



INSTANT

Short | Fast | Focused

jqGrid

Learn how to use the powerful jqGrid library to manage your data
from the frontend

Gabriel Manricks

[PACKT]
PUBLISHING

Table of Contents

[Instant jqGrid](#)

[Credits](#)

[About the Author](#)

[About the Reviewer](#)

[www.packtpub.com](#)

[Support files, eBooks, discount offers and more
packtlib.packtpub.com](#)

[Why Subscribe?](#)

[Free Access for Packt account holders](#)

[1. Instant jqGrid](#)

[So, what is jqGrid?](#)

[The grid](#)

[Installation](#)

[Quick start – creating your first grid](#)

[Top 7 features you need to know about](#)

[Formatting data](#)

[Adding controls](#)

[The pager](#)

[The navigator](#)

[Editing data](#)

[The editable option](#)

[The edittype and editoptions options](#)

[The editrules option](#)

[The formoptions option](#)

[Configuring the modals](#)

[Communicating with jqGrid](#)

[Editing data server-side](#)

[Searching in jqGrid](#)

[Interfacing the API](#)

[Calling methods](#)

[Events](#)

[People and places you should get to know](#)

[Official sites](#)

[Articles and tutorials](#)

[Community](#)

[Twitter](#)

Instant jqGrid

Instant jqGrid

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: July 2013

Production Reference: 1180713

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78328-991-2

www.packtpub.com

Credits

Author

Gabriel Manricks

Reviewer

Tony Tomov

Acquisition Editor

Usha Iyer

Commissioning Editor

Nikhil Chinnari

Technical Editors

Pratik More

Veena Pagare

Project Coordinator

Amigya Khurana

Proofreader

Clyde Jenkins

Production Coordinator

Manu Joseph

Cover Work

Manu Joseph

Cover Image

Ronak Dhruv

About the Author

Gabriel Manricks is a full-stack software and web developer focusing on PHP and both frontend and server-side JavaScript frameworks.

Gabriel works as a staff writer for NetTuts+, where he enjoys learning as well as teaching others. He is also a freelancer in web consulting, development, and writing.

I would like to thank my family for always supporting me. I would like to thank Tony Tomov for his enthusiasm and help in writing this book, and to Packt Publishing for offering me this amazing opportunity.

About the Reviewer

Tony Tomov has a Bachelor's and a Master's degree from the Computer Systems and Automatic Control Faculty at the Technical University of Sofia, Bulgaria.

His main interests cover a wide areas of computer languages, programming, and new web technologies. In 2004, Tony founded Trirand Inc. The company specializes in web development and has developed multiple projects for client-side all over the world.

Tony is the author of the most popular jQuery grid plugin—jqGrid. His current work is focused on creating a suite of server-side and client-side components based around jqGrid and covering a wide spectrum of technologies—PHP, ASP.NET WebForms, and ASP.NET MVC.

www.packtpub.com

Support files, eBooks, discount offers and more

You might want to visit www.packtpub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [<service@packtpub.com>](mailto:service@packtpub.com) for more details.

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

packtlib.packtpub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.packtpub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.



Chapter 1. Instant jqGrid

Welcome to *Instant jqGrid*. This book invites you into the world of jqGrid starting from setup, to customizing grids, and all the way up to extending the default functionalities through its extensive API.

This document contains the following sections:

So, what is jqGrid? will help us take a look at what jqGrid really is, as well as learn a little background about the project and what problems it can solve.

Installation will go through the actual files needed to get started and create a template that we can use in future examples.

Quick Start – creating your first grid is where we will create our first working grid, and learn the minimum that is required in order to get started. This is an important step in understanding what's really happening.

Top 7 features you need to know about is where we will take a look at some of the most useful and common features that jqGrid has to offer. By the end of this section, you will have a working knowledge of how to create pretty complex grids, as well as how to add your own custom functionality.

People and places you should get to know will help us learn about the different people and places that you can check out to stay informed. Learning jqGrid is only the beginning; staying up-to-date with new features and updates is key.

So, what is jqGrid?

If you take a look at the official definition, you will find something along the lines of: jqGrid is an AJAX-enabled JS component for viewing and editing tabular data on the web.

This is a great method for including as many of jqGrid's features in one sentence as possible, but to explain it a little more verbosely, jqGrid is a jQuery plugin, created by Tony Tomov at *Trirand Inc.*, for displaying and editing data in tables.

I think it is safe to say that most web applications use and store data in one form or another, and a lot of your time as a developer, at least at the beginning, goes into creating a lot of boilerplate code for inserting, modifying, and removing data from the database (the basic C.R.U.D. operations).

Well, Tony had the idea to create a fast, yet flexible plugin, which could take care of this, and jqGrid is the culmination of that idea.

From the very start, jqGrid embraced a lot of best practices and open standards, making it a solid investment to learn.

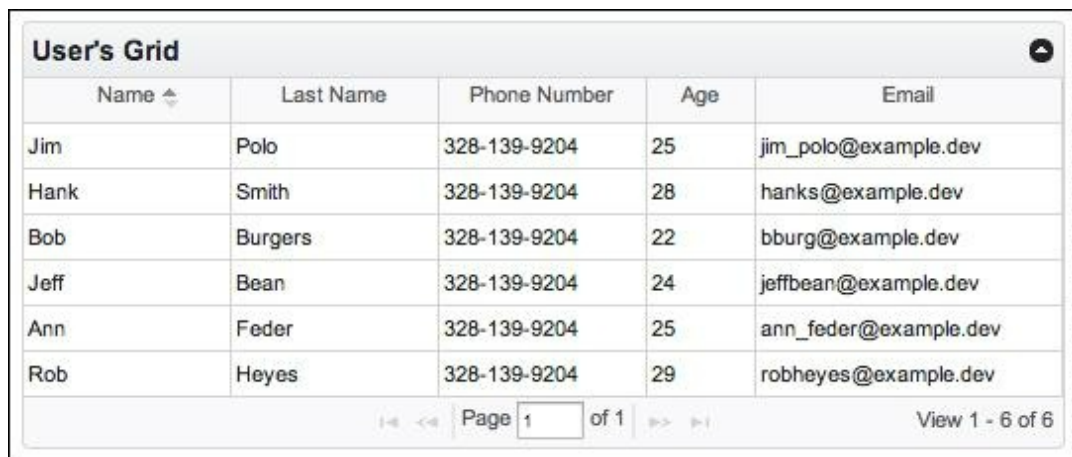
First off, jqGrid makes no assumptions about your backend, this makes it future-proof and very unobtrusive to use, because any—present or future—languages and frameworks can be used with it.

Secondly, even though jQuery UI is not a dependency, jqGrid follows its standard for developing themes, giving you the ability to create your own theme using jQuery UI's theme roller, and if you are already using jQuery UI in your app, the grid will fit right in, utilizing the same styles.

The grid

Everything in jqGrid revolves around—you may have guessed it by now—a grid. It's the common resource where you will be displaying and modifying your data, so it's important to understand how it works.

In its simplest form, a grid can be used to represent a table of data; take a look at the following screenshot:



Name	Last Name	Phone Number	Age	Email
Jim	Polo	328-139-9204	25	jim_polo@example.dev
Hank	Smith	328-139-9204	28	hanks@example.dev
Bob	Burgers	328-139-9204	22	bburg@example.dev
Jeff	Bean	328-139-9204	24	jeffbean@example.dev
Ann	Feder	328-139-9204	25	ann_feder@example.dev
Rob	Heyes	328-139-9204	29	robheyes@example.dev

Page 1 of 1 View 1 - 6 of 6

This screenshot represents a single grid, made up of four layers:

- The **caption** layer
- The **header** layer
- The **body** layer
- The **navigation** layer

These are the standard layers, but you can add more or less, to fit your application's needs.

The caption layer is the first row at the top, where it says **User's Grid** in the previous screenshot. Not much to say about it, the layer is used to provide a header and some context as to what the table below contains.

Next, we have the header and body layers; these are what most people know of as a table. You have the column titles (the header layer) and the rows of values (the body layer).

Last but not least, we have a navigation layer. This is used to provide controls for the grid, such as the pagination control you can see in the previous screenshot.

The main concept you should take home is how the grids are made up of stacked layers. Throughout the book we will be customizing the grid, adding and removing things to illustrate different points, but if you always remember this structure, it will give you perspective and a hint as to how or why something is implemented in a certain way.

Installation

jqGrid is a JS library, so installation is as simple as including a link in your HTML files to the scripts. As for dependencies, you will also need to include jQuery, and a jQuery UI CSS file (jQuery UI itself is not required).

So let's get started, create a folder named `jqGrid` on your computer, and then create two folders within it, one named `js` and the other named `css`.

Next, go to <http://jquery.com/>, and download the latest version to the `js` folder that we just made. I am also going to rename mine to `jquery.js` (for example, `jquery-1.9.1.min.js` to `jquery.js`).

Next, we need to download jqGrid itself from <http://www.trirand.com/blog>, click on the **Download** button in the menu, and you will be able to customize the package. You can leave everything checked for now, so just download the entire library.

Once the ZIP is downloaded, open it, and we will need three files from here. First, open the `js` folder, and copy `jquery.jqGrid.min.js` to our `js` folder (the one where we put jQuery). Next, open the `i18n` folder, and copy the file that corresponds to the language you want to use; I am going to use English, so I will copy `grid.locale-en.js` to our `js` folder. Then the last file we need is the CSS file, so copy the `ui.jqgrid.css` file to our `jqGrid | css` folder.

