* A web service is any piece of software that makes itself available over the internet and uses a standardized XML messaging system. XML is used to encode all communications to a web service. For example, a client invokes a web service by sending an XML message, then waits for a corresponding XML response. As all communication is in XML, web services are not tied to any one operating system or programming language--Java can talk with Perl; Windows applications can talk with Unix applications.
* Web services are self-contained, modular, distributed, dynamic applications that can be described, published, located, or invoked over the network to create products, processes, and supply chains. These applications can be local, distributed, or web-based. Web services are built on top of open standards such as TCP/IP, HTTP, Java, HTML, and XML.
* Web services are XML-based information exchange systems that use the Internet for direct application-to-application interaction. These systems can include programs, objects, messages, or documents.
* A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., between Java and Python, or Windows and Linux applications) is due to the use of open standards.

To summarize, a complete web service is, therefore, any service that:

* Is available over the Internet or private (intranet) networks
* Uses a standardized XML messaging system
* Is not tied to any one operating system or programming language
* Is self-describing via a common XML grammar
* Is discoverable via a simple find mechanism

Components of Web Services

The basic web services platform is XML + HTTP. All the standard web services work using the following components

* SOAP (Simple Object Access Protocol)
* UDDI (Universal Description, Discovery and Integration)
* WSDL (Web Services Description Language)

All these components have been discussed in the [Web Services Architecture](https://www.tutorialspoint.com/webservices/web_services_architecture.htm)chapter.

How Does a Web Service Work?

A web service enables communication among various applications by using open standards such as HTML, XML, WSDL, and SOAP. A web service takes the help of:

* XML to tag the data
* SOAP to transfer a message
* WSDL to describe the availability of service.

You can build a Java-based web service on Solaris that is accessible from your Visual Basic program that runs on Windows.

You can also use C# to build new web services on Windows that can be invoked from your web application that is based on JavaServer Pages (JSP) and runs on Linux.

Example

Consider a simple account-management and order processing system. The accounting personnel use a client application built with Visual Basic or JSP to create new accounts and enter new customer orders.

The processing logic for this system is written in Java and resides on a Solaris machine, which also interacts with a database to store information.

The steps to perform this operation are as follows:

* The client program bundles the account registration information into a SOAP message.
* This SOAP message is sent to the web service as the body of an HTTP POST request.
* The web service unpacks the SOAP request and converts it into a command that the application can understand.
* The application processes the information as required and responds with a new unique account number for that customer.
* Next, the web service packages the response into another SOAP message, which it sends back to the client program in response to its HTTP request.
* The client program unpacks the SOAP message to obtain the results of the account registration process.

**WSDL:**

* WSDL stands for Web Services Description Language
* WSDL is used to describe web services
* WSDL is written in XML
* WSDL is a W3C recommendation from 26. June 2007

WSDL Documents

An WSDL document describes a web service. It specifies the location of the service, and the methods of the service, using these major elements:

|  |  |
| --- | --- |
| **Element** | **Description** |
| <types> | Defines the (XML Schema) data types used by the web service |
| <message> | Defines the data elements for each operation |
| <portType> | Describes the operations that can be performed and the messages involved. |
| <binding> | Defines the protocol and data format for each port type |

The main structure of a WSDL document looks like this:

<definitions>  
  
<types>  
  data type definitions........  
</types>  
  
<message>  
  definition of the data being communicated....  
</message>  
  
<portType>  
  set of operations......  
</portType>  
  
<binding>  
  protocol and data format specification....  
</binding>  
  
</definitions>

WSDL Example

This is a simplified fraction of a WSDL document:

<message name="getTermRequest">  
  <part name="term" type="xs:string"/>  
</message>  
  
<message name="getTermResponse">  
  <part name="value" type="xs:string"/>  
</message>  
  
<portType name="glossaryTerms">  
  <operation name="getTerm">  
    <input message="getTermRequest"/>  
    <output message="getTermResponse"/>  
  </operation>  
</portType>

In this example the**<portType>** element defines "glossaryTerms" as the name of a **port**, and "getTerm" as the name of an **operation**.

The "getTerm" operation has an **input message** called "getTermRequest" and an **output message** called "getTermResponse".

The**<message>** elements define the **parts** of each message and the associated data types.

The <portType> Element

The <portType> element defines **a web service**, the **operations** that can be performed, and the **messages** that are involved.

The request-response type is the most common operation type, but WSDL defines four types:

|  |  |
| --- | --- |
| **Type** | **Definition** |
| One-way | The operation can receive a message but will not return a response |
| Request-response | The operation can receive a request and will return a response |
| Solicit-response | The operation can send a request and will wait for a response |
| Notification | The operation can send a message but will not wait for a response |

WSDL One-Way Operation

A one-way operation example:

<message name="newTermValues">  
  <part name="term" type="xs:string"/>  
  <part name="value" type="xs:string"/>  
</message>  
  
<portType name="glossaryTerms">  
  <operation name="setTerm">  
    <input name="newTerm" message="newTermValues"/>  
  </operation>  
</portType >

In the example above, the portType "glossaryTerms" defines a one-way operation called "setTerm".

