UNIT II

Prepared By – KHAN SANA

RequestDispatcher

ResquestDispatcher Interface, Methods of RequestDispatcher, RequestDispatcher Application.

1. **RequestDispatcher Interface:**
   - Server-side mechanism, invisible to the client.
   - Operates within a single server and web application.
   - Not a substitute for URL redirection.
   - Forwarding ideally occurs before response commitment.

Purpose: Enables server-side forwarding and inclusion of resources within a web application.

Location: javax.servlet package.

2. **Obtaining a RequestDispatcher Object:**

Methods:

- **ServletContext.getRequestDispatcher(path):**

Used for resources anywhere within the web application.

Example: ServletContext context = getServletContext(); RequestDispatcher dispatcher = context.getRequestDispatcher("/TargetServlet");

- **ServletRequest.getRequestDispatcher(path):**

Used for resources relative to the current servlet.

Example: RequestDispatcher dispatcher = request.getRequestDispatcher("header.jsp");

3. **Methods of RequestDispatcher:**

- **forward(ServletRequest request, ServletResponse response):**

Redirects the request to a different resource.

Original request and response objects are passed to the target resource.

Current servlet relinquishes control.

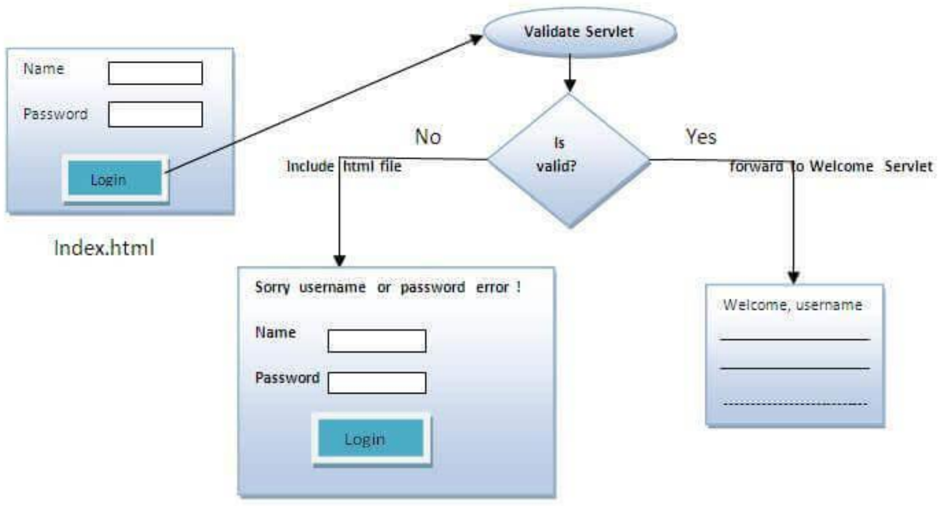
Example: dispatcher.forward(request, response);

- **include(ServletRequest request, ServletResponse response):**

Includes the output of another resource within the current response.

Current servlet retains control.

Example: dispatcher.include(request, response);



Example of RequestDispatcher interface

In this example, we are validating the password entered by the user. If password is servlet, it will forward the request to the WelcomeServlet, otherwise will show an error message: In this example, we have to create following files:

index.html file: for getting input from the user.

Login.java file: a servlet class for processing the response. If password is servet, it will forward the request to the welcome servlet.

WelcomeServlet.java file: a servlet class for displaying the welcome message.

web.xml file: a deployment descriptor file that contains the information about the servlet

Feature	include()	forward()
Control	Remains with the current servlet	Transferred to the target resource
Output	Merges content from both resources	Only shows output from the target resource
Client Response	Single response combining both resources	Response only from the target resource

Use Cases	Reusable headers/footers, dynamic content, modularity	Delegation, security checks, error handling
Timing	Can be used after response initiation	Ideally used before response commitment
Performance	Might incur overhead due to multiple executions	Generally more efficient due to single execution

Cookies

Kinds of Cookies, Where Cookies Are Used? Creating Cookies Using Servlet, Dynamically Changing the Colors of A Page

Cookies : Small text files stored by a server on the client's browser for state management. Essential in HTTP's stateless nature, enabling information retention across requests.

1. HTTP's Statelessness:

Each request-response cycle is independent, lacking built-in memory.

