

Introducing GemStone/J

The GemStone/J application server provides a secure, scaleable, and robust runtime environment for Java server applications. GemStone/J manages resources efficiently, ties together data from a wide variety of sources, and enables different applications to work together.

You build your application from standard components using standard APIs and protocols. We give it a runtime home that:

- starts and stops processes as needed to ensure that the system is available and serving as many users as possible;
- authenticates users and verifies their authorization to perform the requested operations;
- connects to corporate databases;
- coordinates concurrent access to data;
- operates with legacy applications;

- manages resources such as memory, threads, database sessions, or sockets for maximum throughput;
 - manages remote connections to the Internet, or to back-end mainframe systems, including encryption when required;
- and
- stores data in a variety of ways, from direct object persistence to object-relational mapping.

As e-commerce grows increasingly complex, the GemStone/J application server takes care of the infrastructure, leaving you to concentrate on building the business components for your system.

This document gives brief and general answers to two questions about GemStone/J:

How do I use it?

How does it work?

From here, you can use the Table of Contents to move to topics of specific interest, or use the API link to view javadocs. You can also go to GemStone's web site to view a working GemStone/J application and find out more about Java application design.

How Do I Use It?

To take advantage of GemStone/J, you'll make these decisions:

1. Select an application model (or models), based on available resources and anticipated usage.
2. Determine the required access to existing data and define new data storage requirements.
3. Familiarize yourself with proven design strategies and best practices for implementing your business components, as discussed in the *GemStone/J Developer's Guide*.

4. Identify data and operations that require secured access, and choose among the available security mechanisms to implement security policies.
5. Configure the system(s) to maximize resources based on your application's specific requirements.
6. Choose a strategy to maximize system availability.

Application Models

GemStone/J supports these application models:

- Enterprise JavaBeans (EJB),
- web-based applications (servlets and Java server pages),
- CORBA applications,
- Distributed Java Beans (DJB), and
- generic applications.

Each model has its particular strengths; GemStone/J allows you to mix them freely to best satisfy the range of computing needs across your enterprise. You can even mix models within one application: for example, business components might be Enterprise JavaBeans, while a few special-purpose services might be implemented as CORBA objects.

Enterprise JavaBeans

Enterprise JavaBeans are standard Java-based components encapsulating domain objects and business logic, typically used in distributed server applications. EJB developers focus on developing reusable components for business purposes, while transactions, concurrent data access, data persistence, and aspects of security are managed for them.

GemStone/J transparently provides life-cycle services to Enterprise JavaBeans, instantiating and garbage-collecting them, activating and passivating them, and managing their participation in transactions.

GemStone/J makes EJB applications particularly scaleable, starting and managing pools of resources, such as connections to remote databases, to maximize the number of concurrent users.

EJB applications do not ordinarily command a dedicated connection; therefore, the bean itself must carry the application's conversational state. The need to fetch objects anew for each method invocation means that the EJB model best suits applications that don't need a lot of long-lived transactional state.

If scalability, reusability, and code portability are the overarching considerations, EJB applications are the clear choice.

Web-based

Web-based applications have two key advantages: usability and ease of administration. Because users access these applications through web browsers, you need not worry about what's installed on client machines.

GemStone/J supports two programming models for web applications: servlets and Java server pages (JSPs). Both approaches use the server to execute Java code and therefore make minimal demands on the client: the browser need not be Java-enabled.

- A *servlet* is a special Java class that implements the Servlet interface and responds to one or more specific web requests.
- A *Java server page* is an HTML page with embedded Java code. When a user accesses the page, the Java code is compiled into a servlet, which the server then runs.

If your application needs fairly sophisticated processing to determine how to respond to a client request, servlets are the more appropriate choice. JSPs are appropriate if a user request more or less defines the resulting page.

Servlets and JSPs can use HTTP sessions to tie multiple browser requests together into a logical conversation; GemStone/J enhances this feature with distributed tracking of HTTP sessions. HTTP sessions are available to multiple virtual machines, making your web applications more scaleable. Distributed HTTP session-tracking (DHST) also provides high availability: if a virtual machine fails, the user's HTTP session is not disrupted.

GemStone/J web applications allow you to bundle servlets, JSPs, static HTML, and other web resources together in one logical package. Thus, you can partition the work in the most reasonable way, so that it's easy to maintain, extend, distribute across virtual machines, or disseminate to others.