The "setTerm" operation allows input of new glossary terms messages using a "newTermValues" message with the input parameters "term" and "value". However, no output is defined for the operation.

WSDL Request-Response Operation

A request-response operation example:

<message name="getTermRequest">  
  <part name="term" type="xs:string"/>  
</message>  
  
<message name="getTermResponse">  
  <part name="value" type="xs:string"/>  
</message>  
  
<portType name="glossaryTerms">  
  <operation name="getTerm">  
    <input message="getTermRequest"/>  
    <output message="getTermResponse"/>  
  </operation>  
</portType>

In the example above, the portType "glossaryTerms" defines a request-response operation called "getTerm".

The "getTerm" operation requires an input message called "getTermRequest" with a parameter called "term", and will return an output message called "getTermResponse" with a parameter called "value".

WSDL Binding to SOAP

WSDL bindings defines the message format and protocol details for a web service.

A request-response operation example:

<message name="getTermRequest">  
  <part name="term" type="xs:string"/>  
</message>  
  
<message name="getTermResponse">  
  <part name="value" type="xs:string"/>  
</message>  
  
<portType name="glossaryTerms">  
  <operation name="getTerm">  
    <input message="getTermRequest"/>  
    <output message="getTermResponse"/>  
  </operation>  
</portType>  
  
<binding type="glossaryTerms" name="b1">  
   <soap:binding style="document"  
   transport="http://schemas.xmlsoap.org/soap/http" />  
   <operation>  
     <soap:operation soapAction="http://example.com/getTerm"/>  
     <input><soap:body use="literal"/></input>  
     <output><soap:body use="literal"/></output>  
  </operation>  
</binding>

The**binding** element has two attributes - name and type.

The name attribute (you can use any name you want) defines the name of the binding, and the type attribute points to the port for the binding, in this case the "glossaryTerms" port.

The **soap:binding** element has two attributes - style and transport.

The style attribute can be "rpc" or "document". In this case we use document. The transport attribute defines the SOAP protocol to use. In this case we use HTTP.

The**operation** element defines each operation that the portType exposes.

For each operation the corresponding SOAP action has to be defined. You must also specify how the input and output are encoded. In this case we use "literal".

**JSON:**

## What Is JSON?

[JSON](http://en.wikipedia.org/wiki/JSON) is short for **JavaScript Object Notation**, and is a way to store information in an organized, easy-to-access manner. In a nutshell, it gives us a human-readable collection of data that we can access in a really logical manner.

### Storing JSON Data

As a simple example, information about me might be written in JSON as follows:

|  |  |
| --- | --- |
|  | var jason = { |
|  | "age" : "24", |
|  | "hometown" : "Missoula, MT", |
|  | "gender" : "male" |
|  | }; |

[**view raw**](https://gist.github.com/jlengstorf/2760279/raw/af448c95b9b3616d962388df6da11f261bfc9f86/gistfile1.js)[**gistfile1.js**](https://gist.github.com/jlengstorf/2760279#file-gistfile1-js) hosted with  by **[GitHub](https://github.com/)**

This creates an object that we access using the variable jason. By enclosing the variable's value in curly braces, we're indicating that the value is an object. Inside the object, we can declare any number of properties using a "name": "value" pairing, separated by commas. To access the information stored in jason, we can **simply refer to the name of the property we need.**For instance, to access information about me, we could use the following snippets:

|  |  |
| --- | --- |
|  | document.write('Jason is ' jason.age); // Output: Jason is 24 |
|  | document.write('Jason is a ' jason.gender); // Output: Jason is a male |

### Storing JSON Data in Arrays

A slightly more complicated example involves storing two people in one variable. To do this, we **enclose multiple objects in square brackets,**which signifies an array. For instance, if I needed to include information about myself and my brother in one variable, I might use the following:

|  |  |
| --- | --- |
|  | var family = [{ |
|  | "name" : "Jason", |
|  | "age" : "24", |
|  | "gender" : "male" |
|  | }, |
|  | { |
|  | "name" : "Kyle", |
|  | "age" : "21", |
|  | "gender" : "male" |
|  | }]; |

To access this information, we need to access the array index of the person we wish to access. For example, we would use the following snippet to access info stored in family:

|  |  |
| --- | --- |
|  | document.write(family[1].name); // Output: Kyle |
|  | document.write(family[0].age); // Output: 24 |

**NOTE:**This is beneficial if it will be necessary to loop through stored information, as it lends itself to a*for*loop with an automatically incrementing value.