Server forgets the browser after each interaction.

2. Problem for E-commerce:

Impossible to track multiple purchases from the same user for a single bill.

Server treats each request as unique.

3. Cookies as Solution:

Small text files stored by the server on the client's browser.

Enable state persistence across requests.

Allow the server to recognize repeat visits and associate data.

4. How Cookies Work:

Server creates a cookie and sends it in the response header.

Browser stores the cookie locally.

Browser includes the cookie in subsequent requests to the same server/path.

Server identifies the browser and associated data using the cookie.

5. Overcoming Statelessness:

Cookies enable session management, tracking user interactions and preferences.

Essential for online shopping carts, logins, personalization, and more.

6. HTTP Persistent Connections:

Not directly related to cookies.

Involve keeping a connection open for multiple requests, improving efficiency.

Cookies still crucial for state management, even with persistent connections

1. **Cookie Creation and Structure:**

   - Server-side creation: Generated by server-side programs like servlets.
   - Text files: Stored as small text files on the client's computer.
   - Mandatory data: Name and value.
   - Optional attributes: Expiration date, path, domain, secure connection requirement.

2. **Parameters of Cookies:**

   1. Name (required): Unique identifier for the cookie.
   2. Value (required): Data associated with the cookie.
   3. Domain (optional): Specifies the domain where the cookie is valid.
   4. Path (optional): Guides where the cookie is accessible on the server.
   5. Max-Age (optional): Expiration time in seconds (0 for session cookies).
   6. Secure (optional): Enforces HTTPS transmission.
   7. HttpOnly (optional): Impervious to JavaScript access.

3. **Cookie Storage and Transmission:**
   - Browser storage: Saved in a designated subdirectory on the client's hard drive.
   - Automatic transmission: Browser sends relevant cookies to the server with each request.

4. **Server-Side Recognition and State Management:**
   - Identification: Server identifies returning browsers using cookies.

- State maintenance: Overcomes HTTP's stateless nature by associating data with specific browsers.

5. **Application Possibilities:**
   - Personalized experiences: Tailoring content and recommendations.
   - Shopping cart persistence: Remembering items across sessions.
   - User tracking and analytics: Understanding website usage patterns.
   - Targeted advertising: Delivering relevant ads based on interests.
   - Authentication and session management: Maintaining user logins.
   - Multilingual website preferences: Remembering language settings.

Cookies enable stateful interactions between browsers and servers.

They bridge the gap between HTTP's request-response nature and the need for context across multiple requests.

Careful consideration of privacy and security implications is crucial when using cookies.

Kinds of Cookies:

1. Permanent/Persistent Cookies:

1. **Storage:** Remain on the user's computer even after closing the browser.
2. **Applications**:
   - Personalizing website experiences (e.g., language, layout).
   - Analyzing user behavior and website performance.
   - Identifying individual users for targeted advertising or content.
3. **Function:**
   - Remember user preferences across sessions (e.g., login details, language).
   - Analyze user behavior within the website.
   - Track website performance metrics (visitors, time spent on pages).
   - Can be used for user identification and targeted advertising.
4. **Example:** Remembering login information for a specific website.

2. Session/Transient Cookies:

1. **Storage**: Stored temporarily in the browser's memory, deleted when closed.
2. **Applications:**
   - Keeping track of shopping cart items.
   - Facilitating user login sessions without repeated authentication.
   - Enabling website features like dynamic content or multi-page forms.
3. **Function**:
   - Maintain session state (e.g., shopping cart items, form data).
   - Allow navigating within a website without repeated logins.

- Do not collect personal information or identify individual users.

4. **Example:** Keeping track of items added to a shopping cart during a session.

Key Differences:

1. **Persistence**: Permanent cookies last longer, session cookies are temporary.
2. **Usage**: Permanent cookies for preferences and tracking, session cookies for active sessions.
3. **Privacy**: Permanent cookies can be used for identification, session cookies are more anonymous.
4. **Expiration**: Both types can have expiration dates, though session cookies typically expire sooner.
5. **Security**: Consider user privacy when using cookies, especially persistent ones.
6. **Choice**: Users should have control over cookie settings and opt-out options.

