Contents

[Design 3](#_Toc454201758)

[Members overriding. 4](#_Toc454201759)

[Testing: 6](#_Toc454201760)

[Using \_Underscore library. 7](#_Toc454201761)

[Outer Events. 8](#_Toc454201762)

[Context binding 9](#_Toc454201763)

[ReactDataGrid Interface. 10](#_Toc454201764)

[Rerendering. 11](#_Toc454201765)

[The gridStructure parameter. 12](#_Toc454201766)

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 override them 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*}  
 />,

This unobvious fact creates the great possibility to hide the default behavior inside the component and override it at any time outside. F.e., you don’t have to define the row template function as trivial sequence of <td>s inside <tr> each time you render new grid. It is enough to set the [gridStructure](#gridStructure) parameter and ReactDataGrid renders the row by default. From the other hand, if you need more complicated logic for the data row (f.e., inline editing), you can set the data row template as parameter to render function thus overriding it.

# 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

Testing with Selenium.

You cannot run the jsx directly in the Selenium’s Driver.ExecuteJavaScript.

Try to execute Driver.ExecuteJavaScript( @"rdcTesting.reactDataGrid = ReactDOM.render(<ReactDataGrid … and you get the error “Unexpected token <”.

Instead you should include your test .jsx file in the Index.cshtml page.

After that you can execute jsx calling the js wrappers:

Driver.ExecuteJavaScript("rdcTesting.rerender();");

Where rdcTesting.rerender = **function**() {  
 rdcTesting.**reactDataGrid** = **ReactDOM**.**render**(  
 <**ReactDataGrid  
 url='/Items'…**

# 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.

# Jump To Id Functionality Support.

For sometimes, you need to stay on the same row when you when you apply for data. It means that you could select the row on the 2nd page with the id = 21. Then, for some reasons, you decided to reload the grid, but the total number of items has changed, so for to stay on the selected row (id = 21) you switch to the 3rd page and to land on the row with the id = 21. This functionality is supported in ReactDataGrid.

You only need to provide the jumpToId parameter in the loadParameters structure like in the example:

rdcTesting.**reactDataGrid**.**loadData**({**'jumpToId'** : jumpToId});

If you don’t care about selected row just set jumpToId to null.

This functionality is provided with tryToJumpToId and isIdInData functions in the

ReactDataGrid.jsx as well as jumpToId parameter in the controller’s action function.

# 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);})

# ReactDataGrid Interface.

You can control the ReactDataGrid component by calling its function in the following way:

rdcTesting.reactDataGrid.loadData({ page: 2 })

The list of functions:

loadData: function(loadParameters)

# Rerendering.

You can rerender the grid by simple calling js

You cannot do it with other frameworks.

# The gridStructure parameter.

There are two ways of telling to the ReactDataGrid how to render data row:

1. Define the data row template function. This way is the most flexible. At the same time, it demands from you the special jsx skills.
2. Define the gridStructure parameter. It is the simpliest way. All you need is to set the gridStructure parameter as array of js objects.e

Each of these objects describes the row field and has Header and Field members as obligatory and Sortable as conditional

[{Header: 'Id', Field: 'Id'},

{Header: 'Field0', Field: 'Field0', Sortable: false},

{Header: 'Field1', Field: 'Field1', Sortable: true}]

You can use js inside jsx by means of {} like in the headerTemplate:

processHeadersRowFunc: **function**() {  
 **var** sort = **this**.sort.bind(**this**);  
 **var** sortBy = **this**.**state**.**loadParameters**.**sortBy**;  
 **var** sortAsc = **this**.**state**.**loadParameters**.**sortAsc**;  
 **return** <**tr**>  
 {  
 **this**.gridStructure().map(**function** (val, idx) {  
 *//return <th key={idx} dangerouslySetInnerHTML={{\_\_html: (val["Sortable"] === true || val["Sortable"] === "true")?sortHeader( val["Header"], val["Field"]) : val["Header"]}}></th>* **return** <**th key=**{idx}>  
 {(val[**"Sortable"**] === **true** || val[**"Sortable"**] === **"true"**)?  
 <**span onClick=**{sort} **data-sort=**{val[**"Field"**]} **style=**{{**"cursor"**:**"pointer"**}}>  
 {val[**"Header"**]}  
 {(sortBy === val[**"Field"**])?(sortAsc?<**span**>↑</**span**> : <**span**>↓</**span**>) : <**span**></**span**>}  
 </**span**>  
  
 :<**span**>{val[**"Header"**]}</**span**>}  
 </**th**>  
 })  
 }  
 </**tr**>  
 },