PREV
8. DOM Manipulation

NEXT
10. Animations

Chapter 9. Forms

Forms are an essential part of any application that requires even modest input from their users. Traditionally in one page apps, forms are hard to get "right," since they are littered with changing state from the user. Managing this state is complex and often buggy. React.js helps you manage state in your applications, and this also extends to your forms.

By now you will have realized that predictability and testability are central aspects of React.js components. Given the same props and state, any React.js component should render exactly the same every time. Forms are no exception.

There are two types of form components in React.js: **Controlled** and **Uncontrolled**. In this chapter you will learn what the differences between **Controlled** and **Uncontrolled** are and under what circumstances you will want to one over the other.

This chapter also covers:

- How to use form events with React.js.
- Controlling data input with Controlled form components.
- How React.js changes the interface of some form components.
- The importance of naming your form components in React.js.
- Dealing with multiple Controlled form components.
- Creating custom reusable form components.
- Using AutoFocus with React.js.
- Tips for building usable applications.

The example application, Survey Builder, utilises forms in a non standard way as they are being dynamically generated based on the survey criteria. As such, the examples in this chapter are

generic in nature to help convey the concepts utilised in the example application and help you to gain a stronger knowledge of how to work with forms in React.

Uncontrolled Components

In most non trivial forms, you will **not** want to user **Uncontrolled** components. They are however very useful to help understand **Controlled** components. **Uncontrolled** are an anti pattern for how most other components are constructed in React.js.

In HTML, forms work differently to React.js components. When an HTML <input/> is given a value, the <input/> can then mutate the value of the <input/>. This is where the name comes from, since the form component's value is **Uncontrolled** by your React.js component.

In React.js, this behaviour is like setting the `defaultValue` of the <input/>.

To set the default value of an <input/> in React.js you use the `defaultValue` property.

```
//http://jsfiddle.net/5g6n0102/
var MyForm = React.createClass({
  render: function () {
    return <input
      type="text"
      defaultValue="Hello World!" />;
  }
});
```

The example above is referred to as an **Uncontrolled** component. The `value` is not being set by the parent component, allowing the <input/> to control it's own value.

An **Uncontrolled** component is not very useful unless you can access the value. In order to access the value, you need to add a `ref` to your <input/> then access the DOMNode value.

A `ref` is a special non DOM attribute used to identify a component from within `this` context. All refs in a component are added to `this.refs` for easy access.

Let's also add the <input/> to a form and read the value on submit.

```
//http://jsfiddle.net/kvh031wc/
var MyForm = React.createClass({
  submitHandler: function (event) {
    event.preventDefault();
    //access the input by it's ref
    var helloTo = this.refs.helloTo.getDOMNode().value;
    alert(helloTo);
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
        <input
          ref="helloTo"
```

```
        <br />
        <button type="submit">Speak</button>
      </form>;
  }
});
```

Uncontrolled components are fantastic for basic forms that don't require any special validation or input controls.

Controlled Components

Controlled components follow the same pattern for other React.js components. The state of the form component is **Controlled** by your React.js component, with it's value being stored in your React.js components state.

If you want more control over your form components, you will want to use a **Controlled** component.

A **Controlled** component is where the parent component set's the value of the input.

Let's convert the example above to use a **Controlled** component.

```
//http://jsfiddle.net/tc6vkzsz/
var MyForm = React.createClass({
  getInitialState: function () {
    return {
      helloTo: "Hello World!"
    };
  },
  handleChange: function (event) {
    this.setState({
      helloTo: event.target.value
    });
  },
  submitHandler: function (event) {
    event.preventDefault();
    alert(this.state.helloTo);
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
        <input
          type="text"
          value={this.state.helloTo}
          onChange={this.handleChange} />
        <br />
        <button type="submit">Speak</button>
      </form>;
  }
});
```

The major change here is that the value of the <input/> is now being stored in the state of the parent component. As a result, the data flow is now clearly defined.

