QuickStart AG Grid Tutorial in React

An Introductory Guide to Using AG Grid with React

Niall Crosby

ag-grid.com

Contents

QuickStart React and AG Grid Tutorial

Welcome to the "QuickStart React and AG Grid Tutorial," created by Niall Crosby. The tutorial text, example source code, and walk-through videos will help you become proficient with AG Grid using React very quickly.

AG Grid is an Enterprise-level Data Grid that powers applications for most of the Fortune 500; it supports multiple frameworks with the same functionality and API due to its MVC architecture. To make the development experience for each framework seamless, we have rendering engines specific to the framework.

AG Grid Introduction

AG Grid has a 100% React Data Grid rendering engine, so you'll be able to use all the standard React development tools to profile and optimize your application. We have an entire section on optimizing the grid in "React Rendering."

AG Grid comes in two editions, the 100% free and MIT Licensed Community edition. You are free to use this in any commercial application, and we have applications showcased on our website that do precisely this.

- https://blog.ag-grid.com/showcase/

We also have an Enterprise edition if you need more features like pivoting, aggregation, Excel export, integrated charts, and custom toolbars.

We don't mind if you stick with the Community Edition or the Enterprise Edition; it's important to pick the best component that meets your needs. Both editions are fully featured and customizable with React Components, allowing you to create custom editors and renderers to show whatever you want in the grid cells.

Much of the grid is customizable by simple properties; as you'll see in the early sections, we can enable sorting, filtering, and pagination in seconds:

- `sortable: true`
- `filter: true`

```
{ field: 'make', sortable: true, filter: true },
```

And pagination is a grid property:

- `pagination: true`

```
const gridOptions = {
  columnDefs: columnDefs,
  pagination: true
};
```

But we don't want to show you too much code too quickly. We just want to give you a hint as to how much you can achieve with very little code.

We know that writing an interactive data grid or data table can be a fun and challenging project; after all, we've had a team of dedicated programmers working on AG Grid for over seven years.

We also know that it can be very hard. Creating a simple table to sort when you click a heading is relatively easy. Adding filtering, lots of data, high-frequency updates, data grouping, and column ordering becomes more challenging. Not forgetting all the other features that your users will want to see as your project matures.

It takes time to build and maintain a data grid component, so this tutorial will allow you to look at AG Grid and try out the community edition before writing your own. You'll be able to get up and running quickly and add business value to your users faster than writing and maintaining a custom-built data grid.

What's a Data Grid?

A data grid is a tabular rendering of data in a web app, like embedding an Excel sheet on a page.

Data grids differ from data tables in several ways:

- users don't expect tables to be interactive
- tables grow and shrink based on the contents, whereas a data grid is a fixed size leading to a consistent user interface

Standard features you would expect to find in a Data Grid are:

- sorting
- multi-column sorting
- filtering
- drag and drop row and column ordering
- pagination
- scroll bars for data to keep grid size consistent
- cell editing
- column and row virtualization
- row selection

- column grouping
- row and column pinning

All of the above, with a lot more features, are available in the community edition of AG Grid and configurable through simple properties, so it should be powerful enough to meet your needs for a data grid. And if you discover you need a custom filter, editor, or renderer, you can create a custom React Component to implement your domain feature, as we will explain in detail in this tutorial.

A Data Grid also differs from a Data Table in that a Table is built on top of an HTML `table` element, whereas a Data Grid will often manipulate the DOM directly and is built on top of `div` elements. This can simplify the DOM virtualization to reduce memory overhead and improve performance for high-frequency updates.

If you are interested in seeing if AG Grid meets your long term needs, then we have a list of features in the documentation:

- https://ag-grid.com/javascript-data-grid/licensing/

About the Tutorial Author

The tutorial was created by Niall Crosby, CEO of AG Grid and the initial developer. Niall has remained hands-on with AG Grid since it started and was heavily involved with creating the React Rendering Engine, so you'll be learning from someone who knows all the details of AG Grid.

You can find Niall online:

- https://twitter.com/niallcrosby
- https://linkedin.com/in/niallcrosby
- https://blog.ag-grid.com/author/niall/

Introduction To Tutorial

This text contains a self-guided tutorial to help you quickly get started with AG Grid and React. Through a series of text content explaining the incremental development of code, it walks you through the basic concepts that help you get the most from AG Grid with React.

We can't cover everything in this text. Niall has pulled out the most critical concepts to help you get started and cover the most common use-cases. Still, we recommend you read the documentation because the grid API is rich and very flexible.

The AG Grid documentation is online at:

- https://www.ag-grid.com/react-data-grid/

AG Grid supports multiple frameworks, and you can view the documentation for different frameworks by clicking the framework icon.

There are plenty of embedded examples in the documentation that are runnable or click to view and edit them online to help you experiment without installing anything.

All the source code for this tutorial is on Github.

The repo 'react-data-grid' contains all the React tutorials and examples used on our blog:

- https://github.com/ag-grid/react-data-grid

The folder for this tutorial is:

- https://github.com/ag-grid/react-data-grid/tree/main/getting-started-video-tutorial

You can find the source code in the `src` sub-folder.

Each chapter is in its own folder as stand-alone projects.

Installing Prerequisites

Node.js

Pre-requisites are having Node installed because we will use npx to install the packages and we will use the `create-react-app` to create a simple application structure.

You can find instructions and downloads for installling Node.js on nodejs.org

- https://nodejs.org/en/download/

IDE

We typically recommend Visual Studio Code as a good IDE because it is free and easy to install for each platform.

- https://code.visualstudio.com/

Let's Go

Now. Let's get started.

QuickStart Project Setup

In this section you'll learn how to create a project, add AG Grid. You'll understand what CSS is used to style and structure the grid. How to use state and effect to manage data in the grid while loading data from the server. Sorting and Filtering is demonstrated through specific and default column properties. Grid Properties are used to animate rows and support selection. Adding functionality to the grid can be implemented by hooking into the grid events and using the API.

Quick Start Guide Video

This video is a quick start tutorial to getting started with AG Grid in a React project. Niall Crosby, the CEO and creator of AG Grid walks you through the basic knowledge needed to work with AG Grid using React.

https://youtu.be/Pr__B6HM_s4

- 00:00 Create a Project
- 00:30 Get Started
- 01:05 Create an AG Grid
- 02:05 CSS Includes Explained
- 03:11 Using React State
- 03:29 Loading data from server with useEffect
- 03:55 Sorting and Filtering Columns
- 04:35 Default Column Definitions
- 04:55 Grid properties to animate and select rows
- 05:33 hooking into Grid Events
- 06:00 using the Grid API
- 06:42 Summary
- 07:03 Outro

Source Code For This Section

The source code for this section is available in Github in the 'react-data-grid' repo and the subfolder `getting-started-video-tutorial/src/001-quickstart-guide/`:

- https://github.com/ag-grid/react-data-grid/tree/main/getting-started-video-tutorial/src/001-quickstart-guide

Create a Project

Create a project:

```
npx create-react-app hello
```

```
cd hello
```

Install AG Grid

```
npm install --save ag-grid-community ag-grid-react
```

- ag-grid-community
  - the community edition of AG Grid
- ag-grid-react
  - the React Rendering Engine

`--save` will update the **`package`**`.json` file

Start the project with:

```
npm run start
```

Render aa Simple Grid

Change the `App.js` file:

```
import './App.css';

import { AgGridReact } from 'ag-grid-react';
import 'ag-grid-community/dist/styles/ag-grid.css';
import 'ag-grid-community/dist/styles/ag-theme-alpine.css';

function App(){
   const [rowData] = [
       {make: "Toyota", model: "Celica", price: 35000},
       {make: "Ford", model: "Mondeo", price: 32000},
       {make: "Porsche", model: "Boxter", price: 72000}
   ];

   const [columnDefs] = [
       { field: 'make' },
       { field: 'model' },
       { field: 'price' }
   ];
```

```
    return (
        <div className="ag-theme-alpine" style={{height: 500, width: 500}}>
            <AgGridReact
                rowData={rowData}
                columnDefs={columnDefs}>
            </AgGridReact>
        </div>
    );
}

export default App;
```

Notes:

- `ag-grid.css` is the structural CSS
- `ag-theme-alpine.css` is the styling CSS
- Make sure to give the grid a width and height

    - `style={{height: 500, width: 500}}`

- Column Definitions define the column properties
- rowData is the data to render
- the data and column definitions are applied to the Grid as properties

Reactify it with useState

Adding the data as state allows it to be changed easily.

```
import './App.css';

import { AgGridReact } from 'ag-grid-react';
import 'ag-grid-community/dist/styles/ag-grid.css';
import 'ag-grid-community/dist/styles/ag-theme-alpine.css';

import {useState} from 'react';


function App(){
    const [rowData, setRowData] = useState([
        {make: "Toyota", model: "Celica", price: 35000},
        {make: "Ford", model: "Mondeo", price: 32000},
        {make: "Porsche", model: "Boxter", price: 72000}
    ]);

    const [columnDefs, setColumnDefs] = useState([
        { field: 'make' },
        { field: 'model' },
```

```
        { field: 'price' }
    ]);

    return (
        <div className="ag-theme-alpine" style={{height: 500, width: 500}}>
            <AgGridReact
                rowData={rowData}
                columnDefs={columnDefs}>
            </AgGridReact>
        </div>
    );
}

export default App;
```

Load Data From Server

Add a `useEffect` to load data from a url

```
    useEffect(() => {
        fetch('https://www.ag-grid.com/example-assets/row-data.json')
            .then(result => result.json())
            .then(rowData => setRowData(rowData))
    }, []);
```

Adding this means we no longer need to create default state for `rowData`, remember to add `useEffect` to the react imports:

```
import {useState, useEffect} from 'react';
```

Leading to the final code:

```
import './App.css';

import { AgGridReact } from 'ag-grid-react';
import 'ag-grid-community/dist/styles/ag-grid.css';
import 'ag-grid-community/dist/styles/ag-theme-alpine.css';

import {useState, useEffect} from 'react';


function App(){
    const [rowData, setRowData] = useState();

    const [columnDefs, setColumnDefs] = useState([
        { field: 'make' },
        { field: 'model' },
        { field: 'price' }
    ]);
```

```
    useEffect(() => {
        fetch('https://www.ag-grid.com/example-assets/row-data.json')
            .then(result => result.json())
            .then(rowData => setRowData(rowData))
    }, []);

    return (
        <div className="ag-theme-alpine" style={{height: 500, width: 500}}>
            <AgGridReact
                rowData={rowData}
                columnDefs={columnDefs}>
            </AgGridReact>
        </div>
    );
}

export default App;
```

Sorting and Filtering

Add Sorting and Filtering to a column with column properties:

```
    const [columnDefs, setColumnDefs] = useState([
        { field: 'make', sortable: true, filter: true },
        { field: 'model' },
        { field: 'price' }
    ]);
```

Above code makes the ‘make’ column sortable and filterable with a text filter.

Column properties are documented in the documentation:

https://www.ag-grid.com/react-data-grid/column-properties/

Default Column Definitions

Default column definitions create a set of properties to be added to every column.

```
const defaultColDef = useMemo( ()=> (
    {
        sortable: true,
        filter: true
    }
))
```

Note: because this won’t change, it has been memoized.

There is no longer any need to have sortable and filter properties on the column definitions.

```
    const [columnDefs, setColumnDefs] = useState([
        { field: 'make' },
        { field: 'model' },
        { field: 'price' }
    ]);
```

We do have to add the default Column Definitions to the grid properties:

```
            <AgGridReact
                rowData={rowData}
                columnDefs={columnDefs}>
                defaultColDef={defaultColDef}
            </AgGridReact>
```

And remember to add `useMemo` to the react imports.

```
import './App.css';

import { AgGridReact } from 'ag-grid-react';
import 'ag-grid-community/dist/styles/ag-grid.css';
import 'ag-grid-community/dist/styles/ag-theme-alpine.css';

import {useState, useEffect, useMemo} from 'react';


function App(){
    const [rowData, setRowData] = useState();

    const [columnDefs, setColumnDefs] = useState([
        { field: 'make' },
        { field: 'model' },
        { field: 'price' }
    ]);

    useEffect(() => {
        fetch('https://www.ag-grid.com/example-assets/row-data.json')
            .then(result => result.json())
            .then(rowData => setRowData(rowData))
    }, []);

    return (
        <div className="ag-theme-alpine" style={{height: 500, width: 500}}>
            <AgGridReact
                rowData={rowData}
                columnDefs={columnDefs}>
                defaultColDef={defaultColDef}
            </AgGridReact>
        </div>
    );
```

```
}

export default App;
```

Grid Properties

Two useful Grid properties:

- `animateRows` to animate rows when sorting
- `rowSelection` to make rows selectable

```
            <AgGridReact
                 rowData={rowData}
                 columnDefs={columnDefs}>
                 animateRows={true}
                 rowSelection='multiple'
                 defaultColDef={defaultColDef}
             </AgGridReact>
```

Grid properties are documented in the documentation:

https://www.ag-grid.com/react-data-grid/grid-properties/

Listening to Grid Events

Create a listener:

```
const cellClickedListener = useCallback(event => {
    console.log('cellClicked', event);
}, []);
```

This has been wrapped in a callback hook to create a memoized version.

