

CHAPTER 4



Introducing JSON

The JavaScript Object Notation data format, or JSON for short, is derived from the literals of the JavaScript programming language. This makes JSON a subset of the JavaScript language. As a subset, JSON does not possess any additional features that the JavaScript language itself does not already possess. Although JSON is a subset of a programming language, it itself is not a programming language but, in fact, a data interchange format.

JSON is known as the data interchange standard, which subtextually implies that it can be used as the data format wherever the exchange of data occurs. A data exchange can occur between both browser and server and even server to server, for that matter. Of course, these are not the only possible means to exchange JSON, and to leave it at those two would be rather limiting.

History

JSON is attributed to being the creation of Douglas Crockford. While Crockford admits that he is not the first to have realized the data format,¹ he did provide it with a name and a formalized grammar within RFC 4627. The RFC 4627 formalization, written in 2006, introduced the world to the registered Internet media type `application/json`, the file extension `.json`, and defines JSON's composition. In December 2009, JSON was officially recognized as an ECMA standard, ECMA-404, and is now a built-in aspect of the standardization of ECMAScript-262, 5th edition.

Controversially, another Internet working group, the Internet Engineering Task Force (IETF), has also recently published its own JSON standard, RFC 7159, which strives to clean up the original specification. The major difference between the two standards is that RFC 7159 states that a valid JSON text must encompass any valid JSON values within an initial object or an array, whereas the ECMA standard suggests that a valid JSON text can appear in the form of any recognized JSON value. You will learn more about the valid JSON values when we explore the structure of JSON.

It is important to remember, as we get further into the structure of JSON, that as a subset of JavaScript, it remains subject to the same set of governing rules defined by the ECMA-262 standardization. You can feel free to read about the latest specification at the following URL: www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf. At the time of writing, the current edition of the ECMA-262 standard is 5.1; however, 6 is just around the corner.

■ **Note** While edition 5.1 is today's current standard, at the time of JSON's formalization,

the ECMA-262 standard was only in edition 3.

Crockford documented JSON's grammar on <http://json.org> in 2001, and soon word began to spread that there was an alternative to the XML data format. With the widespread adoption of Ajax (Asynchronous JavaScript and XML), JSON's popularity began to soar, as people began to note its ease of implementation and how it rivaled that of XML. You would think that Ajax would have enforced the adoption of XML, as the x within the acronym strictly refers to XML. However, being modeled after SGML, a document format, XML possesses qualities that make it very verbose, which is not ideal for data transmission. One of the reasons JSON has become the de facto data format of the Web, as you will shortly see in the upcoming section, is due to its grammatical simplicity, which allows for JSON to be highly interoperable.

JSON Grammar

JSON, in a nutshell, is a textual representation defined by a small set of governing rules in which data is structured. The JSON specification states that data can be structured in either of the two following compositions:

1. A collection of name/value pairs
2. An ordered list of values

Composite Structures

As the origins of JSON stem from the ECMAScript standardization, the implementations of the two structures are represented in the forms of the object and array. Crockford outlines the two structural representations of JSON through a series of syntax diagrams. As I am sure you will agree, these diagrams resemble train tracks from a bird's-eye view and thus are also referred to as *railroad diagrams*. Figure 4-1 illustrates the grammatical representation for a collection of string/value pairs.

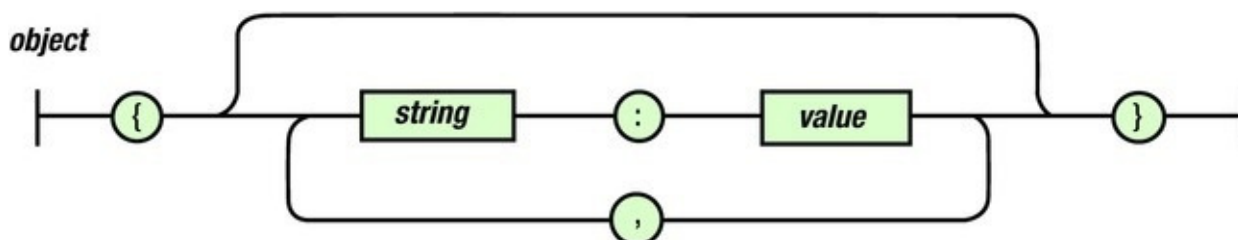


Figure 4-1. Syntax diagram of a string/value pair collection

As the diagram outlines, a collection begins with the use of the opening brace ({), and ends with the use of the closing brace (}). The content of the collection can be composed of any of the following possible three designated paths:

- The top path illustrates that the collection can remain devoid of any string/value pairs.
- The middle path illustrates that our collection can be that of a single

string/value pair.

- The bottom path illustrates that after a single string/value pair is supplied, the collection needn't end but, rather, allow for any number of string/value pairs, before reaching the end. Each string/value pair possessed by the collection must be delimited or separated from one another by way of a comma (,).

■ **Note** String/value is equivalent to key/value pairs, with the exception that said keys must be provided as strings.

An example of each railroad path for a collection of string/value can be viewed within [Listing 4-1](#). The structural characters that identify a valid JSON collection of name/value pairs have been provided emphasis.

Listing 4-1. Examples of Valid Representations of a Collection of Key/Value Pairs, per JSON Grammar

```
//Empty Collection Set  
{};  
//Single string/value pair  
{"abc":"123"};  
//Multiple string/value pairs  
{"captainsLog":"starDate 9522.6","message":"I've never trusted  
Klingons, and I never will."};
```

[Figure 4-2](#) illustrates the grammatical representation for that of an ordered list of values. Here we can witness that an ordered list begins with the use of the open bracket ([) and ends with the use of the close bracket (]).

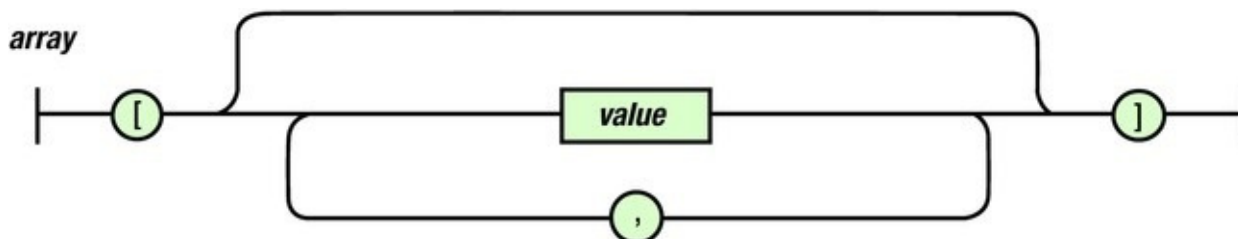


Figure 4-2. Syntax diagram of an ordered list

The values that can be held within each index are outlined by the following three “railroad” paths:

- The top path illustrates that our list can remain devoid of any value(s).
- The middle path illustrates that our ordered list can possess a singular value.
- The bottom path illustrates that the length of our list can possess any number of values, which must be delimited, that is, separated, with the use of a comma (,).

An example of each railroad path for the ordered list can be viewed within [Listing 4-2](#). The structural tokens that identify a valid JSON ordered list have been emphasized.

Listing 4-2. Examples of Valid Representations of an Ordered List, per JSON Grammar

```
//Empty Ordered List
[];
//Ordered List of multiple values
["abc"];
//Ordered List of multiple values
["0",1,2,3,4,100];
```

You may have found yourself wondering how it came to be that the characters `[`, `]`, `{`, and `}` represent an array and an object, as illustrated in [Listing 4-1](#) and [Listing 4-2](#). The answer is quite simple. These come directly from the JavaScript language itself. These characters represent the Object and Array quite literally.

As was stated in [Chapter 2](#), both an object and an array can be created in one of two distinct fashions. The first invokes the creation of either, through the use of the constructor function defined by the built-in data type we wish to create. This style of object invocation can be seen in [Listing 4-3](#).

Listing 4-3. Using the new Keyword to Instantiate an object and array

```
var objectInstantiation = new Object(); //invoking the
constructor returns a new Object
var arrayInstantiation = new Array(); //invoking the
constructor returns a new Array
```

The alternative manner, which we can use to create either object or array, is by *literally* defining the composition of either, as demonstrated in [Listing 4-4](#).

Listing 4-4. Creation of an object and an array via Literal Notation

```
var objectInstantiation = {}; //creation of an empty object
var arrayInstantiation = []; //creation of an empty array
```

[Listing 4-4](#) demonstrates how to create both an array and an object, explicitly using JavaScript's literal notation. However, both instances are absent of any values. While it is perfectly acceptable for an array or object to exist without content, it will be more likely that we will be working with ones that possess values.

Because object literals can be used to design the composition of objects within source code, they can also be provisioned with properties as they are authored. [Listing 4-5](#) should begin to resemble the syntax diagrams we just reviewed.

Listing 4-5. Designing an object and array via Literal Notation with the Provision of Properties

```
var objectInstantiation = {name:"ben",age:36};
var arrayInstantiation = ["ben",36];
```

■ **Note** While [Listing 4-4](#) and [Listing 4-5](#) illustrate the creation of objects through the use of literals, JSON uses literals to capture the composition of data.

The JSON data format expresses both objects and arrays in the form of their literal. In fact, JSON uses literals to capture all JavaScript values, except for the Date object, as it lacks a literal form.

What you may not have noticed, due to its subtlety, is that JavaScript object literals do not require its key identifiers to be explicitly defined as strings. Take, for example, the literal declaration of `{name:"ben", age:36};` from [Listing 4-5](#). It could have equally been declared as `{"name":"ben", age:36};`. Both declarations will create the same object, allowing our program to reference the same `name` property equally. Consider the code within [Listing 4-6](#).

Listing 4-6. Object Keys Can Be Defined Explicitly or Implicitly As Strings

```
var objectInstantionA    = {name:"ben",age:36};
var objectInstantionB    = {"name":"ben",age:36};
console.log( objectInstantionA.name );    // "ben"
console.log( objectInstantionB.name );    // "ben"
```

The reason the preceding example works is because, behind the scenes, JavaScript turns every key identifier into a string. That said, it is imperative that the key of every value pair be wrapped in double quotes to be considered valid JSON. This is due to the many reserved keywords in JSON's superset and the fact that ECMA 3.0 grammar prohibits the use of keywords as the properties held by an object. The ECMA 3.0 grammar does not allow reserved words (such as *true* and *false*) to be used as a key identifier or to the right of the period in a member expression.² [Listing 4-7](#) demonstrates the first JSON text used to interchange data.³

Listing 4-7. The Very First JSON Message Used by Douglas Crockford

```
var firstJSON = {to:"session",do:"test","message":"Hello
World"}; //Syntax Error in ECMA 3
```

However, this JSON text produced an error instantly, due to the use of the reserved keyword **do** as the property name of a string/value pair. Rather than outlining all words that would then cause such syntax errors, Crockford found it simpler to formalize that all property names must be explicitly expressed as strings.

■ **Note** If you were to reference the exact preceding code expecting to arrive at a syntax error, you'll likely be confused why none is thrown. The ECMAScript, 5th edition allows for keywords to now be used with dot notation. However the JSON spec continues to account for legacy.

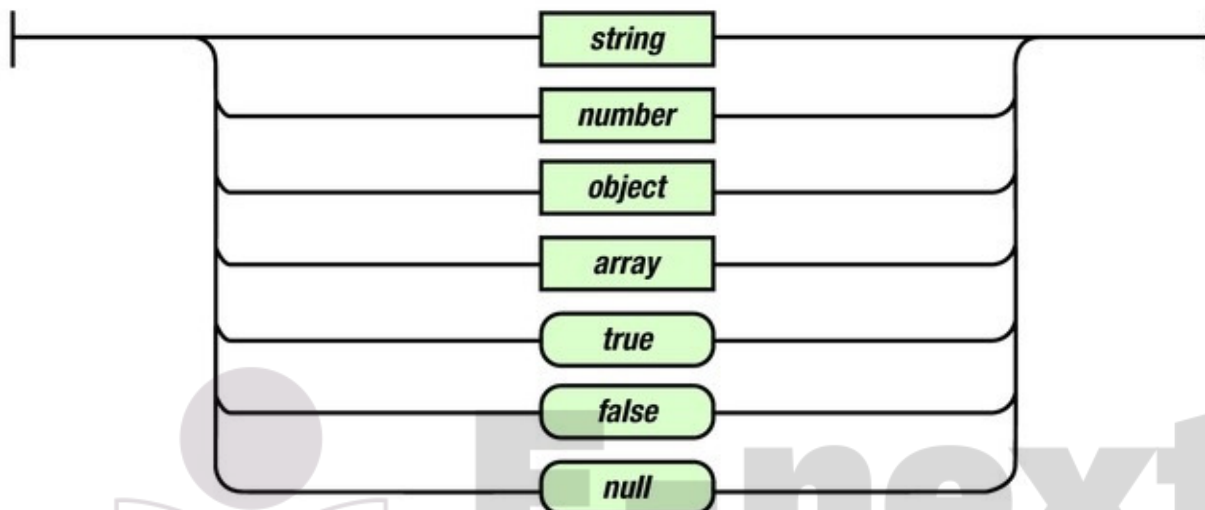
JSON Values

As mentioned earlier, JSON is a subset of JavaScript and does not add anything that the

JavaScript language does not possess. So, naturally, the values that can be utilized within our JSON structures are represented by types, as outlined within the 3rd edition of the ECMA standard. JSON makes use of four primitive types and two structured types.

The next figure in succession, [Figure 4-3](#), defines the possible values that can be substituted where the term *value* appears in [Figures 4-1](#) and [4-2](#). A JSON value can only be a representative of string, number, object, array, true, false, and null. The latter three must remain lowercased, lest you invoke a parsing error. While [Figure 4-3](#) does not clearly demonstrate it, all JSON values can be preceded and succeeded by whitespace, which greatly assists in the readability of the language.

value



[Figure 4-3](#). Syntax diagram illustrating the possible values in JSON

String literals in the JavaScript language can possess any number of Unicode characters enclosed within either single or double quotes. However, it will be important to note, as outlined in [Figure 4-4](#), that a JSON string must always begin and end with the use of double quotes. While Crockford does not justify this, it is for interoperable reasons. The C programming grammar states that single quotes identify a single character, such as *a* or *z*. A double quote, on the other hand, represents a string literal. While [Figure 4-4](#) appears verbose, there are only four possible paths.

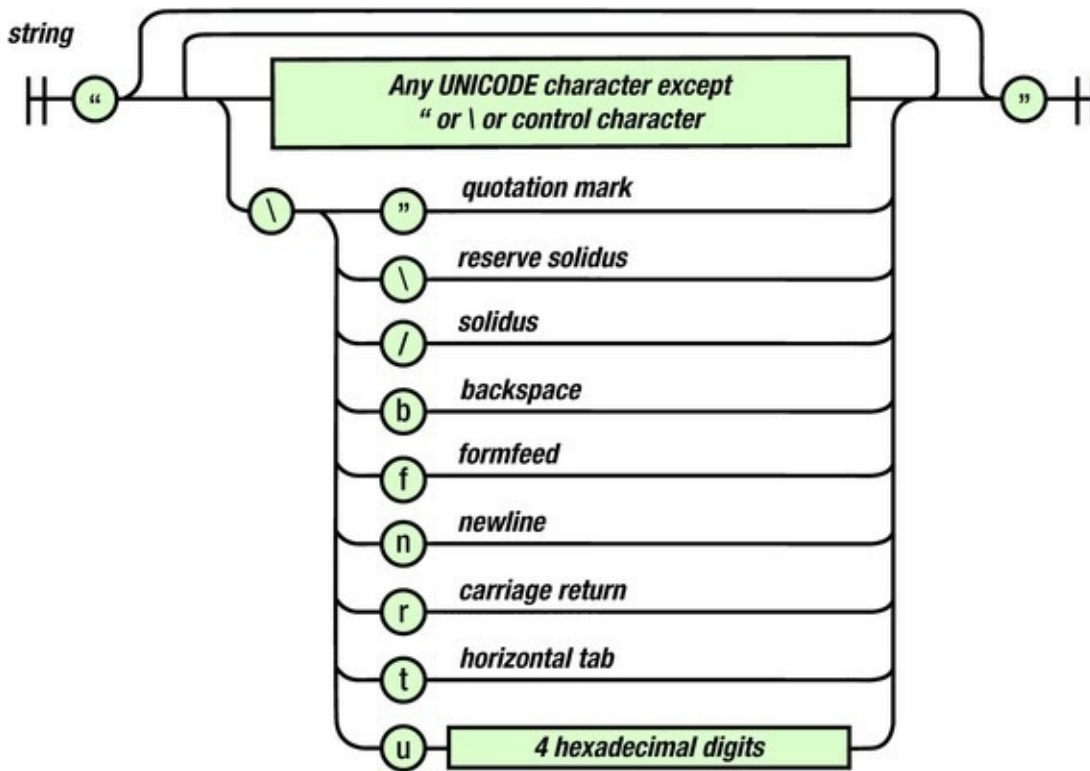


Figure 4-4. Syntax diagram of the JSON string value

- The topmost path illustrates that our string literal can be absent of any Unicode characters.
- The middle path illustrates that our string can possess any Unicode characters (represented in literal form), except for the following: the quotation mark, the backslash (solidus).
- The last several paths illustrate that we can insert into our string control characters with the use of a solidus (\) character preceding it. Additionally, the bottommost rung specifies that any character can be defined in its Unicode representation. To indicate that the preceding *u* character is used to identify a Unicode value, it, too, must be escaped.
- The second topmost path represents our loop, which allows the addition of any of the outlined characters.

Listing 4-8 demonstrates a variety of valid string values.

Listing 4-8. Examples of Valid String Values As Defined by the JSON Grammar

```
//absent of unicode
"";
//random unicode characters
"Σ"; or " ";
//use of escaped character to display double quotes;
" \" \" ";
//use of \u denotes a unicode value
"\u22A0"; // outputs
//a series of valid unicode as defined by the grammar
```


3. Can be in the form of a whole number
 - a. made up of a single BASE10 numeric literal (0-9)
 - b. made. any number of BASE10 numeric literals (0-9)
4. Can be in the form of a fraction
 - 4.1. Made up of a singular base10 numerical literal at the 10s placement
 - 4.2. Made up of any base10 numerical literal per placement beyond the decimal
5. Can possess the exponential demarcation literal
 - 5.1. E notation can be expressed in the form of a uppercase “E” or lowercase “e”
 - 5.2. Immediately followed by a signed sequence of 1 or more base10 numeric literals (0-9)

[Listing 4-9](#) reveals valid numerical values as defined by the JSON grammar.

Listing 4-9. Valid Numerical Values

```
-0.01    //valid use of 0's
00.1     //superfluous 0 produces a SyntaxError
1/3      //fraction form
.3333333333333333 //decimal form
1.2e-1   //scientific notation
```

Any of the values discussed in this chapter can be used in any combination when contained within a composite structure. [Listing 4-10](#) illustrates how they can be mixed and matched. What is necessary is that the JSON grammar covered is followed. The examples in [Listing 4-10](#) demonstrate proper adherence of the JSON grammar to portray data.

Listing 4-10. Examples of JSON Text Containing a Variety of Valid JSON Values

```
// JSON text of an array with primitives
[
  null,  true,  8
]
// JSON text of an object with two members
{
  "first": "Ben",
  "last": "Smith",
}
// JSON text of an array with nested composites
[
  { "abc": "123" },
  [ "0", 1, 2, 3, 4, 100 ]
]
```

```
//JSON text of an object with nested composites
{
  "object": {
    "array": [true]
  }
}
```

JSON Tokens

While the Object and Array are conventions used in JavaScript, JavaScript, like many programming languages, borrowed from the C language in one form or another. While not every language explicitly implements Arrays and Objects akin to JavaScript, they do often possess the means to model collections of key/value pairs and ordered lists. These may take on the form of Hash maps, dictionaries, Hash tables, vectors, collections, and lists. Furthermore, most languages will be capable of working with text, which is precisely what JSON is based on.

At the end of the day, JSON is nothing more than a sequence of Unicode characters. However, the JSON grammar standardizes which Unicode characters or “tokens” define valid JSON, in addition to demarcating the values contained within.

Therefore, when regarding the interchange of JSON and the many languages that do not natively possess Objects and Arrays, the tokens that make up the JSON text are all that is required to interpret if any collections or ordered lists exist and apply all values in a manner required of that language. This is accomplished with six structural characters, as listed in [Table 4-2](#).

Table 4-2. Six Structural Character Tokens

Token	Escaped Value	Unicode Value	Literal	Name
Array Opening	%5b	\u005b	[Left Square Bracket
Array Closing	%5d	\u005d]	Right Square Bracket
Object Opening	%7b	\u007b	{	Left Curly Bracket
Object Closing	%7d	\u007d	}	Right Curly Bracket
Name/Value Separator	%3a	\u003a	:	Colon
Value Separator	%2c	\u002c	,	Comma

One point to note is that JSON will ignore all insignificant whitespace before or after the preceding six structural tokens. [Table 4-3](#) illustrates the four whitespace character tokens.

Table 4-3. Four Whitespace Character Tokens

Token	Name	Escaped Value	Unicode Value
Control Character	Space	%20	\u0020
Control Character	Horizontal Tab	%09	\u0009
Control Character	Line Feed/New Line	%0A	\u000A
Control Character	Carriage Return	%0D	\u000D

Because JSON is nothing more than text, you may find it rather difficult to determine whether your JSON is properly formatted or not. Furthermore, if the syntax is inaccurate to the grammar specified, then you will find that your malformed JSON causes code to come to a halt. This would be due to the syntax error that would be uncovered at the time of trying to parse said JSON. You will learn about parsing in [Chapter 6](#).

For this reason, any attempt to devise JSON by hand should be performed with the aid of an editor. The following list of JSON editors understand the JSON grammar and are able to offer some much needed and immediate validation.

- <http://jsoneditoronline.org/>
- <http://jsonlint.com/>

The first editor, <http://jsoneditoronline.org/>, adheres to the ECMA-262 standardization and, therefore, allows your JSON text to represent a singular primitive value. Whereas the latter follows the RFC 7159 standardization, thus requiring a JSON text to represent a structural value, i.e., array or object literal. It should be made known that the two editors mentioned previously are not the only two in existence. There are many online and offline editors, each with its own nuances. I favor the two mentioned, for their convenience.

Summary

In this chapter, I covered the history of JSON and the specifications of the JSON data format that defines the grammar of a valid JSON text. You learned that JSON is a highly interoperable format for data interchange. This is achieved via the standardization of a simplistic grammar that can be translated into any language simply by understanding the grammar.

As was demonstrated in this chapter, we can use the JSON grammar in conjunction with predetermined data to create JSON. Because we are simply working with text, it will be helpful to rely on an editor that understands JSON's grammar, for validation purposes. However, JSON can be written with a basic text editor and saved as a JSON document, using the file extension `.json`. Furthermore, as a subset of JavaScript, JSON can even be hard-coded within a JavaScript file directly. Both methods are ideal for devising configuration files for an application.

The next chapter will reveal how we can use the JavaScript language to produce JSON at runtime.

Key Points from This Chapter

- The array represents an ordered list of values, whereas the object represents a collection of key/value pairs.
- Unordered collections of key/value pairs are contained within the following opening ({) and closing (}) brace tokens.
- Ordered lists are encapsulated within opening ([) and closing (]) square bracket tokens.
- The key of a member must be contained in double quotes.
- The key of a member and its possessed value must be separated by the colon (:) token.
- Multiple values within an object or array must be separated by the comma (,) token.
- Boolean values are represented using lowercase true/false literals.
- Number values are represented using double-precision floating number point format.
- Number values can be specified with scientific notation.
- Control characters must be escaped via the reverse solidus (\) token.
- Null values are represented as the literal: null.

¹<http://yuiblog.com/yuitheater/crockford-json.m4v>.

²Allen Wirfs-Brock, “ES 3.1 ‘true’ as absolute or relative?” <https://mail.mozilla.org/pipermail/es-discuss/2009-April/009119.html>, April 9, 2009.

³<http://yuiblog.com/assets/crockford-json.zip>.

CHAPTER 5



Creating JSON

Serialization is the process of taking a snapshot of a data structure in a manner that allows it to be stored, transmitted, and reconstructed back into a data structure at a later point in time. As serialization is merely a process rather than the utilization, its applications are mainly limited by your application's needs. This chapter will explore the serialization methods utilized by the JavaScript language and required of the JSON subset.

While serialization may seem like a mystical concept, the result of the snapshot, at the most atomic level, is nothing more than a string. The serialization process is simply the construction of said string, which often occurs behind the scenes. What is important to note is that in JavaScript, the produced string incorporates the representations of data in their literal forms. By capturing data in their literal form, each literal can be evaluated back into its respective JavaScript values.

■ **Note** A serialized value could result in a simple-looking string, such as `"\Hello-World\""` or `"false"`.

You learned in [Chapter 4](#) that any C language can easily work with JSON. The most prominent reason is that all C languages possess a means to represent collections of name/value pairs, ordered lists, Booleans, and strings. Nevertheless, in the few cases in which the literals that make up the JSON subset are not inherently understood by a specific language, a translation among grammars can take place. This occurs by simply deconstructing the JSON text into a series of tokens and deriving meaningful structures that are possible within the grammar of that particular language.

■ **Note** Grammar translation is the process of converting the syntax of one language equivalently into that of another.

Conversely, one can construct JSON from any data structure, simply by following the grammar defined by the JSON specification. In [Chapter 6](#), you will learn more about such reconstruction. This chapter will focus on how to create a JSON text from JavaScript values.

The Serialization Process—Demystified

As was discussed in [Chapter 3](#), all JavaScript values can be converted into their string equivalent form by adding it, via the addition operator, with another string, as seen in [Listing 5-1](#).

Listing 5-1. Concatenating Primitive Values with Strings

```

""+1;           //produces "1"
""+true;        //produces "true"
""+null;        //produces "null"
""+undefined;   //produces "undefined"
""+"Hello";     //produces "Hello"

```

While the string representations for all primitive values are captured as expected, as displayed in [Listing 5-2](#), the same cannot be said of non-primitive values.

Listing 5-2. Concatenating Non-Primitive Values with Strings

```

""+{identifier:"Hello"};           //produces "[object Object]"
""+["Hello",["hello","World"]];   //produces
"Hello,hello,World"

```

As revealed in [Listing 5-2](#), while the JavaScript language possesses the ability to create objects out of literal forms, there is no easy way to perform the contrary. In order to deconstruct an object into that of its literal form, the members of an instance must be traversed, analyzed, and assembled piece by piece into its corresponding literal form.

To accomplish this undertaking, we must rely on the use of loops, string manipulation, and the appropriate sequencing of the necessary structural tokens, listed in [Table 5-1](#).

