Covers
jQuery 1.11.1 or 2.1
jQuery UI 1.11

THE ABSOLUTELY AWESOME

jQuery COOKBOOK

A collection of 70 jQuery recipes & 50 sub-recipes

SUPROTIM AGARWAL

The Absolutely Awesome

jQuery CookBook

(covers jQuery v1.11 or v2.1 & jQueryUI v1.11)

By Suprotim Agarwal

About the Author

Suprotim Agarwal, is an ASP.NET Architecture MVP (Microsoft Most Valuable Professional) and has been developing web applications for over 15 years using Microsoft and JavaScript technologies. Suprotim is also an author and founder of popular .NET websites like DotNetCurry and DevCurry and the Editor-in-Chief of the DNC .NET Magazine. His first book '51 Recipes with jQuery and ASP.NET Controls' was accepted very well by the developer community and he has been since then yearning to author new books.

When he is not churning out code, he spends his time with his family, checks out new gadgets, plays games on his Xbox One and teaches programming to kids.

About the Reviewer

Irvin Dominin works as Technical lead in SISTEMI S.p.A.(Turin, Italy) for .NET and Windows Projects. He is an active member on StackOverflow and spends most of his time answering jQuery/JavaScript questions. You can reach him at: irvin.dominin@gmail.com or on LinkedIn

Table Of Contents

Section I - Some Concepts

Section II - Input Controls

Section III - Tables, Panels and Tabs

Section IV - Unordered List and DropDown Control

Section V - Menus and TreeView

Section VI - Working with Images

Section VII - Ajax

Section VIII - Creating jQuery Plugins

Section IX - Some Generic Recipes

Recipe 6

Exploring Mustache.js for Templating

One of the central tenets of modern web development is the separation of structure/content from behavior, a.k.a separate view from code. In our projects, we either tend to craft our HTML elements using string processing and concatenation; or build elements using native DOM methods, and then insert these elements into the DOM. However as the size and complexity of a project grows, it leads to spaghetti code. Maintainability becomes an issue in such cases.

Templates can be beneficial here as templates simplify the entire process, increase reusability and minimize

the amount of code needed to generate HTML elements from data.

Mustache.js (http://mustache.github.io/) is a logic-less templating library, which essentially supports the philosophy of little or no logic in your HTML templates. That means in your templates, there are no loops or if-else statements; instead there are only tags. There are several other logic-less templates out there like dust.js, handlebars.js etc. which provide more functionality. We even have jQuery templating libraries like jQuery Templates and jsViews, which have been deprecated and superseded by the jsRender project. However we are choosing Mustache.js for our example as I find it very simple to use, and moreover we do not require too much complexity in this example. Having said that, feel free to look into the other templating engines too and decide what's best for your project.

The premise of Mustache.js is quite simple. Think of it as a blueprint of HTML markup which lets you flow data on a page. You can use as many templates as you want on a page, and take the same data and display it in different ways.

To get started, download the Mustache library from Github https://github.com/janl/mustache.js or use a

CDN from cdnjs.com.

In this example, we will read JSON data containing author information and display it on a page using Mustache template. The .json file is kept in the 'scripts' folder within 'S1-Concepts' folder. The structure of the JSON data is as follows:

```
{ "authors" : [
{
    "name": "Suprotim Agarwal",
    "bio": "Suprotim is the founder of DotNetCurry.com, DNC .NET Magazine, SqlServerC
    "image": "suprotim.jpg",
    "twitter": "@suprotimagarwal"
},
{
    "name": "Alex Can",
    "bio": "Alex works as an author, freelance trainer and consultant on various clie
    "image": "alex.jpg",
    "twitter": "@twitter"
},
{
    "name": "Lynda Wasden",
    "bio": "Lynda has over 15 years of experience in IT education and development. SH
    "image": "lynda.jpg",
    "twitter": "@twitter"
}
]
}
```

Make sure your JSON is valid. Use a JSON validator like jsonlint.com to validate your JSON.

There are various methods for defining mustache templates in your application, like using it inline or using it as external templates. For our example, we will create it inline but for better maintainability, you should make it a practice of keeping it in a separate file for your applications. So if you decide to keep the

template in a separate file, all you have to do is create a <templatename>.js file and then add a reference to this file using the `<script>` tag.

```
<script src="../scripts/templatename.js">
</script>
```

Coming back to our example, I have created a new page called '6-Mustache.html' in the 'S1-Concepts' folder. Our template is called *authortmpl* and we will define template data inside of a <script> tag with a MIME type of *text/template*.

```
<script id="authortmpl" type="text/template">
</script>
```

We are using a MIME type other than text/javascript to prevent the browser from executing the template code.

