Web Component Development Using Java

Session: 6

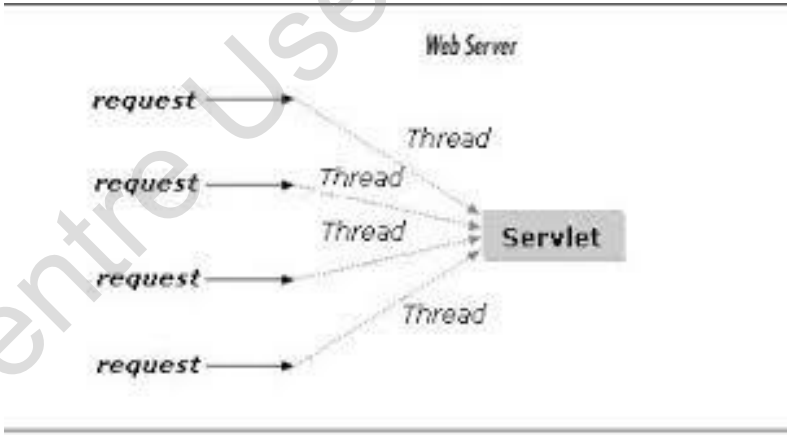
Asynchronous Servlet Communication

Objectives

- Explain the need of Asynchronous Servlet
- Explain how to create Asynchronous Servlet and Asynchronous Listener
- Explain the concept of server push mechanism
- Explain how to create an asynchronous JavaScript client using XMLHttpRequest object
- Explain the need of non-blocking I/O support in Servlet
- Explain how to implement non-blocking I/O in asynchronous Servlet
- Explain the need for protocol upgrade
- Explain the process of protocol upgrade in a Servlet

Introduction

- The architecture of the Servlet is based on **Multithreaded model**.
- The Web container creates Servlet threads to process the client requests.
- Each client will receive its dedicated thread which will serve the request and generate a response to be sent back to the client.



- A thread associated with the client request may not process the request all the time and may sit idle because of the following reasons:
  - Servlet may take a long running task like making a database connection call for query execution or invoking a remote Web service.
  - Servlet may have to wait for some dependent event to proceed for the response generation.

Servlet 3.0 API has provided a new feature in its specification to process the request asynchronously.

Asynchronous Processing in Servlets

- The Java Servlet API provides asynchronous processing support for servlets and filters in the Web applications.
- The steps for processing the request asynchronously in the Servlet are as follows:

The request sent to the Servlet is intercepted by the filters which perform some pre-processing tasks for the Servlet.

The Servlet receive the request parameters to process them.

The Servlet requests for some external connection or data. For example, JDBC connection.

The Servlet returns back to the pool without generating a response.

The JDBC connection operation is assigned to an asynchronous context object which is responsible for handling the request in the background thread.

When the JDBC connection is available, the asynchronous context object gets the connection object and notifies the Servlet container for a Servlet thread to handle the connection.

Handling Asynchronous Servlet

- To support asynchronous processing, the separation of request from the generated response needs to be separated.
- The `javax.servlet.AsyncContext` class is used to process the request asynchronously within the Servlet.
- Apart from creating the object of `AsyncContext`, a user need to enable the Servlet class with asynchronous processing.
- This is done by enabling the attribute `asyncSupported` to true in the `@WebServlet` annotation.