Table 5-1. The Six Structural Character Tokens

Token	Literal	Name
Array Opening	[Left Square Bracket
Array Closing]	Right Square Bracket
Object Opening	{	Left Curly Bracket
Object Closing	}	Right Curly Bracket
Name/Value Separator	:	Colon
Value Separator	,	Comma

The following code in [Listing 5-3](#) demonstrates, as succinctly as possible, a method that transforms a supplied object into that of its literal form counterpart.

Listing 5-3. Converting an object and Its Property into an object literal

```

1  var author = new Object();
2      author.name = "Ben";

3  var literal = stringify(author);

4  function stringify(structure){
    //if the structure supplied possesses the string data
type
5      if(typeof structure=="string"){

```



```

6         return ''+String(structure)+'';
7     }
    //if the structure supplied possess the object data
type
8     if(typeof structure=="object"){
9         var partial=[];
            //for each property held by our structure
10        for(var k in structure){
11            var v= structure[k];
12            v = stringify(v);
13            partial.push(k+" : "+v);
14        }
            //if partial does not possess children capture
opening/closing brackets;
15        v = (partial.length === 0)? '{} '
16            //otherwise, comma delimit all values within
opening/closing brackets
17            : ' { ' + partial.join(' , ' ) + ' } '
18        return v;
19    }
20 }
21 console.log(literal);    // "{ name : "Ben" }"
22 console.log(typeof literal);    // "string"

```

Our demonstration begins (**line 1**) with the creation of an object `author` who is assigned a singular property `name`. We next supply `author` to the `stringify` function as the object we wish to transform into its literal representation. The `stringify` function then analyzes the data type of the structure supplied, in order to determine the appropriate course of action.

When `stringify` ascertains that the supplied structure is an object (**line 8**), the function then proceeds to traverse all members in its possession. The value of each key enumerated this way is in turn supplied to the `stringify` method, to be transformed into its literal form. Alas, this time, the data type is found to be that of a string. In order to capture said string as its literal counterpart, the function surrounds it with double quotes and returns it back to the caller of the invocation (**line 12**). From here, the current key, `k`, and its value, `v`, are sequenced together, separated by a colon (`:`) and stored within the array `partial`, so that any remaining properties can be enumerated similarly.

To keep this example short, `author` is in possession of one property. However, were there more properties possessed by our structure, the preceding process would be repeated until every single one is deconstructed and converted into its literal counterpart and appended to the final string representation. When there are no further properties to analyze, we determine if the length of `partial` is greater or equal to that of zero. If `partial`'s length is 0, it does not possess any values, and, therefore, a string consisting of a pair of opening/closing braces is devised.

Otherwise, we create a string that joins each value with a comma separator (`,`) and

insert it within a pair of opening/closing brace tokens. The serialized literal is then returned to the invoker (**line 3**). The demonstration ends by outputting the final representation, revealing our reverse-engineered object literal (**line 21**).

■ **Note** In the preceding example, `stringify` is only capable of converting strings and objects into their literal counterparts. Crafted for that purpose only, it is not capable of recognizing all types.

We're very close to our goal. However, this literal isn't able to be considered valid JSON, as it does not fully adhere to the JSON grammar. The key `name` in our key/value pair must be surrounded by double quotes. Fortunately, this is easy to remedy with strings: `partial.push('"' + k + '"' + ": " + v);`. If we were to log our result once again, we would see the following: `"{"name": "Ben"}"`.

While the demonstration in [Listing 5-1](#) possessed a singular member, it will not be unlikely that the data requiring serialization possesses the makeup of objects nested within objects. Four objects are used in total to represent our `author` object, as seen in [Listing 5-4](#), and each is used to organize data. One object is used as a list, which includes the pets owned by yours truly. Another two are used to capture the names and ages of each pet. While both pets are contained within the ordered list, the ordered list itself is held as just another property on our `author` instance.

Listing 5-4. A Nested Data Structure

```
var author = new Object();
  author.name = "Ben";
  author.age = 36;
  author.pets = [
    { name : "Waverly" , age : 3.5 },
    { name : "Westley" , age : 4 }
  ]
```

If we were to serialize `author` from [Listing 5-4](#) using the `stringify` function outlined in [Listing 5-3](#), each property possessed by the top-level element would be enumerated. Similarly, the value held by each key would be supplied to its own invocation of the `stringify` function as the top-level element to have its composition serialized. This process continues until all values of all structures have been analyzed, serialized, and concatenated as a valid JSON text.

■ **Note** Object properties and Array indexes represent a key.

As the `stringify` function demonstrates, transforming a JavaScript object into a valid JSON representation requires the use of identifying data types, recursion, and a heavy amount of string manipulation. Fortunately for us, the formalizer of JSON, Douglas Crockford, devised a JSON library that would conveniently produce JSON text from that of a specified datum. The JSON library is a convenient JavaScript file, which can be downloaded from the following GitHub URL:

<https://github.com/douglascrockford/JSON-js/blob/master/json2.js>.

The JSON Object

As a JavaScript file, the `json2.js` library can be included in any existing application, by referencing the downloaded library within the `<head></head>` tags on each HTML page that seeks use of it. [Listing 5-5](#) incorporates the JSON library by referencing the location of the library, relative to the top directory, within the script tag in the head of the following HTML file. In this example, the `json2.js` file has been downloaded within the `js/libs/` directory of the working directory of a project.

Listing 5-5. HTML Markup Referencing the Inclusion of the `json2.js` JavaScript Library

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <link rel="stylesheet" href="css/style.css">
    <script src="js/libs/json2.js"></script>
  </head>
  <body>
  </body>
</html>
```

When the page is viewed in a browser, and as soon as the `json2.js` file is loaded, the JSON Object declared by `json2.js` is added to the global namespace, so that the serialization method can be accessed from within any scope. Unlike the built-in objects, such as `Object` or `Array`, whose global methods can be used as a constructor to create instances of these objects via the keyword `new`, the JSON Object does not possess a constructor at all. Instead, the JSON Object possesses two methods, `parse` and `stringify`. However, this chapter will only discuss one of them: `stringify`.

stringify

As the name suggests, `stringify` is used for serializing JavaScript values into that of a valid JSON. The method itself accepts three parameters, `value`, `replacer`, and `space`, as defined by the signature in [Listing 5-6](#). As I mentioned, the JSON Object is a global object that does not offer the ability to create any instances of the JSON Object. Any attempt to do so will cause a JavaScript error. Therefore, one must simply access the `stringify` method via the global JSON Object.

Listing 5-6. Syntax of the JSON `stringify` Method

```
JSON.stringify(value[, replacer [, space]]);
```

■ **Note** The brackets surrounding the two parameters, `replacer` and `space`, is just a way to illustrate in a method definition what is optional. However, while an argument supplied to the method may be optional, you must follow the proper parameter order, as outlined by the method. In other words, to specify an argument for `space` but not `replacer`, you must supply `null` as the second argument to the `stringify` method.

value

The `value` parameter of the `stringify` method is the only required parameter of the three outlined by the signature in [Listing 5-6](#). The argument supplied to the method represents the JavaScript value intended to be serialized. This can be that of any object, primitive, or even a composite of the two. As both Objects and Arrays are composite structures, the argument supplied can be in possession of any combination of objects and primitives nested within, much like our `author` object from [Listing 5-4](#). Let's jump right in and serialize our `author` object as shown in [Listing 5-7](#).

Listing 5-7. HTML Markup Demonstrating the Output of `JSON.stringify`

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <link rel="stylesheet" href="css/style.css">
    <script src="js/libs/json2.js"></script>
  </head>
  <body>
    <script>
      //obtain a reference to the body tag
      var body = document.getElementsByTagName("body")[0];

      //function log will append a value to the body for
output
      function log(jsonText) {
        //surround supplied jsonText with double quotes
and append a new line
        body.innerHTML += '"' + jsonText + '"<br>';
      }
      var author = new Object();
      author.name = "Ben";
      author.age = 36;
      author.pets = [
        { name : "Waverly" , age : 3.5 },
        { name : "Westley" , age : 4 }
      ];

      var JSONtext = JSON.stringify(author)
      log( JSONtext );
```

```
    </script>
  </body>
</html>
```

[Listing 5-7](#) leverages the markup from [Listing 5-5](#) and inserts within the body a script defining our `author` object. Immediately following, we supply `author` into that of `JSON.stringify`, which returns the following JSON text:

```
"{"name": "Ben", "age": 36, "pets":
[{"name": "Waverly", "age": 3.5}, {"name": "Westley", "age": 4}]}"
```

The produced JSON captures the data precisely as it was housed within the `author` object. The great thing about the serialization process is that all of the work is encapsulated behind the scenes. This allows us to remain unconcerned as to how the encoding logic works, in order to be able to use it as we just have.

Serializing structures equivalent to `author` will work out marvelously, as it possesses only the values that are formalized as valid values of the JSON grammar. On the other hand, as the needs of an application become more complex than that of `author`, you may encounter a few oddities in the way that your data is outputted.

Your program being written in JavaScript will surely take advantage of all the language has to offer, as well it should. Yet, as JSON is a subset of the JavaScript language, many objects and primitives employed by your application may not be serialized as expected. You may come to find that this is both a blessing and a curse. However, either way you see it, it will be an undeniable fact. Therefore, short of learning the inner workings of the `stringify` method, it will be important to understand how the serializer handles particular values it comes across, in order to be able to anticipate arriving at the expected or even necessary results.

■ **Tip** The serialization process occurs in a synchronous manner. In other words, the moment you call the `stringify` method, all remaining code that has to be executed must wait for the serialization to conclude before it can proceed. Therefore, it will be wise to keep your objects as concise as necessary during the serialization process.

EXERCISE 5-1. STRINGIFY

Let's now experiment with a few types of data structures and see what JSON text is outputted. Create an HTML file within the top root of a working directory, and within it, copy the code from [Listing 5-5](#). Within that same directory, create a `js/` directory and a `libs/` directory within it. If you have not already downloaded `json2.js`, do so and save it within `js/libs/`. Revisit the created `.html` file and within the body tag, include the following lines of code:

```
01. <script>
02.     //obtain a reference to the body tag
03.     var body = document.getElementsByTagName("body")
    [0];
```

```

04.      //function log will append a value to the body as
a string value for output
05.      function log(jsonText) {
06.          //wrap all strings with double quotes and
append a new line
07.          body.innerHTML += '"' + jsonText
+ '"<br>';
08.      }
09.
10.      log(JSON.stringify(false));
11.      log(JSON.stringify(undefined));
12.      log(JSON.stringify([undefined]));
13.      log(JSON.stringify(["undefined", false]));
14.      log(JSON.stringify({prop : undefined }));
15.      log(JSON.stringify(new Date("Jan 1 2015")));
16.
17.      var obj = new Object();
18.      obj.name = "name-test";
19.      obj.f = function() { return "function-
test" };
20.
21.      log(JSON.stringify(obj));
22.      log(JSON.stringify("this example \u000A\u000D has
control characters"));
23.      log(JSON.stringify( "true" ));
24.      log( JSON.stringify( 1/0 ));
25.      log( JSON.stringify( Infinity ));
26.      log( JSON.stringify( [ function(){ return "A"}
] ));
27.
28.      var selfReference= new Array();
29.      selfReference[0]=selfReference;
30.      //because line 31 will throw an error, we must
wrap it with a try catch to view the error
31.      try{ JSON.stringify( selfReference ) } catch(e){
log(e) };
32.  </script>

```

Once you've added the following script to your HTML file, open that .html file in your favorite browser and observe the output for each data serialized. Your results should be comparable to the results shown in the following table.

Results of the Code Produced

Exercises	Outputs
JSON.stringify(false);	"false"

JSON.stringify([undefined]);	"[null]"
JSON.stringify(["undefined" , false]);	"[\\"undefined\\",false]"
JSON.stringify({ prop:undefined });	"{}"
JSON.stringify(new Date("Jan 1 2015"));	"\\"2015-01-01T05:00:00.000Z\\""
var obj= new Object(); obj.name="name-test"; obj.f=function(){ return "function-test"	"{\\"name\\":\\"name-test\\"}"
}; JSON.stringify(obj);	
JSON.stringify("this example \u000A\u000D has control characters");	"\\"this example \n\r has control characters\\""
JSON.stringify("true");	"\\"true\\""
JSON.stringify(1/0);	"null"
JSON.stringify(Infinity);	"null"
JSON.stringify([function(){ return "A"}]);	"[null]"
var selfReference= new Array(); selfReference[0]=selfReference; JSON.stringify(selfReference);	TypeError: cyclic object value

As you can see from the results of our exercise, the `stringify` method doesn't acknowledge a few values. First and foremost, you may have realized that an `undefined` value is handled in one of two possible manners. If the value `undefined` is found on a property, the property is removed entirely from the JSON text. If, however, the value `undefined` is found within an ordered list, the value is converted to `'null'`.

Functions are also disregarded by the `stringify` method, even functions that would return a string to the key holding it. The `stringify` method only analyzes and encodes values; it does not evaluate them. Therefore, functions when encountered by `stringify` are replaced with the `undefined` value. The rules I covered previously regarding an `undefined` value will apply to the key that now references the assigned `undefined` primitive. There is one method that will be invoked, if found to have that of a particular method name. I will talk more about this later in the `toJSON` section.

As JavaScript does not possess a date literal, `Dates` are automatically serialized as string literals, based on the (UTC) ISO encoding format.

All number values must be finite. If the number is evaluated to that of an `Infinity` or `NaN`, the number will return as the literal `'null'` value.

When the sole value serialized is that of a string value, its literal form is escaped and

nested within another set of quotes.

The last takeaway from the preceding exercises is that JSON cannot handle cyclic object values, meaning that neither an array nor object can possess a value that is a reference to itself. Should you attempt to define a cyclic structure, an immediate error is thrown.

toJSON

Because dates do not possess a literal form, the `stringify` method captures all dates it encounters as string literals. It captures not only the date but time as well. Because `stringify` converts a date instance into a string, you might rationalize that it's produced by calling the `toString` method possessed by the Date object. However, `Date.toString()`, does not produce a standardized value, but, rather, a string representation whose format depends on the locale of the browser that the program is running.¹ With this output lacking a standard, it would be less than ideal to serialize this value for data interchange.

What would be ideal is to transform the contents into that of the ISO 8601 grammar, which is the standard for handling date and time interchange.

■ **Note** A JavaScript Date Object can be instantiated with the provision of an ISO formatted string.

To enable this feature, Crockford's library also includes the `toJSON` method, which is appended to the prototype of the Date Object so that it will exist on any date. [Listing 5-8](#) reveals the default `toJSON` function that will be inherited by any and all dates. ○ N

[Listing 5-8](#). Default toJSON Implementation

```
Date.prototype.toJSON = function(key) {  
    function f(n) {  
        // Format integers to have at least two digits.  
        return n < 10 ? '0' + n : n;  
    }  
  
    return this.getUTCFullYear() + '-' +  
        f(this.getUTCMonth() + 1) + '-' +  
        f(this.getUTCDate()) + 'T' +  
        f(this.getUTCHours()) + ':' +  
        f(this.getUTCMinutes()) + ':' +  
        f(this.getUTCSeconds()) + 'Z';  
};
```

When `stringify` invokes the `toJSON` method, it expects to be provided a return value. In [Listing 5-8](#), the value being returned is a string that is devised from the concatenation of the methods possessed by the instance being analyzed. The return value can be of any value that is defined in the JSON subset. Upon returning a value, the logic

within `stringify` will continue to ensure that your value is analyzed. It will do so iteratively if returned in the form of an object or, more simply, if the value returned is a primitive, it's converted into a string and sanitized. Because `stringify` continues to analyze the returned value, the rules of [Table 5-1](#) continue to apply.

■ **Note** Because `toJSON` exists as a method of a Date Object, the `this` keyword remains scoped to the particular instance being analyzed. This allows the serialization logic to be statically defined, yet each instance at runtime will reference its own values.

If you are curious as to the purpose of function `f`, function `f` wraps each method and prefixes each result with 0, if the returned number is less than 10, in order to maintain a fixed number of digits. Last, each number is arranged in a sequence combined with various tokens and joined into a string, resulting in a valid grammar, according to the ISO 8601 specification.

What is important to know about the `toJSON` method is that it can be used on more than dates. For each object analyzed, the internal logic of the `stringify` method invokes said `toJSON` method, if it is in possession of one. This means we can add `toJSON` to any built-in JavaScript Object, and even to custom classes, which, in turn, will be inherited by their instances. Furthermore, we can add it to individual instances. This inherit ability to add a `toJSON` method enables each application to provide the necessary encoding that might not otherwise be possible by default, such as that of our `date`.

■ **Note** Custom classes, when serialized, are indistinguishable from the built-in objects types.

Each call to the `toJSON` method is supplied with a key as an argument. This key references the holder of the value that `stringify` is currently examining. If the key is a property on an object, that properties label is supplied as the key to the method. If the key is the index of an array, the argument supplied will be an index. The former provides useful insight when devising conditional logic that must remain flexible or dependent on the instances context, whereas the latter is less indicative. Our `author` object possesses both a collection of key/value pairs and an ordered collection. By adding a `toJSON` method to all object instances, we can easily log the key that is provided to each `toJSON` invocation, as achieved in [Listing 5-9](#).

Listing 5-9. Attaching the `toJSON` Function to the Object Will Cause All JavaScript objects to Possess It

```
Object.prototype.toJSON=function(key){
  //log the key being analyzed
  console.log(key); //outputs the key for the current
context (shown below)
  //log the scope of the method
  console.log(this); //outputs the current context (shown
below)
  //return the object as is back to the serializer
```

```

    return this;
}
var author = new Object();
    author.name = "Ben";
    author.age = 36;
    author.pets = [
        { name : "Waverly" , age : 3.5 },
        { name : "Westley" , age : 4 }
    ];

JSON.stringify(author);

/* captured output from the above Listing */
//the author object being analyzed
//(key)      ""
//(context) Object { name="Ben", age=36, pets=[2],
more...} //truncated
//the pets object being analyzed
//(key)      pets
//(context) [Object { name="Waverly", age=3.5,
toJSON=function()},
↪Object { name="Westley", age=4,
toJSON=function()}]
//index 0 of array being analyzed
//(key)      0
//(context) Object { name="Waverly", age=3.5,
toJSON=function()}
//index 1 of array being analyzed
//(key)      1
//(context) Object { name="Westley", age=4,
toJSON=function()}

{"name":"Ben","age":36,"pets":
[{"name":"Waverly","age":3.5}, {"name":"Westley","age":4}]}"

```

Listing 5-9 demonstrates that each object that possesses the `toJSON` method is supplied with the key by which it is held. These values are logged in the order in which the properties are enumerated by the JavaScript engine. The first key that is logged from our `toJSON` method is that of an empty string. This is because the `stringify` implementation regards key/value pairs. As you can see, the immediate logging of `this` reveals the `author` object. With the return of the invoked method, `stringify` continues onto the next object it encounters.

■ **Note** The key of the initial value is always that of an empty string.

The next object the `stringify` method encounters happens to be that of an array. An array, as a subtype of `Object`, inherits and exposes the `toJSON` method and is,

therefore, invoked. The key it is passed is the identifier `pets`. Respectively, both objects contained within are invoked and provided the index to which they are ordered, those keys being 0 and 1.

The `toJSON` method provides a convenient way to define the necessary logic wherein the default behavior may fall short. While this is not always ideal, it is often necessary. However, the `toJSON` method is not the only means of augmenting the default behavior of the `stringify` method.

replacer

The second parameter, `replacer`, is optional, and when supplied, it can augment the default behavior of the serialization that would otherwise occur. There are two possible forms of argument that can be supplied. As explained within the ECMA-262 standardization, the optional `replacer` parameter is either a function that alters the way objects and arrays are stringified or an array of strings and numbers that acts as a white list for selecting the object properties that will be stringified.²

replacer Array

Suppose I had the following JavaScript data structure (see [Listing 5-10](#)) and decided to serialize it using the built-in JSON Object and its `stringify` method. By supplying the `author` instance as the value into the `JSON.stringify` method, I would be provided with the result displayed in [Listing 5-10](#).

Listing 5-10. Replaced Pets Property with E-mail

```
var author = new Object();
    author.name="ben";
    author.age=35;
    author.email="iben@spilled-milk.com";

    JSON.stringify( author );
    //  '{"name":"ben","age":35,"email":"iben@spilled-
milk.com"}'
```

As expected, the produced JSON text reflects all of the possessed properties of the `author` object. However, suppose that e-mail addresses were not intended to be serialized by our application. We could easily delete the e-mail property and then pass `author` through `stringify`. While that would prevent the e-mail address from being serialized, this method could prove problematic if our application continued to require use of the e-mail address. Rather than delete the value from the `author` object, we could take advantage of the `replacer` method.

Were we to supply the `replacer` parameter with an array whose values outline the properties we desire `stringify` to serialize, the JSON text would only capture those key/value pairs. [Listing 5-11](#) white lists the two properties, `name` and `age`, that our application is permitted to serialize.

Listing 5-11. Supplying a `replacer` array Can Specify What Keys to Output

```
//... continuation of Listing 5-10
JSON.stringify(author, ["name","age"] ); // "{"
name":"ben","age":35}"
```

Providing an ordered list as the `replacer` argument filters the properties that are output during serialization. Any identifiers that are not specified within the `replacer` array will not become a part of the JSON text. As an additional point, the order of our white-listed properties affects the way in which they respectively occur in the serialized output. Listings 5-11 and 5-12 vary by the order of the white-listed properties supplied in the `replacer`. The results reflect the specified order in each JSON text produced.

Listing 5-12. The Order of the White-Listed Properties Determines the Order in Which They Are Captured

```
//... continuation of Listing 5-10
JSON.stringify(author, ["age","name"] ); // "
{"age":35,"name":"ben}"
```

[Listing 5-11](#) displays `name` in the JSON text first, whereas in [Listing 5-12](#), `name` appears last. This has to do with the fact that our `replacer` argument is an array, and an array is simply an ordered list. In this case, the ordered list just so happens to express our white-listed properties. The serialization process then iterates over each white-listed identifier in ascending order for each collection of name/value pairs it may come across.

White-listed properties mustn't be provided in string literal form. They can also be represented as a primitive number. However, any number the method encounters is converted into its string equivalent. This is due to the fact that keys are always stored as strings behind the scenes. This is demonstrated in [Listing 5-13](#).

Listing 5-13. Numbers Used As Keys Are Converted to Strings

```
var yankeesLineup = new Object();
    yankeesLineup['1'] ="Jacoby Ellsbury";
    yankeesLineup['2'] ="Derek Jeter";
    yankeesLineup['3'] ="Carlos Beltran";
    yankeesLineup['4'] ="Alfonso Soriano";
    //...etc
JSON.stringify(yankeesLineup, [1,2] );
// "{"1":"Jacoby Ellsbury","2":"Derek Jeter}"
```

■ **Note** Even array indexes are converted into strings behind the scenes.

■ **Tip** While numbers are allowed as white-listed values, it will always be best to supply a string representation, as it will convey meaning to those who may not know that numbers are converted to strings behind the scenes when used as keys. Furthermore, using numbers as a property identifier is not the best choice for a meaningful label.

replacer Function

The alternate form that can be supplied as the replacer is that of a function. Supplying a function to the `replacer` property allows the application to insert the necessary logic that determines how objects within the `stringify` method are serialized, much like that of the `toJSON` method. In fact, the `replacer` function and the `toJSON` method are nearly identical, apart from three distinguishable characteristics. The first is that one is a function and the other is a method. The second is that the `replacer` function is provided iteratively, the key for every property encountered. Last, the `replacer` function is provided the value held by each key. As you can see from the method definition in [Listing 5-14](#), the `replacer` function expects to be provided with two arguments, `k` and `v`.

■ **Note** Only properties whose values are both owned by the object being traversed, in addition to being enumerable, are discovered during the iterative process.

Listing 5-14. Signature of the `replacer` Function

```
var replacerFunction = function( k, v );
```

The `k` argument will always represent the identifier (key) per object the method seeks to encode, whereas the `v` parameter represents the value held by said key.

■ **Note** If the `replacer` method is used in conjunction with an object that possesses a `toJSON` method, the value of `v` will be that of the result provided by the `toJSON` method.

The context of the `toJSON` method will always be that of the object for which it's defined. A method's scope is always tied to the object for which it exists. Contrary to methods, a function's scope is tied to that of where it was declared. However, within the `stringify` method, the scope of the `replacer` function supplied is continuously set to the context of each object whose key and value are being supplied as arguments. This means that the implicit `this` possessed by all functions will always point to the object whose keys are currently being analyzed within the `stringify` method.

Let's revisit our example from [Listing 5-9](#). However, this time, rather than define a `toJSON` that is inherited by all objects, we will supply `stringify` with a `replacer` function. As we are not concerned with customizing the default serialization of any values for the purpose of this illustration, [Listing 5-15](#) returns back to `stringify` the value, `v`, it has supplied to us.

Listing 5-15. Logging All Keys, Values, and Context with the `replacer` Function

```
var author = new Object();
    author.name = "Ben";
    author.age  = 36;
    author.pets = [
        { name : "Waverly" , age : 3.5 },
```

```

        { name : "Westley" , age : 4 }
    ];

    JSON.stringify(author,
        function(k,v){
            console.log(this);
            console.log(k);
            console.log(v);
            return v;
        });

    //the initial object wrapper being analyzed
    //(context) Object {...} //truncated
    //(key) (an empty string)
    //(value) Object { name="Ben", age=36, pets=[...]}
    //truncated
    //the author object ben property being analyzed
    //(context) Object { name="Ben", age=36, pets=[...]}
    //truncated
    //(key) name
    //(value) Ben
    //the author object age property being analyzed
    //(context) Object { name="Ben", age=36, pets=[...]}
    //truncated
    //(key) age
    //(value) 36
    //the author object pets property being analyzed
    //(context) Object { name="Ben", age=36, pets=[...]}
    //truncated
    //(key) pets
    //(value) [Object { name="Waverly", age=3.5}, Object
    { name="Westley", age=4}]
    //the pets object 0 index being analyzed
    //(context) [Object { name="Waverly", age=3.5}, Object
    { name="Westley", age=4}]
    //(key) 0
    //(value) Object { name="Waverly", age=3.5}
    //the 0 index name property being analyzed
    //(context) Object { name="Waverly", age=3.5}
    //(key) name
    //(value) Waverly
    //the 0 index age property being analyzed
    //(context) Object { name="Waverly", age=3.5}
    //(key) age
    //(value) 3.5
    //the pets object 1 index being analyzed
    //(context) [Object { name="Waverly", age=3.5}, Object
    { name="Westley", age=4}]
    //(key) 1

```

```
//(value)    Object { name="Westley", age=4}
//the 1 index name property being analyzed
//(context)  Object { name="Westley", age=4}
//(key)      name
//(value)    Westley
//the 1 index age property being analyzed
//(context)  Object { name="Westley", age=4}
//(key)      age
//(value)    4

//JSON text '{"name":"Ben","age":36,"pets":'
[{"name":"Waverly","age":3.5}, {"name":"Westley","age":4}]}'"
```

While [Listing 5-15](#) utilizes the same data structure from our `toJSON` example, in [Listing 5-9](#), you will most certainly be able to perceive that the results logged in [Listing 5-15](#) are far more plentiful. This is due to the fact that, unlike `toJSON`, the `replacer` function is triggered for each property encountered on every object.