First off, we start by creating a placeholder for authors. This is done using brace characters ({}) which looks like sideway mustaches (hence the name mustache.js).

```
<script id="authortmpl" type="text/template">
    {{#authors}}
    {{/authors}}
</script>
```

Observe how we are closing the *authors* placeholder using a backslash (/). This is similar to what we do with HTML elements. If you observe, *authors* is the name of the object in our JSON data file authorslist.json. Mustache.js cycles through all the authors in that object and applies the template that is defined here.

```
<script id="authortmpl" type="text/template">
  {{#authors}}
  <div class="divwrap">
    <h2>{{name}}</h2>
    <img src="images/{{image}}"
    alt="{{name}}" class="imgwrap"/>
    <p class="pwrap">{{bio}}</p>
    <p>Twitter: {{twitter}}</p>
  </div>
  {{/authors}}
</script>
```

In the template, we have a `<div>` with a class `divwrap` and the author name, image, bio and Twitter handler defined inside the `<div>`. Image and the paragraph have been decorated with the `imgwrap` and `pwrap` classes respectively. This is to beautify the template with CSS. The css file 'templ.css' is kept in the 'css' folder in the 'S1-Concepts' folder. We have added a reference to this css file inside the <head> section of

the HTML file.

```
<link href="css/templ.css" rel="stylesheet"
/>
```

If you observe, the tags defined inside the double braces match the elements in our .json file. Eg: {{name}} acts as placeholder which targets the *name* element inside the .json file and so on. Overall this template is nothing more than some html and placeholders. As you can see, it is a logic-less template. There are no programming blocks, if-else statements or any loops.

The last step is to use jQuery to grab our authorslist.json file and insert the data into our template. We will use jQuery's `$.getJSON()` method to load JSON-encoded data from the server using a GET HTTP request.

We briefly discussed JSON in the Getting Started with Ajax chapter in Recipe 5. Using $.getJSON() has been explained in Recipe 53.

```
$.getJSON('scripts/authorslist.json',
function (data) {
  var templ = $('#authortmpl').html();
  var mustch = Mustache.to_html(templ, data);
  $("#authorlist").html(mustch);
});
```

Here the `$.getJSON()` function loads the data file authorlist.json and executes a function literal to read the contents of the JSON file, into a `data` object.

A variable called `templ` loads the content of the `authortmpl` template that we created a short while ago. We then use the `Mustache.to_html()` method and pass the template and JSON data as parameters.

Moustache will process the data and feed it into the template and create some HTML for us that we load in an `authorlist` div.

Save the HTML file and view it in a browser. Here's the output:

Suprotim Agarwal

Suprotim is the founder of DotNetCurry.com, DNC .NET Magazine, SqlServerCurry.com and DevCurry.com. He also authors articles and books on various client and server side technologies. Check out his latest book at www.jquerycookbook.com

Twitter: @suprotimagarwal

Alex Can

Alex works as an author, freelance trainer and consultant on various client side technologies from the past 14 years.

Twitter: @twitter

Recipe 12

Automatically add Commas to a Number

Different cultures have different ways of representing Dates, Numbers, Currencies, and Measurements etc. So Christmas day in India is represented as 25/12/2014 (dd/MM/YYYY) whereas the same is represented as 12/25/2014 (M/d/yyyy) in North America and some other parts of the world.

Similarly different parts of the world represent Numbers using different symbols. In the US, commas and decimals are used to represent a number like 10000.20 as 10,000.20. However the same number in

Europe is represented as 10.000,20.

JavaScript is not a culture-aware programming language. Although JavaScript understands the English culture's decimal formatting, however it does not know *how* to use a culture's number separator. Users have worked around this limitation by programmatically determining culture information from the browser's *navigator* object, and representing dates and numbers accordingly. However the approach is not very consistent due to browser inconsistencies.

There are various scripts that can be found on the internet that allows you to automatically add commas to a number. Create a HTML file called '12-NumberFormatting.html' in the 'S2-InputControls' folder and use the following markup:

```
<input id="num" type="text" />
<input type="button" id="btnformat"
value="Format Number" />
```

We have added a textbox and a button control to the page. Now add the following script (by a programmer Elias Z) that adds a comma to a number after every three digits:

```
$(function () {
  $('#btnformat').on('click', function () {
    var x = $('#num').val();
      $('#num').val(addCommas(x));
  });
});

function addCommas(x) {
  var parts = x.toString().split(".");
  parts[0] = parts[0].replace(/\B(?=(\
  d{3})+(?!\d))/g, ",");
  return parts.join(".");
}
```

The script uses a regular expression to add commas. Passing a number like 23456789.12345 produces 23,456,789.12345, which is the desired result.

Live Demo: http://www.jquerycookbook.com/demos/S2-InputControls/12–NumberFormatting.html

The script we just saw works fine, but what if at a later date you want to represent the same number for different countries?

Although JavaScript does not understand a culture's

number separator, it can certainly format numbers based on a certain culture. It does so by using the `Number.prototype.toLocaleString()` method. This method allows you to convert a number to a string that is formatted with local numeric formatting settings.

```
var num = 23456789.12345;
num = num.toLocaleString();
console.log(num);
```

The following code produces the output 23,456,789.123.

Using Globalize.js to format Numbers

Although the approaches shown here work, using a well-tested library function is the right approach to handle such scenarios. When your application changes in the future to accommodate international users, handling internationalization issues becomes a nightmare *without* a library function. This is where the Globalize.js library comes handy!

Globalize.js (https://github.com/jquery/globalize) is an open source JavaScript library for internationalization and localization and supports over 350 different cultures. It was created originally by Microsoft and

contributed to the jQuery library with the name jQuery globalization plug-in. Later it was made a standalone library.