Soon available for evaluation is GemStone/J's next-generation web container, which integrates web applications with GemStone deployment and administration tools.

CORBA

GemStone/J is built on CORBA, a foundation that allows demanding applications to take advantage of CORBA's flexibility and power. If you need maximum throughput or minimum overhead, close control over threads or access to a shared, unique service, implement a CORBA application. CORBA is also the choice for systems that include legacy applications written in other languages.

GemStone/J provides the following CORBA services:

- a fully featured Object Request Broker (ORB),
- a CORBA Naming Service,
- a load-balancing Activation service, and
- a high-speed implementation of the Object Management Group's Object Transaction Service (OTS, also called CosTransactions).

Additional features provide convenience, flexibility, and fault tolerance:

- Message interceptors allow additional operations between client message-sends and server implementations, so you can, for example, piggyback logging or security information.
- Client and server callbacks allow you to track and manage connections.
- Smart stubs allow the client to catch failures and redirect messages; also, if the server should fail, the ORB notifies the client as it tries to reconnect to a new, different server.
- An API allows you to automatically collect ORB statistics.
- Multiple configurable threading models speed message-dispatching and minimize garbage.
- The Portable Object Adapter, fully integrated with the GemStone/J Activator service, supports Servant Managers and Adapter Activators.

GemStone/J's CORBA underpinnings give EJB and other applications many advantages, particularly the ability to manage OTS transactions and the Activator's load-balancing capabilities.

Distributed Java Beans

GemStone/J's unique Distributed Java Beans (DJB) model provides an interface for persistent object applications. A DJB application implements the required data and behavior as Java objects, using GemStone/J's repository to store them. It runs in its own dedicated *session* (a means to hold temporary data and a structure for interacting with the repository), accessing and modifying objects with GemStone/J's transactions and concurrency management. If your application would benefit from long-lived transactions and portability is less of a concern, consider the DJB model.

Generic Applications

Generic applications contain the basic services necessary to deploy and manage GemStone/J applications; all other application models inherit these basic services and attributes, adding to or overriding them as necessary.

You can use a generic application in a stand-alone virtual machine—for example, to perform bulk data-loading or other batch operations that do not require interaction.

Generic applications can also allow a variety of applications (of any model) common access to a class library, thus speeding deployment and reducing the size of these dependent applications.

Data

GemStone/J applications can access data from any relational database having a JDBC driver. Three of the most commonly used come bundled with GemStone/J:

- Oracle Thin
- jConnect, from Sybase, Inc.
- DataDirect SequeLink, from Merant Corp.

For new data, you may decide to use GemStone/J's own repository. It stores Java objects and primitive types, and supports a full suite of database management tools: object locking and transactions for concurrency control, and logging, recovery, backup, and restore utilities to help operators maintain the system and ensure availability.

If you'd like to mix both kinds of data in your enterprise, GemStone/J supports OTS-managed distributed transactions to ensure integrity between its repository and a relational database.

With GemStone/J, you can access data three ways:

- GemStone/J supports the JDBC API. Connect to a relational database, form queries or updates in SQL, and receive results to render or process as desired.

Because GemStone/J creates and manages pools of JDBC connections, a connection is most likely available when needed; if not, one is started for you. No dedicated connections wait idly.

- If you do not wish to embed SQL in an application, you can use an object-relational mapping tool such as Thought Inc.'s *CocoBase* to form queries for you and return Java objects.
- If you're working with new data, you can model it as objects from the start and take advantage of object-oriented programming's ability to capture domain subtleties in a maintainable design. Java objects can be stored in GemStone/J's repository indefinitely; backup and restore utilities, transaction logs, and other database administration tools are available to ensure data availability and integrity.

Large collections can be indexed so that queries return results fast. You can also formulate queries in an SQL-like language using GemStone-specific extensions to Java.

Even for applications using legacy data, GemStone/J's repository can be helpful. If most transactions are read-only, it may be quickest to read all the required objects into the repository when the application starts. Store the resulting Java objects in the repository while the application is running, and render data from the repository.

Or you can fetch data from the relational database and store it in the repository on an as-needed basis, or in batches, pushing changes back to the relational database in the same manner.

For write transactions, the Object Transaction Service coordinates distributed transactions. With the appropriate API call (automatic in most EJB applications), changes to objects representing persistent data propagate to the original database.