The benefit of the keys provided to both the `replacer` function and `toJSON` is that they offer your application a means to flag a property whose value requires custom serializing. [Listing 5-16](#) demonstrates how we can leverage a supplied key to prevent a value or values from being captured in the produced JSON text.

[Listing 5-16.](#) `replacer` Function Can Be Used to Provide Custom Serializing

```
var author = new Object();
  author.name = "Ben";
  author.age = 36;
  author.pets = [
    { name : "Waverly" , age : 3.5 },
    { name : "Westley" , age : 4 }
  ];

var replacer= function(k,v){
    //if the key matches the string 'age'
    if(k=="age"){
        //remove it from the final JSON text
        return undefined;
    } //else
    return v;
}

JSON.stringify(author,replacer);
// '{"name":"Ben","pets":[{"name":"Waverly"},
{"name":"Westley"}]}'"
```

[Listing 5-16](#) leverages the uniqueness of the `age` identifier so that it can determine when to remove it from the final JSON text, by returning the value of `undefined`. While this is a valid example, it could have been equally satisfied by the `replacer` array. The takeaway is that the identifier can be extremely instrumental in the

orchestration of custom serialization.

The return value, much like in the case of `toJSON`, can be that of any value outlined in the JSON subset. The serializer will continue to ensure that your value is iteratively analyzed if returned in the form of an object, or converted into a string and sanitized, if returned as a primitive. Furthermore, the rules of [Table 5-1](#) will always apply to any and all returned values.

space

The third parameter, `space`, is also optional and allows you to specify the amount of padding that separates each value from one another within the produced JSON text. This padding provides an added layer of readability to the produced string.

The argument supplied to the parameter must be that of a whole number equal or greater to 1. Supplying a number less than 1 will have no effect on the produced JSON text. However, if the number supplied is 1 or greater, the final representation of the JSON text will display each value indented by the specified amount of whitespace from the left-hand margin. A margin is established by the inclusion of new line characters after each of the following tokens: `{`, `}`, `[`, and `]`.

In other words, new line-control characters are inserted into the produced JSON after each opening/closing token, for both an array or object. Additionally, a new line character is added after each separator token. [Listing 5-17](#) contrasts the produced JSON text with and without padding.

Listing 5-17. JSON Text with Added Padding

```
var obj={ primitive:"string", array:["a","b"]} ;  
  
JSON.stringify(obj,null,0);  
↪//(no padding)  
// {"primitive":"string","array":["a","b"]}  
  
JSON.stringify(obj,null,8);  
↪/* (8 spaces of added padding)  
"{  
    "primitive": "string",  
    "array": [  
        "a",  
        "b"  
    ]  
}"  
*/
```

The provision of the `space` parameter will have no effect on a JSON text if it does not possess either an array or object, regardless of the value specified. [Listing 5-18](#) indicates that eight spaces should be applied to the produced JSON. However, because it is not in possession of either an object or array, no padding is applied.

Listing 5-18. Space Only Works on objects and arrays

```
var primitive="string";
var JSONtext= JSON.stringify( primitive , null ,8 );
console.log( JSONtext );
// ""string""
```

The added padding appended to the final JSON text will have zero impact on its conversion back into that of a JavaScript object. Additionally, the inclusion of whitespace and new line characters will not add significant weight that would slow its transmission across the Internet.

Summary

In this chapter, we covered the JSON library, which enables JavaScript structures to become serialized for storage and data interchange. This was accomplished via downloading the JSON library and referencing the JSON global object and its `stringify` method. What you may not know is that even though we downloaded the JSON library and referenced it within our `.html` files for this chapter, the odds are you did not require it.

As I mentioned in [Chapter 4](#), JSON is incorporated within the ECMA-262, 5th edition. What this means is that any browser that aligns with ECMA 5th edition standards or greater possesses the native JSON Object as the means for both serializing and deserializing JSON. [Table 5-2](#) lists the versions of each browser that possess the JSON Object.

Table 5-2. *Minimal Browser Versions That Possess the JSON Object*

Browser	Version
FireFox	3.5+
Chrome	5+
Safari	4.0.5+
Opera	10.5+
Internet Explorer	8+

In any browser whose version is greater or equal to what is listed, you would be successful in referring to the native JSON Object. There is absolutely zero harm in incorporating the JSON library as we have, in addition to working with a browser mentioned in the preceding table. The reason for this is because the library first checks to see if a JSON Object currently exists before creating one and attaches it to the global namespace. If one is found to exist when the library is loaded into the script, it does not take any action. [Listing 5-19](#) demonstrates how if there isn't already a global JSON Object, one is created.

Listing 5-19. JSON Object is Instantiated Only if One Does Not Exist

```
if (typeof JSON !== 'object') {  
    JSON = {};  
}
```

What this means is that the library will only have an impact on browsers whose versions are below that of [Table 5-2](#). While it's becoming increasingly less likely you will continue to cater to browsers before Internet Explorer 8, some clients continue to require it.

The benefit of having you download the JSON library rather than reference the native JSON Object is that at any point during our discussion, you possess the ability to open the JSON library and review the code within, whereas you would not be as fortunate to do so with the alternative, because, being native, it's built into the browser. Therefore, there is no code to review.

What is important to remember about this chapter is that much like in the *Matrix*, knowing the rules allows you to bend the rules in your favor.

Key Points from This Chapter

- Numbers must be finite, or they are treated as `null`.
- A value that is not recognized as a valid JSON value produces the undefined value.
- A function whose name is *not* `toJSON` is ignored.
- Properties whose values are undefined are stripped.
- If the value of an array is that of `undefined`, it is treated as `null`.
- The primitive `null` is treated as the string `null`.
- A `TypeError` Exception is thrown when a structure is cyclic.
- `toJSON` and the `replacer` parameter allow applications to supply necessary logic for serialization.
- `toJSON` can be defined on any built-in object and even overridden.
- A `replacer` array identifies the properties that should be serialized.
- A `replacer` function is invoked with every property in the data structure.
- `toJSON` `this` explicitly refers to the object it's defined on.
- A `replacer` function's `this` implicitly refers to the object that is currently being analyzed.
- A key is either a property possessed by an object or the index of an array.

- Custom classes are captured as ordinary objects.

In the next chapter, you will continue to learn how we can use the JSON Object's second method, `parse`, to convert JSON back into a usable JavaScript value.

¹Microsoft, Internet Explorer Dev Center, “toString Method (Date),” <http://msdn.microsoft.com/en-us/library/ie/jj155294%28v=vs.94%29.aspx>.

²ECMA International, *ECMAScript Language Specification*, Standard ECMA-262, Edition 5.1, www.ecma-international.org/publications/files/ECMA-ST/ECma-262.pdf, June 2011.



E-next

THE NEXT LEVEL OF EDUCATION

CHAPTER 6



Parsing JSON

In the last chapter, I discussed how to convert a JavaScript value into a valid JSON text using `JSON.stringify`. In [Chapter 4](#), you learned how JSON utilizes JavaScript's literal notation as a way to capture the structure of a JavaScript value. Additionally, you learned in that same chapter that JavaScript values can be created from their literal forms. The process by which this transformation occurs is due to the parsing component within the JavaScript engine. This brings us full circle, regarding the serializing and deserializing process.

Parsing is the process of analyzing a string of symbols, either in natural language or in computer languages, according to the rules of a formal grammar. As the grammar of JSON is a subset of JavaScript, the analysis of its tokens by the parser occurs indifferently from how the Engine parses source code. Because of this, the data produced from the analysis of the JSON grammar will be that of objects, arrays, strings, and numbers. Additionally, the three literals—`true`, `false`, and `null`—are produced as well.

Within our program, we will be able to refer to any of these data structures as we would any other JavaScript value. In this chapter, you will learn of the manners by which we can convert valid JSON into usable JavaScript values within our program.

JSON.parse

In our investigation of the JSON Object, I discussed how the JSON Object possesses two methods. On one hand, there is the `stringify` method, which produces serialized JSON from a datum. And on the other hand, there is a method that is the antithesis of `stringify`. This method is known as `parse`. In a nutshell, `JSON.parse` converts serialized JSON into usable JavaScript values. The method `JSON.parse`, whose signature can be observed in [Listing 6-1](#), is available from the `json2.js` library, in addition to browsers that adhere to ECMA 5th edition specifications.

Listing 6-1. Syntax of the `JSON.parse` Method

```
JSON.parse(text [, reviver]);
```

Until Internet Explorer 7 becomes a faded memory only to be kept alive as a myth when whispered around a campfire as a horror story, it will continue to be wise to include the `json2.js` library into your projects that work with JSON. Furthermore, `json2.js` is a fantastic way to gain insight into the inner workings of the method, short of interpreting ECMA specifications.

As outlined in the preceding, `JSON.parse` can accept two parameters, `text` and

`reviver`. The name of the parameter `text` is indicative of the value it expects to receive. The parameter `reviver` is used similarly to the `replacer` parameter of `stringify`, in that it offers the ability for custom logic to be supplied for necessary parsing that would otherwise not be possible by default. As indicated in the method's signature, only the provision of `text` is required.

You will learn about the optional `reviver` parameter a bit later. First, we will begin an exploration of the `text` parameter. The aptly named parameter `text` implies the JavaScript value, which should be supplied. Of course, this is a string. However, more specifically, this parameter requires serialized JSON. This is a rather important aspect, because any invalid argument will automatically result in a `parse` error, such as that shown in [Listing 6-2](#).

Listing 6-2. Invalid JSON Grammar Throws a Syntax Error

```
var str = JSON.parse( "abc123" ); //SyntaxError: JSON.parse:
unexpected character
```

[Listing 6-2](#) throws an error because it was provided a string literal and not serialized JSON. As you may recall from [Chapter 4](#), when the sole value of a string value is serialized, its literal form is captured within an additional pair of quotes. Therefore, `"abc123"` must be escaped and wrapped with an additional set of quotation marks, as demonstrated in [Listing 6-3](#).

Listing 6-3. Valid JSON Grammar Is Successfully Parsed

```
var str = JSON.parse( "\"abc123\"" ); //valid JSON string
value
console.log(str)                      //abc123;
console.log(typeof str)               //string;
```

The JavaScript value of a parsed JSON text is returned to the caller of the method, so that it can be assigned to an identifier, as demonstrated in [Listing 6-3](#). This allows the result of the transformation to be referenced later throughout your program.

While the preceding example was supplied with a simple JSON text, it could have been a composite, such as a collection of key/value pairs or that of an ordered list. When a JSON text represents nested data structures, the transformed JavaScript value will continue to retain each nested element within a data structure commonly referred to as a *tree*. A simple explanation of a data tree can be attributed to a Wikipedia entry, which defines a tree as a nonlinear data structure that consists of a root node and, potentially, many levels of additional nodes that form a hierarchy.¹

Let's witness this with a more complex serialized structure. [Listing 6-4](#) revisits our serialized `author` object from the previous chapter and renders it into `JSON.parse`. Using Firebug in conjunction with `console.log`, we can easily view the rendered tree structure of our `author` object.

Listing 6-4. Composite Structures Create a Data Tree

```
var JSONtext= '{ "name": "Ben", "age": 36, "pets": [ { "name": "Waverly", "age": 3.5 },
```

```

{"name":"Westley","age":4}}]';
var author = JSON.parse( JSONtext );
console.log( author);

/*Firebug illustrates the parsed Data Tree of our serialized JSON text
below
    age      36
    name     "Ben"
▶ pets  [ Object { name="Waverly", age=3.5 }, Object
{ name="Westley", age=4 } ]
    ▼ 0      Object { name="Waverly", age=3.5 }
    ▼ 1      Object { name="Westley", age=4 }
*/

```

Once a JSON text is converted into that of a data tree, keys, also called members, belonging to any level of node structure are able to be referenced via the appropriate notion (i.e., dot notation/array notation). [Listing 6-5](#) references various members existing on the author object.

Listing 6-5. Members Can Be Accessed with the Appropriate Notation

```

var JSONtext= '{"name":"Ben", "age":36, "pets":
[{"name":"Waverly", "age":3.5}, {"name":"Westley", "age":4}]}';
var author = JSON.parse( JSONtext );
console.log(typeof author)      //object;
console.log(author.name)        // Ben
console.log(author.pets.length) // 2;
console.log(author.pets[0].name) // Waverly;

```

The magic of `JSON.parse` is twofold. The first proponent that allows for the transformation of JSON text into that of a JavaScript value is JSON's use of literals. As we previously discussed, the literal is how JavaScript data types can be “literally” typed within source code.

The second aspect is that of the JavaScript interpreter. It is the role of the interpreter to possess absolute understanding over the JavaScript grammar and determine how to evaluate syntax, declarations, expressions, and statements. This, of course, includes JavaScript literals. It is here that literals are read, along with any other provided source code, evaluated by the interpreter of the JavaScript language and transformed from Unicode characters into JavaScript values. The JavaScript interpreter is safely tucked away and encapsulated within the browser itself. However, the JavaScript language provides us with a not-so-secret door to the interpreter, via the global function `eval`.

eval

The `eval` function is a property of the global object and accepts an argument in the form of a string. The string supplied can represent an expression, statement, or both and will be evaluated as JavaScript code (see [Listing 6-6](#)).

Listing 6-6. eval Evaluates a String As JavaScript Code

```
eval("alert(\"hello world\")");
```

Albeit a simple example, [Listing 6-6](#) illustrates the use of `eval` to transform a string into a valid JavaScript program. In this case, our string represents a statement and is evaluated as a statement. If you were to run this program, you would see the dialog prompt appear with the text `hello world`. While this is a rather innocent program, and one created to be innocuous, you must take great caution with what you supply to `eval`, as this may not always be the case. [Listing 6-7](#) reveals that `eval` will also evaluate expressions.

Listing 6-7. eval Returns the Result of an Evaluation

```
var answer = eval("1+5");  
console.log(answer) //6;
```

The `eval` function not only evaluates the string provided, but it can also return the result of an evaluated expression so that it can be assigned to a variable and referenced by your application. Expressions needn't be mere calculations either, as demonstrated in [Listing 6-8](#). If we were to supply `eval` with a string referencing an object literal, it, too, would be evaluated as an expression and returned as a JavaScript instance that corresponds to the represented object literal.

Listing 6-8. object Literals Can Be Evaluated by the eval Function

```
var array = eval("['Waverly', 'Westley', 'Ben']");  
console.log(array[1]) //Westley;
```

Because JSON is a subset of JavaScript and possesses its own specification, it is important to always ensure that the supplied text is indeed a sequence of valid JSON grammar. Otherwise, we could be unaware of welcoming malicious code into our program. This will become more apparent when we invite JSON text into our program via Ajax. For this reason, while `eval` possesses the means to handle the transformation of JSON into JavaScript, you should never use `eval` directly. Rather, you should always rely on either the `JSON2.js` library or the built-in native JSON Object to parse your JSON text.

If you were to open the `json2.js` library and review the code within the `parse` function, you would find that the `JSON.parse` method occurs in four stages.

The first thing the method aims to accomplish, before it supplies the received string to the `eval` function, is to ensure that all necessary characters are properly escaped, preventing Unicode characters from being interpreted as line terminators, causing syntax errors. For example, [Listing 6-9](#) demonstrates that you cannot evaluate a string possessing a carriage return, as it will be viewed as an unterminated string literal.

Listing 6-9. String Literals Cannot Possess Line Breaks

```
var str="this is a sentence with a new line
```

```
... here is my new line";  
// SyntaxError: unterminated string literal  
  
// Similarly  
eval("\"this is a sentence with a new line \u000a... here is  
my new line\"");  
// SyntaxError: unterminated string literal
```

However, as stated by EMCA-262, section 7.3, line terminator characters that are preceded by an escape sequence are now allowed within a string literal token.² By escaping particular Unicode values, a line break can be evaluated within a string literal, as demonstrated in [Listing 6-10](#).

Listing 6-10. String Literals Can Only Possess Line Breaks If They Are Escaped

```
eval("\"this is a sentence with a new line \\u000a... here is  
my new line\""); //will succeed
```

The JSON library does not just ensure that Unicode characters are properly escaped before they are evaluated into JavaScript code. It also works to ensure that JSON grammar is strictly adhered to. Because JSON is simply text, its grammar can be overlooked, if it is not created via `JSON.stringify` or a similar library. Furthermore, because a string can possess any combination of Unicode characters, JavaScript operators could be easily inserted into a JSON text. If these operators were evaluated, they could be detrimental to our application, whether or not they were intended to be malicious. Consider an innocent call that has an impact on our system, as shown in [Listing 6-11](#).

Listing 6-11. Assignments Can Impact Your Existing JavaScript Values

```
var foo=123  
eval("var foo = \"abc\"");  
console.log(foo); // abc
```

Because JavaScript values can easily be overwritten, as demonstrated in [Listing 6-11](#), it is imperative that only valid JSON text is supplied to `eval`.

The second stage of the `parse` method is to ensure the validity of the grammar. With the use of regular expressions, stage two seeks out tokens that do not properly represent valid JSON grammar. It especially seeks out JavaScript tokens that could nefariously cause our application harm. Such tokens represent method invocations, denoted by an open parenthesis (`(`) and close parenthesis (`)`); object creation, indicated by the keyword `new`; and left-handed assignments, indicated by the use of the equal (`=`) operator, which could lead to the mutation of existing values. While these are explicitly searched for, if any tokens are found to be invalid, the text will not be further evaluated. Instead, the `parse` method will throw a syntax error.

However, should the provided text in fact appear to be a valid JSON, the parser will commence stage three, which is the provision of the sanitized text to the `eval` function. It is during stage three that the captured literals of each JSON value are reconstructed into their original form. Well, at least as close to their original form as JSON's grammar allows

for. Remember: JSON's grammar prohibits a variety of JavaScript values, such as the literal `undefined`, functions and methods, any nonfinite number, custom objects, and dates. That said, the `parse` method offers the ability for us to further analyze the produced JavaScript values in a fourth and final stage, so that we can control what JavaScript values are returned for use by our application. If, however, the `reviver` parameter is not supplied, the produced JavaScript value of the `eval` function is returned as is.

The final stage of the `parse` operation occurs only if we supply an argument to the method, in addition to the JSON text we seek to be transformed. The benefit of the optional parameter is that it allows us to provide the necessary logic that determines what JavaScript values are returned to our application, which otherwise would be impossible to achieve by the default behavior.

reviver

The `reviver` parameter, unlike the `replacer` parameter of the `stringify` method, can only be supplied a function. As outlined in [Listing 6-12](#), the `reviver` function will be provided with two arguments, which will assist our supplied logic in determining how to handle the appropriate JavaScript values for return. The first parameter, `k`, represents the key or index of the value being analyzed. Complementarily, the `v` parameter represents the value of said key/index.

Listing 6-12. Signature of `reviver` Function

```
var reviver = function(k,v);
```

If a `reviver` function is supplied, the JavaScript value that is returned from the global `eval` method is “walked” over, or traversed, in an iterative manner. This loop will discover each of the current object's “own” properties and will continue to traverse any and all nested structures it possesses as values. If a value is found to be a composite object, such as array or object, each key that object is in possession of will be iterated over for review. This process continues until all enumerable keys and their values have been addressed. The order in which the properties are uncovered is not indicative of how they are captured within the object literals. Instead, the order is determined by the JavaScript engine.

With each property traversed, the scope of the `reviver` function supplied is continuously set to the context of each object, whose key and value are supplied as arguments. In other words, each object whose properties are being supplied as arguments will remain the context of the implicit `this` within the `reviver` function. Last, it will be imperative for our `reviver` method to return a value for every invocation; otherwise, the JavaScript value returned will be that of `undefined`, as shown in [Listing 6-13](#).

Listing 6-13. Members Are Deleted If the Returned Value from `reviver` Is `undefined`

```
var JSONtext='{"name":"Ben","age":36,"pets":
```

```
[{"name":"Waverly","age":3.5}, {"name":"Westley","age":4}]]}';
var reviver= function(k,v){};
var author = JSON.parse( JSONtext,reviver);
console.log(author) //undefined
console.log(typeof author) //undefined
```

If the return value from the `reviver` function is found to be `undefined`, the current key for that value is deleted from the object. Specifying the supplied `v` value as the return object will have no impact on the outcome of the object structure. Therefore, if a value does not require any alterations from the default behavior, just return the supplied value, `v`, as shown in [Listing 6-14](#).

Listing 6-14. Returning the Value Supplied to the `reviver` Function Maintains the Original Value

```
var JSONtext='{"name":"Ben","age":36,"pets":
[{"name":"Waverly","age":3.5}, {"name":"Westley","age":4}]]}';
var reviver= function(k,v){ return v };
var author = JSON.parse( JSONtext,reviver);
console.log( author );
/* the result as show in firebug below
  age      36
  name     "Ben"
  ▶ pets   [ Object { name="Waverly", age=3.5 }, Object
{ name="Westley", age=4 } ]
    ▼ 0    Object { name="Waverly", age=3.5 }
    ▼ 1    Object { name="Westley", age=4 }
*/
console.log(typeof author); //object
```

As was stated earlier, a well-defined set of object keys is not only useful for your application to target appropriate data but can also provide the necessary blueprint to our `reviver` logic for clues leading to how and when to alter a provided value. The `reviver` function can use these labels as the necessary conditions to further convert the returned values of the `eval`, in order to arrive at the JavaScript structures we require for our application's purposes.

As you should be well aware at this point, JSON grammar does not capture dates as a literal but, instead, as a string literal in the UTC ISO format. As a string literal, the built-in `eval` function is unable to handle the conversion of said string into that of a JavaScript date. However, if we are able to determine that the value supplied to our `reviver` function is a string of ISO format, we could instantiate a date, supply it with our ISO-formatted string, and return a valid JavaScript `date` back to our application. Consider the example in [Listing 6-15](#).

Listing 6-15. With the `reviver` Function, ISO Date-Formatted Strings Can Be Transformed into `date` objects

```

var date= new Date("Jan 1 2015");
var stringifiedData = JSON.stringify( date );
console.log( stringifiedData ); // "2015-01-
01T05:00:00.000Z"
var dateReviver=function(k,v){
    return new Date(v);
}
var revivedDate = JSON.parse( stringifiedData
, dateReviver);
console.log( revivedDate ); //Date {Thu Jan 01 2015 00:00:00 GMT-
0500 (EST)}

```

Because the ISO date format is recognized as a standard, JavaScript dates can be initiated with the provision of an ISO-formatted string as an argument. [Listing 6-15](#) shows a program that begins with a known JavaScript `date` set to January 1, 2015. Upon its conversion to a JSON text, our date is transformed into a string made up of the ISO 8601 grammar. By supplying a `reviver` function, which possesses the necessary logic, `JSON.parse` is able to return a date to our application.

For purely illustrative purposes, [Listing 6-15](#) does not have to determine if the value supplied is in fact an ISO-formatted string. This is simply because we know the value being supplied is solely that. However, it will almost always be necessary for a `reviver` function to possess the necessary conditional logic that controls how and when to treat each supplied value.

That said, we could test any string values supplied to our `reviver` function against the ISO 8601 grammar. If the string is determined to be a successful match, it can be distinguished from an ordinary string and thus transformed into a date. Consider the example in [Listing 6-16](#).

[Listing 6-16](#). RegExp Can Match ISO-Formatted Strings

```

var book={};
    book.title = "Beginning JSON"
    book.publishDate= new Date("Jan 1 2015");
    book.publisher= "Apress";
    book.topic="JSON Data Interchange Format"

var stringifiedData = JSON.stringify( book );
console.log( stringifiedData );
// ["value held by index 0","2015-01-
01T05:00:00.000Z","value held by index 2","value held by
index 3"]

var dateReviver=function(k,v){
    var ISOregExp=/^([\+-]?[0-9]{4}(?!0000)(?!000)[0-9])(-?)((0[1-
9]|1[0-2])(\3([12]\d|0[1-9]|3[01]))?|W([0-4]\d|5[0-2])(-?[1-
7])?|((00[1-9]|0[1-9]\d|[12]\d{2}|3([0-5]\d|6[1-6])))([T\s]
(((0[1]\d|2[0-3])(?:[0-5]\d)?|24(?:00)?|(\.|\d+\d+)?)))?)?Z)?$/;

```

```

(\17[0-5]\d([\.,]\d+)?)?([zZ]|([\+-])([01]\d|2[0-3]))?:?([0-5]\d)?)?)?)?$?/;
    if(typeof v==="string"){
        if(ISOregExp.test(v)){
            return new Date(v);
        }
    }
    return v;
}
var revivedValues = JSON.parse( stringifiedData
, dateReviver);
console.log( revivedValues );
/* the result as show in firebug below
► publishDate    Date {Thu Jan 01 2015 00:00:00 GMT-0500
(EST)} ,
    publisher    "Apress",
    title        "Beginning JSON"
    topic        "JSON Data Interchange Format"
*/

```

In the preceding example, our application parses a composite structure, which is simply an array. The value of each key is in the form of a string, one of which, however, represents a date. Within the `reviver` function, we first determine if each value supplied is that of a string, via the operator `typeof`. If the value is determined to be of the `string` type, it is further compared against the ISO grammar by way of a regular expression. The variable `ISOregExp` references the pattern that matches a possible ISO-formatted string. If the pattern matches the value supplied, we know it is a string representation of a date, and, therefore, we can transform our string into a date. While the preceding example produces the desired effect, a regular expression may not prove most efficient in determining which strings should be converted and which should not.

This is where we can rely on a well-defined identifier. The `k` value, when supplied as a clearly defined label, as shown in [Listing 6-17](#), can be incredibly useful for coordinating the return of the desired object.

