Session

8 Introduction to JSON

Concepts

Objectives

In this session, you will learn to:

- *Define and describe JSON*
- *Explain the uses of JSON*
- *List and describe the features of JSON*
- *Compare JSON with XML*
- *Describe the advantages of JSON*
- *Explain syntax and rules*
- *Explain literal notation in JavaScript and conversion to JSON*
- *Explain how to create a simple JSON file*

8.1 Introduction

While XML is adaptable to several application scenarios, it suffers from a few drawbacks, due to which it is not considered ideal in some situations. One such situation is using XML with Asynchronous JavaScript and XML (AJAX)-based applications.

AJAX refers to a development technique for designing interactive applications for the Web. These applications deliver a lively experience by making out-of-band, lightweight, client-server calls through a separate channel aiming to exchange information from remote services. In simple terms, updating and routing sequences occur outside typical client-server context, which results in a screen refresh while information is obtained in the background (out-of-band).

When updates are obtained by a browser, XML parses text by using the Document Object Model (DOM). This model has a high learning curve. Accessing simple data or modifying it through DOM triggers multiple method calls. Further, DOM works differently in each browser, due to which the parsing or modifying process requires implementing a complex programming scheme. However, there are chances for failure in cross-browser compatibility.

This triggered the need of a component that could easily incorporate a markup language to an HTML Web page as per the AJAX requirements in all mainstream browsers. This component was the JavaScript engine. Rather than sending AJAX updates in the XML format depending on complex DOM for parsing, a more discerning and natural format fitting natively to the JavaScript engine is certainly a better approach. Thus, such a format was introduced, which is called JavaScript Object Notation (JSON).

Concepts

8.2 What is JSON?

According to JSON.org, the official definition says JSON is 'a lightweight, text-based, and open standard data-interchange format.' It is readable and is derived from JavaScript language, as defined by ECMAScript Standard, Third Edition.

This text format is platform as well as language-independent. However, it implements conventions, which are known to the programmers of the C-based languages, such as C, C#, C++, Perl, JavaScript, and Java. JSON is not confined to AJAX applications only, but is usable in virtually any environment where applications need to store or exchange structured data.

JSON is more concise than tag-rich XML. The fact that it is derived from JavaScript and is beneficial for using it with Web apps that perform lots of data exchange. This data format is ideal for representing or storing semi-structured data.

Unlike other data models, such as XML or Relational Database, it is easier and quicker to learn JSON. Knowledge of JavaScript, Hypertext Preprocessor (PHP), or Java indicates that the developer already knows about JSON data structures.

Originally, JSON is based on a lenient subset of JavaScript and ECMAScript. While it is used with JavaScript, it permits a few unescaped characters in strings, which are considered illegitimate in ECMAScript and JavaScript.

Code for creating and parsing JSON data is available in several programming languages. Json.org, the official site for JSON, categorizes JSON libraries by language.

8.2.1 Uses of JSON

JSON is widely preferred to serialize and send data over a network, for instance, between a Web application and a Web server. Serialization transforms data structures to a format that is appropriate for transferring over a network or for storing in a memory buffer or file for later retrieval.

JSON, as a data-interchange format, is also used to facilitate communication between incompatible platforms. For instance, it becomes possible for a Java application on UNIX to send information to a Visual Basic application running on Windows by using JSON.

When XML was introduced during the late 90s, it was a boon for several business managers due to its intuitive HTML-like look. On the other hand, developers knew that parsing XML is not simple. Even today, while there are abundant programming libraries to simplify parsing, search for data in hierarchy, and extract node attributes, the learning curve is still high while handling the tags. Later, frameworks, such as Spring and Struts, offered some means to generate 'boiler plate' tags. At present, JSON offers a more recognizable way to handle tags and structure data.

JSON is also preferred for developing JavaScript based applications, such as Web sites and browser extensions. Even Application Programming Interfaces (APIs) and Web services use JSON to send public data.

Imagine a Web retail store offering a variety of products for sale. Each product will represent an element with attributes storing corresponding details such as description, price, and URLs to product images. For using this stored data in the site, the developer can fetch it without hard coding any product in HTML. One of the real life example is eBay wherein instead of hard coding each product, a template defines how such a site will look to the visitors. The required data is then fetched from the JSON file on server.

8.3 History of JSON

JSON was introduced to fulfill the requirement of real-time, stateful exchange of data between a server and client but without plugins such as Java applets. While Douglas Crockford originally specified this new format, the term was first coined at the State Software Company started by co-founders namely, Rebert F, Chip Morningstar, and Crockford.

The co-founders decided to develop a system that can harness the normal browser potential as well as offer an abstraction layer for designing stateful applications for the Web. These applications should maintain a persistent connection with a server by having two Hypertext Transfer Protocol (HTTP) connections open. These connections should be recyclable prior to the typical timeout of the browser, in case of no further exchange of data. One of the co-founders Morningstar developed the notion for such a State Application Framework.

On recognizing the initial potential of AJAX, companies such as digiGroups and Noosh implemented frames to send information to the browsers. However, this was done without refreshing the visual context of the real-time application, recognizing that such applications used only the basic HTTP, JavaScript, and HTML potential of Internet Explorer 5+ and Netscape 4.0.5+. For such a system, Crockford soon discovered that it is possible to implement JavaScript as an object-based format for messaging. An application system was then developed and sold to Sun Microsystems, EDS, and Amazon. In 2002, JSON.org was published. In 2005, Yahoo implemented JSON for its few Web services. A year later, even Google implemented JSON for feeds for its GData Web protocol.

In 2013, JSON became the ECMA-404 standard, the international standard of ECMA.

8.4 Features of JSON

In the world of enterprise applications, an increasing number of developers are choosing the JSON data format. The features of JSON justify why developers should use it in their Web-based applications. Following are the features of JSON:

- **Standard Structure**: The JSON format has a standard structure, making it easier for developers to write code.

Concepts

- **Simplicity**: The representation of data in JSON format is simple, which makes it simpler for a developer to comprehend the code. Even parsing becomes easy, regardless of the program language in use. The compact and simple JSON format allows quick transfer of data across platforms for reaching a variety of users across the globe.

- **Open**: JSON is similar to XML in terms of openness. It is publicly available and is not at the core of corporate/political tussles for standardization. One can easily take advantage of open structure for adding other elements when needed and adapt the system to hold new industry-specific terms. If these terms are in XML format, one can automatically convert them to JSON, as migrating is straightforward.

- **Self-describing**: JSON is self-describing, which means it is possible to describe one's own field names and their specific values. This enables applications to manipulate JSON data.

- **Lightweight**: Data in JSON format does not take much storage space. It is also easier to obtain and load such data quickly as well asynchronously without reloading the page. This makes AJAX applications efficient in terms of performance.

- **Scalable/Reusable**: If a developer needs to change the programming language on the server, it is easier to do so because the JSON structure remains the same regardless of the programming language in use. Let us assume a scenario, wherein a server side code displays an HTML list but the developer wants it in an HTML table. For this to happen, the developer would rewrite code at both server and client sides. Instead of this, by structuring data in JSON format, the developer only have to modify the code that displays the data on the screen. JSON structure remains the same across all programming languages, which ensures reusability and scalability.

- **Clean Data**: With a standardized structure, JSON is less susceptible to errors. The application obtaining JSON data is aware of how to get the data and what to expect, which is different from using HTML. Such a structure also helps the developer in offering robust cross-browser solutions, which is useful considering the advent of different browsers on mobile devices.

- **Efficiency**: While using JSON data structure, the previous outputted data list is created or cloned on the client side. This means that only the changed information is transferred to client on every request. This small optimization is likely to boost both performance and usability of Web applications, considering the swift increase in Internet connections.

- **Interoperability**: Interoperability is achieved when two or more applications, each coded using a different programming language, communicate with each other without any issues. JSON makes the developers to achieve interoperability, as it is supported by almost all the languages due to its simpler syntax.

- **Extensibility**: JSON allows storing only standard data such as numbers and text. On the other hand, XML allows storing any data type. While this is a benefit of using XML, it is also a con or a drawback, as it makes reading a bit more difficult. It all depends on the type of information to transfer. A document may need XML's flexibility for handling formatting elements such as images and graphs. Nevertheless, this flexibility or extensibility is not required for transferring classical data, which simply indicates JSON's simplicity.

- **Internationalization**: JSON implements the Unicode norm due to which generic tools can easily manipulate its structured data. Thus, this structured data is easy to distribute to a wide range of platforms as well as users.

8.5 JSON vs. XML

Both JSON and XML are capable of representing in-memory data in a readable textual format. Similarly, both of them are isomorphic, which means an equivalent one of the given piece of text is possible in the other format. For instance, while invoking a public Web service, a developer can state in the query string parameter whether the output should be in XML or JSON format.

Due to such similarities, it might not be simple to choose a suitable data exchange format from the two. This is where it is essential to consider the differentiating characteristics of both the formats and find out which one is more suitable for a particular application.

Table 8.1 compares the major characteristics of both formats.

Characteristic	JSON	XML
Data Types	Offers scalar data types and articulates structured data in the form of objects and arrays.	Does not offer any idea of data types due to which relying on XML Schema is essential for specifying information about the data types.
Array Support	Provides native support.	Expresses arrays by conventions. For instance, XML employs an outer placeholder element that forms the content in an array as inner elements.
Object Support	Provides native support.	Expresses objects by conventions, usually by combining attributes and elements.
Null support	Recognizes the value natively.	Mandates the use of xsi:nil on elements in an instance document along with the import of the related namespace.
Comments	Does not support.	Provides native support (via APIs).
Namespaces	Does not support namespaces and that naming collisions do not occur, as objects are nested or the object member name has a prefix.	Accepts namespaces to prevent name collisions and safely extend the prevalent XML standards.
Formatting	Is simple and offers more direct data mapping.	Is complex and needs more effort for mapping application types to elements.

Concepts

Characteristic	JSON	XML
Size	Has very short syntax, which gives formatted text wherein most space is taken up by the represented data.	Has lengthy documents, particularly in case of element-centric formatting.
Parsing in JavaScript	Needs no additional application for parsing (JavaScript's eval() function works well).	Implements XML DOM and requires extra code for mapping text to JavaScript objects.
Parsing	Has JSONPath for selecting specific sections of the data structure but is not widely used.	Has XPath specification for doing the same in an XML document and is widely used.
Learning curve	Is not steep at all, due to familiarity with the structure and the underlying dynamic programming languages.	Is steep with the need to know several technologies and concepts such as XPath, XSL Transformations (XSLT), DOM, Schema, and Namespaces.
Complexity	Is complex.	Is more complex.
Schema (Metadata)	Has a schema but is not widely used.	Uses many specifications for defining a schema, including XML Schema Definition (XSD) and Document Type Definition (DTD).
Styling	Has no special specification.	Has XSLT specification for styling an XML document.
Querying	Has specifications such as JSON Query Language (JAQL) and JSONiq but are not widely used.	Has XQuery specification that is widely used.
Security	Is less secure, as the browser has no JSON parser.	Is more secure.

Table 8.1: JSON Versus XML Characteristics

As a new format for data exchange, JSON does not have excessive vendor support, which XML has today. However, JSON is gaining vendor support quickly. Table 8.2 highlights the primary differences in terms of support for JSON and XML.

Support	JSON	XML
Tools	Lacks rich tools, such as editors.	Has an established set of tools from several vendors.
Platform and Language	Has open-source parsers and formatters in many languages and on many platforms.	Has both commercial and open-source parsers and formatters.