Once done, your structure should look something like:



The last thing we will need is the jQuery UI CSS file, like I mentioned, jqGrid takes advantage of the jQuery UI CSS framework (not jQuery UI itself) and, as such, you can find a lot of themes available online for free, as well as tools to create your own.

To download one of the default themes, or even create your own, you can just go to <http://jqueryui.com/themeroller/>, and there is a utility for selecting a base theme and modifying it to fit your site. And like I said, this is pretty

much a standard when it comes to jQuery widgets, so a quick search online will find you many themes made by others as well.

Once you download and unzip it, you will need to navigate to the theme's folder, usually contained inside a directory named `css`, so if your theme was named `ui-lightness`, then you would navigate to `css | ui-lightness` inside the downloaded folder.

Once inside, you will find a folder named `images`, and a CSS file; copy these two things to our jqGrid's `css` folder to finish the installation. Your final structure should look something like this:



Now, to finish our template, let's create a file named `index.html`, which will include the necessary code to get started. The file should be placed at the root of the `jqGrid` folder:

```
<!DOCTYPE html>
<html>
  <head>
    <title>jqGrid Template</title>
    <link rel="stylesheet"
href="css/jquery.ui.theme.css" />
    <link rel="stylesheet"
href="css/ui.jqgrid.css" />
    <script src="js/jquery.js"></script>
    <script src="js/grid.locale-en.js"></script>
    <script
src="js/jquery.jqGrid.min.js"></script>
  </head>
  <body>
  </body>
</html>
```

And of course, switch the first CSS file with the name of the one you downloaded, and if you choose a different language, then change the language file accordingly.

Tip

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Quick start – creating your first grid

As you may have seen, there is a little bit of tweaking necessary in order to customize jqGrid and get it set up the way you like. The good news is, you can use that folder as a template and every time you want to make a new project with jqGrid you can just make a copy of that folder.

So, let's do that right now for this example. Create a copy of the entire jqGrid folder named `quickstart`; easy as pie.

Let's begin by creating the simplest grid possible; open up the `index.html` file from the newly created `quickstart` folder, and modify the body section to look like the following code:

```
<body>
  <table id="quickstart_grid"></table>

  <script>
    var dataArray = [
      {name: 'Bob', phone: '232-532-6268'},
      {name: 'Jeff', phone: '365-267-8325'}
    ];

    $("#quickstart_grid").jqGrid({
      datatype: 'local',
      data: dataArray,
      colModel: [
        {name: 'name', label: 'Name'},
        {name: 'phone', label: 'Phone Number'}
      ]
    });
  </script>
</body>
```

The first element in the `body` tags is a standard `table` element; this is what will be converted into our grid, and it's literally all the HTML needed to make one.

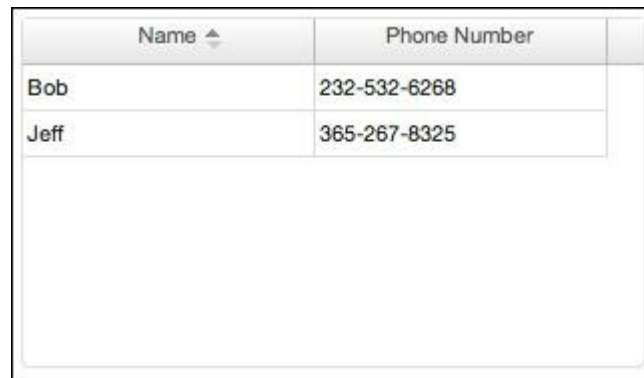
The first four lines are just defining some data. I didn't want to get into AJAX or opening other files, so I decided to just create a simple JavaScript array with two entries. The next line is where the magic happens; this single command is being used to create and populate a grid using the information provided.

We select the `table` element with jQuery, and call the `jqGrid` function, passing it all the properties needed to make a grid. The first two options set the data along with its type, in our case the data is the array we made and

the type is `local`, which is in contrast to some of the other datatypes which use AJAX to retrieve remote data.

After that we set the columns, this is done with the `colModel` property. Now, there are a lot of options for the `colModel` property, which we will get to later on, numerous settings for customizing and manipulating the data in the table. But for this simple example, we are just specifying the name and label properties, which tell jqGrid the column's label for the header and the value's key from the data array.

Now, open `index.html` in your browser and you should see something like the following screenshot:



Name ▲	Phone Number
Bob	232-532-6268
Jeff	365-267-8325

Not particularly pretty, but you can see that with just a few short lines, we have created a grid and populated it with data that can be sorted.

But we can do better; first off, we are only using two of the four standard layers we talked about: the header layer and the body layer. Let's add a caption layer to provide little context, and let's adjust the size of the grid to fit our data.

So, modify the call to `jqGrid` with the following:

```
$("#grid").jqGrid({
  datatype: 'local',
  data: dataArray,
  colModel: [
    {name: 'name', label: 'Name'},
    {name: 'phone', label: 'Phone Number'}
  ],
  height: 'auto',
  caption: 'Users Grid',
});
```

And refresh your browser; your grid should now look like the following screenshot:

Users Grid	
Name	Phone Number
Bob	232-532-6268
Jeff	365-267-8325

That's looking much better. Now, you may be wondering, we only set the `height` property to `auto`, so how come the width seems to have snapped to the content? This is due to the fact that the right margin we saw earlier is actually a column for the scroll bar. By default jqGrid sets your grid's height to 150 pixels, this means, regardless of whether you have only one row, or a thousand rows, the height will remain the same, so that there is a gap to hold the scroll bar in an event when you have more rows than that would fit in the given space. When we set the `height` to `auto`, it will stretch the grid vertically to contain all the items, making the scroll bar irrelevant and therefore it knows not to place it.

Now this is a pretty good quick start example, but to finish things off, let's take a look at the navigation layer, just so we can say we did.

For this next part, although we are going to need more data, I can't really show pagination with just two entries, luckily there is a site <http://www.json-generator.com/> created by Vazha Omanashvili for doing exactly this.

The way it works is, you specify the format and number of rows you want, and it generates it with random data. We are going to keep the format we have been using, of name and phone number, so in the box on the left, enter the following code:

```
[
  '{{repeat(50)}}',
  {
    name: '{{firstName}} {{lastName}}',
    phone: '{{phone}}'
  }
]
```

Here we're saying we would like 50 rows with a name field containing both firstname and lastname, and we would also like a phone number. This site is not really in the scope of this book, but if you would like to know about other fields, you can click on the **help** button at the top.

Click on **Generate** to produce a result object, and then just click on the **copy to clipboard** button. With that done, open your `index.html` file and replace the array for `dataArray` with what you copied. Your code should now look like the following code:

```
var dataArray = {
```

```

        "id": 1,
        "jsonrpc": "2.0",
        "total": 50,
        "result": [ /* the 50 rows */ ]
    };

```

As you can see, the actual rows are under a property named `result`, so we will need to change the data key in the call to `jqGrid` from `dataArray` to `dataArray.result`. Refreshing the page now you will see the first 20 rows being displayed (that is the default limit). But how can we get to the rest? Well, `jqGrid` provides a special navigation layer named "pager", which contains a pagination control. To display it, we will need to create an HTML element for it. So, right underneath the `table` element add a `div` like:

```

<table id="grid"></table>
<div id="pager"></div>

```

Then, we just need to add a key to the `jqGrid` method for the pager and row limit:

```

    caption: 'Users Grid',
    height: 'auto',
    rowNum: 10,
    pager: '#pager'
});

```

And that's all there to it, you can adjust the `rowNum` property to display more or less entries at once, and the pages will be automatically calculated for you.

Users Grid	
Name	Phone Number
Melanie Michaelson	860-469-3319
Sophie Wesley	855-480-2186
Zoey Webster	848-471-3090
Sofia Chesterton	865-570-2949
Paige Webster	886-515-2754
<< << Page 4 of 10 >> >>	

Top 7 features you need to know about

jqGrid's flexibility is all in the configuration, and the way it lets you adjust all the finer details. Although it would be quite hard to make a single example that would encompass all of them, the section will be broken up into multiple, and more specific examples, each making a stronger point.

So far from the *Quick start – creating your first grid* section, you should have a basic understanding of how the grid is set up and have a better grasp on the concepts mentioned in the introduction. The next thing I want to talk about is formatting data into the cells.