Listing 6-17. Well-Defined Label Identifiers Can Be Used to Establish Which objects Require Added Revival

```

var book={};
    book.title = "Beginning JSON"
    book.publishDate= new Date("Jan 1 2015");
    book.publisher= "Apress";
    book.topic="JSON Data Interchange Format"

var bookAsJSONtext = JSON.stringify(book);
console.log( bookAsJSONtext );
// '{"title":"Beginning JSON",
    "publishDate":"2015-01-01T05:00:00.000Z",

```

```

    "publisher":"Apress",
    "topic":"JSON Data Interchange Format"}"

var reviver = function( k , v ){
    console.log( k );

    /* logged keys as they were supplied to the reviver function */
    // title
    // publisher
    // date
    // publishedInfo
    // topic
    //(an empty string)

    if( k ==="publishDate"){
        return new Date( v );
    }
    return v;
}

var parsedJSON = JSON.parse( bookAsJSONtext , reviver );
console.log( parsedJSON );

/* the result as show in firebug below
▶ publishDate    Date {Thu Jan 01 2015 00:00:00 GMT-0500 (EST)}
,
  publisher      "Apress",
  title          "Beginning JSON"
  topic          "JSON Data Interchange Format"
*/

```

[Listing 6-17](#) achieves the same results as [Listing 6-16](#); however, it does not rely on a regular expression to seek out ISO-formatted dates. Instead, the `reviver` logic is programmed to revive only strings whose key explicitly matches `publishDate`.

Not only do labels offer more possibility when determining whether the value should or should not be converted, their use is also more expedient than the former method. Depending on the browser, the speeds can range from 29% to 49% slower when the determining factor is based on `RegExp`. The results can be viewed for yourself in the performance test available at <http://jsperf.com/regexp-vs-label>.

It was briefly mentioned in [Chapter 5](#) that custom classes, when serialized, are captured indistinguishably from the built-in objects of JavaScript. While this is indeed a hindrance, it is not impossible to transform your object into a custom object, by way of the `reviver` function.

[Listing 6-18](#) makes use of a custom data type labeled `Person`, which possesses three properties: `name`, `age`, and `gender`. Additionally, our `Person` data type possesses three methods to read those properties. An instance of `Person` is instantiated using the

`new` keyword and assigned to the variable `p`. Once assigned to `p`, the three properties are supplied with valid values. Using the built-in `instanceof` operator, we determine whether our instance, `p`, is of the `Person` data type, which we soon learn it is. However, once we serialize our `p` instance, and parse it back into that of a JavaScript object, we soon discover via `instanceof` that our `p` instance no longer possesses the `Person` data type.

Listing 6-18. Custom Classes Are Serialized As an Ordinary object

```
function Person(){
    this.name;
    this.age;
    this.gender;
}
Person.prototype.getName=function(){
    return this.name;
};
Person.prototype.getAge=function(){
    return this.age;
};
Person.prototype.getGender=function(){
    return this.gender;
};

var p=new Person();
    p.name="ben";
    p.age="36";
    p.gender="male";
```

```
console.log(p instanceof Person); // true
var serializedPerson=JSON.stringify(p);

var parsedJSON = JSON.parse( serializedPerson );
console.log(parsedJSON instanceof Person); // false;
```

Because the `reviver` function is invoked after a JSON text is converted back into JavaScript form, the `reviver` can be used for JavaScript alterations. This means that you can use it as a prepping station for the final object to be returned. What this means for us is that, using the `reviver` function, we can cleverly apply inheritance back to objects that we know are intended to be of a distinct data type. Let's revisit the preceding code in [Listing 6-19](#), only this time, with the knowledge that our parsed object is intended to become a `Person`.

Listing 6-19. Reviving an object's Custom Data Type with the `reviver` Function

```
function Person(){
    this.name;
    this.age;
    this.gender;
```



```

};

Person.prototype.getName=function(){
    return this.name;
};
Person.prototype.getAge=function(){
    return this.age;
};
Person.prototype.getGender=function(){
    return this.gender;
};
//instantiate new Person
var p=new Person();
    p.name="ben";
    p.age="36";
    p.gender="male";

//test that p possesses the Person Data Type
console.log(p instanceof Person); // true

var serializedPerson=JSON.stringify(p);

var reviver = function(k,v){
// if the key is an empty string we know its our top level
object
    if(k===""){
        //set object's inheritance chain to that of a Person
instance
        v.__proto__ = new Person();
    }
    return v;
}

var parsedJSON = JSON.parse( serializedPerson , reviver );

//test that parsedJSON possesses the Person Data Type
console.log(parsedJSON instanceof Person); // true
console.log(parsedJSON.getName()); // "Ben"

```

The `__proto__` property used in the preceding example forges the hierarchical relationship between two objects and informs JavaScript where to further look for properties when local values are unable to be found. The `__proto__` was originally implemented by Mozilla and has slowly become adopted by other modern browsers. Currently, it is only available in Internet Explorer version 11 and, therefore, shouldn't be used in daily applications. This demonstration is intended for illustrative purposes, to demonstrate succinctly how the `reviver` function offers you the ability to be as clever as you wish, in order to get the parsed values to conform to your application's requirements.

Summary

`JSON.parse` is the available mechanism for converting JSON text into a JavaScript value. As part of the JSON global object, it is available in modern browsers as well as older browsers, by way of including the `json2.js` library into your application. In order to convert the literals captured, `json2.js` relies on the built-in global `eval` function to access the JavaScript interpreter. While you learned that using the `eval` function is highly insecure, the JSON Object seeks out non-matching patterns of the JSON grammar throughout the supplied text, which minimizes the risk of inviting possibly malicious code into your application. If the `parse` method uncovers any tokens that seek to instantiate, mutate, or operate, a `parse` error is thrown. In addition, the `parse` method is exited, preventing the JSON text from being supplied to the `eval` function.

If the supplied text to the `parse` method is deemed suitable for `eval`, the captured literals will be interpreted by the engine and transformed into JavaScript values. However, not all objects, such as dates or custom classes, can be transformed natively. Therefore, `parse` can take an optional function that can be used to manually alter JavaScript values, as required by your application.

When you design the `replacer`, `toJSON`, and `reviver` functions, using clearly defined label identifiers will allow your application the ability to better orchestrate the revival of serialized data.

Key Points from This Chapter

- `JSON.parse` throws a `parse` error if the supplied JSON text is not valid JSON grammar.
- `parse` occurs in four stages.
- `eval` is an insecure function.
- Supply only valid JSON to `eval`.
- A `reviver` function can return any valid JavaScript value.
- If the `reviver` function returns the argument supplied for parameter `v`, the existing member remains unchanged.
- If `reviver` returns `undefined` as the new value for a member, said member is deleted.
- `reviver` manipulates JavaScript values, not JSON grammar.

¹Wikipedia, “Tree (data structure),” http://en.wikipedia.org/wiki/Tree_%28data_structure%29#Terminology, modified January 2015.

²ECMA International, *ECMAScript Language Specification*, Standard ECMA-262, Edition 5.1, Section 7.3,



E-next

THE NEXT LEVEL OF EDUCATION

CHAPTER 7



Persisting JSON: I

In [Chapter 5](#), you learned how `JSON.stringify` captures the data possessed by an identified JavaScript value. This occurs by reverse engineering the specified target into its literal form, in accordance with the JSON grammar, thus capturing the current state of a model for a particular application as JSON text. You further learned that `JSON.parse` taps into the innate ability of the JavaScript engine to “parse” the literals that make up a valid JSON text. This revives the state from a previous model for use within the existing session.

To illustrate how to use `JSON.parse`, each example in [Chapter 6](#) was preceded by the `stringify` method, in order to provide something to be parsed. Furthermore, this was meant to illustrate the lifecycle of how one method gives rise to the other.

While this is sufficient for the purposes of a demonstration, it will be rare to parse data immediately after it has been serialized by our application. This would result in a very linear and limited use case. These two methods really shine, however, when they are paired with data persistence. It is the persistence of data that enables both methods, `stringify` and `parse`, to be used independently of each other. This offers an application many more use-case scenarios. This contrast is illustrated in [Figure 7-1](#).

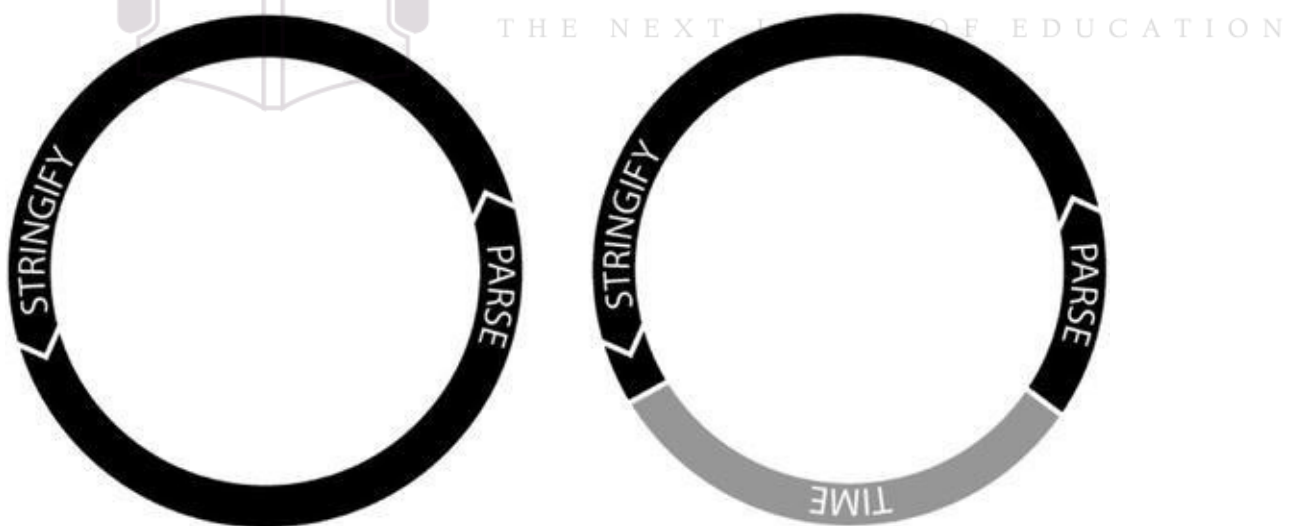


Figure 7-1. Contrast between use-case scenarios

Computer science defines the persistence of data as a state that continues to exist after the process from which it was created.¹ Much like the phrase, “you can’t step in the same spot of a moving river twice,” the process that serializes data will cease to exist the moment the JSON text is produced and the function that ran the process is exited. Therefore, in order to utilize the produced JSON beyond the given process that created it, it must be stored for later retrieval.

Believe it or not, in the examples in [Chapter 5](#), we were using a slight form of data

persistence, according to the aforementioned definition. When the `stringify` method exited, the produced JSON returned by each example was able to continue to be referenced by the application. This is because we had assigned it as the value to a variable, which was often labeled `JSONtext`. Therefore, we managed to persist JSON by definition. However, if we were to navigate away from the application at any point in the course of running the [Chapter 5](#) examples within a browser, the variable `JSONtext` would cease to persist, and the JSON it was assigned would be lost as well.

Because the Internet was founded atop a request-and-response protocol, each request made of a server, regardless of whether it's for `.html`, `.jpg`, `.js`, etc., occurs without consideration of any previous or subsequent requests by the same visitor. This is even if requests made are to the same domain. What is returned from the server is simply the fulfillment of the resource requested. Over the years, many a developer has needed to be able to string together the isolated requests of a common server, in order to facilitate things such as shopping carts for e-commerce. One of the technologies that was forged from this requirement brought forth a technique that we will leverage in order to achieve the persistence of JSON. That technology is the HTTP cookie.

HTTP Cookie

As was previously mentioned, the HTTP/1.1 protocol is incapable of persisting state; therefore, it becomes the duty of the user-agent to manage this undertaking. The HTTP cookie, or cookie for short, was created as a means to string together the actions taken by the user per “isolated” request and provide a convenient way to persist the state of one page into that of another. The cookie is simply a chunk of data that the browser has been notified to retain. Furthermore, the browser will have to supply, per subsequent request, the retained cookie to the server for the domain that set it, thereby providing state to a stateless protocol.

The cookie can be utilized on the client side of an application with JavaScript. Additionally, it is available to the server, supplied within the header of each request made by the browser. The header can be parsed for any cookies and made available to server-side code. Cookies provide both front-end and back-end technologies the ability to collaborate and reflect the captured state, in order to properly handle each page view or request accordingly. The ability to continue to progress the state from one page to another allows each action to no longer be isolated and, instead, occur within the entirety of the user's interaction with a web site.²

Like JSON, cookies possess a specification and protocol all their own. By understanding its syntax, we can tap into the persistence of the HTTP cookie and, by extension, persist JSON for later use with an application. The great news is that HTTP cookies are extremely simple, in addition to being recognized by all major browsers dating back to Internet Explorer 3.

Syntax

At its most atomic unit, the cookie is simply a string of ASCII encoded characters composed of one or more attribute-value pairs, separated by a semicolon (;) token.

[Listing 7-1](#) outlines the syntax for the HTTP cookie.

■ **Note** ASCII is short for “American Standard Code for Information Interchange” and is composed of 128 characters, which are letters from the English alphabet, digits 0–9, basic punctuation, and a few control characters.

Listing 7-1. Set-Cookie Syntax as Defined by RFC 6265

```
set-cookie      =      "Set-Cookie:" cookies
cookies         =      1#cookie
cookie          =      NAME "=" VALUE *("; " cookie-av)
NAME            =      attr
VALUE           =      value
cookie-av       =      "expires" "=" value
                |      "max-age" "=" value
                |      "domain" "=" value
                |      "path" "=" value
                |      "secure"
                |      "httponly"
```

[Listing 7-1](#) uses the grammar defined by the HTTP/1.1 specification to outline the syntax of the HTTP cookie. In order to understand the syntax, I would like to direct your focus to the line `cookie = NAME "=" VALUE *("; " cookie-av)`. This line outlines the entire syntax of the cookie. We will dissect this line in two passes. The first half will regard only `cookie = NAME "=" VALUE`. This portion of the syntax outlines the following: “Set some cookie specified by the indicated NAME, to possess the assigned VALUE.” A cookie, in short, is nothing more than a key/value pair.

As with all key/value pairs, it will be the purpose of the “key” represented by NAME to both identify as well as provide the means to access an assigned value. VALUE, on the other hand, represents the data or state that’s intended to be persisted for the application. To ensure a cookie is stored uniformly among all browsers, it will be imperative that both NAME and VALUE be made up of valid ASCII characters, such as those shown in [Listing 7-2](#).

Listing 7-2. Key/Value Pairs Intended to Be Persisted As a Cookie Must Both Be Valid ASCII Characters

```
"greetings=Hello World!";
"greetingJSON=[\"Hello World!\"]";
```

■ **Note** Safari as well as Internet Explorer do not correctly handle cookies that contain non-ASCII characters.

While the tokens that make up JSON text are valid ASCII characters, the values held

within are not limited to ASCII but, rather, UTF-8. Therefore, if the characters that are represented in your application fall outside of the ASCII range, it will be necessary to encode your UTF-8 characters with Base64 encoding. Two libraries you can use for this purpose are <https://jsbase64.codeplex.com/releases/view/89265> and <https://code.google.com/p/javascriptbase64/>. While both utilize different namespaces, Base64 and B64, they both rely on the same methods to encode and decode. Either of these libraries will be capable of converting your non-ASCII values into ASCII-encoded values. [Listing 7-3](#) demonstrates the use of one of the aforementioned Base64 libraries by converting the characters of our string of UTF-8 characters into those of ASCII, in order to be compliant with the HTTP cookie syntax.

Listing 7-3. UTF-8 Characters Being Converted into ASCII Using a Base64 Library

```
var unicodeValue = "привет мир!"; // Hello World! in Russian;
var asciiString = Base64.encode( JSON.stringify(
unicodeValue ) );
console.log(asciiString); //
"ItC/0YDQuNCy0LXRgidQvNC40YAhIg=="
var decodedValue = Base64.decode( asciiString );
console.log( decodedValue ); // "привет мир!"
```

The second half of the line in review, *("; " cookie-av), explains that our cookie can be supplied a sequence of any of the six optional cookie attribute-value pairs, as required by an application. The token that must separate them from their successor in the string is the semicolon (;). While it is not necessary to supply whitespace characters between the semicolon and the attribute value, it will aid to keep your code clean and legible. The possible cookie-av values are listed in [Listing 7-1](#) as “expires”, “max-age”, “domain”, “path”, “secure”, and “httponly”. Each attribute value defines a specific scope to the defined cookie.

expires

The expires attribute is quite literally the “key,” pun intended, to the duration over the persistence of the specified cookie. Should the expires attribute be specified, its value counterpart will inform the browser of the date and time it is no longer necessary to further store said cookie. The value supplied is required to be in UTC Greenwich Mean Time format. Being that UTC GMT is a standard, we can achieve this value with ease, by way of the built-in methods of the Date object as demonstrated in [Listing 7-4](#).

Listing 7-4. toUTCString Produces a UTC Greenwich Mean Time Value

```
var date= new Date("Jan 1 2015 12:00 AM");
var UTCdate= date.toUTCString() ;
console.log( UTCdate ); // "Thu, 01 Jan 2015 06:00:00 GMT"
```

[Listing 7-4](#) initiates a date instance with the supplied string of January 1, 2015. Furthermore, the time is set to exactly 12 AM. Utilizing date’s built-in method, toUTCString, the date and time it represents is translated into its GMT equivalent and

then returned to the caller of the method. When we log that value, we can clearly note that the date has been converted, as it is signified by the appended abbreviation **GMT**. If you were to run the code from [Listing 7-4](#), you might receive a different value. That is because the JavaScript Date Object correlates to your location and time zone. Nevertheless, the date and time that you specify will be equal to the difference in time zone between your location and Greenwich.

If we were to assign the date from [Listing 7-4](#) to our author cookie in [Listing 7-5](#), the cookie would be available until exactly Thursday, 12:00 AM January 1, 2015, or Thursday, 01 Jan 2015 06:00:00 Greenwich Mean Time.

Listing 7-5. Appending Date to the Key/Value Pair to Provide an Expiration

```
var date= new Date("Jan 1 2015 12:00 AM");  
"author=test; expires="+ date.toUTCString();
```

If the value supplied to the `expires` attribute occurs in the past, the cookie is immediately purged from memory. On the other hand, if the `expires` attribute is omitted, then the cookie will be discarded the moment the session has ended. Essentially, the browser would continue to persist the cookie only as long as the session remained open.

It used to be that the moment you exited the browser, all sessions were immediately closed. Today, however, it's worth noting that sessions may persist well after the browser is exited. This is due to the specific features that vendors have incorporated into their browsers, such as restoring previously viewed pages/tabs if the browser crashes. Additionally, they provide us the ability to restore pages/tabs from History. Therefore, session cookies may continue to persist in memory longer than expected.

As we will be looking to persist our JSON indefinitely, we will almost always supply an `expires` attribute value to our cookies.

max-age

The `max-age` attribute, like the `expires` attribute, specifies how long a cookie should persist. The difference between the two is that `max-age` specifies the life span of the cookie in seconds. While the `max-age` attribute is defined by the original specification and continues to exist today, it is not an attribute that is acknowledged by Internet Explorer 6 through 8. That said, it will be wise to favor the `expires` attribute and ignore `max-age`.

domain

The `domain` attribute explicitly defines the domain(s) to which the cookie is to be made available. However, the domain specified must somehow possess a relationship to the origin setting the cookie. In other words, if www.sandboxed.guru is setting a cookie, it cannot supply apress.com as the domain. This would prove to be a huge security concern, if it were possible to set cookies for other domains.

It is the responsibility of the browser to make available, to both JavaScript and the

server, all cookies whose supplied `domain` attribute matches that of the domain of the visited URL. To ensure that the domains match, the browser will compare the two. This comparison can be illustrated with a regular expression (see [Listing 7-6](#)).

Listing 7-6. Using a Regular Expression to Demonstrate Matching Origins

```
var regExp=  
(/www.sandboxed.guru$/i).test('www.sandboxed.guru'); //true
```

[Listing 7-6](#) defines a pattern that matches against the tail end of a host domain. The pattern `www.sandboxed.guru` represents the cookie's assigned `domain` attribute. The `$` token further specifies that the pattern explicitly ends with `.guru`. This is necessary to prevent the cookies of `sandboxed.guru` from being available to another domain that might just so happen to possess our origin within its subdomain. This would be quite the security risk. Note the difference between the URLs `sandboxed.guru` and `guru.com`. They are two entirely different domains. Now consider what might occur if `guru.com` were to use the following subdomain: `www.sandboxed.guru.com` (see [Listing 7-7](#)).

Listing 7-7. Matching URLs are Determined Through the Top Level Domain (`.com`)

```
(/sandboxed.guru/i).test('sandboxed.guru.com'); //true  
(/sandboxed.guru$/i).test('sandboxed.guru.com'); //false
```

[Listing 7-7](#) demonstrates that without specifying the `$` to force a tail-end match, two completely different properties could potentially be considered a match.

■ **Note** To prevent possible matches that could exist within subdomains, browsers explicitly check that a match must end with the appropriate top-level domain.

The `i` simply informs the pattern to remain case-insensitive during the match. If the `domain` attribute and the server domain are determined to be a match, then for each HTTP request, any and all cookies will be sent to the server and made available to the JavaScript application of each page.

The `domain` attribute is optional, but for security purposes, one must be set. By default, the `domain` attribute will be set to the absolute origin that the cookie is set from. This can be slightly limiting if you have subdomains that require visibility of these cookies, or vice versa. Consider a `domain` attribute that is defaulted to `www.sandboxed.guru` for a particular cookie. That cookie will never be available to `sandboxed.guru` because of the preceding `www`. Similarly, if the `domain` attribute is defaulted to `sandboxed.guru`, that cookie will not be visible to `json.sandboxed.guru`.

However, by assigning the `domain` attribute value, we have the ability to broaden the scope of our cookies. For instance, if we specify a `domain` attribute as the top-level domain, preceded by the `.` token (`.sandboxed.guru`), the `domain` attribute would match not only a top-level domain (`sandboxed.guru`) but any and all subdomains as well (`json.sandboxed.guru`). This is demonstrated in [Table 7-1](#).

Table 7-1. *Illustrating Which Origins Are Considered Matches Against the Value Possessed by the domain Attribute*

domain Attribute	Origin	Match
www.sandboxed.guru	sandboxed.guru	false
sandboxed.guru	www.sandboxed.guru	false
.sandboxed.guru	sandboxed.guru	true
.sandboxed.guru	www.sandboxed.guru	true
.sandboxed.guru	json.sandboxed.guru	true

It is not necessary to apply the . token. As long as we explicitly specify a hostname for the domain attribute, the . token will automatically be prepended to all non-fully-qualified domains by the user agent.

path

While the domain attribute specifies to which domain(s) a set cookie is scoped, the path attribute further enforces to which subdirectories a cookie is available. If a path attribute is not explicitly specified, the value is defaulted to the current directory that set the cookie. Furthermore, every subdirectory of the defaulted directory will be provided access. However, explicitly defining the path attribute allows us to narrow or broaden the scope of the cookie to that of a particular directory and all of its subdirectories. [Listing 7-8](#) demonstrates how cookies can further scope a cookie to that of a particular URL for any domain that is deemed a potential match.

Listing 7-8. Demonstrating Path Scoping with Cookies Set from <http://json.sandboxed.guru/chapter7/fictitious.html>

```
"cookieDefault=test; domain=.sandboxed.guru";
  http://json.sandboxed.guru/chapter7/           //cookieDefault is
provided for this request
  http://json.sandboxed.guru/chapter7/css/        //cookieDefault is
provided for this request
  https://www.sandboxed.guru/                     //cookieDefault is NOT
provided for this request
  http://json.sandboxed.guru/chapter3/js/         //cookieDefault is NOT
provided for this request
  https://json.sandboxed.guru/chapter3/img/       //cookieDefault is NOT
provided for this request

"cookieA=test; domain=.sandboxed.guru; path="/";
  http://json.sandboxed.guru/chapter7/           //cookieA is
provided for this request
  https://www.sandboxed.guru/                     //cookieA is
provided for this request
```

```
http://json.sandboxed.guru/chapter3/js/    //cookieA is
provided for this request
https://json.sandboxed.guru/chapter3/img/  //cookieA is
provided for this request

"cookieB=test; domain=.sandboxed.guru; path=chapter3/js/";
http://json.sandboxed.guru/chapter7/      //cookieB is NOT
provided for this request
http://json.sandboxed.guru/               //cookieB is NOT
provided for this request
https://json.sandboxed.guru/chapter3/js/  //cookieB is
provided for this request
https://json.sandboxed.guru/chapter3/     //cookieB is NOT
provided for this request
```

■ **Note** Cookies that are scoped to a particular domain and/or path are able to be used indistinguishably by HTTP and HTTPS protocols.

secure

The `secure` attribute is slightly misleading, as it does not provide security. Rather, this attribute, which does not require being assigned a variable, informs the browser to send the cookie to the server only if the connection over which it is to be sent is a secure connection, such as HTTPS. Transmitting data over a secure transport reduces the ability for any network hijackers to view the contents being transported. This helps to ensure that the cookie remains concealed from possible snoopers. While this flag ensures that a cookie's value remains hidden from an attacker, it does not prevent the cookie from being overwritten or even deleted by an attacker.

httponly

The `httponly` attribute, when specified, limits the availability of the cookie to the server and the server alone. This means the cookie will not be available to the client side, thereby preventing client-side JavaScript from referencing, deleting, or updating the cookie. This `httponly` flag, when used in conjunction with the `secure` flag, helps to reduce cross-site scripting from exploiting the cookie. As this chapter is focused on the persistence of JSON data from a client-side perspective, we will be avoiding this attribute.

■ **Note** Cookies set with the `httponly` flag can only be set by the server.

When specifying any of the preceding attribute-value pairs, there is no particular order in which they must be specified. Furthermore, each is case-insensitive and can appear in lowercase or uppercase form.

document.cookie

A cookie can be created by a server, server-side code, HTML meta tags, and even JavaScript. In this chapter, we will solely be focused on the creation and the retrieval of cookies by way of the JavaScript language. Up until now, we have been equating a particular syntax of string as the representative for a cookie. The reality is that it is not a cookie until we supply it to our document.

The Document Object Model, or DOM for short, can be referenced via the document object in JavaScript. This document object possesses a variety of interfaces that allows us to manipulate HTML elements and more. One interface on which we will be focusing is the appropriately named document.cookie interface. The cookie attribute of the document object is responsible for supplying the browser with a provided string of name/value pairs, enabling the persistence of said key/value pairs. Additionally, this property acts as the interface for their retrieval from the document. [Listing 7-9](#) uses document.cookie to create our first cookie.

Listing 7-9. Supplying Our First Key/Value Pair to document.cookie in Order to Become a Cookie

```
document.cookie= "ourFirstCookie=abc123";
```

While it appears in [Listing 7-9](#) that we are assigning a string to the cookie property, in actuality we are providing a string as the argument to a setter method. A setter method is a method that is used to control changes to a variable.³ Behind the scenes, the document receives the value being assigned and treats it as an argument to an internal method, which immediately sets the assignment as the value to be stored within an internal collection. This collection, which has come to be referred to as the cookie jar, is stored in a file that is available only to the browser that stores it. Because each browser sets cookies within its cookie jar, cookies are only available to the browser that is used at the time they are set.