- getInitialState now sets the defaultValue
- the <input/> value is set during render
- onChange of the <input/> value the change handler is invoked
- the change handler updates the state
- the <input/> value is updated during render

This is a lot more code than the **Uncontrolled** version. However, this allows you to control the data flow and alter the state as the data is being entered.

Example: You may want to convert all character to uppercase as they are being entered.

```
handleChange: function (event) {
  this.setState({
    helloTo: event.target.value.toUpperCase()
  });
}
```

You will notice when entering data that there is no flicker of the lowercase character before the uppercase character is added to the input. This is because React.js is intercepting the native browser change event. Then this component re-renders the input after setState has been called. React.js then does the DOM diff and updates the value of the input.

You can use this same pattern to limit what characters can be entered, or not allow illegal characters to be entered into an email address.

You may also want to use the value in other components as the data is being entered.

Example:

- Show how many characters are left in a size limited input.
- Display the color of a HEX value being entered.
- Display input validation errors and warnings.

Form Events

Accessing form events is a crucial aspect to controlling different aspects of your forms.

All of the events produced by HTML are supported in React.js. They follow camel case naming conventions and are converted to synthetic events. They are standardized, with a common interface cross browser.

All synthetic events give you access to the DOMNode that emitted the event via `event.target`.

```
  var newValue = DOMNode.value;
}
```

This is one of the easiest ways to access the value of **Controlled** components.

Label

Labels on form elements are important for clearly communicating your requirements of your users, and provide accessibility for radios and checkboxes.

There is a conflict with the `for` attribute. When using JSX the attributes are converted to a JavaScript object and passed to the component constructor as the first argument. Since `for` is a reserved word in JavaScript, we can't use it as the name of an object property.

In React.js, just as `class` becomes `className`, so too does `for` become `htmlFor`.

```
//JSX
<label htmlFor="name">Name:</label>

//javascript
React.DOM.label({htmlFor:"name"}, "Name:");

//after render
<label for="name">Name:</label>
```

Textarea and Select

React.js makes some changes to the interface of <textarea/> and <select/> to increase consistency and make it easier to manipulate.

<textarea/> is changed to be closer to <input/> allowing you to specify `value` and `defaultValue.`

```
//Uncontrolled
<textarea defaultValue="Hello World" />

//Controlled
<textarea
  value={this.state.helloTo}
  onChange={this.handleChange} />
```

<select/> now accepts `value` and `defaultValue` to set which option is selected. This allows easier manipulation of the value.

```
//Uncontrolled
<select defaultValue="B">
  <option value="A">First Option</option>
  <option value="B">Second Option</option>
```

```
</select>

//Controlled
<select value={this.state.helloTo} onChange={this.handleChange}>
  <option value="A">First Option</option>
  <option value="B">Second Option</option>
  <option value="C">Third Option</option>
</select>
```

React.js supports multi select. In order to use multi select you must pass an array to `value` and `defaultValue`.

```
//Uncontrolled
<select multi="true" defaultValue={["A","B"]}>
  <option value="A">First Option</option>
  <option value="B">Second Option</option>
  <option value="C">Third Option</option>
</select>
```

When using multi select, the value of the select component is **not** updated when the options are selected. Only the `selected` property of the option is changed. Using a given `ref` or `syntheticEvent.target` you can access the options and check if they are selected.

In the following example `handleChange` is looping over the DOM to find out what option is currently selected.

```
//http://jsfiddle.net/u97u3tmt/
var MyForm = React.createClass({
  getInitialState: function () {
    return {
      options: ["B"]
    };
  },
  handleChange: function (event) {
    var checked = [];
    var sel = event.target;
    for(var i=0; i < sel.length; i++){
      var option = sel.options[i];
      if (option.selected){
        checked.push(option.value);
      }
    }
    this.setState({
      options: checked
    });
  },
  submitHandler: function (event) {
    event.preventDefault();
    alert(this.state.options);
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
        <select multiple="true"
```

```
          <option value="A">First Option</option>
          <option value="B">Second Option</option>
          <option value="C">Third Option</option>
        </select>
        <br />
        <button type="submit">Speak</button>
      </form>;
  }
});
```

Checkbox and Radio

Checkboxes and Radios have a different mechanism for controlling them.

