Example React.js Applications

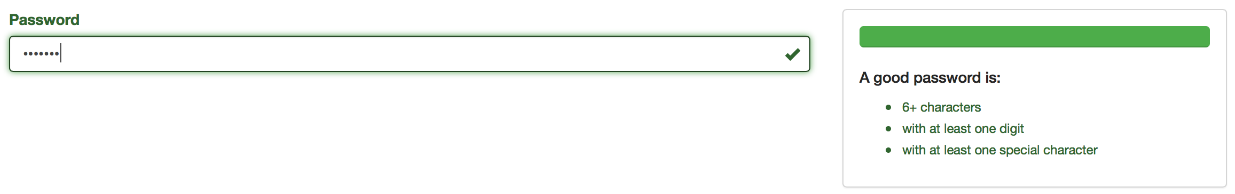
React Kung Fu

Example React.js Applications

React Kung Fu

Contents

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.

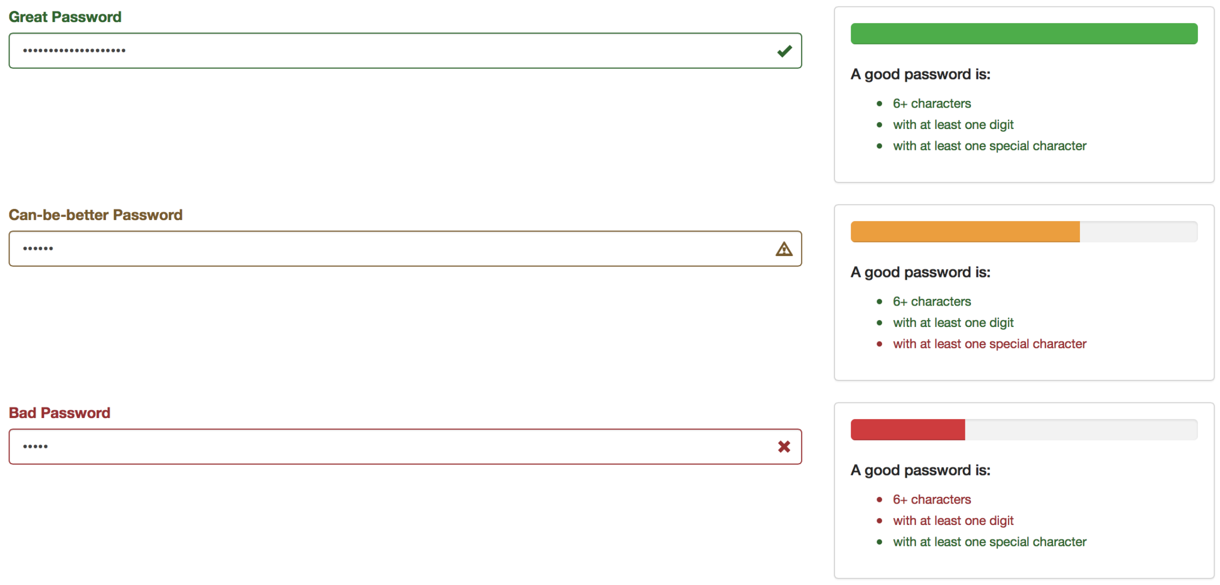
Repository

You can find code from this example in `repositories` directory attached to this book. Repository for this chapter is called `react-book-chapter-password-meter`.

[1]http://react-bootstrap.github.io
[2]https://www.npmjs.com/package/classnames

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 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>
          <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>
    <!-- Rest of states... -->
  </div>
</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:

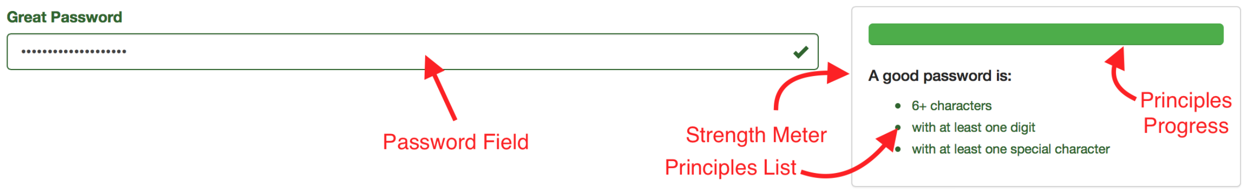
- Awful password - progress bar is red and an input is in “red” state (1/3 or less principles satisfied)
- Mediocre password - progress bar is yellow and an input is in “yellow” state (1/3 to 2/3 principles satisfied)
- Great password - progress bar is green and an input is in “green” state (2/3 or more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

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

    return (
      <ul>
        {principles.map(principle =>
        <li className={this.principleClass(principle)}>
          <small>
            {principle.label}
          </small>
        </li>
        )}
      </ul>
    );
  }
}
```

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:

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

Is equal to saying:

```
<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 `Principle-sProgress` 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;
  }

  render() {
    return (
      <Panel>
        <ProgressBar now={this.satisfiedPercent()} />
        <h5>A good password is:</h5>
        <PrinciplesList {...this.props} />
      </Panel>
    );
  }
}
```

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'.

```
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;
  }

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

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

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

```

  render() {
    return (
      <Panel>
        <ProgressBar now={this.satisfiedPercent()}
                     bsStyle={this.progressColor()} />
        <h5>A good password is:</h5>
        <PrinciplesList {...this.props} />
      </Panel>
    );
  }
}
```

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.

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

class PrinciplesProgress 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;
  }

  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;

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

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:

```
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);
  }

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

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

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:

```
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;

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

Todolist with React.js backed by Flux Architecture

This is a todo list app written with Flux principles in mind. You can watch a video[7] here.

React.js comes with a variety of tools and ideas behind it. There are many awesome tools that can work along with React really well. One of the most interesting ideas behind React is how the whole front-end application is structured.