Formatting data

In the last example we had very basic data, just two strings, but chances are, in a real world example, you will be getting your data from some backend, and there will be some fields that are not display ready. jqGrid provides formatters which you can customize to display, really anything, based on the value of a field. To get started, copy and paste the template we created for new projects, and let's set up a basic grid. Inside the `body` tags of `index.html`, add the `table` and `script` tags:

```
<body>
  <table id="grid"></table>
  <script>
    var data;
    $("#grid").jqGrid({
      //properties go here
    });
  </script>
</body>
```

Next, we need some data to demonstrate formatters; I am going to use three of the most common field types you would want to modify: timestamps, Booleans, and URLs. To generate this data I am going to use the same tool from the previous section (<http://www.json-generator.com/>). The code for the template is as follows:

```
[
  '{{repeat(10)}}',
  {
    name: '{{firstName}} {{lastName}}',
    timestamp: '{{numeric(1104559200,
1357020000)}}',
    verified: '{{bool}}',
```

```

        website:
        'http://{{firstName}}{{lastName}}.com'
    }
]

```

Click on the **Generate** button and then on the **Copy to clipboard** button like the last time; then just paste this after the data variable, so it should look like:

```
var data = //pasted data
```

Now, let's set up the column and data properties to get this grid working:

```

$("#grid").jqGrid({
    datatype: 'local',
    data: data.result,
    colModel: [
        { name: 'name', label: 'Name' },
        { name: 'timestamp', label: 'Date' },
        { name: 'verified', label: 'Verified' },
        { name: 'website', label: 'Website' }
    ]
});

```

Open this in your browser and you should see a basic grid with the columns:

Name	Date	Verified	Website
Allison Davidson	1344849677	true	http://AllisonDavidson.com
Emma Gibbs	1283066820	true	http://EmmaGibbs.com
Amelia Clapton	1210313193	false	http://AmeliaClapton.com
Grace Nathan	1238714128	false	http://GraceNathan.com
Brooklyn Chandter	1113240752	true	http://BrooklynChandter.com
Mackenzie Hailey	1240318065	true	http://MackenzieHailey.com
Sarah Gardner	1242026830	true	http://SarahGardner.com

The default formatter will just print out whatever is stored, no matter the type, and while this is helpful, we can do better. jqGrid comes with a handful of pretty good formatters to use, and they are usually more than enough. You can view the full list of formatters at http://www.trirand.com/jqgridwiki/doku.php?id=wiki:predefined_formatter.

Let's start with the **Date** column, it contains a timestamp and we want to convert it into a human-readable date. jqGrid comes with a built-in date formatter, which we can use and it follows the PHP standard for formatting, you can view the full list at <http://php.net/manual/en/function.date.php>. For your convenience, I have included the following list of some of the more common format specifiers:

d – padded day (01-31)	D – short day (e.g. Mon, Thu)	U – timestamp
j – unpadded day (1-31)	l – full day (e.g. Sunday)	h – padded hour (01-12)
m – padded month (01-12)	M – short month (e.g. Dec)	g – unpadded hour (1-12)
n – unpadded month (1-12)	F – full month (e.g. June)	i – padded minutes (00-59)
Y – year (e.g. 2013)	S – ordinal suffix (e.g. th, st)	s – padded seconds (00-59)

The way it works is you build a string, with the desired specifiers, and each one will be replaced with its corresponding value. The jqGrid date formatter requires an input format and an output format, and these settings default to `Y-m-d` (for example, 2013-03-16) for the input and `d/m/Y` (for example, 16/03/2013) for the output. We have a timestamp for the input, so we will need to change it to a `U` and for the output let's go for a very readable format of `M jS Y` (for example, Aug 13th 2012). The final `colModel` with the previous listed changes looks like:

```
{
  name: 'timestamp',
  label: 'Date',
  formatter: 'date',
  formatoptions:{ srcformat: 'U', newformat: 'M
jS Y' }
},
```

The name stays the same as that's used for the key in the data array, but you can see we added a custom formatter and some options for it; namely, we specify that we are inputting a timestamp and the desired output format. Refresh the page and you will see that with just two simple options we have greatly simplified the grid. Next, let's format the URL column, here we want to make it output an anchor tag, instead of just the plaintext.

This is easier than it sounds, because jqGrid comes with another formatter for this, the `link` formatter. There are no other options that need adjusting, so we can just change the links `colModel` to:

```
{ name: 'website', label: 'Website', formatter:
'link' }
```

Refresh and you will see the website strings are now clickable, almost done. The last column we want to be formatted is the Boolean column. This column holds a native `True` or `False` value, but this isn't the best output for the table. For example's sake, let's make this column show **Yes** or **No** based on the value. Here we are going to create a custom formatter, the way you do this is you pass a function instead of a formatter's name. Change the verified `colModel` to:

```
{ name: 'verified', label: 'Verified', formatter:
isVerified },
```

Now this function will be called for each row and whatever is returned by the function gets displayed in the table. The function receives three arguments you can use, the first is the actual value of the current cell (in our case this will be `True` or `False`), the second parameter is the properties of the current column, and the third parameter is the data object for this row, this allows you to do complex things based on properties besides the cell's own value.

For our case though, we are just building a function to return `Yes` or `No`, so it is going to be quite simple; here is the complete function, add it after the `jqGrid()` call, before the closing `script` tag:

```
function isVerified (val, cellProps, rowData){
    if (val) {
        return "Yes";
    } else {
        return "No";
    }
}
```

What we are doing is just checking if the cell's value is true, in which case we return the word **Yes**, otherwise we return **No**; extremely simple (and yes this could have been done with a ternary operator in a single line).

Opening the page now, your grid should look something like the following screenshot:

Name ↕	Date	Verified	Website
Allison Davidson	Aug 13th 2012	Yes	http://AllisonDavidson.com
Emma Gibbs	Aug 29th 2010	Yes	http://EmmaGibbs.com
Amelia Clapton	May 9th 2008	No	http://AmeliaClapton.com
Grace Nathan	Apr 3rd 2009	No	http://GraceNathan.com
Brooklyn Chandter	Apr 11th 2005	Yes	http://BrooklynChandter.com
Mackenzie Hailey	Apr 21st 2009	Yes	http://MackenzieHailey.com
Sarah Gardner	Mar 21st 2009	Yes	http://SarahGardner.com

Compare it to the screenshot at the beginning of the section and you will see we have come a long way. With the data in our table looking good, let's move onto adding controls.

Adding controls

There are a couple of controls jqGrid comes bundled with, the first of which is the pager.

The pager

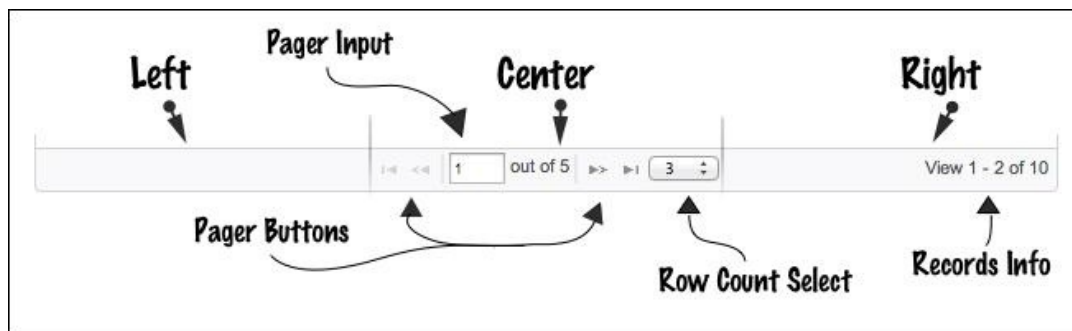
By default adding a pager to your grid creates some standard pagination controls, like we saw in the *Quick start – creating your first grid* section, but there is a lot more to it under the surface. You can think of the pager as the base for all the other custom controls in your grid. I am going to continue with the same grid as we have been working on, just add a `div` tag to hold the pager element and a property in the call to `jqGrid`:

```
<body>
  <table id="grid"></table>
  <div id="pager"></div>
  <script>
    var data = { /* the data */ };
    $("#grid").jqGrid({

      /* other settings */

      pager: "#pager"
    });
  </script>
</body>
```

Refreshing the page in your browser will now reveal the pagination control as mentioned earlier. To continue on with customizing the pager, you will need to learn the pager's terminologies, which I have laid out in the following diagram:



The pager is split into three sections, and has four components built in:

- **The buttons:** These are used for traversing back, forth, first page, and last
- **The input:** This includes the text for how many pages, as well as the input to set the exact page you want
- **The row count select:** This is used to set how many rows should be displayed at a time
- **The record info:** This is just a string which can display which records are currently being displayed, as well as the total number of records

By default, the buttons and input are turned on, whereas the row count select and record info are turned off. All these components are configurable, and the easiest way to show you is to just give you an

example using all of them, so right under the pager option, we just add the rest of these options:

```
pager: "#pager",
pagerpos: "center",
recordpos: "right",
pgbuttons: true,
pginput: true,
viewrecords: true,
rowNum: 3,
rowList: [3, 5, 10],
pgtext: "{0} out of {1}",
recordtext: "Showing [{0} - {1}] of {2}"
});
```

Refresh the page and your grid should now look like the following screenshot:

Name ↕	Date	Verified	Website
Allison Davidson	Aug 13th 2012	Yes	http://AllisonDavidson.com
Emma Gibbs	Aug 29th 2010	Yes	http://EmmaGibbs.com
Amelia Clapton	May 9th 2008	No	http://AmeliaClapton.com

1 out of 4 3 Showing [1 - 3] of 10

I am now going to go through each of these options, providing a little more information:

- `pagerpos` and `recordpos`: These two options are used for positions of the pager controls and the records info respectively. Possible values for this property are `left`, `center`, or `right`, which will position it in one of those slots on the pager bar. By default the pager controls are set to `center`, and the record info is set to `right`.
- `pgbuttons`, `pginput`, and `viewrecords`: These three are Booleans determining whether the buttons, input, and records info should be shown respectively. If you set one of these keys to `True`, it will show the appropriate control, whereas `False` will hide it. By default the pager buttons and input are set to `True`, whereas the records info is set to `False`.
- `rowNum`: You should be familiar with this property as we used it in the *Quick start – creating your first grid* section, but it is used to adjust how many records are shown on the page at once; the value for this key is an integer. By default it is set to show twenty rows.
- `rowList`: This property is used to create the `select` element in the pager controls, which allows the user to change how many rows are

being displayed at once. This property accepts an array with the possible options; so for example, if you want the user to be able to choose between showing 5 rows at a time or 10, you would simply set this option to `[5, 10]` and it would create the necessary `select` box. If the property is set to an empty array, like it is by default, the `select` box won't be made.

