

Instructor Notes:

Java Servlets 3.0

Appendices

Instructor Notes:**Appendix A: XML Mappings in Servlets 3.0****Basic Servlet Mappings:**

If servlets need to be created without annotations, XML mappings can be used as below:

This mapping would create a servlet with name as Hello—servlet name is a logical name (label). Servlet class is HelloServlet—in package com.igate. Thus a container can create a servlet.

```
<web-app> : Root tag; There would be XSD mappings going with the root tag
<servlet>
  <servlet-name>Hello</servlet-name>
  <servlet-class>com.igate.HelloServlet</servlet-class>
</servlet>
</web-app>
```

This mapping would bind the servlet to a particular URL pattern. Using this URL Pattern request would be given in the browser: URL pattern is preceded by '/'

```
<servlet-mapping>
  <servlet-name>Hello</servlet-name>
  <url-pattern>/Hello</url-pattern>
</servlet-mapping>
```

Servlets with Init Parameters:

Servlet can also have initialization parameters and contextual parameters which can be fetched via ServletConfig and ServletContext objects respectively.

Servlet initialization parameters are a part of servlet and cannot be shared among other servlet; whereas contextual parameters are shared by entire web application:

A servlet can have multiple initialization parameters.

```
<servlet>
  <servlet-name>Hello</servlet-name>
  <servlet-class>com.igate.HelloServlet</servlet-class>
  <init-param>
    <param-name>flower</param-name>
    <param-value>rose</param-value>
  </init-param>
</servlet>
```

Instructor Notes:

Context-parameter can be done as follows: This tag should be outside <servlet-mapping> and <servlet> tags

Multiple context-parameters can be set.

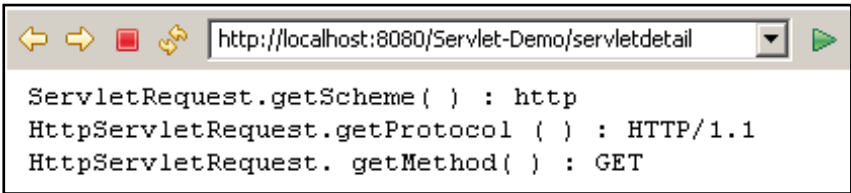
```
<context-param>
    <param-name>email</param-name>
    <param-value>info@test.com</param-value>
</context-param>
```

Mappings for welcome files:

If we deploy the web application, the welcome file is expected to execute. (i.e. the first page to execute when the application runs). This can be set through welcome file list tag. It checks for first file in listing -> index.html if not found then it will search for index.htm and so on. If no welcome file is found then HTTP 404 error is raised. See below listing for same.

```
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
</welcome-file-list>
```

Request URL	Its URI path	Its Servlet path
http://server:port/context/classname	context/classname	context/classname
http://server:port/context/classname?var=val	context/classname	context/classname
http://server:port/context/classname/pathInfo	context/classname/pathInfo	context/classname



Instructor Notes:

Appendix B : Request Object

Idempotent Methods:

GET is idempotent method as GET is NOT supposed to update data, but to get data, although we can(and we do) implement GET to update data, by posting data using query string. So, GET may sometimes behave non – idempotent manner.

POST method is considered non-idempotent. The reason is that the POST is supposed to post data to the server, which means it may be used to save the data in persistent location OR to do some action based on that posted data.

Determining What is Requested:

A Servlet can find out what field or servlet was requested using several methods. There exists no single method to find the original URL used by client to make the request.

- **public static StringBuffer HttpUtils.getRequestURL(HttpServletRequest req):** This method reconstructs the request URL based on information available in the HttpServletRequest object. The StringBuffer result contains the scheme, server name, server port, and extra path information.
- **public String HttpServletRequest.getRequestURI():** This method returns the URI (Universal Resource Identifier). This is the context path plus the servlet path plus any extra path information. A URI can be thought of as a URL minus the scheme, host, port, and query string, but including extra path information.
- **public String HttpServletRequest.getServletPath():** This method returns the part of the URI that refers to the servlet being invoked or null if the URI does not directly point to a servlet. The servlet path does not include extra path information.

For some examples, see the table given below.

