When the Servlet Model Doesn't Serve

Gary Murphy
Hilbert Computing, Inc.
glm@hilbertinc.com
Motivation

- Many decision makers and programmers equate Java with servlets
  - Servlets are appropriate for a class of applications
  - Servlets are inappropriate for other classes of applications
- I have seen customers that chose an inappropriate application architecture
  - Present experiences and why servlets were or were not chosen when I was involved
Servlet Strengths

- No mass software distribution to clients
  - Major cost of client/server vs. terminal-based applications
- Centralized administration
- Economies-of-scale in deployment on large servers
- Easy user-interface development for simple forms-based applications
Servlet Weaknesses

- Considerably added complexity for sophisticated user-interfaces
- (Mostly) single-threaded programming model
- User-driven event model
- Application lifecycle management
- ... more on these issues in the rest of the presentation.
- Servlets do very well what they were designed to do...
Overcoming Weaknesses

Various technologies and products have been produced to overcome these weaknesses...

- Macromedia Flash
- Javascript
- Ajax

... these can be effective in overcoming the weaknesses of the servlet model, but it also illustrates that there is “tool abuse” going on.

- Perhaps some of these applications shouldn't be servlets!
Servlet Specification

- The servlet specification describes the primary characteristics that must be present in container implementations.
  - The programming model mandated by the specification may create issues for certain types of applications
- First, key points from the servlet specification for v2.3 that will be used in this discussion...
  - There are subtle implications in some of the specs
  - This is a little dry, but stick with me!
 Servlet Lifecycle

- Container instantiates the applications (servlets)
  - possibly when the container itself starts
  - possibly when the container determines the servlet is needed to respond to a request
- Servlet is not initialized until `init()` is called.
  - Be careful in the use of static class initializer code. This can be invoked before the servlet is initialized.

Servlet Spec v2.3: SRV.2.3.1 SRV.2.3.22
A servlet container can take a servlet out of service at any time

- All threads running in the `service()` method are completed before `destroy()`
- New requests are serviced on a new instance of the servlet

Servlet Spec v2.3: SRV2.3.4
Servlet Lifecycle - Reloading

- Container providers are not required to implement class reloading, but if they do...
- Implementation of class reloading must ensure all servlets and classes they use are loaded in the scope of a single class loader

Servlet Spec v2.3: SRV.3.7
Session Lifecycle

- HTTP offers no explicit termination indication for a session
  - Sessions are terminated via a timeout mechanism
  - Can be disabled via `setMaxInactiveInterval` to -1

- For distributed containers:
  - All session data must have specific support for distribution or the object must be `Serializable`
  - Container may not use JVM serialization. Can't depend on `readObject()`/`writeObject()` being called.

Servlet Spec v2.3: SRV.7.5 SRV.7.7.2
Request Lifecycle

- ServletRequest/ServletResponse objects are valid within the scope of service() or doFilter().

- Programmers should not keep references to these objects

Servlet Spec v2.3: SRV.4.10
Filters

- Filters were introduced in v2.3 of the spec
  - They allow the request data to be modified (via wrapping) before being passed to the servlet
  - The response data may be altered (e.g. XSL transforms) before being sent to the client
- Only one instance is instantiated for each filter declaration in the deployment descriptor
  - Programmers must ensure that filters are thread safe

Servlet Spec v2.3: SRV.6.2.1
Listeners

- Servlet context events:
  - Lifecycle events: has been created or is about to be terminated
  - Attribute events: attributes in the servlet context have been added, removed or replaced

- Session events:
  - Lifecycle events: session is created, invalidated or timed out
  - Attribute events: added, removed or replaced

Servlet Spec v2.3: SRV.10.2.1
Listeners

- Attribute changes to the servlet context and session objects may occur concurrently.
  - The container is not required to synchronize listener notifications
  - Listener implementations should handle this case explicitly through synchronization mechanisms

Servlet Spec v2.3: SRV.10.5
Threading Model

- Servlet containers implement concurrency by enabling multiple requests through the `service()` method of the servlet
  - For a given request, the process is typically synchronous and single-threaded
  - Synchronization issues occur only within objects available to multiple concurrently executing requests

Servlet Spec v2.3: SRV2.3.3.1
Situations Contrary to Servlets

- Some requirements for applications have suggested other deployment architectures than a servlet
- This will look at some of these issues and:
  - Look at why servlets are not appropriate
  - Look at what approaches for applications can be used instead of a servlet container model
Event/Time Driven Input

- Servlets are based heavily on the HTTP request lifecycle.
  - Typically, requests are initiated via a browser and returned to the browser in the form of HTML or XML.