Flux is an interesting approach to structurize front-end apps. It is a relatively simple idea, taking an inspiration from the CQRS[8] architecture. It specifies how your data flows thorough the whole front-end stack. You may heard about it if you are interested in React.

There are lots of libraries which help with building an app with Flux architecture. It may be hard to choose one. They come in different flavors and ideals in mind. Asynchronicity support, immutability as a core, ability to be isomorphic, or more functional approaches are usual 'ideals' behind them. I personally was daunted when I tried to choose one.

But an idea is the most important part of the architecture. I would like to show you step-by-step how to create a simple application, backed by a Flux architecture. As a bonus, I'll show you how to use Immutable.js[9] to improve performance of your React components.

App:

The app presented is a simple todo list. You can add task to it, as well as removing it. You can also see a list of existing tasks. You can see the full flow in a video at the top of this post[10].

Simplicity of this app would allow you to see a basic flow without distractions.

Why I should use Flux?

Flux with it's tooling allows us to set data flow in components in a more 'free' way. Consider a following components' tree:

[7]https://www.youtube.com/watch?v=gJCx1jcexxM
[8]http://martinfowler.com/bliki/CQRS.html
[9]http://facebook.github.io/immutable-js/
[10]https://www.youtube.com/watch?v=gJCx1jcexxM&feature=youtu.be

```
<Foo>
  <Bar>
    <Baz />
  <Bar>
  <Abc />
</Foo>
```

Without Flux, it's natural that most of state will go to the top-level `<Foo>` component. This is the only component you have an access to from the outside world. It's commonly seen that `<Foo>` component has some kind of `setData` function defined, just to take data from the outside world. Then it's passed down in the tree as properties. That extends the root component (`<Foo>`) responsibilities. Not cool, since such root components tend to grow in terms of code and complexity.

Flux introduces an idea of stores where you keep your data. Any component can register to such store. When data in the store updates, all registered components gets new version of data from the store.

So, without Flux we had:

```
<Foo> // setData(data) { this.setState(data); }
  <Bar dataBarNeeds={this.state.dataBarNeeds1}>
    <Baz dataBazNeeds={this.state.dataBazNeeds} />
  </Bar>
  <Abc dataAbcNeeds={this.state.dataAbcNeeds} />
</Foo>
```

And with Flux we have:

```
<Foo> // Listening for FooDataStore
  <Bar> // Listening for BarDataStore
    <Baz /> // Listening for BazDataStore
  </Bar>
  <Abc /> // Listening for AbcDataStore
</Foo>
```

You can listen to as many stores as you like in your component. Stores allow components in deeper levels of components tree to get data directly. It simplifies the flow of data in your application.

Since store data can change, there must be something which triggers the change. That's why Flux have the concept of actions. Stores subscribe to actions - everytime someone triggers the action, the registered store is notified about it.

The very interesting (and helpful) part of this architecture is that it creates **unidirectional data flow**. Components gets data from stores and emits actions, which updates stores. In vanilla implementation of Flux there is also a dispatcher - but I won't cover it in this blogpost.

Enough theory, get to the code!

Starting from scratch - creating an environment

This set of libraries will be used to create the app:

- Alt[11] - as a library supporting the Flux architecture
- React-Bootstrap[12] - as a set of ready components that can be used to provide a visually appealing user interface
- Immutable.js[13] - as a way to store data in an efficient way

First of all, a project must be created. Yeoman[14] can be used to quickly generate such project. To install it, use the following command:

```
npm install -g yo
```

Then, a suitable generator must be installed. For this project, generator-react-webpack[15] is used. Install it:

```
npm install -g generator-react-webpack
```

The next step would be to create a project:

```
yo react-webpack ReactFluxTodo # ReactFluxTodo is an app name.
```

React-router is not used, so do not enable it. You may be tempted to choose Alt from the list, but don't do it. It will get installed later in the newest version possible. Use LESS for your stylesheet engine - Bootstrap uses it, so it'll be easy to use it in this project. As an extension, choose the default.

The generated template uses Grunt[16] to automate development tasks. You may need to install it. To do so, issue the following command:

```
npm install -g grunt
```

OK. Unfortunately, this generator comes with a bit outdated version of React and Webpack. You have to update it manually. Open the `package.json` in an editor of your choice. Then, find a line like this:

[11]http://alt.js.org
[12]http://react-bootstrap.github.io
[13]https://facebook.github.io/immutable-js/
[14]http://yeoman.io
[15]https://github.com/newtriks/generator-react-webpack
[16]http://gruntjs.com

```
"react": "~0.12.2",
```

This line specifies that React should be installed in a version `0.12.x`, where `x` is higher than 2. Since in this project React 0.13.x will be used, all you need to do is to change it to:

```
"react": "~0.13.1",
```

The same treatment must be done with Webpack. Webpack[17] is a tool which compiles your application. It takes your source files and transforms it into an application that can be served by browsers. It supports transforming all files - not only JavaScript files, but also CSS files, fonts and so on. It comes with a server that serves such application during development.

There's a bug in a bundled version of Webpack that 'breaks' Alt when updating the source code of React components. This bug is fixed in a newer versions of Webpack. You can read more about it here[18].

To update Webpack, find it in a `package.json` file:

```
"webpack": "~1.4.3",
```

And change it to:

```
"webpack": "~1.10.1",
```

Then run:

```
npm install
```

That's it. You can now run and check the default outcome that was generated by a Yeoman:

```
grunt serve
```

A browser should pop up with the current version of the site.

Installing dependencies

Before any line of code is written, Webpack must be configured and dependencies installed to use it. Let's start with dependencies. React is already installed, so there is nothing to do with it. Alt is not installed. To install it, npm will be used with `--save` flag. This flag adds Alt as a runtime dependency to this application. It indicates that without Alt, the app won't run.