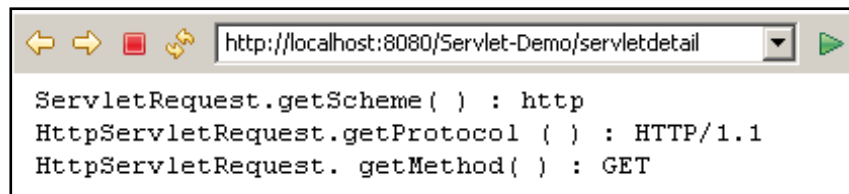
Request URL	Its URI path	Its Servlet path
http://server:port/context/classname	context/classname	context/classname
http://server:port/context/classname?var=val	context/classname	context/classname
http://server:port/context/classname/pathInfo	context/classname/pathInfo	context/classname

Instructor Notes:**Determining How it was Requested:**

Apart from knowing what was requested, a servlet can also find out how it was requested. The methods below provide this information.

- `public String ServletRequest.getScheme()` : It returns the name of the scheme used to make this request, for example, http, https, or ftp.
- `public String HttpServletRequest.getProtocol()` : It returns the name and version of the protocol the request uses.
- `public String HttpServletRequest.getMethod()` : It returns the name of the HTTP method with which this request was made like GET, POST, HEAD, etc.

For an example, see the figure below.

**Appendix B : Request Object****Methods for Using Input Stream:**

We have seen in our examples how every **response** has a **output stream** associated with it. Similarly, each **request** handled by a servlet has an **input stream** associated with it. The servlet can read from the **Reader** or **InputStream** associated with its request object. The data read from the input stream can be of any content type and of any length. The input stream has three purposes:

- To pass the response body from one servlet to another in a servlet chain
- To pass an HTTP servlet the content associated with a POST request
- To pass a non-HTTP servlet raw data by the client

There are several methods that the request object uses to read data:

- **BufferedReader ServletRequest.getReader()**: It retrieves the body of the request as character data using a `BufferedReader`. This method throws an `IllegalStateException` if `getInputStream()` has been called before on the same request and an `UnsupportedEncodingException` if the character encoding of the input is unsupported or unknown.
- **ServletInputStream ServletRequest.getInputStream()**: It retrieves the body of the request as binary data using a `ServletInputStream`. The method throws an `IllegalStateException` if `getReader()` has been called before on this same request. Once you have the `ServletInputStream`, you can read a line from it using `readLine()`.
- **String ServletRequest.getContentType()** and **int ServletRequest.getContentLength()**: It checks the content type and the length of the data being sent via the input stream.

Instructor Notes:**Appendix C : Response Object****Sending Compressed Data Using the “Accept-Encoding” Header :**

Browsers that support content encoding feature, indicate that they do so by setting the ‘Accept-Encoding’ request header.

Servlets can check this header and send compressed response to clients that support encoding and send a regular web page to those that don’t. For example, let us assume that client accepts **gzip** encoding. Implementing compression is straightforward since gzip is built into the Java programming language via

java.util.zip.

The servlet first checks the **Accept-Encoding** header to see if it contains an entry for **gzip**.

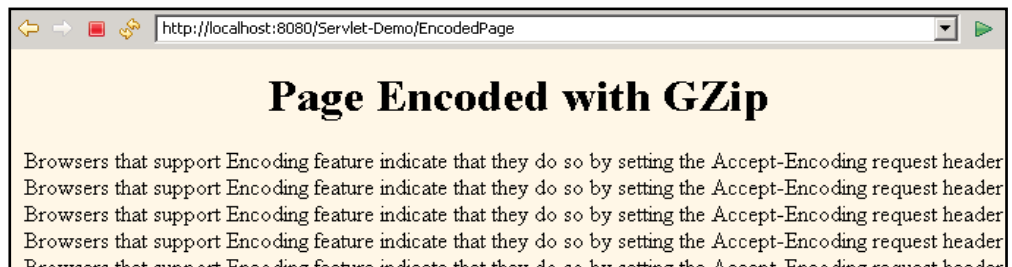
- If so, it uses a **GZIPOutputStream** to generate the page, specifying **gzip** as the value of the **Content-Encoding** header. You must explicitly call **close** when using a **GZIPOutputStream**.

- If **gzip** is not supported, then the servlet uses the normal **PrintWriter** to send the page.

Note: GZIP is often used while loading JavaScript files to reduce the time for loading. Refer com.igate.ch4.EncodedPage.java class. A partial listing is shown below:

```
public void doGet(HttpServletRequest req, HttpServletResponse resp){
    //check which coding style browser accepts
    String encode = req.getHeader("Accept-Encoding");
    PrintWriter out; String title="Page Encoded with Gzip";
    if ((encode != null) && (encode.indexOf("gzip") != -1)){
        OutputStream out1 = resp.getOutputStream();
        out = new PrintWriter(new GZIPOutputStream(out1), false);
        resp.setHeader("Content-Encoding", "gzip");
    } else {out = resp.getWriter(); }
    out.println("<HTML><BODY><H1 ALIGN=CENTER>" + title +
"</H1>\n");
    String line = "Browsers that support Encoding feature indicate that "
        + "they do so by setting the Accept-Encoding request header";
    for (int i = 0; i < 10000; i++) out.println(line);
    out.println("</BODY></HTML>"); out.close();
}
```