The listener is then bound to the grid with a grid property:

```
onCellClicked={cellClickedListener}
```

To setup the grid as follows:

```
            <AgGridReact
                 rowData={rowData}
                 columnDefs={columnDefs}>
                 animateRows={true}
                 rowSelection='multiple'
                 defaultColDef={defaultColDef}
                 onCellClicked={cellClickedListener}
             </AgGridReact>
```

Remember to add `useCallback` to the imports:

```
import {useState, useEffect, useMemo, useCallback} from 'react';
```

Using the API

The grid has an internal API which we can use programmatically, in this example we will add it as a reference using `useRef`.

```
import {useState, useEffect, useMemo, useCallback, useRef} from 'react';
```

And store it in a const:

```
const gridRef = useRef();
```

Allocating it to a Grid property:

```
          <AgGridReact
               ref={gridRef}
               rowData={rowData}
               columnDefs={columnDefs}>
               animateRows={true}
               rowSelection='multiple'
               defaultColDef={defaultColDef}
               onCellClicked={cellClickedListener}
           </AgGridReact>
```

We can use `gridRef` in code to call the API, e.g. from a button click:

```
const buttonListener = useCallback( e =>{
    gridRef.current.api.deselectAll();
}, []);
```

And make sure the button appears in the HTML:

```
   return (
       <div>
            <button onClick={buttonListener}>Push Me</button>
            <div className="ag-theme-alpine" style={{height: 500, width:
               500}}>
                <AgGridReact
                    ...
                </AgGridReact>
            </div>
        </div>
   );
```

When the button is pressed, any selected items in the grid will be deselected.

Documentation for the Grid API is online:

https://www.ag-grid.com/react-data-grid/grid-api/

Summary

This is an overview of key points for working with AG Grid:

- creating and styling a grid
- configuring columns and using default Column Definitions
- configuring the grid with grid properties
- changing data in the grid and loading data from JSON
- listening to grid events
- working with the grid api

Enterprise Overview

In this section you will learn how to enable the enterprise features of AG Grid. These are free to trial and require a license to use in production. All enterprise features are listed in the documentation.

Enterprise Overview Video

In the video Niall Crosby demonstrates how to enable enterprise features and the Row Grouping.

https://youtu.be/pKUhYE1VTP4

- 00:00 Setting up Enterprise
- 02:08 Row Grouping
- 02:30 User Controlled Grouping
- 03:13 Enterprise Features Listed
- 03:33 Enterprise License

Source Code For This Section

The source code for this section is available in Github in the 'react-data-grid' repo and the subfolder `getting-started-video-tutorial/src/002-enterprise-overview`:

- https://github.com/ag-grid/react-data-grid/tree/main/getting-started-video-tutorial/src/002-enterprise-overview

Starting Code

A basic project which setups a grid and loads data from a json file on the server.

```
import './App.css';

import {AgGridReact} from 'ag-grid-react';
import 'ag-grid-community/dist/styles/ag-grid.css';
import 'ag-grid-community/dist/styles/ag-theme-alpine.css';

import 'ag-grid-enterprise';

import {useState, useRef, useEffect, useMemo, useCallback} from 'react';

function App() {

  const gridRef = useRef();
```

```
  const [rowData, setRowData] = useState();
  const [columnDefs, setColumnDefs] = useState([
        { field: 'athlete' },
        { field: 'age' },
        { field: 'country' },
        { field: 'year' },
        { field: 'date' },
        { field: 'sport' },
        { field: 'gold' },
        { field: 'silver' },
        { field: 'bronze' },
        { field: 'total' }
  ]);
  const defaultColDef = useMemo( ()=> ( {
      sortable: true,
      filter: true,
  }), []);

  useEffect(() => {
    fetch('https://www.ag-grid.com/example-assets/olympic-winners.json')
    .then(result => result.json())
    .then(rowData => setRowData(rowData))
  }, []);

  return (
    <div className="ag-theme-alpine" style={{height: '100%'}}>
      <AgGridReact ref={gridRef}
          rowData={rowData} animateRows={true}
          columnDefs={columnDefs} defaultColDef={defaultColDef}
          />
    </div>
  );
}

export default App;
```

The dependencies currently only include `ag-grid-community` and `ag-grid-react`.

We can install the enterprise package using npm

```
npm install --save ag-grid-enterprise
```

This will add `ag-grid-enterprise` into the **`package.json`** file:

```
    "ag-grid-community": "^27.0.1",
    "ag-grid-enterprise": "^27.0.1",
    "ag-grid-react": "^27.0.1",
```

All are needed to use enterprise. The `ag-grid-community` has the base features and `ag-grid-enterprise` adds the enterprise features like grouping, pivoting, aggregation, charts, and excel

Export.

Finally to make the application use the Enterprise features we need to import the library into our code.

By adding:

```
import 'ag-grid-enterprise';
```

Row Grouping

We can add row grouping as a column definition property:

```
        { field: 'country', rowGroup: true },
```

This would group all rows by specific countries.

We can add row grouping for multiple columns:

```
        { field: 'country', rowGroup: true },
         { field: 'year', rowGroup: true },
```

This would group by Country and then year e.g.

```
> United States (1109)
v Russia (706)
   > 2000 (168)
   v 2012 (129)
       ...
```

The cells are expandable to show the data in the group.

User Controlled Grouping

The above code defines default grouping. It is also possible to allow the user to configure the grouping.

To do that, remove the column definition `rowGroup` properties and add a default column definition property `enableRowGroup`.

```
 const defaultColDef = useMemo( ()=> ( {
      sortable: true,
      filter: true,
      enableRowGroup: true
  }), []);
```

It is possible to add this property to individual columns as some might not be suitable for grouping but by adding it to the default column definition the user can group by all columns by dragging the columns into a drop zone.

We can enable the drop zone with a grid property `rowGroupPanelShow='always'`.

This is shown in context below:

```
        <AgGridReact ref={gridRef}
            rowGroupPanelShow='always'
            rowData={rowData} animateRows={true}
            columnDefs={columnDefs} defaultColDef={defaultColDef}
            />
```

Enterprise License

Enabling the enterprise features causes a watermark to appear on the grid and a set of warnings to appear in the console.

These can be removed by purchasing a license.

You can use the community edition of AG Grid free of charge in commercial applications.

You can evaluate the enterprise features of AG Grid, simply by enabling them in the grid, you need a license to develop and release enterprise features to production.

You can trial AG Grid Enterprise for free, without asking AG Grid. If you need a trial license to remove the warning message or watermark then contact info@ag-grid.com

You can find information on the enterprise features in the documentation:

https://www.ag-grid.com/vue-data-grid/licensing/

Customizing Cells

Customizing cells using React Cell Renderers, these are simple functions that return JSX to add your own React component to format and add interactivity to the cells in a data grid. These can be created as functional components, anonymous inline functions or class components.

Tutorial Video

https://youtu.be/9IbhW4z–mg

00:00 React Components in Cells 00:27 Create a Cell Renderer Component 01:40 Adding interactivity 02:32 Reusing Components 03:03 Cell Renderer Params 04:04 Inline Components 04:42 Class Components 05:38 Cell Renderer Selector 07:00 Summary

Source Code For This Section

The source code for this section is available in Github in the 'react-data-grid' repo and the subfolder `getting-started-video-tutorial/src/003-customizing-cells`:

- https://github.com/ag-grid/react-data-grid/tree/main/getting-started-video-tutorial/src/003-customizing-cells

Starting Code

The starting code uses a variant of the olympic winners code from the Enterprise Quickstart, with the enterprise features removed.

This has column definitions which match data from a JSON file retrieved from a server.

```
import './App.css';

import {AgGridReact} from 'ag-grid-react';
import 'ag-grid-community/dist/styles/ag-grid.css';
import 'ag-grid-community/dist/styles/ag-theme-alpine.css';

import {useState, useRef, useEffect, useMemo, useCallback} from 'react';

function App() {

  const gridRef = useRef();
  const [rowData, setRowData] = useState();
  const [columnDefs, setColumnDefs] = useState([
        { field: 'athlete' },
```

```
        { field: 'age' },
        { field: 'country' },
        { field: 'year' },
        { field: 'date' },
        { field: 'sport' },
        { field: 'gold' },
        { field: 'silver' },
        { field: 'bronze' },
        { field: 'total' }
  ]);
  const defaultColDef = useMemo( ()=> ( {
      sortable: true,
      filter: true
  }), []);

  useEffect(() => {
    fetch('https://www.ag-grid.com/example-assets/olympic-winners.json')
    .then(result => result.json())
    .then(rowData => setRowData(rowData))
  }, []);

  return (
    <div className="ag-theme-alpine" style={{height: '100%'}}>
      <AgGridReact ref={gridRef}
          rowData={rowData} animateRows={true}
          columnDefs={columnDefs} defaultColDef={defaultColDef}
          />
    </div>
  );
}

export default App;
```

Create a Cell Renderer Component

To demonstrate a simple cell renderer, create a functional component which renders “Hello World” in the cell.

```
const SimpleComp = p => <>Hello World!</>
```

Then configure the `athlete` column to use this cell renderer:

```
    { field: 'athlete', cellRenderer: SimpleComp },
```

This will cause all Athlete cells to show “Hello World!”

Understanding Cell Renderer Params

Full details of the cell renderer parameters can be found in the documentation:

https://www.ag-grid.com/react-data-grid/component-cell-renderer/

But you can experiment yourself by logging the parameters to the console, this can be a useful way to experiment with the grid without consulting the docs all the time.

```
const SimpleComp = p => {
    console.log(p);
    return <>Hello World!</>
}
```

The two most used properties for cell renderers are:

- `value` which contains the value of the cell's row data
- `data` which provides access to the raw data for each cell in the row

You also have access to the grid and column APIs and a variety of other objects. We provide you with all the information you'll need to create custom cell renderers that are heavily configurable.

We can return the cell to the default behavior by returning the value:

```
const SimpleComp = p => {
    return <>{p.value}</>
}
```

This is a good basis from which to build a more complicated cell renderer.

Adding interactivity

The cell renderer can return any JSX we want so we can easily add interactivity by adding some buttons to render data.

```
const SimpleComp = p => {
    const onDollar = useCallback( ()=> window.alert('Dollar ' + p.value));
    const onAt = useCallback( ()=> window.alert('At ' + p.value));
    return (
        <>
            <button onClick={onDollar}>$</button>
            <button onClick={onAt}>@</button>
            {p.value}
        </>);
}
```

Every cell would now have two additional buttons, clicking the button would show the appropriate alert.

Reusing Components

The same cell renderer can be used on multiple cells, e.g.

```
  const [columnDefs, setColumnDefs] = useState([
        { field: 'athlete', cellRenderer: SimpleComp },
        { field: 'age', cellRenderer: SimpleComp },
        { field: 'country', cellRenderer: SimpleComp },
...
```

Three cells would now have the extra buttons which trigger alerts to show the value, the `athlete`, `age` and `country` columns.

The cell renderer could also be used on the default column to make it easy to style every cell.

```
  const defaultColDef = useMemo( ()=> ( {
      sortable: true,
      filter: true,
      cellRenderer: SimpleComp
  }), []);
```

Cell renderers added to default column definition can be overridden by individual cells by setting the `cellRenderer` to **`null`**, or a different cell renderer e.g.

```
    { field: 'athlete', cellRenderer: null },
```

Cell Renderer Params

The example cell renderer has used the default parameters passed from the grid, it is also possible to pass addition params to the cell renderer like any React component.