[17]http://webpack.github.io
[18]https://github.com/goatslacker/alt/issues/29

```
npm install --save alt
```

Now React-Bootstrap must be installed. As can be read in docs, you need to supply a CSS file from Twitter Bootstrap by yourself. Fortunately, an official bootstrap npm package has it. To install it, use the following command:

```
npm install --save react-bootstrap bootstrap
```

Last, but not least, Immutable.js must be installed. It can be done the same way that other dependencies are installed:

```
npm install --save immutable
```

Since tasks on the todo list will have an unique identifier attached to it, such identifiers must be generated somehow. `node-uuid` library can be used here:

```
npm install --save node-uuid
```

That's it. All dependencies for this project are installed now!

Configuring Webpack

Webpack is pretty well configured by the Yeoman generator. Unfortunately, you need to tweak it before it can be used with React-Bootstrap. Since Bootstrap comes with variety of font formats (Glyphicons), webpack can't figure in this configuration what to do with some formats. Fortunately, fix is quick and simple.

You need to edit `webpack.config.js` and `webpack.dist.config.js` files, where Webpack config is stored. Webpack uses *loaders* to know what to do with files it have to serve. To serve static files like fonts or images, it uses the `url-loader`. It is defined here:

```
{
  test: /\.(png|jpg|woff|woff2)$/,
  loader: 'url-loader?limit=8192'
}
```

Bootstrap comes with Glyphicon font in many formats - like `eot`, `ttf` or `svg`. In current configuration, webpack will bail out because it knows nothing about such extensions. Let's change this:

```
{
  test: /\.(png|jpg|eot|ttf|svg|woff|woff2)$/,
  loader: 'url-loader?limit=8192'
}
```

Now Webpack will know what to do with all font formats that come with Bootstrap. **Remember to do such change in both files!**.

Since you are there, there are two changes that can be done to improve workflow in this project.

- Right now, the starting point of your app is the React component. Usually you want to separate initialization logic from declaration of your code. In this project, this starting point will be changed.
- In this project stores will be in a `src/stores`, actions in `src/actions` and an Alt instance (described later) in `src/lib`. To avoid referencing such files via relative paths (like `../something`), a top-level aliases will be provided (so you can reference store `A` in the whole project as `stores/A`).

Changing the starting point of an application:

This is configured via an `entry` property within a webpack config. Remember to do this change in both Webpack configuration files (`webpack.config.js` and `webpack.dist.config.js`).

Find the `entry` property in `webpack.config.js`. It is defined as a list:

```
entry: [
  'webpack/hot/only-dev-server',
  './src/components/ReactFluxTodo.js'
],
```

First entry is an implementation detail of a `react-hot-loader`. This utility comes bundled with this generated project and is responsible for hot reloading your code when it changes. You don't need to refresh your browser when you change source files thanks to that. You are interested in a second entry. Change it to `'./src/TodoList.js`:

```
entry: [
  'webpack/hot/only-dev-server',
  './src/TodoList.js'
],
```

In `webpack.dist.config.js` there is only one entry (`'./src/components/ReactFluxTodo.js'`). Just change it to `'./src/TodoList.js'`.

This way webpack will start it's *dependency tree* from this file. Create it and paste the following content:

`src/TodoList.js`:

```
import React from 'react/addons';
import ReactFluxTodo from 'components/ReactFluxTodo';

React.render(<ReactFluxTodo />, document.getElementById('content'));
```

Then, remove the line:

```
React.render(<ReactFluxTodo />, document.getElementById('content')); // jshint i\
gnore:line
```

From `src/components/ReactFluxTodo.js`. What you did now is importing React library and the component code and render it. Your code should still work, but now you have extracted initialization out of the component file itself. Requiring it now causes no side effects.

Adding aliases for a todo list project

Webpack stores it's require aliases in `alias` property within its config in a `resolve` subsection. You may see that there are some aliases already defined:

```
alias: {
  'styles': __dirname + '/src/styles',
  'mixins': __dirname + '/src/mixins',
  'components': __dirname + '/src/components/'
}
```

Just add aliases that are needed in this project in a similar fashion:

```
alias: {
  'styles': __dirname + '/src/styles',
  'mixins': __dirname + '/src/mixins',
  'components': __dirname + '/src/components/',
  'stores': __dirname + '/src/stores',
  'actions': __dirname + '/src/actions',
  'lib': __dirname + '/src/lib'
}
```

Remember to do changes in both `webpack.config.js` and `webpack.dist.config.js` files!

Starting with actions:

To use Alt, you need to create an instance of it. You do so to create some kind of 'namespace' for all actions and stores you will create later. Usually there is one instance for each application you create, but with complex scenarios you may need more than one. With two instances you have two sets of stores and actions and no way to interact between them. It may be helpful to create a strong boundaries in a really big apps.

To do so, create a file `src/lib/AltInstance.js` with the following content:

```
import Alt from 'alt';
export default new Alt();
```

The code is rather straightforward - an Alt object prototype is imported and a new instance of it is created.

Now you can start defining your actions. I find such start a preferred way to start working with a Flux application. This makes me focus on features, not on the 'infrastructure' code around it.

In this project our features that user calls by itself are: creating a new task and removing existing ones. Let's model this in actions!

In `src/actions/TodoList.js` put the following code:

```
import UUID         from 'node-uuid';
import Immutable    from 'immutable';
import AltInstance from 'lib/AltInstance';

class TodoListActions {
  addTask(content) { this.dispatch(Immutable.fromJS({ id: UUID.v4(), content }))\
; }
  removeTask(taskID) { this.dispatch(taskID); }
}

export default AltInstance.createActions(TodoListActions);
```

Notice that set of actions is defined as a pure JavaScript class. Then, it is 'decorated' with `createActions` method from Alt instance to allow subscribing to it and dispatching data. Action methods uses `this.dispatch` to send data to subscribed stores. You can perform a simple validations here.

