15 How to Create Dynamic HTML Pages

JavaScript began as a language that ran inside of browsers in order to make web pages more dynamic.

One way to make web pages dynamic was to use JavaScript to create the HTML from data and calculations. There are three ways this can be done:

- call the **document.writeln()** function to write HTML at specific point as a page is loaded
- set the **innerHTML** field of an element on a page after the page is loaded
- modify elements in the document object model (DOM) after the page is loaded

We'll describe each of these methods, but, for most purposes, you will use the second approach.

> Note that many modern frameworks, such as **AngularJS** and **ReactJS**, will handle much of the HTML creation for you, using the fastest methods available. They do require quite a bit of study to use effectively and cleanly.

document.writeln

A browser loads and displays the text of an HTML page from top to bottom. As it proceeds, if it comes to a **script** element, it stops processing HTML and starts executing the code inside the **script**. If that code calls **document.writeln()** one or more times, then any text wrtten by that function will be inserted into the HTML stream, just as if it were in the source text.

Here's a simple example often used to demonstrate this feature in a minimal HTML page.

```
<!DOCUMENT HTML>
<html>
  <head><title>Welcome</title></head>
  <body>
    <p>Welcome to this page.</p>

    <p>Today is
    <script type="text/javascript">
    document.write(new Date().toDateString());
    </script>
    &mdash; What's new?
    </p>
  </body>
</html>
```

TIP:
Copy and paste this code into JSBin and click the **Run with JS** button to see the results.

Notice how the browser executed the **script** right in the middle of the paragraph text.

While **document.write()** still works, it's not popular, because it leads to hard to maintain HTML pages. It's an example of what is called **intrusive** JavaScript. It's better to use **non-intrusive** methods to modify the HTML from JavaScript outside the HTML. Such code is loaded at the end of the HTML page. The code accesses elements of the web page the browser has created and modifies their contents.

Structuring HTML for Modification

The easiest way to prepare a page element in HTML for modification by non-intrusive JavaScript is to give the HTML element an **id**. This doesn't change how the HTML displays, but does make it very easy to find that element in JavaScript.

As an example, let's change the HTML from the example above to be suitable for modification. We

- copy the HTML
- move the existing **script** element to the end of the page
- remove the existing JavaScript code
- add a **span** element where the date will go
- put an **id** on the **span**

```
<!DOCUMENT HTML>
<html>
  <head><title>Welcome</title></head>
  <body>
    <p>Welcome to this page.</p>

    <p>Today is <span id="today"></span> &mdash; What's new?
    </p>

    <script type="text/javascript">
    </script>
  </body>
</html>
```

You can see how this approach already leads to much more readable HTML. The effect is even greater with more complicated JavaScript.

Now we need add the JavaScript code to insert the date strings in the **span**.

The JavaScript to get the element with the **id** "today" is simple: **document.getElementById("today")**.

TIP:
If you're using jQuery, you'd write **$("#today")**. You also have many more ways to select elements to change.

Once you have an element, how do you change it?

innerHTML

The simplest way to add HTML to an element is to set its **innerHTML** field. You can set **innerHTML** to any string containing legal HTML. That means simple text, or text with HTML tags. So we can finish our example with this code:

```
<!DOCUMENT HTML>
<html>
  <head><title>Welcome</title></head>
  <body>
    <p>Welcome to this page.</p>

    <p>Today is <span id="today"></span> &mdash; What's new?
    </p>

    <script type="text/javascript">
      document.getElementById("today").innerHTML = new Date().toDateString();
    </script>
  </body>
</html>
```

Copy and paste this code into JSBin and click the **Run with JS** button to see the results. They should be identical to the previous example.

If we wanted to make the date bold, we just need to use standard JavaScript string building operators to change the text stored to this:

```
document.getElementById("today").innerHTML = "<b>" + new Date().toDateString() + "</b>";
```

TIP:
If you're using jQuery, use the **html()** method to change the inner HTML, e.g., **$("#today").html(new Date().toDateString())**.