Where Cookies Are Used:

1. Enhancing User Experience:

- Temporary Session Management: Remember user progress and choices across page visits.
- User Authentication: Maintain login sessions without repeated logins.
- Personalization: Adapt website content and recommendations based on user preferences.
- Shopping Cart Functionality: Track and persist items added to the cart.
- Low-Security Preference Storage: Remember search settings, recent activity, etc.

2. Data Analytics and Functionality:

- User Statistics: Compile usage data for website optimization and targeted advertising.
- Visitor Profiling: Build individualized profiles of website visitors.
- Fee-Based Services: Store registration information for paid services.

3. Specific Application Examples:

- Shopping Cart Apps: Track shopping cart contents and user IDs.
- Online Banking: Maintain secure login sessions.
- Website Tracking: Analyze user behavior for website improvement.

- Personalized Content Delivery: Tailor content based on user preferences.

4. Benefits for Businesses:

- Repeat Customer Tracking: Identify and incentivize repeat customers.
- Targeted Advertising: Deliver relevant ads based on user interests.
- Improved Functionality: Enhance user experience and website efficiency.

5. Privacy Considerations:

- Cookies can raise privacy concerns due to potential user tracking.
- Transparency and user control over cookie usage are crucial.

Creating a Cookie in Pointwise Steps:

Import the Cookie Class:

Include the necessary library to work with cookies in your language/framework.

Example in Java: import javax.servlet.http.Cookie;

Initialize a Cookie Object:

Use the Cookie() constructor to create a new cookie instance.

Mandatory Arguments:

name: A unique string identifier for the cookie.

value: The data you want to store in the cookie.

Syntax: Cookie myCookie = new Cookie(name, value);

Optional Cookie Attributes (if needed):

Set additional properties using setter methods:

setMaxAge(int expiry): Expiration time in seconds (0 for session cookies).

setPath(String uri): Restricts the cookie to a specific path on the server.

setDomain(String domain): Specifies the domain where the cookie is valid.

setSecure(boolean flag): Marks the cookie for secure HTTPS transmission.

setHttpOnly(boolean flag): Prevents JavaScript access to the cookie.

Add the Cookie to the Response:

Use the appropriate method to attach the cookie to the HTTP response header.

Example in Java Servlets: response.addCookie(myCookie);

Browser Storage and Transmission:

The browser automatically stores the cookie on the client's computer.

It includes the cookie in subsequent requests to the same domain/path.

PTR

Cookie names are case-sensitive and cannot contain spaces or certain special characters.

Values can contain any valid string data.

Manage cookie settings carefully to balance functionality and user privacy.

Additional Methods of Cookie Class:

Setter Methods:

1. setVersion(int version): Specifies the cookie protocol version to use.
2. setDomain(String domain): Restricts cookie access to a specific domain pattern.
3. setPath(String path): Limits cookie transmission to certain URIs on the server.
4. setMaxAge(int expiry): Sets cookie expiration time in seconds (0 = immediate deletion, negative = session cookie).
5. setSecure(boolean flag): Requires cookie transmission over secure HTTPS/SSL channels.
6. setComment(String purpose): Adds a descriptive comment to explain the cookie's purpose.
7. setValue(String newValue): Updates the cookie's value with new data.
8. setHttpOnly(boolean flag): Shields the cookie from client-side scripting access.

Getter Methods:

1. getVersion(): Retrieves the cookie's protocol version.
2. getDomain(): Returns the cookie's domain restriction pattern.
3. getPath(): Retrieves the cookie's path restriction.
4. getMaxAge(): Returns the cookie's expiration time in seconds.

5. getSecure(): Indicates whether the cookie requires secure transmission.
6. getComment(): Fetches the cookie's descriptive comment.
7. getValue(): Returns the cookie's stored value.
8. getName(): Retrieves the cookie's name (unchangeable after creation).
9. isHttpOnly(): Checks if the cookie has the HttpOnly flag set.

Setting a Cookie

1. **Create a Cookie Object:**

Use the Cookie(String name, String value) constructor to create a cookie with a unique name and the desired value.

Example: Cookie userCookie = new Cookie("username", "ABC");

2. **Attach the Cookie to the Response:**

Employ the addCookie(Cookie cookie) method of the HttpServletResponse object to add the cookie to the response headers.

Example: response.addCookie(userCookie);

Ensure this is done before sending any content in the response.

