CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | | |

Internal Assessment Test 3– July. 2023

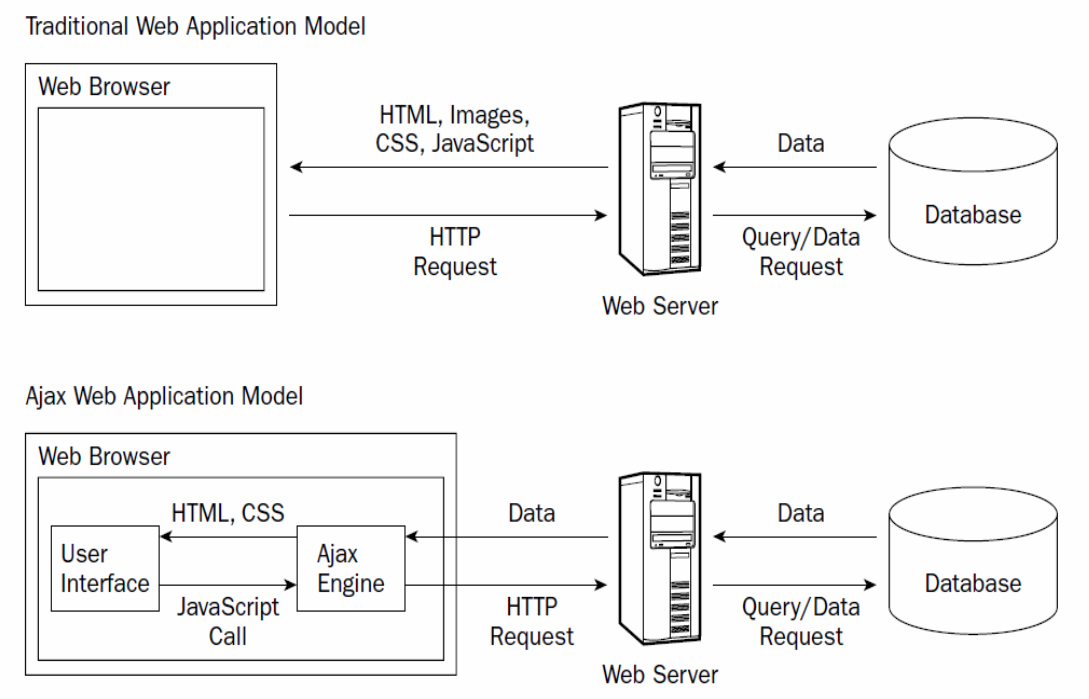
Sub:	Advanced Web Technologies							Sub Code:	20MCA41
Date:	22/7/2023	Duration:	90 min's	Max Marks:	50	Sem:	IV	Branch:	MCA

Note : Answer FIVE FULL Questions, choosing ONE full question from each Module

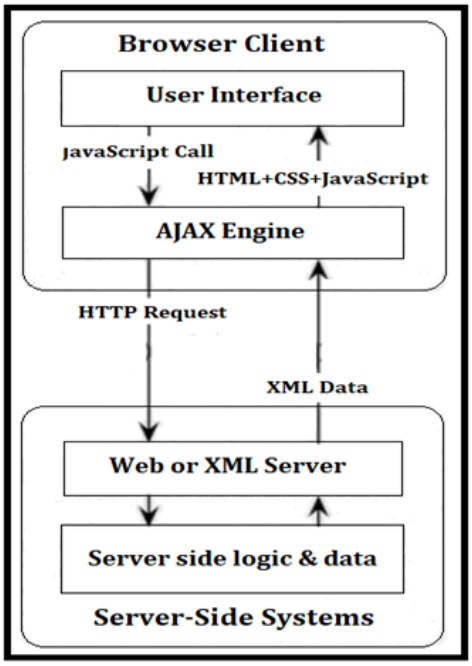
		MARKS	OBE	
			CO	RBT
	PART I			
1	What is Ajax? Explain with an diagram how it is different from traditional web applications.	[10]	CO3	L2
	OR			
2	With a neat diagram explain Ajax web application model	[10]	CO3	L2
	PART II			
3	With an example illustrate sending data to server using GET method for HTTPRequest Object	[10]	CO3	L3
	OR			
4	With an example illustrate sending data to server using POST method for HTTPRequest Object	[10]	CO3	L3
	PART III			
5	How to handle multiple XMLHTTPRequest objects in the same page	[10]	CO5	L3
	OR			
6	How to handle multiple XMLHTTPRequest objects in the same page using array	[10]	CO5	L3
	PART IV			
7	Explain the pattern of predictive fetch using Ajax	[10]	CO3	L3
	OR			
8	Write a note on submission throttling	[10]	CO3	L3
	PARTV			
9	With example explain pattern of periodic refresh and fallback pattern in Ajax	[10]	CO3	L3
	OR			
10	Illustrate how to implement multistage download	[10]	CO3	L5

Q1) What is Ajax? Explain with an diagram how it is different from traditional web applications.

- 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
- 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.



Q2) With a neat diagram explain Ajax web application model



AJAX Web Application Model

- 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
- 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.