Modify the DOM Directly

The DOM, or Document Object Model, is the data structure every browser builds to represent a **parsed** web page.

Parsing is when a browser reads HTML like "<p>Today is <span>Aug 10</span> -- welcome!</p>" and constructs some internal data structures for a paragraph object, with some text, inside of which is a span object with some more text.

The W3C consortium has defined what the DOM looks like. Basically,

- Everything is a **node** -- the document, a paragraph, a list element, etc.
- Nodes have **children**. Children are the nodes nested inside the node, e.g.,
  - the **li** elements in a list,
  - the **img** and text elements in a **p** paragraph
- If node **A** has child nodes **B** and **C**, then **B** and **C** each have **A** as their one and only **parent**.

When you assign the **innerHTML** of a node **X**, you are indirectly modifying the DOM. You're telling the browser to parse the string of HTML, creating a node or nodes, and then make those nodes the new children of **X**.

Alternatively, you can create the nodes in JavaScript and insert them yourself. There is a large library of JavaScript functions for creating and managing the nodes in the DOM for a page.

To give a small taste of how this works, we'll use the same example as before. We start with this HTML, waiting for some non-intrusive JavaScript to fill in a date string.

```
<!DOCUMENT HTML>
<html>
  <head><title>Welcome</title></head>
  <body>
    <p>Welcome to this page.</p>

    <p>Today is <span id="today"></span> &mdash; What's new?
    </p>

    <script type="text/javascript">
    </script>
  </body>
</html>
```

With DOM functions, what we need to do is:

- create a text node to hold the date string
- add the text node as a child of the **span**

The first step is easy. Use the function **document.createTextNode()**.

The second step requires deciding what you want to do about any existing children. If you are adding a new child **C** to a node **X**, your choices are

- add **C** to the end of any existing children, using **X.appendChild(C)**
- insert **C** before some existing child **Y**,using **X.insertBefore(C, Y)**
- replace some existing child **Y** with **C**,using **X.replaceChild(C, Y)**

Most of the time, **appendChild()** is used, either because the node being modified was created with no children, as in our example, or because we don't want to erase existing data. So the complete example is

```
<!DOCUMENT HTML>
<html>
  <head><title>Welcome</title></head>
  <body>
    <p>Welcome to this page.</p>

    <p>Today is <span id="today"></span> &mdash; What's new?
    </p>

    <script type="text/javascript">
      document.getElementById("today").appendChild(document.createTextNode(new Date().toDateString()))
    </script>
  </body>
</html>
```

Choosing a DOM modification method

document.write() is almost never used, because it is intrusive and makes the HTML harder to maintain.

The **innerHTML** method is simple and fast. The string construction might get messy but it's usually still shorter than calling the DOM functions.

Using the DOM functions to create and add a large number of nodes, e.g., for a big table or list, can be slow because every time you add a child, the browser has to update the display. When you use the **innerHTML** approach, the browser does nothing until the final assignment to **innerHTML**.

The DOM functions are most useful when modifying a DOM structure that already exists, e.g., reordering the rows in a table, deleting elements from a list, and so on.

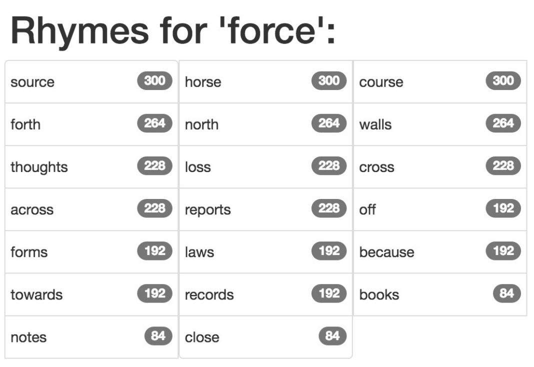
> There are ways to code around the DOM speed problem, by creating the nodes in a node not in the DOM, and then adding at the end.

