PREV
11. Performance Tuning

NEXT
13. Addons

Chapter 12. Server Side Rendering

Server side rendering is vital if you want search engines to crawl your site. It also provides a performance boost as the browser can start displaying your site while your javascript is still loading.

The virtual DOM in React.js is central to why React.js can be used for server side rendering. Each React.js component is first rendered to the virtual DOM, then React.js takes the virtual DOM and updates the real DOM with any changes. The virtual DOM, being an in memory representation of the DOM is what allows React.js to work in non browser environments like nodejs. Instead of updating the real DOM, React.js can generate a string from the virtual DOM. This allows you to use the same React.js components on the client and the server.

React.js provides two functions which can be used to render your components on the server side, `React.renderComponentToString` and `React.renderComponentToStaticMarkup`.

Server side rendering of your components requires forsight when designing your components. You need to consider:

- What render function is best to use.
- How to support Asychronous state of your components.
- How to pass your applications initial state to the client.
- What lifecycle functions are available on the server side.
- How to support Isomorphic routing for you application.
- Your usage of Singletons, Instances and Context.

Render functions

SOME INTRO TEXT HERE

React.renderComponentToString

The first of the two render functions that can be used on the server side is `React.renderComponentToString`. This is function you will mostly use.

Unlike React.renderComponent this function does not take an argument of where to render, instead it returns a string. This is a synchronous (blocking) function that is very fast.

```
var MyComponent = React.createClass({
  render: function () {
    return <div>Hello World!</div>;
  }
});

var world = React.renderComponentToString(<MyComponent/>);

//single line output - formatted for this example
<div
  data-reactid=".fgvrzhg2yo"
  data-react-checksum="-1663559667"
>Hello World!</div>
```

You will notice that React.js has added two data attributes to the `<div>`.

`data-reactid` is used by React.js to identify the DOM node in the browser environment. This is how it knows what DOM node to update when state and props change.

`data-react-checksum` is only added on the server side. As the name suggest it is a checksum of the DOM that is created. This allows React.js to reuse the DOM from the server when rendering the same component on the client. This is only added to the root element.

React.renderComponentToStaticMarkup

`React.renderComponentToStaticMarkup` is the second render function you can use on the server side.

This is the same as React.renderComponentToString except it doesn’t include the React.js data attributes.

```
var MyComponent = React.createClass({
  render: function () {
    return <div>Hello World!</div>;
  }
});

var world = React.renderComponentToStaticMarkup(<MyComponent/>);

//single line output
<div>Hello World!</div>
```

String or StaticMarkup

Each render function has it purposes, you need look at what your requirements are to decide which render function to use.

>Only use `React.renderComponentToStaticMarkup` when you are **not** going to render the same React.js component on the client.

Examples:

- Generating html emails.
- Generating PDF's by html to pdf conversion.
- Testing components.

In most cases you will want to use `React.renderComponentToString`. This will allow React.js to use the `data-react-checksum` to make the initialization of the same React.js component on the client much faster. As React.js can reuse the DOM supplied from the server, it can skip the expensive process of generating DOM nodes and attaching them to the document. For a complex site, this can significantly reduce the load so your users can start interacting with your site faster.

NOTE:
It is very important that your React.js components render **exactly** the same on the server and on the client. If the `data-react-checksum` doesn't match, React.js will destroy the DOM supplied by the server and generate new DOM nodes and attach them to the document. In this case you will loose much of the performance gain of server side rendering.

Server Side Component Lifecycle

When rendering to string, only lifecycle methods **before** render are called. Crucially `componentDidMount` and `componentWillUnmount` are not called during the render process and `componentWillMount` is called from both.

You will need to keep this in mind when creating components that will be rendered on the server and the client. Especially when creating event listeners, as there is no component lifecycle method to let you know when the React.js component is finished.

Any event listeners or timers started in `componentWillMount` have the potential to cause memory leaks on the server.

Best practice is to only create event listeners and timers from `componentDidMount` and stop the event listeners and timers in `componentWillUnmount`.

Designing Components

When rendering on the server side, you need to take special consideration to how your state will be designing your components with server sider rendering in mind.

You should always design your React.js components so that when you pass the same props to the component you will always get the same **initial** render. If you always do this, you will increase testability of your components and when you render on the server and the client you can be guaranteed they will render the same. This is very important to ensure you will get the performance benefit of server side rendering.