Output:



Instructor Notes:**Appendix C : Response Object****Restricting Access to Web Pages:**

- Many web servers support standard mechanisms for limiting access to designated Web pages. These mechanisms can apply to static pages as well as those generated by servlets, so many authors use their server specific mechanism for restricting access to servlets. Furthermore, most users at e-commerce sites prefer to use regular HTML forms to provide authorization information since these forms are more familiar, can provide more explanatory information, and can ask for additional information beyond just a user name and password. Once a servlet that uses form-based access grants initial access to a user, it would use session tracking to give the user access to other pages that require the same level of authorization.
- Nevertheless, form-based access control requires more effort on the part of the servlet developer, and HTTP-based authorization is sufficient for many simple applications. Here, the servlet uses status codes and HTTP headers to manage its own basic authentication policy. It receives encoded user credentials in Authorization header. If it chooses to deny those credentials, it does so by sending SC_UNAUTHORIZED status code and a WWW_Authenticate header that describes the desired credentials. The Authorization header, if sent by the client, contains the client's username and password ("username:password") encoded in Base64. It also tells the client the authorization scheme and the realm against which the users will be verified. (*A realm is just a collection of user accounts and protected resources*). For example: To tell the client to use basic authentication for the realm Admin, the WWW_Authenticate header is:
 - WWW_Authenticate: BASIC realm="Admin"

```

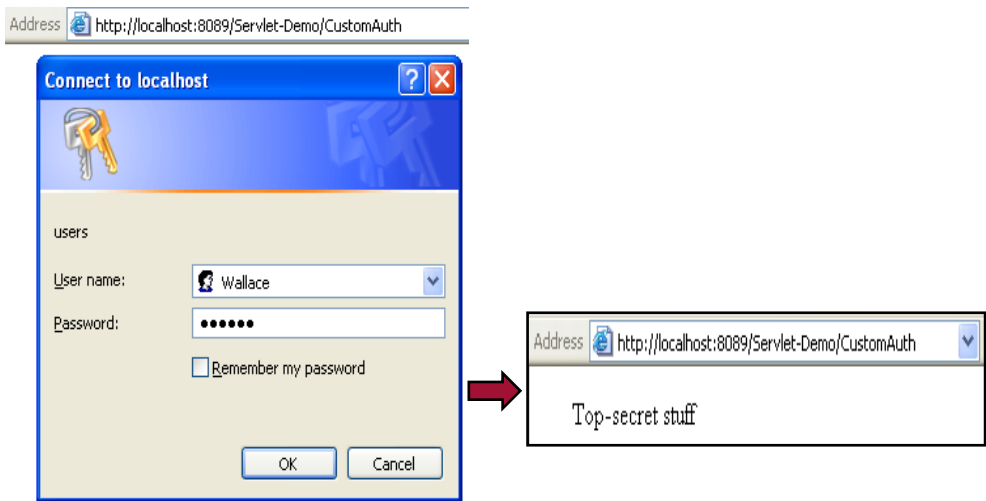
public void init(ServletConfig config) {
    users.put("Wallace:cheese", "allowed"); //Hashtable entry
    //other hardcoded entries ... .. }
    public void doGet(HttpServletRequest req, HttpServletResponse
res) {
        String auth = req.getHeader("Authorization"); // Get
Authorization header
        if (!allowUser(auth)) {
            res.setHeader("WWW-Authenticate", "BASIC
realm=\"users\"");
            res.sendError(res.SC_UNAUTHORIZED);
        } else out.println("Top-secret stuff");
    }
    protected boolean allowUser(String auth) throws IOException {
        if (auth == null) return false; // no auth
        if (!auth.toUpperCase().startsWith("BASIC ")) return false;
        String userpassEncoded = auth.substring(6);
        sun.misc.BASE64Decoder dec = new
sun.misc.BASE64Decoder();
        String userpassDecoded = new
String(dec.decodeBuffer(userpassEncoded));
        if ("allowed".equals(users.get(userpassDecoded))) return true;
        else return false;
    }
}

```

Instructor Notes:

Appendix C : Response Object
Restricting Access to Web Pages:

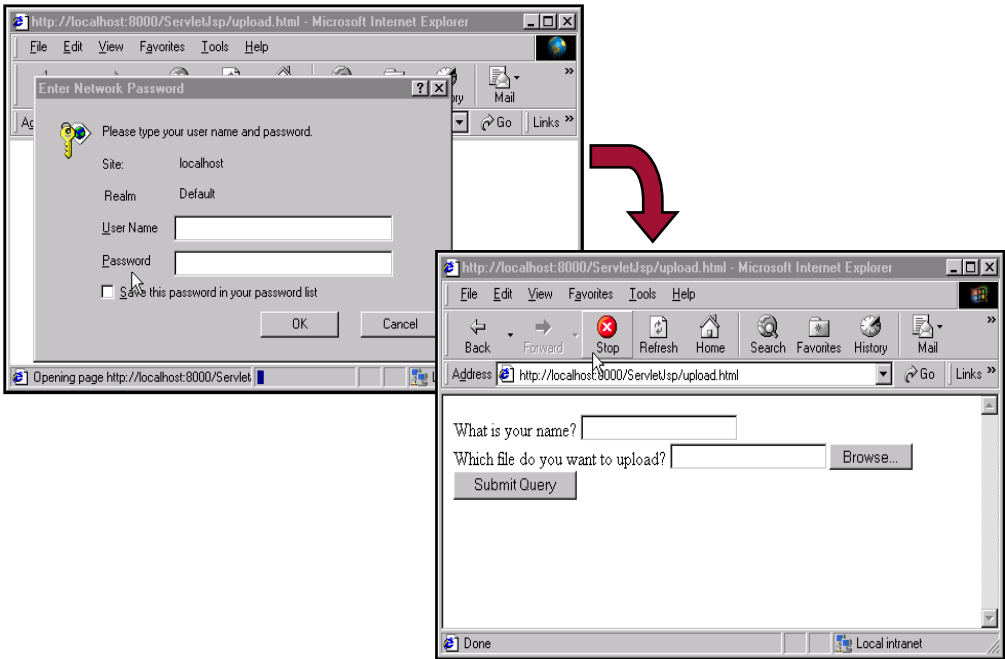
- The previous example shows a servlet that performs custom authorization, receiving an Authorization header and sending the SC_UNAUTHORIZED status code and WWW-Authenticate header when necessary. The servlet restricts access to its “top-secret stuff” to those users (and passwords) it recognizes in its user list. For this example, the list is kept in a simple **Hashtable** and its contents are hard-coded. Of course, for a production servlet data will be retrieved from an external media like database.
- To retrieve the Base64-encoded username and password, the servlet needs to use a Base64 decoder. Fortunately, there are several freely available decoders. For this servlet, we have chosen to use the sun.misc.BASE64Decoder class that accompanies the JDK, so it is probably already on your system.
- Although the web server is asked to grant any client access to this servlet, the servlet sends its “top-secret” output only to those users it recognizes.
- The **doGet()** method checks if there is an Authorization header. If there is no such header, it returns a 401 (Unauthorized) response code and a header of the following form: WWW-Authenticate: BASIC realm=”name”
- With this response, the browser pops up a dialog box telling the user to enter a name and password for *name*, then to reconnect with that username and password embedded in a single base64 string inside the Authorization header.
- If there is a Authorization header skip over the word “basic” and reverse the base64 encoding of the remaining part. This results in a string of the form username:password. Check the username and password, against some stored set. If it matches, return the page.
- Refer the com.igatepatni.ch4.CustomAuth.java class. Invoke this servlet as follows:
 - `http://localhost:8080/Servlet-Demo/CustomAuth`
- When you access this servlet it asks for username and password as shown in the first figure above. Type username as “Wallace” and password as “cheese” to display the servlet output as shown in the second figure in the above slide.



Instructor Notes:

Appendix C : Response Object
Protecting Pages by configuring Server:

- You can protect pages by configuring server. Please refer to server manual at your location to understand how new users may be added to an existing list of users.
- Invoke the protected page as shown in the first figure below. The browser asks the user to enter username and password. This page is valid only for users who have entry in the realm. In this example, assuming upload.html is protected for authorized users, it is shown only after providing valid username and password. We shall see in the SSL session how server can be configured to restrict access to pages.



Instructor Notes:**Appendix D :More on HttpSession API****Session Binding Events**

Some objects may wish to perform an action when they are bound or unbound from a session. For example, a database connection may begin a transaction when bound to a session and end the transaction when unbound. Any object that implements the `javax.servlet.http.HttpSessionBindingListener` interface is notified when it is bound or unbound from a session. The interface is:

```
public abstract interface HttpSessionBindingListener
extends java.util.EventListener {
    public void valueBound(HttpSessionBindingEvent event);
    public void valueUnbound(HttpSessionBindingEvent event);
}
```

The `valueBound()` method is called when the listener is bound into a session, and `valueUnbound()` is called when the listener is unbound from a session.