Todo List store:

There must be a store which consumes actions you defined and keep the data attached to it. Such data will be stored as an immutable list. Immutability is great, because when a React component update, it does not need to re-check all elements of such list to check whether it changed or not. It improves performance of such component.

An example implementation of store can be done in the following way. Put this code in `src/stores/TodoList.js`:

```
import ImmutableStore from 'alt/utils/ImmutableUtil';
import { List }       from 'immutable';

import AltInstance    from 'lib/AltInstance';
import Actions        from 'actions/TodoList';

class TodoListStore {
  constructor() {
    let { addTask, removeTask } = Actions;

    this.bindListeners({
      add: addTask,
      remove: removeTask
    });

    this.state = List();
  }

  add(task) {
    return this.setState(this.state.push(task));
  }

  remove(taskID) {
    let taskIndex = this.state.findIndex((task) => task.get('id') === taskID);

    return taskIndex !== (-1) ? this.setState(this.state.delete(taskIndex)) :
                                this.state;
  }
}

export default AltInstance.createStore(ImmutableStore(TodoListStore));
```

The most important thing happens in a `bindListeners` method. This is how you subscribe to actions in a store. You delegate an action to a method which handles it. Such method will receive as arguments everything that has been sent by a `dispatch` within the action.

The second important part is the `setState`. Since components subscribe to stores, everytime `setState` is called they will get an update.

Another interesting thing is how this store is created. It is created in a similar way to actions - you create a pure JavaScript object and pass it to an alt instance method to enhance it with subscribing. But with this store it is *wrapped* in a `ImmutableStore` function. Why is that?

By default, Alt sets state by mutating the store itself. Since here immutable data structure is used, it can't be done. Every modification of a `List()` used here creates a new object - thus a new reference. In Alt you can modify how `setState` and many other methods work. `ImmutableStore` function modifies the `setState` in a way it assigns a new value of the state, not mutating it (using Object.assign[19]). This is a way how Alt provides optional features - you have a full control whether you want to use it or not.

Todo List component

Since actions and a store backing todo list are already done, React components can be created to actually *render* something.

First of all, provide a `TodoListTask` component to get a simple view of one task. Put it in `src/components/TodoListTask.js`:

[19]https://developer.mozilla.org/pl/docs/Web/JavaScript/Reference/Global_Objects/Object/assign

```
import React                               from 'react/addons';
import { ListGroupItem, Glyphicon, Button } from 'react-bootstrap';

import TodoListActions                     from 'actions/TodoList';

class TodoListTask extends React.Component {
  constructor(props) {
    super(props);
    this.removeTask = this.removeTask.bind(this);
  }

  removeTask() {
    TodoListActions.removeTask(this.props.task.get('id'));
  }

  render() {
    let { task } = this.props;
    return (<ListGroupItem>
              {task.get('content')}
              <Button bsSize="xsmall" bsStyle="danger" className="pull-right" on\
Click={this.removeTask}>
                <Glyphicon glyph="remove" />
              </Button>
            </ListGroupItem>);
  }
}

export default TodoListTask;
```

In this component there is an action called: `removeTask`. So this button will trigger a store behaviour which is binded to `removeTask` action. In this case it is a `remove` method. Now, one-directional data flow is full. It is an example of how things are done in Flux!

A list of tasks have to be listed, though. That means a new component - `TodoList` must be created. Put this code in `src/components/TodoList.js`:

```
import React                      from 'react/addons';
import { Grid, Row, ListGroup } from 'react-bootstrap';
import TodoListStore            from 'stores/TodoList';
import TodoListTask             from 'components/TodoListTask';

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

    let { shouldComponentUpdate } = React.addons.PureRenderMixin;

    this.shouldComponentUpdate    = shouldComponentUpdate.bind(this);
    this.state                    = { tasks: TodoListStore.getState() };
    this.listChanged              = this.listChanged.bind(this);
  }

  componentDidMount()    { TodoListStore.listen(this.listChanged); }
  componentWillUnmount() { TodoListStore.unlisten(this.listChanged); }

  listChanged(taskList)  { this.setState({ tasks: taskList }); }

  render() {
    let {tasks} = this.state;

    return (
      <Grid>
        <Row fluid={true}>
          <h1>Tasks:</h1>
          <ListGroup>
            {tasks.map(task =>
              <TodoListTask key={task.get('id')} task={task} />
             ).toJS()}
          </ListGroup>
        </Row>
      </Grid>
    );
  }
}

export default TodoList;
```

Notice how `TodoList` component connects to the store. First of all, in the component's contructor there is a call to `getState`. It gets the current state stored in a `TodoListStore`. Then, when a component mounts there is a `listen` method of a store used. From now every call to `setState` within a store will trigger the `listChanged` method of a component.

The last part of integration is to specify what happens when component gets unmounted. Otherwise you'd get errors, because even unmounted stores would call `setState`, breaking React invariants. It is done within the `componentWillUnmount` lifecycle method.

Another interesting thing happens within a `render` method itself. While React can iterate through everything that has an iterator interface (and Immutable.js `List` has an Iterator interface), it would be a warning, since deprecated method would get called. To overcome this problem, after mapping tasks to `TodoListTask` components it is transformed into a simple mutable list using `toJS` method.

Also, there is a `PureRenderMixin` used. It makes the component more performant by adding an assumption that your data is immutable. This is a case with an approach presented in this app.

Since this component should be displayed, it is a good moment to clear out the `index.html` file a little. Replace:

```html
<div id="content">
  <h1>If you can see this, something is broken (or JS is not enabled)!!.</h1>
</div>
```

With:

```html
<div id="todo-list">
  <p>Sorry, but this todo list needs JavaScript enabled to work.</p>
</div>
```

Now id of the `<div>` within a component rendered will make a little more sense.

