] https://www.npmjs.com/package/classnames

behave, based on the password input. It is written using the Bootstrap CSS framework, so elements presented will align well with components that React-Bootstrap provides.




**Password Strength Meter States**

```
<div class="container">
  <div class="row">
    <div class="col-md-8">
      <div class="form-group has-success has-feedback">
        <label class="control-label"
               for="password-input">Password</label>
        <input type="password"
               class="form-control"
               id="password-input"
               value="FW&$2iVaFt3va6bGu4Bd"
               placeholder="Password" />
        <span class="glyphicon glyphicon-ok form-control-feedback"
              aria-hidden="true"></span>
      </div>
    </div>
    <div class="col-md-4">
      <div class="panel panel-default">
        <div class="panel-body">
          <div class="progress">
            <div class="progress-bar progress-bar-success"
                 style="width:100%"></div>
          </div>
```

```
23            <h5>A good password is:</h5>
24            <ul>
25              <li class="text-success">
26                <small>
27                  6&plus; characters
28                </small>
29              </li>
30              <li class="text-success">
31                <small>
32                  with at least one digit
33                </small>
34              </li>
35              <li class="text-success">
36                <small>
37                  with at least one special character
38                </small>
39              </li>
40            </ul>
41          </div>
42        </div>
43      </div>
44      <!-- Rest of states... -->
45    </div>
46  </div>
```

This piece of HTML defines the whole structure that will be duplicated. All “creative” work that needs to be done here is to attach the dynamic behaviour and state transitions.

There are some *principles* that states how a good password should look like. You can think that a password *satisfies* or not those principles. That will be important later - your behaviour will be built around this concept.

As you can see, there are three states of the strength meter UI:

- Awful password - progress bar is red and an input is in “red” state (1/3 or less principles satisfied)
- Mediocre password - progress bar is yellow and an input is in “yellow” state (1/3 to 2/3 principles satisfied)
- Great password - progress bar is green and an input is in “green” state (2/3 or more principles satisfied)

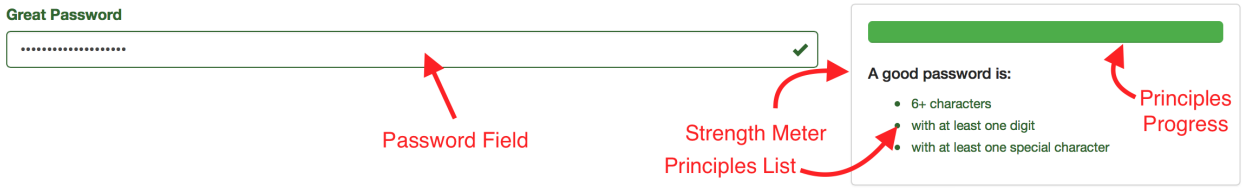
Since you got a HTML mockup, after each step you can compare output produced by React with HTML markup of the static example. Another approach (see `Prefixer` example if you want to see

this approach in action) is to *copy* this HTML and then attach the dynamic behaviour to it. In this example the code will start from the top. First an empty component will be created and then the *logical* parts of this UI will be defined. After those steps there will be iterations to finish with the markup desired.

Enough said, let's start with an empty component:

```
class PasswordInput extends React.Component {
  render() { return null; }
}
```

Let's think about what logical parts this part of UI has. On the highest level it has a *strength meter* and a *password field*. A *strength meter* consist of *principles progress* and *principles list*.




