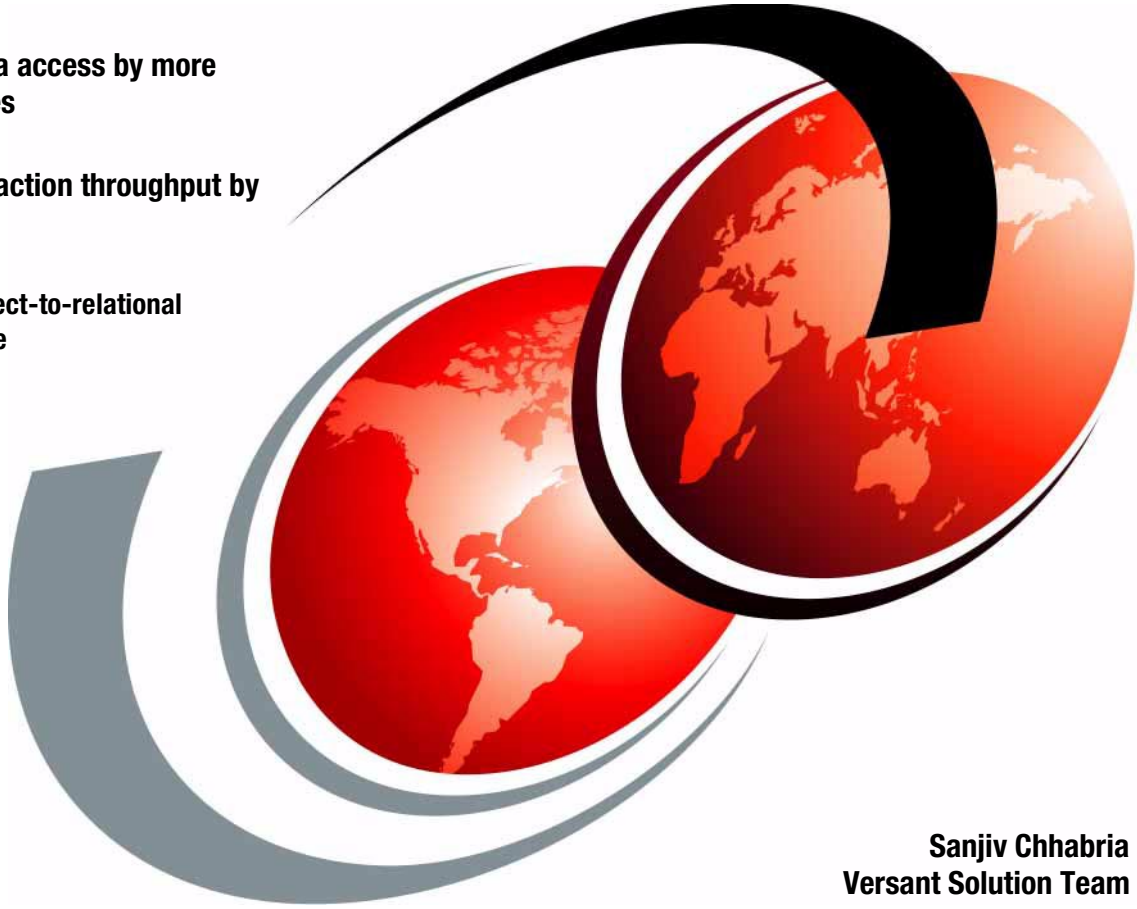
Accelerating IBM WebSphere Application Server Performance with Versant enJin

- Improve data access by more than 50 times
- Speed transaction throughput by 10 times
- Eliminate object-to-relational mapping code



Sanjiv Chhabria
Versant Solution Team

Redbooks

ibm.com/redbooks

International Technical Support Organization

Accelerating IBM WebSphere Application Server Performance with Versant enJin

November 2002

SG24-6561-00

Note: Before using this information and the product it supports, read the information in “Notices” on page v.

First Edition (November 2002)

This edition applies to Versant enJin 2.1.0/2.2.3 and WebSphere Studio Application Developer 4.0.2 on Windows NT 4.0 (SP 6).

Contents

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

Balance®	Redbooks™	Redbooks (logo)™ 
DB2®	SP™	VisualAge®
IBM®	VisualAge®	WebSphere®
Netfinity®	WebSphere®	
Perform™	IBM eServer™	

The following terms are trademarks of International Business Machines Corporation and Lotus Development Corporation in the United States, other countries, or both:

Lotus®	Word Pro®

The following terms are trademarks of other companies:

Versant and enJin are trademarks or registered trademarks of the Versant Corporation in the United States and/or other countries

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

C-bus is a trademark of Corollary, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.

Preface

Application servers have become the choice technology for companies building Web-based distributed e-business applications. Although this has proven to be successful in a number of areas, many organizations still struggle with several performance bottlenecks. Using IBM WebSphere and Versant enJin provides a framework for a solution that addresses some of these challenges - especially the complex object-oriented models of enterprise applications

Versant enJin integrates seamlessly with IBM WebSphere and provides the performance and scalability you need to accelerate your e-business transactions.

The integration of Versant enJin with IBM WebSphere can result in up to 50 times improvement in the performance of your e-business application. This is done by elimination of mapping objects to and from the relational world. In addition, enJin also enables you to get your products to the market faster by eradicating the need to know or develop in any language other than Java - no SQL, no JDBC or relational database programming.

Versant enJin is a flexible infrastructure platform that persists Java objects and EJBs within the middle-tier without overloading your existing database systems. enJin gives you J2EE capabilities, combined with a transparent Java object persistence implementation. It gives you the performance and scalability you need to accelerate your e-business transactions across the Internet. Use Versant enJin with IBM WebSphere Application Server to increase system throughput, time-to-market, and transactional response time.

This IBM Redbook is an in-depth guide for implementation of Versant enJin Session Bean Persistence methodology, development and deployment of your J2EE applications using IBM WebSphere Application Server.

It describes in detail the benchmarking project that was a joint undertaking with IBM and Versant. In the latter part of the book you will learn about the benchmarking project that was conducted at the IBM Silicon Valley Laboratory in San Jose, California, in cooperation with the IBM High Volume Web Sites team.

In Part 1 we will describe and familiarize the reader with the methodology used in developing J2EE applications using Versant enJin, as well as provide information about enJin that will enable your team to fully understand the components, architecture, benefits, and features of enJin.

In Part 2 you will learn how to implement the methodology in developing and deploying a typical J2EE application using IBM WebSphere and enJin. This will

be done building and deploying a fully developed typical EJB Application using Session Managed Persistence. This will be done using WebSphere Studio Application Developer. A downloadable JAR file will allow you to run and explore the example rapidly. In presenting the example that we use, you will learn the same methodology that resulted in the performance that we accomplished in the benchmarking project, and you will be able to utilize the same principles for your own example.

In Part 3 we will share with the user the details of the benchmarking project that was done - including the methodology, the application architecture, the performance metrics, and hardware configuration. The final part of this section is dedicated to present the users with the test results and analysis, as well as a summary of the project.

The team that wrote this redbook

This redbook was produced by a team of Versant specialists from around the world working with the IBM International Technical Support Organization, San Jose Center.

Sanjiv Chhabria is a technical writer at Versant Corporation. His experience includes writing technical and user manuals and has developed the documentation suite for Versant enJin. His previous experience includes writing user manuals, data modeling, and writing applications for small businesses. His current area of expertise includes application servers and EJB.

Thanks to the following people for their contributions to this project:

- The team at Versant without whose contributions this book would not have been possible.
- Tom Miura, Executive Vice President and Chief Operating Officer, Versant Corporation, Fremont, California
- Joe DeCarlo, Manager, IBM International Technical Support Organization, San Jose, California

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  ibm.com/redbooks

- Send your comments in an Internet note to:

  redbook@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. 1WLB Building 80-E2254
  650 Harry Road
  San Jose, California 95120-6099

Part 1

Introduction to Versant enJin

In this part we will describe and familiarize the reader with the methodology used in developing J2EE applications using Versant enJin as well as provide information about enJin that will enable your team to fully understand the components, architecture, benefits and features of enJin.

1

Overview of enJin

Versant enJin is a scalable, robust, infrastructure platform offering advanced object management for Application Servers. Versant enJin provides core object-management services of persistence, and intra-tier replication that are essential in building Java middle-tier applications. The platform is ideal for applications that have complex object models and require high performance, scalability, and fault-tolerance.

Versant Corporation's enJin provides a middle-tier object cache, which allows J2EE applications to access enterprise data without having to synchronously store or retrieve this data from a relational database during every transaction. So, within each business transaction, the user does not need to wait for the information to be retrieved from or propagated to backend systems, before the application can respond to the user's request. For this reason Versant calls its enJin product a *transaction accelerator*.

enJin however, does not merely provide a copy of relational data in each application server. Instead, enJin transparently replaces the persistence and transaction management capabilities of its host application server, in order to provide transparent state management of objects such as Entity Beans without the developer having to write enJin-specific code. This object cache is built upon Versant's sixth generation object-oriented database technology (the Versant ODBMS), and so provides a robust object caching facility with well-proven persistence capabilities.

A simple in-memory caching mechanism would lose committed data if the application server failed, or would become inconsistent between application servers within a cluster. Since enJin is built upon the persistence mechanisms of an enterprise class ODBMS, however, all committed data is stored in a persistent object cache, which is able to survive server crashes. enJin also overcomes any possibility of data inconsistency between application servers in a cluster, by maintaining cached objects both in an application server in-memory cache (for maximum performance), as well as a shared cache accessible to all application servers, so that cached data is guaranteed to be consistent between all application server instances.

Specific topics covered in this section include:

- An overview of Versant enJin
- Architecture of enJin
- Benefits of using enJin
- Key features

1.1 Overview

As application servers have become the platform of choice to build Web-based, distributed e-business applications, customers may find their traditional relational database a performance bottleneck. Whether you are a global enterprise with vast stores of legacy data, or an Internet startup, the performance issue can hamper your ability to remain competitive in the new economy.

Versant enJin provides the additional performance and scalability you need to accelerate your e-business transactions across the Internet. Versant enJin is a flexible infrastructure platform that seamlessly persists Java objects and Enterprise JavaBeans (EJB) within the middle-tier, without otherwise overloading your existing database systems. And your company can count on the reliability of Versant enJin with its fault-tolerant architecture and support for high availability and clustered environments.

For global enterprises, a key challenge is to propagate the business transaction from the application server or middle-tier to the back-end line of business system, which is typically a relational database. The line of business system has no built-in ability to handle the complexities of the object-oriented models deployed in the application server. Therefore, an *impedance mismatch* occurs in moving from one model to the other. A direct impact, aside from the development, testing and maintenance overhead imposed, is the effect on performance. Reconstituting an object from relational tables will involve n-way joins and sorting, a time consuming process not in line with today's real-time processing requirements.

Next-generation Web applications such as B2B exchanges, aggregated portals, and wireless workforce applications also require a robust and highly performing infrastructure. These applications have the ambitious goal of providing differentiated user experiences while aggregating information from a variety of disparate sources. They also use the Internet to integrate information from several sources such as internal applications, partner applications and third party services.

To meet these challenges and support the emerging requirements we now need a set of middle-tier services that help the application developer rapidly implement Web-based applications that are reliable, robust, and scalable. Middle-tier solutions are being used today to manage transactions, security, and data access, greatly reducing the programming burden on application developers. But to be successful, your company needs to support the following advanced middle-tier services:

- Content management
- Workflow
- Personalization
- Real-time analysis
- Publish-subscribe

Versant's approach, unique in the industry, manages these middle-tier services by separating the back-end data, typically stored in a relational database, from the middle-tier. Versant calls data found only in the middle-tier *intermediate data* to support the application logic running there. Examples include:

- *Business intelligence* data, data captured in the middle-tier and used to provide personalized marketing.
- *Meta data*, data that describes how to interact with back-end systems and legacy databases.
- *Workflow data*, data that describes business rules and the state of ongoing business processes.
- *Session data*, data that is relevant only to the on-going interaction with an e-business application, for example, shopping cart data.

While many tools exist for managing business data, intermediate data is often created in an inefficient and ad hoc manner. And while in-memory caching solutions provide an object layer on top of an existing relational database, they lack the agility needed to scale and develop new applications rapidly. Versant enJin lets developers isolate the transactional load to the middle-tier and alleviates the impact of the Internet on business logic. Additionally, Versant enJin automatically propagates the business transaction to the relational database upon completion or through rules-based policies. The upshot is increased

flexibility and scalability without compromising robustness or reliability. Customers can realize up to 50 times application performance improvements and can speed time-to-market by up to 40% reduction in development time frames.

1.2 Architecture

As illustrated in Figure 1-1, enJin exists in the middle tier in conjunction with the business logic provided by Session and Entity Beans running within a J2EE Application Server environment. When Session or Entity Beans read data, they simply read the objects from the in-memory cache running within the same application server - and this data is guaranteed to be consistent with the object caches of all other application servers within the cluster, by virtue of the synchronization mechanism between the in-memory cache and the shared persistent object cache. Similarly, when a Session or Entity Bean updates an object, it is transparently updated to the in-memory object cache, and within the boundaries of the transaction, is updated to the shared persistent cache - this guarantees the integrity of the transaction.

