Java Server Programming:Principles and Technologies

By Subrahmanyam Allamaraju, Ph.D.

Summary

Building and managing server-side enterprise applications has always been a challenge. Over the last two decades, the role and importance of server-side applications has increased. The twenty-first century economy dictates that ecommerce and other enterprise applications are transparent, networked, adaptable, and service-oriented. These demands have significantly altered the nature of applications. As enterprises migrate from closed and isolated applications to more transparent, networked, and service-oriented applications to enable electronic business transactions, server-side technologies occupy a more prominent place. Despite the disparity of applications that exist in any enterprise, server-side applications are what power twenty-first century enterprises!

The purpose of this article is two-fold. Firstly, this article attempts to highlight the technical needs for server-side applications, and thereby establish a set of programming models. Secondly, based on these programming models, this article introduces the reader to the Java 2, Enterprise Edition (J2EE).

If you are a beginner to server-side programming, or the J2EE, this article will help you gain an overall perspective of what this technology is about. If you are familiar with one or more J2EE technologies, this article will provide the basic principles behind server-side programming, and help you relate these principles to specific J2EE technologies.

Introduction

Building and managing server-side enterprise applications[1] has always been a challenge. Over the last two decades, the role and importance of server-side applications has increased. The twenty-first century economy dictates that ecommerce and other enterprise applications are transparent, networked, adaptable, and service oriented. These demands have significantly altered the nature of applications. As enterprises migrate from closed and isolated applications to more transparent, networked, and service-oriented applications to enable electronic business transactions, server-side technologies become more prominent. Despite the disparity of applications that exist in any enterprise, server-side applications are the ones that power twenty-first century enterprises.

The history of server-side applications dates back to the mainframe era, an era during which only mainframes roared across enterprises. They were centralized in

nature, with all the computing—from user interfaces, to complex business process execution, to transactions—performed centrally. The subsequent introduction of minis,
desktops, and relational databases fueled a variety of client-server style applications. This contributed to a shift in the way applications could be built, leading to various architectural styles, such as two-, three-, and multi-tier architectures. Of all the styles, database-centric client-server architecture was one of the most widely adapted. Other forms of client-server applications include applications developed using remoteprocedure
calls (RPC), distributed component technologies, such as Common Object Request Broker Architecture (CORBA), and Distributed Component Object Model (DCOM), etc. Just as these technologies and architectures were diverse, so too were the nature of clients for such applications. The most commonly used client types for these applications were desktop based (i.e., those developed using Visual Basic or other similar languages). Other types of clients included other applications (viz., as in enterprise application integration) and even Web servers.
Subsequently, what we see today are Internet-enabled server-side applications. The underlying technologies for building Internet-enabled server-side applications have evolved significantly and are more mature and less complex to deal with, as you will see later in this article. Apart from providing more flexible programming models, today's server-side technologies are infrastructure intensive.

Programming Model

Having seen the two possible messaging approaches, let us now consolidate the programming requirements, as follows:
1. An infrastructure for setting up queues and topics.
2. An API for connecting to the messaging infrastructure and putting or publishing messages.
3. Infrastructure capable of registering recipients/subscribers against queues/topics.
4. An API for getting messages based on queues or topics.
In addition to these basic requirements, the following features are also be required:
1. **Message Durability:** The infrastructure should allow for message durability (that is, once posted, a message should not be lost until it is delivered).
2. **Guaranteed Delivery:** The infrastructure should guarantee delivery.
3. **Once-Only Delivery:** The infrastructure should deliver the message once and only once. This is crucial as duplicate messages may adversely affect the business processes implemented by recipients.
4. **Message Confirmation:** The infrastructure should also allow confirmation notifications to be sent to the senders.
We will examine some of these features in more detail when we discuss Java Messaging Service (JMS).

Other Server-Side Requirements

In the previous section, we focused on programming aspects that depend how clients

and servers communicate with each other.
In this section, we will discuss the two most important infrastructure-related issues applicable to server-side applications in an enterprise—*distributed transactions* and *security*.