It's worth noting that Globalization is a native feature in JavaScript with EcmaScript 5.1 and above. Read http://www.ecma-international.org/ecma-402/1.0/#sec-8 *to learn more about the ECMAScript Internationalization API Specification.*

Create a HTML file called '12.1-NumberFormatting.html'. Download the Globalize files from https://github.com/jquery/globalize . Look for the *Download ZIP* button on the right hand side of the page and click it to download a .zip archive. Extract the files from the archive and copy the lib/globalize.js and lib/cultures/globalize.cultures.js in the *scripts* folder. *globalize.js* deals with localization whereas *globalize.cultures.js* contains a complete set of the locales (around 352 different cultures are supported as of this writing).

The globalize.culture.js file is around 850 KB. If you are dealing with only a selected few regions, it is advisable to use only those culture files found in the lib/cultures folder. For eg: if you are planning to target only India and US locales, then you can directly reference the globalize.culture.en-IN.js and globalize.culture.en-US.js files in your application to save some bandwidth.

Now reference these two files in your application as shown here:

```
<script src="../scripts/globalize.js"></
script>
<script charset="utf-8" src="../scripts/
globalize.cultures.js"></script>
```

Add the following HTML which contains some Labels, a DropDown and a TextBox to show the formatted number.

```
<body>
  <p id="para"></p>
  <label for="ddlculture">Select Region
  </label>
  <select id="ddlculture">
    <option></option>
    <option value="zh-TW">China</option>
    <option value="fr-FR">France</option>
    <option value="de-DE">Germany</option>
    <option value="en-IN">India</option>
    <option value="ja-JP">Japan</option>
    <option value="ru-RU">Russia</option>
    <option value="es-ES">Spain</option>
    <option value="en-GB">United Kingdom
    </option>
```

```
    <option value="en-US">United States
    </option>
  </select>
  <label for="txtNum">Formatted Number:
  </label>
  <input type="text" id="txtNum"
  readonly="true" />
</body>
```

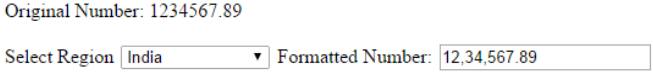
Now use this simple piece of code that uses the Globalize.js library to format a number depending on the country the user has selected from the dropdown.

```
$(function () {
  var number = 1234567.89;
  $("#para").html("Original Number: " +
  number);

  $('#ddlculture').on('change', function () {
    var culture = $(this).val();
    var formattedNumber =
formatNumber(number,          culture);
    $("#txtNum").val(formattedNumber);
  });

  function formatNumber(num, currentculture)
{
    Globalize.culture(currentculture);
```

```
        if (isNaN(num))
            return('Number not valid');
        return (Globalize.format(num, "n2"));
    }
});
```

In the code above, we are tracking the change event of the dropdown control and capturing the culture of the selected dropdown list item. This is possible as the value attribute of each dropdown list item has the culture defined; for eg: `<option value="en-IN">India</option>`. The original number to be formatted and the culture, is passed to the `formatNumber()` function which uses the `Globalize.format()` function to return the formatted number. It's that simple!

Run the code and you will get the formatted number based on the option selected.



Similarly Globalize.js also formats a date using the given format and locale. I will recommend you to read the documentation at https://github.com/jquery/globalize#date *for some handy examples.*

Live Demo: http://www.jquerycookbook.com/demos/S2-InputControls/12.1-NumberFormatting.html

Recipe 20

Add/Delete Rows in a Table

In modern real-life applications, tables are rarely static in nature. When it comes to manipulating tables, the end user expects to add new rows or delete existing ones *on-the-fly*.

This recipe demonstrates how to add new rows and delete existing ones. We have briefly touched upon adding new rows in Recipe 18.3. In this recipe, we will see some additional scenarios.

Please note that these code snippets are suitable for client-side processing only. If you have a database and you need to update data, you will need server-side

processing and use a technology like ASP.NET or PHP to add data to the data source through an Ajax call. Although server-side is beyond the scope of this book, I have mentioned some helpful links in the *Further Reading* section that demonstrates how to add new records in a table using jQuery and ASP.NET Web API.

20.1 - Insert a New row as the Last row of the Table

Create a HTML file called '20.1- AddRemoveRows.html' in the 'S3-TablesTabsPanels' folder and add the following markup:

```
<table id=”someTable”>
<thead>
  <tr>
    <th class=”empid”>EmpId</th>
    <th class=”fname”>First Name</th>
    <th class=”lname”>Last Name</th>
    <th class=”email”>Email</th>
    <th class=”age”>Age</th>
  </tr>
</thead>
<tbody>
  <tr>
    <td class=”empid”>E342</td>
    <td class=”fname”>Bill</td>
    <td class=”lname”>Evans</td>
```

```
    <td class="email">Bill@devcurry.com</td>
    <td class="age">35</td>
  </tr>
  ...
</tbody>
</table>
```

Write the following code to add new rows:

```
$(function () {
  newRow = "<tr>" +
    "<td class='empid'>E333</td>" +
    "<td class='fname'>Fujita</td>" +
    "<td class='lname'>Makoto</td>" +
    "<td class='email'>fujita@devcurry.com</
    td>" +
    "<td class='age'>52</td>" +
  "</tr>";
  $('#someTable > tbody > tr:last').
  after(newRow);
});
```

We are using the jQuery selector extension `:last` to select the last matched row in our table and using `after()` to insert the new row.