**Annotated Password Strength Meter**

This concepts will map directly to React components that you'll create. A static markup is also placed inside the grid. You can use `Grid`, `Row` and `Col` to create a HTML markup like this:

```
<div class="container">
  <div class="row">
    <div class="col-md-8">
      ...
    </div>
    <div class="col-md-4">
      ...
    </div>
  </div>
</div>
```

Which maps directly into:

```
<Grid>
  <Row>
    <Col md={8}>
      ...
    </Col>
    <Col md={4}>
      ...
    </Col>
  </Row>
</Grid>
```

Remember to `import` needed React-Bootstrap components at the top of the file:

```
import { Grid, Row, Col } from 'react-bootstrap';
```

Let's mimic the top structure (the grid) of your markup and define components based on the logical division!

```
class PasswordInput extends React.Component {
  render() {
    return (
      <Grid>
        <Row>
          <Col md={8}>
            <PasswordField />
          </Col>
          <Col md={4}>
            <StrengthMeter />
          </Col>
        </Row>
      </Grid>
    );
  }
}

class StrengthMeter extends React.Component {
  render() { return null; }
}

class PasswordField extends React.Component {
  render() { return null; }
}
```

So far, so good. In this step you have a "framework" to work with. Another step is to add data. Default properties technique can be very helpful here. *Good password principles* will have a name and a predicate to check whether a principle is satisfied or not. Such predicate will get a password as an argument - it's a simple plain JavaScript function.

```
const SPECIAL_CHARS_REGEX = /[^A-Za-z0-9]/;
const DIGIT_REGEX = /[0-9]/;

PasswordInput.defaultProps = {
  goodPasswordPrinciples: [
    {
      label: "6+ characters",
      predicate: password => password.length >= 6
    },
    {
      label: "with at least one digit",
      predicate: password => password.match(DIGIT_REGEX) !== null
    },
    {
      label: "with at least one special character",
      predicate: password => password.match(SPECIAL_CHARS_REGEX) !== null
    }
  ]
};
```

As you can see, the default principles are taken straight from the mockup. You can provide your own while instantiating the `PasswordInput` component, making it powerfully configurable for free.

Since in this stage you got two logical components to implement, you need to choose one. In this recipe `StrengthMeter` will be implemented as the first one.

Let's render *something*. Since in the static mockup the whole strength meter is wrapped within a Bootstrap's Panel, let's render the empty panel at first. Remember to import `Panel` component class from the React-Bootstrap package:

```
import { Grid, Row, Col, Panel } from 'react-bootstrap';
```

Then you can use it:

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

Let's start with implementing a static list of principles, without marking them in color as satisfied/not satisfied. It is a good starting point to iterate towards the full functionality. To do so, you need to pass principles list to the `StrengthMeter` component. To do so, simply pass the `principles` property from the `PasswordInput` component:

```
class PasswordInput extends React.Component {
  render() {
    let { goodPasswordPrinciples } = this.props;

    return (
      <Grid>
        <Row>
          <Col md={8}>
            <PasswordField />
          </Col>
          <Col md={4}>
            <StrengthMeter principles={goodPasswordPrinciples} />
          </Col>
        </Row>
      </Grid>
    );
  }
}
```

Now the data can be used to render a list of principles:

```
class StrengthMeter extends React.Component {
  render() {
    let { principles } = this.props;

    return (
      <Panel>
        <ul>
          {principles.map(principle =>
          <li>
            <small>
              {principle.label}
            </small>
```

```
            </li>
            )}
          </ul>
        </Panel>
      );
    }
}
```

Notice how `<small>` is used inside of the list element. That's how it is done within the static mockup - and ultimately you want to achieve the same effect.

So far, so good. A tiny step to make is to add a header just like on the mockup:

```
class StrengthMeter extends React.Component {
  render() {
    let { principles } = this.props;

    return (
      <Panel>
        <h5>A good password is:</h5>
        <ul>
          {principles.map(principle =>
          <li>
            <small>
              {principle.label}
            </small>
          </li>
          )}
        </ul>
      </Panel>
    );
  }
}
```

Now it's time to implement logic for coloring this list whether a given principle is satisfied or not. Since satisfying process needs the `password` as an argument, it's time to introduce the `password` variable in the `PasswordInput` state. It lies within the state because it'll change in a process - and will trigger appropriate re-renders.

