Riverside JS Workshop
JavaScript Skills Ladder, Lesson 2.4

# The To-Do Application

***Lesson Objectives:***

***When you complete this lesson, you will be able to:***

- *create a JSON file that contains to-do items.*
- *create some HTML and CSS for this application.*
- *add content to your application.*
- *create and array of to-do objects.*
- *update your page with to-do items.*

You've built an Ajax application that reads JSON data from an external file. The ability to retrieve data from an external source is one key feature of an Ajax application; a second feature that's just as important, is the ability to update your web page (or web *application*) dynamically with that data.

In this lesson, we'll begin building a To-Do List application. Eventually this application will let you both retrieve *and* save to-do list items using a web application that you've built. For this first part of the To-Do List application, we'll expand the Pets application and actually parse the JSON data, then update your web page using that data.

You'll recognize the code below. Feel free to reuse some of your project solution, or, if you'd prefer, you can start from scratch. We'll assume that you're starting over, so you can create fresh files using the instructions below.

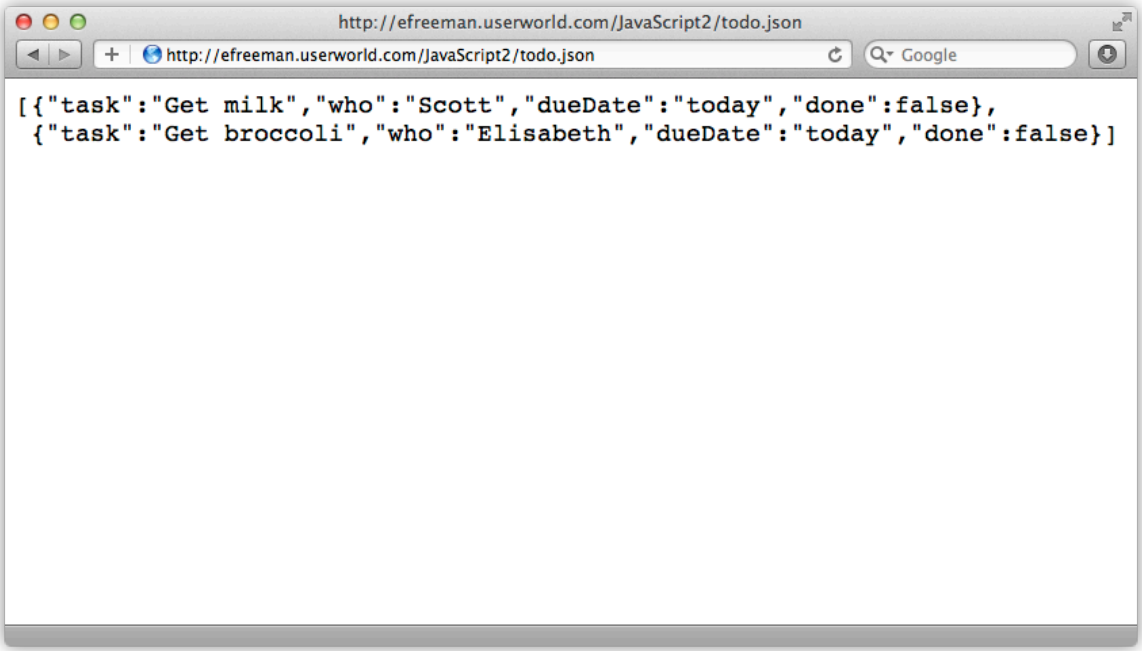
## Creating the JSON Data File

To begin building the To-Do List application, you'll create a JSON file containing some to-do items. (You'll learn how to do that through the application itself.) Just like with the Pets example, you'll create a new file and write some data using the JSON format (or you can reuse the to-do list you created in the previous lesson's project):

***CODE TO TYPE:***

```
[{"task":"get milk","who":"Scott","dueDate":"today","done":false},
 {"task":"get
broccoli","who":"Elisabeth","dueDate":"today","done":false}]
```

Save this file in your work folder as *todo.json,* and navigate to that file with your browser . You see your to-do list JSON data in a web page, it looks like this:




Your web application will read in these items using Ajax. Notice that there are two to-do items: one for Scott and one for Elisabeth. Each item has four properties: `task`, `who`, `dueDate`, and `done`. All the property values are strings, except for `done`, which is a Boolean.