Just as in HTML, an <input/> with type `checkbox` or `radio` behave quite differently to an <input/> with type `text`. Generally, the value of a checkbox or radio does not change. Only the `checked` state changes. To control a checkbox or radio input, you need to control the `checked` attribute. You can also use `defaultChecked` in an uncontrolled checkbox or radio input.

```
//Uncontrolled - http://jsfiddle.net/euusg6bu/
var MyForm = React.createClass({
  submitHandler: function (event) {
    event.preventDefault();
    alert(this.refs.checked.getDOMNode().checked);
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
        <input
          ref="checked"
          type="checkbox"
          value="A"
          defaultChecked="true" />
        <br />
        <button type="submit">Speak</button>
      </form>;
  }
});

//Controlled - http://jsfiddle.net/2k7y2p7r/
var MyForm = React.createClass({
  getInitialState: function () {
    return {
      checked: true
    };
  },
  handleChange: function (event) {
    this.setState({
      checked: event.target.checked
    });
  },
  submitHandler: function (event) {
    event.preventDefault();
    alert(this.state.checked);
  },
  render: function () {
```

```
            type="checkbox"
            value="A"
            checked={this.state.checked}
            onChange={this.handleChange} />
          <br />
          <button type="submit">Speak</button>
        </form>;
    }
  });
```

In both the examples, at all times the value of the <input/> will always be `A`, only the checked state changes.

Names on Form Elements

Names carry less importance on from elements in React.js when controlled form elements have values that are stored in state and the form submit event is being intercepted. Names aren't required to access the form values. For uncontrolled form elements, you can use refs for direct access to the form element also.

However, names are a crucial aspect of your form components.

- Names allow third party form serializers to still work within React.js.
- Names are also required if the form is going to be natively submitted.
- Names are used by your clients browser for auto filling common information like the users address.
- Names are crucial to uncontrolled **radio** inputs as that is how they are grouped to ensure that only one radio with the same name in a form can be checked at once. The same behaviour can be replicated without names using controlled radio inputs.

The following example replicates the functionality of an **uncontrolled** radio group by storing the state in the MyForm component. You will notice that **name** is not being used.

```
//http://jsfiddle.net/hh986e09/
var MyForm = React.createClass({
  getInitialState: function () {
    return {
      radio: "B"
    };
  },
  handleChange: function (event) {
    this.setState({
      radio: event.target.value
    });
  },
  submitHandler: function (event) {
    event.preventDefault();
    alert(this.state.radio);
  },
```

```
        <input
          type="radio"
          value="A"
          checked={this.state.radio == "A"}
          onChange={this.handleChange} /> A
        <br />
        <input
          type="radio"
          value="B"
          checked={this.state.radio == "B"}
          onChange={this.handleChange} /> B
        <br />
        <input
          type="radio"
          value="C"
          checked={this.state.radio == "C"}
          onChange={this.handleChange} /> C
        <br />
        <button type="submit">Speak</button>
      </form>;
  }
});
```

Multiple Form Elements and Change Handlers

When using **Controlled** form elements, you don't want to be writing a change handler for every form element. Fortunately, here are a few different ways of re-using a change handler with React.js.

Example: passing extra arguments to .bind.

```
//http://jsfiddle.net/gb4wxfe2/
var MyForm = React.createClass({
  getInitialState: function () {
    return {
      given_name: "",
      family_name: ""
    };
  },
  handleChange: function (name, event) {
    var newState = {};
    newState[name] = event.target.value;
    this.setState(newState);
  },
  submitHandler: function (event) {
    event.preventDefault();
    var words = [
      "Hi",
      this.state.given_name,
      this.state.family_name
    ];
    alert(words.join(" "));
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
        <label htmlFor="given_name">Given Name:</label>
        <br />
```

```
          type="text"
          name="given_name"
          value={this.state.given_name}
          onChange={this.handleChange.bind(this,'given_name')}/>
        <br />
        <label htmlFor="family_name">Family Name:</label>
        <br />
        <input
          type="text"
          name="family_name"
          value={this.state.family_name}
          onChange={this.handleChange.bind(this,'family_name')}/>
        <br />
        <button type="submit">Speak</button>
      </form>;
  }
});
```