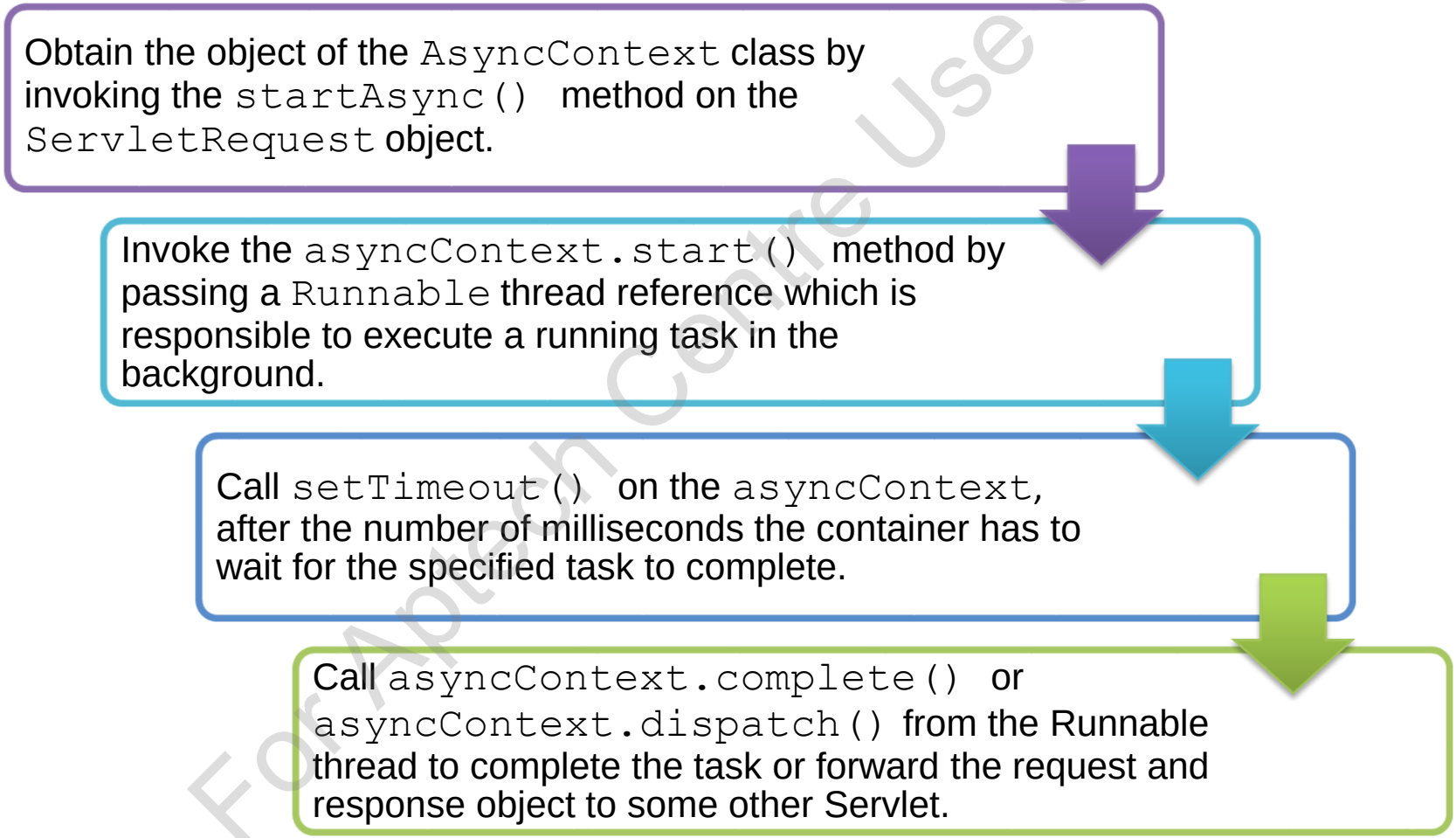
AsyncContext Class 1-10

- The `AsyncContext` class provide methods that can be used to obtain the instance of the `AsyncContext` within the Servlet `service()` method.
- Table lists the methods of the `AsyncContext` class.

Method	Description
`void start(Runnable run)`	Gets a thread from the managed thread pool to run the specified Runnable thread.
`ServletRequest getRequest()`	Obtains the parameters from the request for the asynchronous context.
`ServletResponse getResponse()`	Used inside the asynchronous context to write to the response with the results of the blocking operation.
`void complete()`	Completes the asynchronous operation and closes the response object associated with the asynchronous object.
`void dispatch(String path)`	Dispatches the request and response objects to the URL to be forwarded to another Servlet thread.

