CMR
INSTITUTE OF
TECHNOLOGY

USN

| | | | | | | | | | |

Internal Assessment Test 3 Answer Key– June. 2021							
Sub:	Advanced Web Programming				Sub Code:	18MCA42	Branch:	MCA
Date:	14/06/2021	Duration:	90 min's	Max Marks:	50	Sem	IV	

Q1 a) What is Ajax? What are different technologies incorporated in Ajax (04 marks)
- AJAX stands for – Asynchronous JavaScript and XML
- Ajax is nothing more than an approach to web interaction. This approach involves transmitting only a small amount of information to and from the server in order to give the user the most responsive experience possible.
- Ajax is a set of web development techniques using many web technologies on the client side to create asynchronous web applications. With Ajax, web applications can send and retrieve data from a server asynchronously without interfering with the display and behaviour of the existing page

Technologies incorporated in Ajax
- HTML/XHTML: Primary content representation languages
- CSS: Provides stylistic formatting to XHTML
- DOM: Dynamic updating of a loaded page
- XML: Data exchange format
- XSLT: Transforms XML into XHTML (styled by CSS)
- XMLHttp: Primary communication broker
- JavaScript: Scripting language used to program an Ajax engine

Q1 b) Briefly explain various values of ready state and status property of XMLHttpRequestObject (06 marks)

There are five possible values for readyState:
0 (Uninitialized): The object has been created but the open() method hasn't been called.
1 (Loading): The open() method has been called but the request hasn't been sent.
2 (Loaded): The request has been sent.
3 (Interactive). A partial response has been received.
4 (Completed): All data has been received and the connection has been closed.

The status property is the property that contains the actual status of the download. This is actually the normal HTTP status code that you get when you try to download web pages. For example, if the data you're looking for wasn't found, you'll get a value of 404 in the status property.
200 : File Found
404 : File not found
403 : Forbidden
500 : Internal Server Error

Using the readyState Property
The readyState property lets you know the state of the download, and we want to wait until the value of this property equals 4, which means the download is complete. Here's what that looks

like in the anonymous function:
```
function getData(dataSource, divID)
{
if(XMLHttpRequestObject) {
XMLHttpRequestObject.open("GET", dataSource);
XMLHttpRequestObject.onreadystatechange = function()
{
if (XMLHttpRequestObject.readyState == 4
.
.
.
}
}
}
}
```
Okay, if we're executing code inside this if statement, we know that the data download was completed. But was it completed successfully, or was there an error? To check that, you have to examine the status property.

Using the status Property
The status property holds the HTTP code that corresponds to the status of the data transfer. We want to see a value of 200, which means that the data transfer was OK. Here's how we make sure that the XMLHttpRequest object's status property holds a value of 200:
```
function getData(dataSource, divID)
{
if(XMLHttpRequestObject) {
XMLHttpRequestObject.open("GET", dataSource);
XMLHttpRequestObject.onreadystatechange = function()
{
if (XMLHttpRequestObject.readyState == 4 &&
XMLHttpRequestObject.status == 200) {
.
.
.
}
}
}
}
```