- `pgtext` and `recordtext`: These two options set the text for the pager, by default these are taken from the language file that you selected, but you can overwrite them as you see where it fits. The way it works is you create a string with placeholder for it to attach the dynamic data.

Let's start with the first one, `pgtext`; this property has two possible placeholders, the first is for the current page number, and the second contains the total number of pages. The placeholders are referenced numerically in that order, with `{0}` and `{1}` respectively.

The `recordtext` property has three placeholders, the first contains the first record's number that is being shown, the second placeholder holds the index of the last number currently being displayed, and the third placeholder contains the total number of records altogether. These are referenced the same way with `{0}`, `{1}`, and `{2}`.

Note

These are all the options available for the pager, if you would like to learn more go to <http://www.trirand.com/jqgridwiki/doku.php?id=wiki:pager>.

The navigator

I mentioned that the pager is the base where you can add other custom controls; well the navigator is a set of common controls that come built-in for you to add and use. The six controls that come included are:

- The **New** button to add rows
- The **Edit** button to edit the currently selected row
- The **View** button to view the currently selected row
- The **Delete** button to delete the currently selected row
- The **Find** button to search for specific rows
- The **Reload** grid button to refresh everything

The navigator is added to the pager by calling the `navGrid` jqGrid method and passing through the desired options. To call a jqGrid method, you use the standard jqGrid function on the grid and you pass the requested method

as the first parameter. Here is an example also showcasing the navigator's parameters:

```
$("#grid").jqGrid('navGrid', "#pager", props,  
editOp, addOp, delOp, searchOp, viewOp);
```

This needs to be done after you initialize the grid with the first call to `jqGrid`. You may have noticed that there are a lot of parameters, but this is to be expected due to the nature of this function. What we are doing is creating a toolbar for handling basically every possible scenario you would need to cover, because of this we have the ability to pass in options for each of these situations individually.

Let's start by getting the bare minimum working; just add the following line of code right after the current call to `jqGrid`:

```
$("#grid").jqGrid('navGrid', "#pager", {});
```

This will add the navigator using all the default settings, refresh the browser and try it all out. Playing with it for a little bit, you will notice that the add and edit modals don't work at all, and delete pops up an error because no URL has been provided. This is not a bug, but is because we have not set up a backend to our grid. These tools (add, edit, and delete) are meant to make changes that persist to the server itself, this of course requires some additional configuration both here and in your backend code, but we will get to that later.

For now, just know that these features are here, and it's good to know that a lot of the forms and modals are provided by default.

The **Search** and **Reload** buttons will work, but you will notice the search is happening on the data level and not on what is actually being displayed; so for example, if you search for rows where verified is **Yes**, nothing will come up, although if you search for rows where verified is **True** or **False**, the correct rows will show up.

The last thing worth mentioning here is the sixth button, the **View** button, didn't even show up; that is because it is turned off by default. So far, not a great start on controls, but with a little setup we can get this working just the way you want.

Let's start with the first options parameter to `navGrid` (which is actually the third parameter in the function call), and let's take a look at what we are able to set up:

- `add`, `edit`, `del`, `view`, `search`, and `refresh`: Most of the properties for the navigator come in six pairs, one for each of the buttons, and these are no exceptions. These six properties allow you to turn on and off

their corresponding buttons by specifying a Boolean value of `True` or `False` respectively. By default they are all set to true except the **View** button as we just saw.

- `addicon`, `editicon`, `delicon`, `viewicon`, `searchicon`, and `refreshicon`: These properties allow you to specify the icon on the individual buttons. Because jqGrid uses the jQuery UI CSS framework, you need to specify one of the icons from the pack. The actual icons might look different depending on the theme you are using, but if you want to find out the names, you can hover over the icon pictures on the official theme roller <http://jqueryui.com/themeroller/>. By default these are set to the strings for the icon's classes; for instance the `addicon` default is `ui-icon-plus`.
- `addtext`, `edittext`, `deltext`, `viewtext`, `searchtext`, and `refreshtext`: These next six are for setting the text inside each button; by default this is an empty string, which is why there was no text inside them. Not much to talk about here, you just write a string and it will be the button's text.
- `addtitle`, `edittitle`, `deltitle`, `viewtitle`, `searchtitle`, and `refreshtitle`: Similar to the previous options, these are for setting the title of the links, which is what gets displayed in the little pop over when you hover on top of a button. By default these are set inside the language file you selected to strings, such as `Add new row` for the **Add** button or `Edit selected row` for the **Edit** button.
- `addfunc`, `editfunc`, `delfunc`, `viewfunc`, and `searchfunc`: These five properties allow you to override the default behaviors of these buttons. Basically, if you assign a function to these properties, the attached function will be called instead of the default (the one that creates the modals). The `editfunc`, `delfunc`, and `viewfunc` properties take a function that accepts one parameter, the index of the selected row, whereas the other two (`addfunc` and `searchfunc`) do not receive anything.
- `alertcap` and `alerttext`: These two properties set the title and text of the error modal that comes up when you click on the **Edit/Delete/View** buttons without selecting a row. By default these strings are taken from the language file; in the English file `alertcap` is set to `Warning` and `alerttext` is set to `Please select a row`.
- `position`: This is the same as we had with the pager element; you can set this to `left`, `right`, or `center` to position it on the three sections of the bar. By default this is set to `left`.

Note

Those are the main settings you would probably edit; you can find the full list here: <http://www.trirand.com/jqgridwiki/doku.php?id=wiki:navigator>

The rest of the parameters to the `navGrid` method are for the individual functions, some of which we will now go through.

Editing data

The fourth parameter, right after the navigator's own properties is the properties for the edit functionality. If you remember from the test, the add and edit modals came up blank. This is because these need special configuration in two or three different locations. If you think about it, just taking the cells from the grid and displaying it wouldn't work, because first off, that might only be part of the data. Let's say our data was showing invoices for a specific user; there would be no user column in the table because they all belong to the same person, but to save it to the server you would need this information. The other thing is, even once it knows what columns to create, it doesn't know what types of data are being edited. You might want to constrain the fields to specific values; for instance, you might have a field which is a string that contains the user's city, and you may not want the user to be able to enter anything. You may have only specific cities that your backend understands, so in this case you would want to put a select box instead of a text field.

All these options are available for you to configure in the `colModel`. There are basically five different options you will possibly want to set when configuring them for adding/editing, and they are: `editable`, `edittype`, `editoptions`, `editrules`, and `formoptions`.

The editable option

The first one, `editable`, is just a Boolean of whether or not this field is well, editable. By setting this to `True`, it will appear in the add/edit pop ups using all the other defaults. You can give this a try for the first `colModel`, the one for name. Just change that row to:

```
{ name: 'name', label: 'Name', editable:true }
```

Now, refresh the page in your browser and simply clicking on the **add** or **edit** buttons will bring up the modals, which will now contain a field for name. Submitting the data will still not work, but that is just because we haven't connected the grid to a backend.

The edittype and editoptions options

These properties are used to configure the type of input that shows up for the given column. The default `edittype` is text for a text box, which is exactly what comes up for the name field we just tried out. Other types include: `textarea`, `select`, `checkbox`, `password`, `button`, `image`, `file`, and `custom`; basically one for each HTML form `element` and a `custom` option.

The `editoptions` parameter is where you can further specify options to customize the element. First off, this is where you can place any attributes you want applied to the element itself; this includes standard HTML attributes, such as `class`, `placeholder`, and `value`, as well as any custom attributes you might want to add personally. For example, writing the following code:

```
colModel: [
  {
    name: 'name',
    label: 'Name',
    editable:true,
    editoptions:{
      placeholder:"Enter a Name",
      fake_attr:"some value",
      class:"custom-class"
    }
  },
```

We will create an input with the following attributes (among others):

```
<input type="text" placeholder="Enter a Name"
fake_attr="some value" class="custom-class">
```

So you really aren't restricted by anything in this regard. Besides for adding attributes to the HTML elements, the `editoptions` object also has special properties for certain types of inputs.

For checkboxes you can enter two values split by a colon to specify both the checked and unchecked values. Let's try this out with the verified column:

```
{
  name: 'verified',
  label: 'Verified',
  formatter: isVerified,
  editable:true,
  edittype:"checkbox",
  editoptions:{value: "Yes:No"}
},
```

This will create a checkbox where the checked value is `Yes` and the unchecked value is `No`.

The other time when `editoptions` will handle the `value` option differently is with a `select` HTML input. If you know HTML, then you will know that the `select` input encloses a set of the `option` elements, each having a value and some readable text.