AsyncContext Class 2-10

- Following are the steps to be performed to handle asynchronous request in the asynchronous Servlet:



AsyncContext Class 3-10

- The code snippet shows the skeleton for obtaining the asynchronous context within the Servlet `doGet()` or `doPost()` method.

```
@WebServlet(urlPatterns={"/asyncservlet"}, asyncSupported=true)

public class AsyncServlet extends HttpServlet {

 public void doGet(HttpServletRequest request,
HttpServletResponse
        response) throws ServletException, IOException {

Final AsyncContext = request.startAsync ();
asyncContext.setTimeout (. . .);

   asyncContext.start (new Runnable () {

    @Override

   public void run () { // long running task

    . . .

   }})
  }
}
```

AsyncContext Class 4-10

- The code snippet demonstrates how to process the request asynchronously in the Servlet.

```
. . .
@WebServlet(name = "AsyncDispatchServlet", urlPatterns = { "/
asyncDispatch" }, asyncSupported = true)

public class AsyncDispatchServlet extends HttpServlet {
  @Override
  public void doGet(final HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
IOException {
    final AsyncContext = request.startAsync();
    request.setAttribute("mainThread", Thread.currentThread().
getName());
    asyncContext.setTimeout(5000);
    asyncContext.start(new Runnable() {
      @Override
      public void run() {
     // long-running task
          try {
                    Thread.sleep(3000);
              } catch (InterruptedException e) { . . . }

        request.setAttribute("workerThread", Thread.

                                  currentThread().getName());
        //Dispatch the request object
          asyncContext.dispatch("/threadNames.jsp");
   }});
  }
}
```

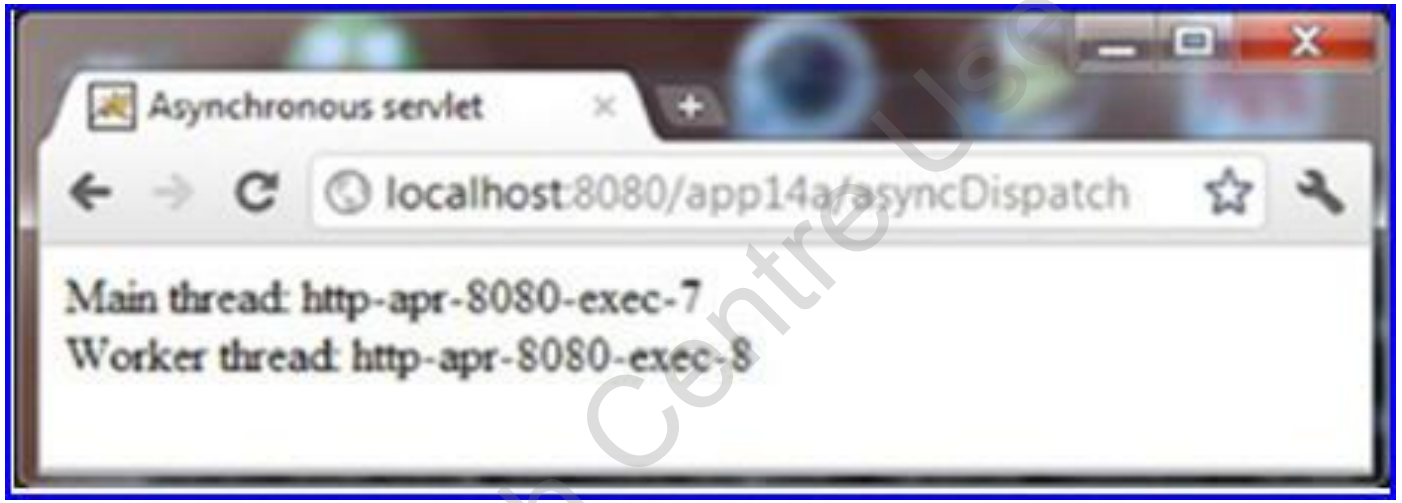
AsyncContext Class 5-10

- The code snippet shows the `threadNames.jsp` page.

```
<!DOCTYPE HTML>
<html>
<head>
<title>Asynchronous servlet</title>
</head>
<body>

  Main thread: ${mainThread}

   <br/>

   Worker thread: ${workerThread}
</body>

</html>
```

AsyncContext Class 6-10

- Figure shows the name of the main thread and the name of the worker thread.



AsyncContext Class 7-10

- The code snippet develops an asynchronous Servlet that sends progress updates to HTML page after every second.

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.AsyncContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class AsyncCompleteServlet extends HttpServlet {
  @Override
  public void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {

  response.setContentType("text/html");

  PrintWriter writer = response.getWriter();

  writer.println("<html><head><title>" + "Async Servlet</
title></head>");

 writer.println("<body><div id='progress'></div>");

 final AsyncContext = request.startAsync();

 asyncContext.setTimeout(60000);
```

AsyncContext Class 8-10

Contd.