- Some applications are driven by external events in addition to user-input events.
  - Air Force application was driven by troop movement events and military equipment movement events.
  - The integrated combat portal allowed a comprehensive view of the current battlefield environment.
Event/Time Driven Input

- Event information may be received via HTTP, but the user interface is not updated until the user refreshes the browser view
  - Real time notification applications have a mismatch with many monitoring applications
Network Protocol Mismatch

- Applications may need to interface with existing networked applications.
- Servlets are modeled after HTTP protocols, which are stateless.
- Existing network applications may model a user session on a network session. For example:
  - Applications may interface via telnet protocol
  - Applications may have extended conversations with other computers such as a mobile device
Batch Jobs

- Batch processes are typically single-threaded, long running applications
- Batch applications should not be run in a servlet container
  - The container can manage the lifecycle of the application
  - If the container takes the application out of service, the servlet request, running as batch, is subject to termination via timeout
Mixed Response Time Tasks

Applications may support a variety of functions with variable response times:

- Servlets have a thread pool that process synchronous events in step with the HTTP protocol.
- Applications may have long running subtasks (e.g. generating a report or recompiling a workspace) that should not block the user-interface.
- For example, O'Reilly Safari
Control Over Lifecycle

- Your application may need to spawn a thread to:
  - Listen on a socket for external network (non-HTTP) events
  - Handle time-based processing
- What happens if the servlet container stops or reloads your application?
- When does your socket/timer get created? Will it get created twice?
Control Over Lifecycle

- If you spawn another thread...
  - If the servlet container reloads your application, it will be loaded with a different instance of the class loader
  - Object assignment may fail with `instanceof` problems. More on next slide...
The `instanceof` operator is used to determine if one class is of the same type as another.

A class is an instance of another if:

- The Class object from which it is constructed is the same type **and**...
- The class loader from which the two compared instances were instantiated is the same
Complex User Interfaces

- Events that occur asynchronously to the user-input events
  - Eclipse and NetBeans are familiar examples
- Drag and drop capabilities are desirable
- ...etc.
- For example, Google Maps on web vs. Google Earth
Alternatives to Servlets

If the application programming model for servlets is inappropriate for your application, what are the alternatives?

- Write a Java application with an appropriate threading and lifecycle model.
- Remember that a servlet container is a Java application, so any application you develop can be a superset of that functionality.
Architecture Considerations

- Key architecture decisions center around:
  - Threading model
  - Lifecycle model

- Other design areas apply, of course:
  - User interface design
  - Data model
  - Application object model
  - ...etc.
Understanding Thread Models

- The servlet gives us a (mostly) single-threaded programming model
- Parallelism occurs by running requests on a symmetric thread pool
- When we write our own application, we can choose a more appropriate thread model
Threads: Single Threaded

- The simplest threading model is a single thread of execution that runs until the process is complete.
- This is appropriate for many batch jobs.
- Typically not appropriate for interactive applications.
Threads: Fork/Join

- Java applications can create their own threads by instantiating a `Thread` object and running it.
  - That subtask runs in parallel with the thread that spawned it.
  - Any shared objects must be synchronized in some manner. Excessive shared objects will reduce the parallelism and increase application complexity.
- Other threads can call `join()` to resynchronize application flow.
Threads: Symmetric Pool

Your application can implement parallelism the same way servlet containers typically do:

- Create a pool of available threads
- When a request for service arrives, obtain an available thread and run the request to completion

Can be an alternative to fork/join approach

- More efficient since threads are not created for each request
- Thread creation will most likely call operating system services
Threads: Asynch Queuing