Support	JSON	XML
Integrated Language	Is supported only in JavaScript/ ECMAScript.	Is under experiment by many vendors for support within languages.

Table 8.2: Support Differences Between JSON and XML

Let us now see how XML and JSON format appears while storing the details of three students. These details are stored in a text-based format for later retrieval.

Code Snippet 1 shows the XML code for this scenario.

Code Snippet 1:

```
<students>
 <student>
 <name>Steve</name> <age>20</age> <city>Denver</city>
 </student>
 <student>
 <name>Bob</name> <age>21</age> <city>Sacramento</city>
 </student>
 <student>
 <name>Peter</name> <age>22</age> <city>Washington</city>
 </student>
</students>
```

Code Snippet 2 illustrates the earlier Code Snippet 1 now presented in JSON format.

Code Snippet 2:

```
{"students":[
 {"name":"Steve", "age":"20", "city":"Denver"},
 {"name":"Bob", "age":"21", "city":"Sacramento"},
 {"name":"Peter", "age":"22", "city":"Washington"}
]}
```

From these snippets, it is clear that JSON is more lightweight as well as compact than XML.

Concepts

8.6 Advantages of JSON

JSON is primarily used for serializing and de-serializing data being sent to and from an application made using JavaScript. Considering this fact, the advantages of JSON relate to the use of JSON over other options for serialization and de-serialization.

Typically, in an XML environment, the sender encodes the data to be serialized, by using a DTD that the recipient can comprehend. This task involves a lot of additional padding around the data, regardless of which DTD in use. Thus, the size of the XML documents is significantly larger than the actual data values to be sent. Then, the recipient obtains the XML data stream and decodes it for putting in the memory.

Comparatively, this process of serialization if done through JSON is quicker and more compact due to the format's inherent structure. JSON structure uses standard programming data types whose structure is simple to understand. Further, the encoding mechanism is efficient enough to add only minimum characters required to reflect the structure and data values. When the recipient obtains this serialized data, the only procedure required is to evaluate the text through a compatible function in any programming language. For example, evaluation can happen by using eval function of JavaScript.

The other standard to compare is Yaml Ain't Markup Language (YAML) that serializes complex data sets without a DTD and relies on a simpler parser than XML. Even such a parser consumes more time than JSON to output larger streams of serialized data.

Following are some more benefits of using JSON:

- JSON parses faster than XML and YAML.
- JSON is simpler to work with some languages such as PHP, Python, and JavaScript.
- JSON is simpler to map to an object oriented system, as it is data-oriented.
- JSON is a valid subset of even Python and YAML apart from JavaScript.
- JSON easily differentiates between the string '2' and the number '2' as well as between a single item and a collection of size one (array).
- JSON is an ideal data exchange format, and not an ideal document exchange format as XML. Thus, it is not necessary for JSON to be extensible for exchanging documents by defining new tags for representing data.

8.6.1 Limitations

While the benefits of JSON are appealing, there are limitations too. The most significant limitation is a bit of difficulty in reading the JSON format, which also requires keeping every single bracket and comma at its correct place. This is also applicable to XML. However, JSON's structure embracing

complicated-looking syntax such as}]} at the end can scare the beginners as well as make debugging a complicated task, initially. This can be understood by examining a complex JSON structure, as shown in Code Snippet 3.

Code Snippet 3:

```
{
"problems": [{
 "Diabetes":[{
 "medications":[{
 "medicationsClasses":[{
 "className":[{
 "associatedDrug":[{
 "name":"Asprin",
 "dose":"",
 "strength":"500 mg"
 }],
 "associatedDrug#2":[{
 "name":"Lisinopril",
 "dose":"",
 "strength":"500 mg"
 }]
 }],
 "className2":[{
 "associatedDrug":[{
 "name":"Metoprolol",
 "dose":"",
 "strength":"500 mg"
 }]
 }]
 }]
 }]
 }]
 ]
}
```

Such a complex nested structure can appear scary to a newcomer, however, with practice, one can gain familiarity and a better understanding of how JSON data is stored.

Concepts

8.7 JSON Data and Syntax Rules

JSON data is represented as name/value pairs. Such a pair contains a field name in double quotation marks suffixed by a colon after which a value is stated. A JSON value can be an integer, a Boolean value, a string, an array, an object, and null.

Example 1 shows a name/value pair in JSON.

Example 1:

```
"lastName":"George"
```

In Example 1, lastName is the field name in string format and George is the string value.

JSON supports two data structures namely, ordered list of values and unordered set of name/value pairs. The ordered list can be an array, vector, list, or sequence; while collection of name/value pairs can be recognized in a different way, depending on the programming language. This collection can be a keyed list, object, record, struct, or a hash table.

Both these data structures are universal, which means almost all modern programming languages accept them. It makes sense that JSON being an interchangeable format supported by different programming languages also accepts these structures.

In JSON, these data structures take up predetermined forms. A sequenced set of values is considered an array. An array starts with the open square bracket ([) and ends with the close square bracket (]). Within these square brackets, elements also known as values are added, each being separated by a comma. The visual representation of this concept is shown in figure 8.1.

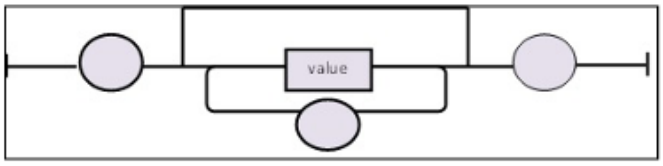

Figure 8.1: Array Representation in JSON

An unordered arrangement of name/value or key/value pairs is considered an object. An object starts with open curly bracket ({) and ends with close curly bracket (}). Within these curly brackets, each name is followed by a colon to separate it from its value and each key/value pair is followed by a comma.

The visual representation of this concept is shown in figure 8.2.

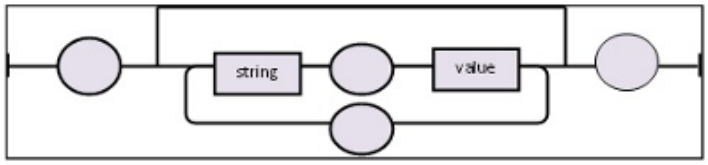

Figure 8.2: Object Representation in JSON

Code Snippet 4 illustrates the creation of a JSON object.

Code Snippet 4:

```
{
 "Name":"Janet George",
 "Street":"123 Ashley Street",
 "City":"Chicago",
 "Code":"60604"
}
```

In Code Snippet 4, it is easy to identify that Name, Street, City, and Code are the names or keys of the object, each having a value. Both keys and values are in quotes. Each key is separated from its value via a colon. While it is not mandatory to quote a numeric value, strings values should always be quoted.

Code Snippet 5 would result in an error, because population is not specified in quotes.

Code Snippet 5:

```
{
 "code": "AL",
 "name": "Albania",
 population: "2986952"
 }
```

The JSON syntax is derived from the object notation syntax of JavaScript. Following syntax rules are acceptable even in JSON:

- Data is in the form of name/value pairs and is separated by commas.
- Names and values are separated using colon.

Concepts

- Curly braces contain objects.
- Square brackets contain arrays.

8.8 Understanding the Literal Notation in JavaScript

In a programming language, literals are used to express fixed values literally, such as the constant integer value of 1 or the string value "Steve". You can use literals in most languages for stating an expression. A few examples of such an expression include an input parameter passed for invoking a function, a part of a condition in a control statement, and a value in a variable assignment statement.

Various programming languages accept different types of literals. Most of them support literals for scalar data types such as floating-point numbers, integers, strings, and Boolean. However, JavaScript also accepts literals for structured data types such as objects and arrays. Therefore, it is possible to have a concise syntax for on-demand initialization of the aforementioned structured data types.

In JavaScript, an array literal can contain zero or more expressions in a sequence and that each such expression is an element of that array. These elements are contained in square brackets ([]) and each is separated by a comma. Such an array declaration is referred to as literal notation. To declare it, you start by inserting the opening square bracket, adding each element or value followed by a comma, and ending it with a closing square bracket.

Consider that you want to create an array in JavaScript to represent five fruit names.

Code Snippet 6 defines such an array using literal notation.

Code Snippet 6:

```
var fruits = ["Pineapple", "Orange", "Melon", "Kiwi", "Guava"];
alert (fruits[4] + " is one of the " + fruits.length + " fruits.");
```

Code Snippet 7 defines a similar array without using literal notation.

Code Snippet 7:

```
var fruits = new Array();
fruits[0] = "Pineapple";
fruits[1] = "Orange";
fruits[2] ="Melon";
fruits[3] = "Kiwi";
fruits[4] = "Guava";
```

JavaScript allows creating objects. While array is an ordered sequence, an object refers to an unordered set of zero or more members existing in the form of name and value pairs. Herein, a name is in the form of a string, while the corresponding value can be a Boolean, null, string, array, object, or number.

Similar to array literal notation, one can create objects using literal notation. An object literal notation allows creating objects by defining its members. The list of members, as name and value pairs, is included in curly braces ({}) and each pair is followed by a comma. However, within each member, the name is separated from the value by a colon (:).

Code Snippet 8 initializes an object with three members namely, Street, City, and PostalCode.

Code Snippet 8:

```
var Address = {
             "Street": "123 New Mansion Street.",
             "City": "Denver",
             "PostalCode": 80123
 };
alert ("The product will be sent to postal code " + Address.PostalCode);
```

Code Snippets 6, to 8 showed use of numeric and string literals within object and array literals. However, you can even express the whole grid by using the notation recursively. This means that member values and elements can have object and array literals.

Code Snippet 9 illustrates an object called contact that has an array named ContactNumbers as a member. This array has a list of objects.

Code Snippet 9:

```
var contact = {
             "Name": "John Doe",
             "PermitToCall": true,
             "ContactNumbers": [
                 {
                     "Place": "Home",
                     "Number": "445-563-3456"
             },
                 {
                     "Place": "Work",
                     "Number":"445-563-7777 Ext.244"
      }
                 ]
```

Concepts

```
      };
if (contact.PermitToCall)
{
      alert("Call " + contact.Name + " at " +
 contact.ContactNumbers[0].Number);
}
```

8.9 From JavaScript Literals to JSON

JSON was actually created from a section of the object literal notation in JavaScript. While the syntax for literal values in JavaScript is flexible, JSON follows stricter rules. The JSON standard mandates for the object member name to be a valid JSON string enclosed in quotation marks. On the other hand, JavaScript enables separating the object member names by apostrophes or quotation marks. It also permits to skip quoting altogether, until the name is not the same as any reserved keyword.

Similarly, a member value or an array element in JSON is confined to a restricted set. However, in JavaScript, the two can refer to any valid expression, including definitions and function calls. Despite these limitations, JSON's simplicity is the appeal.

When formatted as per the JSON standard, a message contains a top-level array or object. The elements of that array or values of those object members can be null, Boolean, numbers, strings, arrays, or objects. Date/time literal is not supported by JSON. The reason is because the literal is also absent in JavaScript, wherein date and time values are stored via the Date object.

As a solution for JSON, many applications use a number or string data type to specify date and time values. This string is usually in the ISO 8601 format. If the data type is number, the value is expressed as the milliseconds passed since the January 1st, 1970 midnight in Universal Coordinated Time (UTC). This is only a convention; it does not belong to the JSON standard.

For exchanging data, it is essential to see the documentation of receiving application to know how date and time values are encoded in a JSON literal. For instance, in AJAX, a JSON string is used to represent a DateTime value.