The above example works great for smaller tables, however in a large table, using `:last` selector may

not give you the best performance. As per the jQuery documentation, "*:last is a jQuery extension and not part of the CSS specs and hence cannot take advantage of the powerful native DOM methods that can parse any CSS selector, like querySelectorAll* ". To achieve better performance, we can rewrite our code as:

```
$('#someTable > tbody > tr').filter(":last").
after(newRow);
```

The code first selects the rows using a pure css selector `#someTable > tbody > tr` and then uses `filter(":last")` to match the last row of the table.

20.2 - Insert a New row in a Table at a Certain Index

Using the same markup of our previous recipe (Recipe 20.1), let's say you want to insert a new row as the 2nd row in a table. Create a new file called '20.2-AddRowsAtIndex.html' in the 'S3-TablesTabsPanels' folder and use the following code:

```
var index = 2;
newRow = "<tr>" +
"<td class='empid'>E333</td>" +
"<td class='fname'>Fujita</td>" +
"<td class='lname'>Makoto</td>" +
```

```
"<td class='email'>fujita@devcurry.com</td>"
+ "<td class='age'>52</td>" + "</tr>";
$('#someTable > tbody > tr').eq(index-1).
before(newRow);
```

EmpId	First Name	Last Name	Email	Age
E342	Bill	Evans	Bill@devcurry.com	35
E333	Fujita	Makoto	fujita@devcurry.com	52
E343	Laura	Matt	laura@devcurry.com	26
E344	Ram	Kumar	ram@devcurry.com	56
E345	Yuri	Gagrin	yuri@devcurry.com	39
E340	Asha	Singh	asha@devcurry.com	42

Since indexes are 0 based and we are passing index=2, `eq(index)` would mean `eq(2)` i.e. the 3rd row. So to add this to the 2nd row of a table, you first need to do index-1 and then you need to go back to the 2nd row of the table and insert `before()` that, so that this new row now becomes the 2nd row of the table.

Alternatively let's say you want to insert a row as the 2nd row in a table, you can do the following:

```
$('#someTable > tbody > tr:first').
after(newRow);
```

which uses the `:first` selector to match the first row and insert a row `after()` it.

Here again just like we saw earlier for the last selector, for large tables, you will get performance benefits by using filter(":first").

Similarly you can also explore other child filter selectors like first-child, nth-child and so on from the jQuery documentation at api.jquery.com/category/selectors/child-filter-selectors/

To become a better developer, I cannot emphasize the fact enough that you should take out some time and go through the jQuery documentation. It's probably one of the most well written documentation of any JavaScript library out there and getting familiar with different selectors and API's, will save you tons of time and less frustration in a project.

20.3 - Remove all Rows Except Header

If your table is well defined and contains a `<thead>` `<tbody>`, this piece of code will work to remove all the rows, except the header row:

```
$('#someTable tbody tr').remove();
```

This code uses the `remove()` method to remove a set of rows inside `tbody`. Since we haven't supplied the `remove()` method with any selector as a parameter, the above code removes all rows, as well as all elements, events and data in the rows.

If you want to remove all rows without removing data and events, use detach() instead of remove().

In case you do not have a `<thead>` defined and want to remove all rows except the first one (assuming first row is meant to be a header), use this code:

```
$('#someTable tr:gt(0)').remove();
```

..where the selector `gt(0)` selects all rows at an index greater than zero, within the matched rows. Similarly if you want to keep the first two rows and remove all others, change the above code to this:

```
$('#someTable tr:gt(1)').remove();
```

For better performance in modern browsers, use '$("your-pure-css-selector").slice(index)' instead. As seen and discussed earlier, in our case, a pure css selector would be '$('#someTable tr')'. All you need to do is use it with slice() to remove all except the first row.

```
$('#someTable tr').slice(1).remove();
```

`slice()` is zero based and takes two arguments, start and end. Since we are supplying *1* as the first parameter to slice(), the above statement will return a new jQuery object that selects from the first row, to the end. Calling `remove()` removes this set of matched element returned by slice and you are left with only the first row.

As you must have observed with all these examples, there are multiple ways in jQuery to achieve a certain requirement. A point to always remember is that jQuery will always use a native method (in our case we discussed *querySelectorAll*) if available, as it's much quicker at getting elements and gives a notable performance with large sets of data. *jsperf.com* is your friend to run tests when you are in doubt as to which selectors or methods to use in your code for the browsers you are supporting. Refer to Recipe 7 if you haven't already, for a quick tutorial on using jsperf.

20.4 - Remove a Row by Clicking on it

To remove a row when it is clicked, use `remove()`

```
$('#someTable tbody tr').on("click", function
() {
  $(this).remove();
  return false;
});
```

`this` here represents the clicked row. `$(this)` converts it into a jQuery object on which the `remove()` method is called to remove the row.