Example: Rendering JSON Data in an HTML List

To give a small but realistic example of using JavaScript to create HTML, consider data returned by a JSON web service. For example, the RhymeBrain web service returns rhymes for English words. The URL http://rhymebrain.com/talk?function=getRhymes&word=force&maxResults=20 gets the top 20 rhymes for the word "force" (according to RhymeBrain) in the following JSON:

```
[
 {"word":"source","freq":25,"score":300,"flags":"bc","syllables":"1"},
 {"word":"horse","freq":24,"score":300,"flags":"bc","syllables":"1"},
 {"word":"course","freq":26,"score":300,"flags":"bc","syllables":"1"},
 {"word":"forth","freq":24,"score":264,"flags":"bc","syllables":"1"},
 {"word":"north","freq":24,"score":264,"flags":"bc","syllables":"1"},
 {"word":"walls","freq":24,"score":264,"flags":"bc","syllables":"1"},
 {"word":"thoughts","freq":24,"score":228,"flags":"bc","syllables":"1"},
 {"word":"loss","freq":25,"score":228,"flags":"bc","syllables":"1"},
 {"word":"cross","freq":24,"score":228,"flags":"bc","syllables":"1"},
 {"word":"across","freq":25,"score":228,"flags":"bc","syllables":"2"},
 {"word":"reports","freq":24,"score":228,"flags":"bc","syllables":"2"},
 {"word":"off","freq":26,"score":192,"flags":"bc","syllables":"1"},
 {"word":"forms","freq":25,"score":192,"flags":"bc","syllables":"1"},
 {"word":"laws","freq":25,"score":192,"flags":"bc","syllables":"1"},
 {"word":"because","freq":27,"score":192,"flags":"bc","syllables":"2"},
 {"word":"towards","freq":25,"score":192,"flags":"bc","syllables":"2"},
 {"word":"records","freq":24,"score":192,"flags":"bc","syllables":"2"},
 {"word":"books","freq":25,"score":84,"flags":"bc","syllables":"1"},
 {"word":"notes","freq":24,"score":84,"flags":"bc","syllables":"1"},
 {"word":"close","freq":25,"score":84,"flags":"bc","syllables":"1"}
]
```

Let's write code to take JSON like that and generate an HTML list like this:



Rhymes for 'force':

source	300	horse	300	course	300
forth	264	north	264	walls	264
thoughts	228	loss	228	cross	228
across	228	reports	228	off	192
forms	192	laws	192	because	192
towards	192	records	192	books	84
notes	84	close	84		

For the HTML, we'll use a few Bootstrap CSS tricks to get this look.

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>Demo List</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
  </head>
  <body>
    <div class="container">
    <h1>Rhymes for '<span id="word"></span>':</h1>

    <ul id="rhymes" class="list-inline">
    </ul>
    </div>
    <script type="text/javascript" src="demo-list.js"></script>
  </body>
</html>
```

There are no items in the list yet. They will be generated by the code in **demo-list.js**.

We want to take the array of rhyme objects in the JSON returned by RhymeBrain and return the HTML for a list of items, with just the word and it's rhyme score.

Here's the code to create the HTML for a single item:

```
function getItemHtml(rhyme) {
  return "<li class='list-group-item' style='width: 12em'><span class='badge'>" +
    rhyme.score + "</span>" + rhyme.word + "</li>";
}
```

This creates the HTML for a Bootstrap list item, and embeds the score in a Bootstrap **badge**.

To create the HTML for entire list, we map this function to every item to produce an array of the HTML strings, then join those strings together with an empty separator:

```
function getListHtml(data) {
  return data.map(getItemHtml).join("");
}
```

To finish the code off, we define a function to create and store this HTML in the **ul** on the page. We define another simple function to store the word being rhymed in the **span** reserved for it in the **h1** element.

