Java Servlets and Enterprise Java Beans In Enterprise Architectures: Friends or Foes

Part I

By Max Dolgicer, Gerhard Bayer and Michael Bardash
International Systems Group (ISG), Inc

Our return to the pages of Application Development Trends comes after a period of intense change - the demise of the dot.com phenomenon (which almost overnight became the dot.bomb experience), the introduction of too many buzzwords and standards, a plethora of new products that implement - or just claim to implement - the latest buzzwords and standards, and much, much more. As always in this business, vendor marketing dollars moved quickly from yesterday's "du jour" technologies a couple of years ago and are now promoting J2EE, .NET and, of course, Web services.

However, though the ever-increasing number of buzzwords does create a degree of confusion, one can oftentimes quite easily follow the common thread and common sense of what we call “the buzzword evolution”.

A couple of years ago one of our engineers jokingly declared: "after the invention of the state machine everything in computing is a hack." While this is far from reality, many industry veterans would concur that while the buzzwords and standards do change all the time, the basic principles of distributed computing and middleware remain the same. Just follow the trail of "buzzword evolution" in the common vocabulary of magazines. IDL (Interface Definition Language) "got replaced" by WSDL (Web Services Definition Language); SII (Static Invocation Interface), DII (Dynamic Invocation Interface) and RMI (Remote Method Invocation) "were taken over" by SOAP (Simple Object Access Protocol); and finally, Service Oriented Architecture replaced "Whatever Oriented Architecture" was in vogue.

Yes, there are more standards now than ever before (skeptics are referred to the list of XML standards governed by the World Wide Web Consortium, or W3C). But at the end of the day, only a few standards matter to developers.

The "buzzwords and standards" evolution begs the question of whether middleware choices are more confusing for IT executives and development managers now than

they were two years ago. As a consulting company always trying to stay at the front of the middleware revolution (and evolution), taking advantage of many opportunities to implement important projects using a multitude of middleware technologies, we can report that unfortunately, the answer to the question is not a binary yes or no. The good news is that there are some middleware choices that are becoming close to binary. And at least when it comes to the development of brand-new component-based e-business applications there are really only two middleware platforms to choose from - J2EE and .NET.

However, the reality is that most companies have long focused less on developing new applications than on integrating existing applications. That reality is even more true today as IT budgets undergo more scrutiny than ever before. According to recent monthly CIO surveys conducted by investment firm Morgan Stanley, application integration was cited consistently as a top priority for 2002 and beyond. The bad news is that no single middleware solution has emerged from the pack to become an obvious choice for projects that are focused on application integration. So the technologies and products that enterprises can evaluate today remain very complex and are even more confusing than they were in earlier years. Today, viable integration technologies include Message Oriented Middleware (MOM), Message Brokers, CORBA, J2EE-based application servers and Microsoft's .NET.

As middleware consultants, important questions we hear regularly include: Are the J2EE-based application servers ready to undertake complex EAI projects? And what are the long-term risks associated with the use of proprietary Message Brokers?

In this and future articles we'll try to answer some of the most asked technical questions, including: What are the differences and similarities between J2EE and .NET frameworks? Are J2EE-based application servers ready to become the technology of choice for EAI projects? And what are the keys challenges in the development and deployment of J2EE applications?

In this article we tackle a question that we have been asked repeatedly during the past few years. As J2EE-based application servers like BEA's WebLogic Server and IBM's WebSphere gain more widespread acceptance, should Java servlets or Enterprise JavaBeans (EJBs) be used as a foundation for e-business applications? Most of the early Web-based J2EE applications have been developed using servlets as a base for two main reasons. First, the servlet model offers more simplicity than EJBs and is easier for IT developers and managers to comprehend as they get started with the technology. And second, the majority of the early projects are focused on the so-called "low-hanging fruit," or new e-business applications that focus on presentation-centric applications and do not have high developer demands in terms of complexity, scalability, availability, reliability and extensibility.

Meanwhile, the vendors state that the purpose of J2EE application servers is to support high-end enterprise applications and to serve as the new, strategic middleware platform for all application development and deployment. It is therefore important to understand the difference in capabilities of servlets vs. EJBs so that the suitability of one approach

over the other can be determined given the requirements of a particular project, the readiness of IT personnel, budgets and realistic time-to-market.

Popular Technologies

Both Java servlets and Enterprise JavaBeans are popular technologies, and both can play a prominent role in enterprise architectures. Frequently, suppliers and users view these technologies as competitive, especially for Web-centric applications. Indeed, on the surface, servlets and EJBs can both be used to allow thin clients, such as browsers, PDAs and the like, to access enterprise data. Two variations of a classical architecture depicted in Figure 1 illustrate this contention.