Q2) Write a program that illustrate the creation of two HttpRequestobject in the same page (10 marks)
Program.html
```
<html>
<head>
<title>Using Two XMLHttpRequest Objects</title>
<script language = "javascript">
var XMLHttpRequestObject1 = false;
if (window.XMLHttpRequest)
{
XMLHttpRequestObject1 = new XMLHttpRequest();
} else if (window.ActiveXObject) {
XMLHttpRequestObject1 = new ActiveXObject("Microsoft.XMLHTTP");
}
var XMLHttpRequestObject2 = false;
```

```
if (window.XMLHttpRequest) {
XMLHttpRequestObject2 = new XMLHttpRequest();
} else if (window.ActiveXObject) {
XMLHttpRequestObject2 = new ActiveXObject("Microsoft.XMLHTTP");
}
function getData1(dataSource, divID)
{
if(XMLHttpRequestObject1)
{
var obj = document.getElementById(divID);
XMLHttpRequestObject1.open("GET", dataSource);
XMLHttpRequestObject1.onreadystatechange = function()
{
if (XMLHttpRequestObject1.readyState == 4 &&
XMLHttpRequestObject1.status == 200) {
obj.innerHTML = XMLHttpRequestObject1.responseText;
}
}
XMLHttpRequestObject1.send(null);
}
}
function getData2(dataSource, divID)
{
if(XMLHttpRequestObject2) {
var obj = document.getElementById(divID);
XMLHttpRequestObject2.open("GET", dataSource);
XMLHttpRequestObject2.onreadystatechange = function()
{
if (XMLHttpRequestObject2.readyState == 4 &&
XMLHttpRequestObject2.status == 200) {
obj.innerHTML = XMLHttpRequestObject2.responseText;
}
}
XMLHttpRequestObject2.send(null);
}
}
<body>
<h1>Using Two XMLHttpRequest Objects</h1>
<form>
 <input type = "button" value = "Fetch message 1" onclick =
"getData1('dataresponder.php?data=1', 'targetDiv')">
 <input type = "button" value = "Fetch message 2" onclick =
"getData2('dataresponder.php?data=2', 'targetDiv')">
</form>
<div id="targetDiv">
 <p> </p>
</div>
</body>
</html>
```

Q3) With a code snippet demonstrate sending data to the server using GET method (10 marks)

```
<html>
        <head>
                <title>An Ajax example</title>
```

```html
<script language = "javascript">
        var ajaxobj = false;
                if (window.XMLHttpRequest) {
                        ajaxobj = new XMLHttpRequest();
                } else if (window.ActiveXObject) {
                        ajaxobj = new ActiveXObject("Microsoft.XMLHTTP");
                }
        function getData(dataSource, divID){
                if(ajaxobj) {
                        var obj = document.getElementById(divID);
                        ajaxobj.open("GET", dataSource);
                        ajaxobj.onreadystatechange = function()
                        {
                                if (ajaxobj.readyState == 4 &&
                                        ajaxobj.status == 200) {
                                        obj.innerHTML = ajaxobj.responseText;
                                }
                        }


                        ajaxobj.send(null);
                }
        }
</script>
</head>
<body>
        <H1>An Ajax example</H1>
        <form>
                <input type = "button" value = "Fetch the first message"
                        onclick = "getData('dataresponder.php?data=1', 'targetDiv')">
                        <input type = "button" value = "Fetch the second message"
                        onclick = "getData('dataresponder.php?data=2', 'targetDiv')">
        </form>
        <div id="targetDiv">
                <p>The fetched message will appear here.</p>
        </div>
</body>
</html>
```

dataresponder.php
```php
<?php
if ($_GET["data"] == "1") {
echo 'The server got a value of 1';
}
if ($_GET["data"] == "2") {
echo 'The server got a value of 2';
}
?>
```

Q4) Explain how data is sent to the server using POST method with URL encoding (10 marks)
```html
<html>
        <head>
                <title>An Ajax example</title>
                <script language = "javascript">
```

```
                        var XMLHttpRequestObject = false;
                            if (window.XMLHttpRequest) {
                                    XMLHttpRequestObject = new XMLHttpRequest();
                            } else if (window.ActiveXObject) {
                                    XMLHttpRequestObject = new
ActiveXObject("Microsoft.XMLHTTP");
                            }
                        function getData(dataSource, divID, data){
                                if(XMLHttpRequestObject) {
                                    var obj = document.getElementById(divID);
                                    XMLHttpRequestObject.open("POST", dataSource);
                                    XMLHttpRequestObject.setRequestHeader('Content-Type',
'application/x-www-form-urlencoded');
                                    XMLHttpRequestObject.onreadystatechange = function()
                                    {
                                            if (XMLHttpRequestObject.readyState == 4 &&
                                                XMLHttpRequestObject.status == 200) {
                                                obj.innerHTML =
XMLHttpRequestObject.responseText;
                                            }
                                    }


                                    XMLHttpRequestObject.send("data="+data);
                                }
                        }
                </script>
        </head>
        <body>
                <H1>An Ajax example</H1>
                <form>
                        <input type = "button" value = "Fetch the first message"
                                onclick = "getData('dataresponder.php','targetDiv',1)">
                                <input type = "button" value = "Fetch the second message"
                                onclick = "getData('dataresponder.php','targetDiv',2)">
                </form>
                <div id="targetDiv">
                        <p>The fetched message will appear here.</p>
                </div>
        </body>
</html>
```

