Java EE 6
Servlet 3.0 Advanced

Sang Shin
JPassion.com
"Learn with Passion!"

Topics

- Servlet 3.0 Asynch. support
  - Technical background
  - Server side push quick tutorial
  - Server side support in Servlet 3.0
  - APIs
- Security enhancements
- Misc. features

Servlet 3.0 Asynch. Support: Technical Background

Factors for Server side Scalability and Performance

- Thread is an expensive resource on the server
  - How efficiently thread resource on the server is being utilized affect scalability (how many clients it can support) and performance
  - Why event-driven server side programming model is becoming popular
- Multi-core architecture of the platform should be leveraged
  - Why parallel programming is becoming popular
  - Why functional programming is becoming popular

Web Server Scalability Evolution

1. HTTP 1.0 to HTTP 1.1
2. Thread per Connection
3. Thread per Request
4. Asynch. Servlet

HTTP 1.0 to HTTP 1.1

- A major improvement in the HTTP 1.1 standard (over HTTP 1.0) is persistent connections (sometimes called “HTTP keep-alive”)
  - In HTTP 1.0, a TCP connection between a Web client and server is closed after a single request/response cycle as a default
  - In HTTP 1.1, a TCP connection is kept alive and reused for multiple HTTP requests/responses as a default – this is called “persistent connections” (HTTP keep-alive is set through HTTP keep-alive header)
- Persistent connections in HTTP 1.1 reduce communication lag perceptibly, because the client doesn't need to recreate TCP connection after each HTTP request/response cycle

Thread per Connection: What is it?

- Used for enhancing scalability of web servers
- Based on HTTP 1.1's persistent connections
  - > Each persistent HTTP connection between client and server is associated with one thread on the server side
  - > Threads are allocated from a server-managed thread pool
  - > Once a connection is closed, the dedicated thread is recycled back to the pool and is ready to serve other tasks
- Scalability is still constrained, however
  - > X number of threads can handle only X number of connections
  - > For many Web sites, users request pages from the server only sporadically (meaning long idle time between requests) so the connection threads at the server are idle most of the time -This is a waste of thread resource
  - > Hence the reason for “Thread per request” is born

Thread per Request: What is it?

- Thanks to the non-blocking I/O capability introduced in Java 4's New I/O APIs for the Java Platform (NIO) package, a persistent HTTP connection doesn't require that a thread be constantly attached to it
- Threads can be allocated to connections only when requests are being processed
  - When a connection is idle between requests, the thread can be recycled, and the connection is placed in a centralized NIO select set to detect new requests without consuming a separate thread
- This model, called “thread per request”, potentially allows Web servers to handle a larger number of user connections with a fixed number of threads
  - Solves the scalability problem of the “Thread per connection”

Thread per Request: Who supports it?

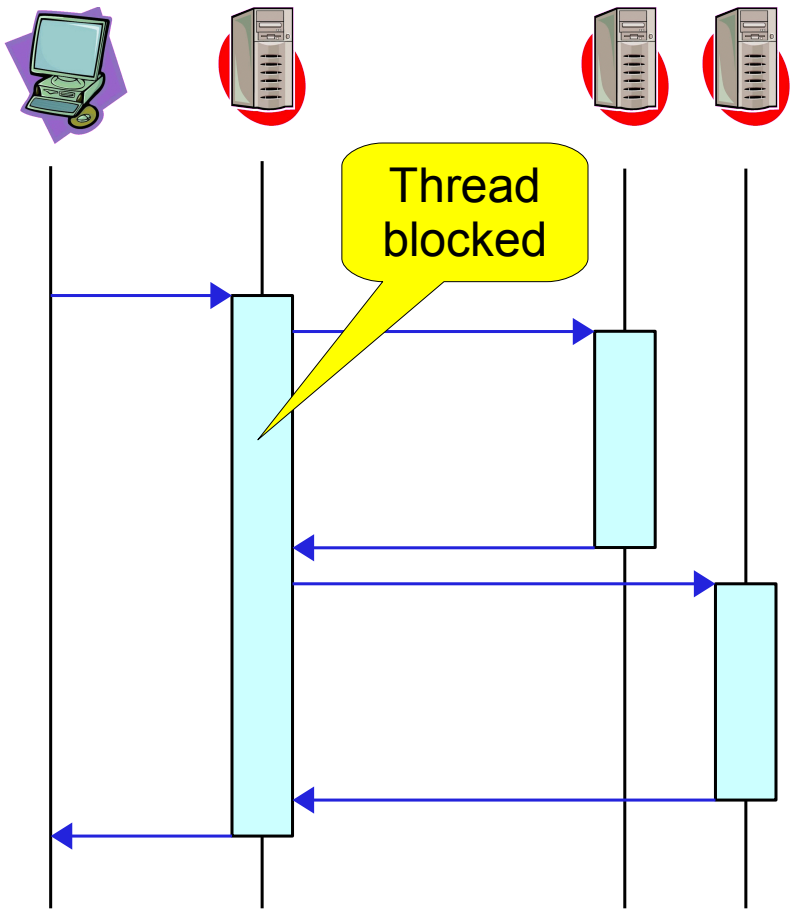
- Popular Web servers support it by default -- Tomcat, Jetty, GlassFish (Grizzly), WebLogic, and WebSphere -- all use “thread per request” through Java NIO (non-blocking I/O)
- Developers do not have to do anything in order to take advantage of “Thread per request” model since web servers support it by default