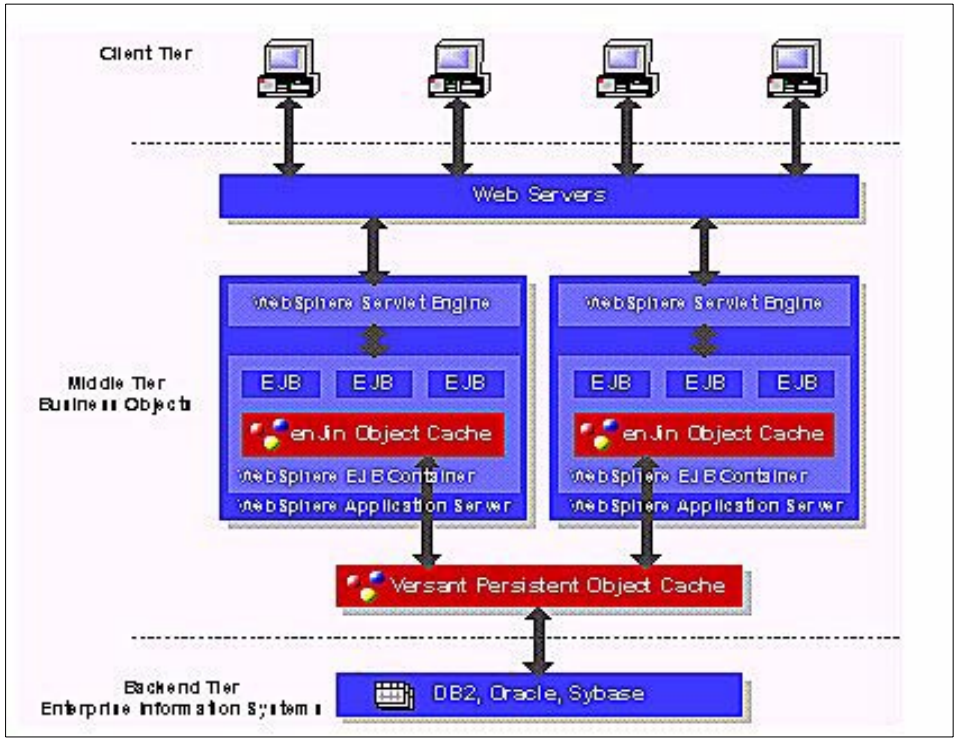


Figure 1-1 enJin architecture

A variety of approaches can be used to update the relational database tier, but typically the backend is not touched until a business transaction is completed.

Since the object to relational updates are performed asynchronously, the processing time required to do this work of backend updates is not incurred within the context of a user's transaction, but rather, it is delayed until after the application server has responded to the user's request. This enables response times to be dramatically improved. Furthermore, since the resources within the application server do not have to be locked waiting for the enterprise information systems to be updated, these resources can be freed more quickly, and can be utilized for other transactions more quickly. This faster freeing up of application server resources also enables the throughput to be greatly improved. All of these performance gains come primarily from the delaying of the object to relational translation work and reading and writing to the enterprise back end systems - and these gains are made without sacrificing any transactional integrity. Versant's Persistent Object Cache guarantees the ACID properties of the transactions: atomicity, concurrency, isolation, and durability are all maintained.

1.3 Benefits

What follows is a list of the benefits of Versant Corporation’s enJin solution.

1.3.1 Ease of development

enJin allows you to leverage its transparent object persistence capabilities. It allows developers to use Bean or Container Managed Persistence, JavaServer pages or servlets. You can define complex data models without the need for mapping code. In addition, as enJin allows the developer to utilize its resource and connection pooling capabilities for integrated connections and transaction management.

1.3.2 Time-to-market

With enJin, the development effort can focus on modelling of business objects. Due to it's transparent object persistence capabilities, there can be up to a 40% reduction in development time, since there is object sharing and reuse, and the resultant significantly less code that is required.

1.3.3 Scalability

enJin's enables caching of data within the application server environment. This can reduce the load on existing business systems and allows for 1000's of concurrent users accessing common data. EnJin scales way beyond the limitations of simple in-memory caching solutions, and is able to handle hundreds of gigabytes of data.

1.3.4 Performance

enJin significantly reduces response and wait times by de-coupling completion of transactions form back-end systems. It is advanced navigational access mechanism, allows traversing from one object to another, eliminating the use of *expensive joins*.

1.3.5 Availability

enJin's proven technology is a result of over 500 years of development effort. It provides support for application clustering, synchronization of data between multiple application servers, and fault tolerance with automatic failover and recovery.

1.3.6 Transparent data distribution

enJin enables seamless distribution of data in the middle-tier, which can be partitioned over multiple machines allowing developers to harness additional hardware for successful deployment and scaling. Multiple sharing of enJin by application servers allows concurrent access to shared data facilities and workload-balancing.

1.4 Key features

What follows are key features of Versant Corporation’s enJin solution.

1.4.1 Persistence for Java objects and EJBs

Persistence for Java objects, bean management, and container management is a key feature of the enJin solution.

Java persistence

Seamless persistence for Java objects has accelerated the development of e-business applications by over 40% when compared to traditional JDBC-based development. Versant enJin provides Object Data Management Group (ODMG) compliant APIs and in the future will also support Sun Microsystem's Java Data Objects (JDO) APIs. Versant enJin provides transparent persistence for Java objects without any changes to the class model; Java objects are stored and shared between applications, as is, without any mapping to/from a relational data model.

EJB persistence

For integration with leading EJB Application Servers, Versant enJin supports both Bean Managed Persistence (BMP) and Container Managed Persistence (CMP) in accordance with the EJB 1.1 specification. This enables programmers to take full advantage of Versant enJin's high performance object storage without having to explicitly manage database connections, transactions, or persistence. In the future, Versant enJin will support the Java Transaction API (JTA) to ensure co-ordination of transactions with other compliant data sources.

1.4.2 JavaServer pages and servlet support

A key component of today's e-business application is to provide highly customized information and presentation to end-users. enJin integrates seamlessly with the Web components of the J2EE standard and provides compliant APIs to fully leverage the functionality of building Web-based applications that serve dynamic content and allow access to seamless persistence to your Java objects.

1.4.3 Data replication

Increasingly, as e-business applications are deployed to multiple sites and data centers, the ability to replicate data is crucial. This is true for performance reasons, as well as to provide for high-availability of service. Support for *intra-tier* replication is achieved using Versant enJin's asynchronous replication capability. This capability provides an open framework for master-slave or peer-to-peer replication between multiple instances of enJin located over a LAN or a WAN. The flexible configuration allows batch or transactional replication driven automatically or explicitly via API, and supports conflict detection and resolution in the case of duplicate updates.

1.4.4 Active data management

You often need the ability to build solutions that are able to react to changes. Versant enJin provides an automatic notification capability that you can use to register interest in changes to your Java objects, and automatically receive notification as these changes happen. This capability is at the core of enJin's data replication feature, and is available to you to develop your own event-driven solutions.

1.4.5 XML interchange

The ability to handle XML data is vital to many e-business applications, Versant enJin supports conversion to and from Java objects stored in the middle-tier and XML. XML documents can be generated from object graphs and object graphs can be generated from XML documents. This capability can be used to exchange information with different and disparate sources and/or provide a means of simplifying manipulation of complex XML documents by first converting them to objects.

1.4.6 Hot-standby for e-business transactions

Today's e-businesses require 100% uptime. Versant enJin can be used in conjunction with application server clustering and/or hardware clustering and high availability solutions. In addition to this enJin offers a fault tolerant capability that provides automatic fail-over and recovery. This means that Internet transactions continue to run, uninterrupted, even in the event of a major system failure. On recovery enJin performs an automatic re-synchronization before recommencing normal fault tolerant operation.

1.4.7 Integration with existing technology

Versant enJin supports today's heterogeneous enterprise ensuring preservation of investment as platform and technology choices change. Versant enJin integrates with IBM's WebSphere and BEA's WebLogic Application Servers and in addition can work with any Java application server using its generic integration framework. Versant enJin runs on Microsoft and all major UNIX platforms.

1.4.8 GUI tools

Your learning curve associated with utilizing enJin's powerful utilities is minimal. We provide a set of GUI tools that can be used from the Developer Console with enJin or allows you to develop your application using the leading IDE's (Jbuider and WSAD) simply by point and click. These include:

- Persistent Tool - which allows you to specify the classes you want to mark as persistent and access transparently
- Class Enhancement
- Easy-to-use wizards that allow you to implement enJin specific code.
- DBAdministrator - allowing you to create, delete, and perform a number of database operations
- DBInspector - allowing you to inspect the contents of your database

2

Key concepts

Versant enJin is a powerful and comprehensive transaction accelerator. With the integrated Java and EJB development and runtime environment that is provided, building high-performance, scalable, and robust enterprise applications need not take the hundreds of man-years that developers were used to. enJin provides the platform for you to do this with up to 40% less code and 50 times better performance.

enJin readily integrates with leading application servers like IBM WebSphere, thereby providing a complete *solution-in-a-box* for your component-based development and deployment.

enJin leverages the proven abilities of Versant's object database to handle Java objects, data complexity, and transactions throughout the middle tier. enJin's object-relational mapping can provide direct access to relational data where required, and coupled with replication techniques can be used to propagate business transactions near-synchronously or asynchronously depending on the need.

As a new user we can understand your excitement in deciding to use enJin. But before you start developing with enJin, there are some concepts that we feel you need to familiarize yourself with, so that you have a good understanding of what happens behind the scenes. These concepts are integral to your success with using enJin. The remaining of this chapter will cover these concepts, some that you may already be familiar with, and some that may be new to you - the unique features of enJin.

The topics covered in this chapter include:

- enJin’s transparent Java language interface
- Class enhancement
- Persistence categorization
- Versant enterprise container

2.1 enJin’s transparent Java language interface

enJin's Java Versant Interface (JVI) provides complete transparent persistence for Java objects without any changes in the Java class model. Java objects are stored and shared between applications without mapping. In addition, JVI provides a rich interface allowing developers to take full advantage of the performance benefits of enJin's object management capabilities.

Topics included in this section include:

- Overview of JVI
- Architecture of JVI
- JVI operations

2.1.1 Overview

JVI combines the features of the Java programming language and the enJin's Versant Object Database to provide efficient, easy-to-use storage of persistent objects.

It is built on Java, a language whose flexibility and performance has proven valuable on all levels of complex two and three-tier applications. JVI embraces the Java philosophy, providing persistence that fits naturally in the Java language.

All JVI programs are written using pure Java syntax and semantics. No special keywords have been added to the language, and no awkward interface is needed to access persistent objects.

Objects of nearly any Java class can be stored and accessed persistently. Most kinds of objects can be stored, including elemental values, such as strings and integers, and Java references to other objects. With JVI, you can access the fields of a persistent object directly in the application without writing tedious data mapping routines.

Java has built-in support for multiple threads of execution working simultaneously in an application. JVI allows each thread to operate in shared or independent transactions.

JVI is tightly integrated with the enJin's Versant object database with a comprehensive set of features that includes:

- High-performance, fully transactional object storage, access, and query
- An advanced 2-tier architecture with full caching of objects on both client and server. All cache synchronization occurs automatically at transaction boundaries.
- Sophisticated locking models including standard and optimistic locking
- Event notifications (such as creation, deletion, and modification)
- Fault-tolerant data replication with online fail-over operation
- Support for rich object models including class inheritance and direct object references

2.1.2 Architecture

What follows is a brief overview of the JVI architecture.

Bindings

The JVI Transparent binding unifies the Java language with the Versant object database. It allows applications to be written in a very natural way, where some of the Java objects are completely persistent and transactional. Persistent Java classes do not use any special syntax or conventions; instead you declare classes to be persistent using a special tool, the enhancer. Your Java program can use a persistent object just like any other Java object. For example, you can call its methods; read and write its fields, use object references and collections; and let other, possibly persistent, objects reference it.

Balanced client-server network architecture

JVI is an application interface to the Versant object database, whose structure consists of multiple servers connecting to remote application clients. The Versant client library caches objects, providing fast object access and navigation within the application server, while queries can be executed at the server using the Versant Query Language (VQL). Unlike a relational database system, the server will not ship an object to the client if the object is already contained in the object cache.

Database access with transparent bindings

Transparent binding helps to solve this problem of writing laborious mapping code by merging the Java and database object spaces. With transparent binding, persistent objects are seen in the application as regular Java objects. JVI automatically manages the reading, writing, and locking of persistent objects as they are accessed by the application.

2.1.3 JVI operations

This section of the document will demonstrate the ease of using enJin's transparent object management into your applications. The complete transparent persistence for Java objects makes object management effortless. If you are familiar with object-oriented development and Java, you can immediately start to use the power of enJin to enhance your application performance. In this section we will build a simple application through which you will learn how to use enJin to:

- First steps
- Create persistent objects
- Read (accessing) persistent objects
- Update (to write changes) persistent objects
- Delete persistent objects

First steps

Prior to proceeding with enJin's JVI operations there are two concepts that we would like to introduce, as both of these concepts are unique to enJin, these will provide a foundation that will facilitate comprehension of enJin's Transparent Java Interface.

This section will cover:

- The TransSession class
- Locking

The TransSession class

enJin provides the TransSession class to encapsulate a connection with the Versant object database, similar to what you may have experienced with JDBC while establishing connectivity with your relational database except, the TransSession class does much more. It is not the intention of this document, to coach users on how to be experts at using enJin's Java Versant Interface, and in this instance, the TransSession class. However, to be able to leverage the transparency provided by enJin, users do need to have to some knowledge about this class and its methods. A brief introduction to this class and its methods are presented below. For more details, please refer to the JVI User's Manual, which can be found in the <ENJIN_ROOT>/ doc/jvi directory.