Q3) With an example illustrate sending data to server using GET method for HTTPRequest Object

```
<html>
	<head>
		<title>An Ajax example</title>
		<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
 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) With an example illustrate sending data to server using POST method for HTTPRequest Object

```
<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
 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) How to handle multiple XMLHTTPRequest objects in the same page

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>
```

Q6) How to handle multiple XMLHTTPRequest objects in the same page using array

```
<html>
	<head>
		<title>An Ajax example</title>
		<script language = "javascript">

			var index = 0;

			var XMLHttpRequestObjects = new Array();

			 function getData1(dataSource, divID)
			 {
				if (window.XMLHttpRequest) {
					XMLHttpRequestObjects.push(new XMLHttpRequest());
				} else if (window.ActiveXObject) {
					XMLHttpRequestObjects.push(new
ActiveXObject("Microsoft.XMLHTTP"));
				}
				index = XMLHttpRequestObjects.length - 1;
				if(XMLHttpRequestObjects[index]) {
					XMLHttpRequestObjects[index].open("GET", dataSource);
					 var obj = document.getElementById(divID);
						XMLHttpRequestObjects[index].onreadystatechange =
function()
						{
							if (XMLHttpRequestObjects[index].readyState == 4
&&
								XMLHttpRequestObjects[index].status == 200)
{
								obj.innerHTML =
XMLHttpRequestObjects[index].responseText;
							}
						}
						XMLHttpRequestObjects[index].send(null);
					 }
				}
				function getData2(dataSource, divID)
				{
					if (window.XMLHttpRequest) {
					XMLHttpRequestObjects.push(new XMLHttpRequest());
				} else if (window.ActiveXObject) {
```

```
                                        XMLHttpRequestObjects.push(new
ActiveXObject("Microsoft.XMLHTTP"));
                                }
                                index = XMLHttpRequestObjects.length - 1;
                                if(XMLHttpRequestObjects[index]) {
                                        XMLHttpRequestObjects[index].open("GET", dataSource);
                                         var obj = document.getElementById(divID);
                                                XMLHttpRequestObjects[index].onreadystatechange =
function()
                                                {
                                                        if (XMLHttpRequestObjects[index].readyState == 4
&&
                                                                XMLHttpRequestObjects[index].status == 200)
{
                                                                obj.innerHTML =
XMLHttpRequestObjects[index].responseText;
                                                        }
                                                }
                                                XMLHttpRequestObjects[index].send(null);
                                        }

                                }
                </script>
        </head>
        <body>
                <H1>An Ajax example</H1>
                <form>
                        <input type = "button" value = "Fetch the first message"
                                onclick = "getData1('dataresponder.php?data=1', 'targetDiv')">
                                <input type = "button" value = "Fetch the second message"
                                onclick = "getData2('dataresponder.php?data=2', 'targetDiv')">
                </form>
                <div id="targetDiv">
                        <p>The fetched message will appear here.</p>
                </div>
        </body>
</html>
```

dataresponder.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';
 }
?>
```

Q7) Explain the pattern of predictive fetch using Ajax

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.

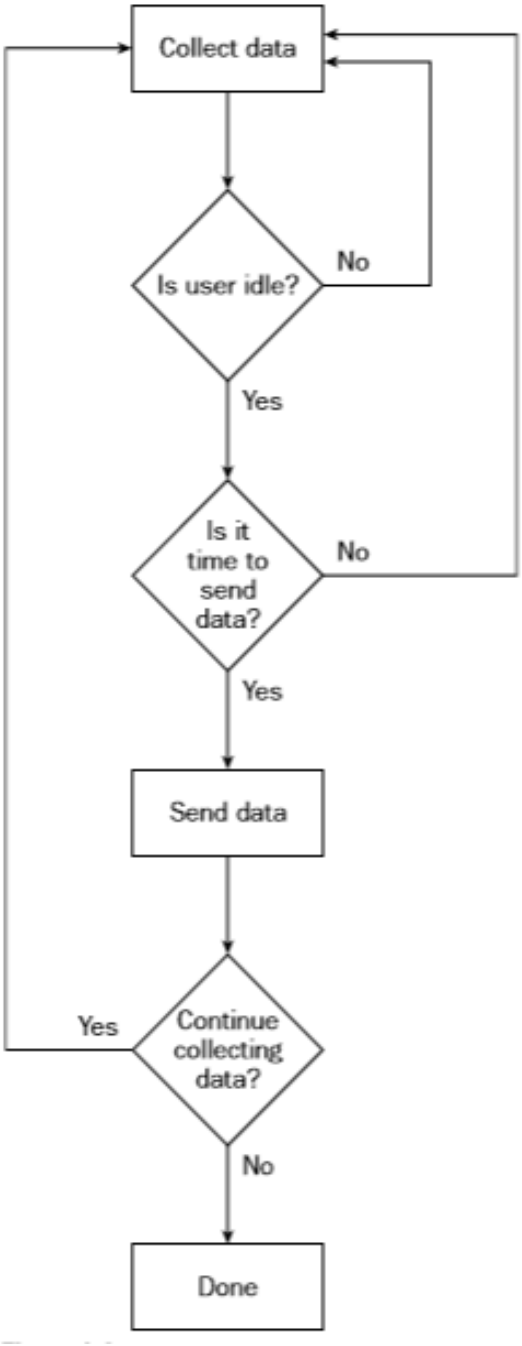
In reality, however, determining future user action is just a guessing game depending on your intentions There are simple use cases where predicting user actions is somewhat easier. Suppose that you are reading an online article that is separated into three pages. It is logical to assume that if you 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
$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 ($page == 2) {
?>
<div id="divPage2"><!-- contents of page 2 --></div>
<?php
} else if ($page == 3) {
?>
<div id="divPage3"><!-- contents of page 3 --></div>
<?php
}
if (!$dataOnly) {
?>
</body>
</html>
<?php
}
?>
```

Q8) Write a note on 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.



Q9) With example explain pattern of periodic refresh and fallback pattern in Ajax

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

```
if (!$dataOnly) {
?>
</body>
</html>
<?php
}
?>
```

i) 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

Q10) Illustrate how to implement 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