### Nesting JSON Data

Another way to store multiple people in our variable would be to **nest objects.**To do this, we would create something similar to the following:

|  |  |
| --- | --- |
|  | var family = { |
|  | "jason" : { |
|  | "name" : "Jason Lengstorf", |
|  | "age" : "24", |
|  | "gender" : "male" |
|  | }, |
|  | "kyle" : { |
|  | "name" : "Kyle Lengstorf", |
|  | "age" : "21", |
|  | "gender" : "male" |
|  | } |
|  | } |

Accessing information in nested objects is a little easier to understand; to access information in the object, we would use the following snippet:

|  |  |
| --- | --- |
|  | document.write(family.jason.name); // Output: Jason Lengstorf |
|  | document.write(family.kyle.age); // Output: 21 |
|  | document.write(family.jason.gender); // Output: male |

Nested JSON and arrays can be combined as needed to store as much data as necessary.

## Why Does JSON Matter?

With the rise of [AJAX](http://en.wikipedia.org/wiki/Ajax_(programming))-powered sites, it's becoming more and more important for sites to be able to load data **quickly and asynchronously,** or in the background without delaying page rendering. Switching up the contents of a certain element within our layouts without requiring a page refresh adds a "wow" factor to our applications, not to mention the added convenience for our users. Because of the popularity and ease of social media, many sites rely on the content provided by sites such as Twitter, Flickr, and others. These sites provide RSS feeds, which are easy to import and use on the server-side, but if we try to load them with AJAX, we run into a wall: **we can only load an RSS feed if we're requesting it from the same domain it's hosted on.** An attempt to load my Flickr account's RSS feed via jQuery's [$.ajax()](http://docs.jquery.com/Ajax/jQuery.ajax) method results in the following JavaScript error:

|  |  |
| --- | --- |
|  | [Exception... "Access to restricted URI denied" code: "1012" |
|  | nsresult: "0x805303f4 (NS\_ERROR\_DOM\_BAD\_URI)" |
|  | location: "http://ajax.googleapis.com/ajax/libs/jquery/1.3.2/jquery.min.js Line: 19"] |

JSON allows us to overcome the cross-domain issue because **we can use a method called**[**JSONP**](http://remysharp.com/2007/10/08/what-is-jsonp/)**that uses a callback function to send the JSON data back to our domain.** It's this capability that makes JSON so incredibly useful, as it opens up a lot of doors that were previously difficult to work around.

## How Do We Load JSON into a Project?

One of the easiest ways to load JSON data into our web applications is to **use the**[**$.ajax()**](http://docs.jquery.com/Ajax/jQuery.ajax)**method available in the jQuery library.**The ease of retrieving data will vary based on the site providing the data, but a simple example might look like this:

|  |  |
| --- | --- |
|  | $.ajax( |
|  | type:'GET', |
|  | url:"http://example.com/users/feeds/", |
|  | data:"format=json&id=123", |
|  | success:function(feed) { |
|  | document.write(feed); |
|  | }, |
|  | dataType:'jsonp' |
|  | ); |

This example would request the latest feed items in JSON format and output them to the browser. Obviously, we wouldn't want to output raw JSON data to the browser, but this example shows the basics of loading JSON from an external source.

## A Practical Example: Loading Flickr Streams with JSON and jQuery

[See the Demo](https://www.copterlabs.com/demo/reading-json) | [Download the Source](https://www.copterlabs.com/demo/EnnuiDesign_reading-json.zip)

To show how JSON works in a real-world example, let's **load photos from Flickr using jQuery and the JSON version of Flickr's "Latest" photo feed.**

### Step 1: Create the AJAX Request

Flickr's photostream feeds are relatively easy to access. All users have a unique ID number, which we will send as part of the request to this URL.

### Step 2: Process the JSON Data

What we're going to do is **display the thumbnails of the latest 16 photos, which will link to the medium-sized display of the image.** The Flickr JSON is a little confusing, and it doesn't provide a direct link to the thumbnail version of our photos, so we'll have to use some trickery on our end to get to it, which we'll cover in just a moment. Each photo entry is stored in an array called items, which we access in our AJAX call using feed.items. To get to the data about each entry, we'll loop through the items until we've either hit the last available photo or 16 total photos; whichever comes first. Let's modify our function and set up the loop:

The element we're interested in is the "m" element stored within the "media" element. This can be accessed within our loop using feed.items[i].media.m. **We're going to run a regular expression on this value to get both the medium and thumbnail image paths, which we'll assemble into a linked thumbnail image.** Then, we'll push the newly assembled HTML into the array of thumbs we created. After we've finished the loop, we'll **combine all the images into one string of HTML**and replace the contents of our display element with the loaded thumbnails. Let's add this functionality to our script:

Note that I've also added a function called addLB() to the end of this function; this adds the [lightbox](http://leandrovieira.com/projects/jquery/lightbox/) effect to our thumbnails, which is purely for aesthetics.

### Step 3: Call Our Function

At this point, we're ready to call our function. To load my Flickr stream, we would need to call our function as follows:

|  |
| --- |
|  |