Protocol:

It defines standard for enabling the connection, communication, and data transfer between two places on a network. It is a system of rules for message exchange among computers.

General Requirements for a protocol:

Data formats for data exchange. The bit strings are divided in fields and each field carries information relevant to the protocol. Conceptually the bit string is divided into two parts called the *header area* and the *data area*. The actual message is stored in the data area, so the header area contains the fields with more relevance to the protocol. Bit strings longer than the maximum transmission unit (MTU) are divided in pieces of appropriate size.

Address formats for data exchange**.** Addresses are used to identify both the sender and the intended receiver(s). The addresses are stored in the header area of the bit strings, allowing the receivers to determine whether the bit strings are intended for themselves and should be processed or should be ignored. A connection between a sender and a receiver can be identified using an address pair *(sender address, receiver address)*. Usually some address values have special meanings. An all-*1*s address could be taken to mean an addressing of all stations on the network, so sending to this address would result in a broadcast on the local network. The rules describing the meanings of the address value are collectively called an *addressing scheme*.

Address mapping. Sometimes protocols need to map addresses of one scheme on addresses of another scheme. For instance to translate a logical IP address specified by the application to an Ethernet hardware address. This is referred to as *address mapping*.

Routing**.** When systems are not directly connected, intermediary systems along the *route* to the intended receiver(s) need to forward messages on behalf of the sender. On the Internet, the networks are connected using routers. This way of connecting networks is called internetworking.

Detection of transmission errors is necessary on networks which cannot guarantee error-free operation. In a common approach, CRCs of the data area are added to the end of packets, making it possible for the receiver to detect differences caused by errors. The receiver rejects the packets on CRC differences and arranges somehow for retransmission.

Acknowledgements of correct reception of packets is required for connection-oriented communication. Acknowledgements are sent from receivers back to their respective senders.

Loss of information *- timeouts and retries*. Packets may be lost on the network or suffer from long delays. To cope with this, under some protocols, a sender may expect an acknowledgement of correct reception from the receiver within a certain amount of time. On timeouts, the sender must assume the packet was not received and retransmit it. In case of a permanently broken link, the retransmission has no effect so the number of retransmissions is limited. Exceeding the retry limit is considered an error.

Direction of information flow needs to be addressed if transmissions can only occur in one direction at a time as on half-duplex links. This is known as Media

Access Control. Arrangements have to be made to accommodate the case when two parties want to gain control at the same time.

***Sequence control*.** We have seen that long bitstrings are divided in pieces, and then sent on the network individually. The pieces may get lost or delayed or take different routes to their destination on some types of networks. As a result pieces may arrive out of sequence. Retransmissions can result duplicate pieces. By marking the pieces with sequence information at the sender, the receiver can determine what was lost or duplicated, ask for necessary retransmissions and reassemble the original message.

Flow control is needed when the sender transmits faster than the receiver or intermediate network equipment can process the transmissions. Flow control can be implemented by messaging from receiver to sender.

Getting the data across a network is only part of the problem for a protocol. The data received has to be evaluated in the context of the progress of the conversation, so a protocol has to specify rules describing the context. These kind of rules are said to express the *syntax* of the communications. Other rules determine whether the data is meaningful for the context in which the exchange takes place. These kinds of rules are said to express the *semantics* of the communications.

Stateless vs Stateful Protocol

A stateless server is a server that treats each request as an independent transaction that is unrelated to any previous request. What makes the protocol stateless is that the server is not required to track state over multiple requests. This simplifies the contract between client and server, and in many cases minimizes the amount of data that needs to be transferred. If servers were required to maintain the state of clients' visits the structure of issuing and responding to requests would be more complex. The simplicity of the model is one of its greatest features. The stateless design simplifies the server design because there is no need to dynamically allocate storage to deal with conversations in progress. If a client dies in mid-transaction, no part of the system needs to be responsible for cleaning the present state of the server.A disadvantage of statelessness is that it may be necessary to include additional information in every request, and this extra information will need to be interpreted by the server.

HTTP as a Stateless Protocol

HTTP is a Stateless Protocol. It is concerned with requests and responses, which are simple, isolated transactions. Each http transaction is of “connect, request, response, disconnect” nature. Each request requires a separate connection.

