Advanced Java Programming

Servlets

Chris Wong
cwh@it.uts.edu.au

Orginal notes by Dr Wayne Brookes

Topics

- Quick HTTP refresher
- Web application models
- Servlets
- Servlet API
- Servlets and Threading
- Sessions
- Request forwarding and including
- Filters
- Servlet/WLS issues
- Web Application deployment

Quick HTTP refresher

- HTTP is a TCPIP protocol used by the web
  - Defaults to TCP port 80

- Client (Web browser) requests web page from a server
  - Identifies source via a URL
  - Uses GET or POST requests
  - Also can send Headers eg: user-agent (browser)

- Server (Web server) responds file requested
  - Typically HTML or GIF/JPG

Web application models

Server can generate DYNAMIC web pages via:

- CGI
  - web server runs external program for processing

- Server APIs (ISAPI / NSAPI)
  - Web server loads library/modules eg: C, C++ DLL, LIB files

- Templates
  - server interprets file, executes operations.
  - Active Server Pages (.asp) eg: VBScript, Jscript
  - PHP, ColdFusion
  - Java Server Pages (.jsp)

- Servlets
  - web server contains JVM
  - this JVM runs Java code “inside” web server
Web app models - CGI

- Advantages:
  - Simple model
  - Reuse of existing command-line programs
  - Robust (crash in program does not affect web server)

- Disadvantages:
  - Very resource-intensive
    - New process created for each request
    - Memory/processor "expensive"
  - "Program-centric" coding
    - i.e. code that generates HTML - not "page-centric"

Web app models - server APIs

- Advantages:
  - Highly efficient
    - Native code running as part of web server
  - Access to internal processing of web server
    - Can build filters, etc.

- Disadvantages:
  - Complex coding
  - Not robust (crash in your code crashes web server)
  - Difficult to debug (concurrent threads)

Web app models - templates

- Advantages:
  - Page-centric
    - Usually have HTML code, with embedded code
  - Can be portable
    - JSP, PHP are de-facto standards
    - ASP, CFM are not

- Disadvantages:
  - Performance
    - Interpreted (but can be cached?)
  - Difficult to debug
    - Run-time errors in HTML?

→ We cover Java Server Pages next lecture
Web app models - servlets

- Advantages:
  - more efficient than CGI (threaded)
  - robust (crash in servlet causes no damage)
  - secure (servlet container can place security restrictions)
  - access to rich Java libraries (e.g., JDBC)
  - portable between platforms/vendors

- Disadvantages:
  - "program-centric" Java coding
    - Your code generates HTML
    - but see Java Server Pages (JSP) next week

Topics

- Quick HTTP refresher
- Web application models
- Servlets
- Servlet API
- Servlets and Threading
- Sessions
- Request forwarding and including
- Filters
- Servlet/WLS issues
- Web Application deployment

Servlet lifecycle

Servlet loaded - either:
- web server starts; or
- first request for servlet arrives

Servlet ready to take requests

service() HTTP request arrives

Web server decides to remove this instance of servlet

destroy()

Servlet may be garbage collected

Methods defined in javax.servlet.Servlet

J2EE & Servlets

Topics

- Quick HTTP refresher
- Web application models
- Servlets
- Servlet API
- Servlets and Threading
- Sessions
- Request forwarding and including
- Filters
- Servlet/WLS issues
- Web Application deployment
Servlet API

- Interfaces
- Abstract classes
- Exceptions

Package javax.servlet

- Generic servlet handling

Package javax.servlet.http

- Specific for web servlets

- For:
  - format of HTTP request/responses
  - Support for HTTP sessions
  - Support for HTTP cookies

Servlet API - inheritance

javax.servlet.GenericServlet extends javax.servlet.Servlet

javax.servlet.http.HttpServlet implements java.io.Serializable

MyServlet

The Servlet API

- The Servlet Interface
  - specifies the contract between the web container and a servlet
  - containers use this interface to reference servlets
  - implement this indirectly by extending
    - javax.servlet.GenericServlet or javax.servlet.http.HttpServlet

- Methods
  - void init(ServletConfig config)
    - called with ServletConfig parameter by the container
    - container calls this before any request - guaranteed
    - allows a servlet to load any initialization of parameters
    - done once only / not per request