The `HttpSessionBindingEvent` constructor argument provide access to the name under which the object is being bound(or unbound) with `getName()` method. `getSession()` method returns the session object.

public HttpSessionBindingEvent(HttpSession session,String name) : It constructs an event that notifies an object that it has been bound to or unbound from a session. To receive the event, the object must implement `HttpSessionBindingListener`.

public java.lang.String getName() : It returns the name with which the object is bound to or unbound from the session.

Refer to `SessionBindings` servlet. This servlet demonstrate the use of `HttpSessionBindingListener` and `HttpSessionBindingEvent` with listener that logs when it is bound and unbound from a session.

```
public class SessionBindings extends HttpServlet {
    public void doGet(HttpServletRequest req,
                      HttpServletResponse res)
                      throws ServletException,
                      IOException {
        res.setContentType("text/plain");
        PrintWriter out = res.getWriter();
        // Get the current session object, create one if
        necessary
        HttpSession session = req.getSession(true);
        // Add a CustomBindingListener
        session.setAttribute("bindings.listener",
                             new
                             com.igatepatni.ch5.CustomBindingListener(
                             getServletContext()));
        out.println("This page intentionally left blank");
    }
}
```

Instructor Notes:**Appendix D :More on HttpSession API**
Session Binding Events

```

class CustomBindingListener implements
    HttpSessionBindingListener {
    // Save a ServletContext to be used for its log() method
    ServletContext context;
    public CustomBindingListener(ServletContext context) {
        this.context = context;
    }
    public void valueBound(HttpSessionBindingEvent event) {
        context.log("BOUND as " + event.getName() +
                    " to " +
                    event.getSession().getId());
    }
    public void valueUnbound(HttpSessionBindingEvent event) {
        context.log("UNBOUND as " + event.getName() +
                    " from " + event.getSession().getId());
    }
}

```

- Each time a CustomBindingListener object is bound to a session, its valueBound() method is called and the event is logged. Each time it is unbound from a session, its valueUnbound() method is called so that event too is logged. We can observe the sequence of events by looking at the server's event log.
- Let us assume that this servlet is called once, reloaded 30 seconds later, and not called again for at least a half hour. The event log would look something like this:

```

[Tue Jan 27 01:46:48 PST 1998]
  BOUND as bindings.listener to INWBUJIAAAAAHQDGPM5QAAA
[Tue Jan 27 01:47:18 PST 1998]
  UNBOUND as bindings.listener from INWBUJIAAAAAHQDGPM5QAAA
[Tue Jan 27 01:47:18 PST 1998]
  BOUND as bindings.listener to INWBUJIAAAAAHQDGPM5QAAA
[Tue Jan 27 02:17:18 PST 1998]
  UNBOUND as bindings.listener from INWBUJIAAAAAHQDGPM5QAAA

```

Instructor Notes:**Appendix E : Inter-Servlet Communication:****Additional Request-scoped Attributes:**

While doing include or forward by default some attributes will get populated. They are:

javax.servlet.include.request_uri : It is the part of this request's URL from the protocol name up to the query string. For example: /cart/products.html.

javax.servlet.include.servlet_path : It is the part of this request's URL that calls the servlet. This path starts with a "/" character and includes either the servlet name or a path to the servlet, but does not include any extra path information or a query string.

javax.servlet.include.context_path : It is the portion of the request URI that indicates the context of the request. The context path always comes first in a request URI. The path starts with a "/" character but does not end with a "/".

javax.servlet.include.path_info : It is any extra path information associated with the URL the client sent when it made this request. The extra path information follows the servlet path but precedes the query string and will start with a "/" character.

javax.servlet.include.query_string : It is the query string that is contained in the request URL after the path.

A servlet that has been invoked by another servlet using the **include()** method of `RequestDispatcher` has access to the path by which it was invoked. The request attributes discussed above must be set. These attributes are accessible from the included servlet via the `getAttribute()` method on the request object and their values must be equal to the request URI, context path, servlet path, path info, and query string of the INCLUDED servlet, respectively. If the request is subsequently included, these attributes are replaced for that include. Eg:

```
request.getAttribute("javax.servlet.include.request_uri")
```

On the other hand, a servlet that has been invoked by another servlet using the **forward()** method of `RequestDispatcher` has access to the path of the ORIGINAL request. The request attributes shown below must be set.