If the cell render was changed as follows, the the value of the `buttonText` para would be shown as the button text rather than "@"

```
const SimpleComp = p => {
    const onAt = useCallback( ()=> window.alert('At ' + p.value));
    return (
        <>
            <button onClick={onAt}>{p.buttonText}</button>
            {p.value}
        </>);
}
```

The param can be set in the column definition code as follows using the `cellRendererParams` property:

```
  const [columnDefs, setColumnDefs] = useState([
```

```
        {
            field: 'athlete',
            cellRenderer: SimpleComp,
            cellRendererParams: {
                buttonText: '='
            }
        },
```

With the above configuration the button would display “=” rather than “@”.

This allows the component to be re-used but have different rendering each time e.g.

```
  const [columnDefs, setColumnDefs] = useState([
        {
            field: 'athlete',
            cellRenderer: SimpleComp,
            cellRendererParams: {
                buttonText: '='
            }
        },
        {
            field: 'age',
            cellRenderer: SimpleComp,
            cellRendererParams: {
                buttonText: '#'
            }
        },
```

The `athlete` cells would have a button showing “=” and the `age` cells would have a buttons showing “#” and both buttons would have the same functionality because they use the same cell renderer.

At this point our component has been outside the main grid definitions.

```
const SimpleComp = p => {
    const onAt = useCallback( ()=> window.alert('At ' + p.value));
    return (
        <>
            <button onClick={onAt}>{p.buttonText}</button>
            {p.value}
        </>);
}

function App() {

  const gridRef = useRef();
  const [rowData, setRowData] = useState();
...
```

It is also possible to inline the renderer alongside the column definition or create class components.

Inline Components

For simple use-cases, or just for experimentation when developing, it is possible to inline the component in the column definition.

```
 { field: 'age', cellRenderer: p => <><b>Age is: </b>{p.value}</> },
```

This will render **Age is:** (in bold) followed by the cell value.

e.g.

Age
Age is: 23
Age is: 19

Class components

To create a class `Component` we will need to import `Component`

```
import {useState, useRef, useEffect, useMemo, useCallback, Component} from
    'react';
```

And then create a class that extends the component

```
class PullComp extends Component {
  render() {
    return (
      <>
        <button onClick={ ()=> window.alert('Pull')}>Pull</button>
        {this.props.value}
      </>);
  }
}
```

This is added to the `country` column definition the same way as the earlier component.

```
        { field: 'country', cellRenderer: PushComp },
```

To keep the components consistent we'll rename the `SimpleComp` to `PushComp`

```
const PushComp = p => {
  const onAt = useCallback( ()=> window.alert('Push'));
  return (
    <>
      <button onClick={onAt}>Push</button>
```

```
      {p.value}
    </>);
}
```

Then our column definitions are using a functional component, an inline component and a class component:

```
  const [columnDefs, setColumnDefs] = useState([
        { field: 'athlete', cellRenderer: PushComp },
        { field: 'age', cellRenderer: p => <><b>Age is: </b>{p.value}</>
           },
        { field: 'country', cellRenderer: PushComp },
...
```

Cell Renderer Selector

If we wanted to make the component rendering dependent upon the data then we can do that using a cell renderer selector.

```
    { field: 'year',
        cellRendererSelector: p => {
        if (p.value==2000) {
            return {component: PushComp, params: {}};
        }
        if (p.value==2004) {
            return {component: PullComp};
        }
        }
    },
```

In the above code the `year` cell will render with a different component depending on the value of the cell data.

The `cellRendererSelector` is the function which determines which cell render to use.

The `component` is the reference to the chosen component.

And `params` is the optional mechanism for passing parameters into the chosen cell renderer component.

Summary

In this Getting started with React Cell Renderers section you learned:

- what is a cell renderer
- creating a cell renderer inline, as a function and as a class

- using `cellRendererSelector` to conditionally choose a cell render
- what the params and `props` for a cell renderer are

React Rendering

AG Grid has a 100% React Rendering Engine, and in this section we will demonstrate that by looking at the controlled rendering for AG Grid.

Tutorial Video

https://youtu.be/oAQ5vavDupU

00:00 AG Grid is 100% React 00:10 Starting Code Explained 00:25 Custom React Component 00:54 Developer Tools Rendering View 01:56 Wasted Renders Demonstration 03:00 Avoiding Waster Renders with memo 03:28 Summary

Source Code For This Section

The source code for this section is available in Github in the 'react-data-grid' repo and the subfolder `getting-started-video-tutorial/src/004-react-rendering`:

- https://github.com/ag-grid/react-data-grid/tree/main/getting-started-video-tutorial/src/004-react-rendering

Starting Code

Our starting point is a simple app which renders data from a server side JSON file in a grid.

```
import './App.css';

import {AgGridReact} from 'ag-grid-react';
import 'ag-grid-community/dist/styles/ag-grid.css';
import 'ag-grid-community/dist/styles/ag-theme-alpine.css';

import {useState, useRef, useEffect, useMemo, useCallback} from 'react';


function App() {

  const gridRef = useRef();
  const [rowData, setRowData] = useState();
  const [columnDefs, setColumnDefs] = useState([
        { field: 'athlete' },
        { field: 'age' },
        { field: 'country' },
        { field: 'year' },
```

```
        { field: 'date' },
        { field: 'sport' },
        { field: 'gold' },
        { field: 'silver' },
        { field: 'bronze' },
        { field: 'total' }
  ]);
  const defaultColDef = useMemo( ()=> ( {
      sortable: true,
      filter: true,
  }), []);

  useEffect(() => {
    fetch('https://www.ag-grid.com/example-assets/olympic-winners.json')
    .then(result => result.json())
    .then(rowData => setRowData(rowData))
  }, []);

  return (
    <div className="ag-theme-alpine" style={{height: '100%'}}>
      <AgGridReact ref={gridRef}
          rowData={rowData} animateRows={true}
          columnDefs={columnDefs} defaultColDef={defaultColDef}
          />
    </div>
  );
}

export default App;
```

To demonstrate the rendering we will create a custom React Component that renders a spinning loading gif to the left of the value of the cell.

The video uses the following code.

```
function MyComp = params => {
    const imageUrl = ="https://d1yk6z6emsz7qy.cloudfront.net/static/images
       /loading.gif" ;
    const imgStyle = {width: 40, top: 0, left:0, position: 'absolute'}
    const style = {marginLeft:20 }
    return (
        <span style={style}>
            <img src={imageUrl} style={imgStyle} />
            {params.value}
        </span>
    );
}
```

But you want to simplify the code then you can use a simple value cell renderer.

```
const MyComp = params => {
```

```
  return (
    <>{params.value}</>
  );
};
```

And then have the `age` column use that cell renderer:

```
{ field: 'age', cellRenderer:MyComp },
```

If we view the React hierarchy in the dev tools we will see that the Grid is 100% React.

Wasted Renders

To demonstrate avoiding wasted renders, we will use a simpler cell renderer that counts the number of times a cell has been rendered.

```
const MyComp = params => {
  const renderCountRef = useRef(1);
  return (
    <><b>({renderCountRef.current++})</b> {params.value}</>
  );
};
```

We can use this in all the cells by adding this to the default column definition.

```
  const defaultColDef = useMemo( ()=> ( {
      sortable: true,
      filter: true,
      cellRenderer: MyComp
  }), []);
```

If the columns of the grid were to be re-ordered manually then would see that the cells were being re-rendered with each column move. This is because we haven't optimized our component to avoid wasted renders.

The grid needs to re-render because the columns have moved, but the cells have not changed value so they should not need to be re-rendered.

Avoiding Wasted Renders

If we memoize the column definition then we will avoid a lot of re-renders.

```
  const defaultColDef = useMemo( ()=> ( {
      sortable: true,
      filter: true,
      cellRenderer: memo(MyComp)
```

```
  }), []);
```

This requires importing `memo` from `react`:

```
import {useState, useRef, useEffect, useMemo, useCallback, memo} from '
    react';
```

This additional `memo` will prevent the cell from being re-rendered if the value has not changed.

React Filters

Columns can be filtered to allow the user to configure the data shown in the grid. Three filters are supplied with AG Grid community edition: text, number and date. An additional set and multi filter are available to the enterprise edition. The filters can also be configured through the API.

https://www.ag-grid.com/react-data-grid/filtering/

Video Tutorial

https://youtu.be/pebXUHUdlos

00:00 about filters 00:20 default filters 00:55 text, number and date filters 02:38 filter parameters 03:55 filter buttons 05:45 filtering by dates 06:30 filter state models 09:18 floating filters 10:10 Enterprise filters 12:43 Filter Menu Container 13:35 Summary

Source Code For This Section

The source code for this section is available in Github in the 'react-data-grid' repo and the subfolder `getting-started-video-tutorial/src/005-react-filters`:

- https://github.com/ag-grid/react-data-grid/tree/main/getting-started-video-tutorial/src/005-react-filters

Starting Code

Our starting point is a simple app which renders data from a server side JSON file in a grid.

```
import './App.css';

import {AgGridReact} from 'ag-grid-react';
import 'ag-grid-community/dist/styles/ag-grid.css';
import 'ag-grid-community/dist/styles/ag-theme-alpine.css';

import {useState, useRef, useEffect, useMemo, useCallback} from 'react';


function App() {

  const gridRef = useRef();
  const [rowData, setRowData] = useState();
  const [columnDefs, setColumnDefs] = useState([
        { field: 'athlete' },
```

```
        { field: 'age' },
        { field: 'country' },
        { field: 'year' },
        { field: 'date' }
  ]);
  const defaultColDef = useMemo( ()=> ( {
      flex: 1
  }), []);

  useEffect(() => {
    fetch('https://www.ag-grid.com/example-assets/olympic-winners.json')
    .then(result => result.json())
    .then(rowData => setRowData(rowData))
  }, []);

  return (
    <div className="ag-theme-alpine" style={{height: '100%'}}>
      <AgGridReact ref={gridRef}
          rowData={rowData} animateRows={true}
          columnDefs={columnDefs} defaultColDef={defaultColDef}
          />
    </div>
  );
}

export default App;
```

Because we are using a small number of columns, `flex: 1` will spread the columns across the width of the grid.

Filtering by Column

To filter by as single column, add the filter property to the column definition:

```
        { field: 'athlete', filter: true },
```

The user can then click the hamburger menu in the column header to choose a text filter.

The value **`true`** can be changed to be a specific filter name, or a custom filter that we code ourselves.

`filter: true` is equivalent to writing:

```
        { field: 'athlete', filter: 'agTextColumnFilter' },
```

`agTextColumnFilter` is the default text filter and is used when the `filter` is set to **`true`**.

Built-in Filters

AG Grid comes with 5 built-in filters.

Documentation for these can be found online: https://www.ag-grid.com/react-data-grid/filter-provided-simple/

The three built-in filters for the Community edition are:

- `agTextColumnFilter` : Text Filter
- `agNumberColumnFilter` : Number Filter
- `agDateColumnFilter` : Date Filter

To add these filters we amend the column definitions:

```
  const [columnDefs, setColumnDefs] = useState([
        { field: 'athlete', filter: `agTextColumnFilter` },
        { field: 'age', filter: `agNumberColumnFilter` },
        { field: 'country' },
        { field: 'year' },
        { field: 'date', filter: `agDateColumnFilter` }
  ]);
```

The 'date' filter does not work on the data in the grid because we are retrieving `String` values and adding those to the grid. We need to convert them into dates for the `agDateColumnFilter` filter to work, as we will see later.

Customizing the Filters with Filter Params

The filters can be customised by adding a `filterParams` object to the column definition:

```
        { field: 'athlete',
            filter: `agTextColumnFilter`,
            filterParams:{
            }
        },
```

The actual parameters are added as properties of the `filterParams` object.

`filterParams` allow us to add buttons, and debounce the filter.

debounceMs

The column filters, filter the content as the user types the information into the filter. A debounce can help improve the user experience for some data.

A debounce millisecond time can be set as a Filter Parameter:

```
        { field: 'athlete',
            filter: `agTextColumnFilter`,
            filterParams:{
                debounceMs: 0
            }
        },
```

When the debounce value is set to 0 the filter is applied immediately.

If the debounce was set to `2000` then it would take 2 seconds after the user stops typing for the filter to be applied.