```
    asyncContext.start(new Runnable() {
    @Override

    public void run() {

    System.out.println("new thread:" + Thread.currentThread());

    for (int i = 0; i < 10; i++) {
              writer.println("<script>");

              writer.println("document.getElementById(" +
              "'progress').innerHTML = '" + i * 10) + "% complete'");

              writer.println("</script>");

              w r i t e r . fl u s h ( ) ;
     try {
         Thread.sleep(1000);
     } catch (InterruptedException e) {. . .}
    }

    writer.println("<script>");
    writer.println("document.getElementById(" +
"'progress').innerHTML = 'DONE'");

    writer.println("</script>"); writer.println("</body></html>");
    asyncContext.complete();
   }});
  }
}
```

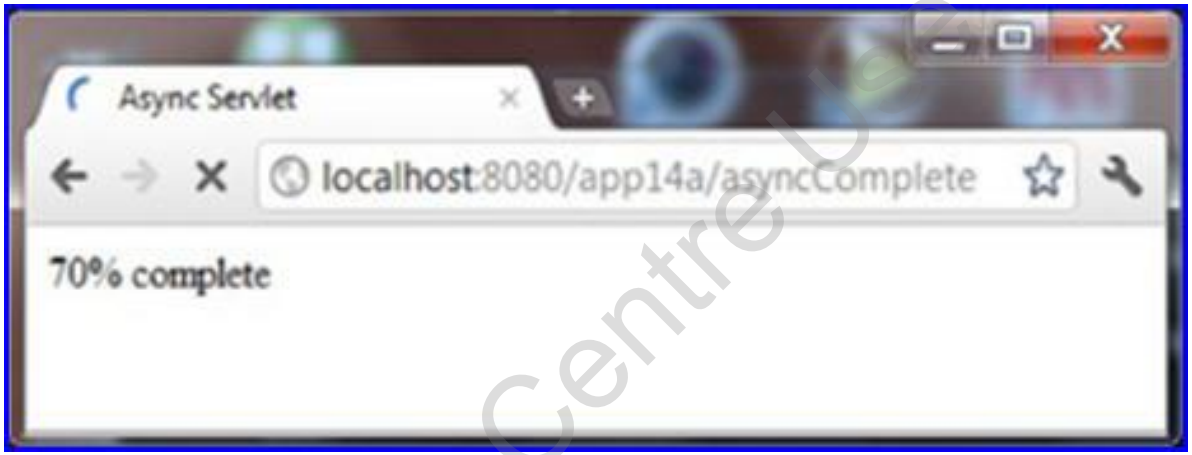
AsyncContext Class 9-10

- The code snippet displays the deployment descriptor, `web.xml` for configuring the `AsyncCompleteServlet`.

```
<web-app>
. . .
  <servlet>
   <servlet-name>AsyncComplete</servlet-name>
   <servlet-
   class>servlet.AsyncCompleteServlet</servlet-
   class>
   <async-supported>true</async-supported>
  </servlet>

  <servlet-mapping>
   <servlet-name>AsyncComplete</servlet-name>

   <url-pattern>/asyncComplete</url-pattern>

  </servlet-mapping>

</web-app>
```

AsyncContext Class 10-10

- Figure shows the result of HTML page that receives progress updates from the asynchronous Servlet.



AsyncListener Interface 1-4

- The `AsyncListener` interface defines the methods that are used to notify the events occurring on the `AsyncContext` object.
- Table lists the methods of the `AsyncListener` interface.

Method	Description
`void onStartAsync (AsyncEvent event)`	It gets called when an asynchronous operation has been initiated.
`void onComplete (AsyncEvent event)`	It gets called when an asynchronous operation has completed.
`void onError (AsyncEvent event)`	It gets called in the event an asynchronous operation has failed.
`void onTimeout (AsyncEvent event)`	It gets called when an asynchronous has timed out.

AsyncListener Interface 2-4

- The code snippet demonstrates the code that `AsyncListenerServlet` class that registers the listener to receive event notifications.