Distributed Transaction Processing

Transaction processing is fundamental to any enterprise application that deals with data shared across various business processes.
If you are familiar with basic database programming, you might also be familiar with the basic idea of transaction processing. With JDBC[40], the `setAutoCommit(boolean)`, `commit()`, and `rollback()` methods on the `java.sql.Connection` interface let you commit or undo a group of database operations as a unit of work. Or, if you have developed client-server applications using, say, the rapid application-development programming environments from Oracle
or Microsoft, you should have used the commit and rollback operations.
What purpose do these operations serve? As you may recall, these operations allow you to permanently record (with commit) or undo (with rollback) all database changes made with a connection. For example, consider that you are implementing a few database operations. Using the commit/rollback operations, you can either complete or abort all changes。
The figure shows how you can implement a transaction using the JDBC API. With the enclosing `setAutocommit(false) and commit()`, all of the database operations performed in between are treated as a single unit of work. Rather than invoke `commit()`, you may also invoke `rollback()` such that all operations performed prior to `rollback()` will be undone in the database. The `setAutocommit(false)` and `commit()` calls let you mark the boundaries of a transaction. Databases use the notion of a connection to implement such transactions.
One of the basic requirements for transaction processing is the preservation of ACID[41] properties of the data held in databases. The acronym *ACID* stands for *atomicity*, *consistency*, *isolation*, and *durability*. These properties imply the following:
 You should be able to group multiple database operations into atomic units of work.
 Database operations should be consistent and complete with respect to the business logic modifying the data.
 Database operations performed by multiple applications on the same data should not affect each other.
 Effects of database operations should be permanent.
While these properties of data can be maintained by today's relational database systems, databases alone cannot preserve the properties in the case of server-side applications. What is special about server-side distributed applications? The next section will introduce you to the issues associated with handling such transactions in distributed server-side environments.
In order to understand the need for a distributed transaction-processing

infrastructure for distributed applications, consider the following figure。
As far as the database server is concerned, there are two different connections and, hence, two database transactions. But, can we treat these two independent transactions as part of a single unit of work? This question might seem more involved when each of the objects is operating on a different database. But, what is required to maintain a single unit of work across these operations? How can we provide the semantics for commit and rollback in this case? The basic operations defined on a connection object are not sufficient for this because we are dealing with two database connections. In order to implement a unit of work (i.e., transaction) across the two connections, both sets of database operations must be either committed or rolled back together. However, the two sets of operations happen in the context of two different database connections, each of which is local to the object that obtained the connection. Coordinating the two sets of operations such that both commit or rollback together requires a third party and uses the following approach:
1. Client (not shown) invokes a method on object A.
2. Object A informs the third party that it will perform some database operations.
3. Object A performs a few database operations.
4. Object A invokes object B.
5. Object B performs a few more database operations.
6. Object B returns.
7. Object A informs the third party once it has completed its database operations.
8. The third party verifies that all of the database operations can be either committed or rolled back.
9. The third party instructs the database to either commit or rollback the respective database operations.
10. Object A returns.
The list of steps now seems longer, but before we look at what is happening, we will discuss the role of the third party.
 Because A and B are concerned only with their respective database operations and could be unaware of all database operations being performed by the other, we need a third party to monitor all operations across both objects.
 Because A and B can commit/rollback the respective sets of database operations, we need the third party to coordinate a single commit/rollback across both.
This third party is called, *transaction manager* or *transaction monitor*; and the process of coordinating commit/rollback across the above-listed objects is called, *twophase*
commit.[42] In this process, the transaction manager polls the database(s) to find out if the database operations can be committed. If the answer is yes, the transaction manager requests to commit the operations.
As you can see from the previous list of steps, object A must inform the transaction manager that it is about to conduct database transactions before starting the
operations. Similarly, at the end of all operations, it must inform the transaction *demarcation*. It is the process through which boundaries are created for transactional

operations. All operations conducted within these boundaries are treated as a single unit of work.

Note: In addition to object A, from the previous example, the client can also demarcate the transaction.

The Open Group[43] (formerly the X/Open group) specified the first widely used transaction-processing model, which is known as the *distributed transaction processing model* (DTP).[44] This model defines a set of APIs, which are the basic building blocks for distributed transactional applications. Following are two of the APIs:

 XA Interface: This defines operations between databases and transaction managers such that databases can register themselves with the transaction manager and the transaction manager can interact with databases to coordinate transactions.

 TX Interface: This defines operations that the transaction manager should implement such that applications (objects A and B above) can demarcate transactions and control transactions when required.[45]

Most databases support the TX interface and the database part of the XA interface. Commercial transaction processing systems[46], such as Tuxedo (from BEA Systems, Inc) and Encina (now part of IBM's WebSphere suite of products), implement the TX interface for transaction managers.