The filter params are often applied to the default column definition:

```
  const defaultColDef = useMemo( ()=> ( {
      flex: 1,
      filterParams:{
          debounceMs: 2000
      }
  }), []);
```

buttons

The buttons filter parameter adds buttons which control the filter e.g. apply, clear.

We can add buttons as follows:

```
  const defaultColDef = useMemo( ()=> ( {
      filterParams:{
          buttons: ['apply','clear']
      }
  }), []);
```

This would cause an `apply` and `clear` button to show under the filter.

There are buttons which can be displayed:

- `apply`: when present the filter is only applied after the user hits the Apply button.
- `clear`: clear the filter form details without removing any active filters on the column.
- `reset`: clear the filter form details and any active filters on that column.
- `cancel`: discard any changes that have been made to the filter, restoring the applied model.

When the `apply` button is used, another filter parameter becomes relevant:

- `closeOnApply` configures the closing of the filter popup when apply button is pressed

Comparators and Date Filters

Date filters are described in the documentation:

https://www.ag-grid.com/react-data-grid/filter-date/

By default the date filter expects a `Date` object in the grid in order to compare the values. We are also able to add our own comparator functions to compare specific `String` formats or other values.

The comparator will work with other filter types, but its most common use case is for dates.

```
          filterParams: {
                comparator: (dateFromFilter, cellValue) => {
                    if (cellValue == null) { return 0; }

                    const dateParts = cellValue.split('/');
                    const day = Number(dateParts[0]);
                    const month = Number(dateParts[1]) - 1;
                    const year = Number(dateParts[2]);
                    const cellDate = new Date(year, month, day);

                    if (cellDate < dateFromFilter) {
                        return -1;
                    } else if (cellDate > dateFromFilter) {
                        return 1;
                    }
                    return 0;
                }
            }
```

The comparator receives the value in the filter and the value in the cell.

Because the cell value is not a date, we have code to split the string into parts and convert it into a date for comparison.

Finally the comparator returns an integer to determine if the filter value is greater than, less than or equal to the cell value:

- -1 cell value less than filter value
- 1 cell value greater than filter value
- 0 for same value

Filter State Models

Filter state models are used to set the state of a filter.

We will add two buttons into the returned JSX which will save and apply the state.

```
      <div>
        <button onClick={onBtSave}>Save</button>
        <button onClick={onBtApply}>Apply</button>
      </div>
```

Then the implementation for the `onClick` methods will use the grid API to get and set the state.

```
  const savedFilterState = useRef();

  const onBtSave = useCallback( ()=> {
    const filterModel = gridRef.current.api.getFilterModel();
    console.log('Saving Filter Model', filterModel);
    savedFilterState.current = filterModel;
  }, []);

  const onBtApply = useCallback( ()=> {
    const filterModel = savedFilterState.current;
    console.log('Applying Filter Model', filterModel);
    gridRef.current.api.setFilterModel(filterModel);
  }, []);
```

The api calls:

- `getFilterModel` - returns an object representing the current state of the filter
- `setFilterModel` - sets the current filter using the filter state object

Filter models are described in the documentation:

https://www.ag-grid.com/react-data-grid/filter-api/

The Filter Model could be used to persist the grid state for later application by the user.

The filters can be cleared by setting an empty model e.g.

```
gridRef.current.api.setFilterModel({})
```

Floating Filters

Floating filters are shown in a tool bar under the heading so that the user doesn't have to access the filters from the popup menu.

Floating filters can be configured using column definitions.

Either on a specific set of columns.

```
    { field: 'athlete',
        floatingFilter: true,
        filter: 'agTextColumnFilter'
```

```
    },
```

Or on the default column definition:

```
  const defaultColDef = useMemo( ()=> ( {
    floatingFilter: true,
    flex: 1,
  }), []);
```

The filter from the popup menu is kept in sync with the floating filter.

Note that setting `floatingFilter: true` on a column would not be enough to create a filter, the actual `filter` property also needs to be set either with default **`true`** or with a custom or built-in filter e.g. 'agTextColumnFilter'.

Enterprise Filters

The enterprise edition of AG Grid comes with two additional, very power filters.

- Set Filter
- Multi Filter

To enable these, the enterprise edition of AG Grid has to be installed, as covered earlier, and we need to import the enterprise modules:

```
import 'ag-grid-enterprise';
```

Set Filter

The Set filter allows the user to filter values using checkboxes to select items from a set of values.

```
    { field: 'year', filter: 'agSetColumnFilter' },
```

The set of values used in the drop down are taken from the values in the column.

In addition to selecting items using the checkboxes it is also possible to use the text box to search for items in the set.

These are documented here:

https://www.ag-grid.com/react-data-grid/filter-set/ ### Multi Filter

The Multi Filter allows multiple filters to be used, by default the Text Filter and Set Filter are shown. It is possible to configure the filters shown using a `filter` array in the `filterParams`.

A default multi filter can be added as a `filter` property on a column or default column definition.

```
    { field: 'country', filter: 'agMultiColumnFilter' },
```

https://www.ag-grid.com/react-data-grid/filter-multi/

Filter Menu Container

By default the filter menus use the grid as the parent container. This means that if the grid is smaller than the filter menu, the filter menu would be cropped to fit the size of the grid.

When working with a smaller grid, it is possible to configure the parent container for the menu by using the `popupParent` property on the grid.

In the example below, the `document.body` is used as the container so no matter what size the grid is, the menu will always be visible.

```
    <AgGridReact ref={gridRef}
        popupParent={document.body}
        rowData={rowData} animateRows={true}
        columnDefs={columnDefs} defaultColDef={defaultColDef}
        />
```

Any element in the DOM could be used, so it could be shown anywhere on the web page to fit the needs of your application design.

React Custom Filters

The default filters provided by AG Grid can be customised both in terms of the GUI and the underlying logic to filter the items.

https://www.ag-grid.com/react-data-grid/filter-custom/

Video Tutorial

https://youtu.be/yO3_nTyDv6o

00:00 creating custom filters 01:24 API Interface 02:24 Creating the GUI 02:50 Implementing Filtering 04:06 Filtering with Options 05:58 Using filterParams and props 07:38 Make Filter GUI Configurable 09:05 Filter Models 11:14 getModelAsString 13:08 onNewRowsLoaded 13:32 onAnyFilterChanged 14:05 destroy 15:06 afterGuiAttached 15:31 App calling filter 17:10 Summary

Source Code For This Section

The source code for this section is available in Github in the 'react-data-grid' repo and the subfolder `getting-started-video-tutorial/src/006-react-custom-filters`:

- https://github.com/ag-grid/react-data-grid/tree/main/getting-started-video-tutorial/src/006-react-custom-filters

Starting Point

The starting code is a simple grid which renders data from a server, as shown by the `App.js` below:

```
import './App.css';

import {AgGridReact} from 'ag-grid-react';
import 'ag-grid-community/dist/styles/ag-grid.css';
import 'ag-grid-community/dist/styles/ag-theme-alpine.css';

import 'ag-grid-enterprise';
import {useState, useRef, useEffect, useMemo, useCallback} from 'react';

function App() {

  const gridRef = useRef();
  const [rowData, setRowData] = useState();
  const [columnDefs, setColumnDefs] = useState([
        { field: 'athlete' },
```

```
        { field: 'year' },
        { field: 'age' },
        { field: 'country' },
        { field: 'date' },
        { field: 'sport' },
        { field: 'gold' },
        { field: 'silver' },
        { field: 'bronze' },
        { field: 'total' }
  ]);

  useEffect(() => {
    fetch('https://www.ag-grid.com/example-assets/olympic-winners.json')
    .then(result => result.json())
    .then(rowData => setRowData(rowData))
  }, []);

  return (
      <div className="ag-theme-alpine" style={{height: '100%'}}>
        <AgGridReact ref={gridRef}
            rowData={rowData} animateRows={true}
            columnDefs={columnDefs} defaultColDef={defaultColDef}
            />
      </div>
  );
}

export default App;
```

Default Filter

As a reminder, a default text filter can be applied to a column definition by configuring the `filter` property:

```
        { field: 'year', filter: true },
```

We can replace both the GUI and the filtering logic by creating a custom filter.

Custom Filter

We can create a custom component in a separate file then using it in `App.js`

A simple functional filter component would be the following code in a file called `YearFilter.js`:

```
import React from 'react';

export default p =>{
```

```
    return(<>Hello World!!!</>);
}
```

We can then import that into our `App.js`

```
import YearFilter from './YearFilter';
```

Then configure the `year` column to use this filter.

```
        { field: 'year', filter: YearFilter },
```

The filter on the `year` column would now display the text “Hwllo World!!!”, this is not very usable, but it is a good start to see how easy it is to configure the AG Grid GUI, the next action is to then make it do something.

Functional Custom Filter

The component needs to be able to answer questions that the grid asks like:

- Is the filter active?
- Does this row pass the active filter?

API Interface

To do this the filter implements an interface that the grid can use.

In react functional components we do this by creating forward refs and using the imperative handle.

```
import React, {forwardRef, useImperativeHandle} from 'react';

export default forwardRef( (props, ref) =>{

    useImperativeHandle(ref, ()=> {
            isFilterActive() {
                return false;
            },
            doesFilterPass(params) {
                return false;
            },
            getModel() {
                return undefined;
            },
            setModel(model) {
            }
    });
```

```
    return(<>Hello World!!!</>);
});
```

The four mandatory methods for a filter are:

- `isFilterActive`
  - if this returns true then the filter is active
- `doesFilterPass`
  - if the filter is active then `doesFilterPass` is called for each row in the grid and it returns false if the row should be filtered and not displayed, true means the row will be displayed as it passes the filter.
- `getModel`
  - used when the Api get model is used.
- `setModel`
  - used when the Api set model is used.

The above code creates the internal API interface, now we need to create the GUI for the user.

The full documentation for the custom filter interface is online:

- https://ag-grid.com/react-data-grid/component-filter/#custom-filter-interface-2

Creating the GUI

```
    return(
        <>
            <div>Year filter</div>
            <label>
                Filter On
                <input type='radio' name='rbYearFilter'
                       onChange={offListener}
                       checked={filterState=='off'}></input>
            </label>
            <label>
                Filter Off
                <input type='radio' name='rbYearFilter'
                       onChange={onListener}
                       checked={filterState=='on'}></input>
            </label>

        </>
    );
```

To make this work we have to use an internal state for the `filterState`:

```
    const [filterState, setFilterState] = useState('off');
```

And add the listener callbacks to set the state when the buttons are clicked.

```
    const onListener = useCallback|(
                ()=> setFilterState('on'), []);
    const offListener = useCallback(
                ()=> setFilterState('off'), []);
```

This will create a filter drop down that has a set of radio buttons to switch the filter on and off. Currently the filter does not apply because it is never active, the `isFilterActive` function always returns **`false`**.

Implementing Filtering

The GUI changes the state in the component. We can now implement the API methods to actually filter the grid.

The filter is active when the 'Filter On' radio button has been selected and the `filterState` is `on`:

```
            isFilterActive() {
                return filterState=='on';
            },
```

If we leave the `doesFilterPass` returning **`false`** then we will see the filter works by filtering everything out.

```
            doesFilterPass(params) {
                return false;
            },
```

We also have to inform the grid that the filter has changed, which we do when the `filterState` changes, and we call the `filterChangedCallback` function on the `props` passed through from the grid.

```
    useEffect( ()=> props.filterChangedCallback(),
                [filterState]);
```

In this current form, the filter can be switched on and off and rows will be displayed or not.

```
import React, {forwardRef, useCallback, useEffect, useImperativeHandle}
   from 'react';

export default forwardRef( (props, ref) =>{
```

```
    const [filterState, setFilterState] = useState('off');

    useImperativeHandle(ref, ()=> {
            isFilterActive() {
                return filterState=='on';
            },
            doesFilterPass(params) {
                return false;
            },
            getModel() {
                return undefined;
            },
            setModel(model) {
            }
    });

    const onListener = useCallback|(
                ()=> setFilterState('on'), []);
    const offListener = useCallback(
                ()=> setFilterState('off'), []);

    useEffect( ()=> props.filterChangedCallback(),
                [filterState]);

    return(
        <>
            <div>Year filter</div>
            <label>
                Filter On
                <input type='radio' name='rbYearFilter'
                       onChange={offListener}
                       checked={filterState=='off'}></input>
            </label>
            <label>
                Filter Off
                <input type='radio' name='rbYearFilter'
                       onChange={onListener}
                       checked={filterState=='on'}></input>
            </label>

        </>
    );
});
```

Filtering with Options

Currently the filter is just on, or off, so we will amend the GUI to have options and then filter when those particular options have been chosen.