Factors that further impact the scalability

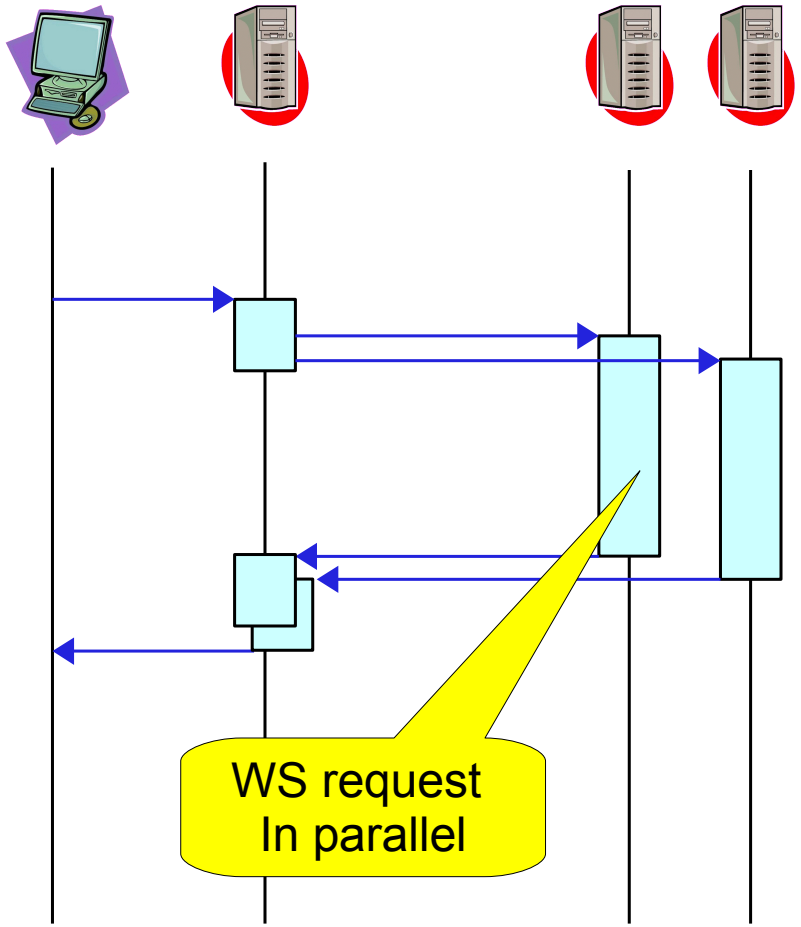
- Ajax
  - Ajax enabled-client sends more requests to the server (than non-Ajax-enabled clients), increasing the traffic between client and server – causes a new scalability problem
- Threads (as part of request handling) accessing slow-responding resources
  - Request handling might involve accessing remote resources - such as web services or database
  - Threads that handle those requests could be waiting for long time for the responses from those remote resources
- So we need a better solution in which
  - Recycle the threads that are waiting for a response from remote resources by placing the request in a centralized queue waiting for available resources and release the thread (This is what Servlet 3.0 Aynch. support is for.)