Another transaction-processing model is Object Management Group's (OMG's) [47] Object Transaction Service (OTS).[48] This model is intended for distributed object applications that use CORBA, and it specifies certain interfaces that support distributed transactions across CORBA objects.

In the Java realm, the following are two APIs that deal with distributed transactions:

1. **Java Transaction Service (JTS)[49]:** This is the Java mapping of the OTS; it is suitable for implementing Java transaction managers.
2. **Java Transaction API (JTA)[50]:** This is a high-level transaction API for Java server-side applications; it is the model used for transaction processing in J2EE applications.

These technologies provide transaction management programmatically (that is, the clients/objects are responsible for transaction demarcation). The J2EE, as well as the Microsoft Transaction Server (MTS), provide an alternative mechanism called, *declarative transaction processing*. Using this process, you can enable transaction processing via certain configuration rather than implementing some transaction logic.

Security

In both business and technical circles, security is one of the most commonly misused and misunderstood concepts. Despite security concerns expressed by various parties who deal with enterprise application development and usage, security is the one of the least cared about requirements. One reason for this is because security lapses can occur at any of the following levels:

1. **Network Security[51]:** The lapses at this level include breach of the networks and servers. Solutions for such breaches involve firewalls, demilitarized zones, etc.

2. **Operating System Security**[52]**:** This is yet another level of security that deals with protecting various operating system-level resources (files, threads, processes, etc.). Once an intruder breaks the network, the operating system is the next target for accessing operating system-level resources.
3. **Server-Side Application Security:** This level deals with preventing unknown or unauthorized users from accessing server-side applications.
4. **Database Security:** In addition to the previous levels, database vendors also provide their own security mechanisms to protect data from unauthorized access. Because we are dealing with server-side applications in this piece, coverage of security will be limited to security mechanisms at the application level.
In enterprises, security concerns arise whenever applications provide resource access to several internal, as well as external, users and client applications.
As far as security is concerned, a resource could be business data (i.e., a purchase order or customer profile) maintained by various enterprise applications, or it could be the execution of, or participation in, a business process implemented by one or more enterprise applications. The following list shows some typical security concerns with enterprise applications:
 Identity of Users—Authentication: When your applications deal with privileged business processes and business sensitive data, it is important to establish the identity of a user/client before providing the user/client access to data or business processes. The process of *authentication* helps establish user/client identity. Authentication commonly involves the user/client exchanging a token (a password) with the application to which the user/client is trying to gain access. For this mechanism, the token should be known to both the user/client and the application. (More elaborate mechanisms include digital certificates, biometrics, etc.)
 Allow/Bar Access—Authorization: Authentication alone is not always enough, particularly when there are different types of users/clients with different privileges in a given environment. In a typical enterprise, you may encounter different users participating in different business processes under different identities and roles. Only those users/clients allowed to participate in certain business processes or access to some data should be allowed to access/do the task. All other users/clients without the required identity or role should not be provided access. The process of *authorization* (sometimes called, *access control*) provides or denies access based on the identity/role of users. (Note the difference between authentication and authorization. While authentication deals with establishing identity, authorization deals with providing or denying access based on identity.)
 Wire-Level Security—Data Confidentiality and Integrity: With today's Internet-networked applications, data travels via several public networks, raising concerns over whether or not someone should be allowed to record/monitor or even modify the data while in transit. Depending on the nature of the data, such breaches may lead to consequences that are very harmful to businesses. In order to prevent this, it is necessary to guarantee data confidentiality and integrity. *Data confidentiality* prevents a person from beginning to decipher the data while in

transit. *Data integrity* involves of the introduction checks that enable communicating parties (clients and servers) to detect any type of tampering with the data. Digital signatures, various encryption technologies[53] and protocols, such as Secure Sockets Layer (SSL), help in maintaining data confidentiality and integrity.

Following are key requirements for the server-side applications and programming models discussed in this article:

1. **Means to authenticate users and clients:** some means for authenticating users and clients and for maintaining the identity across all parts of an application. (This requirements holds for Web-based and other forms of clients).
2. **URL access control:** This is required when server-side applications can be reached via HTTP.
3. **Control of method invocation:** Control over the invocation of methods on remote objects.
4. **Control of message queue access:** Control over the access to message queues and topics.
5. **Ability to plug technologies at the wire level:** The ability to plug technologies at the wire level guarantees data confidentiality and integrity.