Then, change your `src/TodoList.js` to:

```js
'use strict';

require('bootstrap/less/bootstrap.less');

import React    from 'react/addons';
import TodoList from 'components/TodoList';

React.render(<TodoList />, document.getElementById('todo-list'));
```

You can remove `src/components/ReactFluxTodo.js` since it is not needed anymore. Notice that there is a LESS file required - Webpack will handle compilation and serving this LESS file for you!

Yet another full Flux cycle - adding a new task

So now the app knows how to display a list of tasks and how to remove one task from it. What it lacks is an ability to add a new task. Pattern that Flux provides - defining Actions, connecting them to Stores, listening for Stores in Components and emitting Actions from them is repeatable thorough the app designed in a Flux architecture. That's why this feature starts in a similar way.

In the adding a new task form there will be two actions - clearing the form (when the form finishes its job) and handling input change. Let's model it. Put this code in `src/actions/AddNewTaskForm.js`:

```
import AltInstance from 'lib/AltInstance';

class AddNewTaskFormActions {
  changeContent(content) {
    this.dispatch(content);
  }

  clearForm() {
    this.dispatch();
  }
}

/* If your actions are as simple as just dispatching passed values, you can use \
a slightly different (and more concise) API for such use case:
 * export default alt.generateActions('changeContent', 'clearForm');
 */

export default AltInstance.createActions(AddNewTaskFormActions);
```

In the comment there is a handy syntactic sugar for creating very simple actions like this. For consistency it is omitted in this example.

Now, to the store. What store should do? I think working with form validations on the store side is very Fluxy way to design forms. That's why this logic is there. Put this code in `src/stores/AddNew-TaskForm.js`:

```
import AltInstance from 'lib/AltInstance';
import Actions     from 'actions/AddNewTaskForm';

class AddNewTaskFormStore {
  constructor() {
    this.validationError = '';
    this.content = '';
    this.submittable = false;

    let { changeContent, clearForm } = Actions;
    this.bindListeners({ changeContent, clearForm });
  }

  changeContent(newContent) {
    let validationError = this.validate(newContent),
        submittable     = validationError.length === 0;

    this.setState({ validationError,
                    content: newContent,
                    submittable });
  }

  clearForm() { this.setState({ validationError: '',
                                content: '',
                                submittable: false }); }

  validate(newContent) {
    return (newContent.length > 3) ? '' : 'Task content have to be longer than 3\
 characters.';
  }
}

export default AltInstance.createStore(AddNewTaskFormStore);
```

Notice that there is no `ImmutableStore` function used at all. Since only primitive values are used in this store, it is not needed. In this approach store data is defined simply as a fields of the store. In an `ImmutableStore` approach your data must be stored within a `state` field of your store.

Then a component must be created. Put this in `src/components/AddNewTaskForm.js`:

```
import React                   from 'react/addons';
import { Input, Button }       from 'react-bootstrap';

import AddNewTaskFormActions from 'actions/AddNewTaskForm';
import TodoListActions       from 'actions/TodoList';

import AddNewTaskFormStore   from 'stores/AddNewTaskForm';

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

    let { shouldComponentUpdate } = React.addons.PureRenderMixin;

    this.state = AddNewTaskFormStore.getState();

    this.shouldComponentUpdate = shouldComponentUpdate.bind(this);
    this.formChanged = this.formChanged.bind(this);
    this.validationClass = this.validationClass.bind(this);
    this.submit = this.submit.bind(this);
  }

  componentDidMount()    { AddNewTaskFormStore.listen(this.formChanged); }
  componentWillUnmount() { AddNewTaskFormStore.unlisten(this.formChanged); }

  formChanged(formState) { this.setState(formState); }
  changeContent(ev)      { AddNewTaskFormActions.changeContent(ev.target.value);\
 }
  submit(ev) {
    ev.preventDefault();
    if(!this.state.submittable) { return; }

    TodoListActions.addTask(this.state.content);
    AddNewTaskFormActions.clearForm();
  }

  validationClass() {
    return {
      true: 'error',
      false: undefined
    }[!!this.state.validationError.length];
  }

  render() {
    return (
      <form onSubmit={this.submit}>
        <Input
          key="taskContent"
          type="text"
          value={this.state.content}
          placeholder="4+ characters..."
          label="Enter content:"
          bsStyle={this.validationClass()}
          help={this.state.validationError}
          hasFeedback
          onChange={this.changeContent} />
        <Button key="submitButton" type="submit"
                bsStyle="primary" disabled={!this.state.submittable}>Submit</But\
ton>
      </form>
    );
  }
}

export default AddNewTaskForm;
```

React Bootstrap input component have a built-in way to work with validations. That's a great help here - it makes this component quite simple. There are helper methods which 'transforms' validation data to CSS classes and so on. There is an interesting approach with hash syntax presented there, which helps avoiding `if`'s in code, making the component more 'declarative'. The form gets its state straight from the store - there is no internal state whatsoever.

To finish this feature, newly created component must be added to a top-level component. After `</ListGroup>` in a `TodoList` component, put the following code:

```
<h2>Add new task:</h2>
<AddNewTaskForm />
```

Also, new component must be imported. In `src/components/TodoList.js`, after:

```
import TodoListTask              from 'components/TodoListTask';
```

Add the following code:

```
import AddNewTaskForm           from 'components/AddNewTaskForm';
```

That’s it. The Todo list app is complete!

Repository:

There is a GitHub repository which contains the example described above. It is made in around 25∼ commits. If you want to see how it was created in a step-by-step manner with some rationale behind decisions, check out the React-Flux-Alt-Immutable-TodoList repo[20].

