Java Servlets

Servlet Components and Containers

Servlet Basic API

Servlet Deployment and Configuration

Wrap Up

SERVLET COMPONENTS AND CONTAINERS

Java Servlets

A server-side *component* which responds to requests

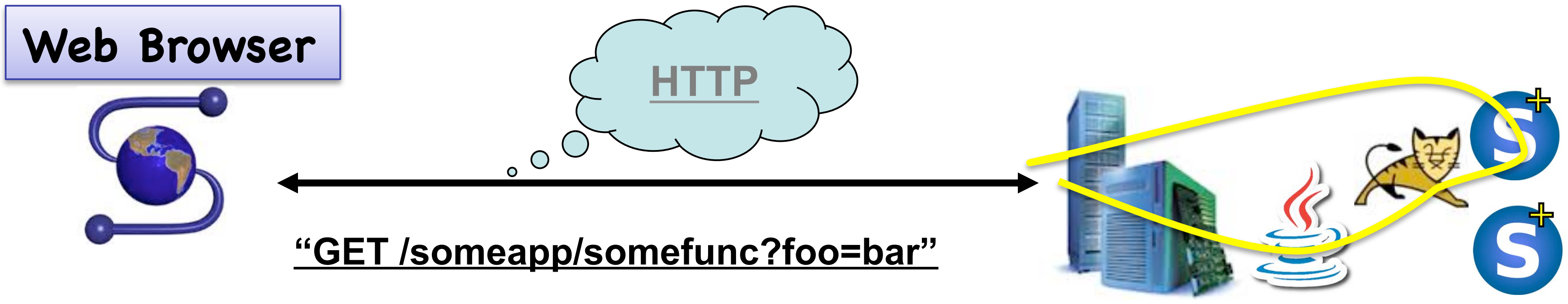
- Servlets introduced a fundamentally new way to write web apps
  - Server-side “components” that run inside a Java “container”
  - JVM process sticks around, as can objects
  - Specification/API formalized component/container model, provided class definitions, and supported deployment models.

Any Java class that implements the Servlet interface

- Has full access to Java APIs and other support classes

Another form of component

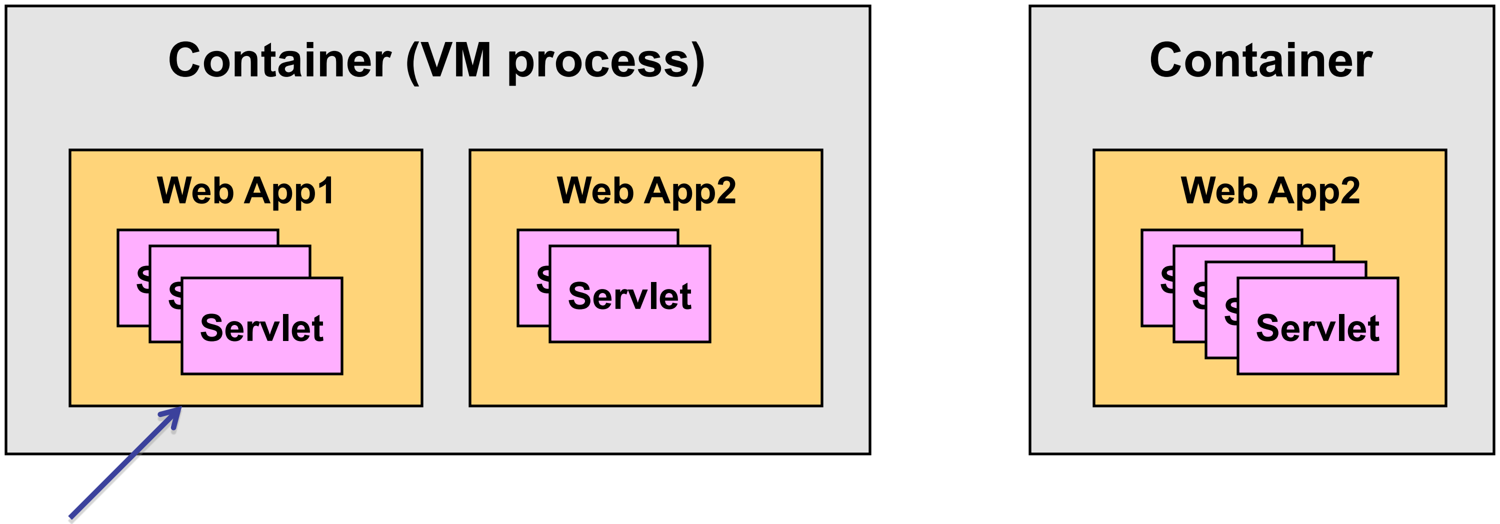
- Resides within a server-side application container
- Receives information through parameters or a stream/reader
- Returns MIME-typed result(s) through a stream/writer