3. **Send Multiple Cookies (if needed):**

Call addCookie() multiple times to send multiple cookies.

Alternatively, combine multiple values into a single cookie using a separator character.

HTTP's Stateless Nature:

Definition: Server cannot remember prior connections or distinguish visitors.

Contrast with FTP: FTP maintains visitor credentials throughout a session.

Limitations:

Server cannot maintain visitor preferences.

Server cannot track individual visitor journeys.

Advantages of Statelessness:

Simplicity: Protocol is straightforward.

Efficiency: Conserves server resources.

Scalability: Supports numerous simultaneous visitors.

Disadvantages of Statelessness:

Overhead: New connections required for each request.

Tracking Limitations: Server cannot track individual visitors.

Association Difficulties: Requests cannot be automatically linked to sessions.

Workarounds:

Sessions: Server stores and retrieves data related to a client who self-identifies with each request.

Cookies: Server sends data to the client, who then sends it back with subsequent requests

Sessions

Purpose:

Mitigate HTTP's stateless nature by tracking visitor-specific data across multiple requests.

Store temporary data not intended for external access.

Key Features:

Server-Side Storage: Data is kept on the server, unlike cookies which reside on the client's device.

Session IDs: Unique identifiers track individual clients and associate requests with their sessions.

Session Manager: Java Servlet module that automatically manages sessions after client login.

Duration:

Configurable on the web server, often expiring after a certain period of inactivity (e.g., 30 minutes).

Applications:

Shopping Carts: Track items added to a cart across multiple pages.

User Authentication: Persist login status, eliminating the need for repeated logins.

Personalization: Store visitor preferences to tailor website experiences.

Relationship with Cookies:

While both track client data, sessions store information on the server, while cookies store it on the client's device.

Sessions can utilize cookies to store session IDs, but they are not strictly dependent on them.

HTTP Session Lifecycle:

1. Request for Resource: Visitor initiates a request from their web browser.
2. Authentication: Server prompts for login credentials.
3. Credentials Submission: Visitor provides valid username and password.
4. Session ID Generation: Server creates a unique Session ID to identify the visitor.
5. Session ID is typically sent as a cookie to the browser.
6. Session Begins: HTTP Session commences with Session ID tracking requests.
7. Multiple Requests: Browser sends subsequent requests with Session ID.
8. Server identifies visitor and privileges based on Session ID.
9. Session Inactivity/Explicit Logout: Session either expires after a timeout (e.g., 30 minutes) or ends immediately if the visitor explicitly logs out.
10. Session Expiration: Server deletes Session ID, ending the session

Session Management Rules:

1. State Storage:

   Maintain user-specific information (e.g., shopping cart items, personal details) across multiple requests.

2. Identifier Association:
   Each request must carry a unique identifier (e.g., session ID) to link it to the stored state.
3. Timeout Enforcement:
   Implement a timeout mechanism to automatically end inactive sessions, preventing indefinite server-side resource consumption.

SESSION TRACKING TECHNIQUES

There are four techniques used in Session tracking:

1. **Cookies**
2. **Hidden Form Field**
3. **URL Rewriting**
4. **HttpSession**

2) HIDDEN FORM FIELD

- In case of Hidden Form Field a hidden (invisible) textfield is used for maintaining the state of an user.
- In such case, we store the information in the hidden field and get it from another servlet. This approach is better if we have to submit form in all the pages and we don't want to depend on the browser.
- Let's see the code to store value in hidden field.
  <input type="hidden" name="uname" value="Vimal Jaiswal">
  Here, uname is the hidden field name and Vimal Jaiswal is the hidden field value.

Real application of hidden form field

- It is widely used in comment form of a website. In such case, we store page id or page name in the hidden field so that each page can be uniquely identified.

