PREV
5. Event Handling

NEXT
7. Mixins

Chapter 6. Composing Components

In traditional HTML the basic building block of each page is an element. In React you assemble a page from components. You can think of a React component as an HTML element with all the expressive power of JavaScript mixed in. In fact with React the *only* thing you do is build components, just like an HTML document is build with only elements.

Since the entirety of a React app is build using components, this whole book can be described as a book about React components. Therefore in this chapter we won't cover *everything* to do with components. Rather you will be introduced to one specific aspect — their composability.

A component is basically a JavaScript function that takes in props and state as its arguments and outputs rendered HTML. They are typically designed to represent and express a piece of data within your app so you can think of a React component as an extension of HTML.

Extending HTML

React + JSX are powerful and expressive tools allowing us to create custom elements that can be expressed using an HTML-like syntax. However they go far beyond plain HTML, allowing us to control their behavior throughout their lifetime. This all starts with the `React.createClass` method.

React favors composition over inheritance, which means we combine small, simple components and data objects into larger, more complex components. If you're familiar with other MVC or object-oriented tools you might expect to find a `React.extendClass` method, as I did. However, just as you don't extend html dom nodes when building a web page, you don't extend React components. Instead you compose them.

React embraces composability, allowing you to mix-and-match various child components into an intricate and powerful new component. To demonstrate this let's consider how a user would answer a survey question

. Specifically let's look at the `AnswerMultipleChoiceQuestion` component responsible for rendering a multiple choice question and capturing the user's answer.

Obviously a survey will be built around basic html form elements. You will be crafting our survey answer components by wrapping the default html input elements and customizing their behavior.

Composition By Example

Let's consider the component representing a multiple-choice question. This has a few requirements.

- Take a list of choices as input
- Render the choices to the user
- Only allow the user to select a single choice

We know html provides us some some basic elements to help us, namely the "radio" type inputs and input groups. Thinking of it from the top-down our component hierarchy will look something like this:

```
MultipleChoice → RadioInput → Input (type="radio")
```

You can think of those arrows as representing the phrase *has a*. The MulitpleChoice component *has a* RadioInput. The RadioInput *has an* input. This is an identifying trait of the composition pattern.

Assemble the HTML

Let's start by assembling the components from the bottom-up. React pre-defines the `input` component for us in the `React.DOM.input` namespace. Therefore the first thing you'll do is wrap it in a `RadioInput` component. This component will be responsible for customizing the generic `input`, narrowing it's scope to behave like a radio button. Within the accompanying sample app it is named `AnswerRadioInput`.

First create the scaffolding, which will include the required render method and the basic markup to describe the desired output. You can already begin to see the composition pattern as the component becomes a specialized type of input.

```
var AnswerRadioInput = React.createClass({
render: function () {
    return (
      <div className="radio">
        <label>
          <input type="radio" />
          Label Text
        </label>
```

```
  }
});
```

Add Dynamic Properties

Nothing about our `input` is dynamic yet, so next you need to define the properties the parent must pass into the radio input.

- What value, or choice, does this input represent? (required)
- What text do we use to describe it? (required)
- What is the input's name? (required)
- We'll might want to customize the id.
- We might want to override the default value.

Given this list you can now define the property types for our custom input. Add these to the `propTypes` hash of the class definition.

```
var AnswerRadioInput = React.createClass({
  propTypes: {
    id: React.PropTypes.string,
    name: React.PropTypes.string.isRequired,
    label: React.PropTypes.string.isRequired,
    value: React.PropTypes.string.isRequired,
    checked: React.PropTypes.bool
  },
  ...
});
```

For each optional property you need to define the default value. Add these to the `getDefaultProps` method. These values will be applied to each new instance when the parent component does not provide their value.

Since this method is only called once for the class, not for each instance, we cannot provide the id here — it must be unique for each instance. That will be solved using state below.

```
var AnswerRadioInput = React.createClass({
  propTypes: {...},
  getDefaultProps: function () {
    return {
      id: null,
      checked: false
    };
  },
  ...
});
```

Track State

Our component needs to keep track of data that changes over time. Specifically, the `id` will be unique for each instance and the user can update the `checked` value at any time. Therefore we define the initial state.

```
var AnswerRadioInput = React.createClass({
  propTypes: {...},
  getDefaultProps: function () {...},
  getInitialState: function () {
    var id = this.props.id ? this.props.id : uniqueId('radio-');
    return {
      checked: !!this.props.checked,
      id: id,
      name: id
    };
  },
  ...
});
```