- void service(ServletRequest req, ServletResponse res)
  - entry point for executing logic

- void destroy()
  - container calls this before removing a servlet
  - deallocate resources (specially non Java resources)

- ServletConfig getServletConfig()
  - return the ServletConfig that was passed to init()

- String getServletInfo()
  - return a String containing servlet information

Defining servlet classes

- Recall:
  - Java application:
    public class MyApp

  - Java applet:
    public class MyApplet extends javax.swing.JApplet

- Now:
  - Java servlet:
    public class MyServlet extends javax.servlet.http.HttpServlet

Servlet programming - theory

- To create a basic HTTP Java servlet, you must:
  - extend javax.servlet.http.HttpServlet
  - implement the service() method to handle each incoming request
    - should avoid this - why? see next slide first!
  - optionally implement init() and destroy() methods
Servlet programming – practice

- javax.servlet.http.HttpServlet provides a default implementation of the service() method
  - casts request/response to HTTP request/response
  - calls its protected service() method
  - service() uses getMethod() to call doXXX() method

- It makes calls to other methods, which **YOU** implement. The main two are:
  - doGet() - HTTP GET requests
  - doPost() - HTTP POST requests

Servlet example so far

```java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class MyServlet extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    // ... other code goes here ...
  }
}
```

Requests and responses

- HTTP is a request/response protocol

- Servlets need to:
  - access information sent as part of the request
  - create a response that is sent back to the client

- When the container calls service() it passes two objects as parameters, of types:
  - HttpServletRequest (extends ServletRequest)
  - HttpServletResponse (extends ServletResponse)

ServletRequest common methods

- int getContentLength() – number of bytes in the body of a request
- String getParameter(String name) – value correspond to name or null
- Enumeration getParameterNames() – names of parameters parsed out of the request
- String[] getParameterValues(String name) – for more than value associated with this name
- null if none associated

HttpServletRequest methods

- Cookie[] getCookies() – array of Cookie objects stored on the client
- String getQueryString() – query string present in the URL of GET request
- HttpSession getSession() – HTTPSession associated with this session
- String getHeader(String name) – value associated with the name; could be null
- Enumeration getHeaderNames() – an Enumeration for header names

ServletResponse methods

- void setContentType(String type) – MIME type
- PrintWriter getWriter()
- ServletOutputStream getOutputStream()

Recall Java I/O:

- Readers & Writers are for TEXT data
- InputStreams & OutputStreams are for BINARY data

- Thus use:
  - getWriter() for Content-Type text/* (e.g. text/html)
  - getOutputStream() for binary (e.g. image/gif)
**HttpServletResponse methods**

- **void setHeader(String name, String value)**
  - modify a value for name in the response

- **void sendRedirect(String location)**
  - redirects the user’s browser

- **String encodeURL(String url)**
  - see Session

- **void setError(int sc)**
  - sends default error page indicating the status code (dreaded 404)

- **void addCookie(Cookie cookie)**
  - adds a Cookie to the response

**Complete servlet example**

```java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class MyServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        res.setContentType("text/plain");
        PrintWriter out = res.getWriter();
        out.println("Hello servlet world!");
        out.close();
    }
}
```

**Using Enumerations**

```java
// Example to print out all headers
java.util.Enumeration e = req.getHeaderNames();
while (e.hasMoreElements()) {
    String hdrName = (String) e.nextElement();
    // Now we have the header name, get the header value
    String hdrValue = req.getHeader(hdrName);
    out.println(hdrName + " : " + hdrValue);
}
```

- Enumerations are common – be familiar with them!
  - they’re not specific to servlets though

**Topics**

- Quick HTTP refresher
- Web application models
- Servlets
- Servlet API
  - ➔ Servlets and Threading
- Sessions
  - Request forwarding and including
  - Filters
  - Servlet/WLS issues
- Web Application deployment

**Servlets and threading**

- By default, servlets are multi-threaded
  - Many clients could be executing the service() method simultaneously

- Beware of updating shared resources from within your service() / doGet() / doPost() method
  - need to ensure synchronized access