We will see how these are achievable in J2EE applications in the next section.

J2EE for Server-Side Programming

Up to this point in this article, we have discussed various principles behind server-side computing. During the discussion, though I have made references to some of the server-side Java technologies, I have intentionally left out details of how these principles map to various server-side technologies with Java. From this section forward, we will discuss specific Java server-side technologies within J2EE.

The J2EE is an integrated platform combining various server-side programming models previously discussed in this article. Sun Microsystems introduced J2EE in late 1999[54], though some J2EE APIs, such as servlets, Java Naming and Directory Interface (JNDI), etc., were introduced before; Sun Microsystems integrated all such technologies with a coherent and integrated architecture for server-side programming with J2EE. Today, J2EE is the de facto standard for building server-side and Internet applications using Java and is endorsed by several vendors offering J2EE-compatible application servers.

J2EE is essentially a specification that stipulates how the various J2EE technologies work together. Each of these technologies is discussed in various specifications released by Sun Microsystems. Sun maintains a reference implementation of J2EE. Nonetheless, J2EE is still a specification with several commercial and open-source implementations. Although such implementations are typically referred to as *J2EE application servers*, we will instead use the term *J2EE platform* to indicate an implementation of the J2EE technologies. J2EE includes the following technologies:

 Enterprise JavaBeans (EJBs): Used for developing distributed components (http://java.sun.com/products/ejb).

 Java servlets and Java Server Pages (JSP Pages): Used to build Web-based applications
(http://java.sun.com/products/servlet, http://java.sun.com/products/jsp).
Let us now discuss how J2EE provides for each of these needs.
Before we go into the details on how these programming models are supported, we will look at some of the most commonly used J2EE terminology, as follows:
1. **Container:** We use the term *container* to denote a runtime that supports application programs. There are two types of containers for developing server-side applications using J2EE.
- **EJB Containers:** EJB containers are capable of hosting server-side objects that are developed for the synchronous request-response model. In the upcoming enhancement via J2EE1.3 and EJB2.0, EJB containers are also capable of hosting server-side objects that can be activated asynchronously via messages.
- **Web Containers:** Web containers are capable of hosting objects developed for the stateless request-response model with communication over HTTP. The Web container maintains instances of Java servlets and JSP pages

2. **Application Components:** The J2EE specification uses the term *application components* to denote container-managed objects, such as EJBs, Java servlets, and JSP pages. These are managed, as the container (i.e., the runtime) maintains the lifecycle of these instances. (Developers do not explicitly create these instances.) The containers also manage such things as transactions and security as we will see later in this article.
3. **Modules:** Apart from the notion of application components, J2EE employs the notion of modules. A *module* consists of one or more of a given type application components. For instance, an EJB module consists of one or more EJB components and, similarly, a Web module consists of one or more servlets and JSP pages (along with static content, such as HTML files, images, etc.). The primary purpose of a module is to allow flexible packaging structure. In J2EE, modules are packaged into Web archive (WAR) or Java archive (JAR) files.
4. **J2EE Applications:** A J2EE application is a combination of or more modules. J2EE applications can be packaged into enterprise archive (EAR) files. Apart from serving as a means for high-level application module packaging, a J2EE application can also define virtual boundaries between various applications deployed on a given container.
5. **Deployment:** This is another word you will encounter frequently in the J2EE world. While a strict definition of deployment is outside the scope of this article, in simple terms, *deployment* is the process of adding your applications to a J2EE platform. This could mean simply copying the classes (suitably packaged as applications) into a directory accessible by a J2EE platform implementation, or it could mean using vendor-specific deployment tools to add applications to a J2EE platform. Once deployed, J2EE application components are available for invocation by J2EE containers based on clients' requests (i.e., requests from Web

browsers or other clients) or even other J2EE application components.
6. **Deployment Descriptor:** In simple terms, a *deployment descriptor* is a configuration file expressed using XML. Along with application components, each module consists of a deployment descriptor. Depending on the type of application component, the deployment descriptor consists of specific configuration information. Deployment descriptors help containers to not only decipher the contents of modules, but also to let containers manage the lifecycle of various application components, transactions, and even security.