This class embodies the Transparent Session concept and provides methods that allow application programs to interact with the Object Database. Listed below are the methods that we will be using in subsequent sections, as well as some that we will not use, but may be useful to you when using enJin.

```
makePersistent:
public void makePersistent(java.lang.Object obj)
```

```
Makes an object persistent (after commit).

makeRoot:
public void makeRoot(java.lang.String rootName, java.lang.Object obj)
Associates an object with a unique name and makes the object persistent (after
commit)

findRoot:
public java.lang.Object findRoot(java.lang.String rootName)
Returns the object with the given root name (after commit)

deleteRoot:
public void deleteRoot(java.lang.String rootName)
Deletes the named root association (after commit)

deleteObject:
public void deleteObject(java.lang.Object obj)
Deletes a persistent object (after commit).

endSession:
public void endSession()
Ends the current session and implicitly commits the current transaction.

commit:
public void commit()
```

This commits the current transaction and implicitly starts a new one. This method commits all object modifications within the session and releases any related locks held by this transaction. An object modification could be an update of any field of an existing persistent object, or an update or creation of a persistent object.

Objects explicitly made persistent through the makePersistent() method call, or those deleted through the deleteObject() method call would also become committed.

Locking

enJin provides several options for locking. Locks provide guarantees that allow multiple processes to access the same objects at the same time in a cooperative, controlled, and predictable manner. There are two types of locking models: optimistic and pessimistic. Pessimistic locking is the default locking model in enJin.

This section will provide a brief description on both models. For a more detailed description, please see your ODB documentation provided in your installation of enJin:

- Types of locks

- Pessimistic locking
- Optimistic locking

Types of locks

Two types of locks are used for concurrency control.

A read lock (RLOCK) is a shared lock and any number of applications can read the same object at the same time.

A write lock (WLOCK) is an exclusive lock and only one application is allowed to write an object at a time.

Read and Write locks will block each other. A read lock will block a request for a write lock and the request will be queued until all the read locks are released. A write lock will block a request for a read lock and will be queued until the write lock is released. Of course, a read lock will not block any other request for a read lock.

Pessimistic locking

As mentioned earlier, this is the default locking model with enJin. When an object is first read into cache a default lock (typically a read lock) is placed on the object. When it is first modified the lock is implicitly upgraded to a write lock. With pessimistic locking locks are held for the duration of the transaction and only released after a transaction is committed or aborted.

Optimistic locking

The default pessimistic locking model is appropriate under most circumstances, however, suppose you have the following scenarios:

- You want to read a large number of objects over a long period of time, but update only a few.
- But holding locks on all the objects could interfere with the work of others.

For situations such as these, enJin provides optimistic locking that allows you to work with objects without holding locks.

For a more detailed description, please see your ODB Documentation provided in your installation of enJin.

Creating persistent objects

Persistent objects created with the transparent binding are cached in Java memory. To create a Java class for persistent objects, first write the .java source code file as usual. Note that there is no special syntax necessary when defining the classes. No changes are necessary to allow objects of this class to be stored in the Versant object database. These changes are done automatically for you in the enhancement process.

Our first example will use a Person class, which is included in the <ENJIN_ROOT>/examples/JVI/tutorial/trans directory. To indicate that instances of Person are persistent objects, a corresponding line will be added to the configuration file that is read during the enhancement process. Enhancement will be described in greater detail in the section on Class Enhancement.

Note: This and the subsequent examples requires enJin's object database in which to store objects. Thus, our first step is to create this. The name of the database is arbitrary; we will use the name tutdb. This can be easily changed, since all of the examples take the database name as a command-line parameter. Creating a new Versant Object Manager can be accomplished in two ways:

- Using the command line
- Using WSAD IDE (with the enJin's Integrated tools)

Using the command line to create the database

There are two steps to creating the database using the command line.

Make the Object Manager Directory

Before creating a new Object Database you must create a subdirectory for it under the VERSANT root database directory. To create the database directory for the Object Manager tutdb, run the makedb utility:

```
makedb tutdb
```

This creates a subdirectory, owned by you, under the VERSANT root database directory. To see the location of this root directory, use the oscp utility:

```
oscp -d
```

In addition, the makedb utility creates front end and backend profiles; we will not be using these profiles in the tutorial. Consult your VERSANT Database Administration Manual for more information on database profiles.

Create the Object Manager

To create the Object Manager, use the createdb utility:

```
createdb tutdb
```

This creates the database data volumes and log files in the database directory tutdb.

Using the WSAD IDE to Create the database

To create the database, run the DBAdministrator tool by either clicking the DBAdministrator icon, or select enJin DatabaseTools>DBAdministrator, as shown in Figure 2-1.

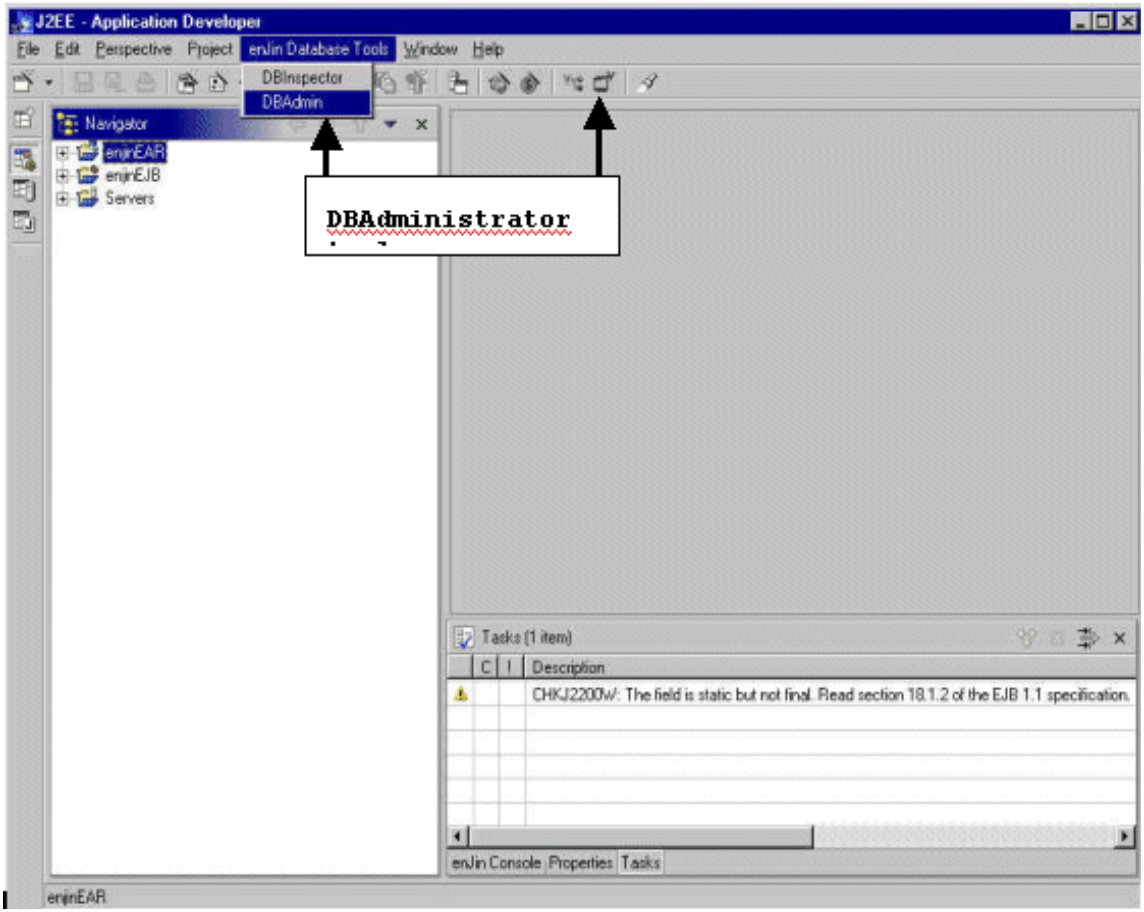


Figure 2-1 DBAdministrator

Doing so, launches a new window as shown in Figure 2-2. Simply select **File -> Create Database**, and this will display a new dialogue box.

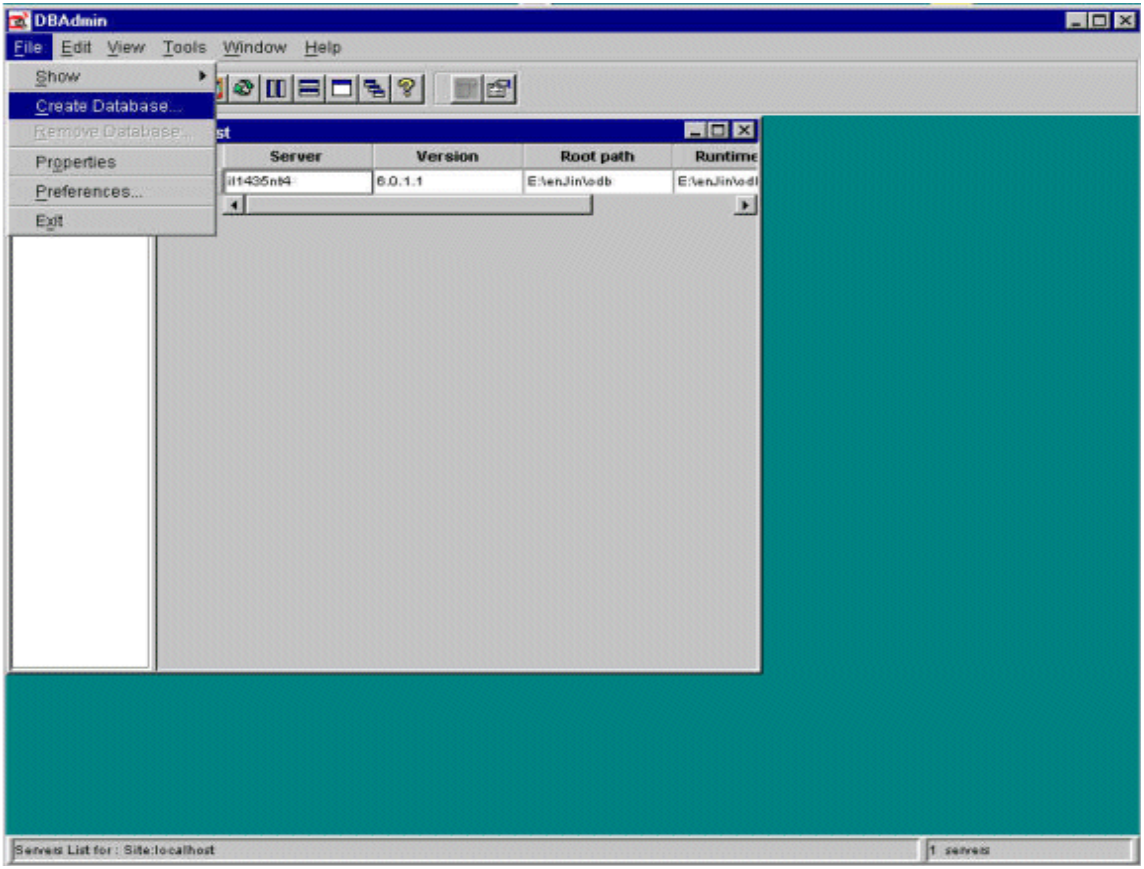


Figure 2-2 Create database

Please perform the steps in sequential order as demonstrated in Figure 2-3, and you will have a database where the objects that you create will be stored. Ensure that the server on which you to create your database has been selected.

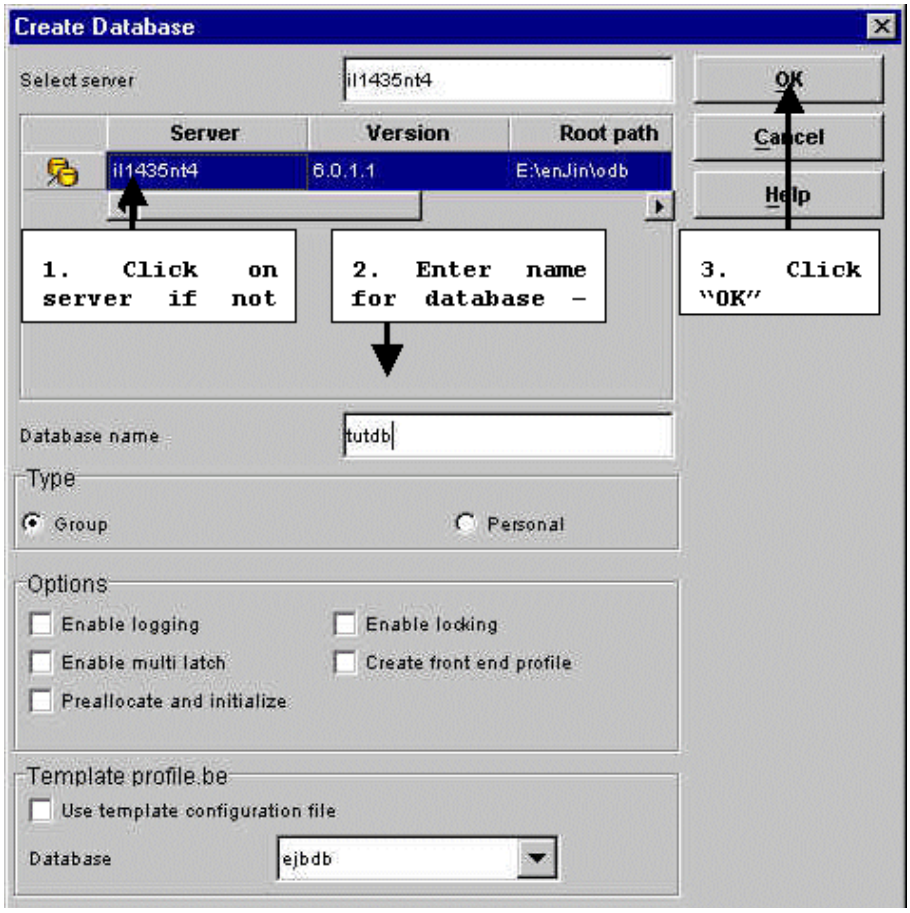


Figure 2-3 Create database steps

Now that we have a database to store persistent objects, you will experience the ease, and speed with which you can make objects persistent. To create a persistent Person object, we simply construct a new TransSession object and call the TransSession.makePersistent() method.