This is perfect for simple web browsing, where each request typically results in a file (an HTML document or a GIF image etc.) being sent back to the client. The server does not need to know whether a series of requests come from the same, or from different clients, or whether those requests are related or distinct. But some web applications may have to track the user's progress from page to page, for example when a web server is required to customize the content of a web page for a user. Solutions for these cases include:

- the use of HTTP cookies.
- server side sessions,
- hidden variables (when the current page contains a form), and
- URL-rewriting using URI-encoded parameters, e.g., /index.php?session_id=some_unique_session_code.

The difference is in the nature of the connection they define between a client and a server. Telnet and FTP, for example are stateful protocols. The client connects to the server, conducts a series of operations via that connection, and then disconnects. Then server can associate all of the requests together and knows that thy all came from the same user.

HTTP Methods:

Http defines different methods. Some methods (for example, HEAD, GET, OPTIONS and TRACE) are defined as *safe*, which means they are intended only for information retrieval and should not change the state of the server. In other words, they should not have side effects. Hence, making arbitrary GET requests without regard to the context of the application's state should therefore be considered safe. By contrast, methods such as POST, PUT and DELETE are intended for actions that may cause side effects either on the server, or external side effects such as financial transactions or transmission of email.

Some of the important methods are:

GET

Requests a representation of the specified resource. Requests using GET should only retrieve data and should have no other effect.

Note that query strings (name/value pairs) is sent in the URL of a GET request:

e.g. */test/demo_form.php**?name1=value1&name2=value2***

Some other notes on GET requests:

- GET requests can be cached
- GET requests remain in the browser history
- GET requests can be bookmarked
- GET requests should never be used when dealing with sensitive data
- GET requests have length restrictions
- GET requests should be used only to retrieve data

POST

Requests that the server accept the entity enclosed in the request as a new subordinate of the web resource identified by the URI. The data POSTed might be, as examples, an annotation for existing resources; a message for a bulletin board, newsgroup, mailing list, or comment thread; a block of data that is the result of submitting a web form to a data-handling process; or an item to add to a database.

Note that query strings (name/value pairs) is sent in the HTTP message body of a POST request:

POST /test/demo_form.asp HTTP/1.1
Host: w3schools.com
name1=value1&name2=value2

Some other notes on POST requests:

- POST requests are never cached
- POST requests do not remain in the browser history
- POST requests cannot be bookmarked
- POST requests have no restrictions on data length

Recommendation When to use get and post:

GET is recommended when submitting "idempotent" forms - those that do not 'significantly alter the state of the world'. In other words, forms that involve database queries only. Another perspective is that several idempotent queries will have the same effect as a single query. If database updates or other actions such as triggering emails are involved, the usage of POST is recommended.

A "GET" request is often cacheable, whereas a "POST" request can hardly be. For query systems this may have a considerable efficiency impact, especially if the query strings are simple, since caches might serve the most frequent queries.

In certain cases, using *POST* is recommended even for idempotent queries:

- **If the form data would contain non-ASCII characters** (such as accented characters), then *METHOD="GET"* is inapplicable in principle, although it may work in practice (mainly for ISO Latin 1 characters).
- **If the form data set is large** - say, hundreds of characters - then *METHOD="GET"* may cause practical problems with implementations which cannot handle that long URLs.
- You might wish to avoid *METHOD="GET"* in order to make it less visible to users how the form works, especially in order to make "hidden" fields (INPUT TYPE="HIDDEN") more hidden by not appearing in the URL. But even if you use hidden fields with *METHOD="POST"*, they will still appear in the HTML source code.

Now, let us summarie the difference between GET and POST.

	GET	POST
BACK button/Reload	Harmless	Data will be re-submitted (the browser should alert the user that the data are about to be re-submitted)
Bookmarked	Can be bookmarked	Cannot be bookmarked
Cached	Can be cached	Not cached
Encoding type	application/x-www-form-urlencoded	application/x-www-form-urlencoded or multipart/form-data. Use multipart encoding for binary data
History	Parameters remain in browser history	Parameters are not saved in browser history
Restrictions on data length	Yes, when sending data, the GET method adds the data to the URL; and the length of a URL is limited (maximum URL length is 2048	No restrictions

	characters)	
Restrictions on data type	Only ASCII characters allowed	No restrictions. Binary data is also allowed
Security	GET is less secure compared to POST because data sent is part of the URL Never use GET when sending passwords or other sensitive information!	POST is a little safer than GET because the parameters are not stored in browser history or in web server logs
Visibility	Data is visible to everyone in the URL	Data is not displayed in the URL
Any more??	?	?