dataresponder.php
```php
<?php
if ($_GET["data"] == "1") {
echo 'The server got a value of 1';
}
if ($_GET["data"] == "2") {
echo 'The server got a value of 2';
}
?>
```

Q5 a) What are the key principles of a good Ajax pattern (04 marks)

Minimal traffic: Ajax applications should send and receive as little information as possible to and from the server. In short, Ajax can minimize the amount of traffic between the client and the server. Making sure that your Ajax application doesn't send and receive unnecessary information adds to its robustness.

No surprises: Ajax applications typically introduce different user interaction models than traditional web applications. As opposed to the web standard of clickand-wait, some Ajax applications use other user interface paradigms such as dragand-drop or double-clicking. No matter what user interaction model you choose, be consistent so that the user knows what to do next.

Established conventions: Don't waste time inventing new user interaction models that your users will be unfamiliar with. Borrow heavily from traditional web applications and desktop applications, so there is a minimal learning curve.

No distractions: Avoid unnecessary and distracting page elements such as looping animations and blinking page sections. Such gimmicks distract the user from what he or she is trying to accomplish.

Accessibility: Consider who your primary and secondary users will be and how they most likely will access your Ajax application. Don't program yourself into a corner so that an unexpected new audience will be completely locked out. Will your users be using older browsers or special software? Make sure you know ahead of time and plan for it.

Avoid entire page downloads: All server communication after the initial page download should be managed by the Ajax engine. Don't ruin the user experience by downloading small amounts of data in one place but reloading the entire page in others.

User first: Design the Ajax application with the users in mind before anything else. Try to make the common use cases easy to accomplish and don't be caught up with

Q5 b) Compare two models of web applications with their respective diagrams. (06 marks)
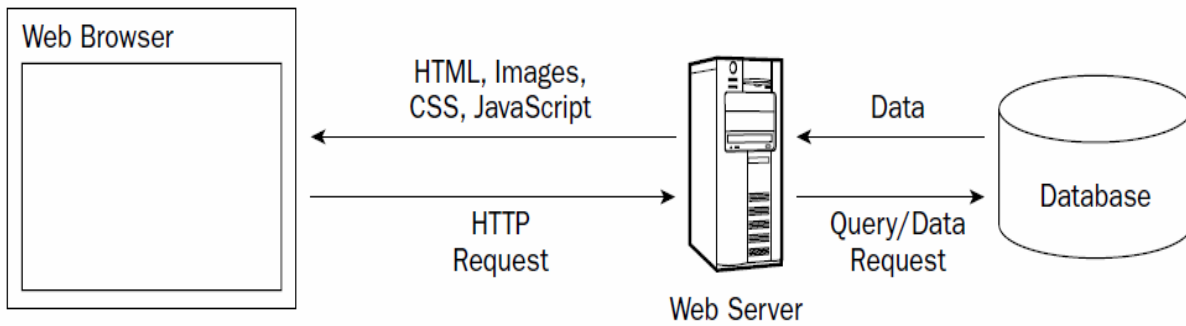
Ajax is nothing more than an approach to web interaction. This approach involves transmitting only a small amount of information to and from the server in order to give the user the most responsive experience possible.