Container-Managed Applications

Servlet applications run inside a container

- Container is multi-threaded and processes requests concurrently
  - Servlets must be thread-safe
- Applications within a container share processes resources
  - Single rogue servlet can cause process to fail
  - Example of thread efficiency vs. process protection
  - **Remember:** 1 application may have many components (servlets)

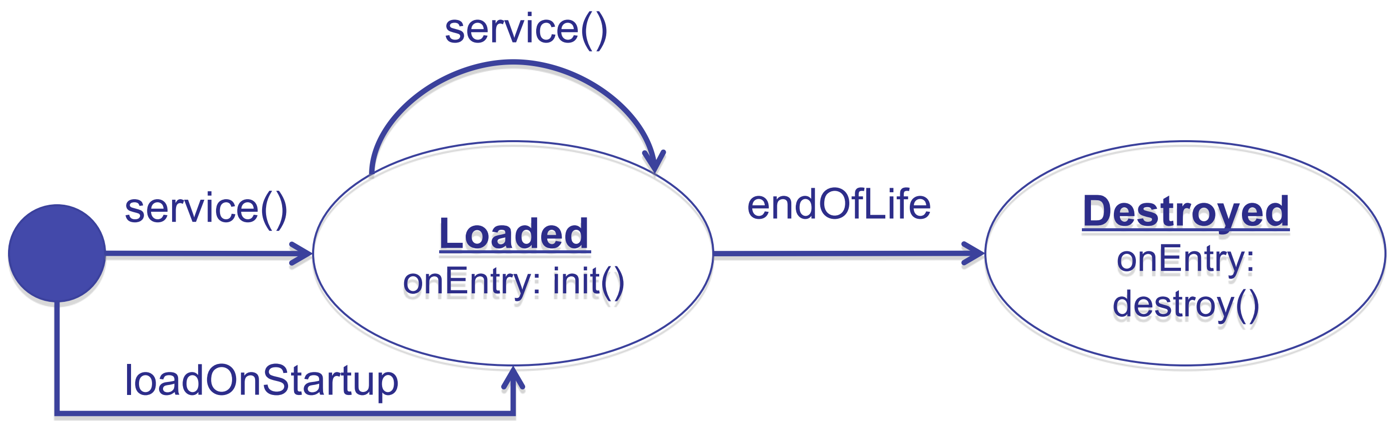


Web App also called the "Context"

Servlet Lifecycle

One instance of each servlet exists per container

- Servlet instances must be thread-safe



endOfLife here means the servlet container may choose when to terminate the lifecycle of this servlet instance (unload the class)

