Multiple Components

So far, we've looked at how to write a single component to display data and handle user input. Next let's examine one of React's finest features: composability.

Motivation: Separation of Concerns

By building modular components that reuse other components with well-defined interfaces, you get much of the same benefits that you get by using functions or classes. Specifically you can *separate the different concerns* of your app however you please simply by building new components. By building a custom component library for your application, you are expressing your UI in a way that best fits your domain.

Composition Example

Let's create a simple Avatar component which shows a profile picture and username using the Facebook Graph API.

Code

```
/** @jsx React.DOM */

var Avatar = React.createClass({
  render: function() {
    return (
      <div>
        <ProfilePic username={this.props.username} />
        <ProfileLink username={this.props.username} />
      </div>
    );
```

```
var ProfilePic = React.createClass({
  render: function() {
    return (
      <img src={'http://graph.facebook.com/' + this.props.username + '/picture'} /
    );
  }
});

var ProfileLink = React.createClass({
  render: function() {
    return (
      <a href={'http://www.facebook.com/' + this.props.username}>
        {this.props.username}
      </a>
    );
  }
});

React.renderComponent(
  <Avatar username="pwh" />,
  document.getElementById('example')
);
```

Ownership

In the above example, instances of `Avatar` *own* instances of `ProfilePic` and `ProfileLink`. In React, **an owner is the component that sets the `props` of other components**. More formally, if a component `X` is created in component `Y`'s `render()` method, it is said that `X` is *owned by* `Y`. As discussed earlier, a component cannot mutate its `props` — they are always consistent with what its owner sets them to. This key property leads to UIs that are guaranteed to be consistent.

relationship is simply the one you know and love from the DOM. In the example above, `Avatar` owns the `div`, `ProfilePic` and `ProfileLink` instances, and `div` is the **parent** (but not owner) of the `ProfilePic` and `ProfileLink` instances.

Children

When you create a React component instance, you can include additional React components or JavaScript expressions between the opening and closing tags like this:

Code

```
<Parent><Child /></Parent>
```

`Parent` can read its children by accessing the special `this.props.children` prop.

Child Reconciliation

Reconciliation is the process by which React updates the DOM with each new render pass. In general, children are reconciled according to the order in which they are rendered. For example, suppose two render passes generate the following respective markup:

Code

```
// Render Pass 1
<Card>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</Card>
// Render Pass 2
<Card>
  <p>Paragraph 2</p>
</Card>
```

according to the *order* of the children.

Stateful Children

For most components, this is not a big deal. However, for stateful components that maintain data in `this.state` across render passes, this can be very problematic.

In most cases, this can be sidestepped by hiding elements instead of destroying them:

Code

```
// Render Pass 1
<Card>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</Card>
// Render Pass 2
<Card>
  <p style={{display: 'none'}}>Paragraph 1</p>
  <p>Paragraph 2</p>
</Card>
```

Dynamic Children

The situation gets more complicated when the children are shuffled around (as in search results) or if new components are added onto the front of the list (as in streams). In these cases where the identity and state of each child must be maintained across render passes, you can uniquely identify each child by assigning it a `key`:

Code

```
  render: function() {
    var results = this.props.results;
    return (
```

```
        return <li key={result.id}>{result.text}</li>;
      })}
    </ol>
  );
}
```

When React reconciles the keyed children, it will ensure that any child with `key` will be reordered (instead of clobbered) or destroyed (instead of reused).

Data Flow

In React, data flows from owner to owned component through `props` as discussed above. This is effectively one-way data binding: owners bind their owned component's props to some value the owner has computed based on its `props` or `state`. Since this process happens recursively, data changes are automatically reflected everywhere they are used.

A Note on Performance

You may be thinking that it's expensive to react to changing data if there are a large number of nodes under an owner. The good news is that JavaScript is fast and `render()` methods tend to be quite simple, so in most applications this is extremely fast. Additionally, the bottleneck is almost always the DOM mutation and not JS execution and React will optimize this for you using batching and change detection.

However, sometimes you really want to have fine-grained control over your performance. In that case, simply override `shouldComponentUpdate()` to return false when you want React to skip processing of a subtree. See the React reference docs for more information.

> **Note:**
>
> If `shouldComponentUpdate()` returns false when data has actually changed, React can't

how fast JavaScript is relative to the DOM.

← Prev Next →

Add a comment...

Post to Facebook Posting as Stoyan Stefanov (Change) Comment

Facebook social plugin