Waiting for responses from Web Services

Blocking thread (idle thread) in Servlet 2.5



Asynchronous support in Servlet 3.0



Idling threads impacts scalability

- Let's say we have a Web Application accessing remote web services with following conditions
  - Handling 1000 requests / sec
  - 50% requests call remote web service
  - 500 threads in container thread pool
- If remote web service is slow (it takes 1000ms to respond)
  - Thread starvation occurs in 1 second!
  - 500 requests (50% of 1000 requests) use all 500 threads
- Asynch. Servlet support addresses this problem

Why Asynchronous Servlets?

- Async Servlets reuse threads that handle requests that are
  - Waiting for responses (eg web services)
  - Waiting for resources (eg JDBC connection)
  - Waiting for events (eg Chat message)

Example Code: Asynch. Servlet

- This is the part of queuing AsyncContext object, which captures pending request/response pair

```
// When the asyncSupported attribute is set to true, the response object is not
// committed on method exit. It will be held in the AsyncContext object.
@WebServlet(name="myServlet", urlPatterns={"/slowprocess"}, asyncSupported=true)
public class MyServlet extends HttpServlet {

   public void doGet(HttpServletRequest request, HttpServletResponse response) {
      // Calling startAsync() returns an AsyncContext object that holds the
      // request/response object pair. The AsyncContext object is then stored in an
      // application-scoped queue. The AsyncContext object  is then retrieved
      // later on by a different thread for sending back response.
      AsyncContext aCtx = request.startAsync(request, response);
      ServletContext appScope = request.getServletContext();
      ((Queue<AsyncContext>)appScope.getAttribute("slowWebServiceJobQueue")).add(aCtx);

      // Without any delay, the doGet() method returns to the runtime, and the original request
      // thread is recycled
   }
}
```

Servlet 3.0 Asynch. Support: Quick tutorial on Server-side Push

What is Server-side Push?

- The first-generation Ajax is not “asynchronous” enough
  - It handles a single user client-to-server asynchronicity only but not server-to-client asynchronocity – in other words, the server cannot send data to the client whenever it wants to
- Full asynchronicity includes “updates pushed from server to the clients”(server-to-client) at any time
  - Server-driven page update at the client
- Allow multiple users to communicate and collaborate within the web application
  - Chat, auction, online gaming
- Sometimes called as “Comet”, “Server-side Push”, “Ajax Push”, or “Reverse Ajax”

Applications with Server-side Push

- Chat
- Auctions
- Distance learning
- Games
- Collaborative authoring
- Shared WebDAV filesystem
- Blogging and reader comments
- SIP-coordinated mobile applications
- Hybrid chat/email/discussion forums
- Customer assistance on sales/support pages
- Multi-step business process made collaborative
- Shared trip planner or restaurant selector with maps
- Shared calendar, “to do” list, project plan

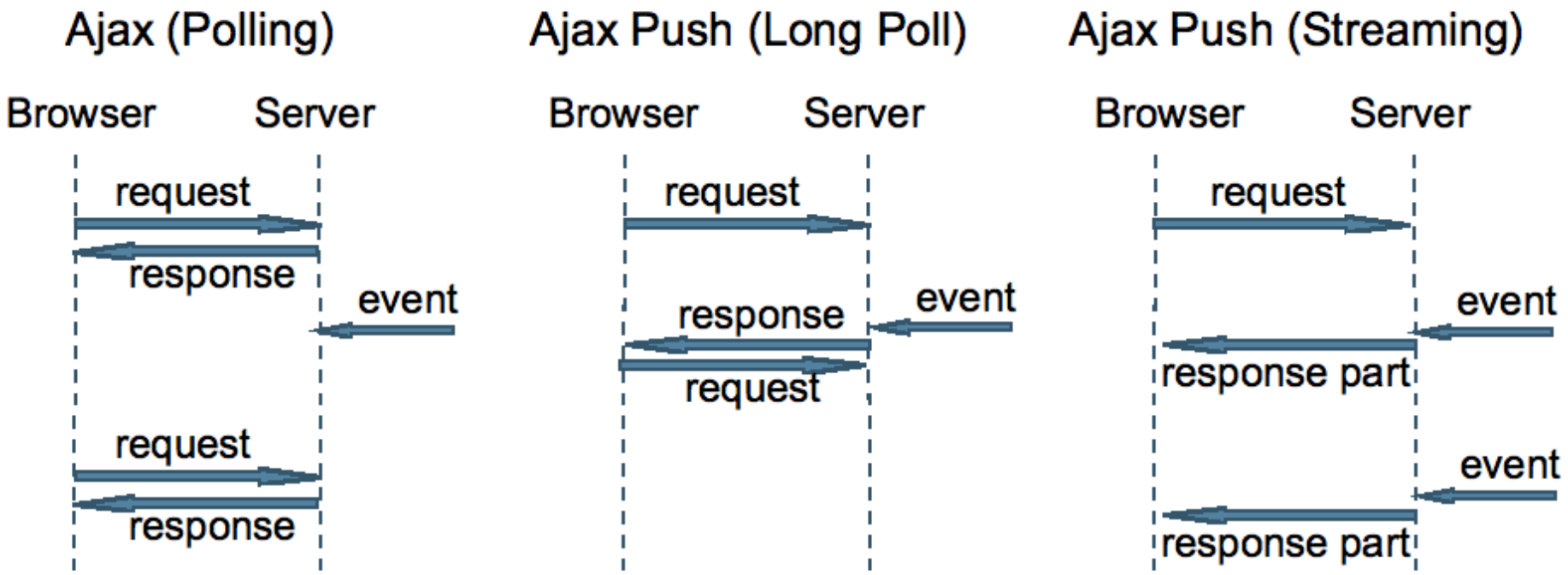
What is Comet (Ajax Push)?