Security

GemStone/J provides authentication, authorization, and encryption services. By default, all connections to the system are anonymous, all operations are permitted, and connections are not secure. But each of these conditions can be tightened as necessary.

Authentication

All connections to the system are subject to authentication, controlled by credentials. You can use either GemStone/J's credentials, or a third-party commercial credential system such as Entrust. Both GemStone/J and Entrust credentials use account passwords and certificates based on the X.500, X.509, and Public Key Infrastructure (PKI) standards.

GemStone/J credentials are based on user-specific keystores. Upon login, the client's identity is confirmed by password, and the system grants access to the user's keystore. GemStone/J then automatically performs two-way authentication between clients and server, as well as system-wide authentication between virtual machines.

You can use GemStone/J's utility to create certificates, or you can use those from any third-party Certificate Authority. You can manage certificates and certificate revocation lists using GemStone/J's CERTMAN utility or a third-party PKI product.

If you choose to use Entrust, credentials are stored on the Entrust server. The Entrust application runs locally and provides a management GUI and tools for security administration.

If you choose to use another third-party PKI product, you need only create a class to implement the Credentials interface using that product's API.

Authorization

Only authorized operations can be performed. Authorization can be granted to users individually or in groups, on a per-method basis. To control who can view or modify data, write the appropriate methods

to consult access control lists (ACLs). You can also set up ACLs to control access to objects by name, using a naming service.

Encryption

When required, connections can use secure encrypted sockets (SSL). GemStone/J provides its own SSL implementation and also supports several commercial SSL packages.

SSL packages are generally subject to a variety of import and export restrictions. Consult your local import regulations and legal counsel to determine what SSL options are available to you.

Efficiency

Whichever application model you choose, GemStone/J manages system resources to give you the most from your hardware and software.

GemStone/J runs a pool of Java virtual machines, which in turn run applications. Each virtual machine (VM) can run more than one application and serve more than one user at a time. The GemStone/J system balances the load across the pool of virtual machines to ensure the fastest possible response time.

You can configure your system to prestart a pool of a specified number of virtual machines. When demand exceeds supply, additional VMs can start as needed, up to the specified maximum.

Within each virtual machine, resources are apportioned for maximum efficiency: classes, instances, threads, security domains, memory, sessions, transactions, and remote connections are all allocated as necessary.

GemStone/J also preferentially locates objects that need to exchange messages in the same virtual machine, enabling local instead of remote messaging, thus minimizing the need for additional sockets.

Applications differ in their demand for these resources; therefore GemStone/J virtual machines can be configured in various ways.

Using the GemStone Console (a GUI tool), the command line, or an API, you can tailor a virtual machine to the needs of a specific application, then name the configuration and save it as a template. Different templates can be used to spawn pools of VMs configured differently for different mixes of applications; in this way, you can ensure that your enterprise's varying demands will be balanced within a pool of the appropriate kind of VMs.

If results indicate the potential for performance improvements, you can adjust a great many configuration parameters without stopping the system. You can tune aspects of memory management, disk access, and transactions while the system is running, evaluating the results and fine-tuning once more without disturbing system users.

Availability

GemStone/J offers defense in depth to ensure that the system is available when needed. Not only can you have high confidence in your primary system, but you can also set up a secondary system on stand-by in case of system failure.

- Load-balancing mechanisms in the system and within each virtual machine ensure that high demand is managed sensibly: no single application can swamp the system.
- GemStone/J processes are protected against single points of failure. One key process checks periodically to ensure that all other required processes are running. If any terminate unexpectedly, it restarts them. A watcher process in turn watches this key process and restarts it, if necessary. Client stubs know when and how to reconnect to the restarted process.
- To ensure graceful failover, you can choose among several strategies. A warm backup system can run in restore mode, continually replaying transaction logs from the system in use as soon as they are written. If the primary system fails for any reason, the backup system replays the final few transactions as clients terminate. Clients reconnect with new configuration

information pointing them to the backup system, now online and ready to resume.

Another possibility is to share disks between two machines. If the primary machine fails, the other is assigned its port numbers and IP addresses. Scripts detect system failure and restart GemStone/J, which then accesses the second machine and the same disks.

Transaction logs, backup and restore utilities, and other database administration tools help ensure data integrity, system availability, and failover capacity.

How Does It Work?