For example:

```
import com.Versant.trans.*;
public class CreatePerson
{
    public static void main (String[] args)
    {
        if (args.length != 3) {
            System.out.println
               ("Usage: java CreatePerson <database> <name> <age>");
```

```
            System.exit (1);
        }
        String database = args [0];
        String name     = args [1];
        int age      = Integer.parseInt (args[2]);
        TransSession session  = new TransSession (database);

        Person person = new Person (name, age);
        session.makePersistent(person);

        session.endSession();
    }
}
```

The example demonstrated above makes a persistent object using the following process:

1. Importing the JVI TransSession class. To import this class into the namespace of the application program, use the Java import directive. This class is located in the package com.Versant.trans.
2. Before any persistent objects can be created, you must connect to the Versant Object Manager by starting an Object Manager session. Starting an Object Manager session initiates enJin processes that allow you to access the enJin Object Manager. You can then use enJin Object Manager methods and persistent objects only within the session. In the Transparent binding, starting an Object Manager session is achieved by creating a new instance of the TransSession class. The constructor for this class takes the name of the Object Manager as a parameter.

   For more information on the available options when starting a session, see the reference to the TransSession class in the JVI User Manual. Starting a session also implicitly starts a transaction. All changes to persistent objects are written to the Object Manager only if the transaction successfully commits.
3. Since the Person class will be designated as a persistent class in the configuration file, no special code must be written to create persistent Person objects. Simply invoke the new operator as usual. The TransSession.makePersistent method causes the given object to be stored persistently in enJin's Versant Object Manager.
4. Ending the session causes the active transaction to commit. To commit a transaction without ending the session, use the TransSession.commit() method.
5. To compile the classes of an enJin application, simply invoke the Java compiler as usual. No special action needs to be taken. However, it is

necessary to correctly set the environment variable CLASSPATH, so that the enJin classes can be accessed by the compiler.

These classes are located in the enjin.jar file in the ENJIN_ROOT\lib directory. Add the enjin.jar file to your CLASSPATH environment variable. In addition, it is necessary to add the directory containing the Java source files to your CLASSPATH. This can be done by including the *current* directory, represented by a single period.

The exact method for setting CLASSPATH depends on the system being used. Please refer to your Java documentation for correctly setting CLASSPATH.

To compile the two Java classes: javac Person.java CreatePerson.java

Note: The process of enhancement is explained in more detail later in this chapter. However, for the purpose of this section, we will demonstrate enhancement of the .class Person and CreatePerson files we have just compiled on the next page.

A configuration file, usually named config.jvi, controls the behavior of the enhancer. This file specifies the persistence category of each of the classes that are being enhanced. The configuration file for the example application above would look as follows:

```
c Person
```

The letter "c" indicates that instances of the Person class are categorized as Persistent Capable. This means that Person objects can be stored in the Versant Object Manager, and the enhancer will modify the Person class so that persistence is possible.

The enhancer looks for .class files in a directory and deposits modified .class files after the process of enhancement. You have the option of storing your enhanced .class files in a separate directory (example: an *input* directory for your non-enhanced .class files, and an *output* directory for your enhanced .class files).

For this example, we have used input and output directories.

The input and output directories for the tutorial examples can be found in your JVI installation as the:

- enJin\examples\jvi\tutorial\trans\in
- enJin\examples\jvi\tutorial\trans\out

The enhancer is itself a Java application. Assuming the CLASSPATH is set as described above in the section on compilation and that you have a subdirectory named *out*, run the enhancer:

```
java com.Versant.Enhance -config config.vj -in in -out out -verbose
```

The -verbose flag causes the enhancer to indicate the persistence category of each class, and to display status information as the enhancement process proceeds.

If separate directories for enhanced classes are not required, enJin can simply perform the enhancement in place, and overwrites the non-enhanced classes and replaces them with the enhanced .class files. The method of invoking the enhancer can be simply executed with the following command:

```
java com.Versant.Enhance
```

Now that you have some exposure to the knowledge, of how enJin makes objects persistent, it is time to run the simple application we have developed. In subsequent parts of this chapter, you will continue to explore the transparency that enJin provides.

To run the application after having enhanced the .class files, simply invoke the Java interpreter as usual. However, it is very important to include the directory containing the enhanced .class files in the CLASSPATH. Failure to do so will result in run-time errors. Since your CLASSPATH contains the current directory (represented by "**.**"), this can be accomplished simply using the command:

```
cd tutorial/trans/out/
```

In addition, since the JVI libraries are implemented using native methods (functions written in the C language), the operating system must be able to locate the enJin's JVI dynamic-link libraries. These are located in the enJin\bin subdirectory of your enJin installation. On Unix machines, the enJin\lib directory is normally added to your LD_LIBRARY_PATH environment variable. On Windows machines, this directory is added to your PATH.

To execute the CreatePerson application:

```
java CreatePerson tutdb Bob 28
```

The above command adds a single Person object to the tutdb Object Manager, with name Bob and age 28. This can be seen using the db2tty utility or by using the DB Inspector by selecting **Start > Programs > enJin > DB Inspector**:

```
db2tty -d tutdb -i Person
```

Reading persistent objects

The transparent binding seamlessly inter-operates with the Versant Object Manager. Persistent objects are automatically fetched and locked as your application accesses them, by traversing object references or using the VERSANT query mechanism. All access to persistent objects occurs within an enJin transaction, so that changes to an object can be atomically committed or rolled back.

Both queries and navigation can be used to read objects. Queries are select statements similar to SQL statements used with relational databases. Navigation is following the references from one object to another object. In both cases, you can retrieve your destination object. However, they have different strengths and weaknesses. In general, enJin has the best flexibility and performance if you use queries mostly for finding starting objects and then use navigation (traverse the references) to retrieve relevant objects from the object graph.

This section will cover:

- Finding objects
- Object navigation

FInding objects

In this section you will learn how to locate your objects. This is a good way to find a starting point into your object graph. In the next section you will be able to use navigation to then traverse to other objects. This combination that enJin provides can dramatically improve performance and provides greater flexibility when building applications.

There are two basic ways of finding existing objects in the Object Manager:

- Finding objects with roots
- Finding objects with queries

Once a persistent object has been found by either of these two methods, its fields can be read and modified and its methods can be invoked. Any changes made will be written back to the Object Manager at the transaction commit.

Finding objects with roots

A *root* is a persistent object that has been given a name. This name can be used to find the object later. A root name is a bit like a file name, although the system of roots in enJin's JVI is much simpler than a true file system. In particular, there is one *space* for root names in each database.

Root names should be applied to only a relatively small number of objects in a database. Many database applications have complex, connected *graphs* of objects; a root provides a simple entry point into these graphs.

There are three fundamental root operations:

- Making a new root by giving an object a name
- Finding an object with a given root name
- Deleting a root name

First let's make a new root by giving a name to an object. The following example application, CreatePersonWithRoot, creates a new Person object and gives it a root name:

```
import com.Versant.trans.*;

public class CreatePersonWithRoot
{
 public static void main (String[] args)
  {
      if (args.length != 4) {
       System.out.println ("Usage: java CreatePersonWithRoot" +
              "<database> <name> <age> <root>");
            System.exit (1);
  }
        String database = args [0];
        String name     = args [1];
        int age      = Integer.parseInt (args[2]);
        String root     = args [3];
        TransSession session  = new TransSession (database);

        Person person = new Person (name, age);
        session.makeRoot (root, person);

        session.endSession ();
    }
}
```

This application is exactly like the previous CreatePerson program, except that it calls the TransSession.makeRoot method instead of makePersistent(). This allows us to give the object a root name, which is taken from the command line.

To run this application, go through the same steps as described in the previous part of the tutorial. Compile the .java source file:

```
javac CreatePersonWithRoot.java
```

Enhance the .class file and put output in subdirectory out:

```
java com.Versant.Enhance -config config.jvi -in in -out out -verbose
```

Run the application:

```
java CreatePersonWithRoot tutdb Mary 43 mary_root
```

This creates a persistent Person object identified by the root name mary_root. This root name can be used to subsequently retrieve the object, using the TransSession.findRoot method.

The findRoot() method returns an instance of Object, so a typecast must be used to recover the actual type, Person. If no root has the given root name, then findRoot() throws an exception.

Please proceed to the next page for the demonstration of findRoot() method.

The following program illustrates the use of findRoot():

```
import com.Versant.trans.*;

public class FindPersonWithRoot
{
    public static void main (String[] args)
    {
        if (args.length != 2) {
            System.out.println
                ("Usage: java FindPersonWithRoot <database> <root>");
            System.exit (1);
        }
        String database = args [0];
        String root = args [1];
        TransSession session  = new TransSession (database);

        Person person = (Person) session.findRoot (root);
        System.out.println ("Found " + person);

        session.endSession ();
    }
}
```

Now compile, enhance, and run the FindPersonWithRoot application.

Compile the .java source file:

```
javac FindPersonWithRoot.java
```

Enhance the .class file:

```
java com.Versant.Enhance -config config.vj -in in -out out -verbose
```

Run the application:

```
java FindPersonWithRoot tutdb mary_root
```

Running the application gives the output:

```
Found Person: Mary age: 43
```

Deleting a root is just as effortless -simply, use the TransSession.deleteRoot() method, to delete an associated root name .

The following program illustrates the use of deleteRoot():

```
import com.Versant.trans.*;

public class DeletePersonWithRoot
{
    public static void main (String[] args)
    {
        if (args.length != 2) {
            System.out.println
                ("Usage: java DeletePersonWithRoot <database> <root>");
            System.exit (1);
        }

        String database = args [0];
        String root     = args [1];

        TransSession session = new TransSession (database);

        session.deleteRoot (root);

        session.endSession ();
    }
}
```

Finding objects with queries

enJin provides a query language, VQL (Versant Query Language), to search for persistent objects that match certain criteria. VQL is a language that is very similar to the SQL that is used to query relational databases. enJin supports simple VQL queries. These queries can be used to find objects that have been stored using the Object Manager.

The purpose of this section is to introduce you to using VQL to access objects. For more information on using VQL with enJin, see the VQL section in the J/VERSANT Interface Users Manual

Transparent JVI expresses VQL queries by creating a VQLQuery object. The constructor accepts a query string argument.

Some values, such as arrays and strings, are difficult or impossible to express in VQL. To solve this problem, use substitution parameters. A substitution parameter is written as "$1", "$2", and so on, in the query string. You can then substitute the values with the bind() method on the VQLQuery object.

To find the objects that satisfy your query string, invoke the execute() method on the VQLQuery object.

Using a VQL query requires four steps:

1. Create a com.Versant.trans.VQLQuery object, passing the query string to the constructor. The basic syntax of a SELECT statement in VQL is as follows:

   ```
   "SELECT SELFOID FROM [class] WHERE [attribute] [operator] [value]"
   ```

2. Bind arguments to the substitution parameters ($1, $2, and so on) in the query string. If the query does not have substitution parameters, then skip this step.
3. Invoke the VQLQuery.execute() method. This method returns an object that implements the com.Versant.trans.VEnumeration interface.
4. Use the Enumeration.hasMoreElements() and Enumeration.nextElement() methods to obtain all of the objects that satisfy the query.

The example shown below demonstrates the finding the Person objects we created earlier to using a VQLQuery object:

```
import com.Versant.trans.*;
import java.util.*;

public class FindPersonWithVQL
{
    public static void main (String[] args)
    {
        if (args.length != 1) {
            System.out.println
                ("Usage: java FindPersonWithVQL <database>");
            System.exit (1);
        }
        String database = args [0];
        TransSession session  = new TransSession (database);

   VQLQuery query=new VQLQuery (session, "select selfoid from Person");
        Enumeration e = query.execute();

        if ( !e.hasMoreElements() ) {
            System.out.println ("No Person objects were found.");
        } else {
            while ( e.hasMoreElements() ) {
                Person person = (Person) e.nextElement ();
                System.out.println ("Found " + person);
            }
        }
        session.endSession ();
    }
```

```
}
```

The following program finds all Person objects located in a database.