To handle this with jqGrid, you have two options; one is to create a string where the values and texts are separated by colons, and the different options are separated by semicolons. The following example demonstrates this:

```
editoptions: {  
  value: "new_york:New York; montreal:Montreal;  
  alaska:Alaska"  
}
```

The other option is to make it an actual object in itself, like shown in the following code:

```
editoptions: {  
  value:{  
    new_york: "New York",  
    montreal: "Montreal",  
    alaska: "Alaska"  
  }  
}
```

Either way the results will be the same, and that is a select element that looks something like this:

```
<select>  
  <option value="new_york">New York</option>  
  <option value="montreal">Montreal</option>  
  <option value="alaska">Alaska</option>  
</select>
```

The last thing to mention is how the custom `input` type works. In a nutshell, this type can be used to create, really anything, as no preconceptions are made at all. Basically you define two functions: one for creating the actual HTML element for the form, and one for getting and setting the value. The first function accepts the cell's value as parameters and the options for the given `colModel` (basically the `editoptions` field). The second function accepts three parameters: the HTML element that you made in the previous function, the request type which can be `get` or `set`, because this function will be called for both, and the last parameter is the actual cell's value.

It's worth noting that the first time you click on `edit`, it will only call the first function to build the element, and on subsequent clicks it will only call the second function, as the form is not rebuilt every time. This means you need to set the value for the input in the first function as well, so that on the first click the values will show up. An example of using these is as follows; here is `colModel`:

```
colModel: [
  {
    name: 'middle_name',
    label: 'Middle Name',
    editable:true,
    edittype:"custom",
    editoptions:{
      custom_element: initialize_element,
      custom_value: access_value
    }
  }
]
```

And then here are the two functions:

```
function initialize_element (value, options) {
  return $("").attr("type",
    "text").val(value);
}

function access_value (elem, operation, value) {
  if (operation === "get") {
    return elem.val();
  } else {
    elem.val(value);
  }
}
```

The first function creates a new input using jQuery, sets it to a text box, and sets the initial value (as mentioned, the second function won't be called on the first run). The second function can then get or set the value based on the operation parameter; all the code is standard JavaScript and jQuery. Of course, if you were only creating a simple text field like in the previous example, it would be much easier to just use the text input type, but I hope it illustrates the idea.

The editrules option

We have now finished with setting the column to `editable` and we have configured the type and options for each of the elements; the next parameter you can set on the individual `colModels` is `editrules`. This property is used for validation, when the user clicks on **Submit** to add or

edit a row, based on this property, some validation could be made before the request goes through. A lot of the possible values are pretty self-explanatory, so I will just run through them. The following are all Booleans:

- `required`: Make sure this field is set
- `number`: Make sure the entered value is a number
- `integer`: Make sure the entered value is an integer number
- `email`: Make sure the entered value has an e-mail format
- `url`: Make sure the entered value has a URL format
- `time`: Make sure the entered value is in a time format
- `date`: Make sure the entered value is in a specific date format

Besides these, if you are checking a number-typed cell, you can use `minValue` and `maxValue` to make sure the value is in the correct range, and `custom` and `custom_func` to perform custom validation. To use a custom validation, just set `custom` to `True` and set `custom_func` to a function that accepts two parameters: the actual value and the column's name. The function should return an array where the first element is a Boolean, expressing whether the validation is passed or not, and the second element is the error message, if any.

All these properties can be combined, so you can use more than one form of validation. Following is a possible example for a column where the value needs to be between 20 and 40, but cannot be 32:

```
colModel: [  
  {  
    name: "special_data",  
    label: 'Special Data',  
    editable: true,  
    editrules: {  
      required: true,  
      integer: true,  
      minValue: 20,  
      maxValue: 40,  
      custom: true,  
      custom_func: not_thirty_two  
    }  
  }  
]
```

Then add the following function:

```
function not_thirty_two (value, name) {  
  var passes = (parseInt(value) !== 32);  
  return [passes, "You can't pick 32!"];  
}
```

This function starts by checking whether the value is `32` and saving it as a Boolean; we then return that variable along with the error message. It is true we are returning the error message even when the field passes, but in that case it will just be disregarded.

The `formoptions` option

The final `colModel` option that has to do with editing data is the `formoptions` field. These are strictly for modifying how the data shows up in the form, as the name suggests. There aren't that many properties, so we can go through all of them:

- `elmprefix`: This sets some text or HTML before the input
- `elmsuffix`: This sets some text or HTML after the input
- `label`: It adjusts the label which shows up in the form
- `rowpos`: This sets the row in the form where this element shows up
- `colpos`: This sets the column in the form where this element shows up

The first two are pretty simple, you can set some text or HTML and whatever you put there will be placed before or after the element respectively. Next, the `label` property lets you adjust the actual label that shows up; this is set to the columns title by default.

The `rowpos` and `colpos` options require a bit more of explanation. The modals for adding or editing a row are built using an HTML `table` element. By default the first `colModel` will be in the first row, the second will be in the second row, and so on. `rowpos` allows you to change the order around by setting its row number (starting from 1). `colpos` on the other hand allows you to create multiple columns of form elements. What I mean by this is you can have two inputs with a `rowpos` of one, but have them side-by-side by setting the first `colpos` to `one` and the second to `two`. This allows you to easily create pretty detailed forms and layouts, just by modifying these two numbers.

An example of using these options can be as follows:

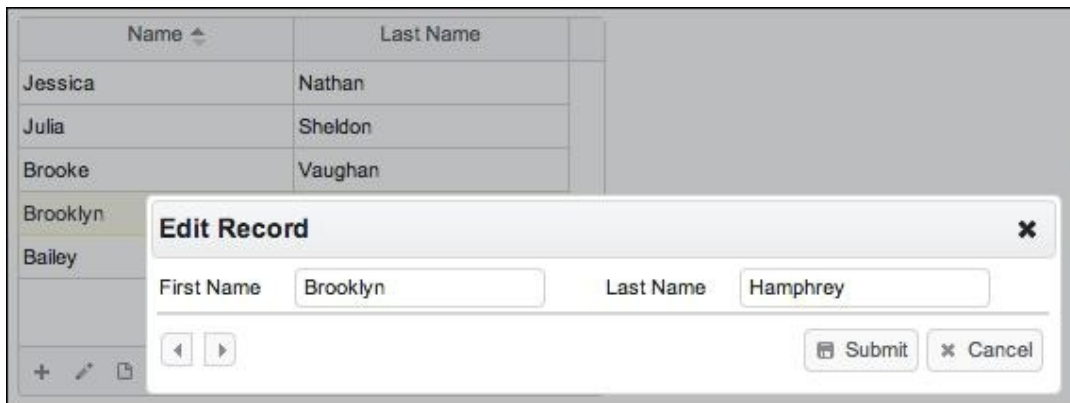
```
colModel:[
  {
    name: 'firstname',
    label: 'Name',
    editable:true,
    formoptions: {
      label: "First Name",
      rowpos: 1,
      colpos: 1
    }
  },
  {
```

```

        name: 'lastname',
        label: 'Last Name',
        editable: true,
        formoptions: {
            label: "Last Name",
            rowpos: 1,
            colpos: 2
        }
    },
]

```

Running this will produce a form that looks something like the following screenshot:



With all the previous information, let's construct the `colModel` for our main example:

```

colModel: [
    {
        name: 'name',
        label: 'Name',
        editable: true,
        editrules: { required: true }
    },
    {
        name: 'timestamp',
        label: 'Date',
        formatter: 'date',
        formatoptions: { srcformat: 'U',
            newformat: 'M jS Y' },
        editable: true,
        editrules: {
            required: true,
            custom: true,
            custom_func: checkDate
        }
    },
    {
        name: 'verified',
        label: 'Verified',

```

```

        formatter: isVerified,
        editable: true,
        edittype: "checkbox"
    },
    {
        name: 'website',
        label: 'Website',
        formatter: 'link',
        editable: true,
        editrules: {
            required: true,
            url: true
        }
    }
],

```

For `name` we just kept the default and set it to `required`; then for the `date`, we set it to `required` and added a custom validator to make sure the value can be processed into a timestamp. After those we have the `verified` field, which we just set to a checkbox, and lastly we have the `website` column, where we set it to be `required`, and add the URL validator.

Next, let's take a look at the function for verifying the date that can be parsed into a timestamp:

```

function checkDate(val, name) {
    var passes = !isNaN(Date.parse(val));
    return [passes, "That's not a valid date"];
}

```

We do this by attempting to parse the input, and then checking if `NaN` was returned. If it returns `NaN`, we know it couldn't and we therefore flip the Boolean and return the error message, otherwise we just return `True`.

Note

With this, we have finished with all the `colModel` options that have to do with editing data; to get more data as well as view more examples you can go to:

http://www.trirand.com/jqgridwiki/doku.php?id=wiki:common_rules

Configuring the modals

With the first aspect to editing data completed, namely configuring `colModel`, the next step is to configure the pop-up modals. These options can be configured inside the call to `navGrid`, the options for the edit modal is the third parameter, where as the options for the add modal is the fourth

parameter. Both forms take the same properties, so I will go through them together.