- Shared resources may include:
  - class variables (static)
  - open files

**Multi-threaded servlets**

```java
public class MyServlet extends HttpServlet {
    private static int numRequests = 0;
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        res.setContentType("text/plain");
        PrintWriter out = res.getWriter();
        out.println("Hello servlet world!");
        out.close();
    }

    private synchronized void incNumRequests() {
        numRequests++;
    }
}
```
**Single-threaded servlets**

```java
public class MyServlet extends HttpServlet
    implements SingleThreadModel {

    private static int numRequests = 0;
    public void doGet ( HttpServletRequest req,
        HttpServletResponse res)
        throws ServletException, IOException {
        numRequests++;
        // ... other code goes here ...
    }
}
```

**SingleThreadModel Interface**

- May send multiple requests from different threads
- `javax.servlet.SingleThread` marker interface
  - ensures that only one thread is executing `service()` method
- Approaches for `SingleThreadModel` servlets
  - instance pooling
    - maintains a pool of servlets
  - request serialization
    - container serializes requests
    - `DEPRECATED UNDER J2EE 1.4/5`

**SingleThreadModel Interface**

- SingleThreadModel is resource intensive
  - more object creation overhead
  - invoke `service()` method in a synchronized block
    - serialization hampers concurrent requests
- Avoid using `SingleThreadModel` in production
  - in general, allowing your servlet to handle multiple simultaneous requests is a good thing → scalability
  - however, `SingleThreadModel` can help when debugging
    - if you think simultaneous access to shared data may be the source of your bug(s)

**Thread-Synchronization Issues**

- Best Practice: Reduce the scope
  - `doGet()` is too wide
  - explicitly use `synchronized` keyword
- Should be aware of implications
  - potentially blocking a container thread
  - Eg: JDBC calls

**Topics**

- Quick HTTP refresher
- Web application models
- Servlets
- Servlet API
- Servlets and Threading
- Sessions
- Request forwarding and including
- Filters
- Servlet/WLS issues
- Web Application deployment

**Sessions**

- Many web sites use session-based interactions:
  - Example: web-based application
    - log in
    - use web-based application
    - log out
  - Example: e-commerce
    - browse online store
    - add items to shopping cart
    - check out and pay
- The web architecture does not support this
  - HTTP is a stateless protocol
Sessions

• Tracking Users
  - Cookies
  - URL rewriting

• Cookies invented by Netscape

Respond with a Token

Client

Request with a Token

Server

Sessions and Cookies

• Cookies
  - a small piece of text sent by the server to the client
    • stored on the client
    • sent by the client with all requests
  - Example in HTTP header:
    • Set-cookie: uid=sm; Max-age=3600; Domain=".uts.edu.au"; Path="/"
    • missing max age?
    - discards the cookie when exit
  - browser sends request header (uid=sm)
  - browser stores cookie against the domain and URL path

Sessions and URL rewriting

• URL rewriting:
  - way of dealing with clients who do not accept cookies
  - adding a session identifier to the end of the URL
  - this id is passed along with clicks on any links
    http://xxx.xxx/index.html?jsessionid=f245bfe0de66a110c
  - URL rewriting cannot be enforced on static HTML pages

• Sites often use both cookies & URL rewriting
  - if cookies not supported, revert to URL rewriting

• Only lasts for the scope of the session

Sessions and Cookies

• RFC 2109 recommends
  - at least 300 cookies with 4096 bytes per cookie
  - 20 cookies per site
  - limited browsers can store 20 cookies

• Browser to keep the cookie or discard it

• Decision to send cookie with a request
  - the host must match
  - path on the host must match the path on the cookie
  - date time limit not expired

Sessions Tracking with HttpSession

• Cookies store session information on the client
  - privacy issues
  - limit
• Session tracking keeps information on the server
• Every session is given a unique session id

• Technically (WLS specific)
  - WLS creates temporary cookies
  - this cookie is sent along with each request
  - WLS maps the session id to the HttpSession object

• HttpSession allows you to store named attributes associated with the current session
Session Tracking with HttpSession

• Accessing the current session
  HttpSession s = req.getSession();

• Storing session data
  ShopCart cart = new ShopCart();
  // ShopCart is my own class
  s.setAttribute("usercart", cart);