Responsive, low-latency interaction for the web

- A programming technique that enables web servers to send data to the client without having any need for the client to request for it
- Allows creation of highly responsive, event-driven web applications
  - Keep clients up-to-date with data arriving or changing on the server, without frequent polling
- Advantages
  - Lower latency, not dependent on polling frequency
  - Server and network do not have to deal with frequent polling requests to check for updates
- In HTML5, this is supported natively through WebSockets

Ajax Poll vs Ajax Push

Bending the rules of HTTP.



- **Comet (or Server side push)** refers to both the Long Polling and Streaming methods of web programming

Ajax Poll vs Ajax Push

Bending the rules of HTTP.

- Poll:
  - Send a request to the server every X seconds
  - The response is “empty” if there is no update
- Long Poll: (most popular)
  - Client sends a request to the server, server waits for an event to happen, then send the response
  - The response is never empty
  - No client side hack needed – the server functions like a slow server
- Http Streaming:
  - Client sends a request, server waits for events, stream multi-part/chunked responses, and then wait for the events
  - The response is continually appended to

Servlet 3.0 Asynch. Support: Server-side Push support in Servlet 3.0

Proprietary Solutions before Servlet 3.0

- Lack of asynchronous support in the Servlet 2.5 has caused server vendors to devise workarounds by weaving proprietary classes and interfaces that promote scalable Comet programming into the Servlet 2.5 API
  - Tomcat has a *CometProcessor* class
  - Jetty 6 has *Continuations* class
  - GlassFish Grizzly has a *CometEngine* class
- These applications using proprietary classes and interfaces were not portable
  - Hence the reason of “Asynch. support in Servlet 3.0” is born

Standard & Portable Solution in Servlet 3.0

- Asynch. Servlet API (in Servlet 3.0) now provides standard and portable way of supporting server-side push

Servlet 3.0 Asynch. Support: APIs

Asynch. Support Configuration

- Asynch. support can be configured in several ways
- Option #1: With annotation (most common)
  - *@WebServlet(asyncSupported=true)*
- Option #2: through “web.xml”
  - *<async-supported>true</async-supported>*
- Option #3: Programmatic
  - *registration.setAsyncSupported(boolean)*

Asynch. Servlet APIs

- ServletRequest#startAsync()
  - Returns *AsyncContext* object, which contains HTTP request and response objects
  - The *AsyncContext* object can be saved somewhere for later processing by another thread
- ServletRequest#isAsyncSupported()
  - Checks if this request supports asynchronous operation