If you had to represent these to-do items as an object, what would that object look like? Give it some thought and we'll come back to that question shortly.

## Creating the HTML and CSS

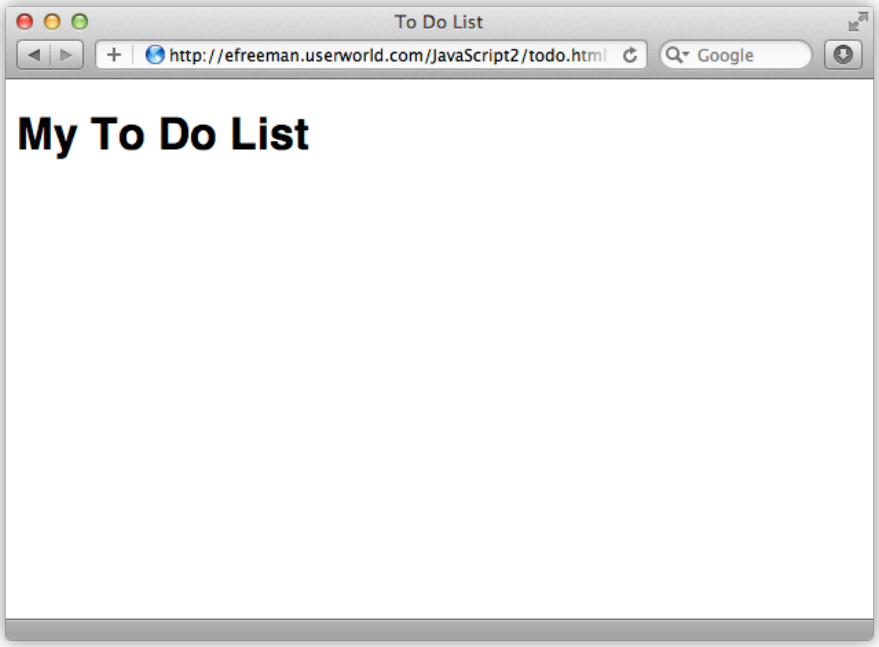
Next, let's create the HTML & CSS for this application. We'll keep it relatively simple, just like we did with pets; for now, all we need is a heading and a `<div>` element. Create a new file and type in this HTML and CSS:

***CODE TO TYPE:***

```
<!doctype html>
<html>
<head>
<title>To-Do List</title>
  <meta charset="utf-8">
  <script src="todo.js"></script>
  <style>
        body {
```

```
            font-family: Helvetica, Arial, sans-serif;
        }
  </style>
</head>
<body>
  <h1>My To-Do List</h1>
  <div id="todoList">
  </div>
</body>
</html>
```

Save this file in your work folder as *todo.html* and load it into your browser. You see this web page:




There's nothing in the web page, because we haven't added any content to it yet.