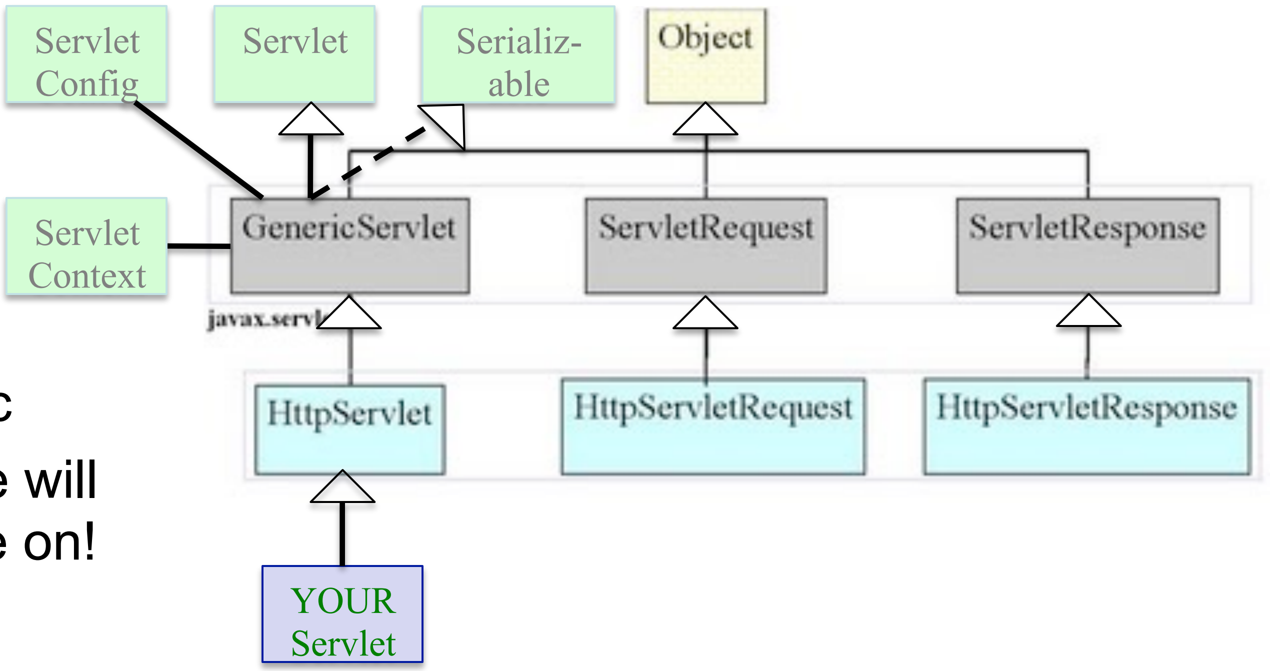
javax.servlet.Servlet defines lifecycle methods

- This means you do not write a "main" method
- **`init(ServletConfig config)`**
  - called once to notify servlet it is going to be utilized
- **`destroy()`**
  - called when servlet is being de-allocated

SERVLET BASIC API

The Servlet Class Hierarchy

- Top are *interfaces*
- So is ServletContext
- Gray boxes are the base class types



- Blue are HTTP specific
- These are the ones we will spend most of our time on!
- Your servlet extends HttpServlet

The servlet container dispatches requests to a component implementing the *Servlet* interface

- *GenericServlet* is provided as a default implementation of that interface, must like many GUI libraries have a base impl of a widget
- HttpServlet is a specialization allowing delegation to protocol-specific methods
    - The only protocol-specific one in the spec. But you could have *FTPServlet*

Servlet Example

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

public class DateServlet extends HttpServlet   {

    public void doGet( HttpServletRequest req,
                       HttpServletResponse res)
                    throws ServletException, IOException    {

          res.setContentType("text/html");

          PrintWriter out = res.getWriter();