To do so, you need to introduce a constructor to the `PasswordInput` component class which will set the default `password` variable to `''`. Let's do it!

```
class PasswordInput extends React.Component {
  constructor(props) {
    super(props);
    this.state = { password: '' };
  }

  render() {
    let { goodPasswordPrinciples } = this.props;

    return (
      <Grid>
        <Row>
          <Col md={8}>
            <PasswordField />
          </Col>
          <Col md={4}>
            <StrengthMeter principles={goodPasswordPrinciples} />
          </Col>
        </Row>
      </Grid>
    );
  }
}
```

So far, so good. But you need the `password` information within the `StrengthMeter`. It can be done simply by passing the property to `StrengthMeter`:

```
class PasswordInput extends React.Component {
  constructor(props) {
    super(props);
    this.state = { password: '' };
  }

  render() {
    let { goodPasswordPrinciples } = this.props;
    let { password } = this.state;

    return (
      <Grid>
        <Row>
          <Col md={8}>
            <PasswordField />
```

```
          </Col>
          <Col md={4}>
            <StrengthMeter principles={goodPasswordPrinciples}
                           password={password} />
          </Col>
        </Row>
      </Grid>
    );
  }
}
```

Strength meter now got the password provided. That means you can provide a handy method for checking whether a principle is satisfied or not.

```
class StrengthMeter extends React.Component {
  principleSatisfied(principle) {
    let { password } = this.props;

    return principle.predicate(password);
  }

  render() {
    let { principles } = this.props;

    return (
      <Panel>
        <h5>A good password is:</h5>
        <ul>
          {principles.map(principle =>
          <li>
            <small>
              {principle.label}
            </small>
          </li>
          )}
        </ul>
      </Panel>
    );
  }
}
```

Since you got your *primary* information defined as a handy method, now you should transform it into something visual. Here the `classNames` utility will be used to set CSS classes on principle list

elements based on the status of principle satisfaction. Remember to `import` the appropriate function:

```
import classNames from 'classnames';
```

With this utility you can create `principleClass` method which will return the appropriate class for the principle list element:

```
class StrengthMeter extends React.Component {
  principleSatisfied(principle) {
    let { password } = this.props;

    return principle.predicate(password);
  }

  principleClass(principle) {
    let satisfied = this.principleSatisfied(principle);

    return classNames({
      ["text-success"]: satisfied,
      ["text-danger"]: !satisfied
    });
  }

  render() {
    let { principles } = this.props;

    return (
      <Panel>
        <h5>A good password is:</h5>
        <ul>
          {principles.map(principle =>
          <li className={this.principleClass(principle)>
            <small>
              {principle.label}
            </small>
          </li>
          )}
        </ul>
      </Panel>
    );
  }
}
```

`classNames` takes an object with CSS classes as keys - and creates an appropriate class string from all keys that has values evaluating to the truthy value. It allows you to work with conditional CSS classes in an easy way. In previous versions of React it was the built in utility from the `React.addons`, called `classSet`. In recent versions of React it's gone and needs to be installed separately.

Next interesting thing in this example is the new ECMAScript 2015 syntax for defining keys. If you use `[` and `]` brackets the key defined will be a return value of an expression inside those brackets. It allows you to define keys based on function return values, use string literals to define keys with special symbols like `"-"`, or use backticks string syntax to define keys with interpolated values in it. Neat!

To test whether this logic works or not, try to change the password default value - you'll see that appropriate CSS classes will get appended to list elements.