Compile, enhance, and run this application using steps similar to those shown previously.

Compile the .java source file:

```
javac FindPersonWithVQL.java
```

Enhance the .class file:

```
java com.Versant.Enhance -config config.vj -in in -out out -verbose
```

Run the application:

```
java FindPersonWithVQL tutdb
```

Running the application gives the following output:

```
Found Person: Bob age: 28
Found Person: Mary age: 43
```

Object navigation

Navigation retrieves objects by traversing object references. Starting with an object obtained prior to navigation that contains a reference, the referenced object is retrieved based on its unique object ID that was assigned by the Object Manager.

Rather than retrieving all objects with queries or only by using navigation, a combination of the two, with a predominance of navigation, is the best combination for flexibility and performance.

References are extremely easy to use. You simply define your classes in exactly the same way that you would when writing a normal, application. The enhancer takes care of all of the work of converting references to links and fetching objects from the Object Manager when they are accessed.

To illustrate how links work in enJin, we will use simple Employee and Department classes. The Employee class contains a reference to the Department class: each employee belongs to one department. Similarly, the Department class contains a reference to the Employee class: one employee manages each department.

The source code for the employee is as follows:

```
public class Employee extends Person
{
    Department department;
```

```
    double salary;

    Employee (String aName, int anAge, double aSalary)
    {
        super (aName, anAge);
        salary = aSalary;
    }

    public String toString ()
    {
        return "Employee: " + name + " age: " + age +
               " salary: " + salary + " " + department;
    }
}
```

Next, we will define the Department class:

```
public class Department
{
    String name;
    Employee manager;
    Department (String aName, Employee aManager)
    {
        name    = aName;
        manager = aManager;
    }
        public String toString ()
    {
        return "Department: " + name + " manager: " +
            (manager == null ? "nobody" : manager.name();
    }
}
```

Now let us create an application that creates some employee objects, and stores them in the Object Manager:

```
import com.Versant.trans.*;

public class AddEmployees
{
    public static void main (String[] args)
    {
      if (args.length != 1) {
          System.out.println
                ("Usage: java AddEmployees <database>");
          System.exit (1);
     }
      String database = args [0];
```

```
      TransSession session  = new TransSession (database);

      Employee the_boss = new Employee("The Boss", 42, 110000);
      Employee jane_jones = new Employee("Jane Jones", 24, 80000);
      Employee john_doe   = new Employee("John Doe", 25, 75000);
      Employee lois_line  = new Employee("Lois Line", 36, 70000);

     Department engineering = new Department("Engineering", the_boss);
     Department marketing = new Department ("Marketing",   lois_line);

        the_boss.department = engineering;
        jane_jones.department = engineering;
        john_doe  .department = marketing;
        lois_line .department = marketing;

        session.makePersistent (the_boss);
        session.makePersistent (jane_jones);
        session.makePersistent (john_doe);
        session.makePersistent (lois_line);
        session.endSession ();
    }
}
```

The next steps are identical, such as compile and update the configuration file, and enhance and run the program.

Compile the .java source file:

```
javac Employee.java Department.java AddEmployees.java
```

Update the configuration file tutorial/trans/config.vj:

```
p  Employee
p  Department
```

Enhance the .class file:

```
java com.Versant.Enhance -config config.vj -in in -out out -verbose
```

Run the application:

java AddEmployees tutdb

Running the application will add four Employee objects to the Object Manager. To see that the four objects exist, you can use the db2tty utility:

```
db2tty-d tutdb -I Employee
```

You will now be able to witness traversing of objects using references by running the FindPersonWithVQL application that we created in the earlier section. This application will also find these employee objects:

```
Java FindPersonWithVQL tutdb
```

Running the application will display the following output:

```
Found Person: Bob age: 29
Found Employee: The Boss age: 42 salary: 11000.0 Department: Engineering
manager: The Boss
Found Employee: Jane Jones age: 24 salary: 80000.0 Department: Engineering
manager: The Boss
Found Employee: John Doe age: 25 salary: 75000.0 Department: Marketing manager:
Lois Line
Found Employee: Lois Line age: 36 salary: 70000.0 Department: Marketing
manager: Lois Line
```

Since the Employee class extends Person, these Employee objects are also Person objects. This is why the VQL query above matched the Employee objects as well.

Updating persistent objects

You will notice that the transparency of enJin makes changes to objects that have been created effortless. The example illustrated below shows how to modify persistent objects.

Like the previous program, it finds all person objects. This time, however, instead of simply displaying the contents of the object, the age of each Person object is increased by one, since we are all getting older every year.

```
import com.Versant.trans.*;
import java.util.*;

public class IncreaseAge
{
    public static void main (String[] args)
    {
        if (args.length != 1) {
            System.out.println
                ("Usage: java IncreaseAge <database>");
            System.exit (1);
        }
        String database = args [0];
        TransSession session  = new TransSession (database);

        VQLQuery query = new VQLQuery(session, "select selfoid from Person");
        Enumeration e = query.execute ();

        if ( !e.hasMoreElements() ) {
            System.out.println ("No Person objects were found.");
        } else {
            while ( e.hasMoreElements() ) {
```

```
                Person person = (Person) e.nextElement ();
                person.age++;
                System.out.println ("Increasing " + person.name+
                                    "'s age to " + person.age);
            }
        }

        session.endSession ();
    }
}
```

Now do the usual such as compile, update the configuration file, enhance, and run.

Compile the .java source file:

```
javac IncreaseAge.java
```

Update the configuration file tutorial/trans/config.vj:

```
a IncreaseAge
```

Note: You may have noticed in this instance we are categorizing the class as persistence aware. This is necessary since we have accessing public attributes. This, however, is not necessary, but if you do choose to engage in similar programming practices, you will be able to access these attributes by classifying your class as persistent aware. For more details on *Persistence Categorization* please refer to your JVI User's Manual.

Enhance the .class file:

```
java com.Versant.Enhance -config config.vj -in in -out out -verbose
```

Run the application:

```
java IncreaseAge tutdb
```

This shows the following output:

```
Increasing Bob's age to 29
Increasing Mary's age to 44
```

You can now use the FindPersonWithVQL program to in fact verify that the ages have indeed increased.

Deleting persistent objects

You will continue to exploit the Transparency of enJin to delete objects. This task is just as effortless as creating, reading or updating objects.

Persistent objects in the Versant Object Manager remain in the Object Manager until explicitly deleted. To delete objects from the Object Manager, use the TransSession.deleteObject method. The following program will delete an object with a given root name.

```
import com.Versant.trans.*;

public class DeletePersonWithRoot
{
    public static void main (String[] args)
    {
        if (args.length != 2) {
            System.out.println
                ("Usage: java DeletePersonWithRoot <database> <root>");
            System.exit (1);
        }

        String database = args [0];
        String root     = args [1];

        TransSession session = new TransSession (database);

        Person person = (Person) session.findRoot (root);
        session.deleteObject (person);
     session.deleteRoot (root);
        session.endSession ();
    }
}
```

Java does not support a delete operation for dynamically allocated objects. Instead, Java relies on garbage collection to rid memory of unreferenced objects. However, enJin does not use garbage collection since all objects are reachable via a query. The deleteObject method deletes the persistent object from the Object Manager only, not from memory. To permanently delete an object, each object must be explicitly deleted this way. After deletion, the object in memory should not be accessed. The object will be deleted from the Object Manager at transaction commit.

Deleting an object is not the same as deleting a root: deleting a root simply removes the root name associated with the object and does not delete the object from its database.

Now compile, enhance, and run this program as we have been doing earlier.

Compile the .java source file:

```
javac DeletePersonWithRoot.java
```

Enhance the .class file:

```
java com.Versant.Enhance -config config.vj -in in -out out -verbose
```

Run the application:

```
java DeletePersonWithRoot tutdb mary_root
```

Running the same program a second time generates an exception because the object was deleted and the root name removed.

2.2 Class enhancement

Enhancement is where the *magic* of the JVI transparency takes place. The enhancer is a tool that *post-processes* Java class files by modifying the classes after they are compiled so that your Java objects can be managed in the Object Manager.

To get a true understanding of the process of enhancement, we will discuss the following topics:

- Overview - This will describe what happens when you enhance.
- Running the Enhancer

2.2.1 Overview

The enhancer is termed a *postprocessor,* because it manipulates Java.class files, not .java source files. The enhancer accepts Java class files as input and produces modified class files as output. enJin's JVI uses the enhancer to achieve transparent object management. The process of enhancement is represented in Figure 2-4.

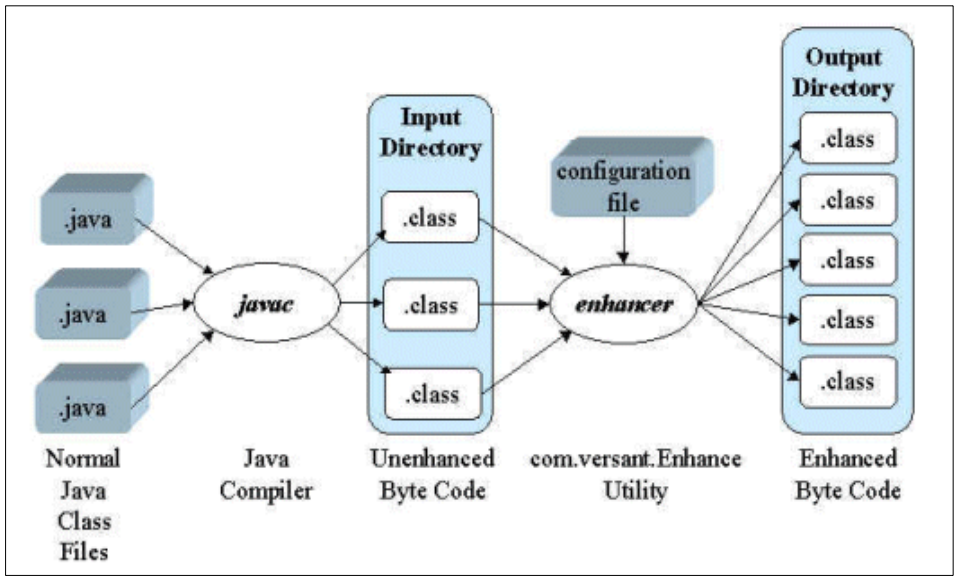


Figure 2-4 Enhancement process

The enhancer makes several changes to the class files, depending on the specifications in a configuration file. (For an explanation of this configuration file, see 2.3, “Persistence categorization” on page 41.) These class file changes include:

- Code is generated to automatically define the schema for the class.
- Code is generated to read and write the persistent objects to and from the Object Manger.
- Code that accesses (gets and puts) fields of an object is modified to first perform additional checks.
- When a field of an object is read or modified, the object is fetched from the Object Manager lazily.
- When a field of an object is modified, the object is marked *dirty*. Changes will then be written to the Object Manager when the transaction is committed.

The objects are *re-rooted* to extend a common base class. This base class is an internal detail of the implementation of Transparent JVI, and should not have a direct affect.

2.2.2 Running the enhancer

The enhancer can be invoked as a separate standalone utility in the following way.

You can explicitly enhance your application classes using the standalone enhancer utility. This utility is the Java application com.Versant.Enhance. When this application is executed, it does the following:

- Reads the configuration file specified on the command line
- Reads each class file (or collection of class files within a directory) specified on the command line
- Enhances each class file, using the information given in the configuration file

The enhancer can place modified class files in a separate directory, or it can enhance class files in-place. The optional -out flag is used to tell the enhancer where to place the modified class files. For example, when using the -out option, consider an application whose classes are contained in the mycorp.myapp package. The directory structure might look as follows:

```
myrootdir/myconfigfile
myrootdir/in/mycorp/myapp/Main.java
myrootdir/in/mycorp/myapp/Main.class
myrootdir/in/mycorp/myapp/MyObject.java
myrootdir/in/mycorp/myapp/MyObject.class
myrootdir/out/
```

Invoke the enhancer utility as follows:

```
Cdmyrootdir/
java com.Versant.Enhance -in in -out out -config myconfigfile
```

These arguments instruct the Enhance application to look for classes in the in directory, modify them according to the settings in myconfigfile, and deposit them in the out directory. This will cause the following directories and files to be created:

```
myrootdir/out/mycorp/myapp/Main.class
myrootdir/out/mycorp/myapp/MyObject.class
myrootdir/out/mycorp/myapp/MyObject_Pickler_Vj.class
```

Note the newly created "MyObject_Pickler_Vj" class. Creation of this class depends on the configuration file (in fact, the pickler class will only be created if the class is categorized as "c" or "p"). See the section "Persistence Categorization" for more information.

Alternatively, class files can be modified *in-place*. This means that the enhanced class files replace the original class files. To use in-place enhancement, omit the -out flag. When using in-place enhancement, separate *in* and *out* directories are not necessary. For instance, the directory structure for the mycorp.myapp would simply be as follows:

```
myrootdir/myconfigfile
myrootdir/mycorp/myapp/Main.java
```

```
myrootdir/mycorp/myapp/Main.class
myrootdir/mycorp/myapp/MyObject.java
myrootdir/mycorp/myapp/MyObject.class
```

Then the enhancer can be invoked in the following manner:

```
cd myrootdir/
java com.Versant.Enhance -config myconfigfile .
```

Note the "." at the end of the command line. This tells the enhancer to examine all classes in the current directory. It is not possible to enhance a class more than once. The enhancer places a special marker inside the class file, so that it can determine whether a class has already been enhanced. If a class file has already been enhanced, then it will not be modified.