Let's say that you want to have a component that prints a random number. This is an issue as the result will almost always be different. If this component were to render on the server then render on the client, the checksum would fail.

```
var MyComponent = React.createClass({
  render: function () {
    return <div>{Math.random()}</div>;
  }
});

var result = React.renderComponentToStaticMarkup(<MyComponent/>);
var result2 = React.renderComponentToStaticMarkup(<MyComponent/>);

//result
<div>0.5820949131157249</div>

//result2
<div>0.420401572631672</div>
```

If you were to change your component so it received the random number by way of props. You can then pass the props to the client to be used for the rendering.

```
var MyComponent = React.createClass({
  render: function () {
    return <div>{this.props.number}</div>;
  }
});

var num = Math.random();

//server side
React.renderComponentToString(<MyComponent number={num}/>);

//pass num to client side
React.renderComponent(<MyComponent number={num}/>, document.body);
```

There are number of different ways to send the props used on the server to the client.

One of the easiest ways is to pass the initial props to the client is by way of a javascript object.

```
<!DOCTYPE html>
<html>
```

```
<!-- bundle contains MyComponent, React, etc -->
<script type="text/javascript" src="bundle.js"></script>
</head>
<body>
<!-- result of MyComponent server side render -->
<div data-reactid=".fgvrzhg2yo" data-react-checksum="-1663559667">
0.5820949131157249</div>

<!-- inject initial props used on the server side -->
<script type="text/javascript">
  var initialProps = {"num": 0.5820949131157249};
</script>

<!-- use initial prop from server -->
<script type="text/javascript">
  var num = initialProps.num;
  React.renderComponent(<MyComponent number={num}/>, document.body);
</script>
</body>
</html>
```

Asynchronous State

Many applications require data from a remote data source like a database or web service. On the client, this is not an issue the React.js component can show a loading view while waiting for the asynchronous data to return. On the server side, this behaviour can't be directly replicated with React.js as the render function is a synchronous function. In order to use asynchronous data, you need to fetch the data first then pass the data to the component at render time.

Example:

You may want to fetch the user record from an asynchronous store for use in the component.

AND

You want the state of the component after the user record has been fetched as what is rendered on the server side for SEO and performance reasons.

AND

You want the component to listen to changes on the client and re-render.

Problem: You can't use any of the component lifecycle methods to fetch the asynchronous data as React.renderComponentToString is synchronous.

Solution: Use a statics function to fetch your asynchronous data then pass it to the component to render. Pass the initialState to the client as props. Use component lifecycle methods to listen to changes and update the state, using the same statics function.

```
var Username = React.createClass({
      User.findById(props.userId)
        .then(function (user) {
          setState({user:user});
        })
        .catch(function (error) {
          setState({error: error});
        });
    }
  },
  //client and server
  componentWillMount: function () {
    if (this.props.initialState) {
      this.setState(this.props.initialState);
    }
  },
  //client side only
  componentDidMount: function () {
    //get async state if not in props
    if (!this.props.initialState) {
      this.updateAsyncState();
    }
    //listen to changes
    User.on('change', this.updateAsyncState);
  },
  //client side only
  componentWillUnmount: function () {
    //stop listening to changes
    User.off('change', this.updateAsyncState);
  },
  updateAsyncState: function () {
    //access static function from within the instance
    this.constructor.getAsyncState(this.props, this.setState);
  },
  render: function () {
    if (this.state.error) {
      return <div>{this.state.error.message}</div>;
    }
    if (!this.state.user) {
      return <div>Loading...</div>;
    }
    return <div>{this.state.user.username}</div>;
  }
});

//Render on the server

var props = {
  userId: 123 //could also be derived from route
};

Username.getAsyncState(props, function(initialState){
  props[initialState] = initialState;
  var result = React.renderComponentToString(Username(props));

  //send result to client along with initialState
});
```

Using the solution above - pre fetching of asynchronous data is only required on the server. On the changes on the client (e.g. html5 pushState or fragment change) should ignore any initialState from the server. You will also want to display loading messages etc as the data is fetched.

Isomorphic Routing

Routing is an essential part of any non trivial application. To render a React.js application with a router on the server side, you need to ensure the router supports the server side.