That's how the logical piece of *principle list* is implemented. As has been said before, it's usual that in React such logical pieces are mapped directly into components. That means you should extract a `PrinciplesList` component out of `StrengthMeter` component and use it. It's simple. You just need to copy logic from the `StrengthMeter` component down and use the newly component as a replacement to a piece of previous tree rendered by `render`. It can be done like this:

```
class StrengthMeter extends React.Component {
  render() {
    return (
      <Panel>
        <h5>A good password is:</h5>
        <PrinciplesList {...this.props} />
      </Panel>
    );
  }
}

class PrinciplesList extends React.Component {
  principleSatisfied(principle) {
    let { password } = this.props;

    return principle.predicate(password);
  }

  principleClass(principle) {
    let satisfied = this.principleSatisfied(principle);

    return classNames({
      ["text-success"]: satisfied,
      ["text-danger"]: !satisfied
```

```
25      });
26    }
27
28    render() {
29      let { principles } = this.props;
30
31      return (
32        <ul>
33          {principles.map(principle =>
34          <li className={this.principleClass(principle)}>
35            <small>
36              {principle.label}
37            </small>
38          </li>
39          )}
40        </ul>
41      );
42    }
43  }
```

As you can see, it's a fairly mechanical step to do - `principleSatisfied` and `principleClass` are moved down to the `PrinciplesList` component. Then you cut the part of the tree from `render` (In this case `<ul>....</ul>`) and rendered it within the lower-level component.

Since it is a new component, you must pass needed properties down to it. And there is a very interesting syntax used. You can use `{...object}` syntax to pass the whole object as properties in JSX. It is part of the bigger feature called *object spread operator*. You can use it in ECMAScript 2016 (a.k.a ECMAScript 7 or ES7) codebase today - and read more about it here[3]. One of the transpilers that support it is Babel.js[4]. It is built into JSX regardless you use ECMAScript 2016 features in your codebase or not.

Since your `this.props` in `StrengthMeter` component is `{ principles: <$1>, password: <$2> }`, the syntax:

```
1  <PrinciplesList {...this.props} />
```

Is equal to saying:

```
1  <PrinciplesList principles={<$1>} password={<$2>} />
```

[3] https://github.com/sebmarkbage/ecmascript-rest-spread
[4] http://babeljs.io/docs/usage/experimental/

It is a very handy shortcut to passing *all* or *all except some* of properties down to the lower-level components.

OK. One of the logical pieces is done - and a component representing it is created. To finish the *strength meter* logical piece, there is one more thing - a progress bar which brings the visual feedback how strong your password is.

Let's start with a static progress bar. Remember to `import` it from your `react-bootstrap` package:

```
import { Grid, Row, Col, Panel, ProgressBar } from 'react-bootstrap';
```

Then, add it in your `StrengthMeter` component. Why there? Because you'll extract the `PrinciplesProgress` component later, just like you did with `PrinciplesList`.

```
class StrengthMeter extends React.Component {
  render() {
    return (
      <Panel>
        <ProgressBar now={50} />
        <h5>A good password is:</h5>
        <PrinciplesList {...this.props} />
      </Panel>
    );
  }
}
```

As you can see, `now` property manages how the progress bar is filled. Let's attach a behaviour which will manage this number.

```
class StrengthMeter extends React.Component {
  satisfiedPercent() {
    let { principles, password } = this.props;

    let satisfiedCount = principles.map(p => p.predicate(password))
                                   .reduce((count, satisfied) =>
                                      count + (satisfied ? 1 : 0)
                                   , 0);

    let principlesCount = principles.length;

    return (satisfiedCount / principlesCount) * 100.0;
  }

```

```
15    render() {
16      return (
17        <Panel>
18          <ProgressBar now={this.satisfiedPercent()} />
19          <h5>A good password is:</h5>
20          <PrinciplesList {...this.props} />
21        </Panel>
22      );
23    }
24  }
```

Computing this percent is made by using two functions from the standard library - `map` and `reduce`.

To compute how many principles are satisfied, an array of principles is taken. Then it is *mapped* to an array which contains boolean values of predicate results. So if your password is `'1$a'`, the `principles.map(p => p.predicate(password))` will return `[false, true, true]` array.

After computing this result, a `reduce` is called to obtain the count of satisfied principles.

`reduce` function takes two parameters:

- an *accumulating function* which will get called with two arguments: an *accumulating value* and an element of the array;
- a starting *accumulating value*:

The idea is simple - `reduce` iterates through your array and modifies its *accumulating value* after each step. After traversing the whole array, the final *accumulating value* is returned as a result. It is called *folding* a collection in functional languages.