• Retrieving an object from a session
  ShopCart mycart = (ShopCart) s.getAttribute("usercart");

• Setting the expiry time on a session
  s.setMaxInactiveInterval(1800); // 30 minutes

Sessions and cookies

• Using HTTP sessions is more common than using cookies
  - cookies are more low-level
• Sessions are active for a short period of time
• Disposes of HTTPSession object once it “timed-out”

Use Sessions for:
  - tracking user shopping cart
  - caching data that user might look more than once
  - Cookies for:
    - recognize a user without requiring to log on

Topics

• Quick HTTP refresher
• Web application models
• Servlets
• Servlet API
• Servlets and Threading
• Sessions
  - Request forwarding and including
  - Filters
  - Servlet/WLS issues
• Web Application deployment

Request dispatching

• Sometimes in a servlet, you might want to:
  - Redirect (forward) a request to another page
    - maybe to a servlet, JSP or HTML page
  - Include the contents of a page in your servlet’s output
    - maybe include a static page, or the output of another servlet
  - A RequestDispatcher provides these facilities

Using “forwarding”

public void doGet (HttpServletRequest req,
  HttpServletResponse res) throws ServletException, IOException {
  ServletContext ctx = getServletContext();
  RequestDispatcher dispatcher = ctx.getRequestDispatcher("/help.html");
  if (dispatcher != null) {
    dispatcher.forward(req, res);
  return;
  }
}
Notes on forwarding

- The forwarding is done at the server-side, behind-the-scenes.

- Can only forward if your servlet has not generated output to the browser:
  - IllegalStateException thrown if output has been sent
  - You could conditionally forward depending on the value of one of the input parameters

- Classic Example:
  - redirect a user to login page if user's session is invalid

Using "including"

```java
public void doGet (HttpServletRequest req,
                HttpServletResponse res)
    throws ServletException, IOException {
    ServletContext ctx = getServletContext();
    RequestDispatcher dispatcher =
        ctx.getRequestDispatcher("/head.inc");
    if (dispatcher != null) {
        dispatcher.include(req, res);
        return;
    }
}
```

Notes on including

- Code is nearly identical to forwarding:
  - except call RequestDispatcher.include()

- Including may be done anywhere in your servlet:
  - even if some output has already been generated

Topics

- Quick HTTP refresher
- Web application models
- Servlets
- Servlet API
- Servlets and Threading
- Sessions
- Request forwarding and including
  - → Filters
  - Servlet/WLS issues
  - Web Application deployment

Filters

- In Servlet specification v.2.3; J2EE 1.3
- You can filter inbound HTTP requests
- Only limited to Web container
- Similar to servlet - container managed object
- Can customize how HTTP requests are handled

- Uses:
  - validate HTTP requests
  - log HTTP requests
  - authorize HTTP requests
  - content management
  - custom HTTP environment for servlets and JSPs
Filter example

```java
public class MyFilter implements Filter {
    public void init(FilterConfig config) {
        ...
    }
    public void doFilter (ServletRequest req, ServletResponse res, FilterChain chain) throws ServletException, IOException {
        // ... other code goes here ...
        chain.doFilter(request, response);
    }
    public void destroy() {
        ...
    }
}
```

Topics

- Quick HTTP refresher
- Web application models
- Servlets
- Servlet API
- Servlets and Threading
- Sessions
- Request forwarding and including
- Filters
- Servlet/WLS issues
- Web Application deployment

Servlet/ Weblogic Issues

- Define the servlet as a public class
- Use invalidate() to cancel a session (eg: logout)
- Session consumes resources!
  - keep inactive intervals short!
- Cookies are not appropriate for sensitive data
  - use database!
- Always activate URL rewriting
  - To deal with paranoid users who turn off cookies

Web applications

- A "web application" consists of:
  - static HTML files
  - static CSS stylesheet files
  - static image (GIF, JPEG) files
  - any other static files
  - Java servlets
  - JSP files (Java Server Pages)
  - Java class libraries (typically in JAR files)
  - a deployment descriptor (web.xml) describing the application
Servlet deployment