Live Demo: http://www.jquerycookbook.com/demos/S3-TablesTabsPanels/20.4-RemoveRow.html

20.5 - Remove Rows by clicking on them, except for the last one left

The previous recipe (20.4) removes any and all rows by clicking on it. However if you want that the table should have atleast one row left, use the following code:

```
var $tbl = $('#someTable tbody tr');
var ctr = $tbl.length;
$tbl.on("click", function () {
  if (ctr == 1) {
    alert('your table should have atleast one
    row')
  }
```

```
else {
      $(this).remove();
      ctr--;
      return false;
    }
});
```

This code uses `$tbl.length` to count the number of rows in the table. Using if..else, we check if the `length == 1`, i.e. if this is the last row left then do not remove the row; else remove the row and reduce the length count by 1.

Live Demo: http://www.jquerycookbook.com/demos/S3-TablesTabsPanels/20.5-RemoveRowLast.html

Further Reading:

http://api.jquery.com/category/manipulation/dom-insertion-outside/

http://api.jquery.com/eq/

http://api.jquery.com/remove/

Simple Application using Knockout.js, jQuery and ASP.NET MVC 4.0 with WEB API (http://www.dotnetcurry.com/showarticle.aspx?ID=847)

Recipe 24

Dynamically Add Thousands of Rows

jQuery gives us great power when it comes to manipulating the DOM, and with *Great power, comes Great responsibility!* Think about some of these functions that you can perform very easily using jQuery:

- hide/delete/insert/update elements
- resize elements/change dimensions
- move/animate elements

and so on..

All these cause what is known as a *reflow* operation

in the browser. Google Developer's documentation defines reflow as "*Reflow is the name of the web browser process for re-calculating the positions and geometries of elements in the document, for the purpose of re-rendering part or all of the document*".

Reflows can be very expensive if not done correctly.

If you plan on becoming a front end engineer, I would advise you to spend some time reading about reflow and repaint operations.

To understand this better, let's take an example where we have to dynamically insert 1000's of rows in a table. We will make use of `append()` to add 5000 rows using two approaches.

In the first approach, we will append the new rows to the table, *every time* the loop iterates.

In the second approach, we will construct a string with the new rows, and then append the string *only once* after the loop is completed. We will then compare the two approaches and derive our conclusion as to which one of them is better and why.

To iterate the loop, I will be using the *for* loop rather than *$.each*. I have observed with some experience

and some tests on jsperf.com (see Recipe 7) that for large loop operations, *for* performs better.

Let's get started. Create a new file called '24-TablePerformance.html' in the 'S3-TablesTabsPanels' folder and the add the following markup:

```
<body>
  <p id="result"></p>
  <table id="someTable">
    <thead>
      <tr>
        <th class="empid">EmpId</th>
        <th class="fname">First Name</th>
        <th class="lname">Last Name</th>
      </tr>
     </thead>
    <tbody>
   </tbody>
  </table>
</body>
```

Now write the following code to add 5000 rows dynamically to the table:

```
$(function () {
  var $tbl = $("#someTable");

  // Approach 1: Using $.append() inside loop
  var t1 = new Date().getTime();
  for(i=0; i < 5000; i++){
    rows = "<tr><td>" + i + "</td><td>FName
    </td><td>LName</td></tr>";
    $tbl.append(rows);
  }
  var t2 = new Date().getTime();
  var t1t2 = t2-t1;
  $('#result').append("Approach 1: Append
  Inside Loop took " + t1t2 + " milliseconds"
  + "</br>");

  // Approach 2: Using $.append() outside
  loop
  var newrows;
  var t3 =  new Date().getTime();
  for(i=0; i < 5000; i++){
    newrows += "<tr><td>" + i + "
    </td><td>FName</td><td>LName</td></tr>";
  }
  $tbl.append(newrows);
  var t4 = new Date().getTime();
  var t3t4 = t4 - t3;
```

```
  $('#result').append("Approach 2:
Append Once    Outside Loop " + t3t4 + "
milliseconds" +      "</br>");
});
```

As discussed, we have two sets of code. Approach 1 calls `append` on each iteration of the loop whereas Approach 2 constructs a string (using +=) with the new rows and calls `append` *only once after* the loop iteration.

The difference is considerable, especially on IE and Safari.

Chrome 30.0.1599.101

Approach 1: Append Inside Loop took 382 milliseconds
Approach 2: Append Once Outside Loop 86 milliseconds

Firefox 23.0.1

Approach 1: Append Inside Loop took 292 milliseconds
Approach 2: Append Once Outside Loop 51 milliseconds

Internet Explorer 10.0.10

Approach 1: Append Inside Loop took 3501 milliseconds
Approach 2: Append Once Outside Loop 533 milliseconds

Safari 5.1.7

Approach 1: Append Inside Loop took 1287 milliseconds
Approach 2: Append Once Outside Loop 51 milliseconds

Our example took just 2 columns and some fixed length data. Imagine in a real world scenario, where there are a couple of columns with variable data; the results would be dramatic.

In Approach 1, every time you are adding a new row to the table *inside the loop*, you are causing a reflow and the entire page geometry gets calculated every time with the new DOM change. In Approach 2, theoretically speaking, the reflow occurs only once, since the rows are constructed and added outside the loop. That's why it's very important for a front-end engineer to understand and evaluate the difference between the two approaches. jsperf.com (see Recipe 7) is your friend here!

