Riverside JS Workshop
JavaScript Skills Ladder, Lesson 2.5

Saving Data with Ajax

Lesson Objectives:

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

- *add a form to the HTML page that lets you enter new To-Do items and save them in the JSON file.*
- *style your forms.*
- *validate your forms.*
- *use Ajax to write data to a file.*

Adding and Saving a To-Do List Item

We're off to a great start with our To-Do List application. We've got a simple Ajax web application that reads data from a JSON file, and updates a page dynamically with that data.

In this lesson, we continue building the application. We'll add a form to the HTML page that lets you enter new To-Do items and save them in the JSON file, so that the next time you load the page, your new items will be there. We'll do some form validation too, which is always good practice (and will be good review of JavaScript techniques). You've used Ajax to *read* data from a file in a previous lesson. In this lesson, you'll see how to use Ajax to *write* data to a file (with a little help from a PHP script). So, let's get to it!

Add a Form to Your Page and Style It

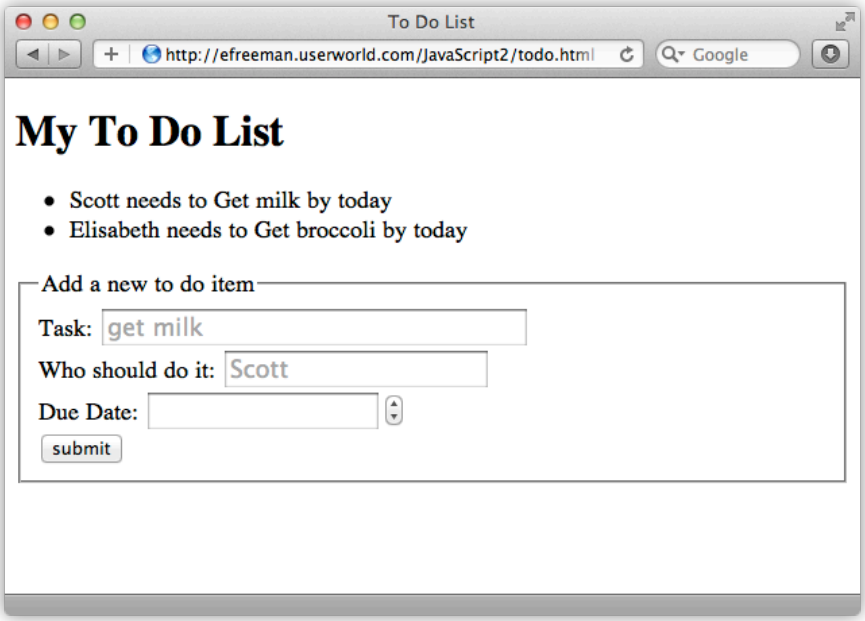
To add new items to our to-do list using a web page, we need a form. Add a form to *todo.html* as shown:

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>
  <link rel="stylesheet" href="todo.css">
</head>
<body>
  <h1>My To-Do List</h1>
  <ul id="todoList">
  </ul>

  <form>
  <fieldset>
    <legend>Add a new to-do item</legend>
    <div class="tableContainer">
      <div class="tableRow">
        <label for="task">Task: </label>
            <input type="text" id="task" size="35" placeholder="get milk">
      </div>
      <div class="tableRow">
        <label for="who">Who should do it: </label>
            <input type="text" id="who" placeholder="Scott">
      </div>
      <div class="tableRow">
        <label for="dueDate">Due Date: </label>
            <input type="date" id="dueDate">
      </div>
      <div class="tableRow">
        <label for="submit"></label>
        <input type="button" id="submit" value="submit">
      </div>
    </div>
  </fieldset>
  </form>
</body>
</html>
```

Save *todo.html*, and load it in your browser. You see the web page, which looks like this:



When you preview, the application loads the to-do items you already have in the *todo.json* file, using the code you added in *todo.js* in the previous lesson. (Your list may look a little different from mine if you've added or changed items.)

In this version of *todo.html*, we added a form with three data inputs and a submit button. Two of the data inputs are for text—the task, and who should do it—and the third is for a date. In many browsers, the date field will look like a text input, but in some browsers you may get a popup date picker.

We've added some extra `<div>` tags for styling the table. You'll see how this works in the next step.

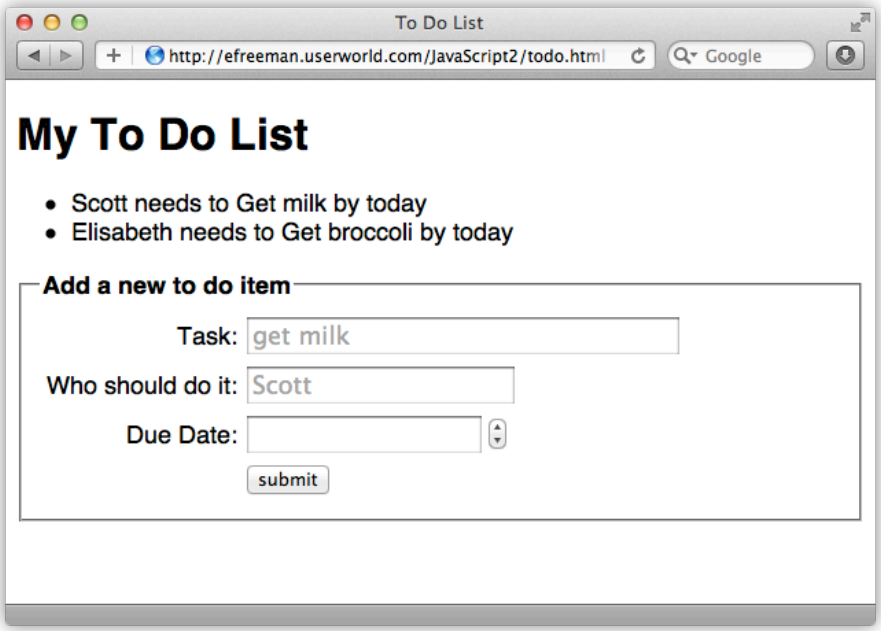
Style the Form

That HTML doesn't look too bad, but let's make it look even better with a little CSS. We removed the CSS we had previously, and added a link to a file. Go ahead and create a new file, and we'll add the extra CSS we need to style the form.

CODE TO TYPE:

```
body {
    font-family: Helvetica, Arial, sans-serif;
}
legend {
    font-weight: bold;
}
div.tableContainer {
    display: table;
    border-spacing: 5px;
}
div.tableRow {
    display: table-row;
}
div.tableRow label {
    display: table-cell;
    text-align: right;
}
div.tableRow input {
    display: table-cell;
}
```

Save this file as *todo.css* in your work folder. Now open or refresh *todo.html* in your browser. You should be able to see the results of the styles you just created in *todo.css*.



The fields are aligned properly in this form which gives it a more professional appearance than the previous one.

This CSS uses the *table display* properties to lay out the `<div>` tags and form controls, similar to the way an HTML table is laid out. If you're not familiar with table display, don't worry about it. It's just a way of laying out a page. In fact, CSS table display properties are the same properties that are used by default in the browser to lay out HTML tables. Using table display works really well for forms, because forms often have a regular, table-like structure to them.

Now that you've got the web page ready to go for entering new to-do items, it's time to update your JavaScript so that you can do something with the items you enter.

Process the Form with JavaScript

You can enter data in the form in the web page you created in the previous step, but nothing happens when you click submit, because we have no action in the `<form>` element, and no JavaScript to capture a click on the submit button. So, as it is, the form does nothing.

In this next step, we need to get the data from the form and check it. We want to get the form data using our own JavaScript code so we can validate the data and then do something with it. So, we'll need a way to determine when the submit button is clicked so we can get the data that was entered into the form. We'll add a *click handler* function to the Submit button.

Update *todo.js* as shown:

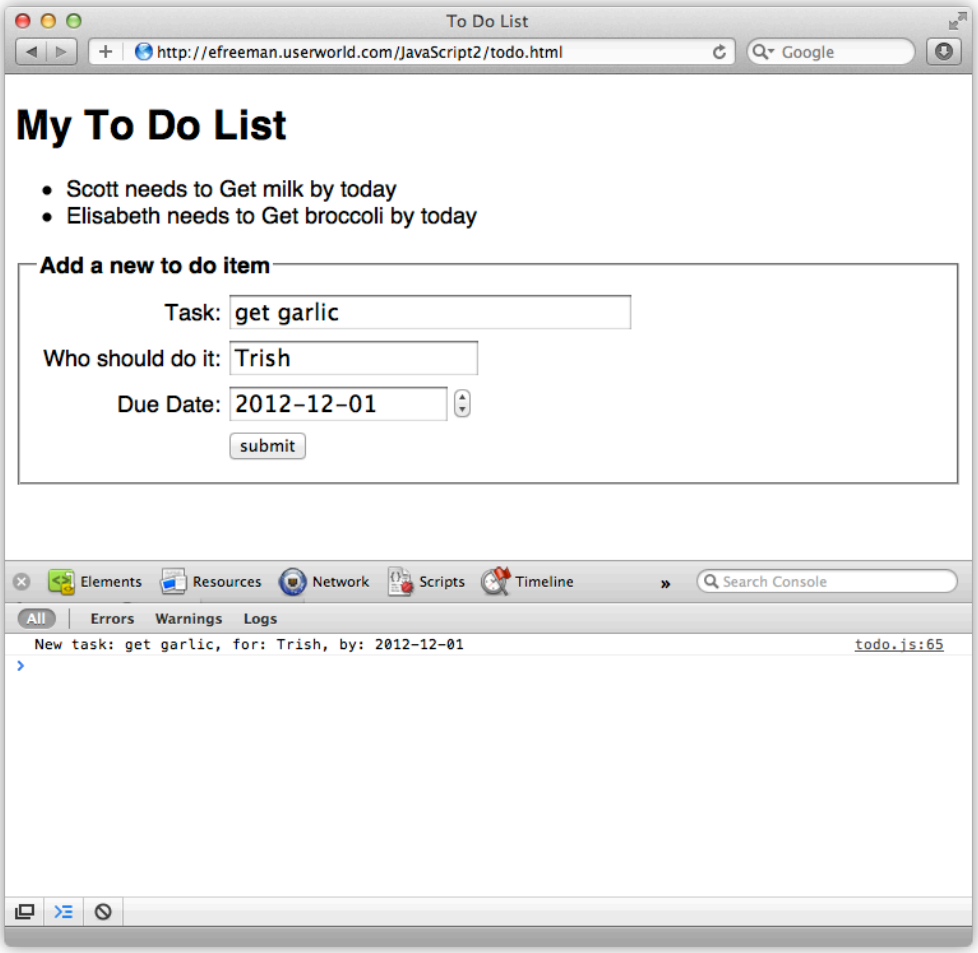
CODE TO TYPE:

```
var todos = new Array();

window.onload = init;

function init() {
    var submitButton = document.getElementById("submit");
    submitButton.onclick = getFormData;
    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);
    }
}

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);
    }
}

function getFormData() {
    var task = document.getElementById("task").value;
    if (checkInputText(task, "Please enter a task")) return;

    var who = document.getElementById("who").value;
    if (checkInputText(who, "Please enter a person to do the task")) return;

    var date = document.getElementById("dueDate").value;
    if (checkInputText(date, "Please enter a due date")) return;

    console.log("New task: " + task + ", for: " + who + ", by: " + date);
}

function checkInputText(value, msg) {
    if (value == null || value == "") {
        alert(msg);
        return true;
    }
    return false;
}
```

Save the file, and open or refresh *todo.html* in your browser. Open the developer console so you can see the console message we're using to display the data. Then, type some data into

the form. After entering the data and clicking submit, you see the console message showing the data you entered. It looks like this:



We get the data from the form with the function `getFormData()`, which we set up as the click handler for the submit button:

OBSERVE:

```
var submitButton = document.getElementById("submit");

submitButton.onclick = getFormData;
```

In `getFormData()`, we check to make sure you enter values for the various fields:

OBSERVE:

```
function getFormData() {
    var task = document.getElementById("task").value;
    if (checkInputText(task, "Please enter a task")) return;

    var who = document.getElementById("who").value;
    if (checkInputText(who, "Please enter a person to do the task"))
return;

    var date = document.getElementById("dueDate").value;
    if (checkInputText(date, "Please enter a due date")) return;

    console.log("New task: " + task + ", for: " + who + ", by: " +
date);
}
```

We get the value of each field in the form using `document.getElementById()` to first get the form control element, and then using the element's `value()` method to get the value of the control. In our form, each control will be a string. If any of the fields is empty, the function just returns, without displaying anything in the console. Notice that we're using a helper function, `checkInputText()` to make sure that each form control has a value:

OBSERVE:

```
function checkInputText(value, msg) {
    if (value == null || value == "") {
        alert(msg);
        return true;
    }
    return false;
}
```

`checkInputText()` has two parameters: `value` and `msg`. The value is the string data that you enter into the form control, and the msg is the text to display in the alert if you didn't enter anything for that control. We check to make sure value isn't empty. If it is, we display the msg and return true. If it is not empty, we return false. Look back at `getFormData()` and you'll see that if we return false from `checkInputText()` for each form field, then we display the console message showing the data that you entered into the form.

Create a New To-Do Object

It's good to see the form data we enter in the console, but we really want to see it in the page. We also need to add the new to-do item to our todos array, which means we need to create a to-do object. You probably remember from the previous lesson that when we read in the to-dos from the file *todo.json*, each to-do item stored in the todos array is an object.

How can we create a to-do object for the data that you've entered in the form? We need a to-do object constructor. Do you remember how to create a constructor function? Modify *todo.js* as shown:

CODE TO TYPE:

```
function Todo(task, who, dueDate) {
    this.task = task;
    this.who = who;
    this.dueDate = dueDate;
    this.done = false;
}

var todos = new Array();

window.onload = init;

function init() {
    var submitButton = document.getElementById("submit");
    submitButton.onclick = getFormData;
    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);
    }
}