Important: A persistent class is changed in such a way that other classes that refer to the fields of the persistent class will no longer compile correctly when using in-place enhancement. Hence, it is necessary to compile all of these classes simultaneously to ensure correct compilation.

As an example, consider the following classes:

```
class PersistentCapable {
int x;
}
class NotPersistent {
int foo (PersistentCapable pc) {
return pc.x;
}
}
```

The following would not work correctly:

```
javac PersistentCapable.java
java com.Versant.Enhance PersistentCapable.class
javac NotPersistent.java
```

Here, the PersistentCapable class has been enhanced in-place. Now, when NotPersistent is compiled, a compilation error will result because the field "x" has been changed to private. (The access control is changed to prevent unenhanced persistent aware classes from accidentally accessing the fields of a persistent object at run-time. By making the field private, this situation can result in an IllegalAccessError instead of silently giving invalid results.)

Instead, you should do this:

```
javac PersistentCapable.java NotPersistent.java
java com.Versant.Enhance PersistentCapable.class \
NotPersistent.class
```

For more information on the enhancer, including command-line arguments and the syntax of the configuration file, see the description of com.Versant.Enhance in the JVI User Manual.

2.3 Persistence categorization

Each class in enJin's has a persistence category. A persistence category describes the Object Manager-related capabilities of the class. For example, can objects be stored in the Object Manager? If so, when do the objects become persistent? A persistence category also determines the modifications the enhancer must make to the class file for a class.

A configuration file is used to specify the persistence category for each class. Each line of the configuration file contains a category specifier and a class name pattern. When the enhancer encounters a class, it looks for the first matching pattern in the configuration file and gives the class the category specified for that pattern.

In this section we will discuss:

- Persistence categories
- How to choose a persistence category

2.3.1 Persistence categories

Transparent JVI has five persistence categories. Table 2-1 demonstrates two of these. For the purposes of your applications, these two categories will suffice. For a more detailed explanation of this concept, and to learn about the other three categories, please refer to your JVI User Manual.

Table 2-1 Persistence categories

Persistence Category	Description
"c", Persistent Capable	Objects of classes that are Persistent Capable can become persistent either explicitly or implicitly. Persistent Capable objects that have not yet become persistent are called transient.

Persistence Category	Description
"n" Not Persistent Aware	The enhancer does not modify classes that are Not Persistent Aware. The methods of Not Persistent Aware classes cannot directly access the fields of an FCO. The methods of Not Persistent Aware classes can call methods on an FCO.

2.3.2 How to choose a persistence category

Choosing the correct persistence category for each of the classes in your enJin application is an important part of the development process. To properly categorize your classes, it is helpful to have an understanding of how the classes will function within your application, as well as what changes the enhancer will make to these classes.

Persistent capable

You should assign a class the Persistent Capable category if you want instances of that class to be stored in the Object Manager as independent persistent objects. A Persistent Capable object can be either transient or persistent. A transient object behaves essentially as it would have before the enhancer modifies the class. A persistent object, however, is considered a First Class Object. It takes advantage of the changes made by the enhancer to enable the object to be stored in the Object Manager. For example:

- The Versant Object Manager assigns each persistent object a unique object identity, its LOID.
- The enhancer generates additional methods that automatically read and write the object when the attributes of the object are read or changed.

Not persistent aware

The enhancer does not modify a Not Persistent Aware class. Classes that do not directly manipulate persistent objects can be categorized as Not Persistent Aware. For example, all standard Java classes are categorized as Not Persistent Aware by default. In general, third-party classes that do not access the fields of persistent objects can be given the "n" category.

2.4 Versant enterprise container

Versant Enterprise Container (VEC) is the integration of enJin's JVI with your Application Server. It provides transaction management, concurrency control, and persistence for entity beans in the Versant Object Manager, or allows you to use enJin's transparent object management capabilities directly from your SessionBeans.

You will be exposed to several of the functions of VEC when you begin to build your EJB Applications in the subsequent chapters.

Specific topics covered in this section include:

- Session manager
- Transaction management
- Helper classes

2.4.1 Session manager

enJin's VEC Session Manager maintains and manages one or more Object Manager *session pools*. A session pool is a collection of Object Manager sessions of a particular session type. An enJin session is similar in concept to a JDBC connection with the key differences that enJin sessions cache objects and they can be connected to several Versant Object Managers.

In the default configuration the Session Manager simplifies the job of managing instances of the TransSession class - instead of explicitly creating a TransSession, you request one from the pool.

The Session Manager provides:

- Session Pooling

- Default shared and exclusive session types (also allows for custom session types)
- Load balancing within a session pool
- Automatic cache management for the Versant Object Manager sessions

A session pool can consist of any combination of the following types of Object Manager sessions as shown in Table 2-2.

Table 2-2 Session pool

Session	Description
Read only/shared	Used for read only operation. These sessions are not exclusively bound to a transaction. These are typically used to process non-transaction, read-only methods.
Read-write/exclusive	Used for Read and Write operations. Only one transaction can use this type of session for the duration of a transaction. As sessions are shared resources, once the transaction completes any client for a subsequent transaction can use the session. This type of session would typically be used for transactional methods. This is the default session type.
Custom	You can create a class that implements any kind of session supported by enJin JVI.

2.4.2 Transaction management

Transaction Management is yet another feature that enJin offers that makes it easier and faster for developers to build applications.

As mentioned earlier in the document, it is beyond the scope of this document to expose all of the functions of enJin. The purpose of the section below is to expose the user to some of the features of enJin's VEC, such as:

- enJin provides automatic transaction management based on the transaction property specified in the deployment descriptor.
- Association of every Entity Bean with an object stored in the Object Manager when the bean is activated. When a transaction completes (commits or aborts), the Object Manager object is released from the enJin session.

- A shared-session is typically used to handle non-transactional requests or to handle methods that only read data (are not mutators). As commits are disabled for shared-sessions, in order to manage memory over a span of multiple transactions, enJin automatically releases objects from the shared-session based on a user configurable parameter.
- There is no requirement that the user create a shared-session pool. Even non-transactional methods (such as finder methods) can be executed in exclusive sessions. For non-transactional user methods, it is up to the user to ensure that the method does not mutate the state of an entity bean or a persistent object.
- enJin generates a call to get a session from the session pool in the EntityBean Implementation class before any method on the bean is invoked. If the bean method is transactional, the get-session will ensure that the same exclusive session is returned to the user for all calls made in the same transaction. After the method has been executed, the session is released.
- As an optimization, enJin attempts to get the same session from the session pool across multiple transactions. The same is true when invoking a non-transactional method after a transaction method.

2.4.3 Helper classes

The purpose of this section is to introduce you to the concept of Helper classes. enJin has been designed to augment your abilities to develop enterprise applications. To augment your application performance and to accelerate the development process, enJin's VEC provides Helper classes.

The Helper classes do a lot of the work in the background for you. All you actually have to do is make the calls to these Helper classes. By using these Helper classes provided in enJin, you can leverage your current knowledge to start the process of development immediately. You need not be an expert at using either Versant enJin's Object Manager or a seasoned JVI user.

To explain the intricacies of these classes is beyond the scope of this document. Please review the accompanying documentation with your enJin installation.

enJin provides two Helper classes:

- ClassEntityBeanHelper
- ClassSessionBeanHelper

ClassEntityBeanHelper

This class is the helper class to manage the session context. BMP beans can use this class to implement EJB methods to:

- Allow the user to quickly develop bean-managed entity beans and session beans that access persistent objects managed by enJin.
- Makes it easy for EJB developers to store and query BMP entity beans by:
  - Managing the session context
  - Implement EJB methods (ejbLoad, ejbStore, ejbRemove and ejbCreate)

ClassSessionBeanHelper

Helper class that may be used by SessionBeans to handle the persistence of objects The three scenarios for using this class could be:

- When the SessionBean methods are being executed using container managed transaction.
- When the SessionBean method being executed is non-transactional
- When the SessionBean method being executed is using a user transaction

Now that you know about yet another of enJin's unique features, you are ready to start building an Enterprise Application the enJin way.

Part 2

Developing an enJin application

In this part you will learn how to implement a methodology for developing and deploying a typical J2EE application using IBM WebSphere and enJin. This will be done building and deploying a fully developed typical EJB application using Session Managed Persistence. This will be done using WebSphere Studio Application Developer. A downloadable JAR file will allow you to run and explore the example rapidly. In presenting the example that we use, you will learn the same methodology that resulted in the performance that we accomplished in the benchmarking project, and you will be able to utilize the same principles for your own example.

3

Developing applications with Versant enJin

Versant works with leading vendors to simplify the development and speed performance of applications using enJin. Versant has integrated tools for enJin with WebSphere Studio Application Developer. The integration of tools makes it easy and fast to use the enJin application accelerator in J2EE applications built using WebSphere Studio Application Developer (WSAD). enJin provides transparent persistence for Java objects in JSPs, servlets, sessions (SMP) and for BMP Enterprise Java Beans (EJBs). enJin lets you minimize tedious, error-prone, and rigid mapping of Java objects. Use enJin's transparent persistence instead of JDBC code. Get faster development, faster performance at run time, and greatly increased ability to evolve your application as requirements change by using enJin.

Note: Please download the enJin WSAD integration from the URL identified below. This guide assumes that WebSphere Studio Application Developer as well as Versant enJin 2.2.3 for IBM WebSphere 4.0 for Windows NT is installed on your machine. To download a trial version of Versant enJin, please visit:

`http://www.versant.com/products/download/index.cfm`

Please note, if you are using WSAD to develop and test your J2EE Applications, you do not need to have WebSphere Application Server installed on your machine.

This chapter will introduce users to the integration of the tools with WSAD. Specific topics in this chapter include:

- Installation and configuration of Versant Integrated Tools
- Introduction to the enJin tools an User Interface
- Invoking the tools

3.1 Installation, configuration of Versant tools

Before you can use the enJin tools integrated with WSAD, the user must make sure that WSAD is configured to use enJin's IDE Toolkit. This is done as follows.

Unzip the downloaded zip file. Its contents include:

- wsadscript.bat
- enJinTools.jar
- infoset.xml, infosetAction.xml, infosetConcepts.xml, infosetTopics.xml, plugin.xml
- Icons directory
- Examples directory
- doc.zip
- ReadMe.html

Please run wsadscript.bat as follows, where <WSAD INSTALL_DIR> is the location of your WSAD Install directory.

wsadscript <WSAD INSTALL_DIR>

This will create a directory (com.versant.enJinTools_1_0_0) in the <WSAD INSTALL_DIR>\plugins directory and will place all the aforementioned files in this directory.

The next time you launch WSAD, you will have the integrated enJin tools accessible from the IDE.

Subsequent to the above step, when brought up with the uenJin IDE Toolkit, you will notice, a new menu item (enJin Database Tools), two new buttons with enJin icons, an enJin console. With this integration that we have provided is also included a Template Generator that will enable developers to rapidly *enJinify* Applications utilizing Servlets and Bean Managed Persistence.

Utilization of these tools as well as developing your J2EE applications that leverage the power of enJin are described in subsequent sections.

3.2 Introduction to the enJin tools and user interface

After running the script that we have provided, the next time you launch the WSAD IDE, direct your attention to the toolbar. You will notice a new menu item enJin Database Tools, which are also represented as buttons on the toolbar as displayed in Figure 3-3.

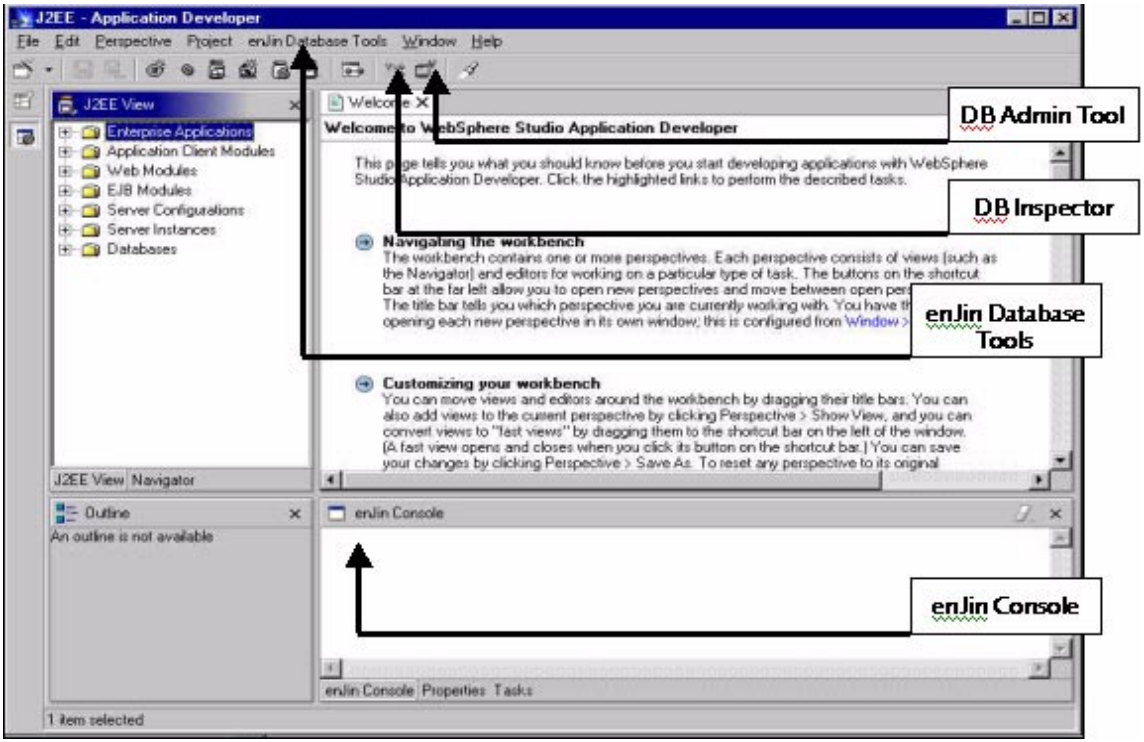


Figure 3-1 enJin interface

- The DBAdministrator can be invoked either by using the menu item **enJin Database Tools -> DBAdministrator**, or by simply clicking the button. You may use it to create, delete, and perform other operations on the databases.
- The DBInspector can be invoked either by using the menu item **enJin Database Tools -> DBInspector,** or by simply clicking the button. You will use view the database contents.

