Beginning JavaScript®, Fifth Edition
By Jeremy McPeak and Paul Wilton

12

JSON

WHAT YOU WILL LEARN IN THIS CHAPTER:

- Discovering the limitations of using XML with JavaScript
- Recognizing the differences between JavaScript and JSON
- Serializing objects using the built-in JSON object
- Parsing JSON back into actual objects and values you can use in your pages

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

You can find the wrox.com code downloads for this chapter at `http://www.wiley.com/go/BeginningJavaScript5E` on the Download Code tab. You can also view all of the examples and related files at `http://beginningjs.com`.

If you aren't already, start thinking of a web page as a program. It does, after all, have all the trappings of a traditional program. It has a user interface, and it can process data with JavaScript. But as you well know, traditional programs can do more; they can store data as well as transmit data to other computers and systems. In the coming chapters, you learn that you can do the same things in a web page—all thanks to JavaScript.

But as you soon learn, you can't just store objects and arrays as they are; instead, you need to *serialize* them. Serialization is the process of translating an object into a string representation of that object. Once an object is serialized, the string representation of that object can then be stored in a more permanent storage facility or transmitted to another computer.

Serialization translates only the structure and pertinent information of an object—that is, only the properties are present in a serialized object. But once you need to work with the object within JavaScript, you can *deserialize* it, converting it back into a native JavaScript object.

The serialization format that web developers overwhelmingly embrace is called JavaScript Object Notation, or JSON (pronounced like the name: Jason). It is a subset of the JavaScript language; as such, it's easy to read, it's concise, and most importantly, it's easy to serialize to and deserialize from.

But the web hasn't always used JSON for serializing JavaScript objects. So before we look at the JSON format, let's look at what web developers used to use.

XML

There was a time when the web development community embraced XML for just about everything. Web services used it to communicate with one another and other computers, and JavaScript developers used it to communicate with the web application's server.

XML is a human-readable language thanks to its declarative syntax. It's not necessary for humans to read XML data, but being able to read and decipher the XML can be useful. Consider the following XML document as an example:

```
<person>
    <firstName>John</firstName>
    <lastName>Doe</lastName>
    <age>30</age>
</person>
```

Despite being a simple document, you know that this XML represents an individual person named John Doe who is 30 years old. You could catch and fix errors that may occur in your application as it generates the preceding XML-formatted data.

XML is also machine-readable, and it was a known commodity when developers started using it. Every modern programming language had the tools and capabilities for reading, parsing, and creating XML-formatted data, and so using XML to communicate between computer systems and applications seemed like a good idea.

But XML has its drawbacks. For one, XML's declarative syntax adds a lot of extra cruft to the data. Look again at the XML describing a person named John Doe:

```
<person>
    <firstName>John</firstName>
    <lastName>Doe</lastName>
    <age>30</age>
</person>
```

This simple XML is 101 bytes. That's not large by today's standards, but remember that this is just an example. This is information that a computer would send over the Internet to another computer.

First, the opening and closing `<person>` tags surround the actual data. Of course, the outer `<person/>` element exists for organizational purposes, but it is 17 bytes—16 percent of the entire payload. Other formats will still have some way to organize the document's real information (the first and last name), but they would be smaller in size.

Next, opening and closing tags surround both the first and last name. Naturally, there needs to be some way to organize that data, but yet, this XML uses 55 bytes to denote the first name, last name, and the age.

Another of XML's issues is the code necessary for reading, parsing, and generating XML data. Yes, most modern programming languages can handle XML, but it requires a lot of code—code that usually has to be rewritten for specific XML formats. For example, the following code is one way you could read the previous XML and parse it into an object called `person`:

```
var personElement = document.querySelector("person");
var firstName = personElement.querySelector("firstName").innerHTML;
var lastName = personElement.querySelector("lastName").innerHTML;
var age = personElement.querySelector("age").innerHTML;

var person = {
    firstName : firstName,
    lastName: lastName,
    age: age
};
```

This code demonstrates a straightforward approach to parsing the John Doe XML. It first retrieves the `<person/>` element using `document.querySelector()`. It then retrieves the `<firstName/>`, `<lastName/>`, and `<age/>` elements and stores their respective contents in the `firstName`, `lastName`, and `age` variables. Finally, it creates the `person` object and assigns the appropriate data to its properties. This code isn't complex, but as you might suspect, it wouldn't work for XML-formatted data with different element names and structures. Naturally, documents with more complex structures require much more code.