Note: In Approach 2, you could squeeze additional performance by not concatenating the string but rather adding it as individual elements of an array, and then use *join* outside the loop to construct the entire string in one go. If you know the length of the array beforehand, that will help too. Check the Further Reading section to learn more about the same.

Live Demo: http://www.jquerycookbook.com/demos/S3-TablesTabsPanels/24-TablePerformance.html

Further Reading:

https://developers.google.com/speed/articles/reflow

http://api.jquery.com/append/

http://www.scottlogic.com/blog/2010/10/15/javascript-array-performance.html

Recipe 26

Performing Calculations in a Table

In this recipe, we will learn to traverse all of the values in a table column, convert the values to numbers, and then sum the values.

Create a new file called '26-TableCalculateTotal.html' in the 'S3-TablesTabsPanels' folder. We will need a simple HTML Table to get started. Our table has an id attribute of `tblProducts` and a <thead> , <tbody> and <tfoot> to go with it.

```
<table id=”tblProducts”>
<thead>
...
</thead>
<tbody>
...
</tbody>
<tfoot>
...
</tfoot>
</table>
```

The Table has 4 columns – Product, Quantity, Price and Sub-Total. It is assumed here that the *Product* and the *Price* info will be prepopulated (in your case probably from a database). When the user enters the Quantity, the *Sub-Total* is automatically calculated using Price x Quantity. The `<tfoot>` contains a row representing a *GrandTotal* which is the sum of all the cells in the Sub-Total column.

```
<thead>
  <tr>
    <td>Product</td>
    <td>Quantity</td>
    <td>Price</td>
    <td>Sub-Total</td>
  </tr>
```

```
</thead>
<tbody>
  <tr>
    <td><input type="text" class="pnm"
    value="Product One" name="pnm" /></td>
    <td><input type="text" class="qty"
    value="" name="qty"/></td>
    <td><input type="text" class="price"
    value="220" name="price"/></td>
    <td><input type="text" class="subtot"
    value="0" name="subtot"/></td>
  </tr>
</tbody>
<tfoot>
  <tr>
    <td></td>
    <td></td>
    <td></td>
    <td><input type="text" class="grdtot"
    value="" name=""/></td>
  </tr>
</tfoot>
</table>
```

Let's now see the script which will calculate the Sub-Total column values. We will also sum all the values in the Sub-Total column, and display the result in the table footer.

The following script selects all the table rows <tr>'s within the table body. The next step is to use jQuery's built-in `each()` iterator to loop over this collection of <tr> elements. For each iteration of the loop, $(this) refers to a <tr> element, which is being assigned to a local variable `$row`.

```
var $tblrows = $("#tblProducts tbody tr");
$tblrows.each(function (index) {
  var $tblrow = $(this);
  ...
});
```

What if your table has no <thead>? We are using a well-structured markup here with a <thead> and <tbody> so we could use $("#tblProd tbody tr") to select all rows in the table body. If you do not have a thead, tbody and instead your first row is a header row, then use this selector to skip the first row:

```
var $tblrows = $("#tblProducts tr:gt(0)");
```

Every time the user enters a value in the Quantity field, the subtotal column should be automatically populated by multiplying the Price with the Quantity entered. The following script achieves this functionality:

```
$tblrow.find('.qty').on('change', function ()
{
  var qty = $tblrow.find("[name=qty]").val();
  var price = $tblrow.find("[name=price]").
  val();
  var subTotal = parseInt(qty,10) *
  parseFloat(price);
  ...
});
```

Both global functions `parseInt` and `parseFloat` convert strings to numbers. I tend to use parseFloat over parseInt, as it is more adaptable in scenarios where I am unsure if all the numbers will be integers. `parseFloat` works with both integers and floating-point numbers. In this example, I am assuming that the values for Quantity are coming from my database, so they do not contain any decimals. In such scenarios, I can safely use `parseInt` for integer columns.

If you observe, the parseInt function has two arguments: a required numeric string, and an optional radix (base). The radix is the number's base, as in base-8 (octal), base-10 (decimal) and base-16 (hexadecimal). If the radix is not provided, it's assumed to be 10, for decimal. Although the second argument is optional, it's considered a good practice to always provide it explicitly.

If the string provided doesn't contain a number, NaN is returned. So we should check to see if the *subTotal* is not a NaN. The next step is to use `toFixed()` method to format the subTotal to two decimal points.

```
if (!isNaN(subTotal)) {

    $tblrow.find('.subtot').val(subTotal.
toFixed(2));
    ...
}
```

We then use `each()` to loop through the subTotal column and sum the text of subtotal in each row and assign the result to the `grandTotal` variable. The last step is to assign the result to the `grandTotal` cell.

```
if (!isNaN(subTotal)) {
  $tblrow.find('.subtot').val(subTotal.
  toFixed(2));
  var grandTotal = 0;
  $(".subtot").each(function () {
    var stval = parseFloat($(this).val());
    grandTotal += isNaN(stval) ? 0 : stval;
  });
  $('.grdtot').val(grandTotal.toFixed(2));
}
```

Save and load the page in your browser, enter the Quantity and you should see the Sub-Total rows populated, as well as a sum of the Sub-Total in the footer of your table.