The *accumulating value* passed to the current element is the return value of the *accumulating function* called on the previous element or the starting value if it is a first element.

So in case of this `[false, true, true]` array described before, `reduce` will do the following things:

- Call the *accumulating function* with arguments `0` and `false`. Since the second argument is `false`, `0` is returned from this function.
- Call the *accumulating function* with arguments `0` and `true`. Since the second argument is `true`, `1` is added to `0`, resulting in a return value of `1`.
- Call the *accumulating function* with argument `1` and `true`. Since the second argument is `true`, `1` is added to `1`, resulting in a return value of `2`.
- There are no more elements in this array. `2` is returned as a return value of the whole `reduce` function.

You can read more about this function here[5]. It can make your code much more concise - but be careful to not hurt maintainability. Accumulating functions should be short and the whole result properly named.

Since your `satisfiedCount` is computed, the standard equation for computing percent is used.

All that is left is to provide a proper style ("green" / "yellow" / "red" state described before) of the progress bar, based on the computed percent.

- Awful password - progress bar is red and an input is in "red" state (1/3 or less principles satisfied)
- Mediocre password - progress bar is yellow and an input is in "yellow" state (more than 1/3 to 2/3 principles satisfied)
- Great password - progress bar is green and an input is in "green" state (2/3 or more principles satisfied)

To do so, let's introduce another method that will check these 'color states'.

```
1  class StrengthMeter extends React.Component {
2    satisfiedPercent() {
3      let { principles, password } = this.props;
4
5      let satisfiedCount = principles.map(p => p.predicate(password))
6                                     .reduce((count, satisfied) =>
7                                        count + (satisfied ? 1 : 0)
8                                     , 0);
9
10     let principlesCount = principles.length;
11
12     return (satisfiedCount / principlesCount) * 100.0;
13   }
14
15   progressColor() {
16     let percentage = this.satisfiedPercent();
17
18     return classNames({
19       danger: (percentage < 33.4),
20       success: (percentage >= 66.7),
21       warning: (percentage >= 33.4 && percentage < 66.7)
22     });
23   }
```

[5] https://developer.mozilla.org/pl/docs/Web/JavaScript/Referencje/Obiekty/Array/Reduce

```
24
25    render() {
26      return (
27        <Panel>
28          <ProgressBar now={this.satisfiedPercent()}
29                       bsStyle={this.progressColor()} />
30          <h5>A good password is:</h5>
31          <PrinciplesList {...this.props} />
32        </Panel>
33      );
34    }
35  }
```

Neat thing about `classNames` is that you can also use it here - look at how it is used in this example. Since all color state options are mutually exclusive, only the single string will get returned - which is also a valid CSS class statement. It allows us to express this logic in an elegant way without `if`'s.

That means we got all pieces of the strength meter done. You can switch to `PasswordField` implementation. But first, extract the logical pieces of *principles progress* into a separate component.

```
1   class StrengthMeter extends React.Component {
2     render() {
3       return (
4         <Panel>
5           <PrinciplesProgress {...this.props} />
6           <h5>A good password is:</h5>
7           <PrinciplesList {...this.props} />
8         </Panel>
9       );
10    }
11  }
12
13  class PrinciplesProgress extends React.Component {
14    satisfiedPercent() {
15      let { principles, password } = this.props;
16
17      let satisfiedCount = principles.map(p => p.predicate(password))
18                                     .reduce((count, satisfied) =>
19                                        count + (satisfied ? 1 : 0)
20                                     , 0);
21
22      let principlesCount = principles.length;
23
```

```
    return (satisfiedCount / principlesCount) * 100.0;
  }

  progressColor() {
    let percentage = this.satisfiedPercent();

    return classNames({
      danger: (percentage < 33.4),
      success: (percentage >= 66.7),
      warning: (percentage >= 33.4 && percentage < 66.7)
    });
  }

  render() {
    return (<ProgressBar now={this.satisfiedPercent()}
                         bsStyle={this.progressColor()} />);
  }
}
```