PUT
Uploads a representation of the specified URI
Requests that the enclosed entity be stored under the supplied URI. If the URI refers to an already existing resource, it is modified; if the URI does not point to an existing resource, then the server can create the resource with that URI.
According to the HTTP 1.1 specifications the GET, HEAD, DELETE, and PUT methods must be idempotent, and the POST method is not idempotent. That is to say that an operation is idempotent if it can be performed on a resource once or many times and always return the same state of that resource. Whereas a non idempotent operation can return a modified state of the resource from one request to another. Hence, in a non idempotent operation, there is no guarantee that one will receive the same state of a resource.
Based on the above idempotent definition, considerations for using the HTTP PUT method versus using the HTTP POST:
Use the HTTP PUT method when:

- The client includes all aspect of the resource including the unique identifier to uniquely identify the resource. Example: creating a new employee.
- The client provides all the information for a resource to be able to modify that resource. This implies that the server side does not update any aspect of the resource (such as an update date).

In both cases, these operations can be performed multiple times with the same results. That is the resource will not be changed by requesting the operation more than once. Hence, a true idempotent operation.
Use the HTTP POST method when:

- The server will provide some information concerning the newly created resource. For example, take a logging system. A new entry in the log will most likely have a numbering scheme which is determined on the server side. Upon creating a new log entry, the new sequence number will be determined by the server and not by the client.
- On a modification of a resource, the server will provide such information as

a resource state or an update date. Again in this case not all information was provided by the client and the resource will be changing from one modification request to the next. Hence a non-idempotent operation.

Example:
Use PUT when you can update a resource completely through a specific resource. For instance, if you know that an article resides at http://mysite.org/article/1234, you can PUT a new resource representation of this article directly through a PUT on this URL.
If you do not know the actual resource location, for instance, when you add a new article, but do not have any idea where to store it, you can POST it to an URL, and let the server decide the actual URL.

DELETE
Deletes the specified resource.

HEAD:
Same as GET but returns only HTTP headers and no document body.

Data Exchange Format

To improve development efficiency of Web Application, we need to implement the parallel development between Web Front End and Web Back End. To achieve this goal, first we should determine the data exchange format between Web Front End and Web Back End.
Data exchange is the process of taking data structured under a source schema and actually transforming it into data structured under a target schema, so that the target data is an accurate representation of the source data. Data exchange is similar to the related concept of data integration except that data is actually restructured (with possible loss of content) in data exchange.

Common data exchange formats between Web Front End and Web Back End are XML ,JSON , CSV,RDF, etc.

XML is a markup language for documents containing structured information. In order to appreciate XML, it is important to understand why it was created. XML was created so that richly structured documents could be used over the web. The only viable alternatives, HTML, are not practical for this purpose. It is a new markup language, developed by the W3C (World Wide Web Consortium), mainly to overcome limitations in HTML. The W3C is the organization in charge of the development and maintenance of most Web standards, most notably HTML.

The popularity of XML for data exchange on the World Wide Web has several reasons. First of all, it is closely related to the preexisting standards Standard Generalized Markup Language (SGML) and Hypertext Markup Language (HTML), and as such a parser written to support these two languages can be easily extended to support XML as well. For example, XHTML has been defined as a format that is formal XML, but understood correctly by most (if not all) HTML parsers. This led to quick adoption of XML support in web browsers and the toolchains used for generating web pages.

Some points to summarize XML:
XML stands for EXtensible Markup Language
XML is a markup language much like HTML
XML was designed to carry data, not to display data. So, XML documents do not carry information about how to display the data.
XML tags are not predefined. You must define your own tags
XML is designed to be self-descriptive
XML is a W3C Recommendation

Basic XML Example:

```
<country>
        <name>Nepal</name>
        <capital>Kathmandu</capital>
        <region>South Asia</region>
        <neighbor>India</neighbor>
</country>
```

XMLHttpRequest(): to send http/https requests to a web server and load the server response data back.