- `top`, `left`, `width`, and `height`: These four allow you to set the position (using `top` and `left`), as well as the size (using `width` and `height`). By default `height` is set to `auto`, so you don't really have to worry about it, but if you are using a multi-column form setup, then you may want to adjust the `width`, as this is set to only `300` by default.
- `url`, `mtype`, and `editData`: These adjust how and what is sent to the server; `url` sets the address of where to save data, `mtype` adjusts the method used to send the data (can be `GET` or `POST`), and `editData` allows you to attach extra data fields besides the data from the form itself.
- `topinfo` and `bottominfo`: You can use these two to add a string or HTML at the top and bottom of the form respectively. By default these are both set to empty strings.
- `addCaption`, `editCaption`, `bSubmit`, `bCancel`, and so on: By default all the labels and strings from the forms are taken from the language file, but they can all be overridden using the following keys:
 - `addCaption`: This is header for the add modal
 - `editCaption`: This is header for the edit modal
 - `bSubmit`: This is text for the **Submit** button
 - `bCancel`: This is text for the **Cancel** button
 - `bClose`: This is text for the **Close** button
 - `bYes`: This is text for the **Yes** button
 - `bNo`: This is text for the **No** button
 - `bExit`: This is text for the **Exit** button
 - `saveData`: This is text for the save confirmation message
- These can also be modified in the language file itself for global changes.
- `viewPagerButtons`: This option accepts a Boolean for whether or not to display the navigation arrows in the modal to go from one record to the next; this is on by default.
- `closeAfterAdd` and `closeAfterEdit`: These are Booleans that determine whether or not to close the modal on submit for both the add and edit modals. By default these are both set to `False`.
- `reloadAfterSubmit`: This determines whether or not the form should reload its data after you submit a row; by default this is set to `True`.

Note

These are the main options you will probably set, but you can get the full list at http://www.trirand.com/jqgridwiki/doku.php?id=wiki:form_editing.

It is worth noticing, that you don't need to set the URL here if that is the only parameter you will be tweaking. It is a lot easier to just set the `editurl` property in the `jqGrid` constructor itself, and that will achieve the same outcome.

Now I am not going to cover the other parameters for the view modal's options and the delete modal's options as they are almost identical to the add/edit modals, but you can find information about them at the previous link as well.

To recap so far, you now know how to set up a grid, configure the `colModels`, add a pager and navigator, and adjust everything to persist changes to the backend. The only problem now is we haven't built the backend itself. In the next section, we will be covering how to communicate with the jqGrid plugin, along with an example to show you how to set this up using PHP.

Communicating with jqGrid

Until now, we have had the data fed in directly using a local array of objects; chances are though in a real-world example, you will be loading these in from a backend.

So far all the specifications such as total and page counts have been calculated on the client side, but when you are dealing with a backend, you really want to transfer only one page at a time. If you have hundreds or thousands of rows, you don't want to transmit that on every page load. So what jqGrid does is, it makes a call to your backend and expects it to provide all the specifications instead; this means the backend defines what page you are on, as well as the total number of rows and pages.

The data you need to provide is as follows:

- Total number of records
- The current page number
- Total number of pages
- And the rows themselves

As you may have come to expect, jqGrid is really flexible on how you provide this information, as well as in which format. In this book I am going to focus on JSON, but jqGrid understands XML as well.

If you want to conform to the default format, then jqGrid expects the JSON data to follow a specific format. The following example demonstrates this:

```
{
  "total": 2,
  "page": 1,
  "records": 40,
  "rows": [
    { "id" : 1, "name": "John Smith" },
    { "id" : 2, "name": "Jane Doe" }
  ]
}
```

So, `total` is the total number of pages, `page` is the current page being returned, `records` is the total number of records on the backend (not the current number of records being returned), and `rows` is the actual rows.

These defaults are defined in the `reader` for each type; since we are dealing with JSON, the settings will be under a key named `jsonReader`. You usually don't need to edit anything, but let's say you want it to know that your page count is under a property called `page_count` instead of `total`, you can just modify the `total` property in the `jqGrid` call. Here is an example:

```
$("#grid").jqGrid({
  datatype: 'json',
  url: 'data.php',
  jsonReader: {
    total: "page_count",
    page: "current_page",
    records: "total_records",
    root: "results"
  }
});
```

Now, instead of the format before, we would need to send the data like so:

```
{
  "page_count": 2,
  "current_page": 1,
  "total_records": 40,
  "results": [
    { "id" : 1, "name": "John Smith" },
    { "id" : 2, "name": "Jane Doe" }
  ]
}
```

With the theory covered, let's take a look at an example in action; now jqGrid is going to make local AJAX requests to our backend, so you are

going to need a server setup. This can be a local PHP server such as MAMP or WAMP, and can even be a makeshift one created with PHP's built in, `S server` command; either way will work.

Once you have a server running, copy our jqGrid template into the server's directory, and let's add a basic grid:

```
<table id="grid"></table>
<div id="pager"></div>

<script>
    $("#grid").jqGrid({
        datatype: "json",
        url: "data.php",
        pager: "#pager",
        colModel: [
            { name: "name", label: 'Name' },
            { name: "phone_number", label: 'Phone
Number' },
            { name: "email", label: 'Email' }
        ]
    });
</script>
```

You should be fairly comfortable with this part by now; the only difference is instead of manually adding the data, we are specifying a `url` property for it from which we get the data.

I am going to be using `SQLite` in my example, but the queries will be pretty much the same for any SQL language, and the concepts have a one-to-one conversion to any database system.

So the first step is to get some data, and for this I am going to use a site called <http://www.generatedata.com>, you basically just set the column names and types (remember to match the column names to the `colModel's`), and then you select the format to output. The following is the screenshot of my settings:

Order	Table Column	Data Type	Examples	Options	Help	Del
1	name	Names	Alex Smith	Name Surname	?	
2	phone_number	Phone / Fax	Canada (2)	(Xxx) Xxx-xxxx	?	
3	email	Email	No examples available.	No options available.	?	

Add Row(s)

EXPORT TYPES

CSV Excel HTML JSON Programming Language **SQL** XML

Database table name: Statement Type: ☒ INSERT ☐ UPDATE

Database Type: Primary Key: ☒ None ☐ Add default auto-increment column

Misc Options: ☒ Include CREATE TABLE query ☐ Include DROP TABLE query ☐ Enclose table and field names with backquotes

Generate rows ☒ Generate in-page ☐ New window/tab ☐ Prompt to download **Generate**

Once done, click on **Generate**, and you will get the SQL queries needed to create a table named `users` and insert 100 rows inside, copy the results and paste it into a file named `seed.sql` inside your server's directory. Then you will need to seed the data into the database; there are many ways to do this and it depends on your platform and software, but on Mac you can just use the `sqlite3` command line program. To do this, just open up a terminal window and `cd` to your server's directory, from there enter the following:

```
sqlite3 -init seed.sql jqGrid.db .quit
```

Now, a couple of things are happening with this command, first of all it will create a new SQLite DB in the current folder named `jqGrid.db`. Next, it will read the seed file, and execute all the commands inside, once all that is done it will run `.quit` which will exit the SQLite prompt and bring you back to the terminal. If all went well, you should see a message that says something like:

-- Loading resources from seed.sql

The last step is to create the actual PHP file, following the name we specified as the `url` attribute, create a file named `data.php`. To begin with, we will need to connect to the database and prepare some SQL queries. The first query is going to retrieve the actual rows; this query has a placeholder for sorting the rows, as well as retrieving a specific subset (for example, 20 rows starting from number 40). This is needed because we don't want to pull all the data all the time, and so based on the `rowNum` property and the page number, we will only want to pull the needed data. Now, the next issue is that we need to tell jqGrid how many rows are there in total, but if we are only pulling a limited number of rows, we won't have this information. To get around this, we have a second query, which will

return the `count` or total number of rows in the table altogether. The following is the code for this part:

```
<?php

//Connect to the DB
$db = new PDO("sqlite:jqGrid.db");

//Prepare the query to get the results
$query = "SELECT ROWID as id, * FROM users "
        . "ORDER BY :column :direction "
        . "LIMIT :offset, :count";

//And the query to count the rows
$countQuery = "SELECT COUNT(*) as row_count
FROM users";
```

Pretty straight forward so far; now the next step is to prepare the data for these placeholders, and we can do this using the properties jqGrid supplies the script through `GET` variables.

The noticeable options given are `rows` for the number of rows, `page` for the current page number, and `sord/sidx` for sorting order (direction) and id (column).

Note

These parameter names are configurable with the `prmNames` jqGrid property found at [http://www.trirand.com/jqgridwiki/doku.php?id=wiki:options&s\[\]=prmNames](http://www.trirand.com/jqgridwiki/doku.php?id=wiki:options&s[]=prmNames).

Using this data we can calculate the placeholders shown in the following code:

```
//Calculate the placeholders
$rowsPerPage = $_GET['rows'];
$currentPage = $_GET['page'];
$offset = ($currentPage - 1) * $rowsPerPage;
$sortDirection = $_GET['sord'];
$sortColumn = $_GET['sidx'];

//Default the sort column to ROWID
if ($sortColumn === "") {
    $sortColumn = "ROWID";
}
```

The first two are taken straight from the `GET` variables; next we calculate the offset by multiplying the number of rows per page with the number of pages

that have passed (which is the current page minus one). After that we pull the sort direction and order, again, from the `GET` variables, and default the column to `ROWID` if it is blank. `ROWID` is the default column in SQLite that contains a row's ID, so if no column was selected in the grid to be sorted, we will just sort based on the ID. This is also a good place to configure your default sorting, so let's say your table has a date column you may want to default it to the date or something like that, but for here we will use the ID.