As we are not truly assigning a value to the document.cookie property, we can add any number of name/value pairs to document.cookie, without fear that we will overwrite what we had previously set as a cookie, as seen in [Listing 7-10](#).

Listing 7-10. Subsequent Assignments to document.cookie

```
document.cookie= "ourFirstCookie=abc123";  
document.cookie= "ourSecondCookie=doeRayMe";  
document.cookie= "ourThirdCookie=faSoLaTeaDoe";
```

As I stated earlier, the name/value pairs are not being overridden with each new assignment. All name/value pairs assigned to document.cookie are not held as the value of cookie but, rather, stored safely within the cookie jar. The cookie jar is simply a resource located on the file system of the user's computer, which is why cookies have the ability to persist.

In order to view all cookies on your machine, follow the outlined steps for the modern browser of your choice.

For Chrome:

1. Open Chrome.
2. Navigate your browser to `chrome://settings/cookies`.
3. Click any site to view all cookies for that particular site.

For Firefox:

1. Open Firefox.
2. From the Firefox menu, select Preferences.
3. Click the Privacy tab.
4. Click the linked words “remove individual cookies.”
5. Click any site to view all cookies for that particular site.

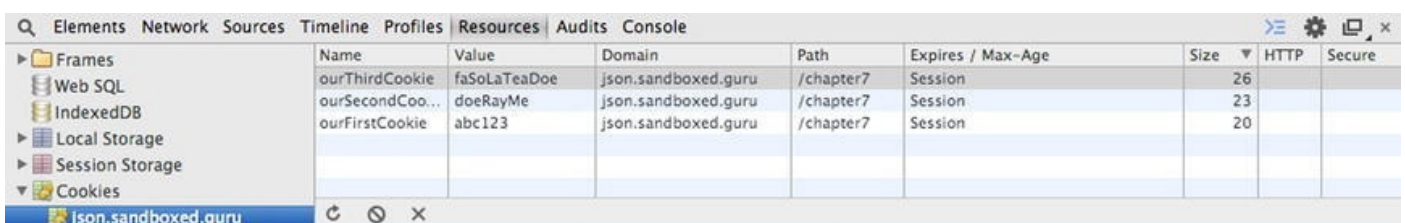
For Safari:

1. From the Safari menu, select Preferences.
2. In the preferences window, select Privacy.
3. In the Privacy window, click Details. (Unfortunately, with Safari, you can only see what sites have set cookies. You won’t be able to view full details.)

For Internet Explorer:

1. Open Internet Explorer.
2. From the Tools menu (the gear icon), select Internet Options.
3. On the General tab, within the section “Browser History,” select Settings.
4. From the Settings panel, click “View objects” or “View Files.”

If you only care to view the cookies that are available to the sites you are currently viewing, this can easily be achieved by way of the developer console. Utilizing the developer’s tools provided by a modern browser, we can easily witness the cookies we have created thus far. [Figure 7-2](#) displays the stored cookies of [Listing 7-10](#), by way of the developer tools provided by Chrome Version 35.0.1916.114.



Name	Value	Domain	Path	Expires / Max-Age	Size	HTTP	Secure
ourThirdCookie	faSoLaTeaDoe	json.sandboxed.guru	/chapter7	Session	26		
ourSecondCookie	doeRayMe	json.sandboxed.guru	/chapter7	Session	23		
ourFirstCookie	abc123	json.sandboxed.guru	/chapter7	Session	20		

Figure 7-2. Chrome’s Developer Tools Console displays the cookies for the currently visited URL `json.sandboxed.guru/chapter7/`

As you can note from the Name column in [Figure 7-2](#), each cookie has, in fact, been stored rather than overwritten. Furthermore, you can see what values are set for each

optional `cookie-av`, as follows:

Domain: `json.sandboxed.guru`
Path: `/chapter7`
Expires: `Session`

As you may recall, [Listing 7-10](#) merely supplied the name/value pair and did not append any optional cookie attribute values. However, the `domain`, `path`, and `expires` attributes are required of the cookie. Therefore, the values supplied, as shown in [Figure 7-2](#), have been set to their defaulted values.

As discussed earlier, both the `domain` and `path` attribute values are defaulted to the respective aspects of the URL from which a cookie is set. The `domain` attribute, which is set to `json.sandboxed.guru`, clearly identifies the domain name from which the application ran. Furthermore, the path set to `/chapter7` is a reflection of the directory from which the resource set the preceding cookies.

■ **Note** The preceding results reflect the cookies set from the following URL: <http://json.sandboxed.guru/chapter7/7-7.html>.

Last, the `expires` attribute is defaulted to a session, which means that the moment the session ends, the browser is no longer required to store the cookie further. In order to provide a level of control over the cookie attribute values, we must append them as required by the syntax of the HTTP cookie. This can be done easily by devising a function to handle this, as portrayed in [Listing 7-11](#).

Listing 7-11. The `setCookie` Function Simplifies the Creation of HTTP Cookie Values

```
function setCookie(name, value, expires, path, domain,
secure, httpOnly) {
    document.cookie = name + "=" + value
    //if expires is not null append the specified
GMT date
    + ((expires)? "; expires="
+ expires.toUTCString() : "")
    //if path is not null append the specified path
    + ((path) ? "; path=" + path : "")
    //if domain is not null append the specified
domain
    + ((domain) ? "; domain=" + domain : "")
    //if secure is not null provide the secure Flag
to the cookie
    + ((secure) ? "; secure" : "");
};
```

The function `setCookie` within [Listing 7-11](#) provides us with a simple means to create a cookie, by supplying the necessary arguments for each `cookie-av` parameter. For each value that you wish to override, the function `setCookie` may be supplied with

the appropriate string value. That is, except for the `expires` attribute, which requires a date. For any optional cookie attribute value that you wish to omit, you can simply provide the `null` primitive. This is demonstrated in [Listing 7-12](#).

Each line within the `setCookie` function relies on what is known as a tertiary operator to determine whether an empty string or a supplied value is to be appended to the cookie. A tertiary operator, which is simply a condensed `if ... else` statement determines if a parameter has been provided an argument to append. If the parameter has not been supplied an argument, an empty string is assigned as the value for the specified cookie attribute.

■ **Note** It is the responsibility of the user-agent to set values for any attribute value that is not valid. Attributes that possess empty strings will be replaced with a default value.

Listing 7-12. The Function `setCookie` Has Been Created to Help in the Provision of Cookie Attribute Values

```
setCookie("ourFourthCookie",           //name
          "That would bring us back to Doe", //value
          new Date("Jan 1 2016 12:00 AM"), //expires
          "/",                          //path
          null);                        //secure
```

[Listing 7-12](#) utilizes the `setCookie` function to create a cookie that will persist until January 1, 2016. The attribute's values can be viewed within the cookie jar, as demonstrated within the Developer Tools Console, as shown in [Figure 7-3](#).

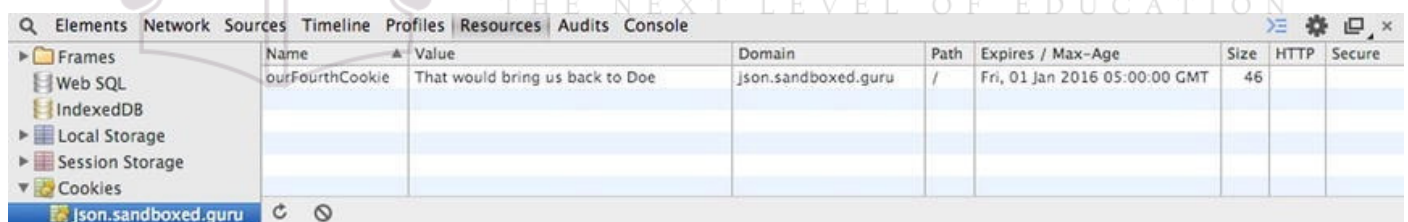


Figure 7-3. Developer Tools Console displaying the configured cookie attribute values for the currently viewed URL

While `document.cookie` is the entry point to the method that controls the storage of cookies, it can also be used to obtain the many name/value pairs that have been stored, provided their `domain` attribute matches the domain from which they are being requested. In order to read from the cookie jar, we simply reference the `cookie` property of the document, without providing it an assignment, as demonstrated in [Listing 7-13](#).

Listing 7-13. Retrieving All Persisted Cookies for the Scoped Origin and Path via `document.cookie`

```
console.log(document.cookie); // "ourFourthCookie=That would
bring us back to Doe"
```

The code within [Listing 7-13](#) simply logs out the returned value from `document.cookie` and sends it to the console for inspection. What is outputted is the name/value pair that has continued to persist. This is assuming you are running this code

prior to January 1, 2016. Otherwise, because the `expires` attribute would be explicitly set to a date that occurred in the past, it would be removed from memory, and nothing would appear.

■ **Note** Running the preceding code after January 1, 2016, 12:00 AM would inform the browser that it no longer is required to store the cookie.

What you may recognize immediately is that the product returned from `document` remains unaltered from what we initially supplied in [Listing 7-12](#). Unfortunately, `document` neither separates the supplied key from its assigned value for ease of use, nor does the document possess a method that can separate them for us. Therefore, in order to extract the value from the string returned, we will have to separate the value from the key ourselves. [Listing 7-14](#) accomplishes this with simple string manipulation.

Listing 7-14. Separating the Value from the Supplied Key from a Singularly Returned Cookie

```
1  var returnedCookie = "ourFourthCookie=That would bring
us back to Doe";
2  //15 characters in is the = sign
3  var seperatorIndex = returnedCookie.indexOf("=");
4
5  //extract the first 15 characters
6  var cookieName =
returnedCookie.substring(0,seperatorIndex);
7
8  //extract all characters after the '=' 15th character
9  var cookieValue
= returnedCookie.substring(seperatorIndex+1,
returnedCookie.length);
10
11 console.log(cookieName); //"ourFourthCookie"
12 console.log(cookieValue); //"That would bring us back to
Doe"
```

[Listing 7-14](#) begins by searching for the first occurrence of the equal (=) token (**line 3**), as that is the token that separates the key from its value. Once this index is made known, we can consider everything up to that index the “key” and everything beyond it the “value.” Utilizing the implicit method of the String Object, we can extract a sequence of characters within a numeric range. We begin with the range of characters from 0 up to the 15th character being the = token for **Name (line 6)**. The next set of characters, which begins at the 16th character, ranges through the remaining characters of the string, thus successfully extracting the value.

You may also notice that the string returned does not supply us with any of the attribute-value pairs that it was initially assigned. This is strictly due to the fact that the `cookie-av` values are intended to be utilized by the browser alone. It is the browser’s job to ensure that cookies are being supplied to the necessary domain, path, and over the

proper transport protocol. Our application merely requires informing the browser, at the moment the cookie is set, how it is necessary to handle the storage and access to the cookie.

While [Listing 7-14](#) outputted only one cookie, this will not always be the case. In the event that numerous cookies are stored and requested from that of a matching origin/path, each persistently stored cookie will be concatenated and returned by the document. Each name/value pair is separated from another by way of the semicolon (;) token, as demonstrated in [Listing 7-15](#).

Listing 7-15. Multiple Cookies Are Concatenated and Delimited by a Semicolon (;)

```
setCookie("ourFourthCookie",
    "That would bring us back to Doe",
    new Date("Jan 1 2016 12:00 AM"), "/", null, null);

setCookie("ourFifthCookie",
    "Doe a dear a female dear",
    new Date("Jan 1 2016 12:00 AM"), "/", null, null);

console.log(document.cookie);
// "ourFifthCookie=Doe a dear a female dear;
ourFourthCookie=That would bring us back to Doe"
```

By identifying the tokens of the grammar that make up the cookie syntax, we can separate the name/value pairs from one another. Additionally, we can separate the value from the specified name. This can be achieved by searching the provided string for the semicolon (;) and equal sign (=) tokens.

Listing 7-16. Extracting the Value from a Specified Key Among Many

```
1 function getCookie(name) {
2     var regExp = new RegExp(name + "=[^\\;]*", "mgi");
3     var matchingValue = (document.cookie).match(regExp);
4     console.log( matchingValue )    //
"ourFourthCookie=That would bring us back to Doe"
5     for(var key in matchingValue){
6         var
replacedValue=matchingValue[key].replace(name+"=", "");
7         matchingValue[key]=replacedValue;
8     }
9     return matchingValue;
10 };
11 getCookie("ourFourthCookie"); // ["That would bring us
back to Doe"]
```

The function `getCookie` within [Listing 7-16](#) utilizes a regular expression to seek out any name/value pairs from the string returned by `document.cookie`. The pattern `name+"=[^\\;]*"`, as highlighted on **line 2**, defines a pattern to match all sequences of

characters within a string that is found to possess a specified name immediately followed by the = token. From there, any valid ASCII character is considered to be a match, as long as that character is not a semicolon (;) token. Should the string returned by the `document.cookie` possess any sequences of characters that match this pattern, they are captured, respectively, within an array and returned for reference (**line 3**).

At this point, if a match has been made, what will be indexed within the returned array are the name/value pairs that match the cookie name supplied to the method. If we were to log out the results found within the array at this point, we should view the following: "ourFourthCookie=That would bring us back to Doe" (**line 4**). In order to separate the value from Name and the equal sign, we iterate over all matched occurrences and replace the found name and = token with those of an empty string (**line 6**), thereby exposing the value. The value is then reassigned back to the key to which it is referenced within the `matchingValue` array (**line 7**). Last, the `getCookie` function returns the array of all found values (**line 9**).

Thus far, you have learned how to successfully write and store persistent values by way of HTTP cookies. Utilizing our new functions, `setCookie` and `getCookie`, let's revisit the Person object from the previous chapter and store its serialized JSON text within a cookie (see [Listing 7-17](#)).

Listing 7-17. Pairing the JSON Object and the Cookie to Store objects

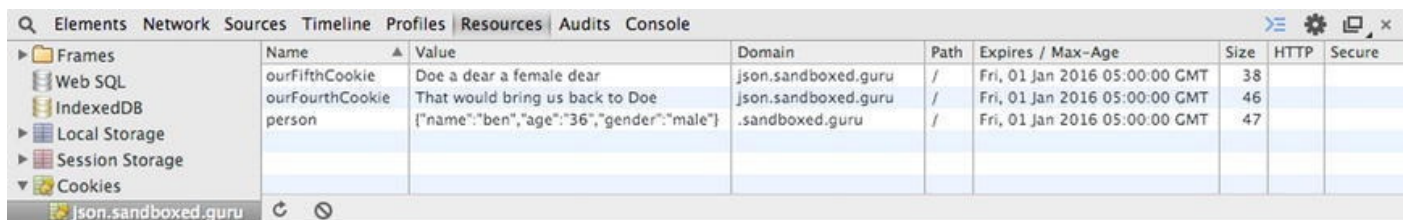
```
1 function Person() {
2     this.name;
3     this.age;
4     this.gender;
5 };
6 Person.prototype.getName = function() {
7     return this.name;
8 };
9 Person.prototype.getAge = function() {
10    return this.age;
11 };
12 Person.prototype.getGender = function() {
13    return this.gender;
14 };
15
16 //instantiate new Person
17 var p = new Person();
18     p.name = "ben";
19     p.age = "36";
20     p.gender = "male";
21
22 var serializedPerson = JSON.stringify(p);
23 setCookie("person", serializedPerson, new Date("Jan
24 1 2016"), "/", "sandboxed.guru", null);
25 console.log( getCookie( "person" )); "
```



```
{"name": "ben", "age": "36", "gender": "male"}
```

Running the preceding code within a browser will create a cookie, as previously, only this time, the cookie created possesses JSON as the supplied value. Also as before, by opening up the developer consoles provided by modern browsers, we can view all stored cookies within the cookie jar for the current origin.

As you can clearly see from [Figure 7-4](#), our `person` cookie, like the others, has been added to the cookie jar. It will remain available to all JavaScript code from within any directory of the scoped domain `sandboxed.guru`, as well as any and all subdomains.



Name	Value	Domain	Path	Expires / Max-Age	Size	HTTP	Secure
ourFifthCookie	Doe a dear a female dear	json.sandboxed.guru	/	Fri, 01 Jan 2016 05:00:00 GMT	38		
ourFourthCookie	That would bring us back to Doe	json.sandboxed.guru	/	Fri, 01 Jan 2016 05:00:00 GMT	46		
person	{"name": "ben", "age": "36", "gender": "male"}	.sandboxed.guru	/	Fri, 01 Jan 2016 05:00:00 GMT	47		

Figure 7-4. Developer console exhibiting the persistence of our `person` cookie and its JSON value

To further illustrate this point, simply navigate to <http://json.sandboxed.guru/chapter7/cookie-test.html> and create your own `person` cookie to store. After you submit your cookie to the document, either refresh the page to find the `person` column populated or navigate to <http://sandboxed.guru/cookie-test.html> to find that this top-level domain has access to your new `person` cookie. Now hit Delete, to remove the persisted cookie, and generate another, this time with different data. Once more, visit the subdomain <http://json.sandboxed.guru/chapter7/cookie-test.html>, and you will see that new cookie pre-populated.

For all of its benefits, the cookie does come with a few limitations. Sadly, the cookie can only store a maximum amount of bytes. In fact, it can only store roughly 4KB, which would be roughly 4,000 ASCII characters. While 4,000 characters is a lot, it can add up quickly, depending on what you are storing. Furthermore, Base64 characters can require up to three times more bytes per character than ASCII.

You learned that `document.cookie` does not provide any information beyond the stored name/value pair. This is problematic, because there is no way to truly know how many bytes are available to us. Another issue that cookies face is that they are scoped to the browser, which means that the preserved state is only available to the specific browser that preserves it. Last, because the cookie was originally crafted to help maintain a visitation between a server and a browser, cookies are automatically sent with every request made to the server that possesses the allowed origin by the cookie. The issue here is that the more cookies that are used, each occupying x number of bytes is sent to the server with every single request. Essentially, unless your server is utilizing the cookie, you are needlessly transmitting 4KB for each cookie stored for every request.

While the cookie has its advantages, it is also archaic. It was just a matter of time before another front-end technology came along. That tool is HTML 5's Web Storage.

Web Storage

HTML5 introduced the concept of Web Storage to pick up where the cookie had left off. While Web Storage may be considered to be the HTTP cookie successor, it would simply be a matter of the context in which you can make that statement. A better way to view Web Storage is simply to look at it as cookies' counterpart. Its creation is not necessarily to replace the cookie. The cookie itself serves a very important purpose, which is to maintain the session between a browser and a server. This is something that Web Storage does not intend to replace, because it exists to meet the growing needs of the times in a way that the cookie is simply incapable of fulfilling, when it comes to the persistence of client-side data.⁴

It strives to reduce the overhead of HTTP requests and offers an incredibly large amount of storage per origin. In fact, the allowed capacity ranges about 5MB. Similar to its predecessor, the Web Storage API enables state to be stored via JavaScript, either indefinitely or solely for the duration of a session. Much like the cookie, Web Storage concerns itself with the persistence of name/value pairs. Because each value supplied to the storage object must be in string form, it can quickly become cumbersome to deal with a plethora of string values, thereby making JSON data the ideal candidate.

Web Storage is accessible to JavaScript, by way of Window Object and can be accessed as `Window.localStorage` and `Window.sessionStorage`. Because the window object is global and can always be reached from within any scope, each storage object can be referenced without the explicit reference of the window object, shortening each reference to `localStorage` and `sessionStorage`.

Both forms of the aforementioned storage objects, whether they be local or session, allow for the storage of state through a similar API. However, as you may have already surmised, the difference between the two regards the contrast among the durations for which the state of data is retained. The `sessionStorage`, as the name implies, allows data to persist only as long as the session exists. Whereas the data stored via `localStorage` will persist indefinitely, either until the state is deleted by the application or user, by way of the browser's interface. Unlike the cookie, all data stored within `localStorage` will not be set to expire.

Web Storage Interface

Web Storage allows for the storing of data, the retrieval of data, and the removal of data. The means by which we will be working with data and the storage object is via the Web Storage API. As [Table 7-2](#) outlines, there are six members that make up the Web Storage API, and each provides a specific need for working with data persistence.

Table 7-2. Six Members of the Web Storage API

Members	Parameter	Return
setItem	string (key), string (value)	void

<code>getItem</code>	string (key)	string (value)
<code>removeItem</code>	string (key)	void
<code>clear</code>		void
<code>key</code>	Number (index)	string (value)
<code>length</code>		Number

Unlike the singular interface of the HTTP cookie, which is used to store, retrieve, and delete data, Web Storage possesses an API to make working with the persistence of data all the more practical. Furthermore, regardless of the storage object you intend to use, whether it's local or session, the API remains uniform.

setItem

The Storage Object method `setItem` possesses the signature of [Listing 7-18](#) and is the method that we will use to persist data. As was mentioned previously, much like the HTTP cookie, Web Storage persists data in the form of name/value pairs. However, while the cookie itself did not distinguish the name from the value it retained, Web Storage does. Therefore, `setItem` does not merely accept a singular string but, rather, requires two strings to be provided. The first string represents the name of the key, and the second string will represent the value to be held.

Listing 7-18. Signature of the `setItem` Method

```
setItem( key , value )
```

When a value is set, it will occur without providing a response back to the invoker of the method. However, if a value is unable to be set, either because the user has disabled the storage or because the maximum capacity for storage has been reached, an `Error` will be thrown. It's as they say, "no news is good news." In other words, if an error does not occur on `setItem`, you can rest assured the data has been set successfully.

Because a runtime error can cause your script to come to a halt, it will be imperative to wrap your call to `setItem` with a `try/catch` block. Then, you can catch the error and handle exceptions gracefully.

Listing 7-19. Storing Our First Item

```
localStorage.setItem("ourFirstItem","abc123");
```

As with the key/value pairs of a JavaScript object, each key must possess a unique label. If you were to store a value with the name of a key that currently exists, that value would effectively replace the previously stored value.

Listing 7-20. Replacing the Value Possessed by the `ourFirstItem` Key

```
localStorage.setItem("ourFirstItem","abc123");
localStorage.setItem("ourFirstItem","sunday Monday happy-
days");
```

At this point in time, if we were to retrieve the value set for `ourFirstItem`, we would witness that the previous value of `"abc123"` had been replaced with the theme song from the television sitcom *Happy Days*.

■ **Tip** Because an error will be thrown if the user has disabled Web Storage, it would be wise to wrap every call to the Storage Object API within a `try/catch` block.

getItem

The Storage Object method `getItem` (see [Listing 7-21](#)) is the counterpart to the `setItem` method. It, like our `getCookie` method from [Listing 7-16](#), allows us to retrieve the persisted state that corresponds to the key provided to the method (see [Listing 7-22](#)).

Listing 7-21. Signature of `getItem`

```
getItem( key )
```

Listing 7-22. Obtaining a Value for a Specified Key

```
console.log( localStorage.getItem( "ourFirstItem" ) );  
//sunday Monday happy-days  
console.log( localStorage.getItem( "ourSecondItem"  
) ); //null
```

The key is the only expected parameter, as indicated in [Listing 7-22](#), and will return the corresponding state for the supplied key. If, however, the name of the key supplied does not exist on the Storage Object, a value of `null` will be returned.

removeItem

The Storage Object method `removeItem` is the sole means of expiring the persistence of an individual key/value pair. Its signature is similar to that of `getItem`, in that it accepts one parameter, as shown in [Listing 7-23](#). This parameter is the key that pertains to the data that you no longer wish to persist (see [Listing 7-24](#)).

Listing 7-23. Signature of `removeItem`

```
removeItem( key )
```

Listing 7-24. Utilizing `removeItem` to Expire a Persisted State

```
console.log( localStorage.getItem( "ourFirstItem"  
)); //sunday Monday happy-days  
        localStorage.removeItem( "ourFirstItem" );  
console.log( localStorage.getItem( "ourFirstItem"  
)); //null
```

clear

As indicated in [Listing 7-25](#), the method `clear` does not require any parameters. This is because this method is simply used to instantly purge each and every key/value pair retained by the targeted Storage Object.

Listing 7-25. Signature of the `clear` Method

```
clear( )
```

key

The Storage Object method `key` is used to obtain the identities of all stored keys that possess accompanying data retained by the given Storage Object. As the signature outlined in [Listing 7-26](#) demonstrates, the method can be provided with that of an index, which will return in kind with the member at the supplied index. If a value does not exist for the provided index, the method will return a value of `null`.

Listing 7-26. Signature of the `key` Method

```
key( index )
```

length

As it will not be beneficial to supply indexes that are beyond the boundaries of stored keys, the Storage Object provides us with access to the length of all values stored by the Storage Object in question. This total can be obtained via the `length` property. The `length` property, when used in conjunction with a loop, as demonstrated in [Listing 7-27](#), provides us with the ability to remain within the boundaries of the values stored.

Listing 7-27. Obtaining the Stored Keys from a Storage Object Is Simple with a Loop

```
var maxIndex= localStorage.length;
for(var i=0; i<maxIndex; i++){
    var foundKey = localStorage.key( i );
}
```

Reusing the key/value pair used by our first cookie, we will demonstrate the ease of the Web Storage API.

Listing 7-28. Utilizing Web Storage to Persist the Value Supplied to Our Person Instance

```
1 function setItem( key , value ){
2     try{
3         localStorage.setItem( key , value );
4     }catch(e){
5         //WebStorage is either disabled or has exceeded
the Storage Capacity
6     }
7 }
8 function getItem( key ){
```

```

9     var storageValue;
10    try{
11        storageValue= localStorage.getItem( key );
12    }catch(e){
13        //WebStorage is disabled
14    }
15    return storageValue;
16 }
17
18 function Person() {
19     this.name;
20     this.age;
21     this.gender;
22 };
23 Person.prototype.getName = function() {
24     return this.name;
25 };
26 Person.prototype.getAge = function() {
27     return this.age;
28 };
29 Person.prototype.getGender = function() {
30     return this.gender;
31 };
32
33 //instantiate new Person
34 var p = new Person();
35     p.name = "ben";
36     p.age = "36";
37     p.gender = "male";
38
39 var serializedPerson = JSON.stringify(p);
40 setItem( "person" , serializedPerson );
41 console.log( getItem( "person" )); // "
{"name":"ben","age":"36","gender":"male"}" ||
"undefined"

```

[Listing 7-28](#) revisits our person example from [Listing 7-17](#) to point out how the Web Storage API and cookie interface vary. The examples similarly use their component to store and retrieve the same value. However, the use of the API provided by Web Storage simplifies things greatly. Unlike in our cookie example, Web Storage requires less work for setting—and especially retrieving—data. **Line 41** of [Listing 7-28](#) simply requests the data of the supplied key and logs it for inspection to the developer console. The reason why the value returned may be either what is stored or (signified by the || operator) “undefined”, is due to the fact that Web Storage may be disabled, which will prevent the variable `storageValue` (**line 9**) from being set. Unlike its cookie counterpart, `getItem` handles the management of key/value pairs for us, so that we don’t have to

manipulate the returned string. Could you imagine performing a JavaScript search over 5MB worth of ASCII characters? The application would become nonresponsive.