Advantage of Hidden Form Field

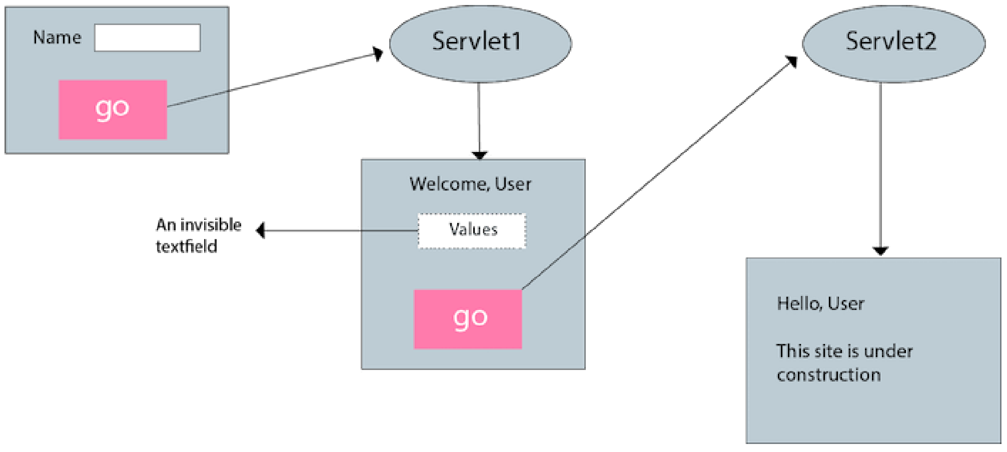
- It will always work whether cookie is disabled or not.

Disadvantage of Hidden Form Field:

- It is maintained at server side.
- Extra form submission is required on each pages.
- Only textual information can be used.

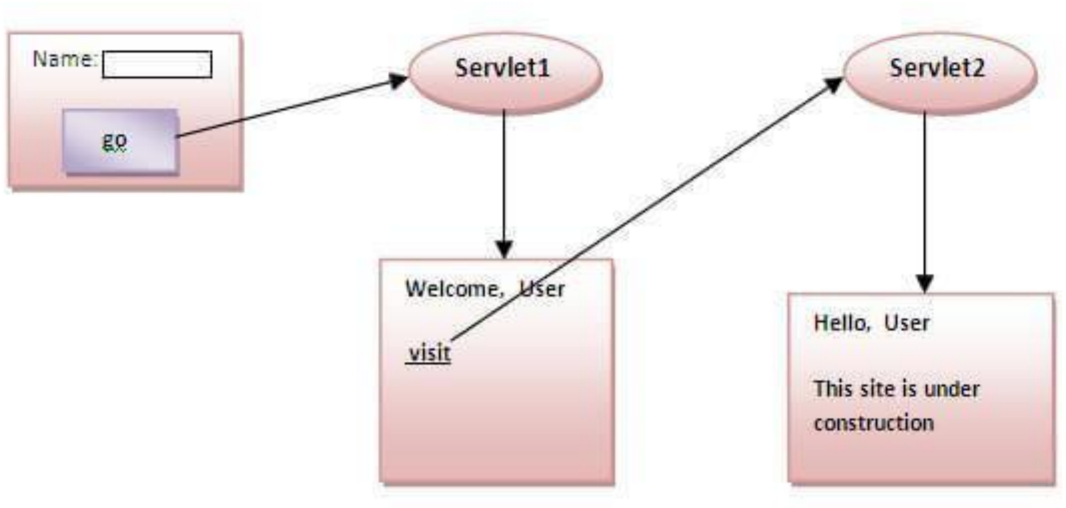
Example of using Hidden Form Field

- In this example, we are storing the name of the user in a hidden textfield and getting that value from another servlet.



3)URL REWRITING

- In URL rewriting, we append a token or identifier to the URL of the next Servlet or the next resource. We can send parameter name/value pairs using the following format: url?name1=value1&name2=value2&??
- A name and a value is separated using an equal = sign, a parameter name/value pair is separated from another parameter using the ampersand(&). When the user clicks the hyperlink, the parameter name/value pairs will be passed to the server. From a Servlet, we can use getParameter() method to obtain a parameter value.



Advantage of URL Rewriting

- It will always work whether cookie is disabled or not (browser independent).
- Extra form submission is not required on each pages.