With the placeholders ready, we can make the requests:

```
$query = str_replace(
    array(
        ":column",
        ":direction",
        ":offset",
        ":count"
    ),
    array(
        $sortColumn,
        $sortDirection,
        $offset,
        $rowsPerPage
    ),
    $query
);
$res = $db->query($query);
$rows = $res->fetchAll(PDO::FETCH_CLASS);

$countRes = $db->query($countQuery);
$numberOfRows =
$countRes->fetchObject()->row_count;
```

The first command replaces the placeholders with their values and the next one executes the query, which we then fetch the results as objects, and store them inside a variable called `$rows`. Then for the second query we only have one row with a single column, the row count, so we just fetch the individual row and pull the column out into `$numberOfRows`.

We now have everything we need to create the response:

```
//Prepare the response
$numberOfPages = ceil( $numberOfRows /
$rowsPerPage );

$response = array(
    "total" => $numberOfPages,
    "records" => $numberOfRows,
    "page" => $currentPage,
    "rows" => $rows
);
```

```
echo json_encode($response);
```

We calculate the total number of pages by dividing the total number of rows by the rows per page, and round the number up, so for example, if we divide and get a number like 2.5 it means two full pages and one half page, but in terms of pages needed that's still three, so rounding up with ceil will do this for us.

Next, we prepare the data for jqGrid by creating an associative array, using the keys that jqGrid expects to see (from the `jsonReader` settings), and we use all the values we just finished processing. Last but not least, we echo out the json-encoded version of that array to jqGrid and with that we should have a fully functioning grid again, that now loads data from the server instead of a variable. You can try clicking on the columns to have them sorted, and if everything was done right, it should be working correctly.

Editing data server-side

With the data loading correctly, let's take a look at editing it again. Before, we were able to configure all the editing features, and even test some of them out, but up to this point, the actual saving didn't work, because we did not have a backend setup to accept the changes. In this section, we will take a look at creating a PHP script for this, which will allow you to add, edit, and delete rows, all from the frontend table.

Let's begin by setting the column's `editable` property in the `colModel` to `true`, and let's also add the navigator to our grid:

```
$("#grid").jqGrid({
    datatype: "json",
    url: "data.php",
    pager: "#pager",
    colModel: [
        { name: "name", label: 'Name',
    editable:true },
        { name: "phone_number", label: 'Phone
Number', editable:true },
        { name: "email", label: 'Email',
    editable:true }
    ],
    editurl: "editData.php"
});
$("#grid").jqGrid("navGrid", "#pager");
```

Note that I also added the `editurl` property, which is the URL for the script to which post changes, besides for that we are just using all the defaults.

The next step is to create the PHP file named `editData.php` in your web server's directory. Now, there are three separate events we need to handle:

adding a row, editing a row, and deleting a row; jqGrid will pass a `POST` variable named `oper` to let you know which event is being handled.

Let's start with the initial setup of connecting to the DB and preparing some queries:

```
<?php

//Connect to the DB
$db = new PDO("sqlite:jqGrid.db");

//Insert Statement
$insert = "INSERT INTO users (name, email,
phone_number) "
        . "VALUES
(:name, :email, :phone)";

//Update Statement
$update = "UPDATE users SET "
        . "name=:name, email=:email,
phone_number=:phone "
        . "WHERE ROWID = :id;";

//Delete Statement
/delete = "DELETE FROM users WHERE ROWID
= :id;";
```

The first line is the same; we are just connecting to the database, and then we create the queries with placeholders. Next, we need to check which operation the grid is trying to perform, and handle the appropriate method; let's start with the `add` method:

```
//Request Operation
$opp = $_POST['oper'];

if ($opp === "add") {
    $q = $db->prepare($insert);
    $q->execute(array(
        ":name" => $_POST['name'],
        ":email" => $_POST['email'],
        ":phone" => $_POST['phone_number']
    ));
}
```

We start by preparing the statement, and then we inject the values into the placeholders and run it using the `execute()` function. The other two are exactly the same, except with the other queries:

```
elseif ($opp === "edit") {
    $q = $db->prepare($update);
    $q->execute(array(
```

```

        ":name" => $_POST['name'],
        ":email" => $_POST['email'],
        ":phone" => $_POST['phone_number'],
        ":id" => $_POST['id']
    ));
} elseif ($opp === "del") {
    $q = $db->prepare($delete);
    $q->execute(array(
        ":id" => $_POST['id']
    ));
}
}

```

It's that simple; this code will handle all three options; you can now use all the buttons on the navigator to add/edit and delete rows, and your changes will persist. This coupled with all the advanced settings we saw for configuring these forms combine to provide a really elegant approach to managing data from the frontend.

Searching in jqGrid

We have now come full circle, all the way back to searching; this worked before, as we were using local data, so jqGrid used its internal search functions to handle it. We are now loading data from the backend, one page at a time, so jqGrid posts the search query to the server, and expects the backend to reply with the rows.

It uses the same `data.php` page we set up earlier, it just adds custom search parameters as `POST` variables that you need to take into account when returning data.

For the basic search, you have four main properties:

- `_search`: This is a Boolean telling you whether this is a search or not
- `searchField`: The column being searched
- `searchOper`: This is the search type
- `searchString`: This is the actual search query

These are pretty simple, except for `searchOper`, which needs more explanation. The search operations are as follows:

- `eq`: This means the column is equal to
- `ne`: This means the column is not equal to
- `lt`: This means the column is less than
- `le`: This means the column is less than or equals
- `gt`: This means the column is greater than
- `ge`: This means the column is greater than or equals
- `bw`: This means the column begins with
- `bn`: This means the column does not begin with
- `in`: This means the column is in an array of values

- `ni`: This means the column is not in an array of values
- `ew`: This means the column ends with
- `en`: This means the column does not end with
- `cn`: This means the column contains
- `nc`: This means the column does not contain
- `nu`: This means the column is not null
- `nn`: This means the column is null

To begin implementing these, we need to modify the existing code inside `data.php` to add a `where` clause. So, change the `$query` variable to the following:

```
//Prepare the query to get the results
$query = "SELECT ROWID as id, * FROM users "
        . ":where "
        . "ORDER BY :column :direction "
        . "LIMIT :offset, :count";
```

And then change the code where we replace the placeholders, as shown in the following code:

```
//Execute the queries
$query = str_replace(
    array(
        ":column",
        ":direction",
        ":offset",
        ":count",
        ":where"
    ),
    array(
        $sortColumn,
        $sortDirection,
        $offset,
        $rowsPerPage,
        $where
    ),
    $query
);
```

The last step is to calculate the `$where` variable, based on the request variables, so if there isn't a search being performed, then it should just be a blank string; otherwise we need to read the type of search and handle it appropriately. Add the following code right before the section where it replaces the placeholders:

```
//Process Search
$where = "";
if ($_GET['_search'] && $_GET['_search'] ===
"true") {
```

```

$searchCol = $_GET['searchField'];
$searchQuery = $_GET['searchString'];
$searchOp = $_GET['searchOper'];
$where = "WHERE ";
switch ($searchOp) {
    /* Handle methods here */
}
}

```

So we begin by setting the default of a blank string for the event there is no search going on, then we have an `if` statement to change this default if we are searching. The first thing we do inside is pull the request variables and start the `where` clause with the word `WHERE`, as they all have this in common. The next thing we need to do is run the search operation through a `switch` statement and handle all the methods above. I am going to write a few at a time so I can explain them better, but they all go inside the `switch` statement. It is also worth noting that all the different search operations are based on SQL commands, so what we are essentially doing is mapping the SQL functions to the jqGrid search parameters. Here is the first couple:

```

case "eq":
    $v = $db->quote($searchQuery);
    $where .= $searchCol . "=" . $v;
    break;
case "ne":
    $v = $db->quote($searchQuery);
    $where .= $searchCol . "!=" . $v;
    break;
case "lt":
    $v = $db->quote($searchQuery);
    $where .= $searchCol . "<" . $v;
    break;
case "le":
    $v = $db->quote($searchQuery);
    $where .= $searchCol . "<=" . $v;
    break;
case "gt":
    $v = $db->quote($searchQuery);
    $where .= $searchCol . ">" . $v;
    break;
case "ge":
    $v = $db->quote($searchQuery);
    $where .= $searchCol . ">=" . $v;
    break;
case "bw":
    $v = $db->quote($searchQuery .
"%");
    $where .= $searchCol . " LIKE " .
$v;
    break;
case "bn":
    $v = $db->quote($searchQuery .
"%");

```



```

                                $where .= $searchCol . " NOT LIKE
" . $v;
                                break;

```

We begin by escaping the search string; unlike with [editData.php](#), where we used the prepared statements feature to automatically escape the values, here we are essentially just concatenating the different components together. Thus, it's necessary to escape them so that they don't mess up the regular quotes that are required. For example, if you were searching for a word such as [that's](#), the apostrophe would close the value's wrapping quotes and would therefore invalidate the query; by escaping the values we can be sure this won't happen.