Instead of the traditional web application model where the browser itself is responsible for initiating requests to, and processing requests from, the web server, the Ajax model provides an intermediate layer —what Garrett calls an Ajax engine— to handle this communication. An Ajax engine is really just a JavaScript object or function that is called whenever information needs to be requested from the server. Instead of the traditional model of providing a link to another resource (such as another web page), each link makes a call to the Ajax engine, which schedules and executes the request. The request is done asynchronously, meaning that code execution doesn't wait for a response before continuing.
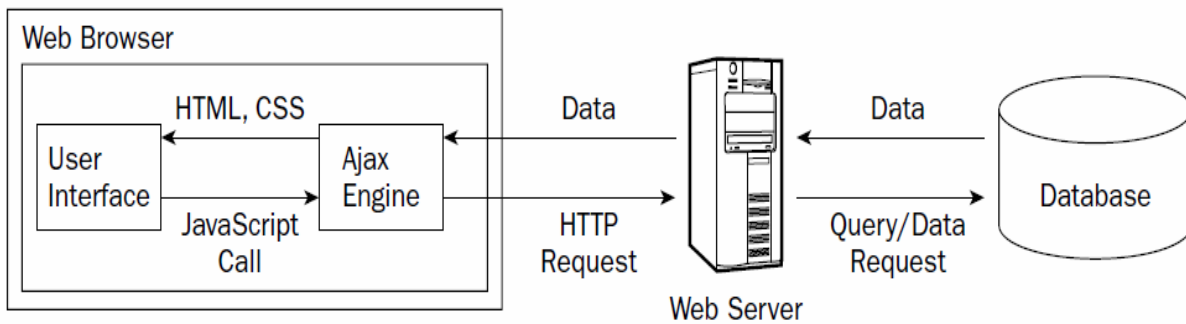
The server — which traditionally would serve up HTML, images, CSS, or JavaScript — is configured to return data that the Ajax engine can use. This data can be plain text, XML, or any other data format that you may need. The only requirement is that the Ajax engine can understand and interpret the data

When the Ajax engine receives the server response, it goes into action, often parsing the data and making several changes to the user interface based on the information it was provided. Because this process involves transferring less information than the traditional web application model, user interface updates are faster, and the user is able to do his or her work more quickly. Figure below is an adaptation of the figure in Garrett's article, displaying the difference between the traditional and Ajax web application models.

Traditional Web Application Model

Web Browser
HTML, Images, CSS, JavaScript
HTTP Request
Web Server
Data
Query/Data Request
Database

Ajax Web Application Model

Web Browser
User Interface
HTML, CSS
JavaScript Call
Ajax Engine
Data
HTTP Request
Web Server
Data
Query/Data Request
Database

Q6) Write a short note on following i) submission throttling ii)Multistage download iii)Fallback pattern (10 marks)