8.10 Creating a Simple JSON File

It has been already stated that JSON is evolved from JavaScript and is compatible with any object-oriented language.

Code Snippet 10 illustrates a JSON file stored with .json extension, which shows how a JavaScript object is converted to JSON string.

Code Snippet 10:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
<script type="text/javascript">
     //Step 1 :: Defining a JavaScript Object
     var jsonObject = {
         "Name" : "James Samual",
         "Address" : "1960 W CHELSEA AVE STE",
         "City" : "ALLENTOWN PA",
         "PostalCode" : "18112"
     };
     //Step 2 :: fetching a JavaScript elements
     document.write("<hr>");
     document.write("<h2><font style=\"color: blue\">Name</font>::"
             + jsonObject.Name + "</h2> ");
     document.write("<hr>");
     document.write("<h2><font style=\"color: blue\">Name</font>::"
             + jsonObject.Address + "</h2> ");
     document.write("<hr>");
     document.write("<h2><font style=\"color: blue\">City</font>::"
             + jsonObject.City + "</h2> ");
     document.write("<hr>");

     document.write("<h2><font style=\"color: blue\">Name</font>::"
             + jsonObject.PostalCode + "</h2> ");
     document.write("<hr>");
     //Step 3 :: Converting a JavaScript object to JSON.
     document.write("<h2><font style=\"color: blue\">JsonFormat</font>::"
             + JSON.stringify(jsonObject) + "</h2> ");
     document.write("<hr>");
</script>
</head>
<body>
</body>
</html>
```

Concepts

In Step 1 of Code Snippet 10, a simple JavaScript object is defined and is referenced as 'jsonObject'. This object contains four properties in the form of name/value pairs, which indicates the address of an individual.

In Step 2, each property is fetched for displaying it along with its corresponding value onto the HTML page.

In Step 3, the JSON.stringify() function converts a JavaScript value to a JSON string. It returns a string that holds serialized JSON text. Following is the syntax of this function:

Syntax:

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

where,

`value` identifies a JavaScript value, which is generally an object or array to be converted and it is a mandatory parameter.

`replacer` represents a function or array that transforms the results and is optional.

`space` adds indentation, line break, and white space characters to the returned JSON string for making it easier to read and is optional.

Working of replacer Parameter

The replacer parameter can take a function or an array as the value.

Following is the description of what happens when the parameter takes these values:

- If the value of the replacer parameter is a function, it is invoked and each member's key and value is passed to that function. The return value is utilized rather than the original value. If the invoked function returns undefined, the code excludes the member. An empty string, which is signified as "", acts as the root object's key.

- If the value of the replacer is an array, the code converts only the members having key values to JSON string. The sequence in which these members undergo conversion is the same as the sequence of the keys in the array. If the value argument is an array, the replacer array is overlooked.

Working of space Parameter

The space parameter can be a number or string. You can even completely omit it. Following is the description of what happens when the parameter takes these values:

- If space is omitted, the outputted JSON string has no additional white space.
- If space is a number, it indicates the number of white spaces to be inserted at each level in the JSON string. If the input value is more than 10, the JSON string is still indented by 10 spaces. No space is used if the value is below 1.
- If space is a string that is not empty, such as '\t', it signifies to insert white spaces in the JSON string after that many characters at each level. If this input string is more than 10 characters, then indentation occurs by using the first 10 characters.

Figure 8.3 shows the output of Code Snippet 10.

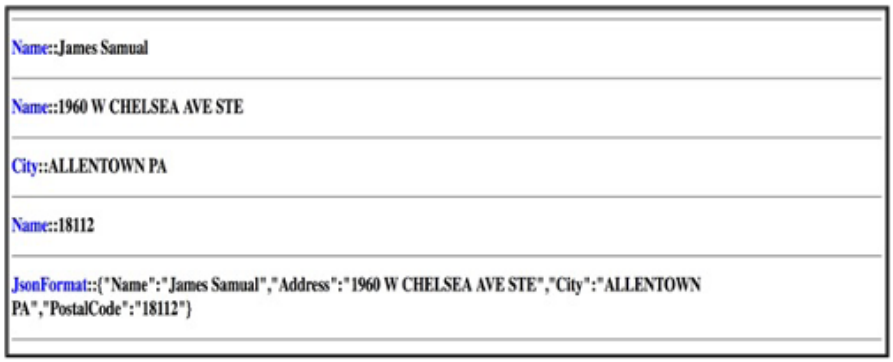

Figure 8.3: JavaScript Object to JSON Conversion Output

Concepts

Summary

- JSON is a text-based and open standard format derived from JavaScript and ECMAScript, for interchanging data between incompatible platforms or applications.
- JSON is lighter, more scalable, less complex, easier, and quicker to learn, and faster to process than XML.
- JSON works independently of any language or platform. It is compatible with most modern languages, including the object-oriented ones.
- JSON is preferred for serializing and de-serializing data to be exchanged to and from a Web client and a Web server.
- Data in JSON format is represented as name/value pairs.
- JSON supports two data structures namely, array as ordered list and objects as unordered set of name/value pairs.
- An object literal notation allows creating objects by defining its members.
- While syntax for JavaScript's literal values is flexible, JSON follows stricter rules for array elements and object member names and their values.
- A JSON file is saved with .json extension.
- The JSON.stringify() function is used to transform a JavaScript object to a JSON string.

Check Your Progress

1. JSON is derived from _____.

   a. Perl

   b. HTML

   c. ECMAScript

   d. XML

2. Which of the following is the JSON feature facilitating communication between two applications written in different programming languages?

   a. Extensibility

   b. Openness

   c. Self-describing

   d. Interoperability

3. Which of the following XML feature is not supported by JSON?

   a. Formatting

   b. Namespaces

   c. Objects

   d. Schema

Check Your Progress

4. Which of the following JSON syntax component needs to be a valid string?

   a. Array element

   b. Object value

   c. Object member

   d. Object literal

5. Which of the following will be the result of the code?

```
var jsonObj = {
          "Firstname": "Stephen",
          "City": "NewYork"
          };
document.write("<hr>");
document.write("<h2>City::"   + jsonObject.City + "</h2> ");
```

   a. NewYork

   b. City

   c. Error

   d. City: : NewYork

5. Which of the following functions convert a JavaScript value to a JSON string?

   a. JSON.string()

   b. JSON.converttostring()

   c. JSON.convert()

   d. JSON.stringify()

Session 9

Work with JSON Data

Concepts

Objectives

In this session, you will learn to:

- *Identify data types supported by JSON*
- *Explain how to represent complex data with JSON*
- *Explain how to execute serialization and de-serialization of JSON with JavaScript*
- *Describe tools and editors that can be used to work with JSON*
- *Describe the syntax and schema of a JSON document*

9.1 JSON Data Types

JSON supports two primary data types namely, primitive and structure. Primitive types include string, number, Boolean, null, and white space; while, structure types include object and array. A visual representation of this is shown in figure 9.1.

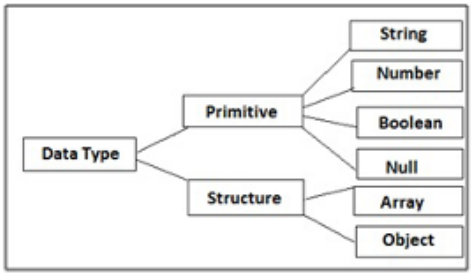

Figure 9.1: Classification of Primary Data Types

In JSON, it is not necessary to declare the type of a variable before using it. The type is recognized automatically during parsing phase. As the parser is responsible for processing the program, element mapping is also dependent on it. Let us now understand each of the primitive and structure data types in detail.

9.1.1 Number

Number is a floating-point format (double precision), similar to the one in JavaScript. However, in JSON, Number data type does not accept Octal, hexadecimal, NaN, and Infinity values.

Concepts

Table 9.1 shows different Number types.

Function	Description
Integer	0, 4, 7, -1, -5, and -8
Fraction	.3, .9, .5, .2, .8, and .9
Exponent	e, e+, e-, E, E+, and E-

Table 9.1: Number Types

Following is the syntax for specifying a numeric value in JSON:

Syntax

```
{"string": number_value, .......}
```

where,

`string`: refers to a sequence of characters within double quotation marks.

Code Snippet 1 illustrates how to specify a Number data type. It stores names of students along with their corresponding age.

Code Snippet 1:

```
{
 "Bob":18,
 "Christina":17,
 "John":16
}
```

9.1.2 String

A series of zero or other Unicode characters within double quotes and backslash escapes is termed as a string. A character denotes a single character string, meaning the length of that string is 1. The string data type is similar to a C or Java string. Strings are delimited with double-quotation marks and support a backslash escaping syntax.

Table 9.2 shows the different string types.

Type	Name	Description

"	Double quotation	All strings should be double-quoted. However, double quotes can also be used as a part of the value as \".
		{
		"Name":"James Butt"
		}

\	Reverse solidus	Reverse solidus or backslash is used as an escape character to represent a single quotation mark or a few other characters in value. It is always suffixed with some special character such as double quotes. Following is an example of displaying R in double quotes by using reverse solidus:
		{
		"Name":"James\"R\"Bett"
		}

/	Solidus	Solidus is used as a part of test value or a Uniform Resource Locator (URL).
		{
		"Name":"James / Bett"
		}

b	Backspace	Backspace is used to represent a backspace character. { "Name":"James \b Bett"}
f	Form feed	Form feed is used to represent a page break, especially while printing. { "Name":"James \f Bett"}
n	New line	New line is used to start displaying on a new line. { "Name":"James \n Bett"}
r	Carriage return	Carriage return is used to specify the action of the entry key. { "Name":"James \r Bett"}

Concepts

Type	Name	Description
t	Horizontal tab	Horizontal tab is used to specify the tab space in between a value. `{ "Name":"James \t Butt"}`

u	Four hexadecimal digits	Four hexadecimal digits indicate Unicode in the string so that is not treated as a simple string.
		`{`
		`"CountryNameInUnicode":"\u20013 \ u22283"`
		`}`

Table 9.2: String Types

Following is the syntax for specifying a strong value in JSON:

Syntax

```
{"string":"string value", .......}
```

Code Snippet 2 shows how to specify a String data type.

Code Snippet 2:

```
{
 "FirstName":"Bob",
 "LastName":"Fernandes",
 "City":"Washington",
 "Country":"USA",
 "SpecialChar":"***$$$"
}
```

9.1.3 Boolean

A Boolean has only two values namely, true and false. Using quotes for Boolean values treats them as String values.

Syntax

```
{string: true/false, .......}
```

Code Snippet 3 shows how to specify a Boolean data type.

Code Snippet 3:

```
{
 "isValidJson":true,
 "isInvalidJson":false
}
```

9.1.4 Null

The null data type is an empty type. Code Snippet 4 shows how to specify the Null JSON data type in JavaScript code.

Code Snippet 4:

```
var p = null;
if(p == 5)
{
document.write("<h1>value is 5</h1>");
}
else
{
document.write("<h1>value is null</h1>");
}
```

9.1.5 White Space

A white space can appear between the characters in string values to make code more comprehensible.

Syntax

```
{string:" ",....}
```

Code Snippet 5 shows how to specify white space.

Code Snippet 5:

```
var firstname = " Raze";
var secondname = " Sand";
```

Concepts

9.1.6 Array

Arrays allow storing various values of the same type in one variable. In other words, an array stores sequential elements of a similar type.

Following are the characteristics of an array in JSON:

- It is a sequential collection of values, not necessarily of the same type.
- It is enclosed in square brackets [.].
- Each value is set apart by comma (,).
- Indexing begins at 0 or 1.

Syntax

```
[value, .......]
```

Code Snippet 6 shows how to define an array having multiple strings.

Code Snippet 6:

```
[
 "C",
 "C++",
 "JAVA",
 "COBOL",
 "VB"
]
```

Code Snippet 7 shows how to define two named arrays, one containing strings and other having numbers.

Code Snippet 7:

```
{
 "Programming Languages": [
 "C",
 "C++",
 "JAVA",
 "COBOL",
 "VB"
```

```
 ]
}

{
 "EvenNum": [
 2,
 4,
 6,
8,
 10
 ]
}
```

9.1.7 Object

An object is an independent data type, having its own attributes. It can often be mapped to real world entities. For example, a chair is an object with its own attributes such as specific hue, mass, shape, and material used to build it. Similar to a chair in real world, Object data type has its own attributes, which describe characteristics of the object. In JSON, attributes are denoted as key and corresponding values of an object are denoted as values. Some main characteristics of this data type are as follows:

- It is a non-sequential (having no order) set of key/value pairs.
- It is enclosed in curly braces {.}.
- Each key/value pairs are set apart by comma (,) and every key is proceeded by colon (:).
- The keys are only strings, each differing from the other.
- It is used when the key names are random strings.

Common examples of objects are: Book, Employee, Student, Person, and so on.

Syntax

```
{string : value, .......}
```

Concepts

Code Snippet 8 shows how to specify a JSON Object that can represent a software learning course.

Code Snippet 8:

```
{
"id": "5634A",
"language": "C",
"cost": 800
}
```

9.1.8 JSON Value

In JSON, value can be of any primitive and structure data type. A JSON value can be a number, string, Boolean such as true or false, null, object, or an array.

Code Snippet 9 shows some examples of JSON values.

Code Snippet 9:

```
var emp-no = 1234;
var name = "Sherill";
var experience = null;
```

9.2 Storing JSON Data in Arrays, Objects, and Nested Data

JSON not only allows storing different values in arrays and objects but also enables you to nest them. Let us understand how you can use these structured data types to store different values and nest them.

9.2.1 Storing Different Values in Arrays

In general, arrays store various values of the same type in one variable, that is, all the elements must be strings, numbers, or Boolean. This holds true for all programming languages. However, this rule does not apply to arrays in JSON. JSON arrays do not have this limitation and that they can store elements of different types.

Code Snippet 10 shows an array holding different types of values.

Code Snippet 10:

```
[
 456,
 "Dog",
 123,
 "Frog",
 true
]
```

In Code Snippet 10, elements 0 and 2 in the array are of the type Number, while elements 1 and 3 are of String type. Element 4 is a Boolean type. In JSON, arrays can also contain other arrays. This is called nesting of arrays. Figure 9.2 shows an array holding four arrays.

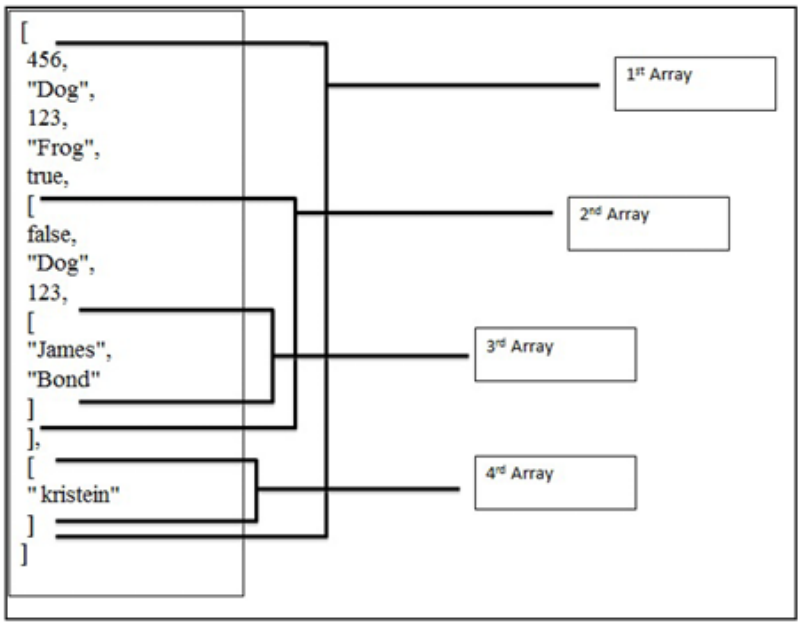

Figure 9.2: Arrays with an Array

In figure 9.2, you can add a new array as the fifth element. You can move any level of the hierarchy and add any number of arrays at any level.

9.2.2 Nesting Array in a JSON Object

A JSON object can have values of any data types, which means it can even have arrays nested within it. Consider that you are building a Web service for a blog application in a business scenario. You will build a JSON object having a nested array. The blog stores user details, such as `firstName`, `lastName`, `blogContent`, `blogCreatedDate`, and `blogID`.

Concepts

Code Snippet 11 shows a JSON blog object storing user details along with an array of tags. The array contains the tag details of blog contents in an array. These tag details help in searching the content on a blog.

Code Snippet 11:

```
{
 "blogID":12345,
 "firstName":"James",
 "lastname":"Gunn",
 "blogContent":"This is my first JSON object and it looks simple",
 "tags":[
 "first",
 "JSON",
 "looks",
 "simple"
 ]
}
```

9.2.3 Nesting Objects in a JSON Object

In Code Snippet 11, the first name and last name are attributes of the user and not those of the blog. Therefore, it is ideal to maintain these user details as a user object, as shown in figure 9.3. In this figure, a blog object contains another object called user as well as an array of tags. It is possible to have any number of objects within a single object but only with one object per key.

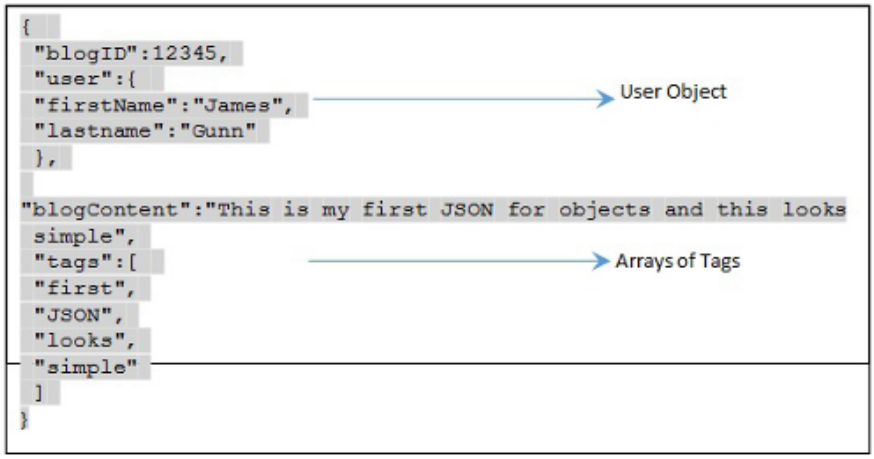

Figure 9.3: Object Nested Within an Object

9.2.4 Building Nested Arrays Object

Let us now consider that after reading a blog, a user wants to leave comments. As multiple users may comment on the blog, it is essential to have information such as the author of the blog, e-mail, and body of the comment. For storing these details, comments should be created with the defined attributes. Within it, there can be numerous comments, which can be represented as arrays.

Code Snippet 12 shows the object of a blog, which contains its relevant details such as blog ID, user name, blog content, and comments. Herein, `user` is an object, while `tags` and `comments` are arrays. The `comments` array includes a list of two objects, each having `body`, `email`, and `author` details.

Code Snippet 12:

```
{
 "blogID":12345,
 "user":{
 "firstName":"James",
 "lastname":"Gunn"
 },
 "blogContent":"This is my first JSON for objects and this looks simple",
 "tags":[
 "first",
 "JSON",
 "looks",
 "simple"
 ],
 "comments":[
 {
 "body":"Nice blog",
 "email":"michaelbolt@rediff.com",
 "author":"Cassi Heal"
 },
 {
 "body":"You could have explained in more detail.",
 "email":"SaraFord23@google.com",
 "author":"Bao Ziglar"
 }
 ]
}
```

Considering Code Snippet 12, it is easy to conclude that a developer can expand the code up to any level in the hierarchy of a JSON document to make it look complex.

Concepts

9.3 Data Structures Supported by JSON

Data structures are an integral part of all the programming languages. In JSON, the built-in data structures can be used to develop other data structures. There are two data structures supported by JSON, which are as follows:

- **Collection of Name/Value Pairs**: This type is supported by different programming languages, in the form of objects.
- **Ordered List of Values**: This type includes arrays, lists, and vectors.

9.3.1 Collections: Maps, Sets, and Lists

Sets and Lists correspond to a set of objects, whereas Maps relate a value to an object. Many programming languages support these data structures. Following is the description of each of these data structures in JSON:

- **List**: Is a more flexible version of an array. The elements in a list follow a defined order and copies or duplicates are allowed. Additionally, the elements are positioned in a specific location.

  In JSON, a list of values or objects can be depicted using arrays. When you club the elements as arrays, the JSON parser arranges them in Lists.

- **Set**: Is a collection without duplicate elements. In JSON, it is not possible to avoid replicas. Arrays can still be used to represent sets; however, the parser removes the duplicates while serializing the data.

- **Map**: Is a type of data structure that mainly aids in quick searching of data. It accepts data in the form of key and value pairs and each key is distinct. Each key tends to map to a corresponding value.

A JSON object can also represent these data structures. Code Snippet 13 shows a map object along with its JSON representation as a JSON object.

Code Snippet 13:

```
Map map = new HashMap();
map.put("Organization", "Apple Inc");
map.put("Founder", "Steve Jobs");

{
 "Organization":"Apple Inc",
 "Founder":"Steve Jobs"
}
```

Finally, it is the parser that takes up the main role of serializing and de-serializing JSON data. The JSON parser maps the core data structure with the internal JSON structure.

9.4 JSON Schema, Metadata, and Comments

JSON schema specifies the rules that defines the structure of a JSON document. This helps in validating and verifying the data in the document. The self-explanatory schema explains which JSON data is needed for any intended application and how to work with that data.

Based on XML Schema Definition (XSD), JSON schema is just an Internet draft; the latest version 4 expired on August 4th, 2013. Though it has expired, application developers still use the concept for generating JSON document, which is a contract between different applications or producers and consumers of Web services. Following are the two main aspects that define why a JSON schema language is required:

- **To specify JSON data structures**, which is handy when the JSON-based Web services need to be made accessible to a large audience.
- **To validate JSON data structures**, which is handy when validating JSON documents from other applications. This helps to avoid parsing an invalid data structure in valid JSON documents.

9.4.1 Schema Overview

`Application/schema+json` is the media type described by JSON schema and prescribes the design of JSON documents. It offers provision for defining the structure of the documents with respect to the permitted values, descriptions, and decoding connections to other resources. The JSON schema format is arranged in following individual definitions:

- **Core Schema Specification**: Explains a JSON structure and states valid elements in it.
- **The Hyper Schema Specification**: Explains the elements in a JSON structure that can be considered as hyperlinks. Hyper schema is based upon the JSON schema and prescribes the hyperlink structure in a JSON document. This allows users to pilot through JSON documents easily, according to their schemas.

Collectively, JSON schema serves as a meta-document that helps in defining the needed type and limitations on values. It also defines the meaning of these values for describing a resource and determining hyperlinks in the representation.

Let us assume that you are interacting with a JSON-based Library Management System (LMS). This system has a number of books, each having an id, name, number of copies, and an optional set of tags.

Concepts

Code Snippet 14 represents these details in a JSON object.

Code Snippet 14:

```
{
 "id":1,
 "title":"Wings that can't fly",
 "author":"James witwitcy"
 "noOfCopies":12,
 "tags":[
 "children",
 "heart",
 "friendship"
 ]
}
```

While going through the code, a user can have the following questions unanswered:

- What is an id?
- Is title mandatory?
- Can the number of copies be zero?
- Are all tags of type String?

The JSON schema is similar to a framework that mentions as well as standardizes the answers to these questions for JSON data. To understand this, let us consider a simple JSON schema.

Code Snippet 15 shows a simple JSON schema.

Code Snippet 15:

```
{
 "$schema":" http://json-schema.org/draft-04/schema#",
 "title":"LibrarySystem",
 "description":"A library Management system for schools and collages",
 "type":"object"
}
```

In Code Snippet 15, the defined schema has four attributes, which are called keywords. The keyword `$schema` states that this schema has been designed with respect to the draft v4 specification. The `title` and `description` keywords are descriptive only. They provide the actual objective of this schema, which is defining a product. The data being validated will not face any constraints from these two keywords. The constraint on the JSON data is put forth by the keyword `type`. Herein, the constraint is that the product type has to be a JSON Object.

Code Snippet 16 answers the unanswered queries by extending the schema in Code Snippet 15.

Code Snippet 16:

```
{
 "$schema":" http://json-schema.org/draft-04/schema#",
 "title":"LibrarySystem",
 "description":"A library Management system for schools and collages",
 "type":"object",
 "properties":{
 "id":{
 "description":"The unique ID for a Book ",
 "type":"integer"
 },
 "title":{
 "description":"Title of the Book",
 "type":"string"
 }
 },
 "noOfCopies":{
 "type":"number",
 "minimum":0,
 "exclusiveMinimum":true
 }
 },
 "tags":{
 "type":"array",
 "items":{
 "type":"string"
 },
 "minItems":1,
 "uniqueItems":true
 }
 },
 "required":[
 "id",
 "title",
 ]
}
```

By going through Code Snippet 16, it is easy to get the answers to the questions in this specific scenario, which are as follows:

- **What is an id?**: A unique ID for a book.
- **Is title mandatory?**: Yes, as it is included under the `array` of required attributes.

Concepts

- **Can the number of copies be zero?**: No. In JSON schema, a number must have a minimum. This minimum is inclusive by default and hence, exclusiveMinimum must be specified.

- **Are all tags of type String?**: Yes. However, it is not mandatory to specify them. Further, there should be at least one tag, as specified by `minItems` and each tag should be different from the other, as mentioned by `uniqueItems`.

9.4.2 JSON Comments

JSON does not have any provision for documentation or comments. Comments are not supported, as its developer Douglas Crockford found users utilizing them to store parsing directives. However, you can still use a workaround to give comments within a JSON document, although you need to delete all before a parser processes it. Comments are still supported by a few JSON parsers and must be provided within `/* … */`.

Code Snippet 17 shows how to put comments in a JSON document.

Code Snippet 17:

```
 {
 "id":2,
 "title":"The fallen Hero", /* This books is about Harvey Dent */
 "noOfCopies":14,
 "tags":[
 "BatMan",
 "Gowtam"
 ],
 "dimensions":{ /*… The book seems to be very big . */
 "length":7.0,
 "width":12.0,
 "height":9.5
 }
 }
}
```

9.5 Creating and Parsing JSON Messages with JavaScript

The `JSON.stringify()` method allows converting a JavaScript object into a JSON String. Considering the same blog scenario, let us use the same JSON object as JavaScript Object and convert it to JSON.

Code Snippet 18 shows how a JavaScript object is converted to a JSON string. This code should be saved as an HTML file.

Code Snippet 18:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
<script type="text/javascript">

     //Step 1:: Defining a JavaScript Object
     var jsonObject = {
         "blogID" : 12345,
         "user" : {
             "firstName" : "James",
             "lastname" : "Gunn"
         },
         "blogContent" : "This is my first JSON for objects and this looks
simple",
         "tags" : [ "first", "JSON", "looks", "simple" ],
         "comments" : [ {
             "body" : "Lorem ipsum dolor sit amet",
             "email" : "whwZHtDw@LbGOObNn.com",
             "author" : "Cassi Heal"
         },
{
             "body" : "consectetur adipisicing elit, sed do eiusmod ",
             "email" : "VzdvZbJK@ziNZbdLG.com",
             "author" : "Bao Ziglar"
         } ]
     };


     document.write("<hr>");

     //Step 2 :: Converting a JavaScript object to JSON.
     document.write("<font style=\"color: blue\">JSON Format</font>::"
             + JSON.stringify(jsonObject, null, "\t"));
     document.write("<hr>");
</script>
</head>
<body>
</body>
</html>
```

Concepts

Figure 9.4 shows the output.

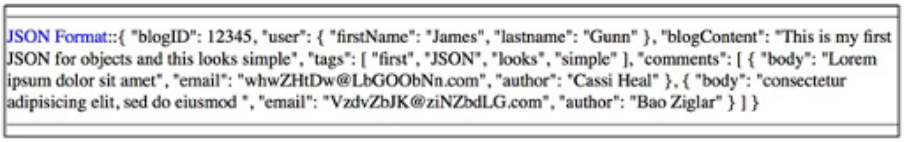

Figure 9.4: Output of JavaScript Conversion to JSON

The `JSON.parse()` method allows parsing a JSON object using JavaScript. It converts a JSON String to an object in JavaScript. You start by defining a string in JSON format, using the method for conversion, and then looping through the attributes for printing its values.

Code Snippet 19 shows how a JSON object is converted to a JavaScript object in an HTML file.

Code Snippet 19:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
<script type="text/javascript">
     //Step 1 :: Defining a JSON String
     var  jsonString  =  "{  \"blogID\":  12345,  \"user\":  {  \"firstName\":
\"James\", \"lastname\": \"Gunn\" },"
             + "\"blogContent\": \"This is my first JSON for objects and this
looks simple\", "
             + "\"tags\": [ \"first\", \"JSON\", \"looks\", \"simple\" ], "
             + "\"comments\": [ { \"body\": \"Lorem ipsum dolor sit amet\",
"
             +  "\"email\":  \"whwZHtDw@LbGOObNn.com\",  \"author\":  \"Cassi
Heal\" },"
             + " { \"body\": \"consectetur adipisicing elit, sed do eiusmod
\","
             +  "  \"email\":  \"VzdvZbJK@ziNZbdLG.com\",  \"author\":  \"Bao
Ziglar\" } ] }";
             //Step 2 :: Parsing the JSON String to JavaScript Object
             var jsonObject = JSON.parse(jsonString);

             document.write("<hr>");
             //Step 3 :: Looping each of the properties in the JavaScript
object and printing values.
             for ( var prop in jsonObject) {
                 document.write("</br></br><font style=\"color:
```

```
blue\">" + prop + "</font>::"
                          + jsonObject[prop]);

            }
            document.write("<hr>");
</script>
</head>
<body>
</body>
</html>
```

Figure 9.5 shows the output.

Figure 9.5: Output of JSON Conversion to JavaScript

9.6 JSON with Developer Tools on Browser

There are numerous plugins or extensions that help in validating and formatting a JSON document or a JSON Hypertext Transfer Protocol (HTTP) response. One such extension in Firefox is JSONView, which allows viewing a JSON document. Usually, when there is a JSON document with the right content type, you are prompted to download the file by the Firefox browser. However, with this extension, the JSON document is displayed in the browser. You can view the formatted and highlighted document, and can also collapse the objects. JSONView displays the raw text even though the JSON document has errors.

Code Snippet 20 shows the usage of JSONView.

Code Snippet 20:

```
{
 "blogID":87654,
 "user":{
 "firstName":"Mike",
 "lastname":"Capuano"
 },
  "blogContent":"Mike is an American politician who serves as the U.S.
Representative",
"tags":[
```

Concepts

```
 "Mike",
 "politician",
 "American"
 ],
 "comments":[
 {
 "body":"content is good",
 "email":"whwZHtDw@LbGOObNn.com",
 "author":"Hames Heal"
 },
 {
 "body":"consectetur adipisicing elit, sed do eiusmod ",
 "email":"VzdvZbJK@ziNZbdLG.com",
 "author":"Bao Ziglar"
 }
 ]
}
```

Figure 9.6 shows the output.

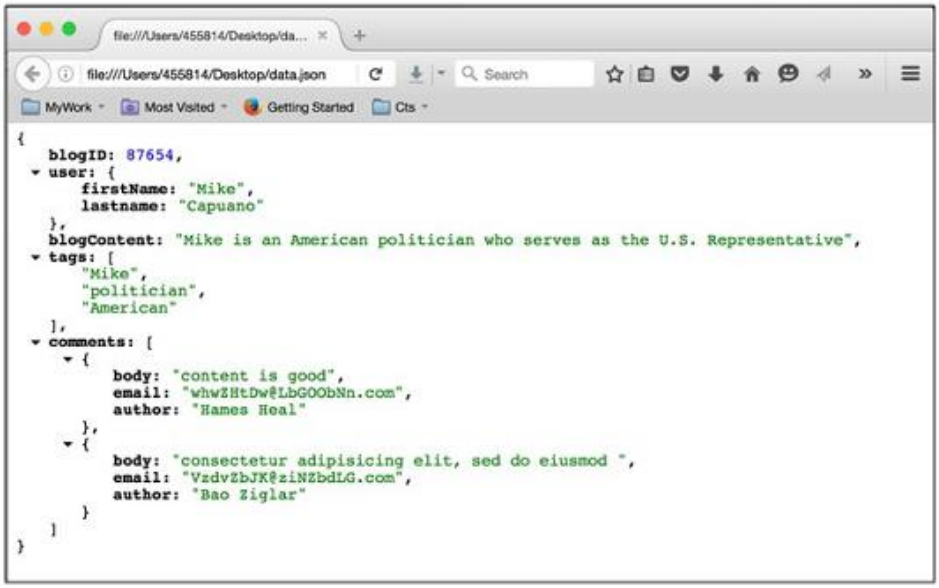

Figure 9.6: JSONView Output

Chrome also offers JSONView for validating a JSON document. Figure 9.7 shows a screenshot of the error message displayed in Chrome when an incorrect JSON document is loaded.

Figure 9.7: JSONView Error Message in Chrome

For modifying the JSON values during runtime, it is recommended converting the JSON document to a JavaScript object before using the built-in browser developer tools.

9.7 Online Tools and Editors

Many online tools and editors are available for validating JSON data. One of the most popular online validators is JSONLint. It is an open source project that helps in validating JSON data. When using JSON with any other programming language, it is wise to check the data in advance because improper formatting of JSON data can cause errors. Using JSONLint or any other online too is easy, as it only requires copying the JSON data to its online editor. You can search for more JSON validator tools online.

Code Snippet 21 shows a sample corrupted JSON code to be validated using JSONLint.

Code Snippet 21:

```
{
 "blogID":12345,
 "user":{
 "firstName":"James,
 "lastname":"Gunn"
 },
 "blogContent":"This is my first JSON for objects and this looks simple",
 "tags":[
 "first",
 "JSON",
 "looks",
 "simple"
 ]
}
```

Figure 9.8 shows the output of code, when validated in JSONLint online editor.

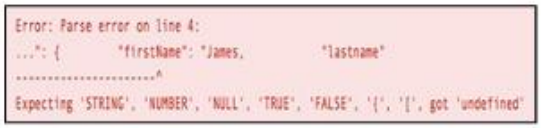

Figure 9.8: Error by JSONLint

The JSON code had an error, which the editor displays in figure 9.8. It shows the line number and the type of error that occurred. Through this output, it is easy to locate the error, which is missing closed quotation marks in the value of `firstName` on line number 4. After fixing the error, the editor displays the output, as shown in figure 9.9.

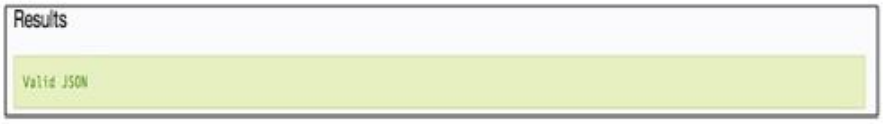

Figure 9.9: Valid Output by JSONLint

Some more JSON validators to consider are Curious Concept and JSON schema Validator.

9.7.1 Online JSON Viewers

While working with JSON, it is often required to use a JSON viewer to see how the document will look in the browser. There are many JSON viewers available for you to choose, depending on your requirement.

One of the most widely used online viewers is jsonviewer.stack.hu, which has several features. JSON text can be directly copied to this online viewer.

Figure 9.10 shows the jsonviewer.stack.hu viewer.

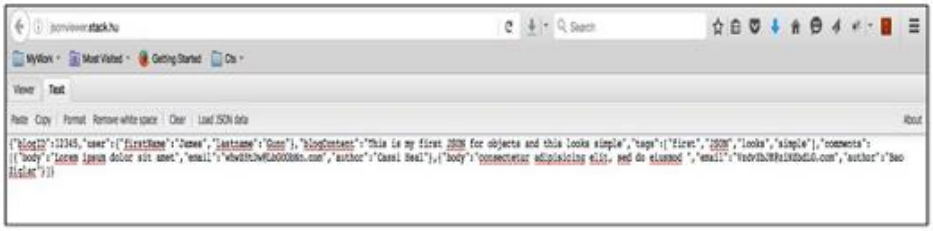

Figure 9.10: Jsonviewer.stack.hu Viewer

Another way of supplying data to the viewer is by clicking **Load JSON data**. The viewer takes data from the URL you provide and loads data from that source. Once the JSON code is visible in the viewer, clicking **Format** formats the code, as shown in figure 9.11.

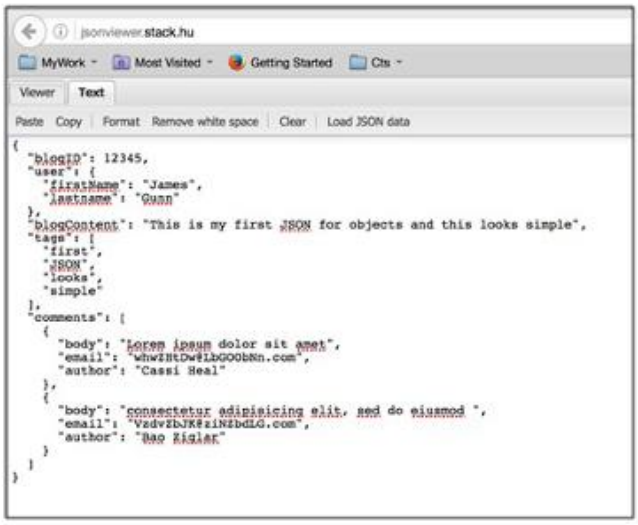

Figure 9.11: Formatted JSON Code in the jsonviewer.stack.hu Viewer

You can minimize the JSON code by clicking the **Text** tab and then clicking **Remove white space**. You can clear the code by clicking **Clear**. Similarly, if you click the **Viewer** tab, you see a tree-like structure of the JSON document on the left and corresponding names and values on the right, as shown in figure 9.12.

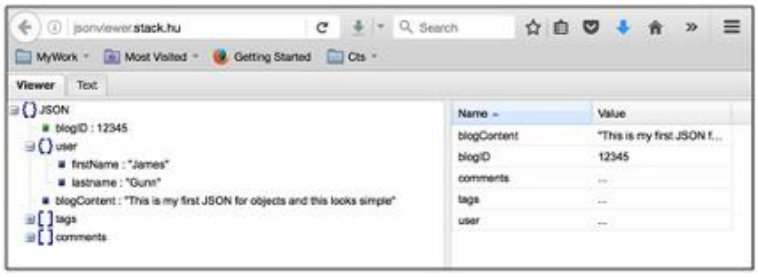

Figure 9.12: Viewer Tab

Apart from viewers, there are online tools that support JSON parsing.

Concepts

Summary

- JSON data types are classified into primitive and structure data types. These are further classified into string, number, Boolean, null, white space, object, and array.
- JSON supports two main data structures namely, collection of name/value pairs and ordered list of values.
- Sets and Lists correspond to a set of objects, whereas Maps relates a value to an object.
- JSON schema specifies the rules that define the structure of JSON document, which help in validating and verifying the document.
- JSON schema serves as a meta-document that helps in defining required data types and limitations on values.
- JSON documents do not support comments, however, they are still supported by a few JSON parsers and must be provided within /* … */.
- JSONLint is an online validator for JSON documents.
- Browsers such as Chrome and Firefox offer extensions for viewing JSON documents in the browser window.

Check Your Progress

1. Which of the following data types is the only accepted type for keys in a JSON object?

   a. Number

   b. Object

   c. String

   d. Array

2. Which of the following code snippets is not a valid JSON document?

   a. `{}`

   b. `{"a" : "","a":"b"}`

   c. `{"a" : [],"m":1}`

   d. `{1 :"m","m":1}`

3. Which of the following statements defines the usage of an array?

   a. Stores sequential elements of similar type.

   b. Stores unordered elements of dissimilar type.

   c. Stores key names that are random strings.

   d. Stores a single Unicode character represented as a single character string.

Check Your Progress

4. Match the data type with the respective syntax.

	Data Type		Syntax
a.	Object	1.	[value, ...]
b.	String	2.	{ string : value, ...}
c.	Boolean	3.	{ "string" : number_value, ...}
d.	Array	4.	{ string : true/false, ...}
e.	Number	5.	{"string" : "string value", ...}

5. Identify output for the given code.

```
var p = null;
if(p == null)
{
document.write("<h1>value is null</h1>");
}
else
{
document.write("<h1>value is not null</h1>");
}
```

a. null

b. value is null

c. value is not null

d. error

Session
10 JSON in Real World

Concepts

Objectives

In this session, you will learn to:

- *Describe the support of different browser and programming languages for JSON*
- *Describe JSON content types*
- *Explain how to use JSON for Web and Data Storage*
- *Compare JSON with relational databases*
- *Identify security and data portability issues with respect to JSON*

10.1 JSON Support with Programming Languages

Each major programming language can incorporate JSON, which can be through either libraries or native support. However, no language can surpass JavaScript for parsing this data interchange format, which translates it directly into an object literal.

JSON is as popular as XML. Thus, each major language has one or more commonly used powerful libraries for formatting and parsing data in JSON format. Typically, only following two core functionalities are required:

- **Parse**: Converts JSON data into the supported data structure of corresponding language, such as array, hash, or dictionary.
- **Format**: Converts array, hash, or dictionary to JSON text.

Developers can easily find these functionalities for almost any modern language. For instance, Ruby incorporates JSON gem, Objective-C supports JSONKit, and Microsoft .NET Framework has Json.NET. Most of these libraries are fast and efficient, considering their extensive use and frequent optimizations over time.

While comparing various languages, it might be pointed out by some developers that using JSON in a C# or Java language is not practical. This is because the idioms support statically typed classes and not objects of Dictionary or HashMap type. As a result, for generating JSON data, developers are required to use a library along with a custom code for converting these data structures to static type instances. Hence, there are some libraries created for this purpose. For instance, the Gson library from Google is designed for transforming JSON data to Java objects directly.

Concepts

10.1.1 Gson

Gson is an open-source Java library for transforming an object in Java to JSON data and vice-versa. For this purpose, it offers easy means such as constructor (factory method) and `toString()`. This library also functions well with arbitrary Java objects, involving the pre-existing ones whose source code is not available with you. Following are the goals of Gson:

- Converting already existing non-modifiable objects to and from JSON
- Permitting custom representations for objects
- Outputting legible and compact JSON data

Gson is capable of deserializing strings of more than 25 MB, deserializing 87,000 objects, and serializing of 1.4 million objects without any issues. Its 1.4 version has increased the deserialization bar from 80 KB to more than 11 MB, applicable for array and collections in bytes.

It is convenient to learn and use Gson. A developer needs to know two methods namely, `toJson()` and `fromJson()`. The `toJson()` method is used for transforming a Java object to JSON data whereas `fromJson()` for converting JSON data to an object in Java.

10.2 JSON with PHP

PHP also allows encoding and decoding JSON data structures. From version 5.2.0 onwards of PHP, by default, the extension for JSON is packaged into PHP.

Table 10.1 lists the functions used in PHP for encoding and decoding JSON structures.

Function	Description
`json_encode`	Serializes the stated array or object and returns it in the JSON format if successful or FALSE in case of error or failure.
`json_decode`	De-serializes JSON data and returns the suitable PHP type.
`json_last_error`	Returns the error that happened last.

Table 10.1: Encoding and Decoding Functions for JSON in PHP

Syntax for Encoding