- `javax.servlet.forward.request_uri`
- `javax.servlet.forward.servlet_path`
- `javax.servlet.forward.path_info`
- `javax.servlet.forward.query_string`

These attributes are accessible from the forwarded servlet via the `getAttribute` method on the request object. Note that these attributes must always reflect the information in the original request even under the situation that multiple forwards and subsequent includes are called.

If the included or forwarded servlet was obtained by using the `getNamedDispatcher()` method, these attributes MUST NOT be set.

Instructor Notes:**Appendix E : Inter-Servlet Communication:****Accessing Passive Server resources:**

Passive server resources (for example: static HTML pages which are stored in local files) cannot be accessed with `RequestDispatcher` objects. The `ServletContext` method `getResource(String path)` returns a `URL` object for a resource specified by a local URI (for example: `"/` for the server's document root) which can be used to examine the resource.

If you only want to read the resource's body you can directly ask the `ServletContext` for an `InputStream` with the `getResourceAsStream ()` method.

A Servlet may need to access additional resources like configuration files whose locations should not need to be specified in init parameters. Those resources can be accessed with the methods `getResource(String name)` and `getResourceAsStream(String name)` of the `java.lang.Class` object which represents the Servlet's class.

Example. The following code gets an `InputStream` for a configuration file named `mycfg.cfg` which resides in the same directory as the class in which the code is executed:

Note that the servlet class loader must implement the `getResource()` and `getResourceAsStream()` methods in order for this to work. This may not be the case with all Servlet engines.

```
InputStream is =  
getClass().getResourceAsStream("mycfg.cfg");
```

Instructor Notes:**Appendix G : Asynchronous Servlet:**

Asynchronous servlet:

Using Asynchronous processing avoids blocking requests by allowing the thread to perform some other operation, meanwhile the input is returned to the client.

Asynchronous Servlet Implementation

First of all the servlet where we want to provide async support should have @WebServlet annotation with asyncSupported value as true.

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.

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.

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.

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.

Consider servlet with following code:

```
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();
    }
}
```

Instructor Notes:

If we hit above servlet through browser with URL as `http://localhost:9090/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.
```

This can leads to Thread Starvation – Since our servlet thread is blocked until all the processing is done, if server gets a lot of requests to process, it will hit the maximum servlet thread limit and further requests will get Connection Refused errors.

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.

Instructor Notes:**Appendix H : SSL and WildFly:**

Application security can be applied in three areas:-

Authentication – A process by which system verifies the identity of the user

Authorization – A process of verifying that the user has the required credentials to access the applications resources

Secure Communication- To ensure Confidentiality, Data integrity and Source Integrity of the Data. The Secure Protocols are:-

Transport Layer Security (TLS) Secure Socket Layer (SSL)

Understanding Application Security:

Security is an important aspect of most enterprise applications because loss of sensitive data and system susceptibilities can be costly. Unauthorized users may access application data or someone may intercept a message being transmitted or hack into your application and run malicious code, as we see Security can be compromised in multiple ways.

Application Security can be applied in three areas. Let us see each one.

Authentication: It is a process by which system verifies the identity of the user.

Authorization: It is a process of verifying that the user has the required credentials/privileges to access applications resources.

To understand this let us consider an simple example, in an application the user is prompted to enter username & password. This username & password is verified against some predefined values either in the database or some file. This is termed as authenticating the user. Once it is determined that user is valid, the application can then check for privileges that user has based on components requested by the user.

Apart from these two methods application security can also be applied by secure communication.

Secure Communication: Data is often sent over open network. In such cases we should be aware of confidentiality, data integrity and source integrity.

Confidentiality protects a message from being read by any third person other than the intended recipient. Often the message is encrypted so that the message contents are not manipulated, this is Data Integrity. Source integrity guarantees the identity of the sender, which means the message was indeed sent by the sender.

Source integrity can be protected by using a reliable certificate issuing authority.

A person who wants to be trusted by other people can obtain a certificate and send it to other parties who can verify the certificate and send it to other parties who can verify the certificate owner's identity by asking the trusted third party which is normally the certificate issuing authority

Secure protocols such as Transport Layer Security (TLS) or Secure Socket Layer (SSL) can be used to provide the three characteristics of secure communication. These protocols use a combination of public and private key cryptography and digital certificates.

Instructor Notes:**Appendix H : SSL and WildFly:**

Configuring Security in WildFly:

Application Server's security implementation is called as JBoss SX (It is built on top of the JAAS (Java Authentication and Authorization Service)).

Each JEE component or resource has a different mechanism for defining security. What method to be used and what should be secured is defined in each component's standard deployment descriptor.

JEE specification does not specify the security implementation, also it does not describe where the security data should be kept or how it should be retrieved or how the validation should happen. Every application server vendor has to create its own security implementation and allow programmers to configure and use it through vendor-specific deployment descriptors.

When deploying applications on WildFly most of the time it is likely that we would be deploying a web application and just require a security realm to be defined with certificates

The security subsystem is enabled by default by the addition of the following extension: -