Example: use DOMNode name to identify what state to change.

```
//http://jsfiddle.net/smgtp2kw/
var MyForm = React.createClass({
  getInitialState: function () {
    return {
      given_name: "",
      family_name: ""
    };
  },
  handleChange: function (event) {
    var newState = {};
    newState[event.target.name] = event.target.value;
        this.setState(newState);
  },
  submitHandler: function (event) {
    event.preventDefault();
    var words = [
      "Hi",
      this.state.given_name,
      this.state.family_name
    ];
    alert(words.join(" "));
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
        <label htmlFor="given_name">Given Name:</label>
        <br />
        <input
          type="text"
          name="given_name"
          value={this.state.given_name}
          onChange={this.handleChange}/>
        <br />
        <label htmlFor="family_name">Family Name:</label>
        <br />
        <input
          type="text"
          name="family_name"
```

```
        <br />
        <button type="submit">Speak</button>
      </form>;
  }
});
```

The last two examples are very similar but take a different approach to solving the same problem. React.js also provides a mixin as part of their addons, `React.addons.LinkedStateMixin` solves this same problem in yet another way.

React.addons.LinkedStateMixin adds the method `linkState` to your component. `linkState` returns an object with two properties, value and requestChange.

`value` takes it's value from state from the property name supplied.

`requestChange` is a function that updates the named state with the new value.

```
this.linkState('given_name');

//returns
{
  value: this.state.given_name,
  requestChange: function (newValue) {
    this.setState({given_name: newValue});
  }
}
```

This object needs to be passed to a React.js special non DOM attribute `valueLink`. `valueLink` updates the value of the input with the value of the object supplied and provides an `onChange` handler that calls requestChange with the new DOMNode value.

```
//http://jsfiddle.net/q0cn6wcw/
var MyForm = React.createClass({
  mixins: [React.addons.LinkedStateMixin],
  getInitialState: function () {
    return {
      given_name: "",
      family_name: ""
    };
  },
  submitHandler: function (event) {
    event.preventDefault();
    var words = [
      "Hi",
      this.state.given_name,
      this.state.family_name
    ];
    alert(words.join(" "));
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
        <label htmlFor="given_name">Given Name:</label>
```

```
            type="text"
            name="given_name"
            valueLink={this.linkState('given_name')} />
          <br />
        <label htmlFor="family_name">Family Name:</label>
        <br />
        <input
          type="text"
          name="family_name"
          valueLink={this.linkState('family_name')} />
        <br />
        <button type="submit">Speak</button>
      </form>;
  }
});
```

This is a much easier way to control an input and store the value in the parent components state. The data flow stays the same as other controlled form elements.

However, using this approach becomes more complex to inject custom functionality into the data flow. This mixin is only recommended to be use in limited circumstances. As the traditional controlled form element approach provides the same functionality and more flexibility.

Custom Form Components

Creating custom form components is an excellent way to reuse common functionality in your applications. It can also be a great way to improve the interface to other more complex form components like Checkboxes and Radios.

When writing custom form components you should try to create the same interface as other form components. This will increase the predicability of your code, making it much easier to understand how your component works without having to look at it's implementation.

Let's create a custom radio component that has the same interface as the React.js select component. We won't implement multi select functionality as radio components don't support multi select.

```
var Radio = React.createClass({
  propTypes: {
    onChange: React.PropTypes.func
  },
  getInitialState: function () {
    return {
      value: this.props.defaultValue
    };
  },
  handleChange: function (event) {
    if (this.props.onChange) {
      this.props.onChange(event);
    }
    this.setState({
```

```
  },
  render: function () {
    var children = {};
    var value = this.props.value || this.state.value;

    React.Children.forEach(this.props.children, function (child, i) {
      var label = <label>
          <input
            type="radio"
            name={this.props.name}
            value={child.props.value}
            checked={child.props.value == value}
            onChange={this.handleChange} />
          {child.props.children}
          <br/>
        </label>;

        children['label' + i] = label;
    }.bind(this));

    return this.transferPropsTo(<span>{children}</span>);
  }
});
```

Essentially we are creating a **Controlled** component that supports the Controlled and Uncontrolled interface.