This time we will make the `filterState` match the chosen options, and simple set the filter state when the option is chosen, rather than wrap it in a callback.

```
    return(
        <>
            <div>Year filter</div>
            <label>
                Filter Off
                <input type='radio' name='rbYearFilter'
                       onChange={()=>setFilterState('off')}
                       checked={filterState=='off'}></input>
            </label>
            <label>
                2004
                <input type='radio' name='rbYearFilter'
                       onChange={()=>setFilterState(2004)}
                       checked={filterState=='2004'}></input>
            </label>
            <label>
                2008
                <input type='radio' name='rbYearFilter'
                       onChange={()=>setFilterState(2008)}
                       checked={filterState=='2008'}></input>
            </label>

        </>
    );
```

We cam now delete the callback code:

```
    const onListener = useCallback|(
                ()=> setFilterState('on'), []);
    const offListener = useCallback(
                ()=> setFilterState('off'), []);
```

And make the filter condition to be ‘not on’.

```
            isFilterActive() {
                return filterState!='off';
            },
```

It is finally time to implement some filtering condition, because we store the actual filter value in the filterState we can compare it with the value of the field.

```
            doesFilterPass(params) {
                return params.data.year == filterState;
            },
```

This will create a working filter with three choices: off, 2004, 2008.

The `params` passed into `doesFilterPass` makes it possible to compare the values in the grid with the filter, and params gives us access to the full data in the row.

The `props` passed into the component gives us access to the api, column definitions and the callbacks required to implement a complete custom filter.

The documentation for `props` is online:

- https://ag-grid.com/react-data-grid/component-filter/#custom-filter-parameters-2

Using `filterParams` as Props to Configure Filter

Currently the filter is hard coded to work with the `year` column, we can use the `props` to configure the filter to make it more flexible and re-usable.

To start with we can configure the title displayed for the filter, first pass in the title we want to render.

In `App.js` when we configure the filter we can add some `filterParams` which become the `props` for the filter component.

```
        { field: 'year',
            filter: YearFilter,
            filterParams: {
                title: 'My Custom Filter'
            }
        },
```

Then we can use the `filterParams` as `props`.

```
    return(
        <>
            <div>{props.title}}</div>
```

`doesFilterPass` params

The function `params` passed through to `doesFilterPass` contains the data and the RowNode, we can use this to implement any complex filtering operation.

Rather than hard code the field name:

```
            doesFilterPass(params) {
                return params.data.year == filterState;
            },
```

We can pull the field information from the props and compare to the params.

```
            doesFilterPass(params) {
                const field = props.colDef.field;
                return params.data[field] == filterState;
            },
```

This now makes the filter work on any field, with only the filter values in the GUI being hard coded.

Make Filter GUI Configurable

We can add class names to make the styling more configurable later.

One entry, to switch the filter off has been hard coded, but the remainder of the options will be passed in as an array named 'values' in the `props` (`filterParams` in `App.js`)

```
    return(
        <>
            <div className='filter-title'>{props.title}</div>
            <div classNAme='filter-state'>
                State = {filterState}
            </div>
            <div className='filter-entry'>
                <button
                    onClick={()=>setFilterState('off')}>
                    Off
                </button>
            </div>
            { props.values.map( value =>(
                <div className='filter-entry'>
                    <button key={value}
                        onClick={()=>setFilterState(value)}>
                        {value}
                    </button>
                </div>
            ))}
        </>
    );
```

The filter can now be used on multiple columns so we should probably import it with a more generic name:

```
import MyFilter from './YearFilter';
```

Now we can configure the filter from the `App.js` column definitions:

```
        { field: 'year',
            filter: MyFilter,
            filterParams: {
                title: 'My Custom Filter',
```

```
                values: [2000,2004,2006]
            }
        },
        { field: 'age',
            filter: MyFilter,
            filterParams: {
                title: 'Age Filter',
                values: [18,19,20,21]
            },
        },
```

This creates a filter on the `year` column with the values: 2000, 2004, and 2006.

Also a filter on the `age` column with the values: 18,19,20 and 21.

Filter Models

Filter models allow us to save and restore filter state.

```
        getModel() {
            if (filterState=='off') {
                return undefined;
            }
            return {
                state: filterState
            }
        },
        setModel(model) {
            if (model==null) {
                setFilterState('off');
            } else {
                setFilterState(model.state);
            }
        },
```

The grid calls `getModel` when it wants to have a saveable representation of the filter.

The grid calls the `setModel` function when it wants the filter to restore state, the `model` parameter is the data that we returned in the `getModel` function.

The 'model' is completely under our control so we can use whatever representation required for the filter.

In the example above it is a simple object with a `state` property e.g. `{state: 'off'}`.

These functions are called when the grid api `getFilterModel` and `setFilterModel` functions are called.

To demonstrate this, add some buttons to the grid in `App.js` to save and restore the filters.

```
  const filterState = useRef();

  const onBtSave = useCallback( ()=> {
    filterState.current = gridRef.current.api.getFilterModel();
    console.log('saving', filterState.current);
  });
  const onBtRestore = useCallback( ()=> {
    console.log('restoring', filterState.current);
    gridRef.current.api.setFilterModel(filterState.current);
  });

  return (
    <div style={{height: '100%'}}>
      <div>
        <button onClick={onBtSave}>Save</button>
        <button onClick={onBtRestore}>Restore</button>
      </div>
```

When the `save` button is clicked the `getFilterModel` is called. This will call the `getModel` on any active filters.

Optional API Methods

There are many methods available for implementation on the Custom Filter API Interface.

- https://ag-grid.com/react-data-grid/component-filter/#custom-filter-interface-2

We have already covered the mandatory methods:

- `isFilterActive`
  - true when the filter is active
- `doesFilterPass`
  - return false if the row should be filtered and not displayed
- `getModel`
  - get representation of the filter
- `setModel`
  - restore representation of the filter

Optional methods:

- `getModelAsString`

    - text shown in a floating filter

- `onNewRowsLoaded`
    - called when new rows are loaded to allow filter to adjust if necessary
- `onAnyFilterChanged`
    - called when any filter is changed
- `destroy`
    - when column is destroyed to allow filter to do any necessary cleanup
- `afterGuiAttached`
    - called when the popup is shown

getModelAsString

The `getModelAsString` function is used when the column shows a floating filter:

```
        { field: 'year',
              filter: MyFilter,
              filterParams: {
                title: 'Year Filter',
                values: [2000,2004,2006]
              },
              floatingFilter: true
          },
```

The addition of the `floatingFilter` property adds a text field filter in addition to the popup menu, by default this is read only and shows the text returned from the `getModelAsString` function.

```
            getModelAsString() {
                return filterState=='off' ?
                        '' : filterState;
            },
```

The implementation above returns an empty string when there is no filter and the `filterState` value when set. This will show the `filterState` in the read only text field for the floating filter which is a useful way of showing the filter to the user.

The floating filter is also customisable, but the default is the read only text field.

onNewRowsLoaded

`onNewRowsLoaded` is called whenever new rows are added to the grid, this is useful if the filter has a dependency on the date, e.g. if the values in the filter are pulled from the rows like a set filter.

onAnyFilterChanged

`onAnyFilterChanged` when any column filter is changed. This can be useful if you want to adapt to the settings in any additional column filter.

destroy

This is used for other frameworks, the more 'react' way of implementing this is to have a `useEffect` with no dependencies.

```
    useEffect( ()=> {
        console.log(props.title + ' filter created');
        return ()=> console.log(
                    props.title + ' filter destroyed');
    }, []);
```

This is only called when the column is destroyed, not when the column is 'hidden' or the filter menu is closed.

afterGuiAttached

`afterGuiAttached` is called when the GUI for the filter is rendered on the screen.

This can be useful to add focus to a specific element in the custom GUI created for the filter.

App Specific API calls

Custom functions can be added to the filter to allow the application to call the filter specifically.

```
    useImperativeHandle(ref, ()=> {
        return {
            somethingToDoWithMyApp(){
                console.log('somethingToDoWithMyApp called');
            }
```

The above API function can be called from the main `App.js` by using the grid api to get an instance of the filter with the `getFilterInstance` function.

```
  const onBtCustomApi = useCallback( ()=> {
    gridRef.current.api.getFilterInstance('year', instance=>{
        instance.somethingToDoWithMyApp();
    });
  });
```

And if this was the handler for an onClick event:

```
<button onClick={onBtCustomApi}>CustomA API</button>
```

This creates a button in the main app, which when clicked will get the current filter instance on the 'year' column and call the `somethingToDoWithMyApp` function on that filter.

We can also call any function on the instance so could call `isFilterActive` or any of the implemented API functions.

Summary

Custom Filters are one of the more advanced features of AG Grid and open a lot of flexibility to your interfaces so are worth mastering.

The documentation for custom filters can be found online:

- https://ag-grid.com/react-data-grid/filter-custom/

Custom Floating Filters

The Floating filters can be customised. The floating filters use the underlying filter for the column and add a layer of rendering and editing to make filtering easier for the user.

https://www.ag-grid.com/react-data-grid/component-floating-filter/

Video Tutorial

https://youtu.be/CxwfX4KodaM

00:00 starting from custom filters 01:00 floating filter basics 01:40 Floating Filter Component 02:25 Accessing Filter Models 03:50 Changing Filter Models 05:15 Floating Filter Responsibilities 05:58 props for Floating Filter 06:45 using filterParams 08:45 Life-cycle of a Floating Filter 10:06 Re-using Floating Filters 10:30 Summary

Source Code For This Section

The source code for this section is available in Github in the 'react-data-grid' repo and the subfolder `getting-started-video-tutorial/src/007-react-custom-floating-filters`:

- https://github.com/ag-grid/react-data-grid/tree/main/getting-started-video-tutorial/src/007-react-custom-floating-filters

Starting Code

The code we are starting from has a basic data grid that reads data from a server, as shown in `App.js` below:

```
import './App.css';

import {AgGridReact} from 'ag-grid-react';
import 'ag-grid-community/dist/styles/ag-grid.css';
import 'ag-grid-community/dist/styles/ag-theme-alpine.css';

import {useState, useRef, useEffect, useMemo, useCallback} from 'react';

import ValuesFilter from './valuesFilter';
import ValuesFloatingFilter from './valuesFloatingFilter';

function App() {

  const [rowData, setRowData] = useState();
```

```
  const [columnDefs, setColumnDefs] = useState([
        { field: 'athlete' },
        { field: 'year',
              filter: ValuesFilter,
              filterParams: {
                values: [2000,2004]
              }
          },
        { field: 'age'},
        { field: 'country' },
        { field: 'date' },
        { field: 'sport' },
        { field: 'gold' },
        { field: 'silver' },
        { field: 'bronze' },
        { field: 'total' }
  ]);

  useEffect(() => {
    fetch('https://www.ag-grid.com/example-assets/olympic-winners.json')
    .then(result => result.json())
    .then(rowData => setRowData(rowData))
  }, []);

  return (
    <div className="ag-theme-alpine" style={{height: '100%'}}>
      <AgGridReact rowData={rowData} columnDefs={columnDefs}
          animateRows={true}/>
    </div>
  );
}

export default App;
```

This application has a custom filter which is being used on the `year` field:

```
        { field: 'year',
              filter: ValuesFilter,
              filterParams: {
                values: [2000,2004]
              }
        },
```

A floating filter builds on the custom filter to provide an easy way to control or display the filter.