```
@WebServlet(name = "AsyncListenerServlet", urlPatterns =
{ "/ asyncListener" }, asyncSupported = true)
public class AsyncListenerServlet extends HttpServlet {

private static final long serialVersionUID = 62738L;
@Override

public void doGet(final HttpServletRequest request,
HttpServletResponse response)throws ServletException,
IOException {

// Obtaining the Asynchronous context object
 final AsyncContext = request.startAsync();

//Setting the time out duration for asyncContext object
 asyncContext.setTimeout(5000);

// Registering the events
 asyncContext.addListener(new MyAsyncListener());
```

AsyncListener Interface 3-4

```
//Obtaining a new thread
 asyncContext.start(new Runnable() {
  @Override
    public void run() {
    try {
            Thread.sleep(3000);
        } catch (InterruptedException e) { ...}
    String greeting = "hi from listener";
    System.out.println("wait….");
    request.setAttribute("greeting", greeting);
    asyncContext.dispatch("/test.jsp");
   }});
  }
}
```

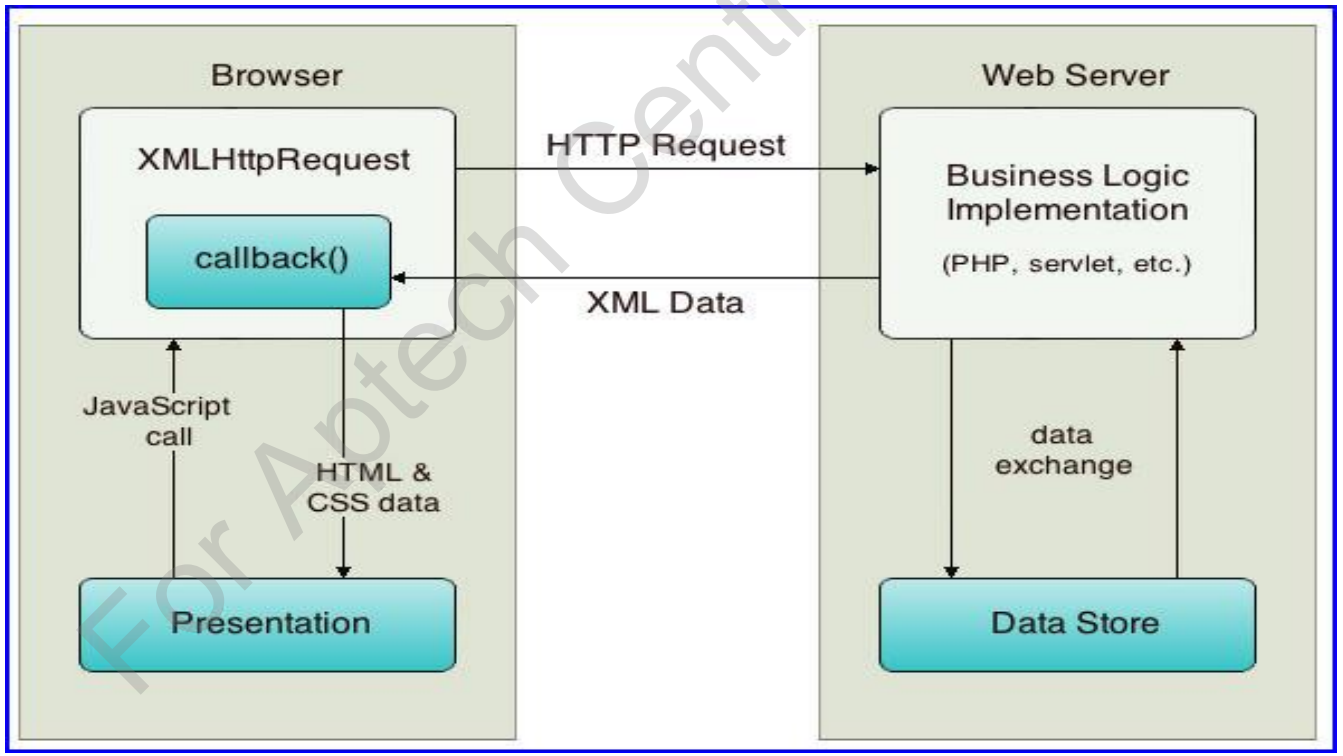
AsyncListener Interface 4-4

- The code snippet shows the implementation of the `AsyncListener` interface methods.