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);
    }
}

function getFormData() {
    var task = document.getElementById("task").value;
    if (checkInputText(task, "Please enter a task")) return;

    var who = document.getElementById("who").value;
    if (checkInputText(who, "Please enter a person to do the task"))
return;

    var date = document.getElementById("dueDate").value;
    if (checkInputText(date, "Please enter a due date")) return;

    console.log("New task: " + task + ", for: " + who + ", by: " +
date);
    var todoItem = new Todo(task, who, date);
    todos.push(todoItem);
}

function checkInputText(value, msg) {
    if (value == null || value == "") {
        alert(msg);
        return true;
    }
    return false;
}
```

Here, we use the `Todo()` constructor function to create a new `Todo` object, and push that object onto the end of the todos array. The `Todo()` constructor takes three arguments: `task`, `who`, and `dueDate`. The constructor function sets the object properties to the values that are passed into it. In `getFormData()`, we pass in the values that we got from the form for those arguments. In addition, the `Todo()` constructor sets the `done` property to false. Can you guess why?

We'll go back to the `done` property in a later lesson, but for now assume that all the to-do items are still to be done!

Adding the New To-Do Item to the Page

Now we'll add the to-do item to the page so we can see the result of all of our hard work in the web page, rather than just in the console. Modify *todo.js* as shown:

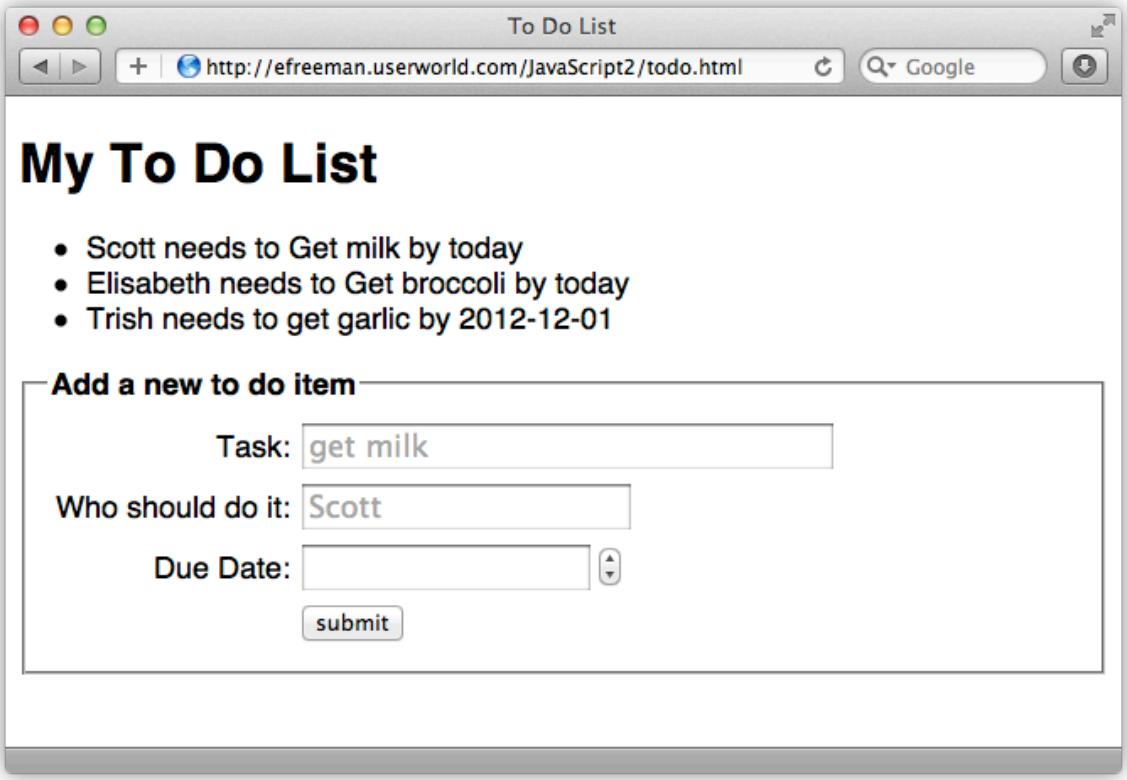
CODE TO TYPE:

```
function Todo(task, who, dueDate) {
    this.task = task;
    this.who = who;
    this.dueDate = dueDate;
    this.done = false;
}

var todos = new Array();

window.onload = init;

function init() {
    var submitButton = document.getElementById("submit");
    submitButton.onclick = getFormData;
    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);
    }
}

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);
    }
}

function getFormData() {
    var task = document.getElementById("task").value;
    if (checkInputText(task, "Please enter a task")) return;

    var who = document.getElementById("who").value;
    if (checkInputText(who, "Please enter a person to do the task"))
return;

    var date = document.getElementById("dueDate").value;
    if (checkInputText(date, "Please enter a due date")) return;

    console.log("New task: " + task + ", for: " + who + ", by: " +
date);
    var todoItem = new Todo(task, who, date);
    todos.push(todoItem);
    addTodoToPage(todoItem);
}

function checkInputText(value, msg) {
    if (value == null || value == "") {
        alert(msg);
```

```
        return true;
    }
    return false;
}

function addTodoToPage(todoItem) {
    var ul = document.getElementById("todoList");
    var li = document.createElement("li");
    li.innerHTML =
        todoItem.who + " needs to " + todoItem.task + " by " +
todoItem.dueDate;
    ul.appendChild(li);
    document.forms[0].reset();
}
```

Save the changes to *todo.js*, and open or refresh *todo.html* in your browser. Enter the information for a to-do item in the form and click submit. Your new item should appear on the page:



To add the to-do item to the page, we use a new function, `addTodoToPage()`, and call it just after we add the new Todo object to the todos array in the function `getFormData()`.

OBSERVE:

```
function addTodoToPage(todoItem) {
    var ul = document.getElementById("todoList");
    var li = document.createElement("li");
    li.innerHTML =
        todoItem.who + " needs to " + todoItem.task + " by " +
todoItem.dueDate;
    ul.appendChild(li);
    document.forms[0].reset();
}
```

The Todo object that's passed into `addTodoToPage()` gets the parameter name `todoItem`. To add it to the page, first we get the "todoList" `<ul>`. Then we create a new `<li>` element to add to this list, and set the `innerHTML` of the `<li>` to the information in the todoItem object. Then we add the new list item to the "todoList" list. Finally, we reset the form to make it easier to create another to-do item using the form.

Don't mix up the two functions: `addTodoToPage()`and `addTodosToPage()`. The function, `addTodoToPage()`, which we just created, adds a single to-do item to the page. The function, `addTodosToPage()`, which we created in an earlier lesson, takes all the to-do items in the JSON file and adds them to the page when the page first loads. The two functions look similar (and have similar names), but they have slightly different purposes. Can you see why we need both? Can you think of how we might make our code more efficient by combining some of the functionality of the two functions? We'll come back to these and other pressing questions in the next lesson!

Try adding a couple more to-do items to your page. Now, reload the page. What happens? Hmm. That's interesting. The items disappear.

Saving the Data with Ajax

Okay, we've got the new to-do item to display in the to-do list on the page, but how do we save it in the *todo.json* file so that it doesn't disappear when you reload the page?

We can use Ajax to send the data in the todos array (where all of our todos are stored) to a PHP script stored on a server (the same server where your *todo.json* file is stored. This PHP script will write the to-do items into the *todo.json* file.

You might be wondering, "Why can't we write the to-do items directly to the file using JavaScript?" Currently, JavaScript does not allow you to read from or write to files directly on a file system, for security purposes. (Imagine if you went to a malicious website that could read or write from your hard drive. Bad news!) That's why we need the help of a server script.

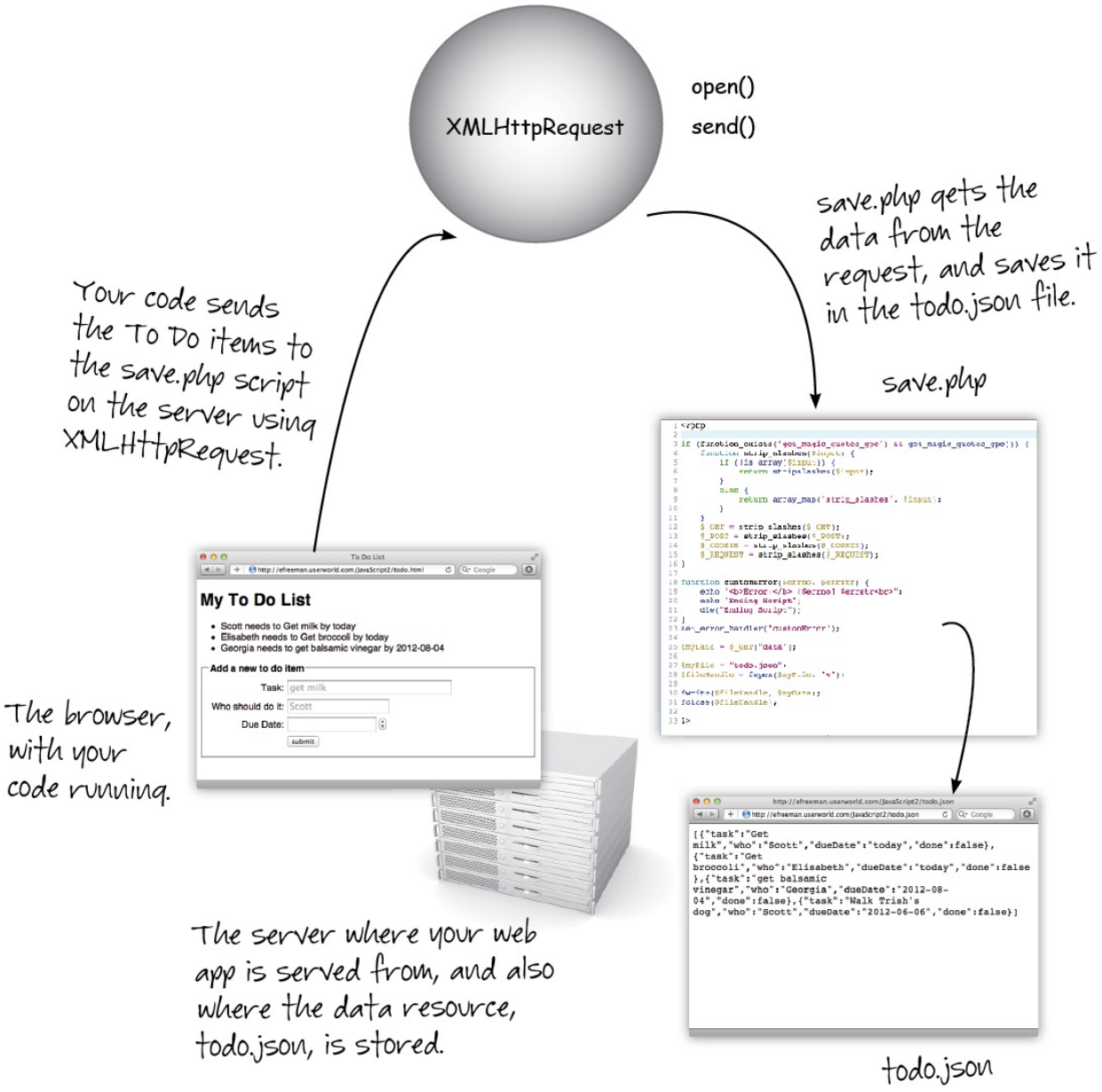
Note Work is underway on a file system library for JavaScript that will allow you to read and write files on your local file system (your own computer, not the file system on the server!), but this is a new project still in development as of this writing.

So you may think that we read directly from the *todo.json* file. Actually we don't. We use an *HTTP request* to get the data, and the web server reads the data from the file and returns it to the browser, which places the data in the *XMLHttpRequest* object, and our JavaScript code gets it from there. Whew!

We need to do something similar, only in reverse to write the data. We'll use the XMLHttpRequest object to send some data back to the web server. But we need a program on the web server that knows what to do with the data. We can't just randomly send data to the web server. We need to send it to a specific program—the server script—that knows how to update your *todo.json* file.

It's really similar to what happens when you submit a form that has a server specified in the *action* attribute, but instead of using the submit action with the form, we'll send the data to the server using the XMLHttpRequest object.

Here's how it works:



Creating the PHP Server Script

First, we need to create the script we're using to save the to-do items we send it in the *todo.json* file. Create a .php file in the work directory where your to-do app is saved. Name this file *save.php*, and enter the code below.

CODE TO TYPE:

```
<?php

if (function_exists('get_magic_quotes_gpc') && get_magic_quotes_gpc()) {
    function strip_slashes($input) {
        if (!is_array($input)) {
            return stripslashes($input);
        }
        else {
            return array_map('strip_slashes', $input);
        }
    }
    $_GET = strip_slashes($_GET);
    $_POST = strip_slashes($_POST);
    $_COOKIE = strip_slashes($_COOKIE);
    $_REQUEST = strip_slashes($_REQUEST);
}

function customError($errno, $errstr) {
    echo "<b>Error:</b> [$errno] $errstr<br>";
    echo "Ending Script";
    die("Ending Script");
}
set_error_handler("customError");

$myData = $_GET["data"];

$myFile = "todo.json";
$fileHandle = fopen($myFile, "w");

fwrite($fileHandle, $myData);
fclose($fileHandle);

?>
```

Save the changes to your *save.php* file. Don't worry if the code seems foreign. To learn more about PHP, you will find dozens of great resources online.

Anyway, since you know JavaScript, you can probably read through this PHP script and understand some of it. The first part of the file is responsible for stripping those extra slashes that get added to your string when you send a string that contains quotation marks. The next part is an error handling function (`customError()`), and below that is where the script actually gets the data from the request and writes it to the file. Again, don't worry about these

details. Just be aware that this script writes all the to-do items you send it to the file *todo.json*, overwriting what was there before. This will be important in just a moment when we consider what data to send the script from our JavaScript.

Adding the JavaScript

Now that you have the PHP script in place that is ready to accept requests from your JavaScript, it's time to update the JavaScript to save your to-do items!

CODE TO TYPE:

```
function Todo(task, who, dueDate) {
    this.task = task;
    this.who = who;
    this.dueDate = dueDate;
    this.done = false;
}

var todos = new Array();

window.onload = init;

function init() {
    var submitButton = document.getElementById("submit");
    submitButton.onclick = getFormData;
    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);
    }
}

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);
    }
}

function getFormData() {
    var task = document.getElementById("task").value;
    if (checkInputText(task, "Please enter a task")) return;

    var who = document.getElementById("who").value;
    if (checkInputText(who, "Please enter a person to do the task")) return;

    var date = document.getElementById("dueDate").value;
    if (checkInputText(date, "Please enter a due date")) return;

    console.log("New task: " + task + ", for: " + who + ", by: " + date);
    var todoItem = new Todo(task, who, date);
    todos.push(todoItem);
    addTodoToPage(todoItem);
    saveTodoData();
}

function checkInputText(value, msg) {
    if (value == null || value == "") {
        alert(msg);
        return true;
    }
    return false;
}
function addTodoToPage(todoItem) {
    var ul = document.getElementById("todoList");
    var li = document.createElement("li");
    li.innerHTML =
        todoItem.who + " needs to " + todoItem.task + " by " +
todoItem.dueDate;
    ul.appendChild(li);
    document.forms[0].reset();
}

function saveTodoData() {
    var todoJSON = JSON.stringify(todos);
    var request = new XMLHttpRequest();
    var URL = "save.php?data=" + encodeURI(todoJSON);
    request.open("GET", URL);
```