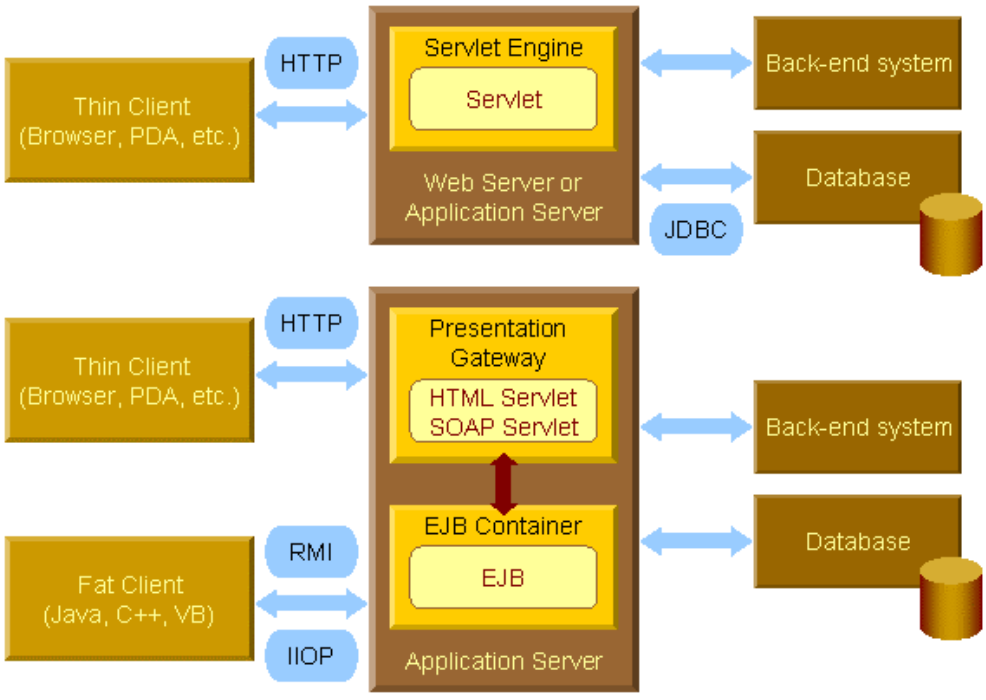

Figure 1

The first diagram shows a thin client (such as a browser) accessing a servlet that is in turn connected to a back-end system or database. The second diagram shows a client accessing an EJB through a presentation gateway, for example an HTML Servlet.

Indeed, from the browser client perspective, there is not much difference between the two technologies - both bridge between presentation and back-end functionality. And, in many cases, servlets are considered to be a better choice because their coding and deployment requires simpler skills and shorter times than those required for using EJBs, which deal with a whole spectrum of complex issues, including many additional APIs and rules. Of course, when an enterprise-strength architecture is in question, neither the browser client perspective nor the simplification of the development process is a good enough justification for the choice of technology.

The fundamental difference between servlets and EJBs becomes apparent when such essential design objectives as architectural robustness and business agility are brought into focus. Then, issues like scalability, state management, flexibility of entity relationships and richness of the cooperation metaphor start to play a prominent role in the technology selection process. We intend to show why EJB-based architectures can handle such concerns better than servlets-based technologies, and why a clear understanding and recognition of this conclusion is required when new, complex applications are being developed.

We may be giving readers the impression that we are biased toward EJBs as the preferred vehicle for enterprise architectures. However, our view is that EJBs and servlets are orthogonal technologies that, rather than competing directly, are mostly complementary. To compare EJBs and servlets is not like comparing apples to apples. It is not even like comparing apples to oranges - it is more like comparing apples and oranges to the crates and barrels that are used to store and transport them. This becomes obvious when the essential differences in intent, focus and, if you like, domain philosophy of the two standards are investigated.

Servlets cater primarily to the delivery of dynamic content to browser-based clients. They are narrowly focused on facilitating presentation, such as programmatic translation and preparation of HTML, and on relief from handling lower-level details of HTTP. This is the extent of the role servlets play in the framework of the J2EE standard.

On the other hand, the EJB standard was devised with the vision of a common pattern for component architectures in mind. It deals principally with enabling the development and deployment of a business application as a collection of components into a framework of powerful and comprehensive infrastructure services (which can include Java Database Connectivity (JDBC), Java Transactional Service (JTS)/ Java Transaction API (JTA), Java Messaging Service (JMS) and the like). Rather than prescribing any specific role (such as HTML rendering or HTTP encapsulation), the Enterprise JavaBean component model is a mold for generic business components whose exact specialization is left to the application designer to establish. In other words, the servlets specification is a somewhat narrow standard specialized on the presentation layer of applications, while the EJB standard is a broad enabling technology.

This distinction becomes more apparent when directly comparing the characteristics of EJBs and servlets to evaluate their respective applicability to enterprise architectures.