The first thing you do is make sure that if `onChange` is supplied that it is always a function. Then we store the defaultValue in state.

Each time the component renders, it creates new labels and radios based of the options that were supplied as children to our component. We also make sure that the dynamic children have consistent keys each render. This ensures that React.js will keep our <input/>'s in the DOM and maintain current focus when using keyboard controls.

We then set the value, name and checked state. And we attach our onChange handler, then render our new children.

```
//Uncontrolled
//http://jsfiddle.net/jqqpk0uh/
var MyForm = React.createClass({
  submitHandler: function (event) {
    event.preventDefault();
    alert(this.refs.radio.state.value);
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
        <Radio ref="radio" name="my_radio" defaultValue="B">
          <option value="A">First Option</option>
          <option value="B">Second Option</option>
          <option value="C">Third Option</option>
        </Radio>
        <button type="submit">Speak</button>
      </form>;
```

We did have to change the interface slightly when using our component **Uncontrolled**. As when you use `.getDOMNode` on `this.refs.radio`, you will get the DOMNode. Not the active <input/>. In React.js you can't change the functionality of `getDOMNode()` to overwrite this behaviour.

As we are storing the value in state of our component, we don't need to access the DOMNode to know what the current value is. We can access the state directly.

```
//Controlled
//http://jsfiddle.net/13ux2797/
var MyForm = React.createClass({
  getInitialState: function () {
    return {my_radio: "B"};
  },
  handleChange: function (event) {
    this.setState({
      my_radio: event.target.value
    });
  },
  submitHandler: function (event) {
    event.preventDefault();
    alert(this.state.my_radio);
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
        <Radio name="my_radio"
            value={this.state.my_radio}
            onChange={this.handleChange}>
          <option value="A">First Option</option>
          <option value="B">Second Option</option>
          <option value="C">Third Option</option>
        </Radio>
        <button type="submit">Speak</button>
      </form>;
  }
});
```

As a **Controlled** component, this operates exactly the same as a select box. The event passed to `onChange` is the event from the active <input/> so you can use it read the current value.

As an exercise, you may want to try and implement support for a valueLink property so you can use this component with `React.addons.LinkedStateMixin`.

Focus

Leveraging control of focus on form components is an excellent way to guide your users as to what the next logical step is in your forms. It also helps cut down on user interaction, increasing usability.

Since React is forms are not always rendered at browser load, the auto focus for form inputs needs

mounted, if no other form input has focus, React.js will place focus on the input. This is how you would expect a simple HTML form with `autoFocus` to operate.

```
//jsx
<input type="text" name="given_name" autoFocus="true" />
```

You can also manually set the focus of form fields by calling `focus()` on the DOMNode.

Usability

React.js is awesome for productivity for developers, however, this can have down sides.

It is really easy to make components that lack usability. For example, you may have forms with little keyboard support where only `onClick` of a hyperlink can submit the form. This stops a user from pressing `enter` on their keyboard to submit the form, which is the default behaviour of HTML forms.

It is also really easy to make fantastic components that are highly usable. Time and consideration are required when building your components. It is all the "little things" that help make a component be highly usable and "feel right".

This following is a collection of best practice for creating usable forms. They are not specific to React.js.

Communicate your requirements clearly

Good communication is important with all aspects of your application, especially in forms.

Use of HTML labels is a great way to communicate to your users what the form element is expecting. These also give the user an extra way to interact with your form element for some input types like radios and checkboxes.