Disadvantage of URL Rewriting

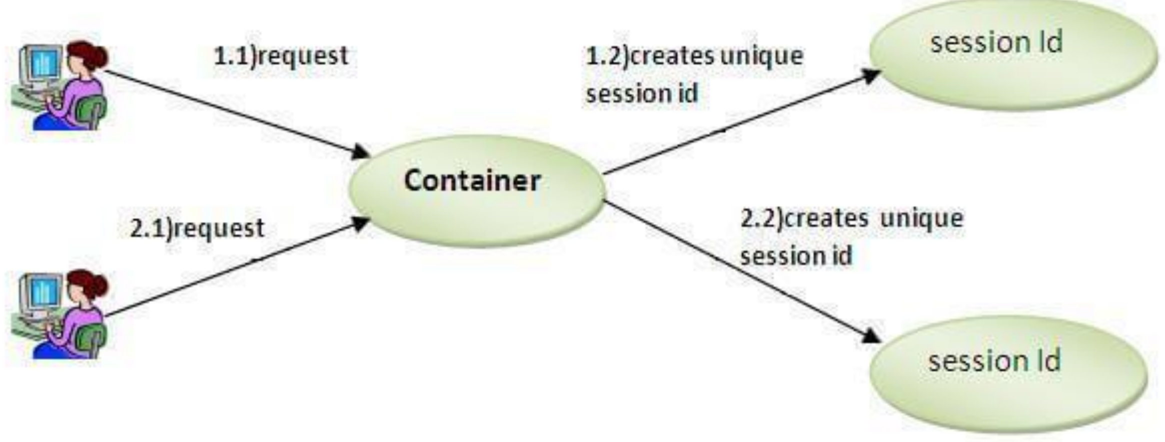
- It will work only with links.
- It can send Only textual information

.

4) HTTPSESSION INTERFACE

In such case, container creates a session id for each user.The container uses this id to identify the particular user.An object of HttpSession can be used to perform two tasks:

- bind objects
- view and manipulate information about a session, such as the session identifier, creation time, and last accessed time.



<u>How to get the HttpSession object ?</u>

The HttpServletRequest interface provides two methods to get the object of HttpSession:

getSession(): Returns the current session associated with this request, or if the request does not have a session, creates one.

getSession(boolean create): Returns the current HttpSession associated with this request or, if there is no current session and create is true, returns a new session.

Commonly used methods of HttpSession interface

getId(): Returns a string containing the unique identifier value.

getCreationTime(): Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.

getLastAccessedTime(): Returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.

invalidate(): Invalidates this session then unbinds any objects bound to it.

HttpSession Class in Servlets:

Purpose:

Server-side mechanism to manage user-specific data across multiple HTTP requests.
Addresses the stateless nature of HTTP.
Session data is stored on the server, not in cookies.
Session IDs are typically sent as cookies for tracking, but URL rewriting can be used.
Consider long-term data storage in a database for persistence.
Use HttpSession for short-term state management and login status.

Key Methods:

Obtaining an HttpSession Object:

HttpSession session = request.getSession(); (creates a new session if none exists)

HttpSession session = request.getSession(false); (returns null if no session exists)
Storing and Retrieving Attributes:

session.setAttribute(String name, Object value);
Object value = session.getAttribute(String name);
session.removeAttribute(String name);
Accessing Session Information:

String sessionId = session.getId();
long creationTime = session.getCreationTime();
long lastAccessedTime = session.getLastAccessedTime();
Invalidating a Session:

session.invalidate(); (destroys session, removing all associated data)
Setting Session Timeout:

session.setMaxInactiveInterval(int interval); (specifies timeout in seconds)
Example:

```
Java
Protected void  doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

   HttpSession session = request.getSession();

   // Store user name in session
   session.setAttribute("userName", "John Doe");

   // Retrieve user name from session
   String userName = (String) session.getAttribute("userName");

   // Invalidate the session (e.g., on logout)
   session.invalidate();
}
```

Uploading Files in Java EE Servlets

Handling file uploads in Java EE Servlets has become simpler since Servlet 3.0 onwards with the introduction of the @MultipartConfig annotation.

Annotation Usage:

```java
java
@WebServlet(urlPatterns = {"/FileUploadServlet"})
@MultipartConfig(
    location = "C:/upload",
    fileSizeThreshold = 1024 * 1024, // 1 MB
    maxFileSize = 1024 * 1024 * 5,   // 5 MB
    maxRequestSize = 1024 * 1024 * 5 * 5 // 25 MB
)
public class FileUploadServlet extends HttpServlet {
    // Servlet code here
}
```

The @MultipartConfig annotation is used to configure settings for handling file uploads. Key attributes include:
- location: Absolute path to a directory on the file system where files will be temporarily stored.
- fileSizeThreshold: File size in bytes after which the file will be stored on disk temporarily.
- maxFileSize: Maximum size allowed for uploaded files.
- maxRequestSize: Maximum size allowed for the entire request.