- A servlet is a faceless Java object. Beyond some features mandated by its base class, it is amorphous and can have arbitrary internal composition. On the other hand, by contract, an EJB is obliged to implement specific interfaces (such as the home interface, activation and deactivation interfaces) and mandatory properties (such as primary key, for example) that promptly tie it into the cooperation environment (the container).
- A servlet has to implement a rigid set of methods with predefined signatures just as Enterprise JavaBeans need to implement a set of methods imposed by the EJB specification to fulfill the contract with the container. However, the EJB can also define arbitrary methods that best suit its business semantics.
- Servlets have very thin support from the environment they are deployed into, which is mostly limited to HTTP-related matters; EJBs enjoy a wide range of powerful and comprehensive infrastructure services that are provided by the container. Examples include the generation and handling of remote interfaces, component factory services, component instance identification, distributed transactions, automatic persistence for Entity Beans and declarative security.
- Basically, servlets are focused on one task: reacting to HTTP requests with HTML responses in a stateless manner (in fact, state may be maintained via an HttpSession object, but it is not comprehensive and requires explicit client cooperation via session cookies). EJB behavior is defined at a higher level of abstraction: EJB method invocation does not stipulate any syntax or semantics to the invocation arguments or to the maintained state.
- By design, the aim of EJB is to segregate business functionality from infrastructure services (such as life cycle management, transaction and security contexts and persistence). Because servlets are not integrated with the J2EE infrastructure services, they encourage application developers to deal with lower-level J2EE APIs directly, and entangle business functionality with logic related to infrastructure services.
- The servlets specification does not address enterprise concerns such as load balancing and failover. On the contrary, inherent scalability and high availability of services is a declared responsibility of the EJB container. (It should be noted that some servlet engines do support clustering and load balancing. Curiously enough, these are the engines that are embedded into EJB application servers such as WebSphere and WebLogic).

In other words, servlets do one thing and do it well - they provide a shell for the presentation layer of a Web-based application; EJB is a complex framework for implementing widely dissimilar business functionality in a coherent and comprehensive fashion.

Servlets expose; EJBs encapsulate

The other point of distinction between servlets and EJBs is that, in our experience, in the majority of cases, servlets are used to *expose business data*, whereas EJBs are used to *encapsulate business functionality*.

A classic use of a servlet would be to parse an HTTP request, access a database for inquiry or update, and to compose an outgoing HTML reply. (A less common use of a servlet would involve passing a serialized Java object between itself and a browser-side applet. This scenario will be discussed later.) For example, a browser may submit a request for an airline schedule, and the servlet will fetch the data from the database and format the reply as an HTML table. Also, the browser may submit a filled form whose elements will be stored by the servlet in a database table. In either case, it is the data, not the behavior, the client is interested in.

Arguably, HTML data-level cooperation achieves a great level of independence between the presentation at a client (in this case the browser) and the processing at the server (the servlet). The interface between the client and the server is primarily concerned with passing data; there is no API specific to any particular business function. The browser client displays HTML returns unconditionally (subject to its validity, of course). Thus, if the servlet implementation is changed to render richer replies (such as embedding sound clips, for example), the client will benefit from this extended business functionality automatically and transparently. However, if the servlet is changed to display, for example, the latest cricket results instead of stock quotes, all the same data will be unconditionally rendered by the client, much to the user's dismay.

In other words, the client's horizon extends all the way to the data and the servlet acts merely as a rendering agent that enables the client to access the database (Figure 2).

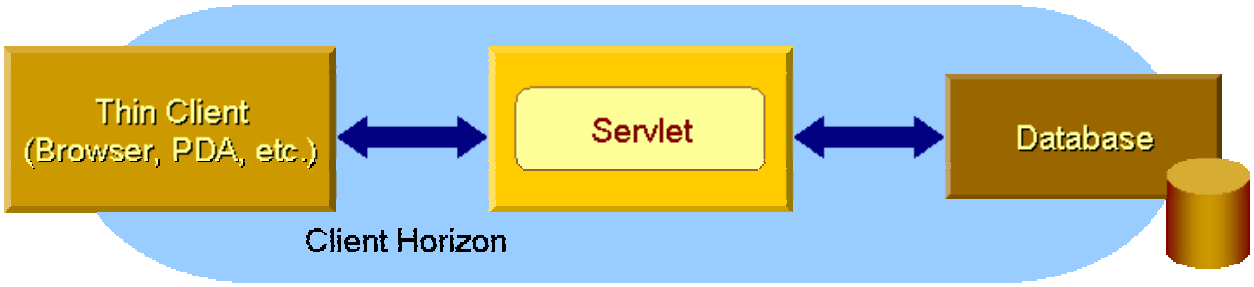

Figure 2

Separating the presentation by the browser client and the processing in servlets achieves a higher degree of independence compared to an EJB-based architecture because it does not rely on the strong coupling that the interface of an EJB with rigid business semantics imposes. However, this comes at the cost of compromising business process integrity. This might not be a cause for concern for simple Web browsing interactions, but from an enterprise-strength B2B/B2C perspective, cohesiveness of the end-to-end business process flow must be protected. This objective is best achieved by encapsulating business behavior in services, and service granulation and exposition is precisely the domain of the EJB.