```
    request.setRequestHeader("Content-Type",
                             "text/plain;charset=UTF-8");
    request.send();
}
```

Save the changes to *todo.js*, and open or refresh your *todo.html* file in the browser. Add one or two to-do items. Now, reload the page. They should still be there. Open your *todo.json* file. You should see all your to-do items there. (Make sure you close it again before you add any more to-do items).

Let's go over how this works, step-by-step. We added a new function `saveTodoData()` where all the action happens. We call this function from `getFormData()`, after we add a `todoItem` to the page. Let's see what `saveTodoData()` does:

OBSERVE:

```
function saveTodoData() {
    var todoJSON = JSON.stringify(todos);
    var request = new XMLHttpRequest();
    var URL = "save.php?data=" + encodeURI(todoJSON);
    request.open("GET", URL);
    request.setRequestHeader("Content-Type",
                             "text/plain;charset=UTF-8");
    request.send();
}
```

We don't pass any arguments to `saveTodoData()`. Why? Because `saveTodoData()` uses the todos array, which is a global variable, so we don't need to pass it. First, `saveTodoData()` converts the todos array to a JSON string using `JSON.stringify()`. The `JSON.stringify()` method takes an object or an array and turns it into a string that we can then send as data to a script and save in a file. We save that JSON string in the variable `todoJSON`. We put *every* to-do item in the todos array in the JSON string because the *save.php* script overwrites the entire *todo.json* file each time we call it. You'll see more on this in a moment.

Next, we create a new XMLHttpRequest object, and save it in the variable request. Just like when we used XMLHttpRequest to send a request to get data from a server, we will use XMLHttpRequest to send a request to send data to a server. We send that data along with the request. There are two ways to send data to a server script using Ajax: **GET** and **POST**. These are the same GET and POST you can use with forms to send data to a server. Let's take a quick look at each of these methods of sending data.

GET

When you use GET to send a request with data to a server, you create a URL that looks very similar to the URLs you typically use, except you add some data on the end of the URL. You append the data you want to send on the end of the URL, with a "?" between the URL of the script and the data. This is the method we use in `saveTodoData()`. We append the to-do items to the end of the URL for the request to *save.php*. But we can't just send the data as is, because it contains all kinds of characters that would mess up the URL and confuse the browser and the server. So, we have to encode the URL first so all the characters that would confuse the browser and the server get replaced with their encoded versions. For instance, the encoded version of a space (" ") is %20, and the encoded version of a double quote (") is %22.

In `saveTodoData()`, we create the URL for the request like this:

OBSERVE:

```
var URL = "save.php?data=" + encodeURI(todoJSON);
```

In the first part of our URL, we specify the name of the script we want to send the request to: save.php.

This URL is a *relative URL*, which means we don't specify the full http://... URL. Rather, we just specify the name of the script, *save.php*, because we know it's in the same directory from which the web application is loaded. This request will be converted automatically to a request for the full URL for us, which is pretty convenient. You could use the full URL (known as the *absolute* URL) if you wanted, but that's not required.

After the name of the script, we add **?** to separate the path to the script from the data we're sending to the server.

Next, we type data=, which gives a name to the data we're sending. This is just like a variable name, and it allows the PHP script to use that name to GET the data from the URL. In the PHP script, we had the line:

OBSERVE:

```
$myData = $_GET["data"];
```

The result of this PHP code is that it can access the data in the URL and store it in the variable `$myData`, which it then uses to write the data to the file, *todo.json*. Finally, we append the data in the todoJSON string to the end of the URL variable. Remember that `todoJSON` is comprised of all the to-do items in our todos array, converted into JSON.

We encode the `todoJSON` string using the built-in JavaScript function, `encodeURI()`, which takes a string and turns it into a form suitable for sending as part of a URL with a GET request. So the actual URL that we use in the XMLHttpRequest looks something like this:

OBSERVE:

```
save.php?data=%5B%7B%22task%22:%22get%20milk%22,%22who%22:%22Scott%22,
%22dueDate%22:%22today%22,%22done%22:false%7D,%7B%22task%22:%22get%20b
roccoli%22,%22who%22:%22Elisabeth%22,%22dueDate%22:%22today%22,%22done
%22:false%7D,%7B%22task%22:%22get%20balsamic%20vinegar%22,%22who%22:%2
2Georgia%22,%22dueDate%22:%222012-08-04%22,%22done%22:false%7D%5D
```

POST

If we want to use POST instead of GET to send the data to the server we can (although in this particular example, we'd have to change the code in the PHP script too, so it won't work). In general, in order to use POST, instead of putting the data in the URL itself, we send it using the request object. We use a similar format for the data (an encoded string), but rather than adding it to the URL string, we send it when we call `request.send()`.

Sending the Request

Okay, we've created a URL that contains the name of the script to which we're sending the request, *save.php*, and we've got the data we're sending—the todos array converted into JSON—encoded and added to the URL, so how do we send the request?

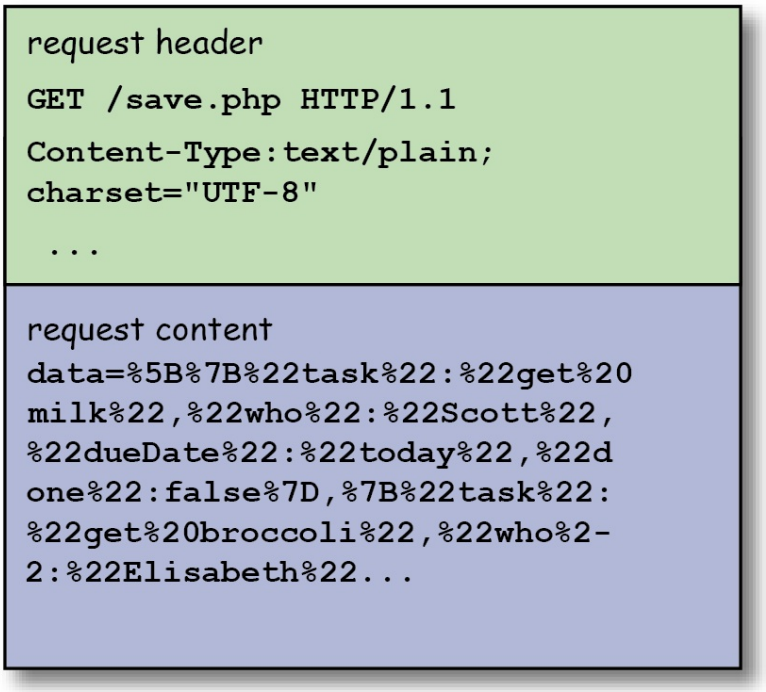
OBSERVE:

```
function saveTodoData() {
    var todoJSON = JSON.stringify(todos);
    var request = new XMLHttpRequest();
    var URL = "save.php?data=" + encodeURI(todoJSON);
    request.open("GET", URL);
    request.setRequestHeader("Content-Type",
                             "text/plain;charset=UTF-8");
    request.send();
}
```

To send the request, we use the `request.open()` method to tell the XMLHttpRequest object which URL we want to send the request to, and the kind of request it is (in this case, it's a GET request). Unlike the request we sent to *retrieve* the data in the *todo.json* file, we don't have to set up an `onreadystatechange` handler now, because we're not expecting any data back from the server (although in some situations, you might expect some data back; it's possible to both send and receive data in the same request).

So we're almost ready to send the request, but first we need to tell the server the kind of data we're sending. There are many kinds of data we could send in a request: XML, HTML, plain text, binary data (like if we're sending an image), and so on. In our case, we're just sending plain text: a JSON string.

Any request sent from a browser to a server contains a lot of information. A request always has a *header* that includes information like the kind of request we're making (called the request "method," in our case, this is GET), information about the encoding, information about the browser sending the request, and the URL to which we're sending the request.



To tell the server the type of data we're sending with the request, we set one of the fields in the header named *Content-Type*, and provide the correct value for plain text. This value consists of a *MIME* type, "text/plain" (a string that represents the type of the file that conforms to the MIME standard of file types), and a "charset" that tells the browser the character encoding of the data. This encoding is usually "UTF-8" which is the current standard for encoding characters. UTF-8, among other capabilities, it can encompass all the languages in the world.

So now we're ready to send! We call `request.send()` to send the request to the server, which sends all the data we want to save in *todo.json* along with the request.

When we send the request, the *save.php* script is called on the server, which saves the to-do data into the file *todo.json*. If you recall, we mentioned that the script overwrites anything that's already in this file. That's the reason we send *all* of the to-do items each time we make the

request. It's not the most efficient way to do it, but it's a bit less complex and a good place for us to start. Can you think of a better way to do it?

You've updated the application so that you can add new items to your to-do list, *and* saved those items back to the *todo.json* file. You can see how Ajax is a powerful way to increase the functionality and flexibility of your web applications. The To-Do List application is much more useful with the ability to save your to-do items, don't you think?

However, our solution isn't perfect by any means. What happens if more than one person uses this web application? For instance, if you give the URL of the web application to a friend, will they be able to see your to-do items? Will they be able to add items to *your* to-do list?

Creating a to-do list application that can support more than one user is beyond the scope of this phase of the Skills Ladder, but you now have a solid understanding of what Ajax is, and how Ajax works.

So now you know that Ajax is a set of techniques that allow you to create what we call "single-page web applications"—web applications that you can use much like a regular desktop application. With Ajax techniques, you can update a web page dynamically with new data, you can change the page when the user interacts with it, and you can retrieve and save data the user enters into the page, all without having to "load" another web page.

In the next lesson we'll look at how to make JavaScript applications better by adding some efficiency to the To-Do List application.

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.