Read more:

- Documentation of React-Bootstrap[21]
- Documentation of Alt[22]
- Documentation of Immutable.js[23]
- Flux architecture explained by Facebook developers[24]

[20] https://github.com/arkency/react_flux_alt_immutable_todolist
[21] http://react-bootstrap.github.io
[22] http://alt.js.org
[23] https://facebook.github.io/immutable-js/
[24] https://www.youtube.com/watch?list=PLb0IAmt7-GS188xDYE-u1ShQmFFGbrk0v&t=621&v=nYkdrAPrdcw

Fair Pizza App written in React Native

When Facebook announced React Native I was amused. It is my chance to start developing mobile applications. I already know React and I can write most of my code using it.

If you haven't seen @Vjeux[25] talking about React Native on React.js Conf, you should definitely check it out[26] before.

I want to share my path in making simple application for iPhone. I believe that experience should be enough to start creating bigger apps.

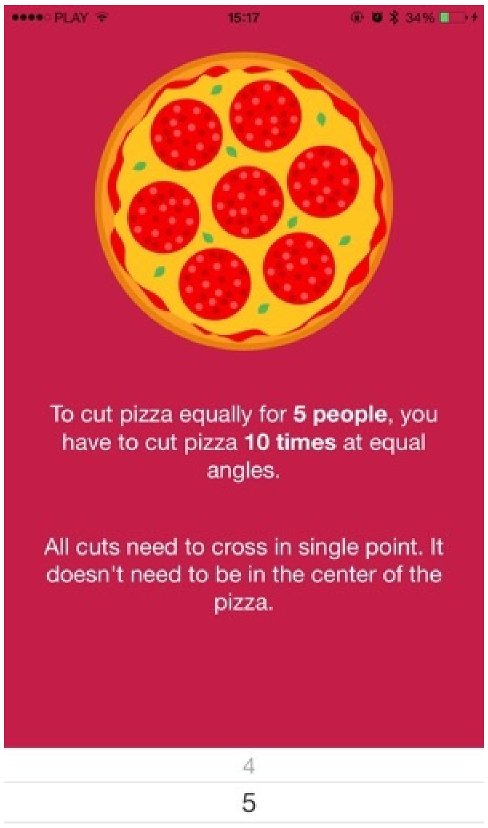
The Fair Pizza app

I wanted to start with something simple. Some time ago I came onto the Pizza theorem[27]. The idea is to make an app that would give you an instruction how to cut pizza equally for 2-5 people.

It's a simple application with single screen and single number picker. There are more features to be implement, but they will be a topic for the future.

Here's the result:

[25] https://twitter.com/Vjeux
[26] https://www.youtube.com/watch?v=7rDsRXj9-cU
[27] https://en.wikipedia.org/wiki/Pizza_theorem



To cut pizza equally for **5 people**, you have to cut pizza **10 times** at equal angles.

All cuts need to cross in single point. It doesn't need to be in the center of the pizza.

Fair Pizza iOS App

The code of the application is open-source. You can see it here[28].

Installation

Let's follow official React Native starting guide[29] here.

React Native let us develop apps only for iOS (Android version is comming soon). You will need computer with OS X and XCode in version >= 6.3. You would also need to have Homebrew[30] to install all the dependencies.

If you have Node.js installed, you will need to unlink it. Don't worry, it's compatible with Node and NPM.

```
brew unlink node
```

Now install io.js:

[28]https://github.com/arkency/fair_pizza/tree/01d6cece1970b3342e81de6ccb862eeaefe4ab13
[29]https://facebook.github.io/react-native/docs/getting-started.html
[30]http://brew.sh/

```
brew install iojs watchman && brew link iojs â€"force
```

Now, we can just install React Native npm package:

```
npm install -g react-native-cli
```

(optional) Facebook team recommends installing watchman[31]:

```
brew install watchman
```

Project setup

Project creation couldn't be simpler. Just enter this command:

```
react-native init OurProjectName
```

After that, you will find entire XCode project in `OurProjectName` directory.

Basics of React Native

File `index.ios.js` is a starting point of React Native application. Babel transpiles the code, so you will have enabled most of the ES6 features. You can read about available transformations in React Native documentation[32].

You can use CommonJS modules, so it's easy to split your app into separate files.

Ok, let's dive into Fair Pizza app code.

Building application window

As mentioned before, all React Native applications starts in `index.ios.js` file. Let's see how this file looks like in Fair Pizza:

[31]https://facebook.github.io/watchman/
[32]https://facebook.github.io/react-native/docs/javascript-environment.html#content

```
var React = require('react-native');
var {
  AppRegistry,
  StyleSheet,
  Text,
  View,
} = React;

var Pizza               = require('./Pizza');
var CuttingInstruction = require('./CuttingInstruction');
var PeopleCountPicker  = require('./PeopleCountPicker');

class FairPizza extends React.Component {
  constructor(props) {
    super(props);
    this.state = { peopleCount: 4 };
    this.setPeopleCount = this.setPeopleCount.bind(this);
    this.cuttingEdges = this.cuttingEdges.bind(this);
  }

  setPeopleCount(peopleCount) {
    this.setState({ peopleCount: peopleCount });
  }

  cuttingEdges() {
    return this.state.peopleCount * 2;
  }

  render() {
    return (
      <View style={styles.app}>
        {this.pizza()}
        {this.cuttingInstruction()}
        {this.peoplePicker()}
      </View>
    );
  }

  pizza() {
    return (
      <View style={styles.pizza}>
        <Pizza cuttingEdges={this.cuttingEdges()} />
      </View>
    );
  }

  cuttingInstruction() {
    return (
      <View style={styles.cuttingInstruction}>
        <CuttingInstruction
          cuttingEdges={this.cuttingEdges()}
          peopleCount={this.state.peopleCount}
        />
      </View>
    );
  }

  peoplePicker() {
    return (
      <View style={styles.peoplePickerWrapper}>
        <View style={styles.peoplePicker}>
          <PeopleCountPicker
            peopleCount={this.state.peopleCount}
            setPeopleCount={this.setPeopleCount}
          />
        </View>
      </View>
    );
  }
}

var styles = StyleSheet.create({
  app: {
    flex: 1,
    backgroundColor: '#C41D47',
  },
  pizza: {
    alignSelf: 'center',
    marginTop: 40
  },
  cuttingInstruction: {
    alignSelf: 'center',
    padding: 20,
    paddingTop: 0
  },
  peoplePickerWrapper: {
    flex: 1,
    justifyContent: 'flex-end',
  },
  peoplePicker: {
    height: 95,
    overflow: 'hidden',
    justifyContent: 'flex-end',
  }
});

AppRegistry.registerComponent('FairPizza', () => FairPizza);
```