With a typical EJB implementation, a client gains access to business functionality through a presentation gateway, whereby the latter provides a rendering agent for business functionality as opposed to data. Thus, a typical use of an EJB would be for the client to remotely invoke the server-side service, which may or may not result in any data being returned to the client, but which would definitely result in some meaningful processing having taken place at the server side. This concept is illustrated in Figure 3.

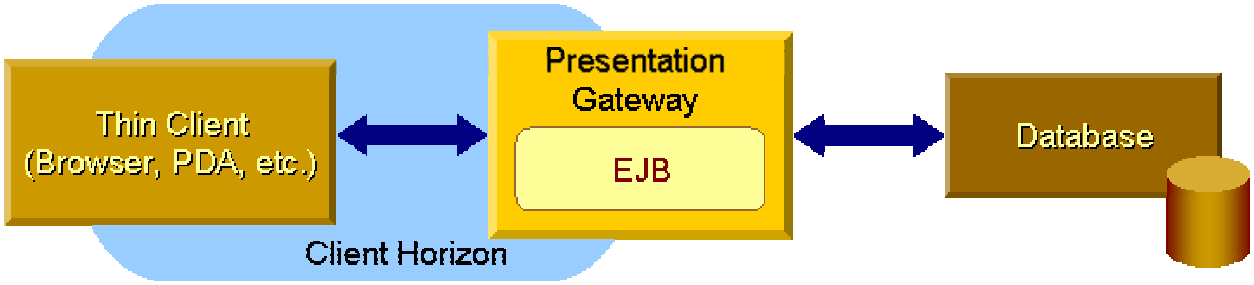

Figure 3

Two-tier vs. three-plus tier

Another way to contrast servlets and EJBs is to assert that servlets by intent serve to accommodate a classical two-tier application (where the presentation and data access/business logic tiers are commingled within the servlet and the data store). On the other hand, EJBs tend to three-tier or multitier complex cooperative environments.

Apparently, the flexibility of interpretation built into the servlets standard permits us to break this data-only paradigm easily. Nothing prevents a servlet from being engineered in such a way that it encapsulates a business service. Practically, this can be achieved in a number of ways. For example, instead of only accessing a database, a servlet implementation may invoke back-end functionality that has been written in Java via an RMI call, or a servlet may exchange serialized Java objects with a browser-resident applet. These and other design patterns that turn servlets into business logic capsules

are well known and quite popular. This approach leads to an architecture that looks like the one in the picture below:

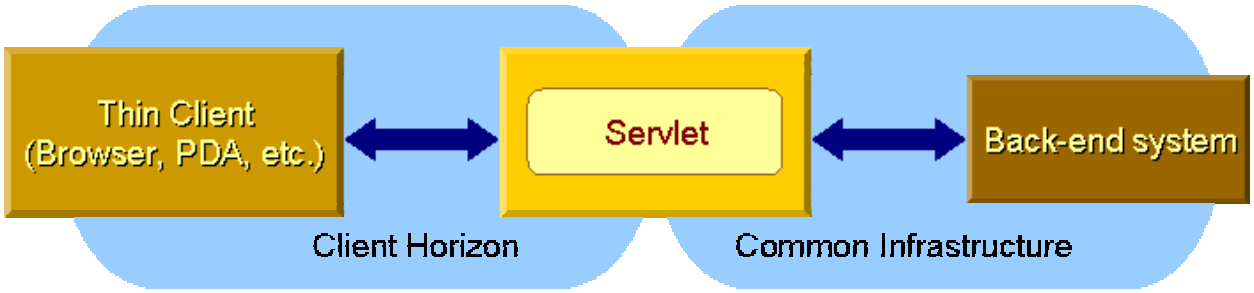

Figure 4

In practice, it has become a common architecture approach to allow servlets to benefit from their amorphous "anything goes" internal structure - or, better yet, their lack of any internal structure. Nothing prevents servlet implementations from internally integrating with a variety of technologies like RMI/CORBA, JNDI, JMS, JTS/JTA and so on; just as no out-of-the-box transparency exists for JDBC - a main vehicle to access databases from within a servlet - the constructs within a servlet for all of the above technologies have to be manually coded on a case-by-case basis. Nevertheless, the ability to embed the whole spectrum of Java platform solutions on the client's behalf turns servlets into a powerful middleware mechanism that extends the Java APIs to the Web.

It seems that both the servlets and the EJB technology are suitable platforms for an enterprise application. Furthermore, they both seem suitable for presentation-centric applications that do not require support for "hard-core" distributed services such as transactions, load balancing, persistence, state management and message queuing, among others. The use of servlets might be even more appropriate and certainly more straightforward. However, even though it is possible to use servlets for truly distributed applications, such use would be sub-optimal. The reason for this far-reaching statement will be clarified in Part II of this article, in which we will compare these two technologies in more detail. Stay tuned.