Example web.xml Configuration:

```xml
xml
<servlet>
    <servlet-name>FileUploadServlet</servlet-name>
    <servlet-class>com.example.FileUploadServlet</servlet-class>
    <multipart-config>
        <location>C:/upload</location>
        <file-size-threshold>1048576</file-size-threshold> <!-- 1 MB -->
        <max-file-size>5242880</max-file-size> <!-- 5 MB -->
        <max-request-size>26214400</max-request-size> <!-- 25 MB -->
    </multipart-config>
</servlet>
```

This XML configuration is an alternative way to set up the @MultipartConfig settings for your servlet. It is useful when annotations are not preferred or not applicable.

Creating an Upload File Application

To create an application that allows file uploads, you would typically need:
1. Two Servlets (One for handling the file upload, and another for displaying the form and processing successful uploads).

2. An HTML form for browsing files, specifying the upload location, and submitting the upload request.
3. A display page showing a success message and providing a link to return for more uploads.

Your application's structure might look like this:

1. FileUploadForm.html (HTML form for file upload):

```
html
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>File Upload Form</title>
</head>
<body>
   <form action="/FileUploadServlet" method="post" enctype="multipart/form-data">
      <label for="file">Select File:</label>
      <input type="file" id="file" name="file" required><br>
      <label for="location">Upload Location:</label>
      <input type="text" id="location" name="location" required><br>
      <input type="submit" value="Upload">
   </form>
</body>
</html>
```

2. FileUploadServlet.java (Servlet for handling file uploads):

```
java
@WebServlet(urlPatterns = {"/FileUploadServlet"})
@MultipartConfig(
   location = "C:/upload",
   fileSizeThreshold = 1024 * 1024, // 1 MB
   maxFileSize = 1024 * 1024 * 5,   // 5 MB
   maxRequestSize = 1024 * 1024 * 5 * 5 // 25 MB
)
public class FileUploadServlet extends HttpServlet {
   // Servlet code for handling file uploads
}
```

3. UploadSuccess.jsp (JSP page to display success message and provide a link to return for more uploads):

```
jsp
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>File Upload Success</title>
</head>
<body>
  <h2>File Upload Successful!</h2>
  <p>Click <a href="/file-upload-form.html">here</a> to upload more files.</p>
</body>
</html>
```

Non-blocking I/O (Input/Output) in servlets and JSP (JavaServer Pages) is introduced in Servlet 3.1 specification, and it allows for improved scalability and performance in handling concurrent requests. The traditional blocking I/O approach creates a new thread for each incoming request, which can lead to resource exhaustion in situations with a large number of simultaneous connections. Non-blocking I/O enables handling more connections concurrently using a smaller number of threads.

Servlets with Non-blocking I/O:

To use non-blocking I/O in servlets, you can leverage asynchronous processing. Here's a simple example:

1. **Servlet with Async Support:**

```
java
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;

@WebServlet(urlPatterns = "/AsyncServlet", asyncSupported = true)
```

```
public class AsyncServlet extends HttpServlet {

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {
      response.setContentType("text/html;charset=UTF-8");

      final PrintWriter writer = response.getWriter();
      writer.println("<html><head><title>Async Servlet</title></head><body>");
      writer.println("<h2>Async Servlet Example</h2>");

      final AsyncContext asyncContext = request.startAsync();
      asyncContext.setTimeout(10000); // Set the timeout for async operation (in
milliseconds)

      // Start a new thread for processing the request asynchronously
      asyncContext.start(() -> {
         try {
            // Simulate a time-consuming task
            Thread.sleep(5000);

            // Write the response back to the client
            writer.println("<p>Processing complete after 5 seconds!</p>");
            writer.println("</body></html>");

            // Complete the asynchronous processing
            asyncContext.complete();
         } catch (Exception e) {
            e.printStackTrace();
         }
      });
   }
}
```

In this example, the AsyncServlet uses the @WebServlet annotation with asyncSupported = true to enable asynchronous support. The doGet method starts an asynchronous context, starts a new thread for processing, simulates a time-consuming task, and completes the asynchronous processing.