i)   submission throttling

Using Submission Throttling, you buffer the data to be sent to the server on the client and then send the data at predetermined times. The venerable Google Suggest feature does this brilliantly. It doesn't send a request after each character is typed. Instead, it waits for a certain amount of time and sends all the text currently in the textbox. The delay from typing to sending has been fine-tuned to the point that it doesn't seem like much of a delay at all. Submission Throttling, in part, gives Google Suggest its speed. Submission Throttling typically begins either when the web site or application first loads or because of a specific user action. Then, a client-side function is called to begin the buffering of data. Every so often, the user's status is checked to see if he or she is idle (doing so prevents any interference with the user interface). If the user is still active, data continues to be collected. When the user is idle, which is to say he or she is not performing an action, it's time to decide whether to send the data. This determination varies depending on your use case; you may want to send data only when it reaches a certain size, or you may want to send it every time the user is idle. After the data is sent, the application typically continues to gather data until either a server response or some user action signals to stop the data collection.
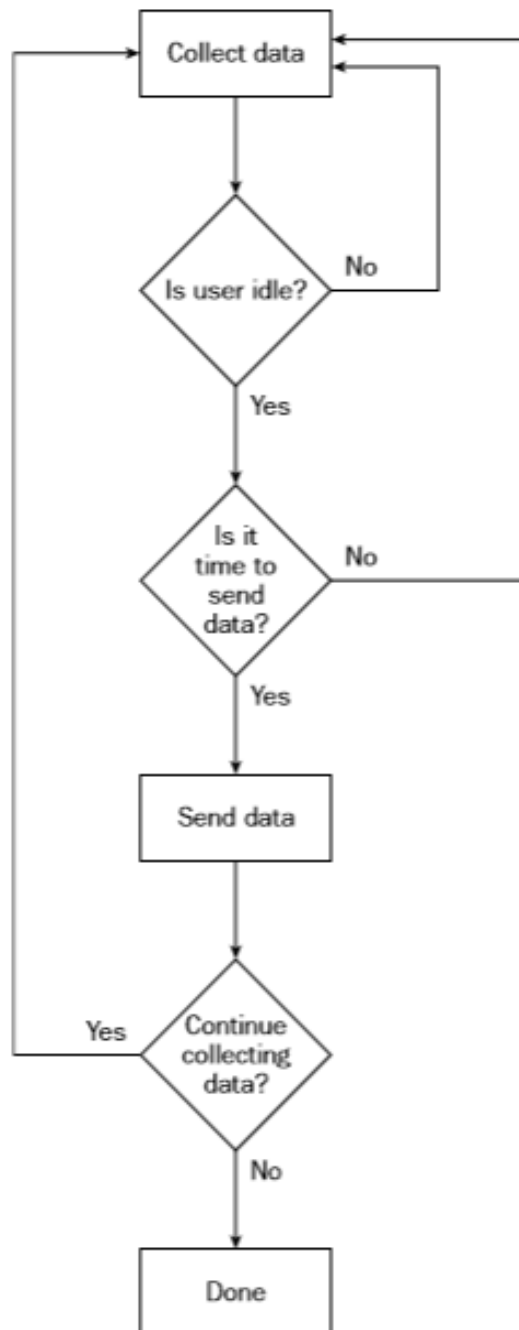
ii)   **Multistage download**

Multi-Stage Download is an Ajax pattern wherein only the most basic functionality is loaded into a page initially. Upon completion, the page then begins to download other components that should appear on the page. If the user should leave the page before all of the components are downloaded, it's of no consequence. If, however, the user stays on the page for an extended period of time (perhaps reading an article), the extra functionality is loaded in the background and available when the user is ready.

The major advantage here is that you, as the developer, get to decide what is downloaded and at what point in time. This is a fairly new Ajax pattern and has been popularized by Microsoft's start.com. When you first visit start.com, it is a very simple page with a search box in the middle. Behind the scenes, however, a series of requests is being fired off to fill in more content on the page. Within a few seconds, the page jumps to life as content from several different locations is pulled in and displayed.

Although nice, Multi-Stage Download does have a downside: the page must work in its simplest form for browsers that don't support Ajax technologies. This means that all the basic

functionality must work without any additional downloads. The typical way of dealing with this problem is to provide graceful degradation, meaning that those browsers that support Ajax technologies will get the more extensive interface while other browsers get a simple, bare-bones interface. This is especially important if you are expecting search engines to crawl your site; since these

iii) Fallback pattern
Cancel Pending Requests
If an error occurs on the server, meaning that a status of something other than 200 or 304 is returned, you need to decide what to do. Chances are that if a file is not found (404) or an internal server error occurred (302), trying again in a few minutes isn't going to help, since both of these require an administrator to fix the problem. The simplest way to deal with this situation is to simply cancel all pending requests. You can set a flag somewhere in your code that says, "don't send any more requests." This clearly has the highest impact on solutions using the Periodic Refresh pattern.
The comment notification example can be modified to take this into account. This is a case where the Ajax solution provides additional value to the user but is not the primary focus of the page. If a request fails, there is no reason to alert the user; you can simply cancel any future requests to prevent any further errors from occurring. To do so, you must add a global variable that indicates whether requests are
enabled:
var oXHR = null;
var iInterval = 1000;
var iLastCommentId = -1;
var divNotification = null;
var blnRequestsEnabled = true;