You can leave `StrengthMeter` for now - it is finished. Let's compare the produced HTML with the static HTML mockup. `"mwhkz1$` is used as a default state to compare:

```
<div class="panel panel-default">
  <div class="panel-body">
    <div class="progress">
      <div class="progress-bar progress-bar-success"
           style="width:100%"></div>
    </div>
    <h5>A good password is:</h5>
    <ul>
      <li class="text-success">
        <small>
          6&plus; characters
        </small>
      </li>
      <li class="text-success">
        <small>
          with at least one digit
        </small>
      </li>
      <li class="text-success">
        <small>
```

```
          with at least one special character
        </small>
      </li>
    </ul>
  </div>
</div>
```

```
<div class="panel panel-default" data-reactid="...">
  <div class="panel-body" data-reactid="...">
    <div min="0" max="100" class="progress" data-reactid="...">
      <div min="0" max="100" class="progress-bar progress-bar-success"
           role="progressbar" style="width:100%;" aria-valuenow="100"
           aria-valuemin="0" aria-valuemax="100" data-reactid="...">
      </div>
    </div>
    <h5 data-reactid="...">A good password is:</h5>
    <ul data-reactid="...">
      <li class="text-success" data-reactid="...">
        <small data-reactid="...">
          6+ characters
        </small>
      </li>
      <li class="text-success" data-reactid="...">
        <small data-reactid="...">
          with at least one digit
        </small>
      </li>
      <li class="text-success" data-reactid="...">
        <small data-reactid="...">
          with at least one special character
        </small>
      </li>
    </ul>
  </div>
</div>
```

Apart from the special `data-reactid` attributes added to be used by React internally, the syntax is *very similar*. React-Bootstrap progress bar component added an accessibility attributes that were absent in the static mockup. Very neat!

The last part of this feature is still to be done. It is a `PasswordField` component. Let's start with adding a static input.

Remember to `import` the `Input` component from the `react-bootstrap` package, like this:

```
import { Grid, Row, Col, Panel, ProgressBar, Input } from 'react-bootstrap';
```

Then, add a static input to the `PasswordField` component:

```
class PasswordField extends React.Component {
  render() {
    return (
      <Input
        type='password'
        label='Password'
        hasFeedback
      />
    );
  }
}
```

`hasFeedback` property takes care of adding feedback icons, like it is done in a mockup. When you set an appropriate `bsStyle` (which will be done later), a proper icon will show up on the right of the input.

You need to modify the password using an input. Since `PasswordField` is the owner of this data, both the data and a handler responsible for changing password must be passed to `PasswordField` component as properties.

Let's write a handler which will take the `password` as an argument and change `PasswordField` password state. Since it will be passed as a property, you must bind it to the `PasswordField` instance in the constructor. It can be done like this:

```
class PasswordInput extends React.Component {
  constructor(props) {
    super(props);
    this.state = { password: '' };

    this.changePassword = this.changePassword.bind(this);
  }

  changePassword(password) {
    this.setState({ password });
  }

  render() {
    let { goodPasswordPrinciples } = this.props;
    let { password } = this.state;
```

```
16
17      return (
18              <Grid>
19               <Row>
20                 <Col md={8}>
21                   <PasswordField password={password}
22                                  onPasswordChange={this.changePassword} />
23                 </Col>
24                 <Col md={4}>
25                   <StrengthMeter password={password}
26                                  principles={goodPasswordPrinciples} />
27                 </Col>
28               </Row>
29              </Grid>
30             );
31    }
32  }
```

As you can see there is `changePassword` method which takes a password and directly calling `setState`. This method is pushed down via the `onPasswordChange` property - an event handler on the lower level will call this method.