But parsing XML into a JavaScript object is only half of the story. Before you can send data from JavaScript to the server, you have to serialize the JavaScript object. Serializing a JavaScript object to XML-formatted data is not a trivial task. Like parsing, the same code usually doesn't work for different data structures. Plus, developers must ensure their generated XML data is well-formed.

Around 2007 and 2008, the web community thankfully adopted a different data format for storing and transmitting JavaScript data.

JSON

In 2006, Douglas Crockford wrote the JavaScript Object Notation specification. JSON is a subset of the JavaScript language, and it uses several of JavaScript's syntactical patterns for organizing and structuring data. As such, it is does a very good job of representing objects and their data (it's so good that other languages use JSON, too). It's extremely easy to parse JSON into JavaScript objects and to serialize objects into JSON. In today's modern browsers, it only takes one line of code!

As you soon see, JSON looks a lot like JavaScript's object and array literals. It's easy to confuse JSON and JavaScript as being the same thing, but it's important to understand the difference between the two. JavaScript is a programming language; JSON is a data format.

JSON lets you represent three types of data: simple values, objects, and arrays.

Simple Values

You can represent simple values like strings, numbers, booleans, and `null`. For example, the following line is valid JSON:

```
"JavaScript"
```

This JSON represents the string value of `"JavaScript"`, and it looks exactly like a normal JavaScript string. But there's a big difference between strings in JavaScript and JSON; JSON strings must use double quotes. Thus, the following is invalid JSON:

```
'JavaScript'
```

Numeric data is represented by what appears to be number literals, like this:

```
10
```

This is valid JSON representing the number 10. Similarly, boolean values and `null` look like JavaScript literals, too:

```
true

null
```

Objects

Objects in JSON are represented with what looks like JavaScript's object literal notation. For example, the following is a JavaScript object that represents the same person from earlier:

```
var person = {
    firstName: "John",
    lastName: "Doe",
    age: 30
};
```

The JSON representation of this object looks similar. Here is the same object represented in JSON:

```
{
    "firstName": "John",
    "lastName": "Doe",
    "age": 30
}
```

A few noticeable differences exist between the JavaScript and JSON representations of this object. First, JSON doesn't have the `person` variable name. Remember that JSON is a data format, not a language. It has no variables, functions, or methods. It simply defines the structure and data of an object.

The second difference is the object's property names. Notice that they are surrounded by double quotes. In JSON, an object's property names are strings, and the values of those properties follow the rules specified in the previous section. Double quotes surround the string values of `"John"` and `"Doe"`, and the number `30` appears as a literal value.

The final difference is the lack of a trailing semicolon after the closing curly brace. This isn't a JavaScript statement, and thus, the semicolon is not needed.

The size of this JSON data structure is 69 bytes. That's 68 percent of the 101 bytes of the equivalent XML.

Like JavaScript objects, JSON objects can be simple or complex. The data structure representing John Doe is rather simple, but you can easily add complexity by incorporating his address:

```
{
    "firstName": "John",
    "lastName": "Doe",
    "age": 30,
    "address": {
        "numberAndStreet": "123 Someplace",
        "city": "Somewhere",
        "state": "Elsewhere"
    }
}
```

This adds an `address` property to the main object, and its value is another object that contains John's mailing address.