Java Servlets and Enterprise Java Beans In Enterprise Architectures: Friends or Foes

Part II

By Max Dolgicer, Gerhard Bayer and Michael Bardash
International Systems Group (ISG), Inc

Earlier we took a high-level view of the Java side in the e-business middleware clash between Microsoft's .NET and Java 2 Extended Edition from Sun Microsystems Inc. and its partners. Now we look more closely at the relationship between Java servlets and Enterprise JavaBeans (EJBs), and at how the two technologies can be best used separately in some cases and as complements in others.

In the earlier "Servlets and EJBs: Friends or foes?" article, we offered some thoughts on the evolution of the middleware platform so far. We also concluded that the phrase "less is more" is certainly applicable when it comes to the development of new, component-based e-business applications now that there are just two middleware platforms to choose from - J2EE and .NET.

In that article, we noted in passing a question that we have been asked repeatedly during the past few years: Should Java servlets or EJBs be used as a foundation for e-business applications?

This is an important question, since the servlet model offers more simplicity than EJBs. The servlet model is also easier for IT developers and managers to comprehend as they get started with the technology. Our opinion is clear. If an application is presentation-centric and does not require support for high-end middleware services - distributed transactions, persistence, application-level load balancing, state management and asynchronous messaging, among others - then opting for heavy EJB usage is overkill. However, if an application requires at least a partial list of such high-end middleware services, then EJBs become the only game in town, assuming of course that the customer has selected or is about to select a J2EE-based application server.

The following table compares and contrasts the two models and provides a more detailed insight into how they stack up against each other using a set of objective parameters. With Table 1 in mind, we can evaluate how well the two technologies address enterprise concerns.

Application Topology		
Servlets	**EJBs**	**Comments**
Web-centric model (client connects via Web server only)	* Web-centric model (via presentation gateway, e.g., HTML servlet) * LAN-based model (clients access services over LAN or WAN via remote method invocations without any intermediary) * Local (several tiers of services collocated within a single host, either in intra- or inter-process fashion)	EJB permits for much more flexible deployment topographies. Web server is not a prerequisite.
Client Access		
Servlets	**EJBs**	**Comments**
HTTP only	HTTP * RMI and/or CORBA * IIOP HTTP tunneling * Value-added proprietary protocols such as t3	HTTP is a stateless protocol optimized for request-reply style text transfer. An attempt to engage it for generic use, such as may be required by common cooperation metaphors like stateful session, involves significant overhead and is not trivial to implement. On the other hand, technologies like RMI and CORBA are specifically optimized for distributed computing and easily account for a variety of cooperation metaphors. HTTP tunneling allows circumventing a firewall in cases when direct client access is precluded.
Transactional Support		
Servlets	**EJBs**	**Comments**
* Transactions must be coordinated manually through integration with JTS. * A client cannot easily pass transaction context to a servlet. * Attempt to allow a single transaction to spawn multiple servlet invocations is at least non-trivial.	* All transaction functions are performed by the container implicitly on behalf of an EJB, including context propagation and transaction demarcation. However, the bean is not precluded from taking full control over transaction management	Though a servlet can ultimately be made as transactional as an EJB, this attempt will involve significant effort, both at design and development times. EJB strongly relieves these concerns.

	* Transaction rules are declarative in nature and can be changed at deployment time.	
Security		
Servlets	**EJBs**	**Comments**
There is no security mechanism readily available to servlets. Rudimentary access security is managed by the servlet engine in a superficial manner (in most cases, limited to differentiating between trusted and untrusted modes).	Comprehensive security mechanism is managed by the server. EJBs are instantiated into security context and service invocations are authorized against access control lists. Security definition is declarative and defined at deployment time.	In a Web-centric environment, both standards benefit from technologies like SSL and digital certificates. But implementation of these features is not mandated by either standard and is left to the discretion of specific products. Most app servers and servlet engines (especially those embedded in app servers) support an end-to-end Web security model.
Persistence		
Servlets	**EJBs**	**Comments**
No specific mechanism exists to provide for servlet persistence, though of course nothing prevents a servlet from implementing persistence on its own by explicitly integrating with JDBC.	EJB allows persisting objects automatically. Support for this comes from two angles. Introspecting the bean can automatically generate database schemas. Container-provided life-cycle management automatically passivates and activates objects as needed and provides for synchronization of EJB state with the database.	
Object Identification and Context Association		
Servlets	**EJBs**	**Comments**
To identify a Servlet into context (such as in the case of a session context, for example) requires explicit cooperation between the client and the engine. Client-side cookies are needed, and additional coding is required on the server side for session tracking. Integration with a naming service is not possible at	EJB differentiates between stateless and stateful objects. While the former are created on demand and do not outlive single invocation, the latter are uniquely identified by a primary key, which permits unambiguous context identification.	EJB benefits from its implicit integration with persistence and naming services. Moreover, while servlets leave the topic to the user's discretion, EJB supplies ground rules and a comprehensive framework for identification of EJBs.