Speaking of which, let's define this handler in the `PasswordField` component:

```
1   class PasswordField extends React.Component {
2     constructor(props) {
3       super(props);
4       this.handlePasswordChange = this.handlePasswordChange.bind(this);
5     }
6
7     handlePasswordChange(ev) {
8       let { onPasswordChange } = this.props;
9       onPasswordChange(ev.target.value);
10    }
11
12    render() {
13      let { password } = this.props;
14
15      return (
16        <Input
17          type='password'
18          label='Password'
19          value={password}
```

```
20          onChange={this.handlePasswordChange}
21          hasFeedback
22        />
23      );
24    }
25  }
```

As you can see, there is a very thin wrapper defined to pass data from an event handler to the `onPasswordChange` callback. Generally you should avoid defining high-level API in terms of events - it's very easy to write a wrapper like this. A higher-level method which is defined in terms of password is a great help when comes to testing such component - both in the manual and the automatic way.

The last thing left to do is implementing logic of setting the proper "color state" of an input. This is a very similar logic that you defined before with progress bar color state. The easiest way to implement it is to copy this behaviour for now - with a very slight modification.

But before doing so your password principles must be passed as a property to the `PasswordField` component. I bet you already know how to do that - just pass it as a property of the `PasswordField` component rendered within the `PasswordInput` higher level component:

```
1   class PasswordInput extends React.Component {
2     constructor(props) {
3       super(props);
4       this.state = { password: '' };
5
6       this.changePassword = this.changePassword.bind(this);
7     }
8
9     changePassword(password) {
10      this.setState({ password });
11    }
12
13    render() {
14      let { goodPasswordPrinciples } = this.props;
15      let { password } = this.state;
16
17      return (
18              <Grid>
19               <Row>
20                 <Col md={8}>
21                   <PasswordField password={password}
22                                  onPasswordChange={this.changePassword}
```

```
                                principles={goodPasswordPrinciples} />
            </Col>
            <Col md={4}>
              <StrengthMeter password={password}
                                principles={goodPasswordPrinciples} />
            </Col>
          </Row>
        </Grid>
      );
  }
}
```

Since you got all the data needed, copying is the very simple step now:

```
class PasswordField extends React.Component {
  constructor(props) {
    super(props);
    this.handlePasswordChange = this.handlePasswordChange.bind(this);
  }

  handlePasswordChange(ev) {
    let { onPasswordChange } = this.props;
    onPasswordChange(ev.target.value);
  }

  satisfiedPercent() {
    let { principles, password } = this.props;

    let satisfiedCount = principles.map(p => p.predicate(password))
                                   .reduce((count, satisfied) =>
                                      count + (satisfied ? 1 : 0)
                                   , 0);

    let principlesCount = principles.length;

    return (satisfiedCount / principlesCount) * 100.0;
  }

  inputColor() {
    let percentage = this.satisfiedPercent();

    return classNames({
```

```
      error: (percentage < 33.4),
      success: (percentage >= 66.7),
      warning: (percentage >= 33.4 && percentage < 66.7)
    });
  }

  render() {
    let { password } = this.props;

    return (
      <Input
        type='password'
        label='Password'
        value={password}
        bsStyle={this.inputColor()}
        onChange={this.handlePasswordChange}
        hasFeedback
      />
    );
  }
}
```

There is a slight modification made while copying this logic. Apart from changing method name from `progressColor` to `inputColor`, one case of the color state was changed from `danger` to `error`. It is an inconsistency present in the React-Bootstrap API. The rest stays the same - you even use the same property to pass the color state (called `bsStyle`). `hasFeedback` takes care of displaying proper icons when the state changes.

