```
    header="Languages"
    languages={languages} />,
  document.getElementById('app')
)
```

Here you can find working example of it.

## Section 2.7: setState pitfalls

You should use caution when using `setState` in an asynchronous context. For example, you might try to call `setState` in the callback of a get request:

```
class MyClass extends React.Component {
    constructor() {
        super();

        this.state = {
            user: {}
        };
    }

    componentDidMount() {
        this.fetchUser();
    }

    fetchUser() {
        $.get('/api/users/self')
            .then((user) => {
                this.setState({user: user});
            });
    }

    render() {
        return <h1>{this.state.user}</h1>;
    }
}
```

This could call problems - if the callback is called after the `Component` is dismounted, then `this.setState` won't be a function. Whenever this is the case, you should be careful to ensure your usage of `setState` is cancellable.

In this example, you might wish to cancel the XHR request when the component dismounts:

```
class MyClass extends React.Component {
    constructor() {
        super();

        this.state = {
            user: {},
            xhr: null
        };
    }

    componentWillUnmount() {
        let xhr = this.state.xhr;

        // Cancel the xhr request, so the callback is never called
        if (xhr && xhr.readyState != 4) {
            xhr.abort();
        }
    }
    componentDidMount() {
        this.fetchUser();
    }

    fetchUser() {
        let xhr = $.get('/api/users/self')
            .then((user) => {
                this.setState({user: user});
            });

        this.setState({xhr: xhr});
    }
}
```

The async method is saved as a state. In the `componentWillUnmount` you perform all your cleanup - including canceling the XHR request.

You could also do something more complex. In this example, I'm creating a 'stateSetter' function that accepts the this object as an argument and prevents `this.setState` when the function `cancel` has been called:

```
function stateSetter(context) {
    var cancelled = false;
    return {
        cancel: function () {
            cancelled = true;
        },
        setState(newState) {
            if (!cancelled) {
                context.setState(newState);
            }
        }
    }
}

class Component extends React.Component {
    constructor(props) {
        super(props);
        this.setter = stateSetter(this);
        this.state = {
            user: 'loading'
        };
    }
    componentWillUnmount() {
        this.setter.cancel();
    }
    componentDidMount() {
        this.fetchUser();
    }
    fetchUser() {
        $.get('/api/users/self')
            .then((user) => {
                this.setter.setState({user: user});
            });
    }
    render() {
        return <h1>{this.state.user}</h1>
    }
}
```

This works because the `cancelled` variable is visible in the `setState` closure we created.