## Adding the Content

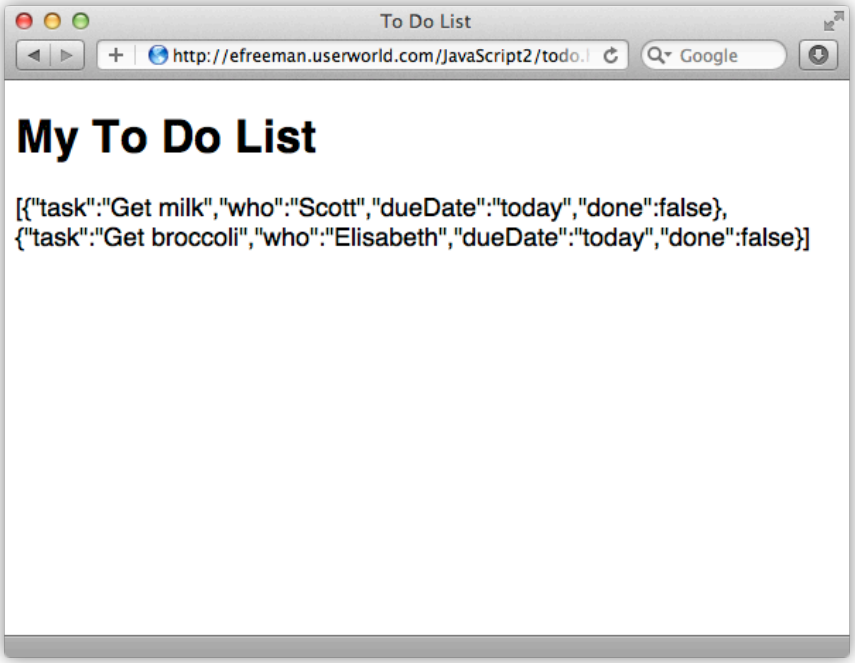
Okay, go ahead and create the JavaScript to read in the JSON from the "todo.json" file, just like we did with Pets:

***CODE TO TYPE:***

```
window.onload = init;

function init() {
    getTodoData();
```

```
}

function getTodoData() {
    var request = new XMLHttpRequest();
    request.open("GET", "todo.json");
    request.onreadystatechange = function() {
        var listDiv = document.getElementById("todoList");
        if (this.readyState == this.DONE && this.status == 200) {
            if (this.responseText) {
                listDiv.innerHTML = this.responseText;
            }
            else {
                console.log("Error: Data is empty");
            }
        }
    };
    request.send();
}
```

Save it in your work folder as *todo.js*. Open or refresh *todo.html* (which links to this *todo.js* file with the <script> element at the top) in your browser. The JSON data from *todo.json* in your web page, looks like this:




So far, so good. It's the same process we applied to Pets, just revised for a To-Do List, and using slightly different JSON data. Here are the steps we take to get there:

1. Call the `function init()` once the page finishes loading.

2. Call `getTodoData()` from `init()`, which is where we use Ajax to retrieve the *todo.json* data.
3. Create an XMLHttpRequest object to make the request for the data.
4. Create an `onreadystatechange` handler for the request.
5. Add the data to the page.

So now that you've got your hands on the JSON data, let's do something better than just spitting it out to the web page as is. What we'd really like to do is make a nice-looking list, right?

## Create an array of To-Do Objects

To create a nice to-do list out of the JSON data, we need to *parse* the data using `JSON.parse()`. As we learned earlier, `JSON.parse()` parses, or *deserializes*, JSON-formatted data and turns that data into a JavaScript object. In our case, that object will be a `Todo` object.

If you were to create a `Todo` object in JavaScript, you'd write something like this:

***OBSERVE:***

```
var todoObject = {
    task: "Get Milk",
    who: "Scott",
    dueDate: "today",
    done: false
};
```

Remember, this is a *literal object*, that is, an object you typed in literally, rather than created using a constructor. Notice how closely this `todoObject` resembles the items in the *todo.json* file you created earlier.

When we read in the JSON data and turn the JSON to-do items into JavaScript objects, we need somewhere to put those objects; we'll stash them in an array. For this step, we'll take a look at that array in the console. After that, we'll actually use the array of objects to update the page. Update *todo.js* as shown:

***CODE TO TYPE:***

```
var todos = new Array();

window.onload = init;

function init() {
    getTodoData();
```

```
}

function getTodoData() {
    var request = new XMLHttpRequest();
    request.open("GET", "todo.json");
    request.onreadystatechange = function() {
        var listDiv = document.getElementById("todoList");
        if (this.readyState == this.DONE && this.status == 200) {
            if (this.responseText) {
                listDiv.innerHTML = this.responseText;
                parseTodoItems(this.responseText);
            }
            else {
                console.log("Error: Data is empty");
            }
        }
    };
    request.send();
}

function parseTodoItems(todoJSON) {
    if (todoJSON == null || todoJSON.trim() == "") {
        return;
    }
    var todoArray = JSON.parse(todoJSON);
    if (todoArray.length == 0) {
        console.log("Error: the to-do list array is empty!");
        return;
    }
    for (var i = 0; i < todoArray.length; i++) {
        var todoItem = todoArray[i];
        todos.push(todoItem);
    }
    console.log("To-do array: ");
    console.log(todos);
}
```

Save it, and open or refresh *todo.html* in your browser. You won't see anything in the web page now, but you'll see some output in your developer console. Remember, if you're in Safari or Chrome, you might need to use the arrow in the Developer console window to access the console.