the client side, and has to be manually coded at the server side.		
Environment Access		
Servlets	**EJBs**	**Comments**
Servlets have no access to the runtime environment other than regular Java mechanisms.	The EJB container manages environment properties on behalf of individual EJBs. These can be defined at deployment time, allowing for greater deployment flexibility.	EJB provides better ergonomics of the runtime environment management, an essential part of system administration.
Naming and Directory Services		
Servlets	**EJBs**	**Comments**
No integration with JNDI except when manually coded. No client-side access to directory services.	JNDI is fully integrated into the server and the namespace is managed transparently. EJB provides naming context and automates object registration.	The servlet client's inability to discriminate its server object is a limiting factor for component-based architectures.
Life-cycle Management		
Servlets	**EJBs**	**Comments**
Servlets are bound into the end-user context only for the duration of a single request. Servlet instances are managed by the server only to assure that the configured number of instances of a certain kind has been pooled.	EJB instance life span can vary from per-request instantiation to the duration of a session, to the persistence over many user sessions. The persistence mechanism allows bean-based implementations to survive catastrophic session failures and container shutdowns.	
State Management		
Servlets	**EJBs**	**Comments**
Servlets do not have any prescribed state management policies.	The container cooperates with the EJB to maintain the state. In contrast to servlets, stateful EJBs are required to implement activation/passivation methods to enable the container to invoke them automatically to persist the EJB's state at well-defined points.	EJBs must comply with the state management contract, which in some scenarios may be perceived as an unnecessary burden. However, a well-articulated and standardized state persistence policy is a definite advantage to application designers.

Resource Binding		
Servlets	**EJBs**	**Comments**
The association of a servlet with back-end resources must be hard-coded.	Back-end resource access is encapsulated in entity beans that act like resource proxies to the session-level objects.	To achieve the same level of indirection and encapsulation as available with EJB, servlets have to manually provide for resource management. This may be prohibitively complex, especially when resource management needs to be integrated with transactional and security management.
Resource Sharing		
Servlets	**EJBs**	**Comments**
Resource locking must be manually implemented and coordinated across servlet instances. It is easy to break resource sharing, as there is no centralized enforcement authority.	Resource sharing is managed at the level of entity beans. The container manages access to the entity beans and synchronizes access to them according to the defined sharing policy.	An EJB resource-sharing mechanism creates an essential foundation for resource load balancing.
Relationship Multiplicity		
Servlets	**EJBs**	**Comments**
The relationship between the client and the servlet is one-to-one only, for the duration of single request.	The multiplicity of relationships between client and server-side objects is not defined. The client may reference many entity and/or session beans simultaneously.	The relationship between client and servlet is inherently damaged by the limitations of the HTTP protocol. However, a number of work-around techniques may be suggested, like having a single servlet to multiplex all types of client requests. The drawbacks of such a scheme are obvious.
Database Integration		
Servlets	**EJBs**	**Comments**
Servlets rely on JDBC facilities to pool database connections. This pooling is managed explicitly. SQL statements have to be prepared manually.	The database connection pool is managed by the server, according to the deployment time policy. The container manages object persistence. SQL statements can be generated automatically.	

Invocation Arguments		
Servlets	**EJBs**	**Comments**
Servlets require manual marshalling of call arguments from their HTML representation. Analogously, return values have to be embedded in HTML. The Servlet API defines a number of programmatic facilities that assist in this task.	EJB invocations do not have fixed signatures, and arguments can be represented as a structure of arbitrary complexity, including, but not limited to HTML strings.	Direct HTML rendering provided by servlets is a strong advantage for a browser-based client.
Threading Model		
Servlets	**EJBs**	**Comments**
Servlets do not have any inherent threading or synchronization model. Threading safety is left to the discretion of the developer. A single threading model is devised specifically to ensure that all calls are single-threaded.	Bean invocations run in dedicated threads. The container manages thread synchronization. Thread pooling is available for performance optimization.	
Metaphor		
Servlets	**EJBs**	**Comments**
Servlets essentially support a single metaphor: request-reply. It is possible to identify requests into a client session, but this requires additional design and, because of API limitations, is not a comprehensive solution.	EJBs support a wide spectrum of metaphors, including request/reply in both stateless and stateful manner, asynchronous communication through message-driven beans, as well as session and persistent session.	
Chaining		
Servlets	**EJBs**	**Comments**
Servlets can be chained to each other to process client requests in a sequential manner.	An EJB that receives the client request can fan out consequent service invocations.	

Table 1: Comparing Servlets and EJBs

Scalability

Both servlets and EJBs scale well. A number of features - such as server clustering, DB connection pooling and location transparency - can facilitate scalability.

Each technology - or rather the products that implement the technologies - has been built to support each of these concepts. However, servlet scalability is based mostly on proprietary, vendor-created implementations of servlet pools and load-balancing mechanisms, which are not stipulated by the servlet API. On the other hand, the EJB architecture was designed from the beginning with a vision of addressing scalability.