Other than that, each of these has a different operation, and we apply the corresponding SQL symbol or function to handle it. The next two cases ([in list](#) and [not in list](#)), require a bit more code to handle them; if you remember, we don't get a list, we get a string in the request. This is entirely up to you how to convert the string into an array, but I decided for this example we will just use a comma and space to split the values. So if someone searches for [Bob, John](#) using the [in list](#) operation, it will return anyone whose name is either Bob or John. These two cases are shown in the following code:

```

                                case "in":
                                    $pieces = explode(", ",
$searchQuery);
                                    $v = "(";
                                    $pref = "";
                                    foreach ($pieces as $segment) {
                                        $v .= $pref .
$db->quote($segment);
                                        $pref = ", ";
                                    }
                                    $v .= ")";
                                    $where .= $searchCol . " IN " .
$v;
                                    break;
                                case "ni":
                                    $pieces = explode(", ",
$searchQuery);
                                    $v = "(";
                                    $pref = "";
                                    foreach ($pieces as $segment) {
                                        $v .= $pref .
$db->quote($segment);
                                        $pref = ", ";
                                    }
                                    $v .= ")";
                                    $where .= $searchCol . " NOT IN
" . $v;
                                    break;

```

They are essentially the same, with the only difference being the `NOT` in the query itself. We begin by splitting the string by comma and space, and we then combine the values together, escaping each one, and surrounding the entire list with brackets as per the SQL syntax.

The rest of the cases are pretty straight forward if you know SQL:

```
        case "ew":
            $v = $db->quote("%" .
$searchQuery);
            $where .= $searchCol . " LIKE " .
$v;
            break;
        case "en":
            $v = $db->quote("%" .
$searchQuery);
            $where .= $searchCol . " NOT LIKE
" . $v;
            break;
        case "cn":
            $v = $db->quote("%" .
$searchQuery . "%");
            $where .= $searchCol . " LIKE " .
$v;
            break;
        case "nc":
            $v = $db->quote("%" .
$searchQuery . "%");
            $where .= $searchCol . " NOT LIKE
" . $v;
            break;
        case "nu":
            $where .= $searchCol . " IS NULL";
            break;
        case "nn":
            $where .= $searchCol . " IS NOT
NULL";
            break;
```

And that is all; with this final code, the search should be fully operational, and you should have a pretty good idea of how each operation can be implemented.

With the navigator fully working, and a good background on all the common settings, let's finish with a look at the API side of things.

Interfacing the API

Up until now, we have been using the grid in the manner it was built; we have been using all the default functionality, such as the pager/navigator,

and even the forms to edit the data. In this section we will take a look at both how to call methods with jqGrid as well as how to listen to some of the events that jqGrid has to offer.

Calling methods

We already have called one jqGrid method, the `navGrid` method, so you should be pretty comfortable with the syntax, but to recap, you basically call the `jqGrid` function on an element, and you pass the name of the method as the first parameter, and all the method's parameters as subsequent parameters. The following screenshot illustrates what I am talking about:

```
$("#grid").jqGrid("methodName", param1, param2, ...);
```

Now, we can't really go through every method jqGrid has to offer; there are a lot of them. But we will try and go through some of the most useful ones, and you can view a more complete list at

<http://www.trirand.com/jqgridwiki/doku.php?id=wiki:methods>.

- `setGridWidth` / `setGridHeight`: These two methods allow you to programmatically adjust the height and width of the grid. The first method `setGridWidth` accepts two parameters, the first is the new width, and the second is whether the columns should shrink to fit, which means whether the individual column's width should be recalculated to fit the new width. Next, we have `setGridHeight`; this method has only one parameter, which is the new height for the grid; the size can be a pixel size, a percentage or set to `auto` to fit all rows. Following is an example of this:

```
$("#grid").jqGrid("setGridWidth", 400, true);  
$("#grid").jqGrid("setGridHeight", "auto");
```

- `showCol` / `hideCol`: These two functions allow you to show and hide a column from the grid. These both accept the name of a column as a parameter, and can be used as follows:

```
$("#grid").jqGrid("hideCol", "email");  
$("#grid").jqGrid("showCol", "email");
```

- `sortGrid`: This method accepts the name of a column, and will sort the table accordingly, an optional second parameter is a Boolean of whether or not the grid should also reload the current pages data with the sort information. Following is an example of this:

```
$("#grid").jqGrid("sortGrid", "email");
```

- `editGridRow`: This will open the modal for editing/adding a row as if you had clicked on the buttons on the navigation layer. The first parameter is either a `rowid` if you want to edit an existing row, or the word `new` to open the add dialog box. The second parameter is a

JavaScript object of options that we covered before when we dealt with the navigation bar:

```
$("#grid").jqGrid("editGridRow", 1,
{closeAfterEdit: true});
```

Events

Now, the last thing we would like to cover is the notion of events with jqGrid. Following the style of open-endedness we have come to expect from jqGrid, basically any method or process the grid is going to perform, you will most likely have an attached event, where you can run a function before or after (and sometimes both).

Like with the methods, we couldn't possibly go through all of them, but you can view a lot of them at <http://www.trirand.com/jqgridwiki/doku.php?id=wiki:events>, as well as throughout the rest of the documentation.

Events are set up a bit differently than methods, as you don't call them, you set them as properties in the jqGrid initializer. Here are some events that are particularly helpful:

- `gridComplete`: This event gets called once the grid is ready, which means it has finished loading and rendering the columns and data. This event gets triggered every time you change the page or sort the data.
- `onSelectRow`: This event gets triggered whenever a row is selected, and sends three parameters on each call: the selected rows `rowid`, the status of the current row, and the actual `click` event.
- `ondblClickRow`: This event is similar to the previous one, except it is called when a row is double-clicked, and receives four parameters. The first is `rowid` like before, the second is the row's index on the current page, so if we were on the second page, `rowid` for the first row could be something like `21`, but the row's index would still be `1`, because on this page it is the first. The third parameter is the index of the column that was clicked, starting at zero from left to right, and the last parameter is the actual `double-click` event.

Using all the previous information, let's construct one last example. We will be using the same data, but let's add a header element to the page, so change the HTML to reflect this:

```
<h1 id="nameTitle"></h1>
<table id="grid"></table>
```

Next, let's add some events to the `jqGrid` call:

```

$("#grid").jqGrid({
    datatype: "json",
    url: "data.php",
    pager: "#pager",
    colModel: [
        { name: "name", label: 'Name',
    editable:true },
        { name: "phone_number", label: 'Phone
Number', editable:true },
        { name: "email", label: 'Email',
    editable:true }
    ],
    editurl: "editData.php",
    onSelectRow: function(rid){
        var data = $("#grid").jqGrid("getRowData",
rid);
        $("#nameTitle").text(data.name);
    },
    ondblClickRow: function(rid){
        $("#grid").jqGrid("editGridRow", rid, {
            afterComplete: function(resp, data){
                $("#nameTitle").text(data.name);
            }
        });
    }
});

```

Now, when you select a row, it will update the header with that row's name; this is a very simple example of intertwining the grid with the rest of your page, allowing you to show a small amount of information in the grid, and then open up to a full document when selected, like you have with an e-mail inbox. Additionally, we have added more intuitive controls by allowing users to just double-click on a row to edit it, all without needing to compromise on features as we can keep the DOM up-to-date using events.

I hope this sparks some ideas of cool ways in which you can mix both events and methods to extend the already powerful library, that is, jqGrid.

People and places you should get to know

In this book, we have only covered the basics needed to get a grid up and running, with only a small glimpse of where it can go. Of course, this is only the beginning, and staying up-to-date on all the new updates is a key component to web development in general.

Here are some people and sites that had a lot of influence on the jqGrid project making it what it is today, and more importantly, providing information and ideas to extend your jqGrid knowledge further.

Official sites

The official jqGrid docs has been referenced throughout the book, and is a key resource for getting a general overview of all the features, and is a great first place to go when you have a problem:

<http://www.trirand.com/jqgridwiki/doku.php>

The jqGrid demo page is a collection showcasing over hundred different examples of features including the code. If you are unsure what a feature does and would like to see it in action, this is a great resource for playing with a live example and viewing the code that makes it work:

<http://trirand.com/blog/jqgrid/jqgrid.html>

To view the source in development, track bugs as well as contribute fixes. New features you can check out the *GitHub* page, where it is actively being worked on: <https://github.com/tonytomov/jqGrid>

If you are looking for a commercial version of jqGrid, which comes with a backend solution to fit your app, then the solution to that is

<http://www.trirand.net/>.

And of course, this list could not be complete without the blog and homepage: <http://www.trirand.com/blog/>

Articles and tutorials

There's a great article/video by Elijah Manor on integrating jqGrid with ASP.NET, which is definitely worth checking out if you are using ASP:

<http://www.elijahmanor.com/2010/07/jquery-jqgrid-plugin-add-edit-delete.html>

Community

For the unfamiliar, stack overflow is a question and answer site where people can post real problems and get practical answers. Chances are if you are having an issue, someone else has solved it, and if not you can always post your own questions. Here is the stack overflow page with the jqGrid query: <http://stackoverflow.com/search?q=jqgrid>

Both in terms of contributions, and support on sites such as www.stackoverflow.com, Oleg has helped move the project along and is a great resource person to follow for such updates on <https://github.com/OlegKi> <http://stackoverflow.com/users/315935/oleg>.

Twitter

You can follow Tony and the jqGrid project on twitter, at the username [@jqGrid](https://twitter.com/jqGrid) to stay up-to-date with releases: <https://twitter.com/jqGrid>