```
function renderList(data) {
  var html = getListHtml(data);
  document.getElementById("rhymes").innerHTML = html;
}

function renderWord(word) {
  document.getElementById("word").innerHTML = word;
}
```

To test this out, put all the above code in **demo-list.js**, followed by these two function calls. The first stores the word "force" and the second renders the list of the first twenty rhymes, as returned by RhymeBrain.

```
renderWord("force");

renderList([{"word":"source","freq":25,"score":300,"flags":"bc","syllables":"1"},
 {"word":"horse","freq":24,"score":300,"flags":"bc","syllables":"1"},
 {"word":"course","freq":26,"score":300,"flags":"bc","syllables":"1"},
 {"word":"forth","freq":24,"score":264,"flags":"bc","syllables":"1"},
 {"word":"north","freq":24,"score":264,"flags":"bc","syllables":"1"},
 {"word":"walls","freq":24,"score":264,"flags":"bc","syllables":"1"},
 {"word":"thoughts","freq":24,"score":228,"flags":"bc","syllables":"1"},
 {"word":"loss","freq":25,"score":228,"flags":"bc","syllables":"1"},
 {"word":"cross","freq":24,"score":228,"flags":"bc","syllables":"1"},
 {"word":"across","freq":25,"score":228,"flags":"bc","syllables":"2"},
 {"word":"reports","freq":24,"score":228,"flags":"bc","syllables":"2"},
 {"word":"off","freq":26,"score":192,"flags":"bc","syllables":"1"},
 {"word":"forms","freq":25,"score":192,"flags":"bc","syllables":"1"},
 {"word":"laws","freq":25,"score":192,"flags":"bc","syllables":"1"},
 {"word":"because","freq":27,"score":192,"flags":"bc","syllables":"2"},
 {"word":"towards","freq":25,"score":192,"flags":"bc","syllables":"2"},
 {"word":"records","freq":24,"score":192,"flags":"bc","syllables":"2"},
 {"word":"books","freq":25,"score":84,"flags":"bc","syllables":"1"},
 {"word":"notes","freq":24,"score":84,"flags":"bc","syllables":"1"},
 {"word":"close","freq":25,"score":84,"flags":"bc","syllables":"1"}]);
```

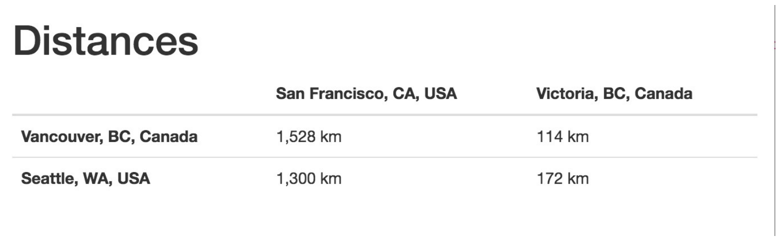
In a real application, you would add a form field to the HTML to enter a word, and use an AJAX call to get the rhymes to display.

Example: Rendering JSON Data in an HTML Table

Here's a more complicated example.

The Google Maps Distance Matrix API return distances between cities in JSON form. You need an API key to use this service for real, but we'll just use the data returned by their small example with 4 cities.

We want to display the data in a nicely formatted table, with a column for each destination city, and a row for each origin city, like this:



Distances

	San Francisco, CA, USA	Victoria, BC, Canada
Vancouver, BC, Canada	1,528 km	114 km
Seattle, WA, USA	1,300 km	172 km

We don't want to assume anything about what the cities are. All that will come from the data.

Here's some simple HTML that will generate the above output, once data is added.

```
<!DOCTYPE HTML>
```

```
<html>
  <head>
    <title>Demo Table</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
  </head>
  <body>
    <div class="container">
      <h1>Distances</h1>

      <table class="table">
         <thead id="table-headers">
         </thead>
         <tbody id="table-rows">
         </tbody>
      </table>
    </div>
    <script type="text/javascript" src="demo-table.js"></script>
  </body>
</html>
```

It uses Bootstrap 3 to get a nice looking table. The headers and rows will be created by the code in **demo-table.js**.

In the JavaScript, we will need a loop has to convert the list of destination cities in the data into an HTML table row of column headers.

We can do this code very similar to the list example. If the variable **data** holds the JSON returned by Google, then **data.destination_addresses** is the array of names we need.