Let's walk through the new code. First, we add an empty array, `todos`, to hold all the objects that we create as we parse the JSON data from the *todo.json* file:

***OBSERVE:***

```
var todos = new Array();
```

Note that this is a global variable so we can access it from within any function.

Next, we update the `onreadystatechange` handler in the `getTodoData()` function to call the function `parseTodoItems()` and pass in the `responseText`, which holds the JSON data from the *todo.json* file:

***OBSERVE:***

```
parseTodoItems(this.responseText);
```

And of course, we implement the `parseTodoItems()` function:

***OBSERVE:***

```
function parseTodoItems(todoJSON) {
    if (todoJSON == null || todoJSON.trim() == "") {
        return;
    }
    var todoArray = JSON.parse(todoJSON);
    if (todoArray.length == 0) {
        console.log("Error: the to-do list array is empty!");
        return;
    }
    for (var i = 0; i < todoArray.length; i++) {
        var todoItem = todoArray[i];
        todos.push(todoItem);
    }
    console.log("To-do array: ");
    console.log(todos);
}
```

This function is where the magic happens: where we turn JSON from a file into real HTML objects that dynamically appear in your web page.  Well, it won't seem like magic anymore, because you'll know how to do it.

In `parseTodoItems`, the JSON data is passed in as a variable named `todoJSON`. In the function we test to make sure `todoJSON` is not null or the empty string. If the file is empty (that is, there are no to-dos yet), then we have no JSON to parse and nothing to add to the page. If we *try* to parse an empty JSON string, we'll get an error message. We can avoid that by testing the string first.

We'll look at strings in more detail in a later lesson, but for now, we're using the `trim()` function (actually a String method!) on the string todoJSON. This gets rid of extra white space at the beginning or end of a string. So if you have a string like this:

***OBSERVE:***

```
var myString = "  There are a few empty spaces.   "
```

...with empty spaces at the beginning and/or end, after calling `myString.trim()`, you'll have a string like this:

***OBSERVE:***

```
"There are a few empty spaces."
```

If the string is empty, then we just return from the function (that is, the rest of the function isn't executed). Now let's review the next bit of code:

***OBSERVE:***

```
var todoArray = JSON.parse(todoJSON);
if (todoArray.length == 0) {
    console.log("Error: the to-do list array is empty!");
    return;
}
```

Once we know the string isn't empty, we can parse it using `JSON.parse()`. This turns the `todoJSON` string into an object. In our case, we know that we actually have an *array of objects* because that's how we designed the data in the todo.json file (take another look at the contents of the file and notice the square brackets, *[ ],* surrounding the rest of the data; that means array). So instead of putting the result of the call to `JSON.parse()` into an object variable, we are putting it into an array variable, the `todoArray`.

So, why don't we just put the result of the `JSON.parse()` into the `todos` array? That's a great question! And the answer is: because later on, we're going to add a form to the page to let you add new to-do items to your list. We want to keep the to-dos that are in the *file* separate from the to-dos managed by the *page*, so we have this temporary array, `todoArray`, which we use to get the to-do items from the file. Later we'll add them to the `todos` array.

Before we try to add items from the `todoArray` to the `todos` array, we need to make sure there's something in it. What if you made an *empty* array (Like this: [ ]) in your todo.json file? That would be a valid JSON string, but it wouldn't contain any objects. We need to check for that, and return from the function if the array is empty.