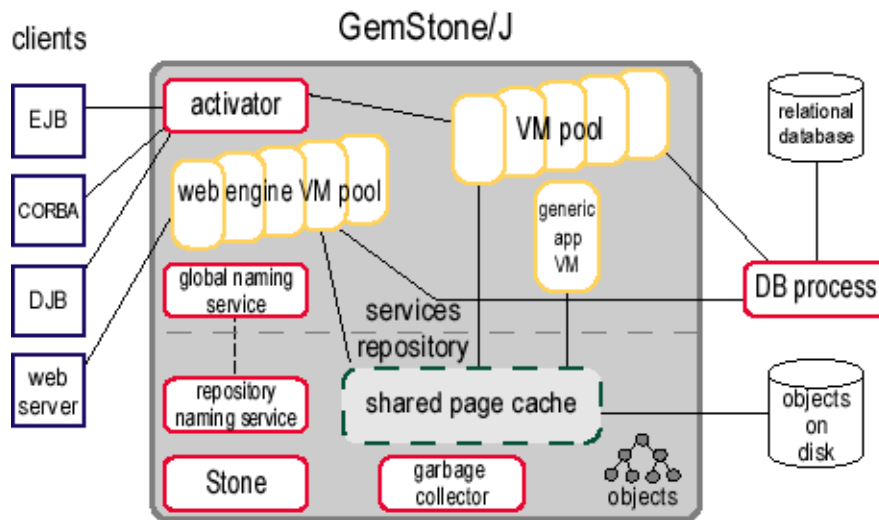
GemStone/J can be logically divided into two functional areas: the repository, and system services.

- The *repository* is disk-based storage for persistent objects and a *shared page cache*: a region of memory accessible to all applications, holding commonly used objects for rapid access.
- *System services* start, stop, and configure VMs, balance loads, manage name spaces, manage transactions, write objects to disk, and recycle memory occupied by objects that are no longer needed.

Within system services, virtual machines are special: they run your applications.

The components within GemStone/J are divided between these two functional areas as Figure 1 illustrates:

Figure 1 GemStone/J Anatomy



The Repository

The GemStone/J repository holds up to two billion Java objects or primitive types, guaranteeing their integrity and durability. At any time, up to two thousand sessions can be accessing these objects; GemStone/J prevents inconsistencies by giving each session its own view of the repository. Repository views are refreshed when transactions are committed or aborted; at these boundaries, an application may see objects change state.

GemStone/J prevents concurrent modifications to the same object by encapsulating a session's operations in atomic, independent transactions. When you commit a transaction, GemStone/J merges the modified objects in this view with the shared object repository.

To prevent concurrent modifications to the same object, GemStone/J implements both optimistic and pessimistic concurrency control; if two transactions both try to commit changes to the same object, one

of the competing transactions will fail to commit. Aborting the transaction discards uncommitted changes and refreshes the session's view of the repository. If your application must not lose the changes it's made, you can lock objects, thus ensuring that a commit will succeed. To reduce the chance of a conflict, GemStone/J also provides special collection classes that allow concurrent updates.

Transactions are logged; all successfully committed transactions can be recovered by replaying transaction logs. Transaction logs are automatically replayed upon restart in the event of a catastrophic failure such as a power outage.

Backup and restore utilities automate these common administrative tasks, which you can also perform manually as needed.

System Services

Italics denote *processes* in the following descriptions of GemStone/J's services.

Virtual Machines

Virtual machines are GemStone/J's execution engines. Each general-purpose VM is a Java virtual machine with two extra built-ins:

- a fully featured CORBA Object Request Broker (ORB), and
- access to the repository's shared page cache for reading and saving changes to persistent objects.

Virtual machines are multithreaded; each thread is a native host operating system thread. Client applications run in one or more threads. Each thread is associated with at most one GemStone/J session at a time; upon commit or abort, the thread refreshes that session's view of the repository.

Multithreaded applications need not concern themselves with which VM runs their threads; threads can run anywhere, although GemStone/J may preferentially (and transparently) run them in the

same VM for efficiency. If this efficiency gain is important for an application, you can also request such collocation explicitly.

Multithreaded applications must, however, synchronize threads at transaction boundaries; no thread should be trying to modify an object at the moment when another thread is trying to commit a change to that same object.

Virtual machines can run on multiple hosts, remote from the repository, where each host is a computer with one or more CPUs sharing access to the same memory.

Activator

The *activator* accepts and authenticates client requests, then redirects the request to an appropriate virtual machine. It starts and stops virtual machines of the proper configurations as necessary, activates requested objects, and balances the system load across the specified pools of virtual machines.