Create an XMLHttpRequest Object
:
All modern browsers (IE7+, Firefox, Chrome, Safari, and Opera) have a built-in XMLHttpRequest object.
Syntax for creating an XMLHttpRequest object:

```
xmlhttp=new XMLHttpRequest();
```

Old versions of Internet Explorer (IE5 and IE6) use an ActiveX Object:

```
xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
```

Open Method:
The HTTP/ HTTPS requests of the XMLHttpRequest object must be initialized through the open method. This method must be invoked prior to the actual sending of a request to validate and resolve the request method, URL, and URI user information to be used for the request. This method does not assure that the

URL exists or the user information is correct. This method can accept up to five parameters, but requires only two, to initialize a request.
open(Method, URL, Asynchronous, UserName, Password)
where,
Method: get, post, etc.
URL: the URL of the HTTP request.
Asynchronous: true (asynchronous) or false (synchronous)
The parameters, UserName, Password, may be provided for authentication and authorization if required by the server for this request.

Send Method:
To send an HTTP request, the send method of the XMLHttpRequest must be invoked. This method accepts a single parameter containing the content to be sent with the request.

send(Data)

Example: Send a Request To a Server

To send a request to a server, we use the open() and send() methods of the XMLHttpRequest object:

```
xmlhttp.open("GET","target.txt",true);

xmlhttp.send();
```

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language. While it was originally not designed for data exchange at all, it was discovered to be useful. In contrast to XML above, there exist no schema definition and no support for dialecting. The key benefits of this language are the low overhead (the amount of data needed for structuring) compared to XML and the similarly wide support: every web browser that has JavaScript support can also process JSON.

XML and JSON, both can be used as data exchange formats. XML is better for human reading than JSON but XML is not lighter as JSON. Though JSON is a great data exchange format for Web Applications over browsers, it is not always the option for replacement of XML . It is the question of When XML and When JSON. The popularity of JSON have increased a lot and recently Twitter and Foursquare removed support for XML in their APIs [More detail on http://norman.walsh.name/2010/11/17/deprecatingXML].

A simple example of storing JSON data:
```
var gces = {
        "age" : "15",
        "address" : "Pokhara",
        "sector" : "education"
};
```

This creates an object that we access using the variable ‘gces’. 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 ‘gces’, 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(‘ The age of gces is ' gces.age); // Output: gces is 15
document.write(‘gces is in ‘ jason.address); // Output: gces is in Pokhara
```

Example of JSON data in an array:

```
var college = [{
   "name" : "gces",
   "age" : "15",
   "address" : "Pokhara"
},
{
   "name" : "ncit",
   "age" : "10",
   "address" : "kathmandu"
}];
```

Access nested JSON data:

```
document.write(college[1].name); // Output: ncit
```

Nesting JSON data:

```
var college = {
   "gces" : {
      "name" : "gandaki college",
        "age" : "15",
        "address" : "Pokhara"
       “people” : [
                     { "id": "1001", "designation": "principal" },
                        { "id": "1002", "designation": "Lecturer" }
              ]
   },
   "ncit" : {
      "name" : "national collage of IT",
        "age" : "10",
       "address" : "kathmandu"
   }
```

}

Accessing nested JSON data:
document.write(college.gces.name); // Output: gandaki college

Php script for encoding and decoding JSON

```
<?php

  class Student {
      public $name = "";
      public $address  = "";
      public $age = "";
      public $birthdate = "";
  }
  $e = new Student();
  $e->name = "sachin";
  $e->age = "23";
  $e->address  = "pokhara";
  $e->birthdate = date('m/d/Y h:i:s a', "8/5/1974 12:20:03 p");

  $json =  json_encode($e);
  echo "<b> Json Encoding example : </b><br>";
  echo $json;

  echo "<br><b> Json Decoding example : </b><br>";

  //var_dump(json_decode($json));
  var_dump(json_decode($json, true));
//When the boolean parameter is set to TRUE then, the returned objects will be converted into associative arrays.
?>
```

OutPut:

```
Json Encoding example :
{"name":"sachin","address":"pokhara","age":"23","birthdate":"01\/01\/1970 05:30:08 am"}
Json Decoding example :
array(4) { ["name"]=> string(6) "sachin" ["address"]=> string(7) "pokhara" ["age"]=> string(2) "23" ["birthdate"]=> string(22) "01/01/1970 05:30:08 am" }
```

RDF Resource Description Framework:

RDF is a standard for describing Web resources.RDF can be used to describe title, author, content, and copyright information of web pages.

RDF is designed to be read and understood by computers. RDF is not designed for being displayed to people. RDF is written in XML. RDF is a part of the W3C's Semantic Web Activity. RDF is a W3C Recommendation.

RDF Rules:

RDF uses Web identifiers (URIs) to identify resources. RDF describes resources with properties and property values. A **Resource** is anything that can have a URI, e.g "http://www.w3schools.com/rdf". A **Property** is a Resource that has a name, such as "author" or "homepage". A **Property value** is the value of a Property, such as "Jan Egil Refsnes" or "http://www.w3schools.com" (note that a property value can be another resource). The following RDF document could describe the resource "http://www.w3schools.com/rdf":

```
<?xml version="1.0"?>
<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:si="http://www.w3schools.com/rdf/">


<RDF>
 <Description about="http://www.w3schools.com/rdf">
  <si:author>Jan Egil Refsnes</author>
  <si:homepage>http://www.w3schools.com</homepage>
 </Description>
</RDF>
```

RDF Statements

The combination of a Resource, a Property, and a Property value forms a **Statement** (known as the **subject, predicate and object** of a Statement).

Let's look at some example statements to get a better understanding:

Statement: "The author of http://www.w3schools.com/rdf is Jan Egil Refsnes".

- The subject of the statement above is: http://www.w3schools.com/rdf
- The predicate is: author
- The object is: Jan Egil Refsnes

Statement: "The homepage of http://www.w3schools.com/rdf is http://www.w3schools.com".

- The subject of the statement above is: http://www.w3schools.com/rdf
- The predicate is: homepage
- The object is: http://www.w3schools.com

RDF Example

Here are two records from a CD-list:

Title	Artist	Country	Company	Price	Year
Aaina Jhyal	Amrit Gurung	Nepal	MN	10	2012
Hide your heart	Bonnie Tyler	UK	CBS Records	9.90	1988

Below is a few lines from an RDF document:

```
<?xml version="1.0"?>

<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:cd="http://www.recshop.fake/cd#">

<rdf:Description
rdf:about="http://www.recshop.fake/cd/ Aaina Jhyal">
 <cd:artist>Amrit Gurung</cd:artist>
 <cd:country>Nepal</cd:country>
 <cd:company>MN</cd:company>
 <cd:price>10</cd:price>
 <cd:year>2012</cd:year>
</rdf:Description>

<rdf:Description
rdf:about="http://www.recshop.fake/cd/Hide your heart">
 <cd:artist>Bonnie Tyler</cd:artist>
 <cd:country>UK</cd:country>
 <cd:company>CBS Records</cd:company>
 <cd:price>9.90</cd:price>
 <cd:year>1988</cd:year>
</rdf:Description>
.
.
.
</rdf:RDF>
```

The first line of the RDF document is the XML declaration. The XML declaration is followed by the root element of RDF documents: **<rdf:RDF>**.

The **xmlns:rdf** namespace, specifies that elements with the rdf prefix are from the namespace "http://www.w3.org/1999/02/22-rdf-syntax-ns#".

The **xmlns:cd** namespace, specifies that elements with the cd prefix are from the namespace "http://www.recshop.fake/cd#".

The **<rdf:Description>** element contains the description of the resource identified by the **rdf:about** attribute.

The elements: **<cd:artist>, <cd:country>, <cd:company>,** etc. are properties of the resource.

Some uses of RDF:

- •Describing properties for shopping items, such as price and availability
- •Describing information about web pages (content, author, created and modified date)
- •Describing content for search engines
- •Describing electronic libraries

Online Validator for RDF:
http://www.w3.org/RDF/Validator/

Beginner examples and details regarding semantic web and RDF is avaliable at http://linkeddatatools.com/ .

References:

http://www.javaworld.com/
http://www.xml.com
http://www.copterlabs.com
http://webopedia.com/
http://www.quackit.com
http://www.tutorialspoint.com
http://www.helpdesksoftware.biz
http://www.w3schools.com
http://en.wikipedia.org
http://linkeddatatools.com