Arrays

Like objects, arrays in JSON are similar to JavaScript's array literal notation. The following line of code is an array literal in JavaScript:

```
var values = ["John", 30, false, null];
```

The same array looks like this in JSON:

```
["John", 30, false, null]
```

Again, notice the JSON array does not have the `values` variable, nor does it have the trailing semicolon. And like objects, arrays are not limited to just simple values; they can contain complex objects, too:

```
[
    {
        "firstName": "John",
        "lastName": "Doe",
        "age": 30,
        "address": {
            "numberAndStreet": "123 Someplace",
            "city": "Somewhere",
            "state": "Elsewhere"
        }
    },
    {
        "firstName": "Jane",
        "lastName": "Doe",
```

```
        "age": 28,
        "address": {
            "numberAndStreet": "246 Someplace",
            "city": "Somewhere",
            "state": "Elsewhere"
        }
    }
]
```

This JSON array contains multiple objects that represent people and their addresses. The first is our familiar John Doe, and the second is his little sister, Jane, who lives down the street. JSON data structures can be as simple or complex as you need them to be.

Serializing Into JSON

It's extremely easy to serialize JavaScript objects into JSON. JavaScript has an aptly named `JSON` object that you use to parse JSON data and serialize JavaScript objects. All major browsers support this `JSON` object. Older browsers, such as IE7 and below, can use Crockford's JSON implementation (`https://github.com/douglascrockford/JSON-js`) to achieve the same results.

To serialize a JavaScript object into JSON, you use the `JSON` object's `stringify()` method. It accepts any value, object, or array and serializes it into JSON. For example:

```
var person = {
    firstName: "John",
    lastName: "Doe",
    age: 30
};

var json = JSON.stringify(person);
```

This code serializes the `person` object with `JSON.stringify()` and stores it in the `json` variable. The resulting JSON-formatted data looks like this:

```
{"firstName":"John","lastName":"Doe","age":30}
```

All unnecessary whitespace is removed, giving you an optimized payload that you can then send to the web server or store elsewhere.

Parsing JSON

Parsing JSON into JavaScript objects is equally simple. The `JSON` object has a `parse()` method that parses the JSON and returns the resulting object. Using the `json` variable from the previous code:

```
var johnDoe = JSON.parse(json);
```

This code parses the JSON text contained in `json` and stores the resulting object in the `johnDoe` variable. And here's the wonderful thing—you can immediately use `johnDoe` and access its properties, such as:

```
var fullName = johnDoe.firstName + " " + johnDoe.lastName;
```

It's really no wonder why developers embraced JSON. It's easy to work with!

JSON is useful when you need to store an object, but the API you're working with only lets you store text. In Chapter 10, you learned that the native drag and drop API has a `dataTransfer` object that you can use to work with data during the drag-and-drop operation. But as you learned, it doesn't let you store objects, but you can store text. JSON is text, so you can serialize a JavaScript object at the beginning of the drag operation and parse the JSON when the drop event fires.

TRY IT OUT Using JSON in Drag and Drop

This example uses `ch10_example21.html` as a basis. Feel free to copy and paste the code from that example and make the highlighted modifications. Otherwise, open your text editor and type the following:

```
<!DOCTYPE html>

<html lang="en">
<head>
    <title>Chapter 12: Example 1</title>
    <style>
        [data-drop-target] {
            height: 400px;
            width: 200px;
            margin: 2px;
            background-color: gainsboro;
            float: left;
        }

        .drag-enter {
            border: 2px dashed #000;
        }

        .box {
            width: 200px;
            height: 200px;
        }

        .navy {
            background-color: navy;
        }

        .red {
            background-color: red;
        }
    </style>
</head>
<body>
    <div data-drop-target="true">
        <div id="box1" draggable="true" class="box navy"></div>
        <div id="box2" draggable="true" class="box red"></div>
    </div>
```

```
<div data-drop-target="true"></div>

<script>
    function handleDragStart(e) {
        var data = {
            elementId: this.id,
            message: "You moved an element!"
        };

        e.dataTransfer.setData("text", JSON.stringify(data));
    }

    function handleDragEnterLeave(e) {
        if (e.type == "dragenter") {
            this.className = "drag-enter";
        } else {
            this.className = "";
        }
    }

    function handleOverDrop(e) {
        e.preventDefault();

        if (e.type != "drop") {
            return;
        }

        var json = e.dataTransfer.getData("text");
        var data = JSON.parse(json);

        var draggedEl = document.getElementById(data.elementId);

        if (draggedEl.parentNode == this) {
            this.className = "";
            return;
        }

        draggedEl.parentNode.removeChild(draggedEl);

        this.appendChild(draggedEl);
        this.className = "";

        alert(data.message);
    }

    var draggable = document.querySelectorAll("[draggable]");
    var targets = document.querySelectorAll("[data-drop-target]");

    for (var i = 0; i < draggable.length; i++) {
        draggable[i].addEventListener("dragstart", handleDragStart);
    }

    for (i = 0; i < targets.length; i++) {
        targets[i].addEventListener("dragover", handleOverDrop);
        targets[i].addEventListener("drop", handleOverDrop);
```

```
            targets[i].addEventListener("dragenter", handleDragEnterLeave);
            targets[i].addEventListener("dragleave", handleDragEnterLeave);
        }
    </script>
</body>
</html>
```