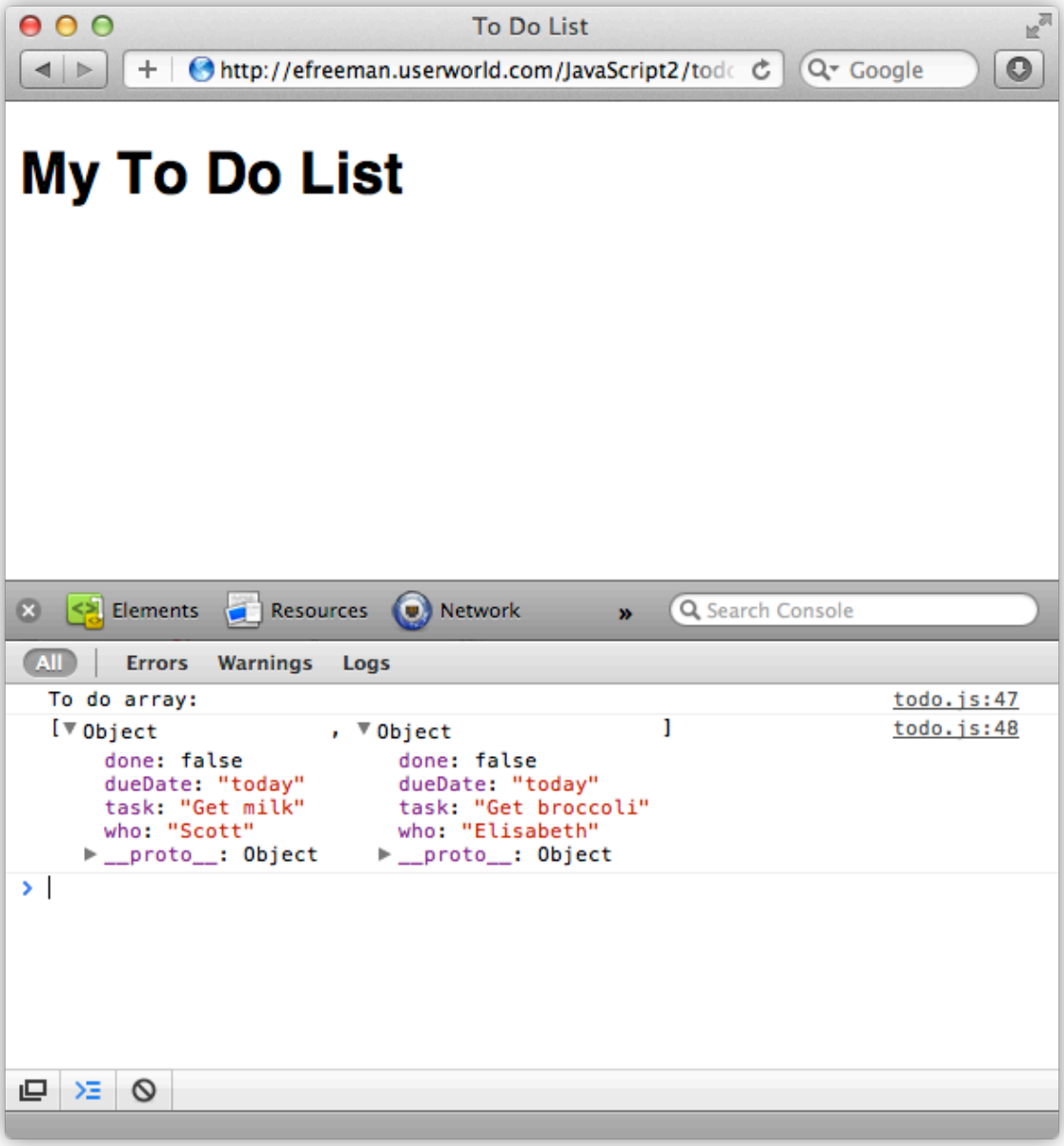
Okay, at this point we have an array with at least one object in it. Here's the next (and final) piece of code from the `parseTodoItems` function:

***OBSERVE:***

```
for (var i = 0; i < todoArray.length; i++) {
    var todoItem = todoArray[i];
    todos.push(todoItem);
}
console.log("To-do array: ");
console.log(todos);
```

Now, we want to put the objects in the `todoArray` into the `todos` array. We'll iterate (loop) through all the items in the `todoArray` and copy them into the todos array. First we store the item from the `todoArray[i]` in the variable `todoItem`, and then use the `Array` object method `push()` to add the `todoItem` onto the end of the `todos` array. (We don't actually need the intermediate `todoItem` variable; can you see how you'd do this without it?)