```
<extension module="org.jboss.as.security"/>
```

The namespace used for the configuration of the security subsystem is urn:jboss:domain:security:1.0, the configuration is defined within the <subsystem> element from this namespace.

Within WildFly 8 we make use of security realms to secure access to the management interfaces, these same realms are used to secure inbound access as exposed by JBoss Remoting such as remote JNDI , the realms are also used to define an identity for the server - this identity can be used for both inbound connections to the server and outbound connections being established by the server.

The general structure of a management realm definition is: -

```
<security-realm name="ManagementRealm">
  <plug-ins></plug-ins>
  <server-identities></server-identities>
  <authentication></authentication>
  <authorization></authorization>
</security-realm>
```

Instructor Notes:

plug-ins - This is an optional element that is used to define modules what will be searched for security realm Plug-In Providers to extend the capabilities of the security realms.

server-identities - An optional element to define the identity of the server as visible to the outside world, this applies to both inbound connection to a resource secured by the realm and to outbound connections also associated with the realm.

authentication - This is probably the most important element that will be used within a security realm definition and mostly applies to inbound connections to the server, this element defines which backing stores will be used to provide the verification of the inbound connection.

authorization - This is the final optional element and is used to define how roles are loaded for an authenticated identity. At the moment this is more applicable for realms used for access to EE deployments such as web applications but this will also become relevant as we add role based authorization checks to the management model.

Consider a sample mapping of the standalone.xml file:

```
<security-realm name="UndertowRealm">
    <server-identities>
        <ssl>
            <keystore path="clientkeystore.jks" relative-
to="jboss.server.config.dir" keystore-password="igate123"/>
        </ssl>
    </server-identities>
</security-realm>
```

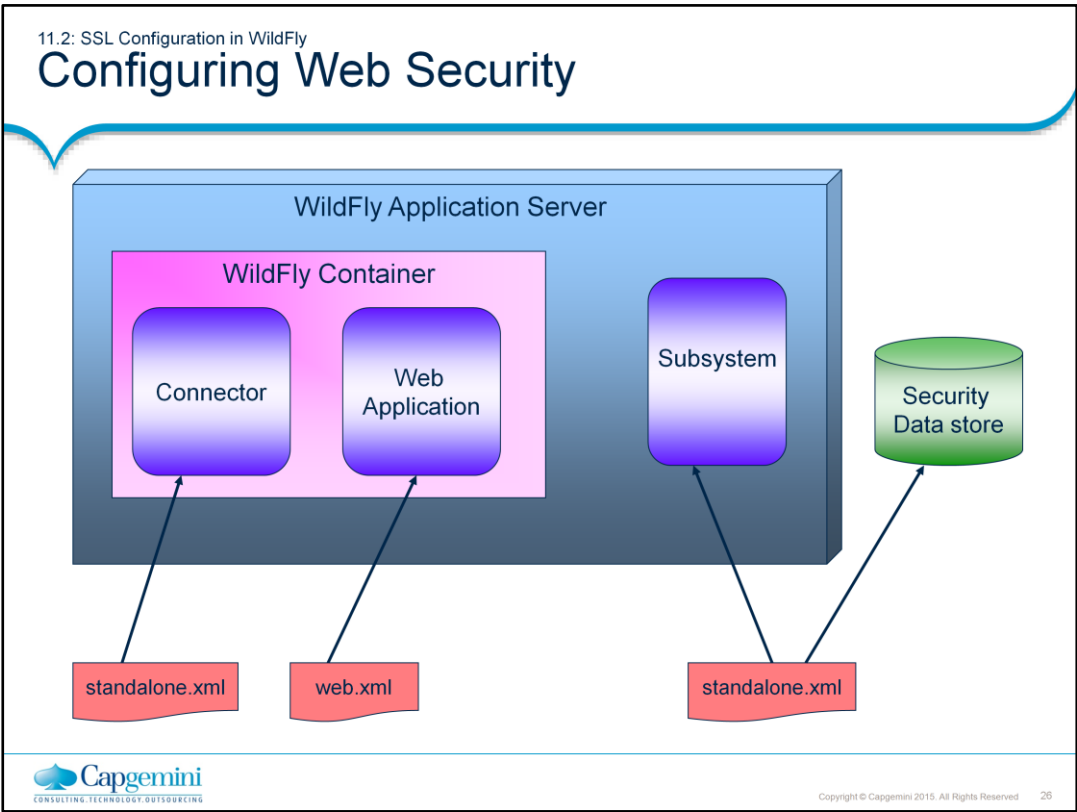
We need to generate a keystore certificate. Save the certificate with .jks extension – Java Key store. Generating certificates would be a part of Module 2

Need to save keystore in D:\wildfly-8.1.0.Final\standalone\configuration folder

The password while generating the keystore for the above certificate is "igate123"; The password could be changed.

Configuring Security for Web Applications is quite simple. Simply configure the SSL through the HTTP connector. But for authenticating clients based on certificate information we need to define SSL aware sub systems

Instructor Notes:




SSL Configuration in WildFly:
Configuring Web Security:
By default, web applications are not secured. The URL of your web application will be accessible publicly.
The WildFly Server is also not secure by default since the Secure HTTP connector is not enabled. The figure on the slide shows which configuration files are used to configure security within the server and for applications. Every file include here has a specific purpose.

Instructor Notes:

11.2: SSL Configuration in WildFly

Configuring Web Security

Configuration Files	Parameters to be configured
WEB-INF/web.xml	Authentication Strategy, URL Patterns, Set of logical roles
D:\wildfly-8.1.0.Final\standalone\configuration (standalone.xml)	Security Realms specifying the keystore certificate
D:\wildfly-8.1.0.Final\standalone\configuration	Secure HTTP Connector

Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Copyright © Capgemini 2015. All Rights Reserved27

SSL Configuration in WildFly:
Configuring Web Security:
The table on the slide mentions about the primary configuration files which are required to configure security for your web applications.

- web.xml: This is the deployment descriptor for your web application. This file configures application security by defining the authentication strategy (BASIC, FORM, DIGEST, CLIENT-CERT). Specifies which URLs of the application are secured and any logical roles
- standalone.xml: This is the configuration file for WildFly Server and is used to define the secure HTTP Connector and set the security certificate generated

Also we need to set the HTTP port to 8443. See below listing to update the change