In addition, there are three other powerful tools that you will use while developing with enJin. These are:

- Persistence Tool - Used to mark the persistence category of classes. Upon categorization of your classes, another tool automatically gets invoked, and result is that your classes are now *enhanced*.
- enJin Wizards - You may use this to generate templates for applications using servlets or Bean Managed Persistence.

4

Session managed persistence

This approach is a powerful design approach. It gives the developer the full flexibility while using enJin's transparency, while allowing the IBM WebSphere to manage resources thereby leveraging enJin features such as session pooling and load balancing.

You can use enJin's transparent Java language interface (JVI) to store and access persistent objects from Session Beans. The persistent data in your application can be directly stored in the enJin database using the object model that you create. Once the java classes that represent persistent objects have been created, you can develop the EJB interfaces using session beans.

The helper classes give you an easy way to tie in with the application server session pooling and transaction management functionality, and allow you to work with objects just as you would in any Java program and use of enJin's APIs for transparent object persistence.

In this chapter, we will demonstrate the process of developing an EJB Application using Session Managed Persistence. You will notice that for most parts, if you are familiar with the J2EE standards, *enJinizing* your applications is not much different. You will however, be able to develop your applications faster, and will notice an improvement in the performance.

Since we used SMP in the benchmarking project to measure the performance, we have chosen to focus demonstration of this method of persistence. You may also use enJin and leverage its powerful features to implement Bean Managed Persistence, Java server pages, and servlets.

On the CD accompanying this Redbook, we have included a jar file which contains the source Java classes that demonstrate SMP.

Topics covered in this chapter include:

- The outline of the example
- Running the example

4.1 Outline of the example

This is a HRApplication, which allows a human resource manager to enter and retrieve employee and department information. When a request is submitted, it calls ManagerServlet, which invokes the appropriate bean method. This application provides the following functionalities :

- CREATE New Department - Creates a new department
- CREATE New Employee - Creates a new employee. The department of the employee needs to be specified. If the specified department does not exist, then a new department is also created.
- REMOVE Department - Removes the specified department
- FIND Employees - Finds all employees of the specified department
- FIND All Employees - Finds all the employees in all departments

This example demonstrates a SessionBean directly accessing the persistent objects. It illustrates a SessionBean using SessionBean helper classes and Java Versant Interface (JVI) to access the persistent objects. This SessionBean accesses the persistent objects directly and provides the required interfaces by the client. Since the SessionBean accesses the persistent entities directly, it provides a much faster execution when compared to accessing entity beans, which has high overheads of locating and accessing beans.

4.2 Running the example

Upon completion of this section, you will successfully build, deploy, and test a typical EJB (Session Managed Persistence) Application. This means, you will utilize and witness the features of enJin, as well as learn about specific enJin implementation.

The six steps are as follows:

1. Import the jar file.
2. Implementation of Session Managed Persistence.
3. Categorize the persistent classes.
4. Create the database.
5. Deploy and test the application.
6. View the results in the database.

4.2.1 Import the jar file

Launch your WSAD IDE. Rather than having to write every line of code, the JAR file that we have provided you with has all of the necessary Java files that you will need for running the example.

From the WSAD IDE, select **File -> Import** as shown in Figure 4-1.

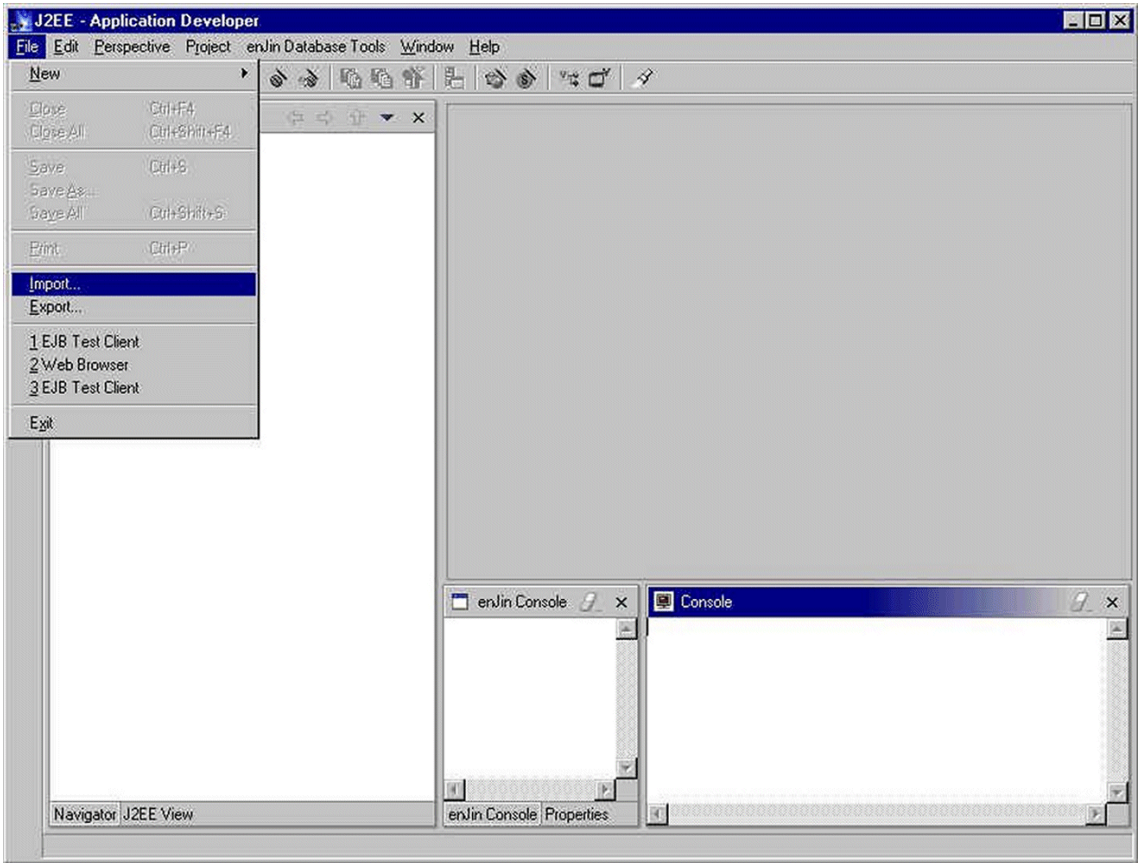


Figure 4-1 WSAD IDE

This will bring up the Import dialog box as shown in Figure 4-2. There are several steps to successfully importing the example. The first step involves, selecting the resource that you would like to import. From this dialog box, select **EJB JAR** as shown in Figure 4-2, and select **Next**.

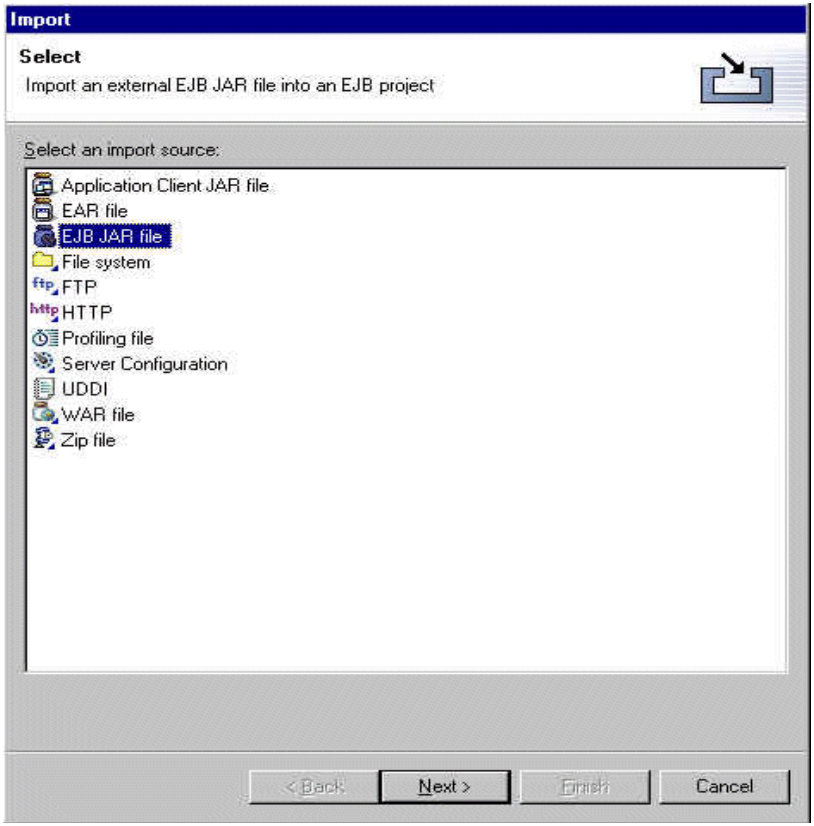


Figure 4-2 Import select

Upon selecting Next, a new dialog box (as shown in Figure 4-3) will appear representing the second step of the import process. Here you will be asked to provide the location of the EJB JAR file which can be found at:

`http://www.versant.com/developer/redbook`

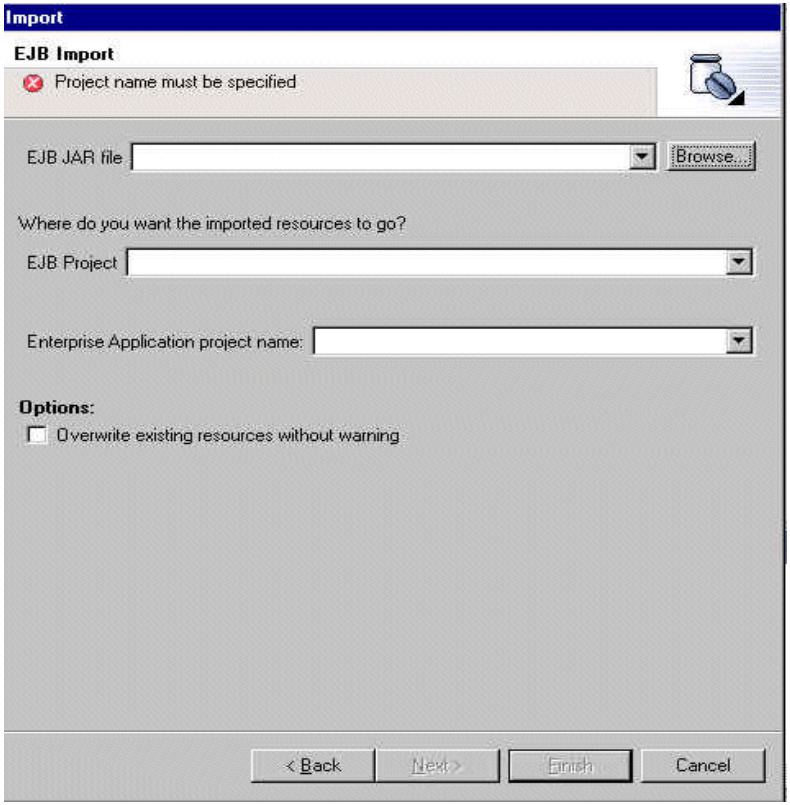


Figure 4-3 EJB import

Select the **Browse** button and locate enjinSMP.jar as shown in Figure 4-4.