Save this file as `ch12_example1.html`.

You need just a few changes to make this example different from `ch10_example21.html`. The first is in the `handleDragStart()` function:

```
function handleDragStart(e) {
    var data = {
        elementId: this.id,
        message: "You moved an element!"
    };
```

The new code creates an object called `data`. It has an `elementId` property to contain the element's `id` value, and a `message` property that contains arbitrary text. You want to use this object as the drag and drop's transfer data; so, you have to serialize it:

```
    e.dataTransfer.setData("text", JSON.stringify(data));
}
```

You call the `JSON.stringify()` method to do just that, and the resulting JSON text is set as the transfer's data.

The remaining changes appear in the `handleOverDrop()` function. Its first few lines are the same:

```
function handleOverDrop(e) {
    e.preventDefault();

    if (e.type != "drop") {
        return;
    }
```

But the next two lines are new:

```
    var json = e.dataTransfer.getData("text");
    var data = JSON.parse(json);
```

You retrieve the transferred data with the `getData()` method and store it in the `json` variable. You then parse the JSON into a JavaScript object that you store in the `data` variable. You need to retrieve the dragged element object from the document. So, you use `data.elementId` and pass it to `document.getElementById()`:

```
    var draggedEl = document.getElementById(data.elementId);

    if (draggedEl.parentNode == this) {
        this.className = "";
        return;
```

```
        }

        draggedEl.parentNode.removeChild(draggedEl);

        this.appendChild(draggedEl);
        this.className = "";
```

After you remove the dragged element from its parent and append it to the drop target, you reach into your `data` object and alert its `message` to the user:

```
        alert(data.message);
    }
```

This technique of using JSON to store object data is useful in a variety of scenarios. In the next chapter, you use the same technique to store object data directly in the browser.

SUMMARY

In this chapter, you looked at JSON, a text format for storing and transmitting objects, arrays, and simple values. Let's look at some of the things discussed in this chapter:

- Serialization is the process of translating objects and values into a string representation of those objects and values.
- The web used to use XML for storing and transmitting JavaScript data, but JSON is now the format of choice.
- JSON is not JavaScript, but a subset of JavaScript. Its syntax looks similar, but key differences exist between the two. For one, JSON does not have variables or functions. It is simply a data format.
- JSON strings must be surrounded by double quotes. Single quotes result in an error.
- Numbers, booleans, and `null` appear as literal values in JSON.
- JSON objects look very much like JavaScript object literals except their properties are strings and there are no trailing semicolons.
- JSON arrays are almost identical to JavaScript array literals, but they do not have a trailing semicolon.
- You serialize JavaScript objects, arrays, and values using the `JSON` object's `stringify()` method.
- You parse JSON text into a JavaScript object or value using `JSON.parse()`.

In the next chapter, you look at how to store data in and for the browser using local storage and cookies.

EXERCISES

You can find a suggested solution to this question in Appendix A.

1. The code for alerting a single message in Example 1 isn't very exciting. Modify the code to display a random message from a set of three possible messages.