```
public class MyAsyncListener implements AsyncListener {

  @Override
  public void onComplete(AsyncEvent asyncEvent)
    throws IOException {
     System.out.println("onComplete");
 }

  @Override
  public void onError(AsyncEvent asyncEvent)
     throws IOException {
      System.out.println("onError");
 }

  @Override
  public void onStartAsync(AsyncEvent asyncEvent)
    throws IOException {
    System.out.println("onStartAsync");
 }

  @Override
  public void onTimeout(AsyncEvent asyncEvent)
    throws IOException {
     System.out.println("onTimeout");
 }
}
```

Server Push Mechanism in Servlet

- The mechanism of sending the data asynchronously from the Web server to the client, without redrawing the whole page is referred as **server push**.
- The most popular programming approaches used in modern Web applications is **Asynchronous Java and XML (AJAX)**.
- Figure shows the asynchronous communication of Web client and Web server based on AJAX mechanism in the Web application.



Developing Asynchronous Client 1-5

- The `XMLHttpRequest` object is used in the JavaScript code to interact asynchronously with the Web server.
- The steps for creating the asynchronous JavaScript client are as follows:

Create an `XMLHttpRequest` object.

Send the request to the server using the methods of the `XMLHttpRequest` object.

Receive and process the response received from the server.

Developing Asynchronous Client 2-5

- The code snippet demonstrates how to check the support for `XMLHttpRequest` object in the HTML page and process the request and response using the `XMLHttpRequest` object.

```
<HTML>
<HEAD>
<SCRIPT TYPE="text/javascript">
  function getXMLHttpRequest() {
   var xmlHttpReq = false;

   // Creating XMLHttpRequest object for non-Microsoft browsers
   if (window.XMLHttpRequest) {

          xmlHttpReq = new XMLHttpRequest();

     } else if (window.ActiveXObject) {

    try {

   // Creating XMLHttpRequest object in later versions
   // of Internet Explorer

         xmlHttpReq = new ActiveXObject("Msxml2.XMLHTTP");
        } catch (exp1) {
```

Developing Asynchronous Client 3-5

```
  try {
        // to create XMLHttpRequest object in older versions
        // of Internet Explorer
        xmlHttpReq = new ActiveXObject("Microsoft.XMLHTTP");
      } catch (exp2) {
         xmlHttpReq = false;     } //end of inner-catch
  }
}
      return xmlHttpReq;
}

/* AJAX call starts with this function*/
function makeRequest() {
  var xmlHttpRequest = getXMLHttpRequest();
xmlHttpRequest.onreadystatechange =
        getReadyStateHandler(xmlHttpRequest);
    xmlHttpRequest.open("GET", "helloWorld.do", true);
    xmlHttpRequest.send();
}
```

Contd.

Developing Asynchronous Client 4-5

```
/* Returns a function that waits for the state change in
XMLHttpRequest*/
function getReadyStateHandler(xmlHttpRequest) {


    // an anonymous function returned
    // it listens to the XMLHttpRequest instance
    return function() {
      if (xmlHttpRequest.readyState == 4) {
       if (xmlHttpRequest.status == 200) {
            document.getElementById("hello").innerHTML =
            xmlHttpRequest.responseText;
        } else {
            alert("HTTP error " + xmlHttpRequest.status + ": "
+ xmlHttpRequest.statusText);
        }
    }
  };
}
```

Developing Asynchronous Client 5-5

- To send the request, the two methods namely, `open()` and `send()` of `XMLHttpRequest` object are invoked.
- These methods are used to prepare the HTTP Request and send it to the server.
- **Syntax:**

```
open(method,url,async)
```

where,

- `method`: Specifies the type of request method, that is, GET or POST is used to send the request.
- `url`: Specifies the location of the component handling the request on the server.
- `async`: Specifies whether the asynchronous communication is supported or not. The value can be true or false.

Implementation of Non-blocking I/O Support 1-3

- The non-blocking I/O is supported by:
  - `javax.servlet.ServletInputStream`
  - `javax.servlet.ServletOutputStream`
  - `javax.servlet.ReadListener`
  - `javax.servlet.writeListener`

- The steps used for non-blocking I/O to process requests and writing responses inside service methods are as follows:
  - Enable the asynchronous mode for the Servlet using the `@WebServlet` annotation.
  - Obtain an input stream and/or an output stream from the request and response objects in the `service()` method.
  - Assign a read listener to the input stream and/or a write listener to the output stream.
  - Process the request and the response inside the listener's callback methods.

Implementation of Non-blocking I/O Support 2-3

- The code snippet demonstrates the use of non-blocking I/O in the asynchronous Servlet.