Product	Quantity	Price	Sub-Total
Product One	1	220	220.00
Product Two	5	18.32	91.60
Product Three	3	29	87.00
Product Four	2	19.99	39.98
			438.58

Live Demo: http://www.jquerycookbook.com/demos/S3-TablesTabsPanels/26-TableCalculateTotal.html

Further Reading:

Why Radix? https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseInt

Recipe 34

Extending the jQuery UI Accordion

By default, the jQuery UI Accordion sets an equal height for all its Panels. However this behaviour leads to extra whitespace in those content panels which have less content, in comparison to the others. In our previous recipe (Recipe 33.1), we saw how to save whitespace by using the `heightStyle` property to enable the panel to be only as tall as its content.

Alternatively, we can also extend the existing jQuery UI Accordion widget and provide a consistent way for users to manipulate the panels. In this recipe, we will provide users with the option to drag and resize the content panel at their own will.

Understanding the jQuery UI Widget Factory

The jQuery UI Widget Factory is simply a function or factory method (`$.widget`) on which all of jQuery UI's widgets are built. It takes the widget name and an object containing widget properties as arguments and creates a jQuery plugin - that provides a consistent API to create and destroy widgets, encapsulate its functionality, allow setting options on initialization, and provide state management. In simple words, it allows you to conform to a defined standard, making it easier for new users to start using your plugin.

All jQuery UI's widgets use the same patterns as defined by the widget factory. If you learn how to use one widget, you'll know how to use all of them.

Create a new HTML page '34-jQueryUIAccordion-extending.html' in the 'S3-TablesTabsPanels' folder. The markup remains the same as the one used in Recipe 33.

Creating a Widget Extension

You can create new widgets with the widget factory by passing the widget name and prototype to `$.widget()` in the following manner:

```
$.widget("custom.newAccordion", {});
```

This will create a `newAccordion` widget in the custom namespace.

However in our case, we want to *extend* the *existing* Accordion widget. To do so, pass the constructor of the jQuery UI Accordion widget (`$.ui.accordion`) to use as a parent, as shown here:

```
$.widget("custom.newAccordion", $.ui.
accordion, {
});
```

By doing so, we are indicating that the `newAccordion` widget should use jQuery UI's Accordion widget as a parent. In order to make our new widget useful, we will add some methods to its prototype object, which is the final parameter passed to `$.widget()`. The shell of our widget will look similar to the following:

```
(function ($) {
  $.widget("custom.newAccordion", $.ui.
  accordion, {
    options: {
    },
    _create: function () {
    },
```

```
 destroy: function () {
    },
    disable: function () {
    },
    enable: function () {
    },
  });
})(jQuery);
```

If you observe, the jQuery UI plugin has been encapsulated in a self-executing anonymous function (function ($) {}). This aliases $ and ensures that jQuery's noConflict() method works as intended.

Let's add some functionality to our widget. Observe the following code:

```
(function( $ ) {
$.widget( "custom.newAccordion", $.ui.
accordion, {
    options: {
        resizable: true
    },
    _create: function () {
        this._super();
        if ( !this.options.resizable ) {
            return;
        }
```

```
this.headers.next().resizable({ handles: "s"
})
        .css({
            "margin-bottom": "5px",
            "border-bottom":"1px dashed",
            "overflow": "hidden"
        });
    },
    _destroy: function () {
        this._super();
        if (!this.options.resizable) {
            return;
        }
        this.headers.next()
        .resizable("destroy")
        .css({
            "margin-bottom": "2px",
            "border-bottom": "1px solid",
            "overflow": ""
        });
    },
});
})(jQuery);
```

We start by providing flexible configuration through `options` and initialize them with default values. In this case, we are setting `resizable` to true by default. The `_create()` method is the widget's constructor. In

this method, we are replacing the `original_create()` method with our new implementation. `this_super()` ensures that the default setup actions of the accordion widget happens.

Widget properties which begin with an underscore such as _create, are considered private.

The next step is to find all content sections of the accordion and apply the `resizable()` widget.

```
this.headers.next().resizable({ handles: “s” })
```

The “s” handle redirects the resizable widget to only show a *south* handle which means the user can use the cursor at the bottom of each Accordion section to resize it up or down.

We are also using some CSS to set the `margin-bottom`, `border-bottom` and `overflow` properties. Instead of adding CSS properties manually, you can even use a class like this

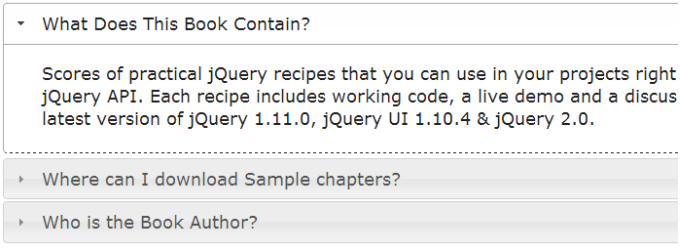
```
this.headers.next().resizable({ handles: “s” }).addClass( “mycss” );
```

The `destroy()` method removes the widget functionality and returns the element back to its pre-init state.