The essential point of distinction here is that while servlets can scale at the server level, EJBs can scale at the architecture level. Server-level scalability depends on specific product features (which, no doubt, are almost universally supported nowadays by best-of-breed products, and are thus sufficiently mature and comprehensive). Reliance on product characteristics rather than on a robust, scalable architecture is a gamble because, in some cases, loads can quickly outgrow product capacity and require emergency re-architecting, which is a very expensive necessity.

Load balancing

Both EJB servers and servlet engines can provide mechanisms for load balancing that include resource pooling, default dispatch mechanisms and entity clustering. While benchmarking results of servlet engines vs. EJB application servers show more or less compatible performance curves under heavy loads, a load-balanced EJB-based application has a better guarantee of operational integrity.

There is no enforcement on an individual servlet design to mandate its guaranteed incorporation into the distributed environment. As a result, load balancing has to be accounted for at design time and thus becomes a condition to proper architecture design and mature vision.

If load balancing is not taken into account, later attempts to introduce new resources create the potential for resource sharing conflicts, breach of transactional guarantees, or for sub-optimal performance caused by custom dispatch decisions or resource locking. On the other hand, EJBs can supply an infrastructure with rich component management capabilities that allow for the plug-and-play-style introduction of new resources.

Business logic hosting

Because servlets have no impositions on their internal structure and therefore have to manage all aspects of their existence explicitly, they attain only a mediocre level of separation between business functionality and cooperation management (such as database connection management calls, for example).

Business logic becomes intermingled with service calls to databases, name servers or other infrastructure services. EJBs, by design, attain a good level of business separation. Another limitation imposed on servlets' capability to host business logic is their responsiveness obligations. Because they operate within an HTTP session, a lengthy calculation - which may be necessitated by the business functionality - could time out the user session.

Business agility

EJBs and servlets do not preclude design-level atomization of business services, so application components can be replaced with other components transparently to the overall architecture. However, because of its more flexible cooperation model, an EJB provides a better vehicle to handle paradigm shifts.

For example, if a browser client must be replaced for some reason with a programmatic agent (such as in the case of business process automation), servlets would not be able to handle the change, while an EJB-based solution would require only modest and well-confined modifications.

Integrity

An EJB application is homomorphic throughout: The same component model is utilized for the component hierarchy, and all components are guaranteed to function under the single umbrella of transactional and security services.

In contrast, a servlet-based application is free to agglomerate any technology without restrictions, with the unavoidable risk of creating points of tension, and the consequent need to account for technology blending at the analysis and architecture stages.

High availability

Both EJB application server and servlet engine products are designed to provide high availability of services on the front end. This is achieved through clustering and failover mechanisms that, though differing between implementations, still pursue the same goal with more or less compatible effectiveness.

However, on the back end - at the level of business logic implementation - EJBs use advanced component distribution techniques, such as object factories and smart proxies that are not available to servlets.

Failover

In the case of EJB, a failover mechanism is a prescribed key functionality and is readily available without any additional analysis or design effort. EJB failover is facilitated by automatic persistence guarantees and by automatic failure detection. Servlets have no equivalent mechanisms unless they are coded explicitly.

Deployment

Deployment is not a priority issue for servlets, but in the EJB worldview, deployment is considered to be one of the key elements of application delivery.

For this purpose, EJBs define a whole methodology for defining deployment characteristics that is supported by extensive APIs dealing with environment, deployment descriptors and object properties. As a rule, deployment description is declarative in nature, and the container is capable of adjusting runtime characteristics (such as transaction guarantees) automatically.

Portability

Both servlets and EJBs are published standards that are backed up by reference implementations and product compliance certification programs.

Despite that, vendors choose to introduce non-standard elements that, in their opinion, enhance product functionality. Because of the huge difference in the breadth of coverage and depth of penetration between servlets and EJBs, EJBs more easily fall victim to such enhancements; so, applications developed over different EJB products carry a greater risk of not being completely portable.

Management and administration

Runtime management and administration is a high priority for both EJBs and servlets, and most products supply facilities for component start and stop, configuring operational parameters, event logging and health monitoring.

EJBs supply a runtime environment that enables easier component monitoring and management. Integrating a component into a management infrastructure is a seamless activity with EJBs, whereas with servlets, if it is feasible at all, it requires the use of product-specific APIs.

Development ergonomics

Servlets is a relatively simple standard that requires basic knowledge of Java and

HTML, at the most. EJBs, on the other hand, are complex and multifaceted, and require an understanding of the J2EE platform, as well as proficiency with essential computer science fundamentals (such as threading, transacting, object technologies and so on).
It is probably clear at this point to the development world that Enterprise JavaBeans are better suited for enterprise application platforms than pure servlets are. We say "pure" servlets to underscore the fact that a complete (though, as explained, not necessarily adequate for certain uses) architecture can be built exclusively upon servlets supplemented with a mix of other Java technologies. In reality, however, there are shades of gray between pure EJB and pure servlet solutions.