          out.println("<html>");
          out.println("<head><title>First Servlet</title></head>");
          out.println("<body><H2>Current time is</H2>");
          out.println( (new Date()).toString() );
          out.println("</body></html>");
    }
}
```

Import these packages from servlet-api.jar

Your servlet must extend HttpServlet

Processes HTTP GET requests

These encapsulate HTTP request/response

Need these exceptions

Set the Content-type of our response header

Get an output stream to send output to from the response object

Servlet Methods

The javax.servlet.http.HttpServlet defines the set of processing methods for HTTP servlets

- doGet – responds to HTTP GET requests
- doPost – responds to HTTP POST requests
- doPut – responds to HTTP PUTrequests
- doDelete - responds to HTTP DELETE requests
- doHead - responds to HTTP HEAD requests

HttpServlet extends javax.servlet.GenericServlet

- Servlets can be used to handle requests on other application protocols, not just HTTP (SMTP, POP, FTP supported)
- GenericServlet includes several convenience methods that interact with the servlet container *(…stay tuned)*
- The *service()* from GenericServlet should not be overridden, it is a dispatcher to your above *doXXX* methods.

Handling HTTP Request and Responses

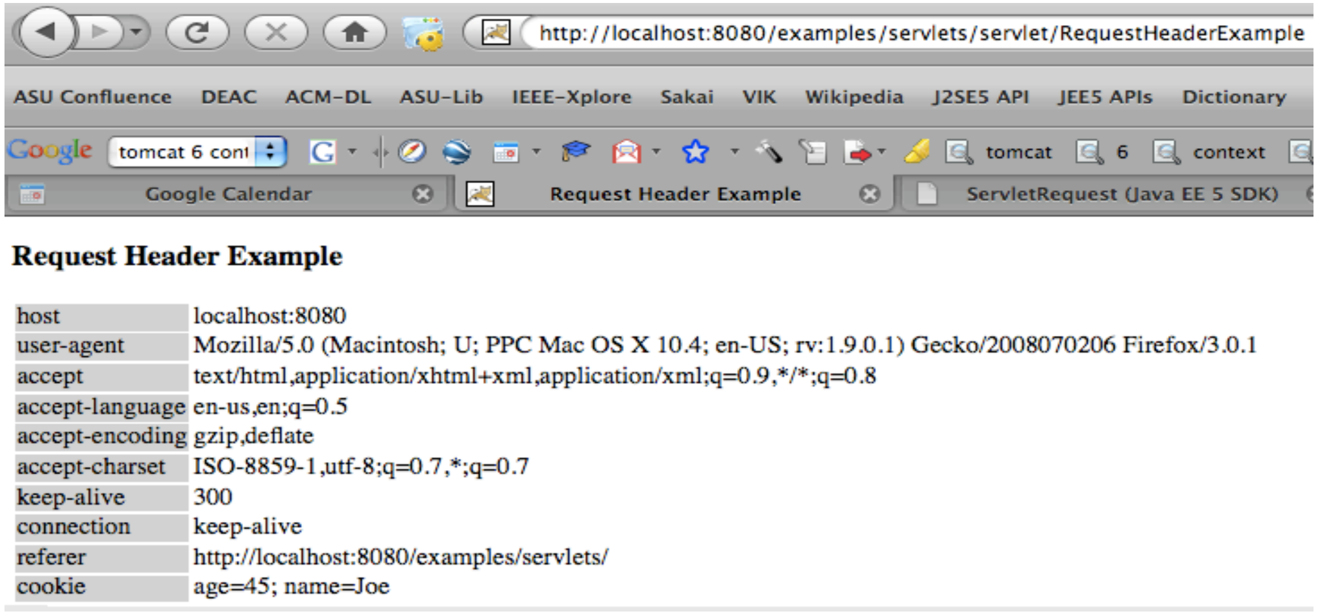
HttpServletRequest holds information from client

▪ getParameter()	returns a passed parameter
▪ getParameterMap()	returns a map of all parameters
▪ getHeader()	returns an HTML header value

HttpServletResponse object represents the HTTP response

- addHeader() adds a name/value pair to the response header
- containsHeader() checks to see if the header has been set
- encodeURL() encodes the URL
- encodeRedirectURL() – used in conjunction with sendRedirect
- sendError() sends an error with a specific code
- setStatus() sets the status code of the response
- sendRedirect() sends a temporary redirect to the client (302)

RequestHeadersExample


http://localhost:8080/examples/servlets/servlet/RequestHeaderExample
ASU Confluence DEAC ACM-DL ASU-Lib IEEE-Xplore Sakai VIK Wikipedia J2SE5 API JEE5 APIs Dictionary
Google tomcat 6 conf tomcat 6 context
Google Calendar
Request Header Example
ServletRequest (Java EE 5 SDK)

Request Header Example

host	localhost:8080
user-agent	Mozilla/5.0 (Macintosh; U; PPC Mac OS X 10.4; en-US; rv:1.9.0.1) Gecko/2008070206 Firefox/3.0.1
accept	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
accept-language	en-us,en;q=0.5
accept-encoding	gzip,deflate
accept-charset	ISO-8859-1,utf-8;q=0.7,*;q=0.7
keep-alive	300
connection	keep-alive
referer	http://localhost:8080/examples/servlets/
cookie	age=45; name=Joe

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class RequestHeaderExample extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
                                            throws IOException, ServletException    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        Enumeration e = request.getHeaderNames();
        while (e.hasMoreElements()) {
            String name = (String)e.nextElement();
            String value = request.getHeader(name);
            out.println(name + " = " + value);
}    }    }
```

Servlet Generic Code Idiom

Servlet template:

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

public class XXXServlet extends HttpServlet {
  public void init(ServletConfig cfg) throws ServletException
    { <any code to execute when servlet is loaded into JVM> }

  public void doGet(HttpServletRequest rq, HttpServletResponse rs)
    throws ServletException, IOException  // or doPost
    {
        <process request headers>
        <process request>
        <perform processing>
        <aggregate response payload>
        <set Content-type and other response headers>
        <write out results>
    }
  public void destroy() {}  // not typically implemented
  public String getServletInfo() { return "XXX"; }
}
```

SERVLET DEPLOYMENT AND CONFIGURATION

Container configuration

Server.xml configures the container for Tomcat

- In Tomcat, the container is an “Engine”
  - You can configure resources (loggers, database pools) at this level
- What is a “Service” then?
  - The combination of the Engine with the Connectors – the ways you can access the container!
  - You can devote thread pools to requests coming from a given Connector
- You can optionally configure a *context* for your web application
  - If you do not define one then Tomcat automagically creates one itself