```
<socket-binding-group name="standard-sockets" default-interface="public" port-offset="{jboss.socket.binding.port-offset:0}">  
  <socket-binding name="https"  
    port="{jboss.https.port:8443}"/>  
</socket-binding-group>
```

Instructor Notes:**Appendix I : Servlet Security:**

There are several ways in which you can secure web applications. These include the following options:

You can define a user authentication method for an application in its deployment descriptor. Authentication verifies the identity of a user, device, or other entity in a computer system, usually as a prerequisite to allowing access to resources in a system. When a user authentication method is specified for an application, the web container activates the specified authentication mechanism when you attempt to access a protected resource.

The options for user authentication methods are discussed in Specifying an Authentication mechanism. All of the example security applications use a user authentication method.

You can define a transport guarantee for an application in its deployment descriptor. Use this method to run over an SSL-protected session and ensure that all message content is protected for confidentiality or integrity. The options for transport guarantees are discussed in Specifying a secure connection.

When running over an SSL-protected session, the server and client can authenticate one another and negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data.

SSL technology allows web browsers and web servers to communicate over a secure connection. In this secure connection, the data is encrypted before being sent, and then is decrypted upon receipt and before processing. Both the browser and the server encrypt all traffic before sending any data. For more information, see Establishing secure connection using SSL.

Digital certificates are necessary when running HTTP over SSL (HTTPS). The HTTPS service of most web servers will not run unless a digital certificate has been installed. Digital certificates have already been created for the Application Server.

Instructor Notes:**We shall see how to secure the users with specific roles****@ServletSecurity**

The @ServletSecurity annotation provides an alternative mechanism for defining access control constraints equivalent to those that could otherwise have been expressed declaratively via security-constraint elements in the portable deployment descriptor.

Using the @ServletSecurity annotation one can now have a descriptor free secure WebApplication.

Prior to Servlet 3.0

Web.xml

```
<security-constraint>
```

```
<web-resource-collection>
```

```
<url-pattern>/TutorialServletBASIC</url-pattern>
```

```
</web-resource-collection>
```

```
<auth-constraint>
```

```
<security-role-name>TutorialUser</security-role-name>
```

```
<security-role-name>guest</security-role-name>
```

```
</auth-constraint>
```

```
</security-constraint>
```

Now:

```
//for all HTTP methods, auth-constraint requiring
membership in Role TutorialUser or guest
```

```
@ServletSecurity(@HttpConstraint(rolesAllowed =
{"MyUser", "guest"}))
```

```
//for particular methods:
```

```
@RolesAllowed("TutorialUser")
```

```
public rettype meth() { ... }
```