Placeholders are designed to show example input or a default value if no data is entered. There has been a fad to place validation hints in the placeholder. This can be quite problematic as when the user starts to type, their validation hints disappear. It is generally better to show your validation hints along side your inputs or as a popover when your validation requirements are not met.

Give feedback constantly

This follows on from communicating your requirements clearly. It is very important to give feedback to your users as quickly as possible.

Validation errors are a perfect example of giving feedback constantly. It is well known that showing validation errors as they occur increase the usability of your forms. Back in the early days of the web applications all users had to wait till they finished completing their form to find out if they

It is also important to show your users that you are working on their request. This is especially true for actions that can take some time to complete. Showing spinners, progress bars, notification messages etc are a great way to inform your users that your application has not frozen. Users are very impatient at times, however they can be very patient if they know your application is processing their request.

Transitions and animations are another great way of informing your users what is happening in your application. They are another visual prompt that something in your application has changed.

Be fast

React.js has a very powerful rendering engine, it helps your application to be quite fast right out of the box. However there are times when the speed of updating the DOM is not what is slowing down your application.

Transitions are a perfect example. As transitions that take to long may frustrate your users and slow your power users down as it reduces how much time your user can engage with your applications.

Other factors outside your application can also alter how fast your application performs. Long running AJAX calls and poor network performance also impact the speed of your applications. How to solve these kinds of issues can be very specific to your application and may be outside of your control like third party services.

It is important to remember that speed is relative. It is a perception of the user. It is more important to appear fast than be fast. Example, when a user clicks “like” in your application, you can increment the like count before you send your AJAX call to the server. This way if the AJAX call takes a long time, your users won’t see the delay. This can however lead into other issues with handling errors etc.

Be predictable

Users have a predetermined idea of how things work. This is based on their prior experience. In most cases their prior experience is **not** from using your application.

If you application resembles the platform your user is on, your user will expect your application conforms to the default behaviours for the platform you are on.

With this in mind, you are left with two options, either conform to the default behaviours of the platform, or, radically change your user interface so it no longer resembles he platform.

Consistency is another form of predictability, if your interactions are always the same in different parts of your application, your users will learn to predict the interaction in new parts of your application. This ties into conforming with platform that your application runs on.

Be accessible

Accessibility is often over looked by developers and designers when creating user interfaces. It is important to keep your users in mind when considering all aspects you user interfaces. As already covered, your users have a predefined expectation of how things work, this is based on **their** past experience.

Their past experience also governs their preference for different input types. In some cases your users may have physical issues with using particular input device like a keyboard or mouse. They may also have issues with using an output device like a display or speakers.

Building support for all the different types of input and output device may not be viable for every aspect of your application. It is important to understand your users requirements and preferences, then work on those areas first.

A great way to test how accessible your application is to try and navigate your application exclusively via one input device keyboard / mouse / touch screen. This will highlight usability issues with that device.

You may also want to consider how to interact with your application if you were visually impaired. Screen readers are the eyes for visually impaired people.

Reduce user input

Reducing how much data your users need to enter is a great way to improve usability for your applications. The less information your users need to enter, the less chance they have of making mistakes and the less they need to think about.

As with **Be fast** user perception is important. Users feel daunted by large forms with many input fields, their mind struggles to cope. Breaking your forms into smaller more manageable chunks gives the impression of less user input. This in turn enables the user to be more focused on their data input.

Autofill is another great way to reduce data input. Leveraging the users browser autofill data can save the user from re-entering common information like their address or payment details.

Autocomplete can help guide your users as they are entering information, this ties in with giving constant feedback. Example, when searching for a movie, autocomplete can help alleviate issues with bad spelling of the movie title.

Another useful way to reduce input is to derive information from previously entered data. Example, if a user is entering credit card details, you can look at the first 4 numbers and determine what type of card it is, then select the card type for your user, this both reduces input and provides validation feedback to the user that they are typing their card number correctly.

Autofocus is a small but very effective way to increase usability of your forms. Autofocus helps to

on their own. This small tool can really have a large impact on how quickly your users can start entering data.