```
function getHeadersHtml(data) {
  return "<tr><th></th>" + data.destination_addresses.map(function(dest) {
    return "<th>" + dest + "</th>";
  }).join("") + "</tr>";
}
```

This creates an HTML table row, with a **th** for each destination. It use **map()** and **join()** as before. Note that it also includes an empty first column. That will be needed for the origin cities on the data rows.

Rendering this HTML into the desired element of the page is easy:

```
function renderHeaders(data) {
  var html = getHeadersHtml(data);
  document.getElementById("table-headers").innerHTML = html;
}
```

Doing the data rows is more complicated. There is a loop nested in a loop:

- There is a loop over every origin city.
- For each origin, there is a loop over every destination.

The JSON for one origin city looks like this:

```
{"elements":
  [{"distance":{"text":"1,528 km","value":1528361},
    "duration":{"text":"14 hours 47 mins","value":53236},
    "status":"OK"},
   {"distance":{"text":"114 km","value":114166},
    "duration":{"text":"3 hours 10 mins","value":11415},
    "status":"OK"}
  ]
}
```

We want to map this to a set of **td** entries with just the distance texts, i.e., "1,528 km" and "114 km". Let's define a function that takes data in this form and returns the desired HTML:

```
function getColumnsHtml(row) {
  return row.elements.map(function(element) {
    return "<td>" + element.distance.text + "</td>";
  }).join("")
}
```

Now we need to call that function inside another loop over all the origin cities. The top-level JSON looks like this

```
{
   "destination_addresses" : [ "San Francisco, CA, USA", "Victoria, BC, Canada" ],
   "origin_addresses" : [ "Vancouver, BC, Canada", "Seattle, WA, USA" ],
   "rows" : [
      { "elements" : [ ... ] },
      { "elements" : [ ... ] },
      ...
   ]
}
```

Therefore we want to map over **data.rows**. For each row, we need to retrieve the appropriate origin city. Fortunately, in JavaScript, ***array*.map(*function*)** passes two arguments to ***function***:

1. each element of the array, one at a time
2. the index (zero-based) of that element

We can use the index to get the origin city from **data.origin_addresses**.

```
function getRowsHtml(data) {
  return data.rows.map(function(row, i) {
    return "<tr><th>" + data.origin_addresses[i] + "</th>" +
      getColumnsHtml(row) + "</tr>";
    }).join("");
}
```

Again, storing this HTML on the web page is easy:

```
function renderRows(data) {
  var html = getRowsHtml(data);
  document.getElementById("table-rows").innerHTML = html;
}
```

All that's left is to define one function to render both parts of the table, and finally a line of code to call that function with our desired data. In a real application, the data would come from an AJAX call. For demonstration purposes, we pass a hard-coded JSON object, created by taking the JSON from this link and minifying to save space with this page.

```
function renderTable(data) {
  renderHeaders(data);
  renderRows(data);
}

renderTable({"destination_addresses":["San Francisco, CA, USA","Victoria, BC, Canada"],"origin_addresses":["Vancouver, BC, Canada","Seattle, WA, USA"],
"rows":[{"elements":[{"distance":{"text":"1,528 km","value":1528361},"duration":{"text":"14 hours 47 mins","value":53236},"status":"OK"},{"distance":
{"text":"114 km","value":114166},"duration":{"text":"3 hours 10 mins","value":11415},"status":"OK"}]},{"elements":[{"distance":{"text":"1,300 km",
"value":1299975},"duration":{"text":"12 hours 21 mins","value":44447},"status":"OK"},{"distance":{"text":"172 km","value":171688},"duration":
{"text":"4 hours 37 mins","value":16602},"status":"OK"}]}],"status":"OK"});
```