Fetching of asynchronous data is the job of a router and it's route controllers. Let's say a deeply nested component needs some asynchronous data. If the data is needed for SEO purposes then the responsibility for fetching the data should be moved higher up to the route controller and the data passed down to the deeply nested component. If it is not needed for SEO then it is ok to just fetch the data from within componentDidMount on the client. This is akin to classical ajax loaded data.

When looking at an isomorphic routing solution for React.js, make sure it either has asynchronous state supported or can be easily modified to support asynchronous state. Ideally you would also like the router to handle passing the initialState down to the client.

Singletons, Instances and Context

In the browser, your application is wrapped in a isolation bubble. Each instance of your application can't mix state with other instances, as each instance is usually on different computers or sandboxed on the same computer. This allows you to easily use singletons in your application architecture.

When you start moving your code to operate on the server, you need to be careful, as you may be having multiple instances of your application running at the same time, in the same scope. This has the potential that two instances of you application may mutate the state of a singleton, resulting in unwanted behaviour.

React.js render is synchronous, so you can reset all singletons used prior to rendering your application on the server side. You will have issues if asynchronous state requires use of the singletons also, you will need to account for this when fetching the asynchronous state for use in render.

Even with resetting singletons, prior to render, running your application in isolation will always be better. Packages like **contexify** allow you to run your code in isolation from one anther on the server. It is similar to using webworkers on the client.

The core development team of React.js discourage the use of passing context and instances down your component tree. As this makes your components less portable and changes to the dependencies of one component deep in your application have a flow on effects with all components up the component hierarchy. This in turn increases the complexity of your application, reducing maintainability as your application grows.

Serverside Testing

In Chapter 6 you learned about unit testing React.js components, where all of that testing was in the context of the “clientside”, ie the browser. But in this chapter, you will learn about techniques for using React.js components on the serverside, i.e. Nodejs, so it’s important that you learn how to test that part too. For these examples, we will be using the `mocha` testing framework. We could use the `jasmine-node` project too, but mocha is very popular in the node ecosystem for it’s excellent async support, so we’ll use that here.

NO RELIGIOUS WARS NEEDED

Both Jasmine and Mocha are excellent testing frameworks. You will be happy with either choice you make. The React.js codebase uses Jasmine for it’s tests (it actually wraps Jasmine in a project called Jest, which we’ll cover in Chapter 19), so we wanted to give you exposure to Jasmine. But Mocha is picking up a lot of momentum for node projects (and works very well paired with chai for assertions), so we wanted to give you exposure to Mocha too.

In the SurveyApp, the routing for the application uses `react-router` and these routes are used both on the clientside and the serverside. This dual use makes them “isomorphic” JavaScript code. Let’s walk through how we would test this routing code when used on the serverside. If you look at `client/app/app_router.js`, you will notice it does a few things:

1. `require`’s our application router:

```
var app_router = require("../../client/app/app_router");
```

2. Uses an express router as your middleware:

```
var router = require('express').Router({caseSensitive: true, strict: true});

...
router.use(function (req, res, next) {
...
});
```

3. Calls react-router with the URL:

```
Router.renderRoutesToString(app_router, req.originalUrl)
```

4. Defines a success handler to render the output HTML into your template:

```
var template = fs.readFileSync(__dirname + "/../../client/app.html", {encoding:'utf8'
...
```

```
      html = html.replace(/\{\{title\}\}/, data.title);
      res.status(data.httpStatus).send(html);

    }, ...);
```

To test this process, let's get started by making a spec file called `test/server/routing.test.js` and giving it the following boilerplate:

```
var request = require('supertest');
var app = require('../../server/server.js');

describe("serverside routing", function(){
});
```

Here's what is happening in our serverside spec boilerplate:

1. We are requiring our `server.js` node.js application.
2. We are using the `supertest` module. This will allow us to execute requests to our node server without having to boot up an actual server.
3. The `describe` block is a `mocha` function, not a jasmine function. We'll start to see differences with the mocha api in a bit.