Now, the blnRequestsEnabled variable must be checked before any request is made. This can be accomplished by wrapping the body of the checkComments() function inside of an if statement:

```
function checkComments() {
if (blnRequestsEnabled) {
if (!oXHR) {
oXHR = zXmlHttp.createRequest();
} else if (oXHR.readyState != 0) {
oXHR.abort();
}
oXHR.open("get", "CheckComments.php", true);
oXHR.onreadystatechange = function () {
if (oXHR.readyState == 4) {
if (oXHR.status == 200 || oXHR.status == 304) {
var aData = oXHR.responseText.split("||");
if (aData[0] != iLastCommentId) {
iLastCommentId = aData[0];
if (iLastCommentId != -1) {
showNotification(aData[1], aData[2]);
}
}
setTimeout(checkComments, iInterval);
}
}
};
oXHR.send(null);
```

```
}
}
```

But that isn't all that must be done; you must also detect the two different types of errors that may occur: server errors that give status codes and a failure to reach the server (either the server is down or the Internet connection has been lost).

To begin, wrap everything inside of the initial if statement inside a try...catch block. Different browsers react at different times when a server can't be reached, but they all throw errors. Wrapping the entire request block in a try...catch ensures that you catch any error that is thrown, at which point you can set blnRequestsEnabled to false. Next, for server errors, you can also set blnRequestsEnabled to false whenever the status is not equal to 200 or 304. This will have the same effect as if the server couldn't be reached:

```
function checkComments() {
if (blnRequestsEnabled) {
try {
if (!oXHR) {
oXHR = zXmlHttp.createRequest();
} else if (oXHR.readyState != 0) {
oXHR.abort();
}
oXHR.open("get", "CheckComments.php", true);
oXHR.onreadystatechange = function () {
if (oXHR.readyState == 4) {
if (oXHR.status == 200 || oXHR.status == 304) {
var aData = oXHR.responseText.split("||");
if (aData[0] != iLastCommentId) {
if (iLastCommentId != -1) {
showNotification(aData[1], aData[2]);
}
iLastCommentId = aData[0];
}
setTimeout(checkComments, iInterval);
} else {
blnRequestsEnabled = false;
}
}
};
oXHR.send(null);
} catch (oException) {
blnRequestsEnabled = false;
}
}
}
```

Now, when either of the two error types occurs, an error will be thrown (either by the browser or by you), and the blnRequestsEnabled variable will be set to false, effectively canceling any further
requests if checkComments() is called again.

Try Again
Another option when dealing with errors is to silently keep trying for either a specified amount of time or a particular number of tries. Once again, unless the Ajax functionality is key to the user's experience,

there is no need to notify him or her about the failure. It is best to handle the problem behind the scenes until it can be resolved.

To illustrate the Try Again pattern, consider the Multi-Stage Download example. In that example, extra links were downloaded and displayed alongside the article. If an error occurred during the request, an

error message would pop up in most browsers. The user would have no idea what the error was or what caused it, so why bother displaying a message at all? Instead, it would make much more sense to continue trying to download the information a few times before giving up.

To track the number of failed attempts, a global variable is necessary:

var iFailed = 0;

The iFailed variable starts at 0 and is incremented every time a request fails. So, if iFailed is ever greater than a specific number, you can just cancel the request because it is clearly not going to work. If, for example, you want to try 10 times before canceling all pending requests, you can do the following

```
function downloadLinks() {
var oXHR = zXmlHttp.createRequest();
if (iFailed < 10) {
try {
oXHR.open("get", "AdditionalLinks.txt", true);
oXHR.onreadystatechange = function () {
if (oXHR.readyState == 4) {
if (oXHR.status == 200 || oXHR.status == 304) {
var divAdditionalLinks =
document.getElementById("divAdditionalLinks");
divAdditionalLinks.innerHTML = oXHR.responseText;
divAdditionalLinks.style.display = "block";
} else {
iFailed++;
downloadLinks();
}
}
}
oXHR.send(null);
} catch (oException) {
iFailed++;
downloadLinks();
}
}
}
```