That's it. The whole component is implemented. To be sure whether it is done correctly, let's compare the output produced by React with the static HTML mockup that has been presented before. Password used to render this 'snapshot' of the state is `"mwhkz1"` - so the "yellow" state.

```
<div class="container">
  <div class="row">
    <div class="col-md-8">
      <div class="form-group has-warning has-feedback">
        <label class="control-label"
               for="password-input">Password</label>
        <input type="password"
               class="form-control"
               id="password-input"
               value="mwhkz1"
               placeholder="Password" />
```

```
        <span class="glyphicon glyphicon-warning-sign form-control-feedback"
              aria-hidden="true"></span>
      </div>
    </div>
    <div class="col-md-4">
      <div class="panel panel-default">
        <div class="panel-body">
          <div class="progress">
            <div class="progress-bar progress-bar-warning"
                 style="width:66%"></div>
          </div>
          <h5>A good password is:</h5>
          <ul>
            <li class="text-success"><small>6&plus; characters</small></li>
            <li class="text-success"><small>with at least one digit</small></li>
            <li class="text-danger"><small>with at least one special character</\
small></li>
          </ul>
        </div>
      </div>
    </div>
  </div>
</div>
```

```
<div class="container" data-reactid="...">
  <div class="row" data-reactid="...">
    <div class="col-md-8" data-reactid="...">
      <div class="form-group has-feedback has-warning" data-reactid="...">
        <label class="control-label" data-reactid="...">
          <span data-reactid="...">Password</span>
        </label>
        <input type="password"
               label="Password"
               value=""
               class="form-control"
               data-reactid="...">
        <span class="glyphicon form-control-feedback glyphicon-warning-sign"
              data-reactid="..."></span>
      </div>
    </div>
    <div class="col-md-4" data-reactid="...">
      <div class="panel panel-default" data-reactid="...">
```

```
        <div class="panel-body" data-reactid="...">
          <div min="0" max="100" class="progress" data-reactid="...">
            <div min="0" max="100"
                 class="progress-bar progress-bar-warning"
                 role="progressbar"
                 style="width: 66.667%;"
                 aria-valuenow="66.66666666666666"
                 aria-valuemin="0"
                 aria-valuemax="100"
                 data-reactid="...">
            </div>
          </div>
          <h5 data-reactid="...">A good password is:</h5>
          <ul data-reactid="...">
            <li class="text-success" data-reactid="...">
              <small data-reactid="...">
                6+ characters
              </small>
            </li>
            <li class="text-success" data-reactid="...">
              <small data-reactid="...">with at least one digit</small>
            </li>
            <li class="text-danger" data-reactid="...">
              <small data-reactid="...">with at least one special character</sma\
ll>
            </li>
          </ul>
        </div>
      </div>
    </div>
  </div>
</div>
```

Apart from the `<span>` elements wrapping "text" nodes and accessibility improvements to progress bar that React-Bootstrap provides, the markup matches. That means you achieved your goal of implementing password strength meter logic in React. Great work!

## What's next?

It is a smell that logic of the color state is duplicated. It can be fixed by moving it to the higher-level component (`PasswordInput`) or by introducing a *higher-order component* which is a mixin

replacement for ECMAScript 2015 classes. You can read more about it here[6].

You may notice that `StrengthMeter` component is very generic - you can use it everywhere where your data can be checked against a set of predicates. That means you can change its name and re-use it in the other parts of your application.

The same can be done with a `PasswordField` component. In fact all that defines it is a *password strength meter* is defined in a top-level component. The rest can be re-used in many other contexts.

# Summary

As you can see, being backed by HTML mockup in React allows you to check your work while iterating. You can construct your mockup from the top, like it was done here. Alternatively you can start with *pasting* the code of the markup and changing properties to match React (like `class` becomes `className` and so on). Then, you split it into logical parts and add behaviour with the same starting point as you had with the static markup.

Password Strength Meter is a very handy widget to have - it is ready for usage with very small modifications - namely, adding a way to inform about the `password` state the rest of the world. You can do it by using *lifecycle methods* or by moving state even higher - and passing it as a property like it was done with `StrengthMeter` and `PasswordField`. Good luck!

[6] https://gist.github.com/sebmarkbage/ef0bf1f338a7182b6775