To run our node tests, create the following file `test/server/main.js`:

```
require('./routing.test.js');
```

And then execute the following command: `npm run-script test-server`. You should see the following output:

```
tom:bleeding-edge-sample-app (master) $ npm run-script test-server
> bleeding-edge-sample-app@0.0.1 test-server /Users/tom/workspace/bleeding-edge-sample-a
> mocha test/server/main.js



  0 passing (5ms)
```

Great. Our boilerplate is fine and our Mocha test run doesn't have any obvious errors. Let's now add our first test, which should make a GET request to the node server for to the `/add_survey` endpoint:

```
...
describe("serverside routing", function(){
  it("should render the /add_survey path successfully", function(done){
    request(app)
      .get('/add_survey')
  });
});
```

This test looks *similar* to a Jasmine test, but there is one major difference - the `done` function. Do you notice how the anonymous function passed to `it` takes an argument which we call `done`. This is a callback function that we need to call when the test is "done" and we've made our assertions. In this test we are making a GET request to the "/add_survey" endpoint to the `server.js` app. We are then expecting a response code of `200`. And then add the end, we are calling `done` to tell Mocha our assertions have been called.

SUPERTEST

The `request`, `get`, `expect`, and `end` functions are all part of the supertest project. Feel free to read the supertest documentation here: `https://github.com/visionmedia/supertest.`

When we run this test, it should pass:

```
tom:bleeding-edge-sample-app (master) $ npm run-script test-server
> bleeding-edge-sample-app@0.0.1 test-server /Users/tom/workspace/bleeding-edge-sample-a
> mocha test/server/main.js



  serverside routing
    ✓ should render the AddSurvey component for the /add_survey path (87ms)


  1 passing (97ms)
```

Hooray! Now we should add some assertions about the actual content we get back.

BE CAREFUL CALLING DONE()

Don't write the above test like this. It won't work because the `done` function will be called before the `expect` code has run in the chain.

```
it("should render the /add_survey path successfully", function(done){
  request(app)
    .get('/add_survey')
    .expect("666");

    // DON'T DO THIS!!  This test will pass every time, even though the assertion is not
    done();
});
```

To validate that the `AddSurvey` React component is rendered correctly, we should make some assertions against the returned HTML. Let's first start by making some minor tweaks to our test:

```
...
  it("should render the AddSurvey component for the /add_survey path", function(done){

    request(app)
      .get('/add_survey')
      .expect(200)
      .end(function (err, res) {
        console.log("OUR HTML IS: " + res.text)
        done();
      });
  });
...
```

This should output something like this:

```
tom:bleeding-edge-sample-app (master) $ npm run-script test-server
> bleeding-edge-sample-app@0.0.1 test-server /Users/tom/workspace/bleeding-edge-sample-a
> mocha test/server/main.js



  serverside routing
HTML IS: <!DOCTYPE html>
<html>
  <head lang='en'>
    ...
    <title>Add Survey to SurveyBuilder</title>
    ...
  </head>
  <body>
  <div class="app" data-reactid=".qajmfw0740" data-react-checksum="2024417999">...</div>
  <script src="/build/bundle.js" type="text/javascript"></script>
  </body>
</html>
    ✓ should render the AddSurvey component for the /add_survey path (89ms)


  1 passing (100ms)
```

Very cool! We can see that our Node application is correctly rendering the `AddSurvey` component to string, and placing it in our template inside of the `body` tag. Now we *could* do basic string comparisons against `res.text`, but this would be very painful and brittle, so let's parse the HTML response and then make some assertions based on that. To parse the HTML, we will use a library called Cheerio. Cheerio is a node module that allows you to "load" some HTML and perform jQuery-like operations on it. So our test will look like this:

```
var cheerio = require('cheerio');
```

```
    request(app)
      .get('/add_survey')
      .expect(200)
      .end(function (err, res) {
        var doc = cheerio.load(res.text);
        expect(doc("title").html()).to.be("Add Survey to SurveyBuilder");
        expect(doc(".main-content .survey-editor").length).to.be(1);
        done();
      });
  });

  ...
```

In the `end` function we are doing the following:

1. Getting the html response from the `res.text`
2. Parsing it with a call to `cheerio.load`
3. Asserting on the innerHTML of the `<title>` element
4. Asserting the selector ".main-content .survey-editor" is on the page

This example should pass just fine! To get another example under our belt, this is a test to verify the 404 handler is working correctly:

```
  it("should render a 404 page for an invalid route", function(done){
    request(app)
      .get('/not-found-route')
      .expect(404)
      .end(function (err, res) {
        var doc = cheerio.load(res.text);
        expect(doc("body").html()).to.contain("The Page you were looking for isn&apos;t he
        done();
      });
  });
```

PREV
11. Performance Tuning

NEXT
13. Addons