What you may have also noticed is that we never specified a domain or path at any point in time during our review of Web Storage. This is because, unlike the cookie, the Storage Object strictly adheres to the same-origin policy, meaning that resources can only be shared/accessed from the same document origin, if the two share the same protocol, hostname, and port. You will learn more about the same-origin policy in [Chapter 9](#).

Summary

The HTTP cookie and Web Storage are extremely useful client-side tools for storing and persisting JSON data. They can be utilized to retain the state of a user's engagement with a web site, web app, or even a game. As cookies and Web Storage are stored on the user's browser, each visitor can potentially possess different information, which can further add to the benefit of local persistence. Such benefit would be personalization/optimization. However, for all their benefits, the cookie and Web Storage are not without their limitations.

The first and foremost concern surrounds security. As both the cookie and a Storage Object can be set and retrieved with JavaScript, it's best practice to store information that is not particularly sensitive. While it may not be understood by the average visitor of your site how data is being utilized between your application and their browser, those who are seeking to exploit these technologies do understand. As this data is accessible to JavaScript, by utilizing the same techniques covered in this chapter, a user or a site hijacker can manipulate or alter persisted state at any point in time, for malicious or benign intent. This, of course, will vary, based on the data as well as the nature of the application that makes use of it.

As I previously indicated, the HTTP cookie and Web Storage are scoped to a visitor's browser. Data that may have been set to persist, whether by cookie or Storage Object, is dependent on the browser the visitor previously used to interact/view your application. This means the persistence of state has the potential to vary from one browser to the other, each time a user visits your application. This inconsistency may prove to be problematic, depending on your application's needs. Last, as the data that is being retained will persist on the visitor's file system and not the server's, it can easily be removed by the visitor at any point he or she chooses, through the interface provided by the browser.

These aforementioned issues can be avoided when used in conjunction with a server-side database, which will be the topic of discussion in [Chapter 12](#). In the next chapter, I will discuss how to transmit JSON to and from our applications via JavaScript.

Key Points from This Chapter

- Data persistence is the continued existence of state after the process that created it.
- HTTP/1.1 is a stateless protocol.

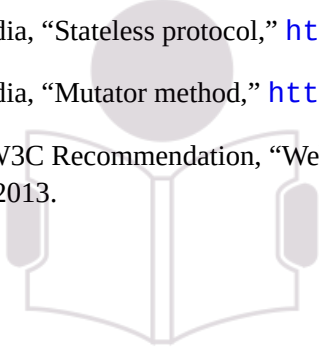
- Cookies and Web Storage are used to retain state.
- Cookies are sent with every HTTP/1.1 request.
- Session data will cease to exist after the session exits.
- Sessions do not necessarily end when a browser is closed.
- Cookies are exchanged via HTTP and HTTPS, unless flagged as secure.
- Cookies can only store 4KB worth of ASCII characters.
- Cookies can be shared among subdomains.
- Web Storage can store 5MB of data.
- Each origin possesses its own Storage Object.
- Web Storage strictly adheres to a same-origin policy.

¹Wikipedia, "Persistence (computer science)," http://en.wikipedia.org/wiki/Persistence_%28computer_science%29, 2014.

²Wikipedia, "Stateless protocol," http://en.wikipedia.org/wiki/Stateless_protocol, 2014.

³Wikipedia, "Mutator method," http://en.wikipedia.org/wiki/Mutator_method, 2014.

⁴W3C, W3C Recommendation, "Web Storage," <http://www.w3.org/TR/webstorage/#introduction>, July 30, 2013.



THE NEXT LEVEL OF EDUCATION

CHAPTER 8



Data Interchange

Thus far, you have been learning how to work with JSON data that has been stringified, parsed, persisted, and retrieved—all from within the scope of a running application. However, as JSON is a data-interchange format, it is capable of being transmitted across the Internet, which offers our applications much more possibility than we have currently been allowing them.

With the use of data interchange, we can send JSON across the Internet into a database that is owned/controlled by us. The visitor cannot as easily delete data this way, as it could be with Web Storage and the HTTP cookie. Furthermore, the ability to transmit data allows our application the ability not only to push out JSON but also to load it into our applications. In other words, not only can we load into our application the data that we've stored, but we can also tap into the data that others are willing to share as well. This may be data that is available to the general public free of charge or by a paid service. Consider the vast array of social sites out there that offer to the public free of charge the data that they capture. Twitter, Facebook, and Instagram are prime examples of social properties that are willing to offer aspects of their data via an API. Because of the many positive attributes that JSON possesses, it is the favored data format of nearly every social API.

In upcoming discussions, you will learn how to load JSON into our application, transmit JSON from our application, and persist JSON into a database over which we have control. Then, we will look at how to incorporate the data from the API of the social property Twitter. However, before we jump into those topics, it will be of great benefit to understand the communication that takes place under the hood of our browser during the request for a resource and the response from a server, as well as the underlying technologies that we will utilize to enable both.

Hypertext Transfer Protocol

The Hypertext Transfer Protocol, or simply HTTP, is the underlying mechanism responsible for our daily interactions with the Internet. It is used in conjunction with many underlying networks of protocols, in order to facilitate the appropriate request/response between a client and a server. Typically, the client utilized in the request/response exchange is that of a web browser, such as Chrome, Firefox, Internet Explorer, or Safari. However, it can also be that of another server. Regardless of whether the client is a browser or a server, the request/response can only take place upon the initiation of a request. Furthermore, a response can only be provided from a web server.

Anytime a resource is requested from a server, whether it's a document, an image, a style sheet, etc., a request must be initiated.

HTTP-Request

It is the role of the request to outline the specifics that detail the required resource from the server. It will be these details that help to ensure that the server provides the appropriate response. A request can be thought of as your order at a restaurant. When you provide a waiter with your order, you are outlining what you are expecting from the kitchen.

Additionally, it may include your preferences of how you would like it to be cooked or served. In the preceding analogy, the HTTP protocol is the waiter, the order is the HTTP request, and the food provided represents the HTTP response.

The HTTP request consists of three general components, each with a particular use for detailing what resource is required from a server. These three components can be viewed in [Table 8-1](#).

Table 8-1. *Structure of the HTTP Request*

	Parts	Required
1	Request Line	Yes
2	Headers	No
3	Entity Body	No

Request Line

The first component, known as the request line, is absolutely mandatory for any request. It alone is responsible for the type of request, the resource of the request, and, last, which version of the HTTP protocol the client is making use of. The request line itself is composed of three parts, separated from one another by whitespace. These three components are Method, Request-URI, and HTTP-Version.

Method represents the action to be performed on the specified resource and can be one of the following: GET, POST, HEAD, PUT, LINK, UNLINK, DELETE, OPTIONS, and TRACE. For the purposes of this chapter, I will only discuss the first two.

The method GET is used to inform the server that it possesses a resource that we wish to obtain. GET is most commonly used when navigating to a particular URL in a browser, whereas the POST method is used to inform the server that you are providing data along with your request. The POST method is commonly used with HTML forms. The response that is supplied upon a form's submission often reflects content that accounts for the form submission.

Because the GET method does not concern itself with any alterations to a server, it is commonly referred to as a *safe method*. The POST method, on the other hand, is referred to as an *unsafe method*, as it concerns working with data.

The URI of the request line simply identifies the resource, which the request method applies. The specified URI may be that of a static resource, such as a CSS file, or that of a dynamic script whose content is produced at the moment of a request.

Last, the request line must indicate the HTTP-Version utilized by the client. Since 1999, the Request-Version of browsers has been HTTP/1.1. Examples of a request line are shown in [Listing 8-1](#).

Listing 8-1. Syntactic Structure of a Request Line

```
GET
http://json.sandboxed.guru/chapter8/css/style.css      HTTP/1.1
GET
http://json.sandboxed.guru/chapter8/img/physics.jpg    HTTP/1.1
POST http://json.sandboxed.guru/chapter8/post.php      HTTP/1.1
```

Headers

The second component of the request concerns the manner by which the request is able to provide supplemental meta-information. The meta-information is supplied within the request in the form of a header, whereas a header, at its most atomic unit, is simply a key/value pair separated by the colon (:) and made up of ASCII characters. The server can utilize this information in order to best determine how to respond to the request.

The HTTP protocol has formalized a plethora of headers that can be utilized to relay a variety of detail to the server. These headers fall under one of three categories: general headers, request headers, and entity headers.

General Headers

The first category of header is that of the general headers. The headers that apply to this category identify general information pertaining to the request. Such general information may regard the date of the request, whether or not to cache the request, etc. The following are general headers:

- Cache-Control
- Connection
- Date
- Pragma
- Trailer
- Transfer-Encoding
- Upgrade
- Via
- Warning

Request Headers

The second category of headers is that of the request headers. These headers can be

supplied with the request to provide the server with preferential information that will assist in the request. Additionally, they outline the configurations of the client making the request. Such headers may reveal information about the user-agent making the request or the preferred data type that the response should provide. By utilizing the headers within this category, we can potentially influence the response from the server. For this reason, the request headers are the most commonly configured headers.

One very useful header is the Accept header. It can be used to inform the server as to what MIME type or data type the client can properly handle. This can often be set to a particular MIME type, such as application/json, or text/plain. It can even be set to */*, which informs the server that the client can accept all MIME types. The response provided by the server is expected to reflect one of the MIME types the client can handle. The following are request headers:

- Accept
- Accept-Charset
- Accept-Encoding
- Accept-Language
- Authorization
- Expect
- From
- Host
- If-Match
- If-Modified-Since
- If-None-Match
- If-Range
- If-Unmodified-Since
- Max-Forwards
- Proxy-Authorization
- Range
- Referer
- TE
- User-Agent

At this point, feel free to navigate the browser of your choice to the following URL: <http://json.sandboxed.guru/chapter8/headers.php>. The content that is displayed is the response to that of an HTTP request. Ironically, the content displayed presents the HTTP request for the requested URI. Here, you can view the combination of general headers and the request headers submitted with the request. Generally speaking, as

we navigate the Internet, the browser supplies the various headers with each request on our behalf. Therefore, some of the request headers supplied possess values that reflect those configured within our browser settings. Because each browser may vary in its values supplied to the reflected headers, your results may not reflect mine, shown in [Listing 8-2](#).

Listing 8-2. The Composition of an HTTP GET Request

```
GET /chapter8/headers.php HTTP/1.1
Host: json.sandboxed.guru
Cache-Control: max-age=0
Connection: close
X-Insight: activate
Cookie: person={"age":"36","name":"ben","gender":"male"}
Dnt: 1
Accept-Encoding: gzip, deflate
Accept-Language: en-us,en;q=0.7,fr;q=0.3
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9;
rv:30.0) Gecko/20100101 Firefox/30.0 FirePHP/0.7.4
```

■ **Note** The Referer header is the result of a spelling mistake that was not caught before it was incorporated within the HTTP specification.

As you can clearly see, the first line, the request line, details the method to apply to the indicated URI of `/chapter8/headers.php`. While the URI is that of a dynamic page, the request line states: GET the resource provided by `headers.php`. That resource, of course, generates its content upon receipt of the HTTP request, in order to reveal the headers as your browser configures them.

While this information will only be present for the particular URI utilizing our developer console, we will be able to view any and all HTTP requests and their responses for any resource. This can be accomplished by profiling the network activity from within the developer's console of your favorite modern browser. Feel free to refresh the page once you have your developer console open and the network tab in view. [Figure 8-1](#) displays the HTTP request and its headers for the request URI <http://json.sandboxed.guru/chapter8/headers.php>.

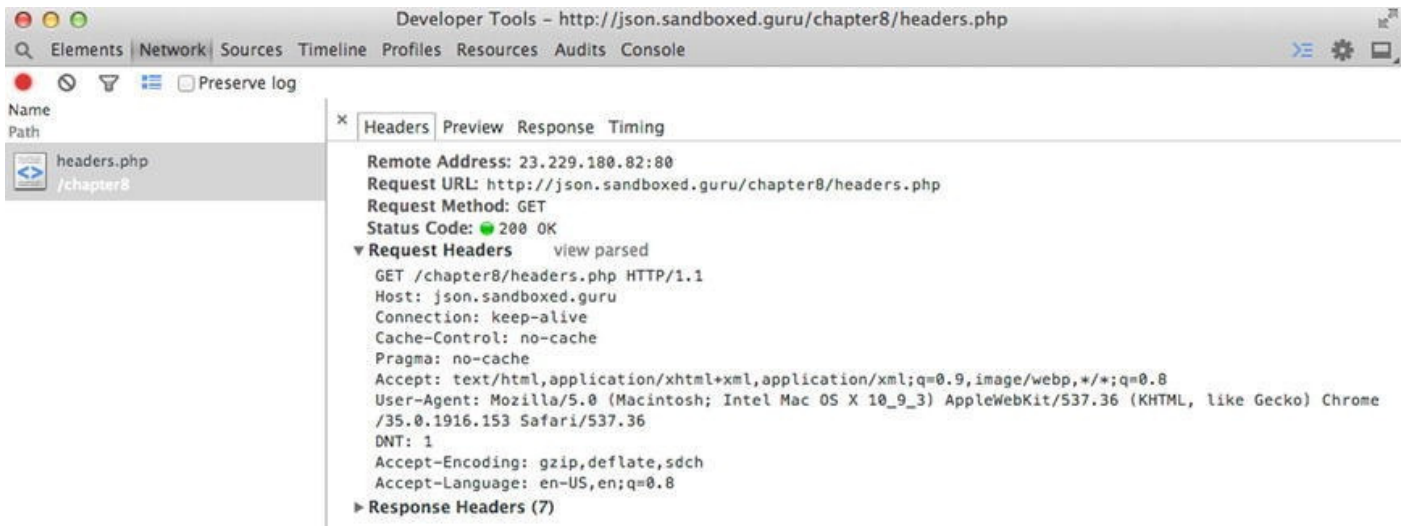


Figure 8-1. The request headers exhibited by the Chrome developer console

Entity Headers

The third category of headers is that of the entity headers. These headers are used to supply meta-information regarding any data that is being sent to the server along with the request. The provision of data that accompanies a request is always tied to the unsafe HTTP methods, such as PUT and POST. Safe methods, on the other hand, will never possess an entity body. However, when data is supplied, it will be these headers that describe the data type being sent, the character encoding it possesses, and the amount of bytes of data being transferred. The following are entity headers:

- Allow
- Content-Encoding
- Content-Languages
- Content-Length
- Content-Location
- Content-MD5
- Content-Range
- Content-Type
- Expires
- Last-Modified

Entity Body

The final component of the request is the entity body. While the entity headers carry the meta-information, the entity body is strictly the nomenclature for the data being sent to the server. The syntax of the entity can reflect that of HTML, XML, or even JSON. However, if the Content-Type entity header is not supplied, the server, being the receiving party of the request, will have to guess the appropriate MIME type of the data provided.

I will now review the request of an unsafe method, so that you can observe a request that is in possession of an entity body. Feel free to navigate your browser to the following URL: <http://json.sandboxed.guru/chapter8/post.php>. By filling out the two form fields and clicking submit, the form post will automatically trigger an HTTP request that will supply the filled-in fields as data. The response that will be outputted to the screen will reflect the captured headers of the POST request. [Listing 8-3](#) reveals the HTTP request and the entity it possesses. Feel free to utilize your developer's console, to compare the request with the results shown below.

Listing 8-3. The Composition of an HTTP POST Request

```
POST /chapter8/headers.php HTTP/1.1
Host: json.sandboxed.guru
Cache-Control: max-age=0
Connection: close
X-Insight: activate
Referer: http://json.sandboxed.guru/chapter8/post.php
Dnt: 1
Accept-Encoding: gzip, deflate
Accept-Language: en-us,en;q=0.7,fr;q=0.3
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9;
rv:30.0) Gecko/20100101 Firefox/30.0 FirePHP/0.7.4
Content-Length: 37
Content-Type: application/x-www-form-urlencoded

fname=ben&lname=smith&mySubmit=submit
```

As you can see from [Listing 8-3](#), an empty line following the other two request components separates the entity body. Furthermore, the two supplied entity headers, Content-Length and Content-Type, provide the server with an understanding of what is being supplied, relieving the server from having to guess how to properly parse the data.

HTTP Response

For every HTTP request there is an HTTP response. Additionally, the structural composition of the HTTP response, as displayed in [Table 8-2](#), is identical to that of the HTTP request with one major exception: the request line is replaced with a status line.

Table 8-2. Structure of the HTTP Response

	Parts	Required
1	Status Line	Yes
2	Headers	No
3	Entity Body	No

Status Line

The first component of the HTTP response is the status line, which details the result of the request. The composition of the status line is composed of three parts: the version of the HTTP protocol utilized by the server, a numeric status code, and an associated textual phrase that describes the status of the request. Each component is separated from the other with whitespace.

The HTTP version simply reflects the version of the HTTP protocol used by the server.

The status code represents a three-digit number that reflects the status of the request. It is the duty of the status code to inform the client whether the request was understood, if it resulted in an error, and/or if the client must take further action. There are five categories of statuses, and each three-digit status code is a member of an appropriate status class.

The status classes, as illustrated in [Table 8-3](#), are divided into groups of hundreds, meaning that the indicated classes can possess 100 different unique status codes. While this is not currently the case, by providing each class with ample padding, additional statuses can be incorporated in the future.

Table 8-3. *Response Status Classes of the HTTP-Request*

Status Class	Reason Phrase
100–199	This class of status code indicates a provisional response, consisting only of the status line and optional headers.
200–299	This class of status code indicates that the client's request was successfully received, understood, and accepted.
300–399	This class of status code indicates that further action needs to be taken by the user-agent, in order to fulfill the request.
400–499	This class of status code is intended for cases in which the client seems to have erred.
500–599	This class of status code indicates cases in which the server is aware that it has erred or is incapable of performing the request.

The most common classes that will be used by the average user will be among the following: 200's, 400's, and 500's. These represent the response messages from the server that will help to indicate if the resource requested has been satisfied or if there were errors along the way. The most common status codes encountered by front-end developers are the following: 200, 204, 404, and 500.

- **200 OK:** The server has successfully recognized the request.
- **204 No Content:** The server has successfully recognized the request; however, there is no new entity body to return.
- **404 Page Not Found:** The indicated resource is unable to be located by the server.

- **500 Internal Server Error:** The server has encountered an issue preventing the request from being fulfilled.

The textual phrase of the status line is utilized, so that it can be easily read and interpreted by humans. Each phrase details the meaning of its associated status code.

■ **Note** You can read more on the existing status codes here:

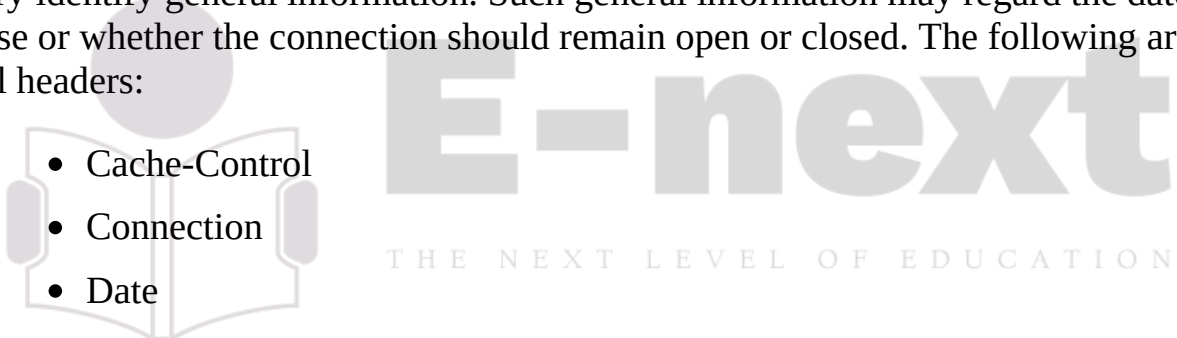
<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>.

Headers

The second component of the response concerns the mechanism by which the response is able to provide the client with supporting meta-information. As with requests, response headers are grouped into three categories: general headers, request headers, and entity headers.

General Headers

The first category of headers is the general headers. The headers that apply to this category identify general information. Such general information may regard the date of the response or whether the connection should remain open or closed. The following are general headers:

- 
- Cache-Control
 - Connection
 - Date
 - Pragma
 - Trailer
 - Transfer-Encoding
 - Upgrade
 - Via
 - Warning

Response Headers

The second category of headers is the response headers. These headers provide the client of the request with information pertaining to the configurations of the server, as well as the requested URI. For example, the server can provide response headers to inform the request of what HTTP methods are accepted, as well as whether authorization is required in order to access the specified URI. These headers can even inform the request whether it should occur at a later point in time. The following are response headers:

- Accept-Ranges

- Age
- ETag
- Location
- Proxy-Authentication
- Retry-After
- Server
- Vary
- WWW-Authenticate

Entity Headers

The third category of headers is the entity headers. These headers are used to supply meta-information regarding the data being sent along with the response. As with entity headers for a request, the most beneficial entity headers for a response will be those that describe the MIME type of the entity provided, so that it may be parsed/read properly. This is achieved via the Content-Type header. The configured value of the Content-Type will often reflect a MIME type that was indicated as the value of the Accept header within the request. The following are entity headers:

- Allow
- Content-Encoding
- Content-Languages
- Content-Length
- Content-Location
- Content-MD5
- Content-Range
- Content-Type
- Expires
- Last-Modified

Entity Body

The final component of the response is that of the entity body. Whereas entity headers outline the meta-information, the entity body is the data provided by the server.

Let's now revisit our earlier HTTP request from [Figure 8-1](#), only this time, let's focus on the response captured in [Figure 8-2](#). [Figure 8-2](#) reveals the response that is returned by the server for the following URL:

<http://json.sandboxed.guru/chapter8/headers.php>. The first thing to note is the status line located below the response headers heading. It begins by revealing

the HTTP version and is immediately followed by the status of the request.

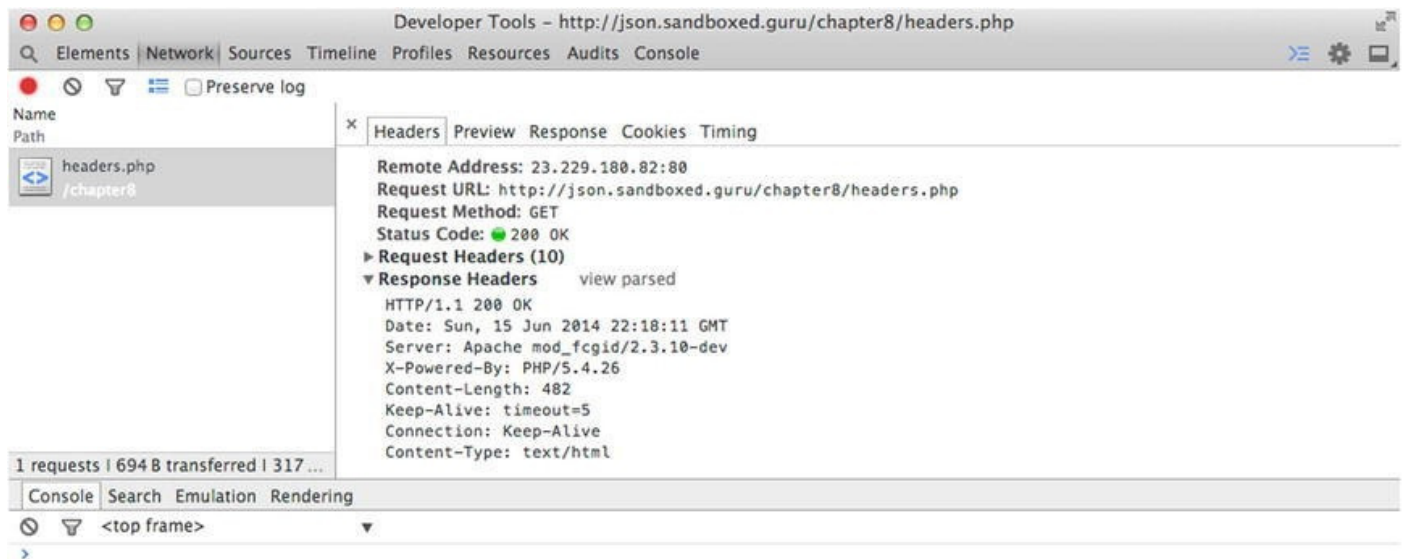


Figure 8-2. The response headers exhibited by the Chrome developer console

In this particular case, the response is successfully fulfilled, as indicated by the status code of 200. Furthermore, from the textual phrase that follows the status code, we can read that the messaging is that of OK. Below the status line, we are able to observe a variety of headers, which belong to the general headers and entity headers categories. I want to draw your attention to the final header in the listing. This particular entity header is configured to define the MIME type of the entity body being returned. This enables the browser to parse it accordingly and display it upon its arrival. In this particular case, the data being provided is HTML and, therefore, possesses the Content-Type of text/HTML.

The actual data that is returned can be viewed in the response tab, which is none other than the markup that is being presented upon arrival of the URL.

If the preceding content is new to you, don't worry, for you are not alone. In fact, typically, only those who are server-side developers know the preceding information. This is because they generally write the code to analyze the request headers and, in turn, configure the appropriate response. Typically, HTTP requests are made behind the scenes and handled by the browser, allowing front-end developers like us to remain ignorant of the communications taking place. However, in the upcoming section, I will discuss the technique that enables us to initiate and configure our own HTTP requests, allowing us to send and receive JSON via JavaScript.

Ajax

Ajax itself is not a technology but, rather, a term coined by Jesse James Garrett in 2005. Ajax stands for *Asynchronous JavaScript and XML* (a.k.a. Ajax) and has become synonymous with modern-day front-end development, and for great reason. It offers the ability to initiate HTTP-Requests such as GET and POST on demand and without having to navigate away from the current web page, as shown in [Figure 8-3](#).