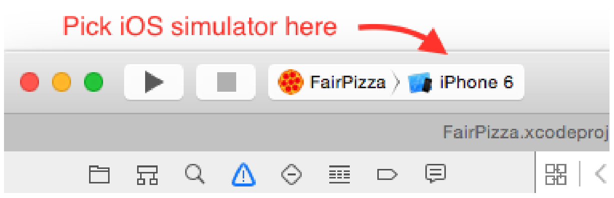
If you are familiar with React, you won't have any troubles with reading this code. I splitted the main application window into 3 separate components:

- Pizza image
- Cutting instruction
- People count picker

The next thing you could have noticed is `styles` variable. You can pass styles just as a simple hashes or use a `StyleSheet`. They are quite like a styles you may know from web version of React.

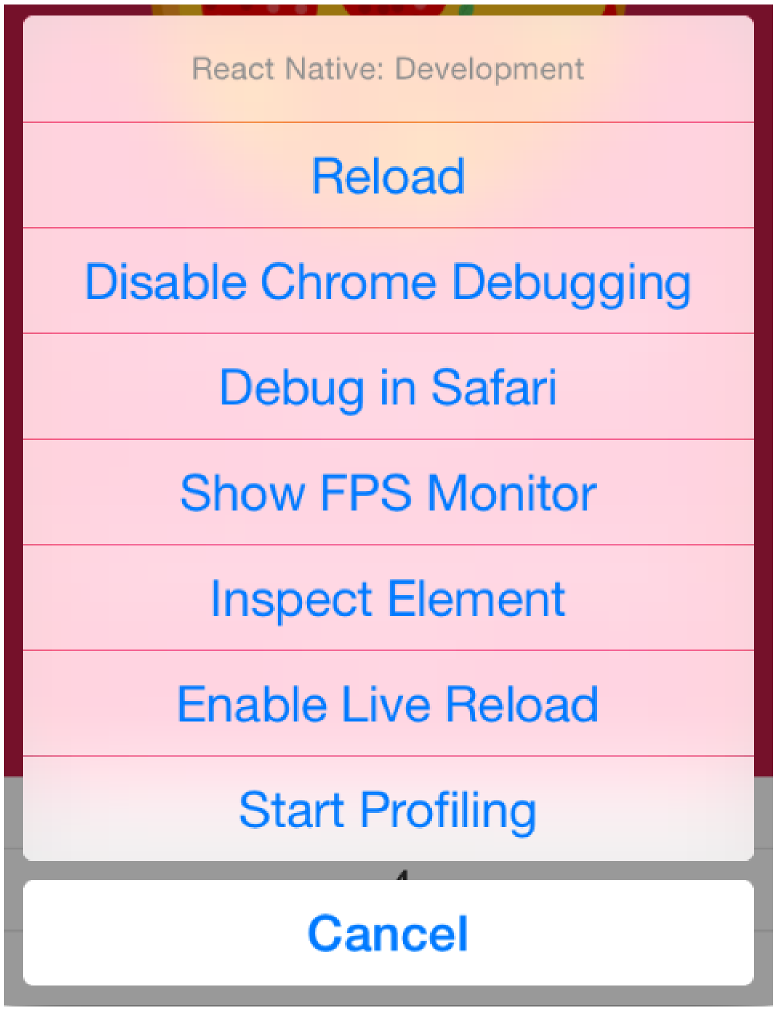
The last thing we need to do is to register component - last line handles that.

You can run the app without any iOS device - you can just use simulator built-in XCode.



JavaScript code debugging

When you open-up the application in iOS simulator, you can launch React Native Debugger in your browser. Just press `cmd-d` key combination in simulator window and pick browser for debugging.



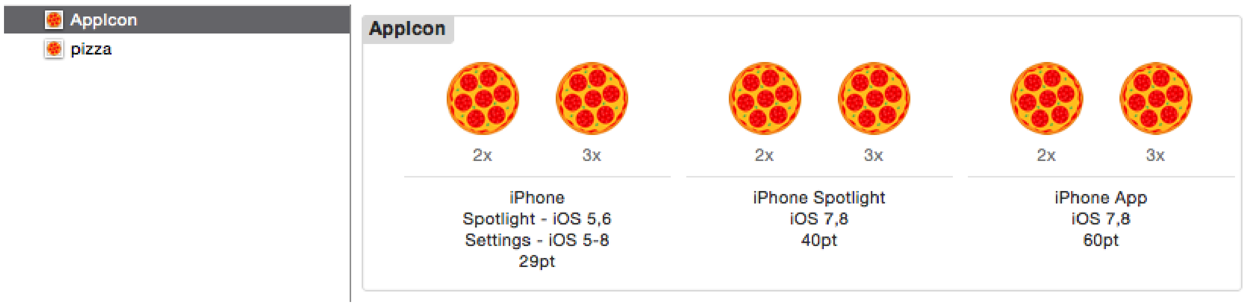
Problems

If you had some experience with React, the code of whole application should be fairly understandable for you. It may seem simple, but I came onto some issues during development.

Adding images to XCassets

To render pizza image from image file, I had to store the image somewhere in the application. I followed the official guide[33]. The app didn't want to compile until I have declared `AppIcon` (with all required sizes!). I found this website[34] to convert my image for all required size.

