The rdg expects its data in json format that follows:

{ Items : [{}, ...{}], NOfItems = 0,NOfPages = 0, CurrentPage = 0}

When one deals with the reactives in ASP.Net MVC she deals with 2 of them. The first one is primary

Controller. Your primary ASP.NET MVC controller should contain the Index action which respond with the View on which Rdg is located. It is its single goal.

After that rdg loads it data from the data controller. F.e. it can be something like ItemsConroller.



So, data controller action should return the viewModel of ReactDataGridViewModel type.

public class ReactDataGridViewModel<TDataItemModel>

{

public IEnumerable<TDataItemModel> Items { get; set; }

public int NOfItems { get; set; }

public int NOfPages { get; set; }

public int CurrentPage { get; set; }

}

For example, you can declare SalesGridViewModel: ReactDataGridViewModel<SalesModel>,

Where SalesModel is

Public class SalesModel {

Public int Id;

Public DateTime SalesDate;

…

}

Design

The spinner container has min-height = height of spinner image to show the spinner on initial loading (when no data displayed)

Rendering

The rendering is based on the principle of “minimalism”. It means that grid render function provides with the base (minimum) data grid DOM, i.e. table markup which consists of <thead> and <tbody> parts only. The detailed content is rendered by means of template functions – header and row templates. These template function are out of the grid component itself. They are developed by the data grid user.

Thus flexibility is provided.

Members overriding.

Members overriding is based on the fact that when you create the React class, like in the example

**var *ReactDataGrid*** = **React**.**createClass**({…}), the newly created object inherits from the anonymous construction object {…} via prototype



Fig.1

As you can see in fig.1, the object has only getDefaultProps function as its own one. All others are concentrated in the prototype. Prototype has ReactClassComponent type and includes all members from the {…} construction object.

It means that you can define the default members (f.e. loadErrorHandler function) in the

getDefaultProps: **function**() {  
 **return** {  
 **noDataMessage** : **"No data"**,  
 **defaultLoadParameters** : {**"page"** : 1, **"itemsOnPage"** : 16, **jumpToId** : **null**},  
 **loadErrorHandler** : **this**.**prototype**.**loadErrorHandler** }  
},

loadErrorHandler: **function** (xMLHttpRequest) {  
 alert(**"Status: "** + (xMLHttpRequest ? xMLHttpRequest.**status** : **"No info"**) + **" "** + (xMLHttpRequest ? xMLHttpRequest.**statusText** : **""**));  
},

and redefine it later when rendering:

rdcTesting.**reactDataGrid** = **ReactDOM**.render(  
 <**ReactDataGrid  
 url="/Items"  
 processHeadersRowFunc =** {*headerTemplate*}  
 **processDataRowFunc =** {*dataRowTemplate*}  
 **noDataMessage="No data"  
 reactDataGridClass="reactDataGrid"  
 tableClass="table table-striped"  
 spinnerClass="spinner"  
 idfield="Id"  
 loadParameters=**{{**"sortAsc"** : **false**, **"sortBy"**: **"Field0"**}}  
 **loadErrorHandler =** {*ownLoadErrorHandler*}  
 />,

Testing:

1. Karma starts. It launches the webpack as a preprocessor for the tests.webpack.js
2. Webpack processes each –test.js file from /src directory

Using \_Underscore library.

Due to React component is a state machine, there is a need to operate with the state parameters. The base grid operations like paging, sorting are based on the state.loadParameters. The fact is that we cannot control state parameters directly. Instead, we \_clone them, modify copy, and save copy into the state via setState.

Outer Events.

On the client events are raised by means of raiseEvent: function(eventHandler, eventArgs). F.e, the

onBeforeLoad event and onDataLoadedOK event are raised in the data loading function:

**this**.raiseEvent(**this**.**props**.onBeforeLoadData, **this**.**state**.**loadParameters**);

On the client, you can subscribe to the event in the render call:

rdcTesting.reactDataGrid = ReactDOM.render(

<ReactDataGrid

// obligatory parameters

...

// non obligatory parameters

onBeforeLoadData={onBeforeLoadData}

…,

Where

function onBeforeLoadData(component, eventArgs) {

rdcTesting.beforeLoadPars = eventArgs;

$("#loadParsBeforeLoading").html(beforeLoadPars.toString()));

}

So, you bind the events via props. The caller gets the react component in the first argument and state.loadParameters in the second one.

Context binding

You bind the context of the react component by means of bind function. See, f.e. the render function. You can do it in a single line:

render: function() {

return ( ...

tbody>

{

this.state.data.Items.map(function (row, idx) { return this.props.processDataRowFunc.call(this, row, idx);}.bind(this))

or in two lines:

render: function() {

var processDataRowFunc = this.props.processDataRowFunc.bind(this);

return ( ...

<tbody>

{

this.state.data.Items.map(function (row, idx) { return processDataRowFunc( row, idx);})