The custom filter itself is contained in `valuesFilter.js`. This was explained in the React Custom Filters section of this tutorial.

```
import React, {forwardRef, useCallback, useEffect, useImperativeHandle,
   useState} from 'react';
```

```
export default forwardRef( (props, ref) => {

    const [filterState, setFilterState] = useState('off');

    useImperativeHandle(ref, ()=> {
        return {
            isFilterActive() {
                return filterState!='off';
            },
            doesFilterPass(params) {
                const field = props.colDef.field;
                return params.data[field] == filterState;
            },
            getModel() {
                if (filterState=='off') {
                    return undefined;
                }
                return {
                    state: filterState
                }
            },
            setModel(model) {
                if (model==null) {
                    setFilterState('off');
                } else {
                    setFilterState(model.state);
                }
            }
        };
    });

    useEffect( ()=> props.filterChangedCallback(),
                [filterState]);

    return (
        <>
            <div className='filter-state'>
                State = {filterState}
            </div>
            <div className='filter-entry'>
                <button
                    onClick={()=>setFilterState('off')}>
                        Off
                </button>
            </div>
            { props.values.map( value => (
                <div key={value} className='filter-entry'>
                    <button
                        onClick={()=>setFilterState(value)}>
                            {value}
                    </button>
```

```
                </div>
            )) }
        </>
    );
});
```

Floating Filter Basics

Floating filters build on custom filters.

To enable Floating filters we add the `floatingFilter` property to the appropriate column definitions:

```
        { field: 'year',
              filter: ValuesFilter,
              filterParams: {
                values: [2000,2004]
              },
              floatingFilter: true
          },
```

Setting the `floatingFilter` to true uses the default read only floating filter, where the value shows is the string representation of the filter model returned by the custom filter.

In the starting point code, no function has been created to render the model as a String so initially it will be blank, no matter which filter is set on the column.

To rectify this we add a `getModelAsString` function to the custom filter which returns the filter model as a String.

```
            getModelAsString() {
                return filterState=='off' ?
                        '' : filterState;
            },
```

This would be added to the `valuesFilter.js` component e.g.

```
    useImperativeHandle(ref, ()=> {
        return {
            isFilterActive() {
                return filterState!='off';
            },
            getModelAsString() {
                return filterState=='off' ?
                        '' : filterState;
            },
```

With this code in place, the default read only floating filter would show the value of the `filterState` variable.

Rather than use `getModelAsString` we can create our own React components to render as the floating filter.

Floating Filter Component

Using a component as a custom floating filter means that we have full control over the data rendered and can offer additional functionality to help the user filter and work with the data in the grid.

We'll start by creating a simple custom filter in a new file called `valuesFloatingFilter.js` and it will render some static text to start with:

```
import React from `react`;

export default props =>{
    return (<>Hello World!!!</>);
}
```

Then we configure the column definition to use this component instead of the default read only renderer.

```
        { field: 'year',
              filter: ValuesFilter,
              filterParams: {
                values: [2000,2004]
              },
              floatingFilter: true,
              floatingFilterComponent: ValuesFloatingFilter
          },
```

The `floatingFilterComponent` property specifies the component to use as the `floatingFilter`. Note that `floatingFilter: true` still needs to be set, otherwise the `floatingFilter` will not be displayed.

We need to remember to import the filter component into `App.js`.

```
import ValuesFloatingFilter from './valuesFloatingFilter';
```

Accessing Filter Models

The underlying custom filter has a filter model with custom data and functions. We can access this in the custom floating filter when we amend the floating filter component as follows:

```
import React, {forwardRef, useImperativeHandle} from 'react';

export default forwardRef( (props, ref) => {

    useImperativeHandle(ref, ()=> {
        return {
            onParentModelChanged(parentModel) {
                console.log('onParentModelChanged',parentModel);
            }
        };
    });

    return(<>Hello World!!!</>);
```

The above code does not change the external appearance of the floating filter, but we have now implemented the interface that allows the grid to communicate with the floating filter.

`forwardRef` and `useImperativeHandle` alow the grid to communicate to the component.

We implement the `onParentModelChanged` interface as the grid will call this method when the underlying custom filter model changes i.e. when someone changes the value of the filter.

The `parentModel` object is the same object that is returned by the `getModel` function in the custom filter.

Rather than writing the model out to the console, which is useful for debugging and learning the various properties of the AG Grid API, we can render the filter state as the custom filter.

```
import React, {forwardRef, useImperativeHandle, useState} from 'react';

export default forwardRef( (props, ref) => {

    const [value, setValue] = useState();

    useImperativeHandle(ref, ()=> {
        return {
            onParentModelChanged(parentModel) {
                if(parentModel){
                    setValue(parentModel.state);
                }else{
                    setValue();
                }
            }
        };
    });

    return(<>{value}</>);
```

The Custom Floating Filter component now maintains a state, which is the value of the underlying filter

component, and renders this as the floating filter.

Learn more in the documentation:

https://www.ag-grid.com/react-data-grid/component-floating-filter/#custom-floating-filter-interface-2

Changing Filter Models

The floating filter is not limited to rendering the underlying custom filter values, it can also amend the underlying filter. This allows the Custom Floating Filter component to offer an alternative GUI to set the filter for a column.

To demonstrate this we will create a single button as the custom filter, and when pushed it will set the filter to a hard coded default value:

```
    const myListener = useCallback(()=>{
        props.parentFilterInstance( instance=>{
            instance.setValue(2000);
        }, []);
    }, []);

    return(
        <>
            <button onClick={myListener}>Push</button>
            {value}
        </>
        );
```

`parentFilterInstance` is a function exposed on the `prop` which we can use in a callback to access the `instance` of the parent custom filter.

We have full access to the `instance` which is the object of the parent custom filter API returned in the `useImperativeHandle` code and can access any of the API functions e.g. `isFilterActive`, `getModel`, `setModel`.

In this case we are using a `setValue` method, so we would need to write the `setValue` method in the `valuesFilter.js`

```
    useImperativeHandle(ref, ()=> {
        return {
            setValue(value) {
                setFilterState(value);
            },
```

This `setValue` method allows us to change the state of the filter and set a new filter on the column.

Floating Filter Responsibilities

The main filter `ValuesFilter` represented in `valuesFilter.js` and configured on the column as the `filter`, is the object that holds the filter state and implements the custom filtering logic. The grid uses this to know if the filter has changed. It also controls the internal representation model for the filter state.

```
        { field: 'year',
              filter: ValuesFilter,
```

The floating filter has fewer responsibilities. It is another way of representing the filter state to the user on the GUI and can present an alternative GUI to the user for changing the filter. All changes to the filter are carried out by the parent filter through the exposed API returned by the parent filter.

Props for Floating Filter

The props passed into the floating filter offer a lot of flexibility.

We have access to the:

- `api` which is the full Grid API if we want to create a very flexible filter.
- `column` which is an object representing the full details of the column the filter is attached to.
- `currentParentModel` is a function to access the parent filter model.
- `parentFilterInstance` is a function to access the parent filter API.
- `filterParams` are the props passed into the parent filter.
- `showParentFilter` can be used to display the pop up for the parent filter.
- `suppressFilterButton` tells us if we have configured the parent filter button to display or not.

The props for the Floating Filter are fully documented in the online documentation:

https://www.ag-grid.com/react-data-grid/component-floating-filter/#custom-filter-parameters-2

Using filterParams

`filterParams` are the props passed into the parent filter, this gives us access to all the information that was used to configure the parent filter, allowing us to fully create an alternative filter GUI with no limitations.

In our example code, the parent filter was configured with `filterParams` to configure the filter dynamically.

```
        { field: 'year',
              filter: ValuesFilter,
              filterParams: {
                values: [2000,2004]
              },
```

Using the `filterParams` on our floating filter `props` gives us access to those values in our custom floating filter.

```
const allValues = props.filterParams.values;
```

We can then use those values in our custom filter GUI to render the values as a series of buttons to control the filter, and we can see the value of the filter which as been set.

```
    return(
        <div>
            <div>
                <button onClick={()=>clickListener('off')}>
                    Off
                </button>
                </div>
                { allValues.map(
                    value =>
                        <button key={value} onClick={()=>clickListener(
                           value)}>
                            {value}
                        </button>
                ) }
            </div>
            <div>{value}</div>
        </div>
    );
```

The earlier `myListener` is renamed to `clickListener` because it now covers multiple buttons and the `value` is passed in and used to set the value of the parent filter.

```
    const myListener = useCallback( value =>{
        props.parentFilterInstance( instance=>{
            instance.setValue(value);
        }, []);
    }, []);
```

Life-cycle of a Floating Filter

We can visually see the life-cyle of a floating filter by writing to the console when a floating filter is created and when it is destroyed.

This can be done by using a `useEffect` for mounting the component with a returned callback for when the filter is unmounted.

```
    useEffect( ()=> {
        console.log('Floating Filter Created');
        return ()=> console.log('Floating Filter Destroyed');
    }, []);
```

With this code added to the component we would see that:

- the floating filter is created as soon as the grid is drawn
- if the column is not visible, then the floating filter is destroyed, this is due to the DOM virtualisation to only render what is necessary
- as soon as the column is brought back into view the floating filter is re-created

Custom Floating filters are created when the column is visible and destroyed when the column is not visible. This is a marked difference to the Custom Filters, these are created when displayed for the first time, but remain even when not visible because the state from that component is used to control the filtering on the data grid and the filter can still be active even if you can't see the filter GUI.

Re-using Floating Filters

In the same way that Custom Filter and other components can be re-used, the Custom Floating Filters can be re-used and applied to other columns.

For example, we could additionally configure the `age` column to use the `ValuesFilter` and `ValuesFloatingFilter`.

```
        { field: 'age',
              filter: ValuesFilter,
              filterParams: {
                values: [18, 19]
              },
              floatingFilter: true,
              floatingFilterComponent: ValuesFloatingFilter
        },
```

Summary

Floating Filters can be customised to provide an alternative way of visualising the column filter state and an alternative GUI for the filter.

The don't maintain the filter state, they are simply a rendering mechanism so the component is created when visible and destroyed when not visible.

Full documentation is available online:

https://www.ag-grid.com/react-data-grid/component-floating-filter/

React Components Overview

Previous sections of this tutorial have shown some of the Custom AG Grid components in detail. To help solidify your understanding of components we will look at them from a general level, this will really help you understand how AG Grid works and make it easy to create any of the other components available.

https://www.ag-grid.com/react-data-grid/component-types/

Video Tutorial

https://youtu.be/eglfpHRpcu0

00:00 Component Types Overview 00:27 Starting Code 00:38 Generic Component 01:52 Component Props 02:54 Custom Properties 03:30 Naming Pattern 04:13 Component Selectors 05:40 Class and JavaScript Components 08:05 Registering Components 09:28 Grid Provided Components 11:16 Grid Components 12:24 Enterprise components 14:25 Summary

Source Code For This Section

The source code for this section is available in Github in the 'react-data-grid' repo and the subfolder `getting-started-video-tutorial/src/008-react-components`:

- https://github.com/ag-grid/react-data-grid/tree/main/getting-started-video-tutorial/src/008-react-components

Starting Code

The basic code for `App.js` to start with, renders a grid and pulls information from the server.

```
import './App.css';

import {AgGridReact} from 'ag-grid-react';
import 'ag-grid-community/dist/styles/ag-grid.css';
import 'ag-grid-community/dist/styles/ag-theme-alpine.css';

import {useState, useRef, useEffect, useMemo, useCallback} from 'react';

function App() {

  const [rowData, setRowData] = useState();
  const [columnDefs, setColumnDefs] = useState([
```

```
        { field: 'athlete' },
        { field: 'year'},
        { field: 'age'},
        { field: 'country' },
        { field: 'date' },
        { field: 'sport' },
        { field: 'gold' },
        { field: 'silver' },
        { field: 'bronze' },
        { field: 'total' }
  ]);

  useEffect(() => {
    fetch('https://www.ag-grid.com/example-assets/olympic-winners.json')
    .then(result => result.json())
    .then(rowData => setRowData(rowData))
  }, []);

  return (
    <div className="ag-theme-alpine" style={{height: '100%'}}>
      <AgGridReact rowData={rowData} columnDefs={columnDefs}
          animateRows={true}/>
    </div>
  );
}

export default App;
```

This really has no additional customisation.

Generic Component

To demonstrate the similarity between components we will start by creating a very basic component and then we will use that in various places across the grid.

We'll create the component in `comps.js`, and we'll use this to store all the components we create in this section:

```
export const HelloWorldComp = p => {
    return <>Hello World!!</>;
};
```

We can then import this component into `App.js` for use by the grid:

```
import {HelloWorldComp} from './comps';
```

This component can be used in various ways, because the component isn't customisable it won't be a good example of each use case, but it will demonstrate the generic nature of components and how AG

Grid handles them.