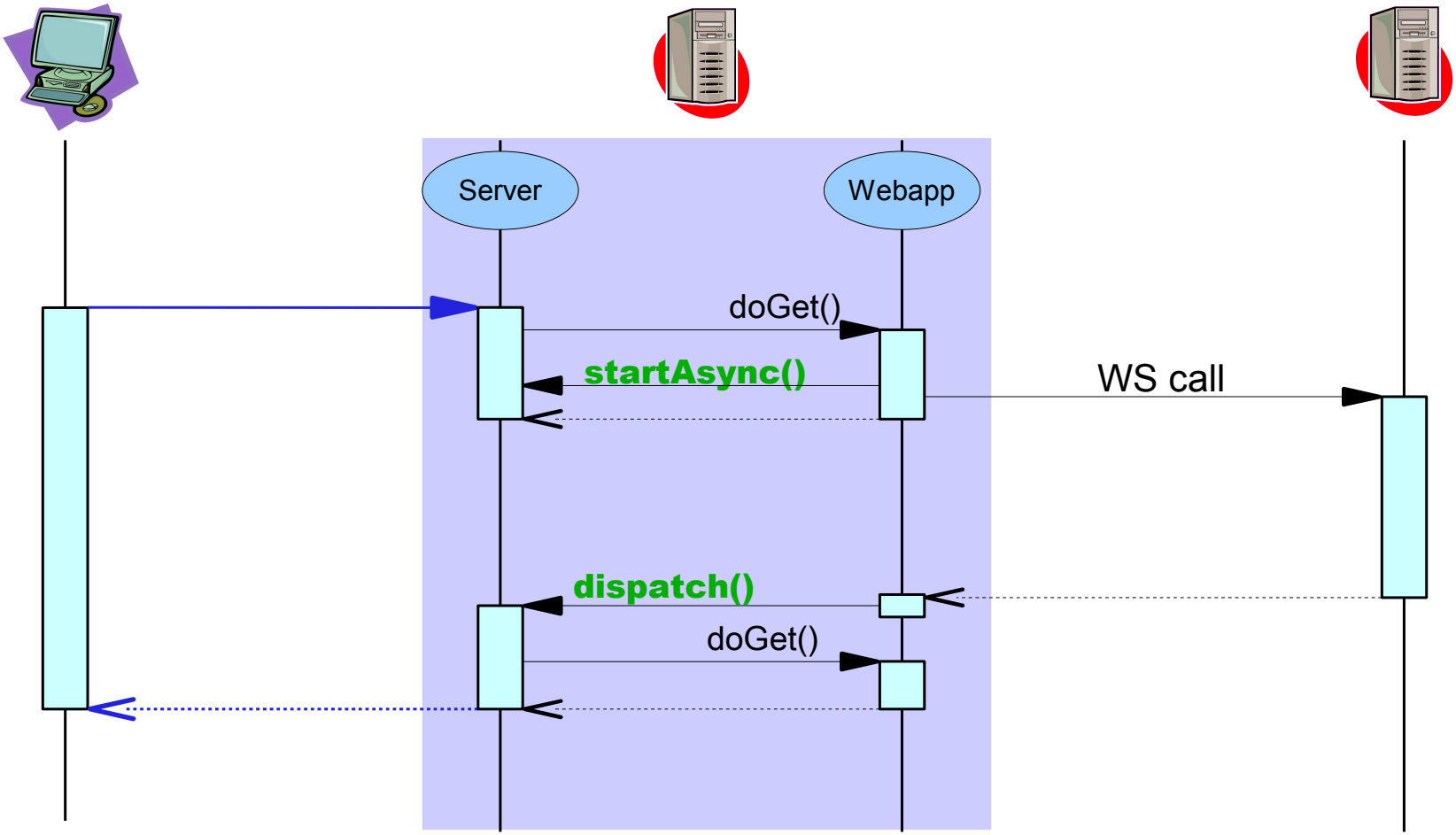
Asynch. Servlet APIs

- AsyncContext#dispatch(String path)
  - Dispatches the request and response objects of this AsyncContext to the given path scoped to the given context
  - The path parameter is interpreted in the same way as in ServletRequest#getRequestDispatcher(String path)
- AsyncContext#complete()
  - Completes the asynchronous operation that was started on the request that was used to initialize this AsyncContext, closing the response that was used to initialize this AsyncContext

Asynch. Servlet APIs: Event Handling

- AsyncContext#addListener(AsyncListener listener)
  - Registers the given AsyncListener
- AsyncListener#OnTimeout(AsyncEvent event)
  - Notifies this AsyncListener that an asynchronous operation has timed out
- AsyncListener#OnComplete(AsyncEvent event)
  - Notifies this AsyncListener that an asynchronous operation has been completed
- AsyncListener#OnError(AsyncEvent event)
  - Notifies this AsyncListener that an asynchronous operation has failed to complete

Asynchronous Web Service



Lab:

Exercise 1: Asynch. Servlet
Exercise 2: Chat application
4547_javaee6_servlet3.0_advanced.zip

Security Enhancements

New Security Annotations

- In Servlet 2.5
  - Only @DeclareRoles and @RunAs are supported in servlets
  - @DenyAll, @PermitAll, @RolesAllowed are only supported for EJBs
- In Servlet 3.0
  - @RolesAllowed -> auth-constraint with roles
  - @DenyAll -> Empty auth-constraint
  - @PermitAll -> No auth-constraint
  - @TransportProtected -> user-data-constraint
- Annotations enforced on *javax.http.Servlet* class and doXXX methods of HttpServlet
  - Method-targeted annotations take precedence over class-targeted annotations

Programmatic Login/Authentication/Logout

- HttpServletRequest#login(String username, String password)
  - Replacement for “Form Based Login” – removes the need of JSP-based login forms
  - Allows an application to collect user name and password information in any way it wants
- HttpServletRequest#logout
  - Allow an application to reset the caller identity of a request.
- HttpServletRequest#authenticate(HttpServletResponse)
  - Application initiates container mediated authentication from a resource that is not covered by any authentication constraints in web.xml
  - Application decides when authentication must occur
  - A login dialog box displays and collects the user name and password for authentication purposes.

Security Enhancements

- Security constraints in *web.xml* override security annotations
- <web-resource-collection> enhanced with <http-method-omission> to
  - Allow constraints to be specified on non-enumerable HTTP method subsets (i.e., all other methods)

Servlet 2.5 vs Servlet 3.0

```
<security-constraint>
    <display-name>WebConstraint</display-name>
    <web-resource-collection>
      <web-resource-name>test</web-resource-name>
      <description/>
      <url-pattern>/test.jsp</url-pattern>
      <http-method>POST</http-method>
      <http-method>HEAD</http-method>
      <http-method>PUT</http-method>
      <http-method>OPTIONS</http-method>
      <http-method>TRACE</http-method>
      <http-method>DELETE</http-method>
    </web-resource-collection>
    <auth-constraint>
      <description/>
      <role-name>niceguys</role-name>
     </auth-constraint>
  </security-constraint>
```

```
<security-constraint>
    <display-name>WebConstraint</display-name>
    <web-resource-collection>
      <web-resource-name>test</web-resource-name>
      <description/>
      <url-pattern>/test.jsp</url-pattern>
      <http-method-omission>GET
      </http-method-omission>
    </web-resource-collection>
    <auth-constraint>
      <description/>
      <role-name>niceguys</role-name>
     </auth-constraint>
  </security-constraint>
```

The above two are equivalent: Both indicate that the resource referenced by the url pattern */test.jsp*, when accessed by all the http-methods except GET, should be constrained to be viewed only by authenticated users belonging to the role *niceguys*

Lab:

Exercise 4: Security
4547_javaee6_servlet3.0_advanced.zip

Misc. Features

Session Tracking Cookie

- Session Tracking Cookie configuration
  - Via web.xml
  - Programmatic via *javax.servlet.SessionCookieConfig*
- Support for HttpOnly cookie attribute
  - *servletContext.getSessionCookieConfig().setHttpOnly(true)*

Servlet 3.0 File Upload

- Servlet 2.5
  - Servlet API does not have built-in support for file upload
  - Open source libraries such as Commons file upload and COS multipart parser were used
- Servlet 3.0
  - Out of the box support of file upload
  - *@MultipartConfig* annotation - when present on any servlet, it indicates that the servlet expects request of type multipart
  - Two new methods have been introduced to HttpServletRequest interface
    - public Collection<Part> getParts()
    - public Part getPart(String name)

Servlet 3.0 File Upload

```
@WebServlet("/upload.html")
@MultipartConfig(location="c:\\tmp", fileSizeThreshold=1024*1024,
                 maxFileSize=1024*1024*5, maxRequestSize=1024*1024*5*5)
public class FileUploadServlet extends HttpServlet {

    @Override
   protected void doPost(HttpServletRequest req, HttpServletResponse resp)
                        throws ServletException, IOException {
           resp.setContentType("text/html");
           PrintWriter out = resp.getWriter();

           Collection<Part> parts = req.getParts();

           out.write("<h2> Total parts : "+parts.size()+"</h2>");



}
```

Learn with Passion!
JPassion.com