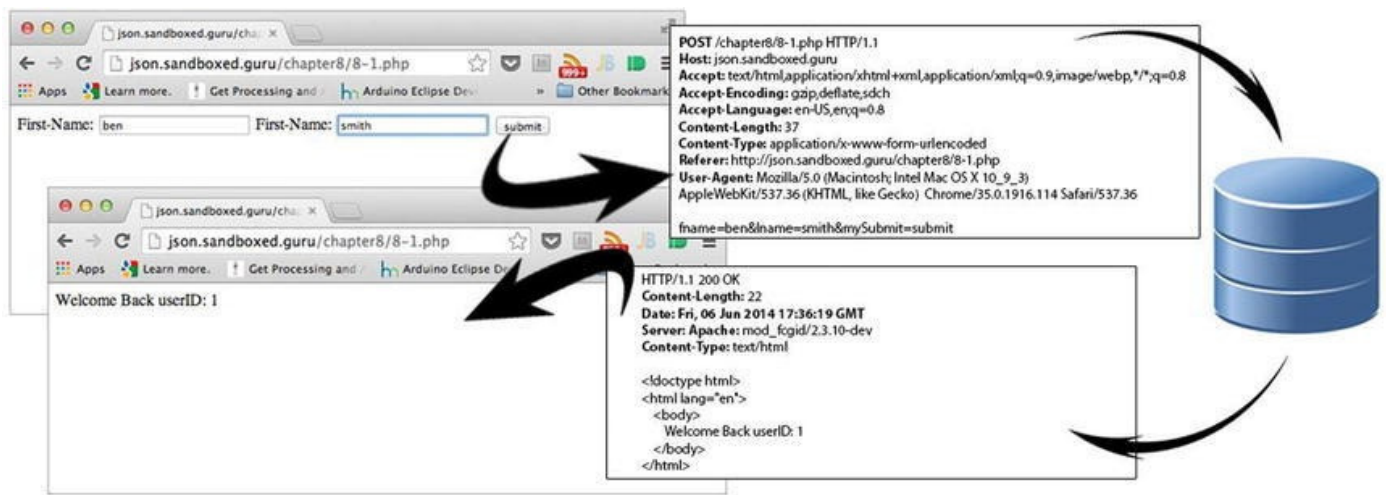


Figure 8-3. The full life cycle of an HTTP GET request

Figure 8-3 demonstrates the process by which data is integrated into a web page when solely handled by the server. The demo begins with a user landing on a web page and being invited to sign in to the site experience via a simple form. Upon clicking submit, the browser initiates a new request to the server, in order to retrieve the appropriate response that reflects the data that has been provided by the user. The headers within that request detail the necessary information for the server to respond accordingly. Once the server receives the request, it fetches the resource being requested, retrieves some information from the database, and inserts it within the content to be returned, thereby revealing an updated page for the visited URL: json.sandboxed.guru/chapter8/8-1.php.

The terms *Asynchronous JavaScript* and *XML* refer to the various web technologies that are used to incorporate the exchange of data between the current web page and a server in the background. You might be thinking that if the *x* in *Ajax* stands for XML, and this is a book on the use of JSON, why then should we care about Ajax? While the *x* does stand for *XML*, the request/response initiated via Ajax continues to remain bound to the rules of the HTTP protocol. Therefore, the server can return any and all valid data types, such as HTML, Text, XML, JSON, etc. We, of course, will be working with JSON. The *x* in *Ajax* came to be simply because the original `XMLHttpRequest` only supported XML parsing.¹

The `XMLHttpRequest` object provides the interface by which JavaScript can initiate an HTTP-Request directly from within a running application, enabling communication with a server. This allows for data to be pushed out or consumed. Furthermore, as the *A* in *Ajax* suggests, this communication occurs asynchronously, implying non-blocking. This allows the executing application and the user to continue, without requiring either to stop what they're doing, until the request has been fulfilled by the server. The HTTP request occurs outside of the process used to run our JavaScript application. More specifically, it occurs in a separate process that is used only by the browser. When the server has fulfilled the request, the browser will alert our application to its availability, by notifying our application via an event. By listening in on this event, we can obtain the response from the server to parse and use, as our application requires.

The `XMLHttpRequest` object, which is the ECMAScript HTTP API,² originated as a proprietary feature within Internet Explorer 5, as a part of the Active X framework. Its practicality and implications became immediately recognized and were quickly

implemented by competing browsers. Anticipating the possible variations and problems that could soon arise among vendor implementations, the W3C urged to formalize the standard of the syntax, which can be read at the following URL:

www.w3.org/TR/2014/WD-XMLHttpRequest-20140130/. This standard outlines the API that developers can leverage to invoke an HTTP request that will facilitate the invocation of an HTTP request.

XMLHttpRequest Interface

The HTTP API, as exposed by the `XMLHttpRequest` object, consists of a variety of methods, event handlers, properties, and states, all of which provide our JavaScript application the ability to successfully facilitate an HTTP request, in addition to obtaining the response from a server. For this reason, each method, property, handler, and state will be integral in a particular aspect of the request or the response.

Global Aspects

The sole global method of the `XMLHttpRequest` interface is that of the constructor (see [Table 8-4](#)), which, when invoked, will return to our application a new instance of the `XMLHttpRequest` object. It will be through the interface inherited by this object that we will initiate and manage our requests. Furthermore, by instantiating multiple instance of the `XMLHttpRequest` object, we can manage simultaneous requests.

Table 8-4. *XMLHttpRequest Constructor*

Method/Property	Parameter	Returned Value
constructor	N/A	XMLHttpRequest (object)

[Listing 8-4](#) demonstrates the instantiation of an `XMLHttpRequest` object and assigns the instance to a variable labeled `xhr`. It will be fairly common to see `xhr` as the reference, as this is simply the acronym for the `XMLHttpRequest` object.

Listing 8-4. Creating an Instance of the `XMLHttpRequest` Object

```
var xhr = new XMLHttpRequest();
```

Whether you are working with one `xhr` or many, as the HTTP request occurs asynchronously, it is necessary for our application to be notified of any change in state, for the duration of the request. Such notifications may be whether the response has been fulfilled or the connection has timed out. The `XMLHttpRequest Level 2` standard outlines the event handlers possessed by each `xhr` instance, so that we may remain aware of the status of the request. These event handlers can be viewed in [Table 8-5](#).

Table 8-5. *The `xhr` Event Handlers for Monitoring the Progress of the HTTP Request*

Event Handlers	Event Handler Event Type
----------------	--------------------------

onloadstart *	loadstart *
onprogress	progress
onload	load
onloadend *	loadended *
onerror	error
ontimeout	timeout
onabort *	abort *
onreadystatechange	readystatechange

■ **Note** The progress events that do not appear with an asterisk (*) beside them are implemented by all modern browsers, in addition to Internet Explorer 8. However, those beside an asterisk require IE 10 or greater.

The event handlers in [Table 8-5](#) will alert our application to a variety of notifications pertaining to the state of the request. Furthermore, they can be utilized in one of two possible implementations.

The first is that we can remain object-oriented and register the event of the state to which we choose to listen. For each event to which we listen, we can assign a particular function to be triggered upon notification, such as that in [Listing 8-5](#). As different browsers implement various ways to register an event, it is necessary to make use of a cross-browser solution, as I have on line 11.

Listing 8-5. The Registration for Event Listeners Belonging to the xhr object for Each Notification of State

```

1  var xhr = new XMLHttpRequest();
2  addListener(xhr, 'loadstart', function() { alert("load-
start"); });
3  addListener(xhr, 'progress', function()
{ alert("progress"); });
4  addListener(xhr, 'load', function() { alert("load");
});
5  addListener(xhr, 'loadended', function()
{ alert("loadended"); });
6  addListener(xhr, 'timeout', function()
{ alert("timeout"); });
7  addListener(xhr, 'abort', function() { alert("abort");
});
8  addListener(xhr, 'readystatechange', function()
{ alert("readystatechange"); });
9
10 //cross browser addListener

```

```

11 function addListener(elem, eventName, handler) {
12     if (elem) {
13         elem.addEventListener(eventName, handler,
false);
14     } else if (elem.attachEvent) {
15         elem.attachEvent('on' + eventName, handler);
16     } else {
17         elem['on' + eventName] = handler;
18     }
19 }

```

The alternative to being notified of a change in a particular state is to assign a function as the callback to the event handler, which exists as a property of the object itself. This manner of implementation is demonstrated in [Listing 8-6](#).

Listing 8-6. Assigning Callback Functions to Each of the xhr Status Event Handlers

```

1 var xhr = new XMLHttpRequest();
2     xhr.onloadstart      = function()
{ alert("onloadstart"); };
3     xhr.onprogress      = function()
{ alert("onprogress"); };
4     xhr.onload           = function()
{ alert("onload"); }; };
5     xhr.onloadend        = function()
{ alert("onloadend"); };
6     xhr.ontimeout        = function()
{ alert("ontimeout"); }; };
7     xhr.onabort          = function()
{ alert("onabort"); }; };
8     xhr.onreadystatechange = function()
{ alert("onreadystatechange"); };

```

Whether the implementation you choose to be made aware, regarding state notifications of the HTTP request, reflects that of [Listing 8-5](#) or that of [Listing 8-6](#), both will produce the equivalent results. The result produced is the invocation of the corresponding function that has been assigned as the receiver of a particular notification, when that event is dispatched.

There are eight progress notifications in total that will inform an application as to the particular state of the HTTP request. These notifications are the following: `loadstart`, `progress`, `error`, `load`, `timeout`, `abort`, `loadend`, and `onreadystatechange`.

The `loadstart` event is dispatched the moment the HTTP request begins. This is not to be confused with the moment communication occurs between the client and the server. As the `loadstart` event reflects the start of a request, it should be expected to be dispatched a total of one time for each request initiated.

The **progress** event is dispatched the moment the HTTP connection is established and the request/response is effectively relaying data. During the course of the transmission, the **progress** event will continue to fire until there is no further data to transmit. This, however, does not always indicate that a successful request has been fulfilled.

The **error** event will be dispatched exactly once, or not at all, during the course of each HTTP request initiated by the **xhr** object. Should the request result in an error, the **error** event will immediately be dispatched. This event is useful for being informed that the request was unsuccessful.

The **load** event will be dispatched exactly once, or not at all, during the course of each HTTP request initiated by the **xhr** object. Should the request be successfully fulfilled, the **load** event will be immediately dispatched. This event is useful for being informed that the request has been completed. It should be mentioned that just because a load is considered completed by the **xhr** object does not necessarily mean that the request was successfully satisfied. Therefore, it will be imperative to provide your callback method with the logic to determine the status code, in order to ensure that it was truly successful. The status code, in addition to the status text, can be obtained by the **status** and **statusText** properties of the **xhr**. I will discuss these two properties a bit later in the chapter.

The **timeout** event will be dispatched exactly once, or not at all, during the course of each HTTP request initiated by the **xhr** object. Should the duration of the request be determined to have surpassed a particular interval, the connection will have been deemed to be timed out, notifying our application of the matter.

The **abort** event is dispatched exactly once, or not at all, during the course of each HTTP request initiated by the **xhr** object. Should the request at any time be aborted, the **abort** event will be immediately dispatched.

The **loadend** event is dispatched exactly once during the course of each HTTP request initiated by the **xhr** object. The **loadend** notification is dispatched the moment the HTTP request is no longer active in its attempt to fulfill a request. This event is dispatched after the following possible notifications: **error**, **abort**, **load**, and **timeout**.

The **onreadystatechange** is the original, and at one time the only, event handler of the **XMLHttpRequest** implemented by earlier browsers. This event is used to notify a supplied function of the progress of the initiated HTTP request. The **onreadystatechange** event is dispatched multiple times during the course of each HTTP request initiated by the **xhr** instance. In fact, the event is dispatched each time the **readyState** property of the **xhr** instance is assigned a new state. The possible states that can be assigned to the **readyState** property are those outlined in [Table 8-6](#).

Table 8-6. *The Possible States of the **xhr** object and Numeric Representation*

States	Numeric Representation
UNSENT	0

OPENED	1
HEADERS_RECEIVED	2
LOADING	3
DONE	4

The states outlined in [Table 8-6](#) are assigned to that of the `readyState` property that exists on each `xhr` instance. The assigned state reflects the progress of the HTTP request itself. There are five possible states that can be assigned to the `readyState` property, and each infers the given state of the request.

The state `UNSENT` is the default state of the `readyState` property. This state is used to inform our application that the `xhr` object, while instantiated, is not yet initialized. The `readyState` property during this state returns a value of `0`.

The state `OPENED` replaces the `UNSENT` state the moment the request method, `open`, has been invoked, initializing our `xhr` instance. The `readyState` property during this state returns a value of `1`.

The state `HEADERS_RECEIVED` is assigned as the value of the `readyState` property upon receiving the headers that accompany the response that will ultimately be received from a server. The `readyState` property during this state returns a value of `2`.

The state `LOADING` is assigned as the value of the `readyState` property as the transmission of data pertaining to the response entity body is received. The `readyState` property during this state returns a value of `3`.

The state `DONE` is assigned as the value of the `readyState` property upon the conclusion of the HTTP request. This state reflects only the closure of the request. As with the `load` event, the `done` state does not identify if the request resulted in an error, a timeout, or a successful fulfillment of a request. Therefore, it will be imperative to determine the `statusCode` when determining how to process the request. The `readyState` property during this state returns a value of `4`. [Listing 8-7](#) demonstrates an event handler that monitors all states of the `readyState` property.

Listing 8-7. Determining the State of the `xhr` object for Each Change in State

```
1  var xhr = new XMLHttpRequest();
2      xhr.onreadystatechange = handleStateChange;
3
4  function handleStateChange() {
5      if (xhr.readyState === 0) {
6          alert("XHR is now instantiated");
7      } else if (xhr.readyState === 1) {
8          alert("XHR is now Initialized");
9      } else if (xhr.readyState === 2) {
10         alert("Headers are now Available");
11     } else if (xhr.readyState === 3) {
```

```

12         alert("Receiving Data");
13     } else if (xhr.readyState === 4) {
14         alert("HTTP Request ended");
15     }

```

As an older implementation, the `onreadystatechange` does not offer an application as accurate a notification system as the other seven progress events. Furthermore, the processing that is required by our JavaScript to determine the state of the HTTP request, if extensive, has the ability to block the thread, thereby delaying the events from being triggered.

The Request Aspect

The methods and properties that are outlined within this section make up the facade that enables one to correctly configure the metadata of the HTTP request. (See [Table 8-7](#).)

Table 8-7. *The Request Methods of the `xhr` object*

Method	Parameters	Returned Value
<code>open</code>	String (method), String (URI), Boolean (async), String (user), String (password)	N/A
<code>setRequestHeader</code>	String (field), String (value)	N/A
<code>send</code>	String (entity body)	N/A
<code>abort</code>	N/A	N/A

open

The `open` method, whose signature can be viewed in [Listing 8-8](#), acts as the starting point that will be used to configure the HTTP request.

Listing 8-8. The Signature of the `open` Method of the `xhr` object

```

open( HTTP-Method, request-URI [, async [, user [,
password]]]);

```

As revealed by [Listing 8-8](#), the `open` method accepts five arguments. Three are optional, and two are required.

The first parameter, `HTTP-Method`, indicates to the server what method it requires to be performed on the specified request URI. A resource may be the target of a “safe” or “unsafe” method. As discussed in the earlier sections of the chapter, the two types of methods this chapter will focus on are `GET` and `POST`.

The second parameter, `request-URI`, identifies the target of our request. The argument supplied to the `request-URI` can be specified either as a relative URL or,

alternatively, an absolute URL. As the `XMLHttpRequest` object is subject to the same-origin policy, the URI supplied must possess the same origin as the application configuring the request. If, however, the URL provided is that of another host outside of the current origin, the server of the URL being targeted must allow for cross-origin resource sharing. I will discuss cross-origin resource sharing in the next chapter.

■ **Note** The `XMLHttpRequest` object is subject to the same-origin policy.

The required parameters will be appended together, along with the HTTP protocol version, which is typically 1.1, to form the very first line of the request, which is the request line, as shown in [Listing 8-9](#).

Listing 8-9. A GET Request for the URI `xFile.php` via the HTTP/1.1 Protocol

```
GET /xFile.php HTTP/1.1
```

The third parameter of the `open` method does not supply metadata to the request but, rather, indicates if the request will occur asynchronously or synchronously. When this parameter is left undefined, it defaults to `true`, thereby processing the HTTP request in another thread.

The final two optional parameters, `user` and `password`, are used to supply credentials that may be required of a resource whose access requires basic authentication. These values will add to the metadata of the request only if the server responds with a 401 `Unauthorized` status code.

setRequestHeader

The next method, `setRequestHeader`, offers our application the opportunity to specify particular headers that will complement the request by providing supplemental information. These can be any of the recognized standard HTTP/1.1 attribute-value fields. As indicated by the signature of the `setRequestHeader` defined in [Listing 8-10](#), the field and value are to be supplied as individual strings. Behind the scenes, the `xhr` object will append them together, separated by a colon (`:`) token. Furthermore, any number of request headers can be supplied to the request in question.

Listing 8-10. Signature of the `setRequestHeader` Method of the `xhr` object

```
setRequestHeader( field , value );
```

Via `setRequestHeader`, our application can supply any attribute value that aids in the fulfillment of the response from the server. Such headers, as illustrated in [Listing 8-11](#), are the `Accept` headers, which outline the preferred media types that our application recognizes. As the content we will be requesting most commonly from the server will be that of JSON, we will be using the **`application/json`** media type.

Additionally, if the HTTP-Method is specified to be that of an “unsafe” method, we can assign the Content-Type as a request header, to outline the encoding and MIME type of the supplied entity body provided with the request. I will discuss how to append an

entity body in the **send** method later in this section.

The headers supplied can also represent custom attribute values, which can be useful for supporting custom requests. It's common practice to precede all custom headers with an X.

Listing 8-11. The Provision of the Accept Header and a Custom Header via the `setRequestHeader` Method

```
setRequestHeader( "Accept" , "application/json" );  
//requesting JSON as the response  
setRequestHeader( "X-Custom-Attribute" , "Hello-World"  
); //custom header
```

For the most part, all standard HTTP/1.1 headers can be supplied. However, there are a few particular headers that cannot be overridden, due to security measures as well as maintaining the integrity of data.³ These values are listed in [Table 8-8](#). If your application attempts to supply values for any of the listed headers in [Table 8-8](#), they will be overridden to their default values.

Table 8-8. The Assorted HTTP Headers That Cannot Be Set Programmatically via JavaScript

Accept-Charset	Cookie	Keep-Alive	Trailer
Accept-Encoding	Cookie2	Origin	Transfer-Encoding
Access-Control-Request-Headers	Date	Referer	Via
Access-Control-Request-Method	DNT	Upgrade	
Connection	Expect	User-Agent	
Content-Length	Host	TE	

send

The `send` method of the `xhr` object is what prompts the submission of the request. As indicated by its signature in [Listing 8-12](#), the `send` method can be invoked with an argument supplied. This argument represents the entity body of the request and is typically used if the request method is specified as one of the “unsafe” methods, such as `POST`.

Listing 8-12. The Signature of the `send` Method of the `xhr` object

```
send ( data );
```

The data supplied can consist of nearly anything; however, it must be supplied in the form of a string. Data can be as simple as a word or a series of key/value pairs strung together to resemble a form post, or even that of JSON text. [Listing 8-13](#), [Listing 8-14](#) and [Listing 8-15](#) demonstrate three different Content-Types being submitted via a form post.

Listing 8-13. Data Sent As the Entity Body of the Request with the Content-Type Defaulted to `text/plain`

```

var xhr = new XMLHttpRequest();
    xhr.open("POST",
"http://json.sandboxed.guru/chapter8/xss-post.php");
    xhr.send( "fname=ben&lname=smith" );
    //content-type will be defaulted to text/plain;
charset=UTF-8.

```

Listing 8-14. Data Sent As the Entity Body of the Request with the Content-Type Specified As x-www-form-urlencoded

```

<form action="8-1.php" method="post" onsubmit="return
formSubmit();">
    First-Name:<input name="fname" type="text" size="25" />
    Last-Name:<input name="lname" type="text" size="25" />
</form>
<script>
function formSubmit(){
    var xhr = new XMLHttpRequest();
        xhr.open("POST",
"http://json.sandboxed.guru/chapter8/xss-post.php");
        xhr.setRequestHeader("Content-Type", "application/x-
www-form-urlencoded");
        xhr.send( "fname=ben&lname=smith&mySubmit=submit" ) ;
        return false;
}
</script>

```

Listing 8-15. Data Sent As the Entity Body of the Request with the Content-Type Specified As JSON

```

var person={name:"ben", gender:"male"};
var xhr = new XMLHttpRequest();
    xhr.open("POST",
"http://json.sandboxed.guru/chapter8/xss-post.php");
    xhr.setRequestHeader("Content-Type", "application/json");
    xhr.send( JSON.stringify( person) );

```

Whatever the supplied data, if you do not define the MIME type of the data by way of the Content-Type header, the type for the data provided will be defaulted to **text/plain; charset=UTF-8**, as in [Listing 8-13](#). At this point, if you were to run the preceding listings (8-13 through 8-15) from your local machine, the request would fail. This is due to the fact that `xhr` strictly adheres to the same-origin policy. Requests can only be to a server if the request is initiated from the same origin. There is a way around this, which I will discuss further in the next chapter. In the meantime, feel free to run these listings and monitor the HTTP request via the developer console. Each listing can be viewed at the following URLs:

- <http://json.sandboxed.guru/chapter8/8-12.html>

- <http://json.sandboxed.guru/chapter8/8-13.html>
- <http://json.sandboxed.guru/chapter8/8-14.html>

■ **Note** If you have been following along with the supplied URLs and have yet to clear your cookies, you may have witnessed some of the cookies from the previous chapter sent within the above requests.

abort

The final method of the request, **abort**, informs the HTTP request to discontinue/cancel the request. This method effectively closes any connection that has been made to a server or prevents one from occurring if a connection has not yet been made.

In addition to methods, the **xhr** object provides a few attributes that can help us with configuring our request. These properties can be found in [Table 8-9](#).

Table 8-9. *The Request Attributes of the **xhr** object*

Properties	Returned Value
Timeout	Number (duration)
withCredentials *	Boolean (credentials)
upload *	XMLHttpRequestUpload (object)

■ **Note** The request properties that are not distinguished by an asterisk (*) are implemented by all modern browsers, in addition to Internet Explorer 8. Those marked by an asterisk require IE 10 or greater.

timeout

The **timeout** property can be set in milliseconds to that of any duration. The value supplied will be the maximum allotted time for a request to complete. If a request surpasses the provided time, the time-out event is dispatched to notify our application.

withCredentials

The **withCredentials** property can be set to that of either **true** or **false**. The value supplied is used to inform the server that credentials have been supplied with a cross-origin resource request.

upload

The **upload** property, when read, provides our application with a reference to an **XMLHttpRequestUpload** object. This object provides our application with the ability to monitor the transmission progress for the entity body of a supplied request. This will be

useful for any entity body that contains an excessive amount of data, such as when allowing users to post various file attachments, such as images, or media.

At this point in time, you should possess the necessary understanding of the various methods and properties possessed by the `xhr` object that will allow for devising and configuring an HTTP request from a JavaScript application. The `xhr` provides us the vehicle we can leverage to transmit JSON to and from our application.

EXERCISE 8-1. AJAX FORM POST

With this newfound knowledge, you should be able to convert the HTML `<form>` element of the following code into an Ajax call.

```
<body>
  <div class="content">
    <form
action="http://json.sandboxed.guru/chapter8/exercise.php"
      method="post" onsubmit="return ajax();">
      First-Name:<input name="fname" type="text"
size="25" />
      Last-Name: <input name="lname" type="text"
size="25" />
      <input name="mySubmit" type="submit"
value="submit" />
    </form>
  </div>
  <script>
    function ajax() {
      //... insert HTTP Request here
    }
  </script>
</body>
```

As we will be controlling the request via JavaScript, and because our favored Content-Type is JSON, make sure that the data of the entity body is provided as JSON. You can compare your answer to that of the preceding code.

Normally, the `XMLHttpRequest` object is incapable of making successful requests to servers that do not possess the same origin as the document from which the request it initiated. However, I have employed a technique, which you will learn about in [Chapter 9](#), that will allow your `xhr` instances to successfully make requests to the following request URI: <http://json.sandboxed.guru/chapter8/exercise.php>.

Unfortunately, if you are authoring your code using Internet Explorer 8 or 9 to make requests against varying origins, you cannot utilize the `XMLHttpRequest` object. Instead, you must initialize the `XDomainRequest` object. Furthermore, while the `XMLHttpRequest` enables you to specify the Content-Type via the `setRequestHeader`, the `XDomainRequest` does not possess this capability.

The Response Aspect

While the `xhr` object enables us to configure the request, it will serve no purpose without the understanding of how to extract the response provided. Therefore, the `xhr` object also incorporates various methods and properties that are concerned solely with working with the response provided by the server.

As you learned earlier in the chapter, both the HTTP request and the response of said request are broken into three components. These represent the request-line/status-line, headers, and the payload. While both the headers and the payload are used in collaboration to arrive at a parsed response, they are obtained separately via the `xhr` interface. The methods listed in [Table 8-10](#) reflect the three methods of the `xhr` interface that are utilized for working with the headers of the HTTP response, which will ultimately inform our application of any details pertaining to the response.

Table 8-10. *Response Methods of the `xhr` object*

Method	Parameters	Returned Value
<code>getAllResponseHeaders</code>	N/A	String (value)
<code>getResponseHeader</code>	String (key)	String (value)
<code>overrideMimeType</code>	String (Content-Type)	N/A

getAllResponseHeaders

The `getAllResponseHeader's` method of the `xhr` interface is used to return the various headers that have been configured by the server to accompany the supplied response. When invoked, `xhr` returns a string of all headers of the response as key/value pairs, each of which remains separated from another by a carriage return and new line control characters. These control characters are represented by the following Unicode values respectively: `\u000D` and `\u000A`. Furthermore, each key/value pair is separated from another via the colon (`:`) token.

Knowing the syntax of the value returned, we can parse the string and simply extract each header into an array, with the help of some minor string manipulation, as revealed in [Listing 8-16](#).

Listing 8-16. Extracting All Values That Are Configured to the Provided Response Headers

```
...truncated code
5 //when the xhr load event is triggerd parse all headers
6 xhr.onload = parseHeaders;
7
8 //parseHeaders will manipulate the string
9 function parseHeaders() {
10     var headers = new Object();
```

```

11     var responseHeaders = (this.getAllResponseHeaders());
12     //match sequences of characters that preceded control
characters into an array
13     var headerArray
= (responseHeaders.match(/[\u000D\u000A].*/gi));
14     for (var i = 0; i < headerArray.length; i++) {
15         var akeyValuePair = headerArray[i];
16         var colonIndex = akeyValuePair.indexOf(":");
17         var headerKey   = akeyValuePair.substring(0,
colonIndex);
18         var headerValue
= akeyValuePair.substring(colonIndex + 1);
19         headerValue = (headerValue.charAt(0) == " ")
? headerValue(1) : headerValue;
20         headers[headerKey] = headerValue;
21     }
22 }

```

[Listing 8-16](#) demonstrates how all headers can be extracted with a simple function labeled `parseHeaders`. Once the `xhr` load event notification is dispatched, `parseHeaders` is invoked (**line 6**). Once the `parseHeaders` function runs, we initialize an object, which will be used to retain any and all found headers and their values.