First we can add the component as a Cell Renderer, a Custom Filter and a Custom Header:

```
        { field: 'athlete', cellRenderer: HelloWorldComp},
        { field: 'age', filter: HelloWorldComp},
        { field: 'country', headerComp: HelloWorldComp},
```

This will cause:

- every `athlete` cell to display “Hello World!!”
- the header of `country` column to display “Hello World!!”
- when the user clicks the filter of `age` column they will see “Hello World!!”

Component Props

Each component will be treated as a normal React component and sent `props`. The object representing the `props` will be different depending on the use of the component i.e. a `cellRenderer` receives different `props` from a `filter`.

A simple way to experiment with the `props` available, without reading the documentation, is to print them to the console and see them in action.

```
export const HelloWorldComp = p => {
    console.log(p);
    return <>Hello World!!</>;
};
```

This allows you to explore all the output values in the console when you interact with the grid.

The props for each component type can be found in the documentation for each component type and are documented as the IxParams interface e.g.

- `cellRenderer` uses the `ICellRendererParams`
  - https://www.ag-grid.com/react-data-grid/component-cell-renderer/
- `filter` uses the `IFilterReactComp`
  - https://www.ag-grid.com/react-data-grid/component-filter/
- `headerComp` uses the `IHeaderParams`
- ag-grid.com/react-data-grid/component-header/

Custom Properties

In addition to the grid supplied `props` it is also possible to pass in any custom properties we want. This is done using the *componentType*`Params` property in the column definition.

```
export const HelloComp = p => {
    return <>Hello {p.name}!</>;
};
```

The `name` value can be configured in the column definition by using the associated `Params` property for each component type.

```
        {
          field: 'athlete',
          cellRenderer: HelloComp,
          cellRendererParams: {name: 'Tom'}
        },
        {
          field: 'age',
          filter: HelloComp,
          filterParams: {name: 'Dick'},
        },
        {
          field: 'country',
          headerComponent: HelloComp,
          headerComponentParams: {name: 'Harry'}
        },
```

Remember to import the `HelloComp` into `App.js` for this to work.

```
import {HelloComp} from './comps';
```

Naming Pattern

The basic naming pattern for the properties is:

- `[key]` - component
- `[key]Params` - custom parameters passed to the component
- `[key]Selector` - a conditional way of choosing between components

So for a `cellRenderer` we have `cellRendererParams` and `cellRendererSelector`.

Component Selectors

A Component Selector allows us to add some simple functionality to choose between components at run time.

Here is a simple example for a cell renderer. Based on the naming pattern we just explained, this would be a `cellRendererSelector`.

```
        {
          field: 'athlete',
          // cellRenderer: HelloComp,
          cellRendererParams: {name: 'Tom'},
          cellRendererSelector: p => {
              return {
                  component: Math.random() > 0.5 ?
                    HelloComp : GoodbyeComp
              }
          }
        },
```

The `cellRendererSelector` in the above example randomly returns either a `HelloComp` or a `GoodbyeComp`, the code for `GoodbyComp` is shown below.

Remember to remove or comment out the `cellRenderer` when using a `cellRendererSelector`.

The `cellRendererSelector` is passed props, these include the grid api, column definitions so it is possible to use all the information in the row, column, cell or grid to support any decision process required.

```
export const GoodbyeComp = p => {
    return <>Goodbye {p.name}!</>;
};
```

More detailed documentation can be found online:

https://www.ag-grid.com/react-data-grid/components/

`Selector`s are only documented for `cellRenderer` and `cellEditor`.

Class Components

Currently we have seen React functional components:

```
export const HelloComp = p => {
    return <>Hello {p.name}!</>;
};
```

It is also possible to use class components.

To do that we would have to import the `Component` from `React`:

```
import {Component} from 'react';
```

And amend the previous `GoodbyeComp` to be a class component:

```
export class GoodbyeComp extends Component {
    render() {
        return <>Goodbye {this.props.name}!</>;
    }
}
```

JavaScript Components

JavaScript components make it possible for different teams in an organisation to share custom components for AG Grid across the different frameworks that AG Grid supports.

AG Grid has a 100% React Rendering Engine when used with React. AG Grid also has framework specific rendering engines for Vue, Angular and Vanilla JavaScript.

JavaScript components can be shared between teams and use no matter which framework rendering engine is being used.

To create a JavaScript Component we have to implement the methods for an AG Grid component interface:

```
export class GreetJSComp {
    init(p) {
        this.eGui = document.createElement('span');
        this.eGui.innerHTML = 'GreetJS ' + p.name;
    }
    getGui() {
        return this.eGui;
    }
}
```

Different Components

It doesn't matter if the component is function based, class based or JavaScript based, they are all added to the column definitions in the same way:

```
        {
          field: 'athlete',
          cellRenderer: HelloComp,
```

```
          cellRendererParams: {name: 'Tom'},
        },
        {
          field: 'age',
          filter: GoodbyeComp,
          filterParams: {name: 'Dick'},
        },
        {
          field: 'country',
          headerComponent: GreetJSComp,
          headerComponentParams: {name: 'Harry'}
        },
```

Registering Components

Components can be defined in a column definition by a symbolic key if we register the components with the grid first.

In the `App.js`:

```
  const components = useMemo( ()=> ({
    aaa: HelloComp,
    bbb: GoodbyeComp
  }), []);
```

Then register the component mapping with the grid in the grid definition:

```
return (
<div className="ag-theme-alpine" style={{height: '100%'}}>
    <AgGridReact ref={gridRef}
        components={components}
        rowData={rowData} animateRows={true}
        columnDefs={columnDefs}
        />
</div>
```

It is then possible to refer to a component using the mapping:

```
        {
        field: 'age',
        cellRenderer: 'aaa',
        cellRendererParams: {name: 'Mick'},
        filter: HelloComp,
        filterParams: {name: 'Dick'},
    },
```

The `cellRenderer` is provided with the String `'aaa'` which maps on to a `HelloComp` so the `HelloComp` would be used as a `cellRenderer` for the `age` column.

Referencing by name from a set of registered component mappings often makes it easier to configure the grid more easily through JSON or other data driven configuration approaches.

Grid Provided Components

The grid comes with some pre-registered components. These all use the namespace `ag` to reduce conflict with your own components. e.g. `agDateInput`, `agColumnHeader`.

These components are all listed in the documentation:

https://www.ag-grid.com/react-data-grid/components/#grid-provided-components

As an example, we could use the `agNumberColumnFilter` as follows:

```
        {
          field: 'age',
          cellRenderer: 'bbb',
          cellRendererParams: {name: 'Mick'},
          filter: 'agNumberColumnFilter',
          filterParams: {name: 'Dick'},
        },
```

All built in components would be reference by String value.

Grid Components

All the examples above have been column based custom components.

The grid also has customisable components which are configured on the grid definition.

- No Rows Overlay component
- Loading Overlay Component

No Rows Overlay component

When the grid has no rows to show, it uses an overlay to hide the grid rows.

```
  useEffect(() => {
    fetch('https://www.ag-grid.com/example-assets/olympic-winners.json')
    .then(result => result.json())
    .then(rowData => setRowData([]))
  }, []);
```

You would see the default “No Rows To Show” overlay when the row data is set to an empty array.

This can be configured using a `noRowsOverlayComponent` and the configuration is made using grid properties:

```
      <AgGridReact ref={gridRef}

          noRowsOverlayComponent={HelloComp}
          noRowsOverlayComponentParams=
                  {{name: 'Susan No Rows'}}
```

Loading Overlay Component

When the grid row data is `undefined` then the `Loading...` overlay is shown:

```
//   useEffect(() => {
//     fetch('https://www.ag-grid.com/example-assets/olympic-winners.json
   ')
//     .then(result => result.json())
//     .then(rowData => setRowData([]))
//   }, []);
```

This can be customised with a `loadingOverlayComponent`.

```
          loadingOverlayComponent={GoodbyeComp}
          loadingOverlayComponentParams=
                  {{name: 'Rachel Loading'}}
```

Enterprise Grid Components

The Enterprise version of AG Grid has two additional components that can be customized.

You can see how to configure and work with the Enterprise version in the earlier “Enterprise Overview” section.

Summary:

- `npm install --save ag-grid-enterprise`
- in `App.js` add `import 'ag-grid-enterprise';`

The two components are:

- Status Bar Panels
- Sidebar Tool Panels

Status Bar Panels

The status bar is shown below the grid.

```
            statusBar={{
              statusPanels: [
                {
                  statusPanel: HelloComp,
                  statusPanelParams: {name: 'Peter'},
                },
                {
                  statusPanel: GoodbyeComp,
                  statusPanelParams: {name: 'Paul'},
                }
              ]
            }}
```

Sidebar Tool Panels

The sidebar allows you to have custom tabbed pop out panels on the right hand side of the screen.

```
            sideBar={{
              toolPanels: [
                {
                  id: '3',
                  labelDefault: 'Columns',
                  toolPanel: 'agColumnsToolPanel',
                },
                {
                  id: '1',
                  labelDefault: 'Custom 1',
                  toolPanel: HelloComp,
                  toolPanelParams: {name: 'Summer'}
                },
                {
                  id: '2',
                  labelDefault: 'Custom 2',
                  toolPanel: GoodbyeComp,
                  toolPanelParams: {name: 'Winter'}
                }
              ]
            }}
```

Summary

You should now have a good generic overview of how components work. The nuances of each can be found in the documentation.

https://www.ag-grid.com/react-data-grid/components/

Updating Row Data

This section covers the functionality of the grid API for programmatically updating data in the the grid: updating, inserting and deleting rows. We will also cover techniques for making updates efficient and handling high volume performant transactions and asynchronous transactions for high frequency transactions.

Video Tutorial

https://youtu.be/_V5qFr62uhY

00:00 Updating Row Data 00:22 Starting Code 01:15 Inserting Data 03:43 Row Ordering 04:38 Deleting Rows 05:24 Updating Rows 07:27 Adding Large Amounts of Data 10:43 Update Transactions 12:21 Remove Transactions 13:12 High Frequency Updates 18:16 Summary

Source Code For This Section

The source code for this section is available in Github in the 'react-data-grid' repo and the subfolder `getting-started-video-tutorial/src/009-updating-row-data`:

- https://github.com/ag-grid/react-data-grid/tree/main/getting-started-video-tutorial/src/009-updating-row-data

Starting Code

App.js

The code starts from a very simple `App.js` which displays a grid with some car data.

```
import './App.css';

import {AgGridReact} from 'ag-grid-react';
import 'ag-grid-community/dist/styles/ag-grid.css';
import 'ag-grid-community/dist/styles/ag-theme-alpine.css';

import {useState, useRef, useEffect, useMemo, useCallback} from 'react';
import {createOneCarRecord} from './carFactory';

var numberFormatter = Intl.NumberFormat('en-US', {
  style: 'currency',
  currency: 'USD',
  maximumFractionDigits: 0
```

```
});
var myValueFormatter = p => numberFormatter.format(p.value);

let cars = [...new Array(4)].map(() => createOneCarRecord());

function App() {

  const gridRef = useRef();
  const [rowData, setRowData] = useState(cars);
  const [columnDefs, setColumnDefs] = useState([
        { field: 'type'},
        { field: 'year' },
        { field: 'color' },
        { field: 'price', valueFormatter: myValueFormatter}
  ]);

  return (
    <div className="ag-theme-alpine" style={{height: '100%'}}>
      <AgGridReact ref={gridRef}
          animateRows={true}
          columnDefs={columnDefs}
          />
    </div>
  );
}

export default App;
```

Value Formatter

The `numberFormatter` is a simple `valueFormatter` which takes the value in the row data and formats it to render it in USD format.

```
var numberFormatter = Intl.NumberFormat('en-US', {
  style: 'currency',
  currency: 'USD',
  maximumFractionDigits: 0
});
var myValueFormatter = p => numberFormatter.format(p.value);
```

The columns use value formatters when they are configured using the `valueFormatter` property on the column definition:

```
{ field: 'price', valueFormatter: myValueFormatter}
```

Full details for value formatters can be found in the documentation:

- https://www.ag-grid.com/react-data-grid/value-formatters/

Data Factory

To make populating the data simple, we have created a `carFactory.js` which will randomly create the car data rendered in the grid.

```
let sequence = 0;
let carTypes = ['Mazda MX5','BMW M3','Porsche 911',
                'Mercedes S-Class','Aston Martin DBX',
                'Bentley Bentayga'];
let colors = ['Red','Blue','Green','White','Black'];

export function createOneCarRecord() {
    const res = {
        id: sequence,
        type: carTypes[sequence%carTypes.length],
        year: 2010 + (sequence * 7) % 10,
        color: colors[sequence%colors.length],
        price: 20000 + (sequence * 3 * 17 * 19 * 5 * 7) % 40000
    }
    sequence++;
    return res;
}
```

This will help us easily manipulate the data without having to deal with any server side calls or interaction.