```
string json_encode ($value [, $options = 0 ] )
```

where,

`value` indicates Unicode Transformation Format (UTF)-8 encoded JSON data to be serialized.

`options` indicates an encode integer bitmask having a few JSON constants.

In a nutshell, JSON is supported by all popular programming languages, such as, object-oriented (Java), client-side scripting (JavaScript), and server-side scripting (PHP) languages.

10.2 MIME Type of JSON

Also called media type or content type, MIME type refers to a two-part identifier composed of a type separated from its following subtype by a slash. It helps in identifying the type of formatted content or the format of file being sent over the Web. Accordingly, the browser then easily processes the data. For JSON text, formal MIME type is 'application/json', indicating that the 'json' subtype belongs to the 'application' type. While a majority of modern applications have accepted this MIME type, several others still extend legacy support to other types.

Several libraries, browsers, Web applications, and servers support the MIME type 'text/json' or 'text/Javascript' (unofficial). Prominent examples include Facebook API, Yahoo!, and Google Search API.

10.3 Applications of JSON

There are several useful applications of JSON, such as in APIs, NoSQL databases, Web development, and application packages.

10.3.1 APIs

JSON is commonly used to interchange data to and from APIs. Web applications, especially popular social networking sites, have an API for collecting huge amounts of data using which developers design derivative apps. Social applications, such as LinkedIn, Facebook, Flickr, and Twitter implement an API that uses JSON for sending data to developers. A few of these APIs support XML as well as JSON, while rest use only JSON.

10.3.2 NoSQL

JSON format is also used in NoSQL databases, such as CouchDb and MongoDb, for storing data. It is easy to work JSON for storing data into these databases. This is because it is possible to convert a JSON structure to an object data structure of JavaScript in the browser environment. Additionally, it is possible to integrate the structure with server-side JavaScript too.

Concepts

10.3.3 Asynchronous JavaScript and JSON (AJAJ)

AJAJ is similar to AJAX but uses JSON rather than XML. This dynamic Web development methodology allows a loaded Web page to ask for new data. Usually, the new data is sent from the server as a response to an action of a user on that page. For instance, when a user starts entering characters in a search box, it is sent to the server through the client-side code. Then, the server responds instantly by giving a list of matching items from the database, which the browser displays in a drop-down list.

10.3.4 JSON-RPC

JSON-RPC is a lightweight and simple protocol for Remote Procedure Call (RPC). It is developed on JSON for replacing Simple Object Access Protocol (SOAP) or XML-RPC. The protocol defines only a few commands along with some data types for allowing a system to send multiple calls and notifications to the server. While notifications do not await a response, calls are responded without any order.

10.3.5 Package Management

Due to the increased complexity in Web development, developers have started using tools for building packages of their applications. These packages make it easier to deploy the corresponding applications. They also make it easier to modify and integrate any changes, later. In this way, not only development but also maintenance becomes easier through packages.

Currently, several package management tools are provided, such as Node Package Manager (NPM), Yomen, and Bower. Most of these tools implement a package.json file that contains metadata such as version, name, description, license details, dependencies, and file structure.

10.4 JSON Hypertext Transfer Protocol (HTTP) and Files

One of the most widespread uses of JSON is to decipher data fetched from a Web server and display it on a Web page. For reading the data, the developers need to use the `XMLHttpRequest` object. This object exchanges data in the background with a server, which eliminates the need to reload the full page for updating only its few sections. All latest browsers, such as Chrome, Opera, and Internet Explorer 7+ support this object, as they have it built-in.

Syntax

```
variable = new XMLHttpRequest();
```

Code Snippet 1 shows how to fetch JSON data from a Web server and display it in the browser. It uses an already available Web service that helps in getting some mock data according to the specified User ID. This code can be saved as an HTML file and tested in a browser.

Code Snippet 1:

```
<!DOCTYPE html>
<html>
<body>
     <div id="Title">Content From JSON WebService</div>
     <div id="id01"></div>
     <script>
         var xmlhttp = new XMLHttpRequest();
         //Step 1
         var url = "http://jsonplaceholder.typicode.com/posts/1";
         //Step 2
         xmlhttp.onreadystatechange = function()
        {
              if(xmlhttp.readyState == 4 && xmlhttp.status == 200)
               {
                   //Step 3
                   var myArr = JSON.parse(xmlhttp.responseText);
                   var design = '</br>';
                   //Step 4
                   for ( var key in myArr)
                    {
                        if (myArr.hasOwnProperty(key))
                      {
                      design = design + ' ' + key + " > " + myArr[key];
                   }
                   design = design + '</br></br>'
                   }
              //Step 5
              document.getElementById("id01").innerHTML = design;
       }
         };

         //Step 6
         xmlhttp.open("GET", url, true);
         xmlhttp.send();
     </script>
</body>
</html>
```

Concepts

In Code Snippet 1, a call to a random Web service, `http://jsonplaceholder.typicode.com/`, is made for fetching data from the server. JSONPlaceholder is a free online service for fetching mock data. The code refers to `http://jsonplaceholder.typicode.com/posts/{userID}` for fetching mock data for a specific userID. This data includes UserID, ID, title, and body.

In Step 1 of Code Snippet 1, the URL is initialized for making a call to the Web service. This URL can be any JSON Rest Web service, such as Facebook, Google, or Twitter. Then, the callback method is defined in Step 2, once the response is received. In Step 3, the response is parsed. Then, a loop is created to scan through the JSON object data, in Step 4. In Step 5, the HTML format is designed for the JSON data so that the data can be displayed well. Finally, the actual call is made to the Web service.

Figure 10.1 shows the output of Code Snippet 1.

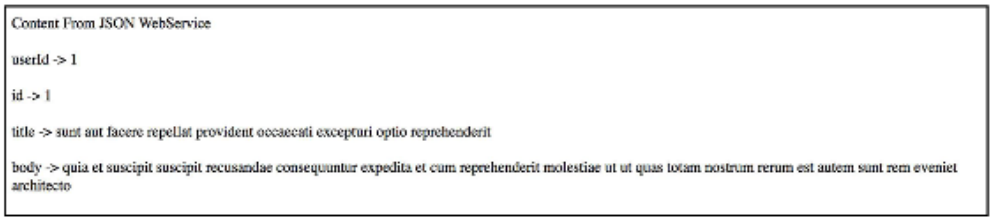

Figure 10.1: Output of Reading From a JSON File on Server Through XMlHttpRequest Object

Code Snippet 2 shows how to read a local JSON file and display its data in the browser. This JSON file has a list of animated movies along with their brief descriptions.

Code Snippet 2:

```
[
 {
 "title":"Angry Birds",
 "description":"Film based on the video game"
 },
 {
 "title":"Captain England",
 "description":"Loosely based on the life of cricketer"
 },
 {
 "title":"Kung Fu Panda",
 "description":"Panda Chi"
 }
]
```

Concepts

Code to Read the JSON File

```
<!DOCTYPE html>
<html>
<body>

     <div id="Title">Content From JSON WebService</div>
     <div id="id01"></div>

     <script>
         var xmlhttp = new XMLHttpRequest();

//Step 1
         var url = "data.json";
         //Step 2
         xmlhttp.onreadystatechange = function()
{
             if (xmlhttp.readyState == 4 && xmlhttp.status ==
             200)
{
                 //Step 3
                 var myArr = JSON.parse(xmlhttp.responseText);
                 console.log(myArr);
                 var design = '<table style="width:100%" border
="1px"><tr><th>Title</th><th>Description</th></tr>';
var format = '<tr><th>Title</th><th>Description</th></tr>';
                 //Step 4
                 for ( var key in myArr)
{
                     var obj = myArr[key];
                     var out = format.replace("Title",
                     obj.title).replace("Description",
                     obj.description);
                     design = design + out;
                 }
                 design =design+ '</table>';
                 console.log(design);
                 //Step 5
document.getElementById("id01").innerHTML = design;
```

Concepts

```
            }
        };
        //Step 6
        xmlhttp.open("GET", url, true);
        xmlhttp.send();
    </script>
</body>
</html>
```

Figure 10.2 shows the output of Code Snippet 2.

Content From JSON WebService

Title	Description
Angry Birds	Film based on the video game
Captain England	Loosely based on the life of cricketer
Kung Fu Panda	Panda Chi

Figure 10.2: Output of Reading Data from Local JSON File

10.5 JSON for Data Storage

Initially introduced as the serialized object notation for Web applications, JSON is now being used beyond the Web. For example, it is used in storing data fetched from services and applications. This is perhaps because of its simplicity and 'just adequate structure'.

The long domination of RDBMS and Structured Query Language (SQL) has ended with the introduction of several NoSQL databases, which are especially serving developers who prefer JSON. With the increasing implementation of JSON, database vendors have also started providing document databases that are JSON-centric. As the current trend, more number of traditional RDBMS' are incorporating JSON features, offering the benefits of both not only to developers but also to database administrators. These benefits include agility and developer friendliness.

10.5.1 Developer-friendly

It is obvious that developers need to handle code as well as data. While code acts as verbs to describe or instruct how to do a task, data acts as nouns to define entities such as users along with their details. As an increasing number of developers are favoring the JSON as the primary data format, it is inevitable to have JSON-friendly databases. Therefore, many NoSQL database vendors are choosing JSON over conventional relational schemes in Oracle, MS SQL Server, VoltDB, MySQL, and PostgreSQL. This is more suitable for developers favoring JSON format for their applications.

10.5.2 Agility

JSON records are not only legibly structured but are also easily extensible, making the format agile. This makes a significant appeal to developers who do not want the painful hassle of database schema migrations in dynamic environments. In terms of volume, it can be tough to change both data and schema. Moreover, rewriting a big record stored on a disk, while maintaining the linked applications is perhaps a time consuming job demanding several days of background processing. In contrast, the absence of predefined schema in JSON allows for easier upgrades. Developers can save and update documents, without any limitations.

10.5.3 Document vs. Relational Databases

It is a fact that removing limitations and permitting arbitrary data in a database can induce its own drawbacks. Renouncing a relational database model has its own limitations for the developers, businesses, and enterprise architects. In fact, relational databases were introduced to replace systems that had structure similar to the JSON files. In short, documented-oriented databases possess some inherent flaws, which are as follows:

- Need to navigate the document while querying, leading to tighter coupling between the applications and database
- Introduction of new query language for each document store, which unnecessarily weakens standardization, unlike well-defined SQL query core ensuring portability across database systems
- Removal of vital abstraction between the logical data structure and its physical storage, which restricts optimization; unlike a relational system that makes stored data definition more queryable through such an abstraction
- No support for constraints enforcing consistency and normalization to store redundant information only once, which puts the burden of ensuring the same on the developer

Nevertheless, the database vendors are accepting the utility and growing popularity of JSON along with its integration to their conventional relational systems. In these systems, incorporating the mechanism to store and query JSON records provides the advantage of a powerful choice to the developers. They can easily decide the portions of data to be abstracted, areas where flexibility is more important than consistency checks, and locations where strict schema and constraints are essential. This is because integration of JSON in a conventional relational database gives the best of both worlds.

Moreover, such integration allows JSON query to be a native SQL extension, which enables making robust queries across random collections of JSON records merged with data in conventional table cells. In practice, it shall be significantly easier for RDBMS vendors to expand their systems to incorporate JSON than for the vendors of new document stores. The latter vendors, by incorporating JSON, are willing to bring back the features lost while they were switching from powerful relational structures.

Concepts

Yet JSON is much in demand for such powerful relational databases. This is because it is a small format that has not only affected Web-based programming models but also data management tiers. JSON has introduced new vendors serving to the choices of modern developers. Eventually, JSON may bring in new advancements in relational databases.

10.6 Comparison of JSON with Relational Databases

Although even JSON hosts or represents data, it differs significantly from the conventional relational database model implemented in RDBMS applications such as SQL Server and MySQL. Knowing these differences can help you to prefer JSON over RDBMS or vice-versa, as per the data structure and type. Following are the basic differences:

- **Structure**: A table stores data in a relational database, while JSON uses arrays and objects for the same, which can be recursively nested.
- **Metadata**: A schema stores data about the type as well as structure of the stored data in a relational database. Further, schemas are created while creating a database and tables, not at the time of storing data. Even JSON can use a schema but it is not pre-created like in a relational database. Usually, a JSON structure is self-describing but if it uses a schema, it ensures more flexibility than a schema in a relational database.
- **Data Retrieval**: A relational database utilizes SQL, which is a robust and expressive language based on relational algebra, for obtaining data from tables. On the other hand, JSON does not use any such popular language. In fact, it uses JSON Query Language (JAQL) and JSONiq, which are still evolving query languages.
- **Sorting**: In a relational database, SQL easily helps in retrieving data and displaying it in ascending or descending order. In JSON, a developer can sort arrays.
- **Learning Curve**: JSON has a much smooth learning curve. This is because the data types and structure supported here are used in several programming languages. Therefore, a developer with basic programming background grasps JSON concepts and coding quite fast. On the other hand, RDBMS is a distinct field to learn and explore, which takes time to master. However, once mastered, it has its own opportunities and benefits to offer.
- **Application**: The market has several commercial as well as open source relational databases to offer. There are also NoSQL databases but they use JSON to store data. JSON is usually implemented in several programming languages.

10.7 JSON Security and Data Portability Issues

JSON is a data serialization format but it is also a flexible subset of JavaScript, which gives rise to a few security issues. These issues focus on a JavaScript interpreter executing JSON data as embedded JavaScript. In simple words, JSON's compatibility is exploited to the `eval()` function. This makes an

application vulnerable to malicious scripts, which is a critical concern while handling data obtained from the Web. Similarly, there are other issues as well with regards to implementation.

10.7.1 Issues Regarding the Eval() Function for Executing JSON Text

Most JSON text is syntactically JavaScript, which makes it easy for the `eval()` function to parse JSON data. Instead of a JSON-based parser, the interpreter of JavaScript executes the JSON data for generating corresponding JavaScript objects. This is actually risky, unless the data is first validated before parsing. Using the `eval()` approach makes such a non-validated data vulnerable to security issues, when the full JavaScript environment and data are out of control of a trustworthy source. For instance, if data is not reliable, it is exposed to malicious code injection attacks. Such breaches can then lead to authentication forgery, data and identity theft, and misuse of resources. In short, the following code is a big security concern:

```
var get_JSON_data = eval('(' + JSONresponse + ')');
```

As an attempted solution, a regular expression can partially validate the data before parsing it. RFC 4627 recommends the following code for validation prior to evaluation of JSON data:

```
var   get_JSON_object   =   !(/[^,:{}\[\]0-9.\-+Eaeflnr-u   \n\r\t]/.test(text.
replace(/"(\\.|[^"\\])*"/g, ''))) && eval('(' + text + ')');
```

where,

`text` is the JSON input.

However, this code was found to be insufficient for validation. Therefore, `JSON.parse()` was introduced as a more reliable alternative. The function processes text only in JSON format.

Web browsers are intending to include or have already included support for this native function for encoding and decoding JSON text. This removes the `eval()` breach as well as boosts performance as compared to the formerly used libraries in JavaScript. Following browsers are supporting both the `parse()` and `stringify()` methods, as of June 2009:

- Microsoft Internet Explorer 8+
- Opera 10.5+
- Google Chrome
- Opera
- Mozilla Firefox 3.5+

- Apple Safari

Following are the famous JavaScript libraries, which have pledged to implement native JSON.

- jQuery
- Dojo Toolkit
- Prototype
- YUI Library
- MooTools

Another issue while using the `eval()` function is the need of extra escaping in a few cases. This is because a few legal Unicode characters in JSON strings are considered illegal in JavaScript. By executing JSON data as JavaScript code, the function exposes the application to several security breaches such as Cross-site Scripting (XSS) and Cross-site Request Forgery (CSRF).

10.7.2 Issues of XSS and CSRF

XSS refers to inserting random HTML (along with JavaScript) into Web pages for usually bypassing access control mechanisms. To resolve it, developers are required to escape text aptly such that it cannot have random HTML tags. Many escaping schemes are available, which a developer chooses depending on the location of placing the non-reliable string (escaping) in an HTML document. These schemes escape the mentioned string.

Another issue is CSRF or XSRF, which is applicable even to JSON, primarily because of the manner in which the browsers work. In CSRF, unauthorized commands are sent from a user that seems to be trustworthy to the site. Unlike XSS in which the trust of a user for a specific site is exploited, the site's trust developed for a user's browser is exploited in CSRF.

Luckily, a strange section of JavaScript specification can solve this issue easily but that is only applicable if a developer uses the `jsonify()` function for generating JSON structures. This clearly indicates that the risk still prevails if other methods of generating JSON structures are used. So, where actually is the risk? Well, it is in top-level JSON arrays.

Consider a scenario wherein a user sends user names and e-mail addresses in the JSON format. This is similar to exporting these details of all intended entities, say colleagues, for a portion of the user interface designed in JavaScript.

The JSON format is:

```
[

 {"username": "admin","email": "admin@localhost"}

]
```

The format is exported to each GET request to a specific URL, which for example, is http://mysocialsite.com/api/get_colleagues.json. Now, imagine a smart hacker injecting the following code to his/her site:

```
<script type=text/javascript>
var captured = [];
var oldArray = Array;
function Array()
{
 var obj = this, id = 0, capture = function(value)
 {
 obj.__defineSetter__(id++, capture);
 if (value)
 captured.push(value);
 };
 capture();
}
</script>
<script type=text/javascript
 src= http://mysocialsite.com/api/get_colleagues.json></script>
<script type=text/javascript>
Array = oldArray; //all data now in the captured array.
</script>
```

If the developer is even a little familiar with JavaScript internals, he/she is aware of the possibility to register callbacks and patch constructors. An attacker can grab this opportunity to obtain the exported data. The browser also overlooks the application/json MIME type, as the script tag specifies it as text/javascript. This means the script will be evaluated as JavaScript. Due to the array being hooked in the constructor, data from the responded JSON structure is the array, once the page is fully loaded.

10.7.3 Issues Regarding Implementation

In the past, several implementations of JSON parser were exposed to Denial-of-Service (DoS) attack and vulnerability of mass assignment.

- **DoS Attack**: Refers to an attempt to render a resource or a machine on a network unavailable to its targeted users. For example, these users get messages, such as temporarily suspended or service interrupted, while connecting to the Internet. At times, Distributed Denial-of-Service (DDoS) also strikes wherein the source of attack source is not one but multiple, usually hundreds of IP addresses. This is similar to a collection of people crowding at the office building's gate. This crowd is so much that the legitimate ones, such as system administrators, do not get a chance to reach the office floor on time, which delays normal chores. Apart from this vulnerability, JSON is also prone to security-bypass vulnerability, wherein attackers bypass a few security rules, which increases the chances of other attacks. JSON versions before 1.5.5, 1.6.8, and 1.7.7 are vulnerable to such attacks.

- **Mass Assignment Vulnerability**: Refers to maltreating a functioning pattern of record in a Web application for changing confidential data items, such as file permissions and passwords, in an unauthorized manner. Frameworks of several Web applications support such record and object-relational mapping capabilities through which serialized external data is converted to internal objects and consequently to record fields in the database. In case the interface of the framework is too liberal for that exchange, an attacker can easily overwrite fields such as admin permissions.

10.7.4 Issues Regarding Data Portability

While JavaScript prohibits the use of Unicode line terminators without being escaped in strings, JSON does allow them. These terminators are line separator (U+2028) and paragraph separator (u+2029). JSON prohibits only control characters, which is why it accepts these terminators. To ensure optimal portability, it is essential for these terminators to be backslash-escaped.

JSON allows the null character (U+0000) in a string if escaped as "\u0000". However, the null character creates issues with a few JSON implementations, particularly with strings of C language-type.

Another issue is with the UTF encoding. A developer can encode a JSON file in UTF-8, UTF-16, or

UTF-32. Of these, UTF-8 is the default encoding standard. All the encodings encompass the whole Unicode character set, including characters external to the Basic Multilingual Plane. These characters are from U+10000 to U+10FFFF. When escaped, it is essential to write these characters using surrogate

pairs of the UTF-16 standard. For example, to include the Emoji character U+1F606,  , in JSON, the code will be as follows:

```
{"face": "😆"}

 // or

{"face": "0xD83D 0xDE06"}
```

However, a few JSON parsers do not have the details of these pairs. Another issue is with the representation of JSON numbers in relation to programming languages. There is no difference between a floating-point value and an integer; a few implementations do not consider 32, 32.0, and 3.2E+1 as identical, while others do consider them as same.

Additionally, no specifications exist for other implementation issues, such as rounding, overflow or too big exponent, underflow or too small exponent, and precision loss. There is also no requirement stated for treating signed zeros, such as 0.0 and -0.0. While most implementations including JavaScript follow the IEEE 754 floating-point standard and conserve signed zeros, all JSON implementations necessarily do not do so.

Portability is also affected, as JavaScript has many native data types, which JSON does not support. These data types are Error, Date, Undefined, Function, and Regular Expression. To represent these data types, some data format must be accepted by applications on both sides (client and server). A few de facto solutions do exist, such as accepting Date as String type but none of them is unanimously accepted. However, some languages might support distinct native types, which need to be serialized well for such conversions.

10.7.5 Handling JSON Securely

Although the `eval()` function imposes security hacks, proper precautions can ensure secure JSON data. For this purpose, there are two approaches namely, to avoid using the function or to ensure that data is safe. A developer can choose any one or both the approaches for an application.

- **Avoiding eval()**: This is possible by using a JavaScript library as an alternative. A few libraries are available at www.json.org, such as JSON sans `eval()`, wherein the French word 'sans' means 'without' and JSON2 parser. Keeping in mind the extra security advantage of directly not using `eval()`, there is no reason for not switching to one of these parsing libraries.

- **Assuring the JSON Data Integrity**: This is the second approach, which comes from XMLHttpRequests. It is possible to make these requests only on the same domain, which indicates some integrated integrity capability. Despite this, the client-side code is still dependent on the security of the code on the server side. Further, if a developer is not utilizing XMLHttpRequests, it is still possible to ensure security by replicating the same-domain rule. This

is done by making both the server and client pass a token to each other. Validating the token shall verify that the communication is valid, and not for malicious attacks.

For extra security, a developer can even use HyperText Transfer Protocol Secure (HTTPS) connections, cookie-based authentication, or Direct Web Remoting (DWR). The DWR library is written in Java for interacting with the browser's JavaScript while running on the server.

Summary

- Modern programming languages can incorporate JSON structures through either libraries or native support for parsing.
- Gson is an open-source Java library for transforming an object in Java into JSON data and vice-versa.
- Formal MIME type for JSON text is 'application/json'.
- JSON is used in several applications, such as in APIs, NoSQL document-based databases, Web development, and application packages.
- JSON is incorporated in traditional relational databases due to its simplicity, agility, and developer friendliness.
- Unlike relational databases, JSON does not have tables, pre-created metadata, and support for SQL although it is easier to learn.
- Using eval() for parsing JSON data can lead to security attacks such as XSS and CSRF.
- JSON is prone to DoS attack and vulnerability of mass assignment.
- JSON has a few portability issues such as null character not fully compatible with C strings, no differentiation in floating point and integer types, and no support for a few JavaScript data types.
- Developers can secure JSON structures by replacing eval() with a JavaScript library and/or using XMLHttpRequests.

Concepts

Check Your Progress

1. An instance of Gson class is created using the _____ class.

   a. `Gson`

   b. `GsonBuilder`

   c. `GsonJson`

   d. `GsonCreator`

2. Which of the following methods is used to transform a PHP object into a JSON object?

   a. `json_encode`

   b. `json_decode`

   c. `json_convert`

   d. `json_serialize`

3. Which of the following MIME types is the official one for JSON?

   a. `text/javascript`

   b. `text/json`

   c. `application/json`

   d. `application/javascript`

Check Your Progress

4. Which of the following objects eliminates the need to reload the entire Web page while updating a few of its sections?

   a. `HTTPRequest`

   b. `HttpJSONRequest`

   c. `JSONHttpRequest`

   d. `XMLHttpRequest`

5. Which of the following functions aims to eliminate the safety risks induced by `eval()`?

   a. `JSON.format()`

   b. `JSON.parse()`

   c. `JSON.string()`

   d. `JSON.eval()`