As `parseHeaders` is invoked by `xhr`, references to `this` remain implicitly set to the context of the `xhr` object. Therefore, referencing `this` enables our function to invoke the `getAllResponseHeaders` method, obtaining the string of all header-value pairs (**line 11**). The returned string is assigned as the value to the variable labeled `responseHeaders`.

Utilizing a regular expression, we can extract any sequence of characters that precede the two control characters, thereby separating one header-value pair from another. All found matches are then appended to an array in the order they are encountered. Once the entire string has been compared against the pattern, an array is returned, containing all matches respectively. In order to manipulate these matches further, we assign the array as the value to variable `headerArray` (**line 13**). From there, we iterate over each indexed value, so that we can separate the key from its value. Knowing that a colon (`:`) token is used to separate the two, we can determine the location of said token (**line 16**), allowing us to extract everything up to the token (**line 17**) and everything after the token (**line 18**). The two substrings, respectively, reflect the header and its value. While the HTTP protocol states that headers and values are separated via the colon (`:`) token, they are also separated by an additional space. Therefore, if the first character of the substring that represents our value is that of a space, it is effectively removed (**line 19**). From there, we apply each key and its correlating value to the `headers` object.

While it may not be immediately apparent why you would have to analyze all supplied headers, it will simply come down to the use case. The `getAllResponseHeaders` is essential when your actions rely on the metadata of the response. Such a use case would be when you pair an HTTP request with that of the request method `HEAD`, which is used to

solely fetch header information from a server.

getResponseHeader

The `getResponseHeader` method, whose signature can be viewed in [Listing 8-17](#), can be utilized to obtain the value for the specified response header, as configured by the server. The key supplied can be either uppercase or lowercase, but the format of the argument must be that of a string.

Listing 8-17. The Signature of the `getResponseHeader` Method of the `xhr` object

```
getResponseHeader( key );
```

If the key supplied is not a configured header among those possessed by the response, the value returned will be that of `null`. Much like `getAllResponseHeaders`, being able to analyze the meta-information supplied within the response can be vital in coordinating how you display, update, or even utilize the data provided.

As was explained earlier, the *X* in *Ajax* represents XML, because, at the time, XML was the only data type outside of plain/text able to be parsed by the `xhr` object. While many browsers have been making great strides to offer a variety of natively returned data types, ranging from plain text to JSON, Internet Explorer 8 and 9 continue to provide us only with the original two flavors. This makes for a particularly strong case as to why one would require the use of `getResponseHeaders`. If the data type supplied from the server is not in fact XML, with the use of the `getResponseHeaders` method, one is able to obtain the correct Content-Type of the supplied entity body and correctly parse the string per the syntax of said data format, as demonstrated in [Listing 8-18](#).

Listing 8-18. HTTP POST to `exercise.php` with Configured Content-Type and Accept Headers

```
1 var xhr = new XMLHttpRequest();
2 xhr.open("POST",
"http://json.sandboxed.guru/chapter8/exercise.php");
3 xhr.setRequestHeader("Content-Type", "application/json");
4 xhr.setRequestHeader("Accept", "application/json");
5 xhr.onreadystatechange = changeInState;
6 xhr.send('{"fname":"ben","lname":"smith"}');
7
8 function changeInState() {
9     var data;
10    if (this.readyState === 4 && this.status === 200) {
11        var mime = this.getResponseHeader("content-
type").toLowerCase();
12        if (mime.indexOf('json')) {
13            data = JSON.parse(this.responseText);
14        } else if (mime.indexOf('xml')) {
15            data = this.responseXML;
16        }
```

```
17     }  
18 }
```

Listing 8-18 leverages our earlier exercise to help demonstrate the benefit of the `getResponseHeader` method. Thus far, I have not discussed what data type the earlier exercise returns as the response entity. I also have not yet discussed any of the properties that enable you to read the obtained request. Unless you looked at the headers provided by the response via the developer console, you may not have known whether the entity body returned was that of XML, HTML, plain text, or JSON. Odds are you cleverly deduced it was JSON, as you realized the context of this book. However, the point is that you may not have known for certain. Therefore, rather than assuming, it's best to account for the varying possibilities, so that you are able to work with the supplied data accordingly.

Listing 8-18 begins with the initialization of our `xhr` object and supplies it with the necessary HTTP-Method and request-URI (**line 2**). As our request method is specified as `POST` and will be supplying data to the server, we continue to configure the Content-Type of the provided data (**line 3**), informing the server how to parse it correctly. As this book concerns working with JSON, we inform the server that our application accepts the Content-Type of `application/json` (**line 4**). In order to monitor the state of the request, the `changeInState` function is assigned as the callback (**line 5**). While I chose to make use of the `onreadystatechange` to monitor the state of the request, I could have just as easily used the `onload` event handler. However, as the event handlers are only available in Internet Explorer 8, I wanted to demonstrate how to achieve the results of the `onload` notification, for those who must continue to work with older browsers.

Last, we use the `send` method of the `xhr` object to invoke the HTTP request and, in doing so, provide it with the necessary JSON data to `POST` (**line 6**).

The function `changeInState` (**line 8**) supplied as the callback to the `onreadystatechange` is not only used to determine the change in state but also the Content-Type, if the request is successful (**line 11**). If you relied on the `onload` event handler, you would not have to determine the state, as the event suggests it's done. However, because the `onreadystatechange` is triggered each time the `readyState` property of the `xhr` object is updated, it's imperative to query the status of the request.

In order to distinguish among the five various states of the `xhr` object, it is necessary to determine the value of the `readyState` property. If the `readyState` value is 4, we know the current state of the `xhr` object is `DONE`. However, in order to determine if the response has successfully provided us with an entity body, the status code is also analyzed (**line 10**). If the status code is found to be 200, which signifies that a response is successful, we can begin to determine how to parse the data of the response.

We begin by utilizing the `getResponseHeader` to obtain the lowercase value of the specified Content-Type for the response, as configured by the server (**line 11**). Once we have obtained the value, we determine if it matches the JSON MIME type (**line 12**) or that of XML (**line 14**). Depending on the outcome of the determined type, the appropriate value is assigned to the `data` variable. If the Content-Type is found to be that of XML,

the value is obtained via the `responseXML` property of the `xhr` object (**line 15**). However, should it be determined that the response has been provided in the JSON data format, we must obtain the raw string from the `responseText` and supply it to the native JSON Object to be parsed (**line 13**). I will discuss the `responseXML` and `responseText` properties in the next section.

overrideMimeType

The `overrideMimeType` method enables our application to override the configured Content-Type of the response body when obtained. FireFox, Chrome, and Safari have implemented this method, which was added in the XMLHttpRequest Level 2 draft standard. However, at the time of this writing, it is currently unavailable in Internet Explorer 11.

Obtaining the Response

The variety of properties of the `xhr` object listed in [Table 8-11](#) provides us with the necessary means to obtain the provided response of the HTTP request. It will be with the help of these attributes that we will come full circle in our ability to initiate a request and, ultimately, obtain the response of that request.

Table 8-11. *The Response Properties of the `xhr` object*

Properties	Access type	Returned Value
<code>readyState</code>	Read	Integer (state)
<code>status</code>	Read	Integer (HTTP status Code)
<code>statusText</code>	Read	string (HTTP status)
<code>responseXML</code>	Read	XML (value)
<code>responseText</code>	Read	string (value)
<code>responseType</code>	Read/Write	XMLHttpRequestResponseType (object)
<code>response</code>	Read	* (value)

readyState

The `readyState` property of the `xhr` object exhibits the current state of the HTTP request. Throughout the asynchronous process of the HTTP request, the `readyState` attribute will be updated regularly to reflect the status of the request. The values for which it can be assigned are the integers discussed previously in [Table 8-6](#).

■ **Note** As the states reflected are rather broad, the `readyState` property will often be paired with other properties, such as the `status` or `statusText` properties, in order to arrive at the necessary outcome.

status

The `status` property of the `xhr` object supplies an application with the ability to obtain the HTTP status code of the response. Currently, there are five classes for the status codes. These classes are those outlined earlier in the chapter in [Table 8-3](#).

[Listing 8-18](#) relied on both the `readyState` and the `status` property to determine if the load had completed successfully. As shown on **line 10**, `if(this.readyState === 4 && this.status === 200)`, we determined via the `readyState` if the `xhr` request had ended, in addition to determining whether the status of the response is that of 200. A status code of 200 indicates that the request has been acknowledged.

statusText

`statusText`, like the `status`, is yet another property of the `xhr` object that is concerned with providing us the appropriate status regarding the fulfillment of the response. Each status code is accompanied by a textual phrase that provides additional information regarding the status. Via `statusText`, the description that accompanies the status code can be obtained and read by our application.

Using our 200 status code as an example, it is accompanied with the textual phrase OK. This is very helpful when obtaining descriptive issues that can be relayed back to the user, or even a developer, during the course of debugging.

■ **Note** The textual phrase that accompanies the status code is intended more for debugging than for controlling the flow of an application.

responseXML

`responseXML` is the attribute of the `xhr` object that enables an application to obtain an XML response provided by the server. As the data supplied within the response will not always be configured as one of the XML Content-Types, `application/xml` or `text/xml`, the `responseXML` attribute will not always provide a value. In the case of a server providing a response with the Content-Type that is not indicative of XML, a value of `null` will be returned when read from our application.

It should be made known that `responseXML` is not solely for an XML document. Due to the resemblance, the `responseXML` attribute can also be used to retrieve HTML documents identified by the `text/html` Content-Type.

responseText

`responseText` is a property of the `xhr` object that provides our applications with the ability to obtain the raw text of the entity body, as provided by the response. While `responseXML` may often possess a value of `null`, `responseText` will always possess a value.

Because the `responseText` attribute provides our application with the raw entity

body received as a string, we must obtain the value of the Content-Type header. The configured Content-Type header will give us insight as to the syntax required for parsing the string. Once this is obtained, we can parse the string into the intended format, as demonstrated on line 13 of [Listing 8-18](#).

responseType

The `responseType` property of the `xhr` object is concerned with the parsing of data types natively, beyond that of mere XML. As has been previously stated, the `xhr` object has the ability to parse a response as XML data. However, as XML is not today's data interchange standard, and has not been for quite some time, much of the parsing that occurs is forced to take place on the client side. Unfortunately, this puts the onus on the application to parse a string. Essentially, this increases the odds of blocking the single thread of the JavaScript engine. By allowing the browser to parse the request, the JavaScript thread is less likely to become blocked.

The `responseType` property has been added to the XMLHttpRequest Level 2 draft standard in an attempt to offload the parsing from the client side for five particular Content-Types. These are the following: `arraybuffer`, `blob`, `document`, `text`, and `json`. This is great news for JSON because, as you may recall, `JSON.parse` is a blocking method. In order to offload the parsing of our response entity to the process handling the request, we must configure the `responseType` before we invoke the `send` method. Any one of five aforementioned data types can be assigned as the value for the `responseType` attribute.

By configuring our request with a `responseType` attribute, we are able to inform the `xhr` process to parse the entity body against the indicated syntax. In [Listing 8-19](#), I've indicated that the syntax is that of JSON.

Listing 8-19. HTTP Request Configured to Parse JSON

```
1 var xhr = new XMLHttpRequest();
2 xhr.open("POST",
  "http://json.sandboxed.guru/chapter8/exercise.php");
3 xhr.setRequestHeader("Content-Type", "application/json");
4 xhr.setRequestHeader("Accept", "application/json");
5 xhr.onreadystatechange = changeInState;
6 xhr.responseType = "json";
7 xhr.send('{"fname":"ben","lname":"smith"}');
```

response

The `response` property of the `xhr` object, like `responseXML` and `responseText`, provides our application with a way to obtain the entity body of the fulfilled request. However, the major difference is that the value `read` will be parsed, that is, if we have configured the HTTP request with `responseType`. Otherwise, the value returned is an empty string.

[Listing 8-20](#) revisits the previous listing and configures the request to utilize the

`responseType` of JSON (**line 6**). As the parsing will now occur within a separate process from our application, we no longer need to parse the JSON ourselves. Therefore, we can replace line 14 with that of the `response` attribute, which should now hold a JavaScript object.

Listing 8-20. HTTP Request Obtaining the Parsed JSON from the `xhr` Response Property

```
1 var xhr = new XMLHttpRequest();
2     xhr.open("POST",
"http://json.sandboxed.guru/chapter8/exercise.php");
3     xhr.setRequestHeader("Content-Type",
"application/json");
4     xhr.setRequestHeader("Accept", "application/json");
5     xhr.onreadystatechange = changeInState;
6     xhr.responseType = "json";
7     xhr.send('{"fname":"ben","lname":"smith"}');
8
9 function changeInState() {
10     var data;
11     if (this.readyState === 4 && this.status === 200) {
12         var mime = this.getResponseHeader("content-
type").toLowerCase();
13         if (mime.indexOf('json')) {
14             data = this.response;
15         } else if (mime.indexOf('xml')) {
16             data = this.responseXML;
17         }
18     }
19 }
```

While the `responseType` and `response` properties have been implemented in most browsers, Internet Explorer continues to remain behind the times. XMLHttpRequest Level 2 methods and attributes are only available in IE 10 or greater.

The preceding examples relied on the provision of dynamic data from a database on my server. However, Ajax does not necessarily have to work with dynamic data. In fact, Ajax is fantastic at loading static files as well. **Listing 8-21** exposes the content body of a file labeled `images.json`, which reveals the following JSON within.

Listing 8-21. JSON Content Within `/data/imagesA.json`

```
{
  "images": [
    {
      "title": "Image One",
      "url": "img/AndroidDevelopment.jpg"
    }, {
      "title": "Image Two",
```

```

        "url": "img/php.jpg"
    }, {
        "title": "Image Three",
        "url": "img/Rails.jpg"
    }, {
        "title": "Image Three",
        "url": "img/Android.jpg"
    }
]
}

```

Listing 8-21 reveals an object that possesses a singular member labeled “images”. Images, as a key, reference the value of an ordered list, where each index of said ordered list references an object. These objects represent the necessary details pertaining to various images that will be added dynamically to our page. The key `url` reflects the location from which the image is supplied, while the title is used to populate the `alt` tag of the dynamically inserted image. **Listing 8-22** reveals the code that will load, parse and insert `data/imagesA.json` into an HTML document.

Listing 8-22. The Body of an HTML File That Utilizes Ajax to Load the JSON Document `data/imagesA.json`

```

1 <body>
2 <input type="submit" value="load
images" onclick="loadImages('data/imagesA.json')"/>
3 <script>
4     function loadImages(url) {
5         var body = document.getElementsByTagName("body")
[0];
6         var xhr = (window.XDomainRequest) ? new
XDomainRequest() : new XMLHttpRequest();
7         xhr.open("GET", url);
8         xhr.onload = function() {
9             var data = JSON.parse(this.responseText);
10            var list = data.images;
11            for (var i = 0; i < list.length; i++) {
12                var image = list[i];
13                var listItem
= document.createElement("li");
14                var img = document.createElement("img");
15                img.src = image.url;
16                img.alt = image.title;
17                listItem.appendChild(img);
18                body.appendChild(listItem);
19            }
20        };
21        xhr.onerror = function() {

```

```

22         alert( this.status + " "
+ this.statusText);
23     };
24     xhr.send();
25 };
26 </script>
27 </body>

```

[Listing 8-22](#) demonstrates the use of Ajax to load the static file from [Listing 8-21](#), populating a variety of images within the page. The document reveals nothing but a submit button within the page (**line 2**). This button, when clicked, will trigger the JavaScript code that will both load the `image.json` file and dynamically insert each found image into the body of our page. This will allow users to load our image set at a time of their choosing, rather than adding to the initial file size of the web page. When the button is clicked, the function `loadImages` (**line 4**) initiates the HTTP request. Because only modern browsers and later versions of Internet Explorer possess the `XMLHttpRequest` object, we must first determine what object must be instantiated, to make the proper request. We do so by determining whether the `window` object possesses the `XDomainRequest` object (**line 6**). If the `XDomainRequest` object is available, we use our tertiary operator as a condensed `if/else` block, to instantiate an `XDomainRequest` instance. If, however, the evaluation to determine whether the `XDomainRequest` is available fails, our code will instantiate the more modern `XMLHttpRequest`. Once our `xhr` object is instantiated, we configure it with the appropriate request method and URL (**line 7**).

Because we are working with static content, rather than making a `POST` request, we will rely on the `GET` HTTP-Method to obtain the provided URI. Using the `onload` and `onerror` event handlers of the `xhr` object, we will monitor the state of the request. If the request is successful, the `onload` event handler will initiate the body of code that will obtain the request body from `responseText`. Knowing that the content provided within is JSON, we will obtain the plain/text from `responseText` and parse it utilizing the JSON Object (**line 9**). Once we obtain our data tree, we can reference the ordered list of images via the `images` key (**line 10**). From there, using a `for` loop, we iterate over each and every index possessed by our ordered list (**line 11**). By regarding each image object individually, we can obtain the values held within to construct the necessary markup that will be used to present our images.

In order to have our images display as a vertical list, we create a list item for each image. By using the `document.createElement` method, we are able to create HTML elements simply by providing the method with a string representing the tag we wish to create. In this case, as we wish to create a list item, we supply the `document.createElement` method with the string `li` and retain the reference to the `HTMLObject` returned (**line 13**); Next we create another `HTMLObject` (**line 14**), only this time it will be an element that represents the `img` tag. Using the reference to the image, we supply its attributes `src` and `alt` with the details that were extracted from the image objects (**line 15** and **line 16**). Next, we use the `appendChild` method to append the image as a child of our list item (**line 17**). Additionally, we add the list item as a child

of the body of the page, so that it will be visible to the document (**line 18**). This process is repeated until all images have been account for.

If the request fails, our application will alert us to the status code and the status description of the failure (**line 22**). Last, we invoke the request to begin by calling the `send` method on the instantiated `xhr` object (**line 24**). The preceding code should result as shown in [Figure 8-4](#).

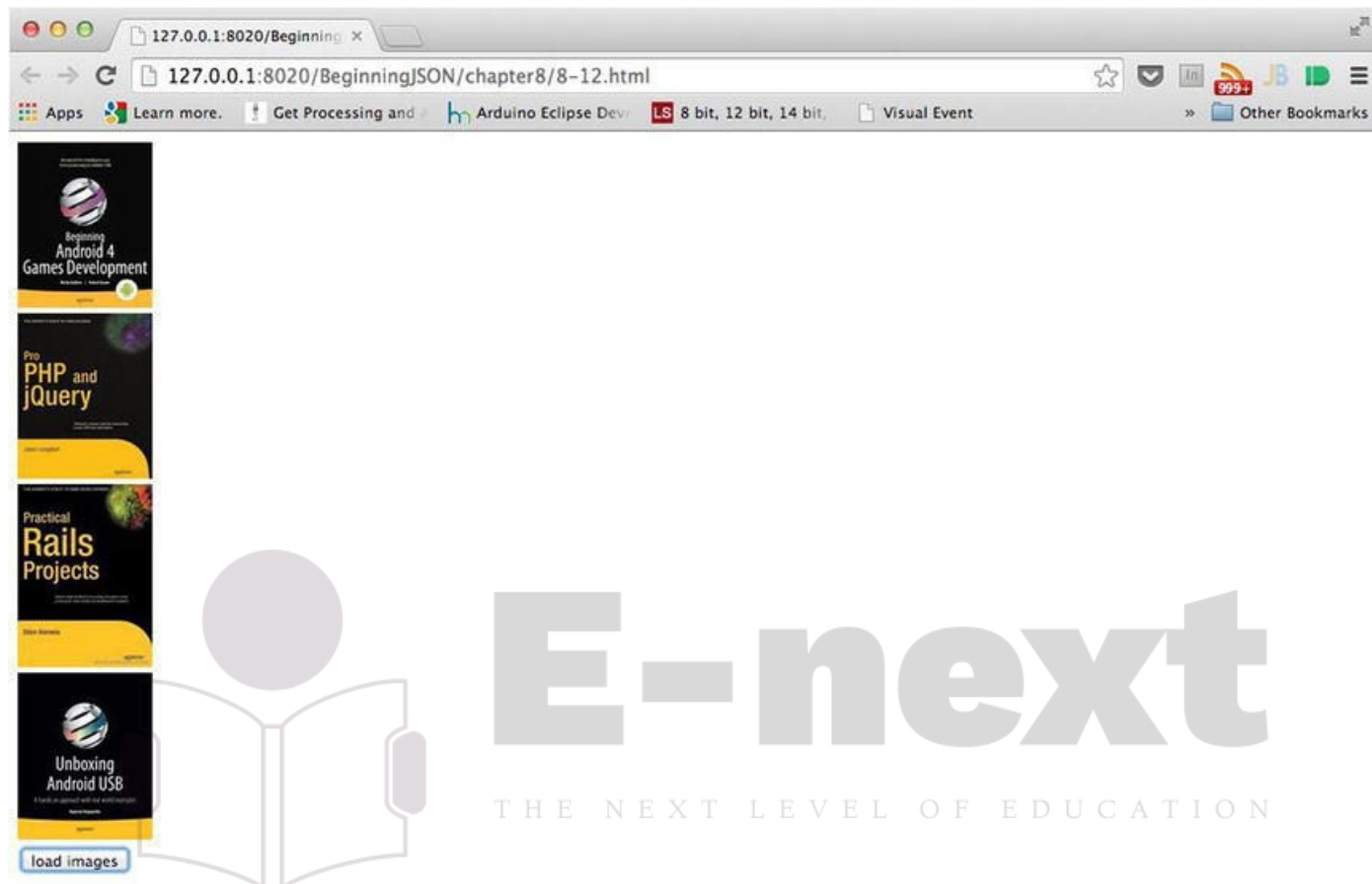


Figure 8-4. Use of Ajax to load and display images

It should be mentioned that the object that enables HTTP requests are strictly for making requests from a web server. Therefore, attempting to load files via Ajax locally will not work, unless they are run from a web server. Many web editors, such as WebStorm, Aptana, and VisualStudio, will run your local code within a temporary server, in which case, you would have no trouble following along with the provided source code.

Despite earlier discussions surrounding Content-Type and how the server should always configure it, you may have recognized that we did not have to configure the Content-Type, even though we were being provided JSON. Yet, if by some chance you were to have inspected the response header of [Listing 8-22](#) with the developer console, you would have witnessed that the Content-Type of the response read “application/json,” as indicated in [Figure 8-5](#).

▼ **Response Headers** view parsed

```
HTTP/1.1 200 OK
Date: Sat, 14 Jun 2014 17:38:17 GMT
Server: HttpComponents/4.1.3
Content-Length: 270
Content-Type: application/json
Connection: keep-alive
```

Figure 8-5. The response header for *imagesA.json* exhibits the configured Content-Type as *application/json*

As was mentioned in the history of JSON in [Chapter 4](#), Douglas Crockford’s formalization of JSON included the registered Internet media type *application/json*, in addition to the file extension *.json*. While a file extension doesn’t explicitly define the encoding of the content contained within, servers are able to infer Content-Types for commonly recognized file extensions. As JSON is the preferred interchange format, it should come as no surprise that most servers can equate the *.json* extension with the Content-Type of *application/json*. Therefore, the response is configured with the inferred Content-Type: *application/json*.

EXERCISE 8-2. LOAD MORE IMAGES

If you haven’t done so already, click the “load images” button from the previous listing two more times and take note of what’s occurring. With each click, a new *xhr* object is instantiated, initiating a new HTTP request. Providing the request is being fulfilled, the page should now display duplicates of the images loaded. As it serves little use to display duplicate content, rewrite the code from [Listing 8-22](#), so that each subsequent request will load a new JSON file containing no more than four different images.

You will find more images within the *img* folder that accompanies the source code for this chapter. (You can find the code samples for this chapter in the Source Code/Download area of the Apress web site [www.apress.com]). Reference these images within two more static JSON documents to be loaded in and displayed via Ajax. Feel free to duplicate the *images.json* file located within the *data* folder and simply replace the titles and URLs. Or, you can devise the JSON with the assistance of one of the editors discussed in [Chapter 4](#).

Summary

This chapter covered the essentials of the Hypertext Transfer Protocol (HTTP), which is necessary to comprehend when working with the interchange of data. By applying this knowledge, combined with the built-in objects that enable HTTP requests via JavaScript, we have been able to send, as well as receive, JSON in the background of our applications. Furthermore, using the techniques that make up Ajax, we were able to incorporate data without the need for full-page refreshes.

Ajax has surely broadened the scope of possibility for modern-day front-end

development. Conversely, its popularity has also resulted in an increase of security concerns. As browsers continue to improve measures to thwart malicious behavior, the ease of data interchange across origins has often been a difficult task to circumvent. In the upcoming chapters, you will not only learn how to overcome these issues from a server-side implementation, you will also set up a local server, so that you can employ these techniques.

Key Points from This Chapter

- A request/response possesses three components.
- A request is initiated by a client.
- A response can only be provided from a web server.
- The **GET** method is a safe method.
- The **POST** method is an unsafe method.
- The request URI identifies the resource that the request method applies.
- The current HTTP version is 1.1.
- General headers pertain to general information.
- Request headers communicate preferential information.
- Entity headers supply informative information regarding the supplied entity body.
- General headers and entity headers can be configured by both client and server.
- Response status codes are used to indicate the status of the request.
- The Content-Type header regards the MIME type of an entity.
- The Accept header is used to inform the server of the data types it can work with.
- The `XMLHttpRequest` Object enables HTTP requests from JavaScript.
- The `XMLHttpRequest` Object is available in all modern browsers as well as IE 8.
- `XMLHttpRequest` cannot be used for cross-origin requests in IE 8/9.
- `XDomainRequest` can be used for cross-origin requests in IE 8/9.
- `XDomainRequest` lacks the `setRequestHeader` method.
- `XMLHttpRequest` and `XDomainRequest` expose event handlers to notify of state.

- The `.json` extension is recognized by servers and will default the Content-Type to `application/json`.
- Custom headers begin with an `X`.
- Status code 200 represents a successful request.
- Prior to IE 10, `XMLHttpRequest` could only parse XML/HTML documents.

¹MDN: Mozilla Developer Network, “HTML in XMLHttpRequest,” https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/HTML_in_XMLHttpRequest, May 26, 2014.

²World Wide Consortium (W3C), “XMLHttpRequest,” www.w3.org/TR/2012/WD-XMLHttpRequest-20121206/#introduction, December 6, 2012.

³A. van Kesteren et al., “XMLHttpRequest,” dvcs.w3.org/hg/xhr/raw-file/tip/Overview.html, May 2014.



E-next
THE NEXT LEVEL OF EDUCATION