Naming Services

The *global naming service* (GNS) is the resource that provides the root naming context for a GemStone/J installation. It is a JNDI-compliant naming service visible to JNDI clients as `javax.Naming.Context`.

The global naming service can combine namespaces from more than one GemStone/J repository, thus permitting more than one repository per GemStone/J installation. Namespaces are organized hierarchically; a given repository is a branch in the path to the desired resource.

Each GemStone repository has its own *repository naming service* (RNS); application name spaces are branches within a specific repository naming service.

CORBA clients use the ORB method

```
orb.resolve_initial_references("NamingService"),
```

which returns a reference to the CosNaming service in the repository.

Transaction Management

GemStone/J supports two kinds of transactions:

- GemStone/J transactions
The *Stone transaction monitor* (Stone) coordinates commits or aborts of GemStone/J transactions and certain aspects of disk access.
- CORBA transactions
GemStone/J's CORBA infrastructure supports CosTransactions, also known as OTS—the Object Transaction Service, which coordinates simultaneous commits to the GemStone/J repository and, through JDBC, to outside databases.

Session Management

The *system agent* manages communication between virtual machines, authenticates connections, and establishes sessions. It also manages object modifications and disk access.

Garbage Collection

Garbage collectors reclaim memory and object identifiers when no longer needed. Each virtual machine runs a garbage collection thread as needed to reclaim temporary objects. The system agent periodically runs a garbage collection session to reclaim repository objects that are no longer needed.

Garbage collection takes into account distributed remote references, as well as multiple clients' differing views of the repository.

Developing and Deploying Applications

You are the expert on your application and business needs, but many customers find it helps to divide an application into the functional layers described below. The responsibilities of each layer, and the

kinds of entities used to implement it, may help you decide on the appropriate application model or mix of models.

Layer	Responsibilities
Presentation	handles input, output, and display.
Application	structures user interaction, maintains conversational state, invokes services, handles exceptions.
Services	balances resources, performs functions common to many applications.
Domain	implements domain and business logic, manipulates business objects.
Persistence	manages long-term storage, transactions that create, read, update, or delete data.

Table 1 shows the kinds of entities that you might use to implement these layers in each of the application models:

Table 1 Implementing the Layers in Various Application Models

App Model	Presentation	Application	Services	Domain	Persistence
EJB	Java client	Java objects	session beans	entity beans	RDB rows or repository objects
DJB	Java client	Java objects	GsSession	Java objects	RDB rows or repository objects
CORBA	client (Java, Smalltalk, C, C++, COBOL...)	objects	unshared server	servant or Java objects	RDB rows or repository objects
web	HTML pages	JSP or servlet	session beans or GsSession	entity beans or Java objects	RDB rows or repository objects
generic	—	—	—	Java objects	RDB rows or repository objects

GemStone/J works with a variety of development environments and also provides its own debugger for code running on either the client or the server. To aid in diagnosis, you can force both client and server

to run in the same virtual machine and examine them both with the same GemStone/J debugger.

After your application is developed, you install it into GemStone/J by a process called *deployment*. Application deployment is straightforward:

1. Invoke the Console.
2. Point it to your compiled classes and deployment descriptor.
3. Name your application.
4. Click **Deploy**.

GemStone/J:

- creates an instance of the specified application,
- creates a JNDI name space context for the application and its objects in the GNS,
- copies the compiled classes and associated resources to the GemStone/J server's system directory, and
- performs other steps as necessary, depending on the application model.

(Until the next-generation integrated web container is available, web applications are installed by means of a different process.)

Numerous other conveniences help GemStone/J developers stay productive:

- You can separate development, testing, and production environments. Because a GemStone/ installation can contain multiple repositories, you can use one repository to develop your applications, another to test them, and a third to run your production system.
- You can administer the system using the GemStone/J Console, the command line, or an API.

- A library of scripts automate common administration and testing tasks.
- You can develop GemStone/J applications using any integrated development environment that supports the version of Java in which GemStone/J is written. Consult the Release Notes to determine current compatibility requirements.

Related Topics

GemStone/J Demo Application:

<http://edemo.gemstone.com/foodsmart/>

GemStone/J Developer's Guide:

http://www.gemstone.com/products/j/trial_version.html

Developer's Center for Java Success:

<http://www.gemstone.com/javasuccess/index.html>