Now you can update the rendered markup to access the new dynamic state and props.

```
var AnswerRadioInput = React.createClass({
  propTypes: {...},
  getDefaultProps: function () {...},
  getInitialState: function () {...},
  render: function () {
    return (
      <div className="radio">
        <label htlmFor={this.props.id}>
          <input type="radio"
            name={this.props.name}
            id={this.props.id}
            value={this.props.value}
            checked={this.state.checked} />
          {this.props.label}
        </label>
      </div>
    );
  }
});
```

Integrate into a Parent Component

At this point you have enough of a component to use it within a parent so you're ready to build up the next layer, the `AnswerMultipleChoiceQuestion`. The primary responsibility here is to display a list of choices for the user to choose from. Following the pattern introduced above, let's lay down the basic html and the default props for this component.

```
var AnswerMultipleChoiceQuestion = React.createClass({
  propTypes: {
    value: React.PropTypes.string,
    choices: React.PropTypes.array.isRequired,
```

```
  getInitialState: function() {
    return {
      id: uniqueId('multiple-choice-'),
      value: this.props.value
    };
  },
  render: function() {
    return (
      <div className="form-group">
        <label className="survey-item-label" htmlFor={this.state.id}>{this.props.label}<
        <div className="survey-item-content">
            <AnswerRadioInput ... />
            ...
            <AnswerRadioInput ... />
        </div>
      </div>
    );
  }
});
```

In order to generate the list of child radio input components we must map over the choices array, transforming each into a component. This is easily handled in a helper function as demonstrated here.

```
var AnswerMultipleChoiceQuestion = React.createClass({
  ...
  renderChoices: function() {
    return this.props.choices.map(function(choice, i) {
      return AnswerRadioInput({
        id: "choice-" + i,
        name: this.state.id,
        label: choice,
        value: choice,
        checked: this.state.value === choice
      });
    }.bind(this));
  },
  render: function() {
    return (
      <div className="form-group">
        <label className="survey-item-label" htmlFor={this.state.id}>{this.props.label}<
        <div className="survey-item-content">
            {this.renderChoices()}
        </div>
      </div>
    );
  }
});
```

Now the composability of React is becoming more clear. You started with a generic input, customized it into a radio input, and finally wrapped it into a multiple-choice component — a highly refined and specific version of a form control. Now rendering a list of choices is as simple as this:

```
<AnswerMultipleChoiceQuestion choices={arrayOfChoices} ... />
```

The astute reader has probably noticed there is a missing piece — our radio inputs have no way of communicating changes to their parent component. You need to wire up the `AnswerRadioInput` children so the parent is aware of their changes and can process them into a proper survey-result data payload. Which brings us to the parent/child relationship.

Parent / Child Relationship

At this point you should be able to render a form to the screen, but notice that you have yet to give the components the ability to share user changes. The `AnswerRadioInput` component does not yet have the ability to communicate with its parent.

The easiest way for a child to communicate with its parent is via props. The parent needs to pass in a callback via the props, which the child calls it when needed.

First you need to define what `AnswerMultipleChoiceQuestion` will do with the changes from it's children. Add a `handleChanged` method and pass it into each `AnswerRadioInput`.

```
var AnswerMultipleChoiceQuestion = React.createClass({
  ...
  handleChanged: function(value) {
    this.setState({value: value});
    this.props.onCompleted(value);
  },
  renderChoices: function() {
    return this.props.choices.map(function(choice, i) {
      return AnswerRadioInput({
        ...
        onChanged: this.handleChanged
      });
    }.bind(this));
  },
  ...
});
```

Now each radio input can watch for user changes, passing the value up to the parent. This requires wiring up an event handler to the input's `onChange` event.

```
var AnswerRadioInput = React.createClass({
  propTypes: {
    ...
    onChanged: React.PropTypes.func.isRequired
  },
  handleChanged: function (e) {
    var checked = e.target.checked;
    this.setState({checked: checked});
    if(checked) {
      this.props.onChanged(this.props.value);
```

```
  render: function () {
    return (
      <div className="radio">
        <label htmlFor={this.state.id}>
          <input type="radio"
            ...
            onChange={this.handleChanged} />
          {this.props.label}
        </label>
      </div>
    );
  }
});
```

Wrap Up

Now you've seen how React uses the pattern of composability, allowing you to wrap html elements or custom components and customize their behavior for your needs. As you compose components they become more specific and semantically meaningful. Thus React takes generic inputs,

```
<input type="radio" ... />
```

and transforms them into something a bit more meaningful,

```
<AnswerRadioInput ... />
```

finally ending up with a single component to transform an array of data into a useful UI for users to interact with.

```
<AnswerMultipleChoiceQuestion choices={arrayOfChoices} ... />
```

Composition is just one way React offers to customize and specialize your components. Mixins offer another approach, allowing you to define methods that can be shared across many components. Next we show you how to define a mixin and how they can be used to share common code.