```
@WebServlet(urlPatterns={"/asyncioservlet"},
asyncSupported=true)
public class FirstServlet extends HttpServlet{
    @Override
    public void doPost(HttpServletRequest request,
HttpServletResponse response)throws IOException {
final AsyncContext acontextResponse = request.startAsync();
final ServletInputStream inputStream = request.
getInputStream();
 inputStream.setReadListener(new ReadListener() {
        byte buffer[] = new byte[4*1024];
       StringBuilder sbuilder = new StringBuilder();
       public void onDataAvailable() {
```

Implementation of Non-blocking I/O Support 3-3

```
        try {
        do {
                int length = inputStream.read(buffer);
                sbuilder.append(new String(buffer, 0, length));
          } while(inputStream.isReady());
         } catch (IOException ex) { ex.printStackTrace(); }
         }
         public void onAllDataRead() {
          try {
                acontextResponse.getResponse().getWriter().write("...the
response...");
             } catch (IOException ex) { ex.printStackTrace(); }
                    acontextResponse.complete();
         }
        });
      }
}
```

Protocol Upgrade Processing

- With Servlet 3.1 API, the HTTP protocol has been upgraded from HTTP 1.0 to HTTP 1.1.
- HTTP 1.0 was a stateless protocol.
- HTTP 1.1 can persist the connection between the client and the server.
  - The advantage of persistent connection is that the connection is alive and can be used for multiple requests with the client.

Upgrading to HTTP 1.1 Protocol 1-2

- In HTTP/1.1, on a current connection, clients can switch to a different protocol using the Upgrade header field.
- The code snippet demonstrates the use of headers to upgrade the protocol information in the response.

```
@WebServlet(urlPatterns={"/ABCDresource"})
public class ABCDUpgradeServlet extends HttpServlet {

 @Override
 public void doGet(HttpServletRequest request,
    HttpServletResponse response) {

  if ("ABCD".equals(request.getHeader("Upgrade"))) {

          /* Accept upgrade request */
           response.setStatus(101);
           response.setHeader("Upgrade", "ABCD");
    } else { /* ... write error response ... */ }
 }
}
```

Upgrading to HTTP 1.1 Protocol 2-2

- The code snippet shows the implementation of `HttpUpgradeHandler`.

```
package com.authentication;
import javax.servlet.ServletInputStream;
import javax.servlet.ServletOutputStream;

public class ABCDUpgradeHandler implements
HttpUpgradeHandler {

 public XYZPUpgradeHandler upgrade(XYZPUpgradeHandler
                                          protocol){
    return protocol;
 }

    public void destroy() { }
}
```

Summary

- The architecture of the Servlet is based on multithreaded model. In this model, the Web container normally creates a pool of threads ready to serve the client with the Servlet instance.
- The Java Servlet API provides asynchronous processing support for servlets and filters in the Web applications.
- In asynchronous processing, a Servlet thread waiting for a resource is released by the container and returned to the pool.
- The class javax.servlet.AsyncContext is used to process the request asynchronously within the Servlet.
- To handle these events, the AsyncContext object can be registered with the AsyncListener interface.
- The mechanism of sending the data asynchronously from the Web server to the client, without redrawing the whole page is referred as server push.
- One of the most popular programming approach used in modern Web applications to push the server data to the client is performed using AJAX mechanism.
- The AJAX mechanism works with an XMLHttpRequest object that is used to pass the requests and responses asynchronously between the client and server.
- Servlet API provides non-blocking I/O support for Servlets and filters to process input and output asynchronously in the request.
- With Servlet 3.1 API, the HTTP protocol has been upgraded from HTTP 1.0 to HTTP 1.1. HTTP 1.0 was a stateless protocol, whereas HTTP 1.1 can persist the connection between the client and the server.