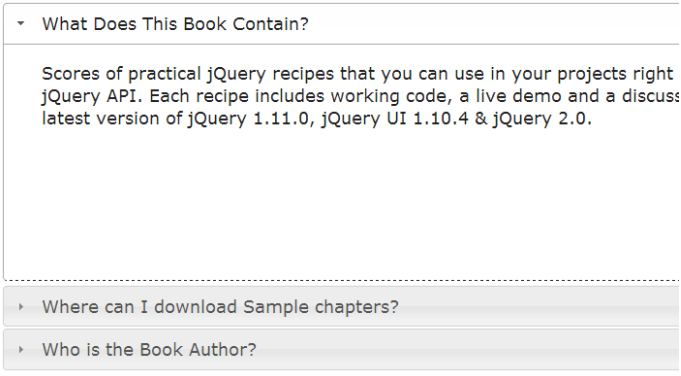
The final step is to call our `newAccordion` widget on the faq div and passing `resizable` to `true`:

```
$(function () {
    $("#faq").newAccordion(
        { resizable: true }
    );
});
```

Save and view the example in your browser and you will see the following Accordion with a dashed bottom border, 5px spacing and a resizable panel:



You can resize the panel by placing the cursor at the bottom of each section and dragging it up and down to suit your needs. The screenshot here shows the expanded accordion:



Live Demo: http://www.jquerycookbook.com/demos/S3-TablesTabsPanels/34-jQueryUIAccordion-extending.html

Further Reading: http://api.jqueryui.com/jQuery.widget

Recipe 56

Chain AJAX Requests with Deferred

Imagine a scenario where you have a bunch of functions to execute asynchronously, but each function depends on the result of the previous one and you do not have an idea when each function will complete execution and pass the result to the next function.

In such cases, you can write callbacks. Callbacks are useful when working with background tasks because you don't know when they will complete. Here's a prototype of callbacks in action:

```
A(function () {
  B(function () {
    C()
  })
});
```

and the callbacks can be defined as:

```
function A(callback) {
  $.ajax({
  //...
    success: function (result) {
    //...
      if (callback) callback(result);
      }
  });3
}
```

and so on..

However this code style leads to too much nesting and becomes unreadable if there are too many callback functions. The same functionality can be achieved very elegantly without too much nesting using Deferred and Promise. Let's cook up an example:

We have four functions A(), B(), C() and D() that will execute asynchronously. However function B() relies

on the result of function A(). Similarly function C() relies on the result of function B() and so on. The task is to execute these functions *one after the other*, and also make the results of the previous function available in the next one.

Asynchronous requests cannot be guaranteed to finish in the same order that they are sent. However using deferred objects, we can make sure that the callbacks for each async request runs in the required order

Create a new file '56-ChainFunctions.html' in the 'S7-Ajax' folder. Use the following code:

```
$(function () {
  function A() {
    writeMessage("Calling Function A");
    return $.ajax({
      url: "scripts/1.json",
      type: "GET",
      dataType: "json"
    });
  }
function B(resultFromA) {
  writeMessage("In Function B. Result From A
=       " + resultFromA.data);
  return $.ajax({
```

```
    url: “scripts/2.json”,
    type: “GET”,
    dataType: “json”
  });
}

function C(resultFromB) {
  writeMessage(“In Function C. Result From B
  = “ + resultFromB.data);
  return $.ajax({
    url: “scripts/3.json”,
    type: “GET”,
    dataType: “json”
  });
}

function D(resultFromC) {
  writeMessage(“In Function D. Result From C
  = “ + resultFromC.data);
}

A().then(B).then(C).then(D);

function writeMessage(msg) {
  $(“#para”).append(msg + “</br>”);
    }
});
```

Observe this important piece of code:

```
A().then(B).then(C).then(D);
```

From the jQuery Documentation:

Callbacks are executed in the order they were added. Since deferred.then returns a Promise, other methods of the Promise object can be chained to this one, including additional .then() methods.

And that's what we are doing here. Every Ajax method of jQuery already returns a *promise*. When the Ajax call in function A() completes, it resolves the *promise*. function B() is then called with the results of the Ajax call as its first argument. When the Ajax call in B() completes, it resolves the *promise* and function C() is called with the results of that call and so on. Here we are just adding `return` in every function to make this chain work.

Live Demo: http://www.jquerycookbook.com/demos/S7-Ajax/56-ChainFunctions.html

By the way, you may have observed that we are using the same `type` and `datatype` settings for all the AJAX calls. You can clean up your code by specifying global settings using `$.ajaxSetup`.

Create a new file '56.1-ChainFunctions.html' in the 'S7-Ajax' folder and use the following code:

```
$.ajaxSetup({
  type: 'GET',
  dataType: "json",
  delay: 1
});
```

and then reduce each call to:

```
function A() {
  writeMessage("Calling Function A");
  return $.ajax({
    url: "scripts/1.json",
  });
}
```

Run the sample and you will get the same output.

Live Demo: http://www.jquerycookbook.com/demos/S7-Ajax/56.1-ChainFunctions.html

End of Sample Chapters