Finally, to make sure the `todos` array has the right stuff in it, we use `console.log()` to display the array in the console. Here's how it looks in Safari and Chrome:




## Update the Page with To-Do Items

The next step is to get the items from the `todos` array into your web page. We're creating a to-do *list*, so let's use an HTML list to put the items in. First, change *todo.html* so that the "todoList" element is a `<ul>` instead of a `<div>`, like this:

***CODE TO TYPE:***

```
<!doctype html>
<html>
```

```
<head>
<title>To-Do List</title>
  <meta charset="utf-8">
  <script src="todo.js"></script>
  <style>
        body {
            font-family: Helvetica, Arial, sans-serif;
        }
  </style>
</head>
<body>
  <h1>My To-Do List</h1>
  <div id="todoList"><ul id="todoList">
  </div></ul>
</body>
</html>
```

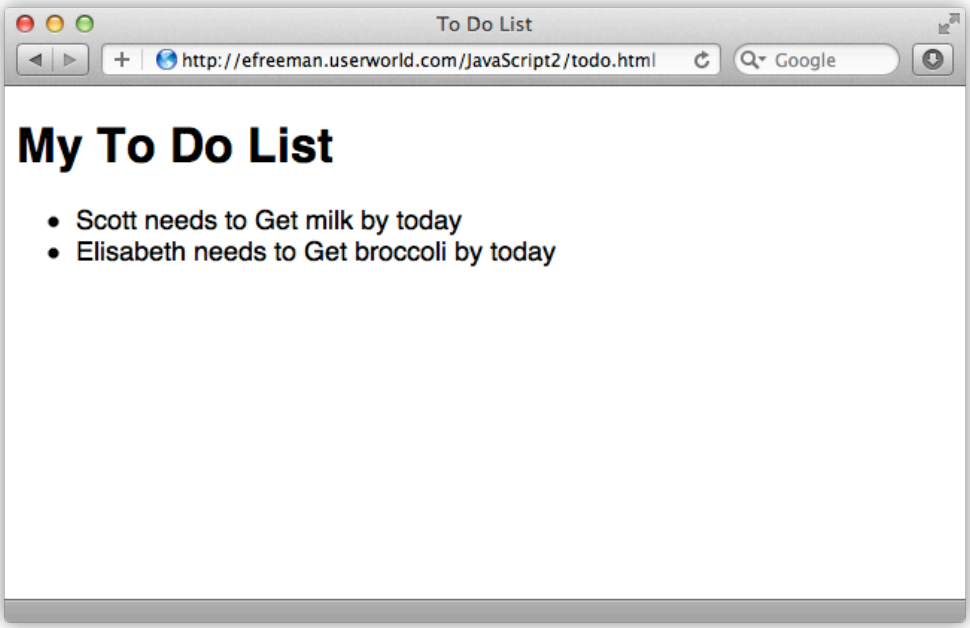
Of course, the "todoList" list is empty (it has no `<li>` elements in it) because we're going to use JavaScript to add `<li>` elements created using the objects stored in the `todos` array. Update *todo.js* as shown:

***CODE TO TYPE:***

```
var todos = new Array();

window.onload = init;

function init() {
    getTodoData();
}

function getTodoData() {
    var request = new XMLHttpRequest();
    request.open("GET", "todo.json");
    request.onreadystatechange = function() {
        if (this.readyState == this.DONE && this.status == 200) {
            if (this.responseText) {
                parseTodoItems(this.responseText);
                addTodosToPage();
            }
            else {
                console.log("Error: Data is empty");
            }
        }
    };
    request.send();
}

function parseTodoItems(todoJSON) {
```

```
    if (todoJSON == null || todoJSON.trim() == "") {
        return;
    }
    var todoArray = JSON.parse(todoJSON);
    if (todoArray.length == 0) {
        console.log("Error: the to-do list array is empty!");
        return;
    }
    for (var i = 0; i < todoArray.length; i++) {
        var todoItem = todoArray[i];
        todos.push(todoItem);
    }
    console.log("To-do array: ");
    console.log(todos);
}

function addTodosToPage() {
    var ul = document.getElementById("todoList");
    for (var i = 0; i < todos.length; i++) {
        var todoItem = todos[i];
        var li = document.createElement("li");
        li.innerHTML =
            todoItem.who + " needs to " + todoItem.task + " by " +
todoItem.dueDate;
        ul.appendChild(li);
    }
}
```

Save it. Then, open or refresh *todo.html* in your browser. You see the to-do items in your *todo.json* file displayed as list items in your web page, like this:




Try adding another to-do item to your *todo.json* file and reload the page. Do you see it?

So how did we add the array items to the page? Let's review:

***OBSERVE:***

```
if (this.responseText) {
    parseTodoItems(this.responseText);
    addTodosToPage();
}
```

In the `onreadystatechange` handler in `getTodoData()`, in addition to calling `parseTodoItems()`, we're calling `addTodosToPage()`. This gets called only after all the JSON has been processed and all the to-do items have been added to the `todos` array, so we know all the data will be in the array at this point.

This `addTodosToPage` function actually adds the items to the page by adding them to the "todoList" `<ul>`:

***OBSERVE:***

```
function addTodosToPage() {
    var ul = document.getElementById("todoList");
    for (var i = 0; i < todos.length; i++) {
        var todoItem = todos[i];
        var li = document.createElement("li");
```

```
        li.innerHTML =
            todoItem.who + " needs to " + todoItem.task + " by " +
todoItem.dueDate;
        ul.appendChild(li);
    }
}
```

In `addTodosToPage()`, first we get the `<ul>` element with the id "todoList" from the page, so we can add a new list item to it. Then we loop through all the objects in the `todos` array and create a temporary variable `todoItem` for each item in the array. Then we create a new `<li>` element for each item using the `document.createElement()` method, and set the HTML contents of that element to a string with data from the `todoItem` object. The `todoItem` is an *object* that was created when we parsed the JSON in the *todo.json* file using `JSON.parse()`, an object that looks something like this:

***OBSERVE:***

```
var todoObject = {
    task: "Get Milk",
    who: "Scott",
    dueDate: "today",
    done: false
};
```

You can access each property in the object as you normally would, using dot notation. The string we create to add to the list item uses the properties of the `todoItem`, like `todoItem.who`, `todoItem.task`, and `todoItem.dueDate`.

Once we've set the content of the `<li>` element, we can add it to the "todoList" `<ul>` element using the `appendChild()` method. As soon as you add the element to the page with `appendChild()`, the page updates to reflect the new element and you see your to-do list item.

Did you notice that we're not passing the `todos` array into the `addTodosToPage` function? That's because it's a global variable, and we don't need to—we can access it from any function. We made the `todos` array a global variable so we could access it in both the `parseTodoItems()` and `addTodosToPage()` functions.

In this lesson, you've brought together several things you already knew about— JSON, objects, arrays, using XMLHttpRequest to get data from a file, and updating the page with new elements—to create an Ajax web application that loads to-do items from a file, *todo.json*, and updates a page with those to-do items. All these parts work together in your web application to do something that's really kinda cool!

## *More about the material in this lesson*

The Riverside JS Workshop would like to acknowledge the generosity of O'Reilly Media, Inc. for making this material available to us through the Creative Commons License. We would also like to acknowledge the great work of Elisabeth Robson who authored this content. She is one of our favorites. Elisabeth has written a number of *Head First* programming books for web developers. We recommend them highly. You can browse her work here.