Updating all Data

An easy way to insert data is by amending the `rowData` values in the state.

We can create a simple button, with a handler to manipulate the state:

```
  return (
    <div className="ag-theme-alpine" style={{height: '100%'}}>
      <div>
        <button onClick={onInsertOne}>Insert One</button>
```

And the code to insert the data will be in the `onClick` handler function:

```
  const onInsertOne = useCallback( ()=> {
    const newRecord = createOneCarRecord();
    cars = [newRecord, ...cars];
    setRowData(cars);
  });
```

The `onInsertOne` handler, uses the `carFactory` utility function to create a new random record, and we add this to the front of a new array of rowData, and we amend the grid by calling `setRowData`.

This is the easiest way of updating data but we are completely refreshing the grid each time because the data is seen as 'new' by the grid.

Efficient Data Inserts

We can make the data inserts more efficient by adding row ids to the data so the grid knows to insert or update row data.

This is similar to the `key` attribute used by React for the same purpose with list elements.

We set the `getRowId` property on the grid with a mechanism for finding the id of the data.

```
      <AgGridReact ref={gridRef}

          getRowId={getRowId}
```

The `{getRowId}` implementation will be a callback function:

```
  const getRowId = useCallback( params => {
    console.log(params);
    return params.data.id;
  });
```

The `params` object has been written to the console so that when you run the code you can see that we are supplied with all the information we need to implement complex functionality:

- `api` - the full grid API
- `columnApi` - the column API
- `data` - which contains all the data for the row

Our callback function returns `params.data.id` because when the data was created by the `carFactory` one of the properties was a unique id:

- `id: sequence,`

This is not rendered as a `field` or column definition but is still present in the data that the grid holds.

When rows are inserted now, the grid will insert a new row and animate the existing rows downwards creating a smooth UI.

The row animation is configured using the `animateRows={true}` code in the grid definition, but it only comes into effect when the `getRowId` function has been configured.

Row Ordering

To demonstrate ro ordering we will create a new button which will reverse the data in the `rowData`.

The button is added to the JSX in `App.js`:

```
        <button onClick={onReverse}>Reverse</button>
```

And the handler code, will create a new array using the existing objects in the array but in reverse order:

```
  const onReverse = useCallback( () => {
    cars = [...cars].reverse();
    setRowData(cars);
  });
```

We will also configure the `type` column to be sortable:

```
{ field: 'type', sortable: true },
```

The order that records are added into `rowData` is the order that the rows are displayed, as you will see if you click the `Reverse` button.

But, when the `type` column has been sorted, the grid sorting is maintained even when the underlying rowData is re-ordered by the `onReverse` call.

Deleting Rows

We will demonstrate this by creating functionality to remove selected rows.

We need to configure the grid to allow row selection:

```
          rowSelection={'multiple'}
```

Then add a button which will trigger a function to remove selected rows:

```
<button onClick={onRemove}>Remove Selected</button>
```

Finally write the callback function code to remove the selected rows:

```
  const onRemove = useCallback( () => {
    const selectedNodes
            = gridRef.current.api.getSelectedNodes();
    const selectedIds = selectedNodes.map(
              node => node.data.id);
    cars = cars.filter(
              car => selectedIds.indexOf(car.id) < 0 );
    setRowData(cars);
  });
```

This uses the `gridRef` to access the api for the grid and return the selected nodes:

```
    const selectedNodes
            = gridRef.current.api.getSelectedNodes();
```

Then we add all the id's into an array of ids and filter out any rows which have that id.

Finally using `setRowData` to update the grid data.

The grid will detect that rows have been removed and animate the grid to nicely remove the rows.

Updating Rows

To demonstrate updating, we will create a button to trigger the update process:

```
        <button onClick={onUpdate}>Update Some</button>
```

Then the implementation will be a callback:

```
  const onUpdate = useCallback( () => {
    cars = cars.map( car => {
      if (Math.random()>0.5) { return car; }

      return {
        ...car,
        price: car.price
            + (1000-Math.floor(Math.random()*2000))
      }
    });
    setRowData(cars);
  });
```

To simulate an update process, the `onUpdate` function randomly updates some car records prices and then uses `setRowData` to update the data in the grid.

Because the row ids have been set, the grid will smoothly update only the cell values which have been updated, and React will only re-render the cells which have been updated.

To easily see the updates, the grid has built in functionality for rendering data changes. When the grid property `enableCellChangeFlash` is set to true then the cell background will change colour as it is update.

```
    enableCellChangeFlash={true}
```

Alternatively, the grid has a built in cell renderer called `agAnimateShowChangeCellRenderer` and when a column is configured to use this `cellRenderer` the grid will show the + or - difference in addition to highlighting the value in the cell.

```
        {
```

```
            field: 'price',
            valueFormatter: myValueFormatter,
            cellRenderer: 'agAnimateShowChangeCellRenderer'
          }
```

Remember to make sure the `enableCellChangeFlash` configuration is commented out when you do this.

Large Amounts of Data

When using large amounts of data the grid can use transactions instead of the `setRowData`.

Adding Data with Transactions

To demonstrate this we will create a button to update via transactions:

```
<button onClick={onTxInsertOne}>Tx Insert One</button>
```

The `onTxInsertOne` code to use the transaction is below:

```
  const onTxInsertOne = useCallback( ()=> {
    const newRecord = createOneCarRecord();
    gridRef.current.api.applyTransaction({
      add: [newRecord]
    });
  });
```

`onTxInsertOne` uses the `gridRef` to call the `applyTransaction` api call.

`applyTransaction` takes an object with the items to `add`, `update` or `remove`.

The documentation for `applyTransaction` is online:

- https://www.ag-grid.com/react-data-grid/data-update-transactions/

Transactions are a highly performant way of updating the grid.

We should note however that when we use transactions the `rowData` in the state is not kept in sync because the grid is now managing the row data. `rowData` in this example is now just being used to manage the initial state of the grid.

It is possible to use the api to retrieve the row data from the grid at any time. You can find api calls to do this in the documentation:

- https://www.ag-grid.com/react-data-grid/accessing-data/

Updating Data with Transactions

To demonstrate updating, we will add a button and a button listener which will perform the update transaction.

```
<button onClick={onTxUpdate}>Tx Update Some</button>
```

The `onTxUpdate` is a variation of our earlier update code, but this time:

- we use the grid api `forEachNode` method to iterate over all the grid data
- instead of using `setRowData` we create an array of all the records to update and then pass them in as the transaction `update` property:

```
  const onTxUpdate = useCallback( () => {
    const updatedRecords = [];

    gridRef.current.api.forEachNode( node => {
      if (Math.random()>0.5) { return; }

      const car = node.data;

      updatedRecords.push({
        ...car,
        price: car.price
            + (1000-Math.floor(Math.random()*2000))
      });
    });

    gridRef.current.api.applyTransaction({
      update: updatedRecords
    });
  });
```

We are using the `forEachNode` method because we are letting the grid manage the state of the data internally and `forEachNode` allows us to work with the data stored in the grid directly.

The `update` transaction uses the row id to match up the data and perform efficient updates and this is implemented using the `getRowId` property on the grid definition as covered earlier.

Removing Data with Transactions

To demonstrate removal with transactions we will once again create a button and an event handler.

```
<button onClick={onTxRemove}>Tx Remove Selected</button>
```

The transaction handler is shown below:

```
  const onTxRemove = useCallback( () => {
    const selectedNodes
            = gridRef.current.api.getSelectedNodes();
    const selectedData = selectedNodes.map(
              node => node.data);

    gridRef.current.api.applyTransaction({
      remove: selectedData
    });
  });
```

The handler uses the grid api to get the current selected nodes, adds the data object to a `selectedData` array which is then used as a the `remove` property of the transaction object.

High Frequency Updates

The transaction approach above is good for large amounts of data. When we work in a high frequency update environment we should consider using asynchronous transactions.

Asynchronous Transactions

Asynchronous transactions are a way of grouping updates together which reduces the number of renders required to keep the grid up to date.

To demonstrate we will create a button and listener to add a record asynchronously.

```
<button onClick={onTxAsyncInsertOne}>Tx Async Insert One</button>
```

The listener:

```
  const onTxAsyncInsertOne = useCallback( ()=> {
    const newRecord = createOneCarRecord();
    gridRef.current.api.applyTransactionAsync({
      add: [newRecord]
    });
  });
```

Some differences between `applyTransactionAsync` and `applyTransaction`.

- `applyTransaction` would return a result object.
- `applyTransactionAsync` takes a callback function to handle any results

e.g.

```
    gridRef.current.api.applyTransactionAsync({
      add: [newRecord]
    }, res => {
      console.log(res);
    });
```

The difference between the transaction types will not be noticeable when manually triggered. Behind the scenes the `applyTransactionAsync` is waiting for 50 milliseconds before applying the transaction in case any additional transactions are added to the transaction cache, and if so they are batched in and processed together.

Asynchronous Transaction Times

The time that the grid waits is configurable with a grid property `asyncTransactionWaitMillis`.

By changing the `asyncTransactionWaitMillis` property to be 5 seconds, there will be a noticeable delay between clicking the button and the transaction taking effect.

```
    asyncTransactionWaitMillis={5000}
```

Asynchronous Transaction Events

When the batched transaction is complete, the grid triggers an `AsyncTransactionsFlushed` event and it is possible to listen for this event, should we want to have follow on processing to handle the results associated with the batched transaction.

A listener is shown below, and this would write the resulting object to the console.

```
  const onAsyncTxFlushed = useCallback( e => {
    console.log('==============');
    console.log(e);
    console.log('==============');
  }, []);
```

We would also have to set the event handler in the grid properties:

```
    onAsyncTransactionsFlushed={onAsyncTxFlushed}
```

There is one more feature of asynchronous transactions that it is important to know and that is flushing the asynchronous cache.

Flushing Asynchronous Transaction Cache

We can programmatically bypass the `wait` time and immediately process everything in the transaction cache by using the `flushAsyncTransactions` api call.

To demonstrate we will code a button with an on click handler that calls the API.

The button:

```
    <button onClick={onFlushAsyncTx}>Flush Async Tx</button>
```

The on click handler:

```
  const onFlushAsyncTx = useCallback( () => {
    gridRef.current.api.flushAsyncTransactions();
  }, []);
```

When the button is clicked, any transactions in the cache will be immediately processed, rather than allowing the wait time to timeout.

The puts much more control over the transaction processing and handling in the hands of the programmer.

Summary

`setRowData` is an easy way to add, remove and update values in the grid. This uses the react component state store to manage the data.

To allow the grid to efficiently match up insertions, removals and updates the `getRowId` callback on the grid needs to be configured so that rows can be identified uniquely via an `id`.

Transactions are used when there are many rows to update or the data in the grid is large, this is much more efficient. The difference is that when using transactions the grid itself is maintaining the row data store, not the React component.

Asynchronous Transactions are used for high frequency updates, these are cached until the configurable timeout for the async cache is reached. The cache can be flushed and the transactions processed immediately with an API call.

End Notes

Thanks for following the tutorial. You should now be able to implement and customize AG Grid within your React projects.

The documentation for the grid has more examples and goes deep into the many options available via the API.

- https://ag-grid.com/react-data-grid

We also have a blog where you can find case studies and more tutorials:

- https://blog.ag-grid.com/

If you prefer video, then you can subscribe to our YouTube channel:

- https://www.youtube.com/ag-grid

And you can follow us on Social Media:

- https://twitter.com/ag_grid
- https://www.linkedin.com/company/ag-grid
- http://instagram.com/ag_grid

We build AG Grid in the open, and the Community Edition is MIT Licensed, you can find our code on Github:

- https://github.com/ag-grid

If you haven't yet signed up for our mailing list, we announce new releases and periodically share tutorial information via email. You can sign up online:

- https://blog.ag-grid.com/newsletter/

This document was collated and edited by Alan Richardson, Head of Digital Marketing at AG Grid

- *https://blog.ag-grid.com/author/alan/*