```
<Service name="Catalina”>
 <!-- Some stuff cut out here… -->
 <Connector port="8080" protocol="HTTP/1.1” connectionTimeout="20000”redirectPort="8443”/>
 <Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true” maxThreads="150"
                scheme="https" secure="true” clientAuth="false" sslProtocol="TLS" />

 <!-- Define an AJP 1.3 Connector on port 8009 -->
 <Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />

 <!- Catalina is the engine Tomcat runs for the JVM -->
 <Engine name="Catalina" defaultHost="localhost">
    <Host name="localhost"  appBase="webapps” unpackWARs="true" autoDeploy="true”
              xmlValidation="false" xmlNamespaceAware="false”>
    </Host>
 </Engine>
</Service>
```

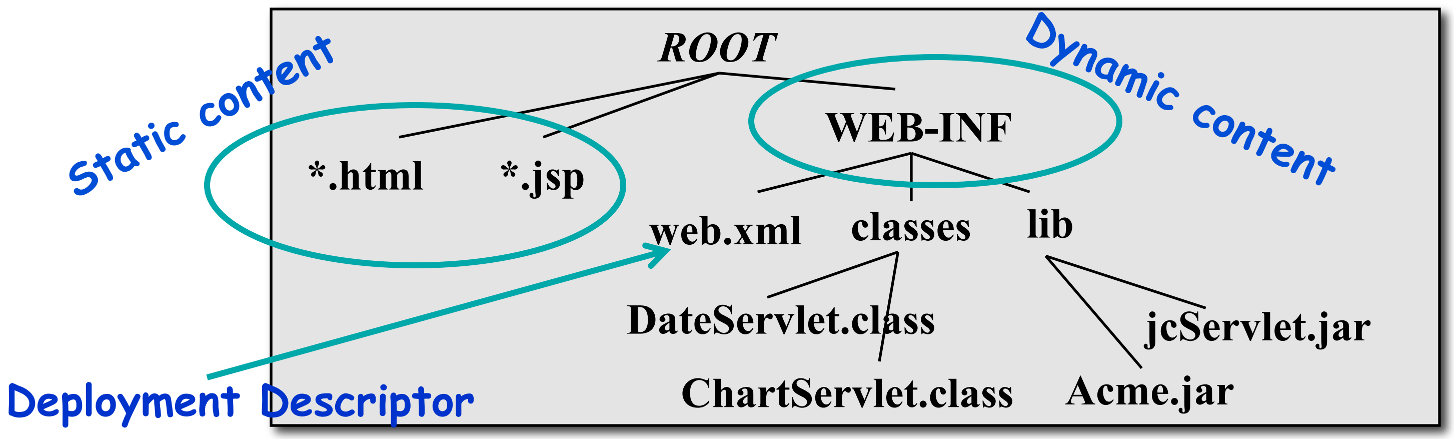
Web Application Deployment

A Web application consists of

- Static content (resources): HTML, images, stylesheets, Javascript, etc.
  - URL mapping based on file location on server
- Dynamic content – code generating responses
  - URL mapping defined in *deployment descriptor*
- Deployment descriptors
  - Describe a deployment package using a standard XML format for portability

Web Application Archives (WAR files)

- Enables consistent deployment across servlet engines
- A jar file with the following structure:



Web.xml

web.xml: placed in $TOMCAT_HOME/webapps/<context>/WEB-INF

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <display-name>Date Servlet</display-name>
  <description>Sample for ser422</description>

  <servlet>
    <servlet-name>DateMe</servlet-name>
    <servlet-class>edu.asupoly.ser422.servlets.DateServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>DateMe</servlet-name>
    <url-pattern>/date</url-pattern>
  </servlet-mapping>
</web-app>
```