- Servlets may be deployed as either:
  1. Individual files (aka “exploded directory”)
  2. A WAR file – Web Application Archive

1. Individual files good for debugging
   - Don’t have to constantly rebuild/redeploy WAR file

2. WAR file good for production
   - WAR files are portable between J2EE-compliant web servers

Sample deployment directory

/index.html
/aboutus.html
/mystyles.css
/images/logo.gif
/images/banner.gif
/WEB-INF/web.xml
/WEB-INF/classes/MyServlet.class
/WEB-INF/classes/MyFilter.class
/WEB-INF/lib/jstl.jar

WAR files

- WAR files are special zip archives
  - Represents a virtual directory
  - Looks like previous slide

- Use jar command from Java SDK to create them
  - Eg:
    cd src
    jar cvf ../myApp.war *

Deployment descriptor

- web.xml file is a deployment descriptor
  - Tells the web server about this web application
    - A key to the portability of WAR files between servers

- May contain:
  - Name of the application
  - Welcome file for the application (if not index.html)
  - Error documents to display (e.g. in case of 404)
  - List of servlets in this application
  - How to map from URL requests to servlet invocation

WEB-INF

- It is case-sensitive and contains:
  - /WEB-INF/web.xml
    - Deployment descriptor file – required!
  - /WEB-INF/classes/*
    - Java class files for servlets, filters and utility classes
  - /WEB-INF/lib/*.
    - Java Archive (JAR) files containing classes
### web.xml without servlets

```xml
<web-app>
  <display-name> My First Application </display-name>
  <welcome-file-list>
    <welcome-file> start.html </welcome-file>
    <welcome-file> index.html </welcome-file>
  </welcome-file-list>
  <error-page>
    <error-code> 404 </error-code>
    <location> /404.html </location>
  </error-page>
</web-app>
```

This is the default web pages to display. Can also be servlet-mapping or JSP.

You can define your own error pages here (eg: 404 not found)

**This is the default web pages to display.**

Can also be servlet-mapping or JSP.

You can define your own error pages here (eg: 404 not found)

### web.xml with one servlet

```xml
<web-app>
  <servlet>
    <servlet-name> MyFirstServlet </servlet-name>
    <servlet-class> MyServlet </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name> MyFirstServlet </servlet-name>
    <url-pattern> /home/Welcome </url-pattern>
  </servlet-mapping>
</web-app>
```

Repeat `<servlet>` & `<servlet-mapping>` block for each servlet

### web.xml with a filter

```xml
<web-app>
  <filter>
    <filter-name> MyFirstFilter </filter-name>
    <filter-class> MyFilter </filter-class>
  </filter>
  <filter-mapping>
    <filter-name> MyFirstFilter </filter-name>
    <url-pattern> *.html </url-pattern>
  </filter-mapping>
</web-app>
```

This filter runs MyFilter on EVERY html file requested.

This filter runs MyFilter on EVERY html file requested.

### Servlet initialisation parameters

```xml
<web-app>
  <servlet>
    <servlet-name> MyFirstServlet </servlet-name>
    <servlet-class> MyServlet </servlet-class>
    <init-param>
      <param-name> favecolour </param-name>
      <param-value> Blue </param-value>
    </init-param>
  </servlet>
</web-app>
```

init-param’s can be retrieved in your servlet:

```java
String colour = getInitParameter("favecolour");
```

### Actual deployment

- As individual files: just copy your files into the correct directory
  web server checks periodically for file modifications and will reload the servlet classes when changed

- As a WAR file: dependent upon the web server!
  WebLogic only:
  - copy the WAR file into autodeploy directory or-
  - choose ‘Install a new Web Application’ in the WebLogic server console
  - uploading a new WAR file overwrites the previous one

### Summary

- Servlets are for presentation logic – create HTML
  efficient yet powerful mechanism

- API is relatively simple once you get used to it

- Application support in the form of:
  - Session management
  - Request forwarding and inclusion

- Portable deployment of web applications
  through WAR files and deployment descriptors
References

• Official reference (Sun)

• Specific to WebLogic:
  - http://edocs.bea.com/wls/docs100/webapp/

• API reference
  - http://java.sun.com/products/servlet/2.4/javadoc/