JSP with Non-blocking I/O:

JSP pages can also take advantage of asynchronous processing. Here's an example:

1. **JSP Page for Async Processing:**

```jsp
jsp
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ page import="java.io.PrintWriter"%>
<%@ page import="javax.servlet.AsyncContext"%>
<%@ page import="java.util.concurrent.TimeUnit"%>

<%
   response.setContentType("text/html;charset=UTF-8");

   out.println("<html><head><title>Async JSP</title></head><body>");
   out.println("<h2>Async JSP Example</h2>");

   AsyncContext asyncContext = request.startAsync();
   asyncContext.setTimeout(TimeUnit.SECONDS.toMillis(10)); // Set the timeout for
async operation

   // Start a new thread for processing the request asynchronously
   asyncContext.start(() -> {
      try {
         // Simulate a time-consuming task
         Thread.sleep(5000);

         // Write the response back to the client
         out.println("<p>Processing complete after 5 seconds!</p>");
         out.println("</body></html>");

         // Complete the asynchronous processing
         asyncContext.complete();
      } catch (Exception e) {
         e.printStackTrace();
      }
   });
%>
```

In this JSP example, the scriptlet starts an asynchronous context, starts a new thread for processing, simulates a time-consuming task, and completes the asynchronous processing.

Certainly! Let's break down the provided code for the non-blocking I/O examples in both servlets and JSP.

Servlet with Non-blocking I/O:

1. **Servlet Annotation:**
  java
  @WebServlet(urlPatterns = "/AsyncServlet", asyncSupported = true)

  This annotation declares the servlet named AsyncServlet and specifies that it supports asynchronous processing (asyncSupported = true).

2. **doGet Method:**
  java
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

  This method handles the HTTP GET request.

```
java
final PrintWriter writer = response.getWriter();
writer.println("<html><head><title>Async Servlet</title></head><body>");
writer.println("<h2>Async Servlet Example</h2>");
```

  Here, a PrintWriter is used to write HTML content to the response. The initial HTML structure is set up for the response.

```
java
final AsyncContext asyncContext = request.startAsync();
asyncContext.setTimeout(10000);
```

  An AsyncContext is created by calling startAsync(). The setTimeout method sets the timeout for asynchronous operations to 10 seconds.

```
java
asyncContext.start(() -> {
   // Asynchronous processing logic goes here
});
```

  The start method is called on the AsyncContext to initiate asynchronous processing. A new thread is started, and the logic inside the lambda expression is executed asynchronously.

```
java
asyncContext.complete();
```

  Finally, the complete method is called to signal the completion of the asynchronous processing.

 JSP with Non-blocking I/O:

1. **JSP Page:**

```
jsp
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ page import="java.io.PrintWriter"%>
<%@ page import="javax.servlet.AsyncContext"%>
<%@ page import="java.util.concurrent.TimeUnit"%>
```

These are JSP directives specifying the language, character set, and imported classes.

```
jsp
<%!
   AsyncContext asyncContext;
%>
```

A scriptlet is used to declare a variable asyncContext, which will store the asynchronous context.

```
jsp
<%
   response.setContentType("text/html;charset=UTF-8");

   out.println("<html><head><title>Async JSP</title></head><body>");
   out.println("<h2>Async JSP Example</h2>");

   asyncContext = request.startAsync();
   asyncContext.setTimeout(TimeUnit.SECONDS.toMillis(10));
%>
```

The scriptlet sets up the HTML structure, starts an asynchronous context using startAsync(), and sets the timeout to 10 seconds.

```
jsp
<%
   asyncContext.start(() -> {
      // Asynchronous processing logic goes here
   });
%>
```

Similar to the servlet example, the asynchronous processing logic is placed inside a lambda expression, and start is called on the AsyncContext to initiate asynchronous processing.

```
jsp
<%
```

```
    asyncContext.complete();
  %>
```

Finally, the complete method is called to signal the completion of the asynchronous processing.

Both examples demonstrate the use of non-blocking I/O by leveraging asynchronous processing. The servlet example uses an annotated servlet class, while the JSP example uses scriptlets. In both cases, an AsyncContext is used to initiate and complete asynchronous processing, allowing the server to handle more concurrent requests efficiently. It's important to note that proper error handling and resource management should be considered in a real-world scenario.