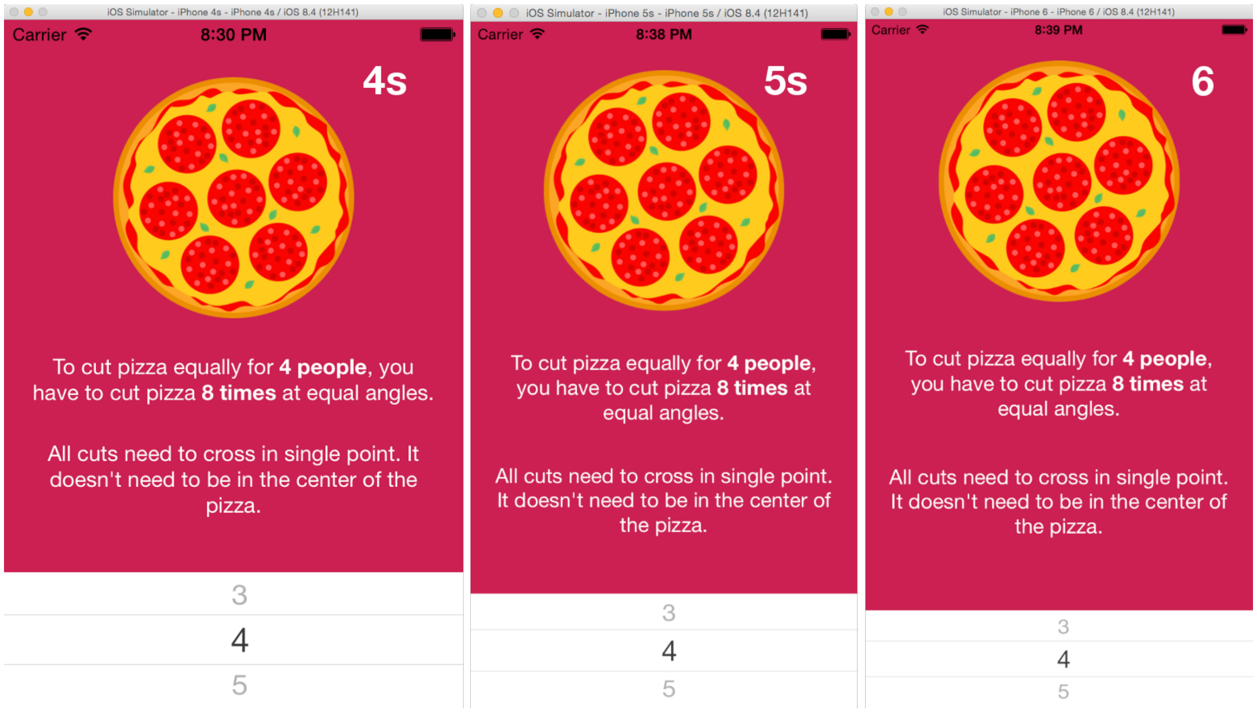
[33]https://facebook.github.io/react-native/docs/image.html#content
[34]http://makeappicon.com/



Styling app for different screens

Apple currently support phones with four different screen types. All of them can have different aspect ratio, different size, different resolution. That's a bit problematic for Fair Pizza app - it is supposed to be single-screen.

React Native has a `PixelRatio` class that make it easier to develop look of apps for screens with different pixel density. It's not very useful in this case, but code[35] of this class pointed out to class[36] called `Dimensions`. It's not mentioned in web documentation, but it is well commented.


iOS Simulator - iPhone 4s - iPhone 4s / iOS 8.4 (12H141)
Carrier
8:30 PM
4s
iOS Simulator - iPhone 5s - iPhone 5s / iOS 8.4 (12H141)
Carrier
8:38 PM
5s
iOS Simulator - iPhone 6 - iPhone 6 / iOS 8.4 (12H141)
Carrier
8:39 PM
6
3
4
5

To cut pizza equally for **4 people**, you have to cut pizza **8 times** at equal angles.

All cuts need to cross in single point. It doesn't need to be in the center of the pizza.

`Dimensions` class let me get real width and height of device with running app.

[35]https://github.com/facebook/react-native/blob/master/Libraries/Utilities/PixelRatio.js
[36]https://github.com/facebook/react-native/blob/master/Libraries/Utilities/Dimensions.js

```
let isLegacy = () => {
  let window = Dimensions.get('window');
  return (window.height * window.scale) < 1100;
}
let isIphone5 = () => {
  let window = Dimensions.get('window');
  return (window.height * window.scale) < 1300;
}
```

Running the app on an iPhone

App is ready, it works fine on XCode simulator. It's high time to run it on the real device. We just need to plug the phone to the computer and start the application directly on device. It will require us to log in with our AppleID.

By default, JavaScript code will be loaded from Node server that works on development machine. If you want to run the code on your device, you need to pass ip address of the server explicitly in `AppDelegate.m` file.

```
jsCodeLocation = [NSURL URLWithString:@"http://0.0.0.0:8081/index.ios.bundle"];
```

In this case, the code will be loaded from ip address `0.0.0.0` (localhost). You need change that value to the IP address of your computer. You just need to take care of both Mac and iPhone being connected to the same network.

That way of serving JavaScript is obviously not very convenient. It requires connection to React Native Node server. We can serve this code directly from app. You just need to change the value of `jsCodeLocation` in `AppDelegate.m`

```
jsCodeLocation = [[NSBundle mainBundle] URLForResource:@"main" withExtension:@"j\
sbundle"];
```

And now, our application is ready to be run independently on the phone.

What's next?

There are couple of features I would like to add to this application later:

- Drawing cutting lines on pizza - I want to learn how to draw lines on my pizza image. I already found the library[37] that may be useful for this.
- More elegant solution for styling different types of screens
- Rotate animation for pizza

[37]https://github.com/lwansbrough/react-native-canvas