This code is constructed similarly to the previous example. The try...catch block is used to catch any errors that may occur during the communication, and a custom error is thrown when the status isn't 200 or 304. The main difference is that when an error is caught, the iFailed variable is incremented and downloadLinks() is called again. As long as iFailed is less than 10 (meaning it's failed less than 10 times), another request will be fired off to attempt the download.

In general, the Try Again pattern should be used only when the request is intended to occur only once,

as in a Multi-Stage Download. If you try to use this pattern with interval-driven requests, such as Periodic Refresh, you could end up with an ever-increasing number of open requests taking up memory

Q7) Write a program to demonstrate the Ajax application for the following (10 marks)

1. fetch the data from the server behind the scene and updates its web page in the browser without causing page refresh
2. use open an onClick command

```html
<html>
        <head>
                <title>An Ajax example</title>
                <script language = "javascript">
                        var XMLHttpRequestObject = false;
                                if (window.XMLHttpRequest) {
                                        XMLHttpRequestObject = new XMLHttpRequest();
                                } else if (window.ActiveXObject) {
                                        XMLHttpRequestObject = new
ActiveXObject("Microsoft.XMLHTTP");
                                }
                        function getData(dataSource, divID){
                                if(XMLHttpRequestObject) {
                                        var obj = document.getElementById(divID);
                                        XMLHttpRequestObject.open("GET", dataSource);
                                        XMLHttpRequestObject.onreadystatechange = function()
                                        {
                                                if (XMLHttpRequestObject.readyState == 4 &&
                                                        XMLHttpRequestObject.status == 200) {
                                                        obj.innerHTML =
XMLHttpRequestObject.responseText;
                                                }
                                        }


                                        XMLHttpRequestObject.send(null);
                                }
                        }
                </script>
        </head>
        <body>
                <H1>An Ajax example</H1>
                <form>
                        <input type = "button" value = "Fetch the message"
                                onclick = "getData('data.txt', 'targetDiv')">
                </form>
                <div id="targetDiv">
                        <p>The fetched message will appear here.</p>
                </div>
        </body>
</html>
```

Q8) Illustrate with an example predictive fetch pattern (10 marks)

In a traditional web solution, the application has no idea what is to come next. A page is presented with any number of links, each one leading to a different part of the site. This may be termed "fetch on demand," where the user, through his or her actions, tells the server exactly what data should be retrieved. While this paradigm has defined the Web since its inception, it has the unfortunate side effect of forcing the start-and-stop model of user interaction upon the user. The Predictive Fetch pattern is a relatively simple idea that can be somewhat difficult to implement: the Ajax application guesses what the user is going to do next and retrieves the appropriate data. In a perfect world, it

would be wonderful to always know what the user is going to do and make sure that the next data is readily available when needed.

 In reality, however, determining future user action is just a guessing game depending on your intentions There are simple use cases where predicting user actions is somewhat easier. Suppose that you are reading an online article that is separated into three pages. It is logical to assume that if you are interested in reading the first page, you're also interested in reading the second and third page. So, if the first page has been loaded for a few seconds (which can easily be determined by using a timeout), it is probably safe to download the second page in the background. Likewise, if the second page has been loaded for a few seconds, it is logical to assume that the reader will continue on to the third page. As this extra data is being loaded and cached on the client, the reader continues to read and barely even notices that the next page comes up almost instantaneously after clicking the Next Page link. The Google Maps is another real world example for predictive fetch pattern. It predicts the nearby places when we search a particular destination.