First, many servlet engines are embedded into app servers and share a common implementation architecture with EJB servers. This allows vendors to merge the servlet execution environment with the EJB server and thus bring servlet execution under the same failover and load balance guarantees as EJBs.

Second, EJBs by definition declare compatibility with the other J2EE standards, including Java Server Pages (JSP), a technology that aims for the same goals as servlets (though admittedly in a different way). Moreover, JSP uses servlets as an implementation vehicle, as JSPs are compiled and cached as servlets.

If we return to our initial assertion - that servlets are a narrowly specialized technology that is excellent for delivering dynamic HTML content, and that EJBs are a wide, generalized specification designed to componentize business functionality - it becomes apparent that the class of enterprise-strength applications dealing with browser-based clients can benefit from both servlet and EJB technologies simultaneously and without inherent conflict. The basic architecture of an application utilizing both technologies is depicted in Figure 1.

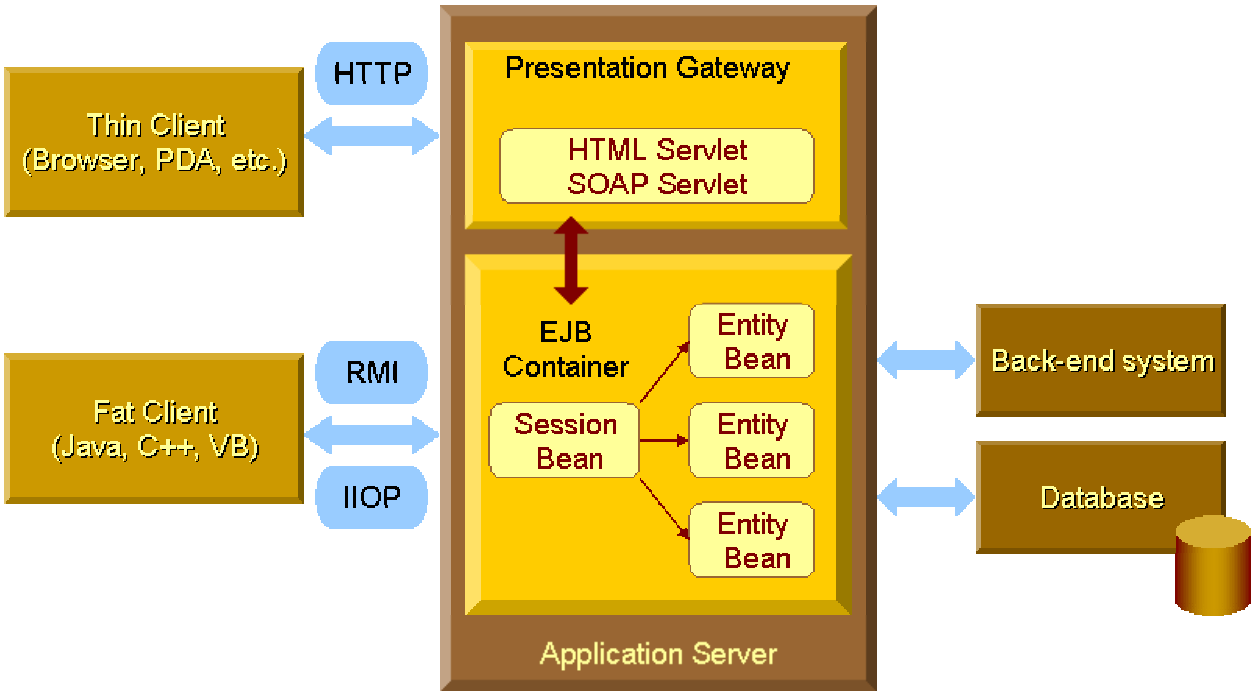

Figure 1

Here, the browser client initiates an HTTP request that is served by a servlet. Rather than implementing complex business logic, the servlet simply redirects the call, along with parameters and perhaps some identification information, to a session bean. The session bean implements the top-level process flow associated with the call, but delegates specific activities to the specialized entity beans. These serve later as proxies to various resources both inside and outside the application server. Developers should notice that all of the business logic is under EJB container management and that it benefits fully from the rich infrastructure services and container-side management.

It should be noted that this architecture has the inherent flexibility to support not only browser-based clients, but also richer clients (in an intranet scenario). In addition, it can be easily extended to make business functionality available through new interface mechanisms - for example, Web services or whatever the next great idea might be.

In conclusion, it is easy to see that because servlets are good at exposing Java interfaces to the Web, and EJB is the tool to enable good Java interfaces, there seems to be a perfect match in using a lightweight servlet as a Web-exposing front end to EJB-encapsulated business functionality.

About the Authors:

Max Dolgicer (mdolgicer@isg-inc.com) is a Technical Director of International Systems Group (ISG), Inc. (www.isg-inc.com), a consulting firm that specializes in development and integration of enterprise applications using leading edge Middleware technologies. Gerhard Bayer (gbayer@isg-inc.com) and Michael Bardash (mbardash@isg-inc.com) are Senior Consultants with ISG, Inc.