- Some applications can benefit from internal asynchronous messaging
  - Similar to the SDK-level Windows or OS/2 Presentation Manager threads
  - The SWT UI thread processing uses this technique
  - Events are placed on a queue. A thread is blocked until an event is placed in the queue. The thread then processes that event, possibly generating events that are placed on other event queues

- These threads typically live as long as the app
Threads: Asynch Queuing

- Other threading models are more familiar, so I want to go into more detail on this model
  - My personal favorite
  - Very resilient to extreme short-term fluctuations in workloads
  - Very robust. High arrival rates for extended periods of time tend to increase memory utilization, but don't cause “death spiral”
public class BlockingQueue {
    private LinkedList queue = new LinkedList();
    public BlockingQueue() {
        super();
    }
    public synchronized Object dequeue() {
        while (getQueue().isEmpty()) {
            try {
                wait();
            } catch (InterruptedException exception) {}
        }
        return getQueue().removeFirst();
    }
    public synchronized void enqueue(Object request) {
        getQueue().addLast(request);
        notify();
    }
}

/* This code is a simple implementation of a blocking queue using a LinkedList. The queue allows for
 * safe access and manipulation of a queue of elements. It provides methods to add elements to the end
 * of the queue and remove elements from the start of the queue. The queue can also block if it is
 * empty, allowing other threads to continue execution. */
Queuing Thread

```java
public static Object SHUTDOWN = new Object();

public void run() {
    Object request = getQueue().dequeue();
    while (SHUTDOWN != request) {
        getProcessor().serviceRequest(request);
        request = getQueue().dequeue();
    }
}
```
Queue-to-Queue

- Each of the queuing threads are decoupled from one other
  - Items placed in the queue shouldn't have references to objects in other queues unless those are made thread-safe
  - As in fork/join, a minimum number of critical sections (synchronized code) will give the best throughput and reduce deadlock opportunities
Queue-to-Queue

- The queues are loosely coupled together through a controller (as in MVC) that is responsible for dispatching to the various queues.
- The controller is also responsible for the lifecycle of the queues.
Understanding Lifecycles

- The servlet defines the lifecycles for:
  - Application
  - Session
  - Request

- If we write our own application, we need to have an understanding of how to spec these ourselves
  - Lifecycle management is one of the primary reasons why servlets don't work for my needs.
Application Lifecycle

- The application lifecycle matches that of the JVM.

- We programmatically handle the instantiation of the network and timer services, UI, etc. as well as the termination of those services.

- We avoid being taken out of service or being reloaded, which avoids the corresponding class loader issues.
Session Lifecycle

- Servlets have a somewhat non-explicit lifecycle
  - Created when the session is “joined”
  - Terminated when it times out
- Before HTTP, session lifecycles were often matched to a network session
  - Connect to a service such as telnet, ssh, ftp, etc.
  - Authenticate and do appropriate work
  - Disconnect
Session Lifecycle

- Matching session lifecycle to network lifecycle has some benefits
  - Identity is less easily spoofed because the network session itself would have to be hijacked
  - Less startup time (including key exchange and authentication) than is required with stateless HTTP. This is important for communicating with devices over low bandwidth, etc.
Request Lifecycle

- In a servlet, the request/response lifecycle is modeled after the HTTP request/response lifecycle.

- In a custom application, this can be a little more esoteric:
  - Can be the effect of a user input event, but may not have a response
  - Can be processing after a timer event
  - May be a single network record from a partner application
Hybrid Models

- Could use an embedded servlet engine like Jetty.
  - Talk to operational folks about deployment issues like sharing port 80

- Could use a two-process model and talk between the servlet and the application via sockets
  - The socket lifecycle could match the servlet lifecycle (i.e. `init()`/`destroy()`)

Summary

- Servlets are appropriate for a certain class of applications...
- ... however, they aren't appropriate for all applications.
- When a new application is going to be developed, take the time to understand the deployment and usage of the application so the most appropriate architecture can be used.
Thank You

- Thank you for taking the time to attend this session. I hope it has been helpful.
- Fill out the session evaluations. They are helpful to Wayne & Peggy and me.
- Feel free to contact me via e-mail at: glm@hilbertinc.com
- I will be at the conference all week. Feel free to ask any questions that arise after the session.