ArticleExample.php contains code for displaying an article online:

```php
<?php
$page = 1;
$dataOnly = false;
if (isset($_GET["page"])) {
$page = (int) $_GET["page"];
}
if (isset($_GET["dataonly"]) && $_GET["dataonly"] == "true") {
$dataOnly = true;
}
if (!$dataOnly) {
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Article Example</title>
<script type="text/javascript" src="zxml.js"></script>
<script type="text/javascript" src="Article.js"></script>
<link rel="stylesheet" type="text/css" href="Article.css" />
</head>
<body>
<h1>Article Title</h1>
<div id="divLoadArea" style="display:none"></div>
<?php
$output = "<p>Page ";
for ($i=1; $i < 4; $i++) {
$output .= "<a href=\"ArticleExample.php?page=$i\" id=\"aPage$i\"";
if ($i==$page) {
$output .= "class=\"current\"";
}
$output .= ">$i</a> ";
}
echo $output;
}
if ($page==1) {
?>
<div id="divPage1"><!-- contents of page 1 --></div>
<?php
} else if ($page == 2) {
```

```php
?>
<div id="divPage2"><!-- contents of page 2 --></div>
<?php
} else if ($page == 3) {
?>
<div id="divPage3"><!-- contents of page 3 --></div>
<?php
}
if (!$dataOnly) {
?>
</body>
</html>
<?php
}
?>
```

Q9) Create an ajax application to display the output "The second employee is Akhil , employee id 1AB36 from Sales department ".

Employee.xml

```xml
<employee>
    <e1>
       <eid>1AB23</eid>
       <ename>Akash</ename>
       <dept>Marketting</dept>
    </e1>
    <e1>
       <eid>1AB36</eid>
       <ename>Akhil</ename>
       <dept>Sales</dept>
    </e1>
    <e1>
       <eid>1AB39</eid>
       <ename>Arun</ename>
       <dept>Purchase</dept>
    </e1>
```

Xmlajax.html

```
function callajax()
{
If(window.XMLHTTPRequest)
var xhttp = new XMLHTTPRequest();
else
var xhttp = new ActiveXObject("Microsoft.XMLHTTP");
xhttp.open("GET","student.xml",true);
xhttp.send();
xhttp.overrideMimeType("text/xml");
xhttp.onreadystatechange = function()
{
If((xhttp.readyState == 4) && (xhttp.status == 200))
{
var xmldoc = xhttp.responseXML;
var eid = xmldoc.getElementByTagName("eid");
var ename = xmldoc.getElementByTagName("ename");
var dept = xmldoc.getElementByTagName("dept");
document.getElementById("id1").innerHTML = "The second
```

employee is "+ename[2].nodeValue+" , employee id "+eid[2].nodeValue+" from "+dept[2].nodeValue +" department " ;
 }
 }
}
Page.html
<html>
<body onload="callajax()"><div id="id1"></div>
</body>
</html>

Q10) Explain how does Ajax handles whitespace in firefox with an example (10 marks)

When it comes to whitespace, Firefox by default acts differently than
Internet Explorer. In Firefox, whitespace that you use to indent the elements in your
XML counts as text nodes. So when navigating, we have to take all the whitespace
nodes into account in Firefox, by default. We can strip out indentation whitespace
before Firefox gets its hands on it. To do that, we might write a JavaScript function
named removeWhitespace, which is passed a JavaScript XML document object:

```
xhttp.onreadystatechange = function()
{
if (xhttp.readyState == 4 && xhttp.status == 200)
{
 var xmlDocument = xhttp.responseXML;
 removeWhitespace(xmlDocument);
}
}
function removeWhitespace(xml)
{
 var loopIndex;
 for (loopIndex = 0; loopIndex < xml.childNodes.length;loopIndex++)
 {
 var currentNode = xml.childNodes[loopIndex];
 if (currentNode.nodeType == 1)
 {
 removeWhitespace(currentNode);
 }
 if (((/^\s+$/.test(currentNode.nodeValue))) && (currentNode.nodeType == 3))
 {
 xml.removeChild(xml.childNodes[loopIndex--]);
 }
 }
}
```