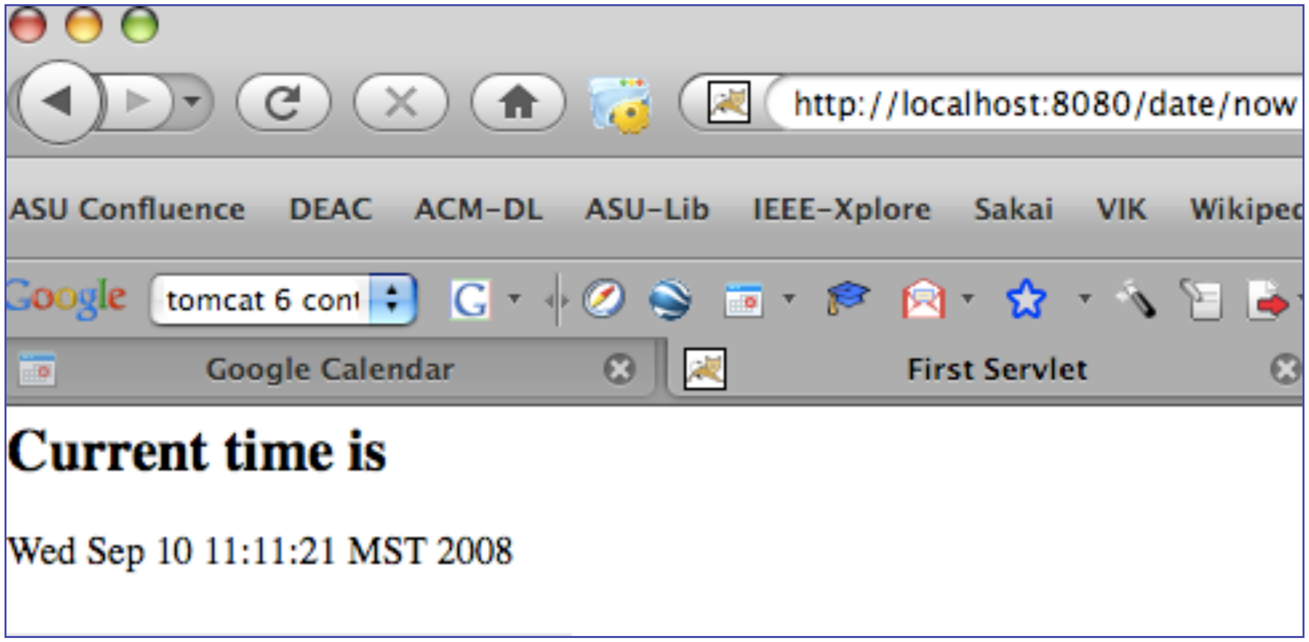
Servlet block associates the DateServlet class with logical name DateMe

Servlet-mapping block says all URLs *to this context* that begin with date will be mapped to this servlet.

Executing the servlet

So what is the final URL?

- http://hostname:port/<context>/<servlet-mapping-URL>
  - Where <context> is the directory name of your webapp/jar, and
  - <servlet-mapping-URL> matches the <url-pattern> from the <servlet-mapping> block in your web.xml

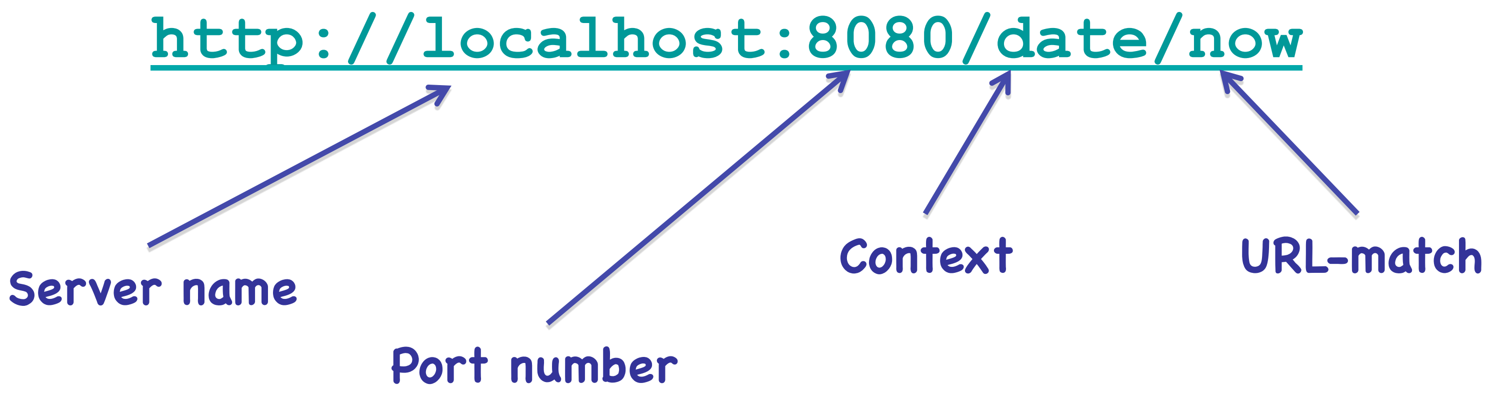


Current time is

Wed Sep 10 11:11:21 MST 2008

URL Mapping

Let’s parse that URL:



Components:

- Server name we know is the DNS name of the host
- Port number – defined in server.xml, Tomcat-specific
- Context – defined by Tomcat to be the directory name under $CATALINA_BASE/webapps
- URL-match – defined by the servlet-mapping block in web.xml

```
<servlet-mapping>
   <servlet-name>DateMe</servlet-name>
   <url-pattern>/now/*</url-pattern>
 </servlet-mapping>
```

ServletConfig Information

ServletConfig contains parameters from web.xml

```
<servlet>
  <servlet-name>MyDateServlet</servlet-name>
  <servlet-class>DateServlet</servlet-class>
  <init-param>
    <param-name>format</param-name>
    <param-value>yyyy.MM.dd-hh:mm.ss</param-value>
  </init-param>
</servlet>
```

```
protected SimpleDateFormat format = null;  // state variable in servlet

public void init(ServletConfig config) throws ServletException {
    super.init(config);
    String dateFormat = config.getInitParameter("format");
    if (dateFormat != null) {
        format = new SimpleDateFormat(format);
    } else {
        format = new SimpleDateFormat();
}
```

ServletContext

Recall, the “context” is what the servlet runs inside of

- This object, accessible on each request, is shared across all servlets that exist within a single context
  - Because it represents the view the servlets have of the container!
- Your servlet gets it by calling `getServletContext()`

What is useful in the ServletContext object?

- `getInitParameter[Names]` – gets setup info from web.xml
- `get[set]Attribute` – Allows name-value pairs to be shared across servlet instances within the same context.
- `getResource[AsStream]` – allows the app developer to access local resources within the context

Context best practices

- Do not to modify initial parameters programmatically in the context
  - It was created on init, you cannot dependably be sure when other servlets in your context might get/set information
- Do use it to obtain paths to resources, instead of absolute paths

ServletContext Example

Classic examples: reading data files

- Problem: Suppose you have app-specific data in a file
  - Typically, this info is in a file (XML or binary or text or…)
- Solution: Use the ServletContext to get it as a resource

```
// web.xml
  <servlet>
    <servlet-name>Phonebook</servlet-name>
    <servlet-class>edu.asupoly.ser422.PhoneServlet</servlet-class>
    <init-param>
     <param-name>phonebook</param-name>
     <param-value>/resources/phonebook.txt</param-value>
  </init-param>
</servlet>
```

```
// in init
public void init(ServletConfig config) throws ServletException {
  // if you forget super.init your getServletContext() will get a NPE!
  super.init(config);
  _filename = config.getInitParameter("phonebook");
```

```
// in doGet
ServletContext sc = getServletContext();
InputStream is = sc.getResourceAsStream(_filename);
```

WRAP UP

Servlets and Thread Safety

Remember (*always*!) that a servlet has a single instance in its context's JVM at any given time!

- This means your servlets must be thread-safe!

Thread-safety and servlets

1. Do **not** use servlet class member variables
   - Not thread-safe
2. Do **not** use synchronized blocks
   - These can cause bottlenecks or deadlocks among threads
   - Do not help in a horizontally scaled environment
3. **Do** be careful about objects reference-able from many servlet threads
   - These may need to be made thread-safe too

Summary

Java Servlets (at the time) represented the next evolutionary step in Web application architectures

1. CGIs as a separate process
2. Server-side scripts as an embedded process
3. JVM as a separate component-container process
    - The next evolutionary steps on the server-side after this included asynchronous high-throughput architectures (NodeJS, but servlets now include asynch support too), and API-driven apps (stay tuned…)

Pros and Cons:

Pros:

- Significant install base
- Lots of trained folks with lots of tools and lots of frameworks
- Has been shown to be secure and scale for heavy computational apps

Cons:

- Operations staff has always had a hard time supporting
- Code can be quite verbose, bloated, and “boring”
- Does not easily play well with client-side focus