Asynchronous Servlet Feature of Servlet 3

Before we jump into understanding what Async Servlet is, let's try to understand why do we need it. Let's say we have a Servlet that takes a lot of time to process, something like below.
LongRunningServlet.java

```
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/LongRunningServlet")
public class LongRunningServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
        long startTime = System.currentTimeMillis();
        System.out.println("LongRunningServlet Start::Name="
                + Thread.currentThread().getName() + "::ID="
                + Thread.currentThread().getId());

        String time = request.getParameter("time");
        int secs = Integer.valueOf(time);
        // max 10 seconds
        if (secs > 10000)
            secs = 10000;

        longProcessing(secs);

        PrintWriter out = response.getWriter();
        long endTime = System.currentTimeMillis();
        out.write("Processing done for " + secs + " milliseconds!!");
        System.out.println("LongRunningServlet Start::Name="
                + Thread.currentThread().getName() + "::ID="
                + Thread.currentThread().getId() + "::Time Taken="
                + (endTime - startTime) + " ms.");
    }

    private void longProcessing(int secs) {
        // wait for given time before finishing
        try {
            Thread.sleep(secs);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

}
```

If we hit above servlet through browser with URL as http://localhost:8080/AsyncServletExample/LongRunningServlet?time=8000, we get response as "Processing done for 8000 milliseconds!!" after 8 seconds. Now if you will look into server logs, you will get following log:

```
LongRunningServlet Start::Name=http-bio-8080-exec-34::ID=103
LongRunningServlet Start::Name=http-bio-8080-exec-34::ID=103::Time Taken=8002 ms.
```

So our Servlet Thread was running for ~8+ seconds, although most of the processing has nothing to do with the servlet request or response.
Prior to Servlet 3.0, there were container specific solution for these long running threads where we can spawn a separate worker thread to do the heavy task and then return the response to client. The servlet thread returns to the servlet pool after starting the worker thread. Tomcat's Comet, WebLogic's FutureResponseServlet and WebSphere's Asynchronous Request Dispatcher are some of the example of implementation of asynchronous processing.
The problem with container specific solution is that we can't move to other servlet container without changing our application code, that's why Async Servlet support was added in Servlet 3.0 to provide standard way for asynchronous processing in servlets.

- **Asynchronous Servlet Implementation**

1. First of all the servlet where we want to provide async support should have **@WebServlet** annotation with **asyncSupported** value as **true**.
2. Since the actual work is to be delegated to another thread, we should have a thread pool implementation. We can create thread pool using Executors framework and use servlet context listener to initiate the thread pool.
3. We need to get instance of **AsyncContext** through ServletRequest.startAsync() method. AsyncContext provides methods to get the ServletRequest and ServletResponse object references. It also provides method to forward the request to another resource using dispatch() method.
4. We should have a Runnable implementation where we will do the heavy processing and then use AsyncContext object to either dispatch the request to another resource or write response using ServletResponse object. Once the processing is finished, we should call AsyncContext.complete() method to let container know that async processing is finished.
5. We can add AsyncListener implementation to the AsyncContext object to implement callback methods – we can use this to provide error response to client incase of error or timeout while async thread processing. We can also do some cleanup activity here.