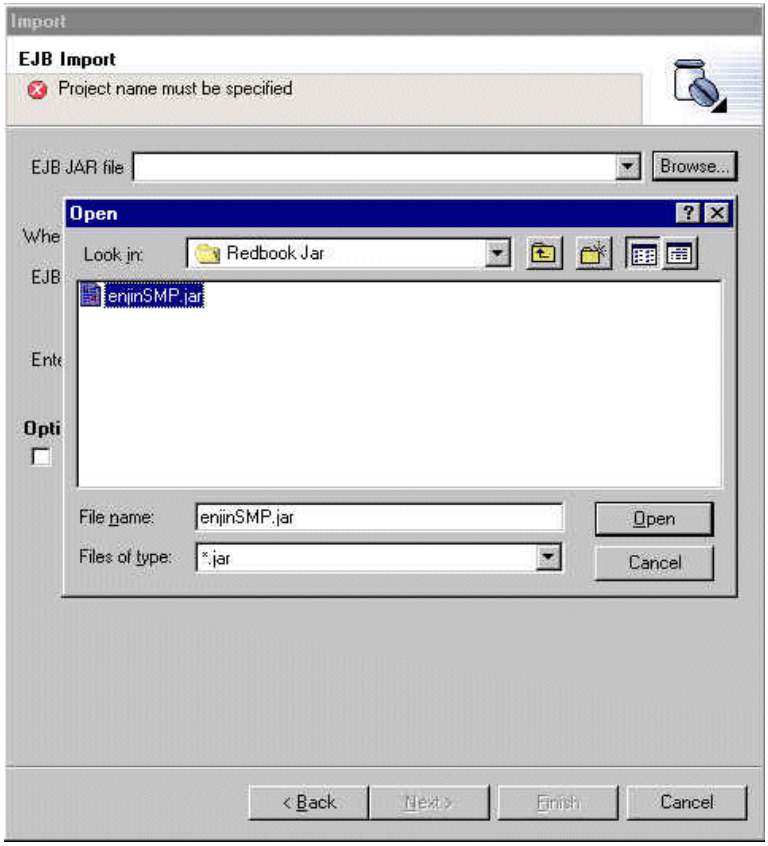


Figure 4-4 EJB import open

Provide a name (we will use enjinSessionManagedEJB) in the text box titled EJB Project and Enterprise Application project name (we will use enjinSessionManagedEAR.). The dialog box when completed will appear as shown in Figure 4-5.

.

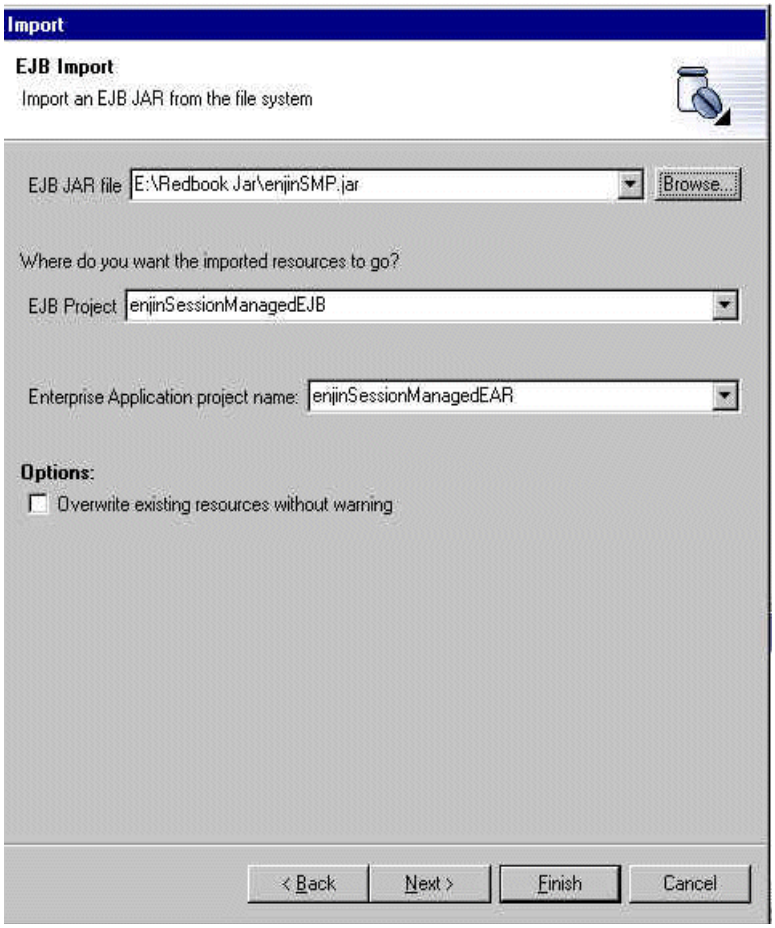


Figure 4-5 EJB import

Select **Next** after completing the fields as shown above.

Figure 4-6 shows the dialog box that appears allows the user to select specific files that they would like to select. Since all of the Java files within enjinSMP.jar are required, simply select the radio button titled **All Files** and then select **Next**.

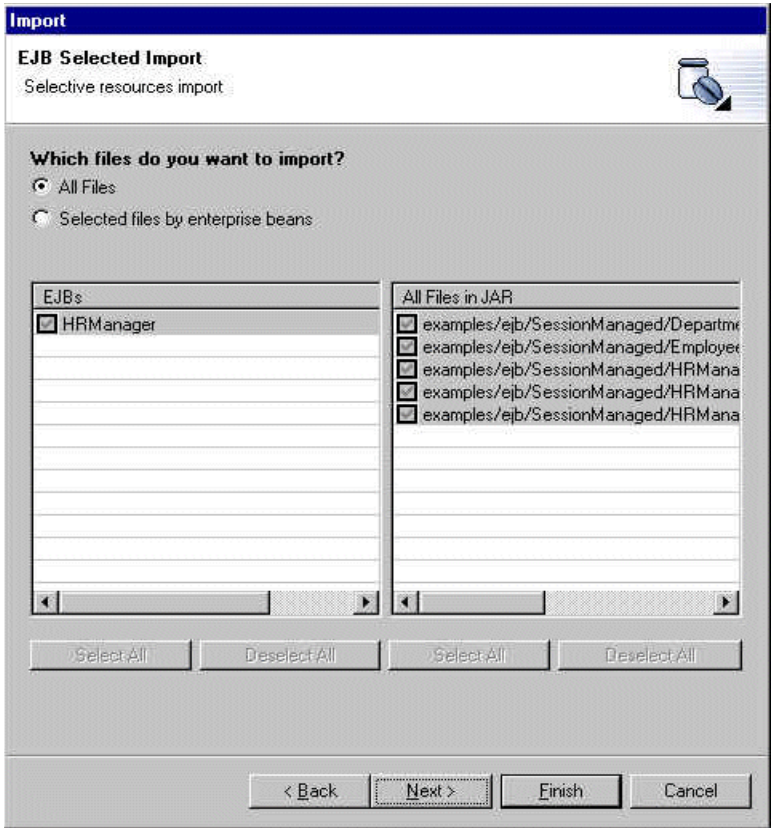


Figure 4-6 Import selection

The next step allows the user to establish if there are any module dependencies. Since there are none for the project that we have been building thus far, simply select **Next**.

The final dialog box will now appear. This allows the user to specify the source and output folders, as well as the Java classpath in the appropriate order.

Ensure you are in the Libraries tab as shown in Figure 4-7. Select the **Add External JARs** button, and locate enjin.jar, which can be found at <ENJIN_ROOT>\lib\enjin.jar, and then select **Finish**.

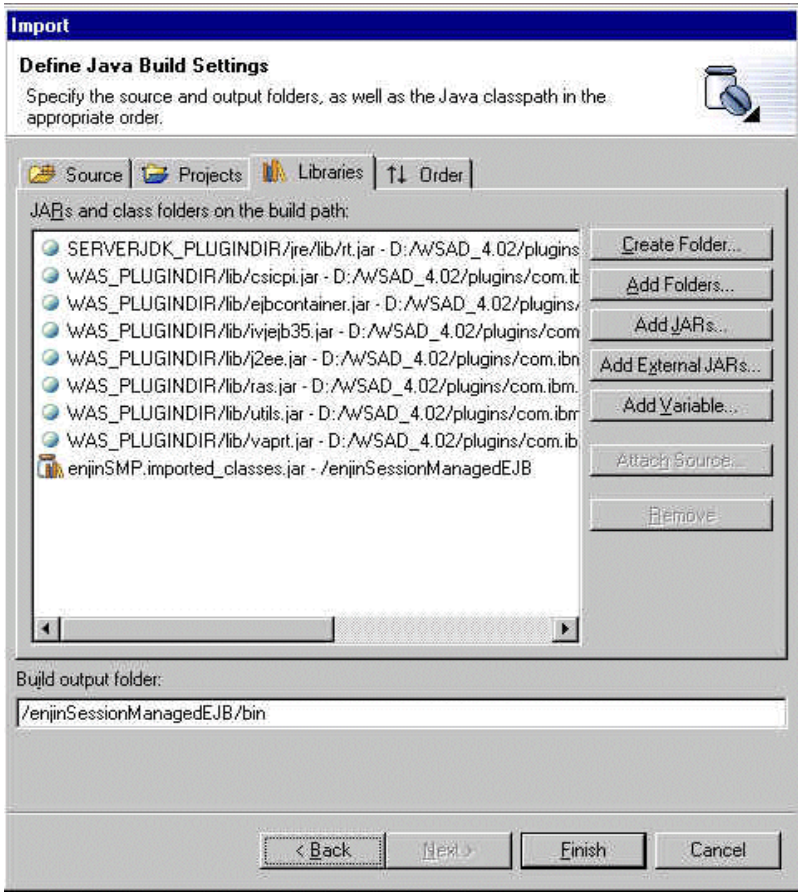


Figure 4-7 Java build settings

At this point you will have a fully developed enterprise application with enJin Session Bean Implementation.

Prior to proceeding to the next step in running this application, let us examine what is involved in using this powerful approach.

4.2.2 Implementation of enJin session managed persistence

The process for development using this approach is no different from traditional SessionBean development. When you examine the code in any of the classes, you may notice that there is nothing different from what would normally appear in these classes. The only difference is in the HRManagerBean.java code. The steps required for implementation of Session Bean Persistence the enJin way involves the following.

Configure the datasource and poolname

The VEC configuration file (vec-config.xml) is an XML file that is used to specify the session pool properties and other properties such as logging level. This configuration file is used by the VEC Startup class to initialize Versant session pools during WebSphere startup.

A Versant session is similar in concept to a JDBC connection, with the key differences being that Versant sessions cache objects and a Versant session can be connected to several Versant databases.

The following are the parameters for initializing VEC Session Pools:

- Datasource name
- Session pool name
- Initial number of sessions
- Maximum number of sessions

A sample vec-config.xml file is displayed below:

```
\  <!DOCTYPE vec-config (View Source for full doctype...)>
<vec-config>
   <log-level>debug</log-level>
<session-pool>
       <pool-name>ejbPool</pool-name>
       <data-source>ejbdb</data-source>
  </session-pool>
</vec-config>
```

For our example we have used ejbdb and ejbPool. You may choose to specify the initial number of sessions as well as the maximum number of sessions in the pool.

Declare a static variable of type SessionBeanHelper

Versant enJin provides a helper class com.versant.ejb.generic.SessionBeanHelper that makes it easy for non-expert users of enJin API's to take advantage of enJin's object management capabilities and handle the persistence of objects.

In your Bean class declare a static variable (say helper) of type SessionBeanHelper as follows:

```
static SessionBeanHelper helper = null;
```

Initialize the SessionBeanHelper

Instantiate the class in your setSessionContext method as follows:

```
public void setSessionContext(SessionContext ctx)
    throws RemoteException
```

```
    {
        say("setSessionContext called");
        _ctx = ctx;
        helper =  SessionBeanHelper.GetInstance(this, POOLNAME);
        say("setSessionContext :" + " got SessionHelper.");
  }
```

The instantiation of this class makes the coding of the business methods that need to be executed within Versant transactions effortless.

Write the code for your business logic

For *transactional* business methods that need to access enJin objects, add this call at the beginning of each business method, in every method of the session bean, you can get hold of a session bean helper using the call:

```
Transession session= helper.getSession(this);
```

The session is accessed from the pool that was initialized at the startup of the application server. This session bean helper maintains a reference to the enJin session that has been assigned to the current transaction.

This method below creates instances of the *Department* object.

```
public void createDepartment(String depName)
        throws Exception
    {
   say ("Session Bean->createDepartment called");

        try {
            TransSession session = helper.getSession (this);
            Department currentDepartment = new Department(depName);
            session.makePersistent (currentDepartment);
        } catch (Exception e) {
            say("Exception in creation of Department!!!");
            throw e;
        }
```

For *non-transactional* methods, vejbPreMethod must be invoked, which basically does the same job as getSession, but returns com.versant.ejb.common.VEJBSessionContext. To get hold of an enJin session (TransSession), for doing database operations, getTransSession method can be invoked on VEJBSessionContext. You can get the session referenced by the session helper by:

```
VEJBSessionContext sc= helper.vejbPreMethod(this);
TransSession session = sc.getTransSession();
```

Every vejbPreMethod should be followed by a vejbPostMethod in order to release the session back to the pool.

```
helper.vejbPostMethod( sc,this);
```

The vejbPostMethod should be invoked in the finally block of the bean's methods to ensure that the resources used are released even if an exception is encountered in the processing of the method.

4.2.3 Categorize the persistent class

Once the class files are created, the Persistence Tool can be used to mark the persistence category of each class. The basic persistence categories are: Not Persistent and Persistence Capable. There are other, more advanced persistence categories that may be used under certain circumstances and include, Persistence Aware, Persistent Always, or Transparent Dirty Owned. For details on these categories, and appropriate use of each of them, please refer to the Java Versant Interface Usage Manual, which may be accessed directly from your enJin installation <ENJIN_ROOT>\doc\jvi\usage\html\frame.html or from our Web site, at:

http://www.versant.com /developer/_docs/vds/jvi/doc/index.html

In this section, we will merely demonstrate the utilization of the enJin Persistent tool.

The enJin Persistence tool can be invoked by selecting the project we have created in the navigator view and by right clicking once the project has been selected as shown in Figure 4-8.

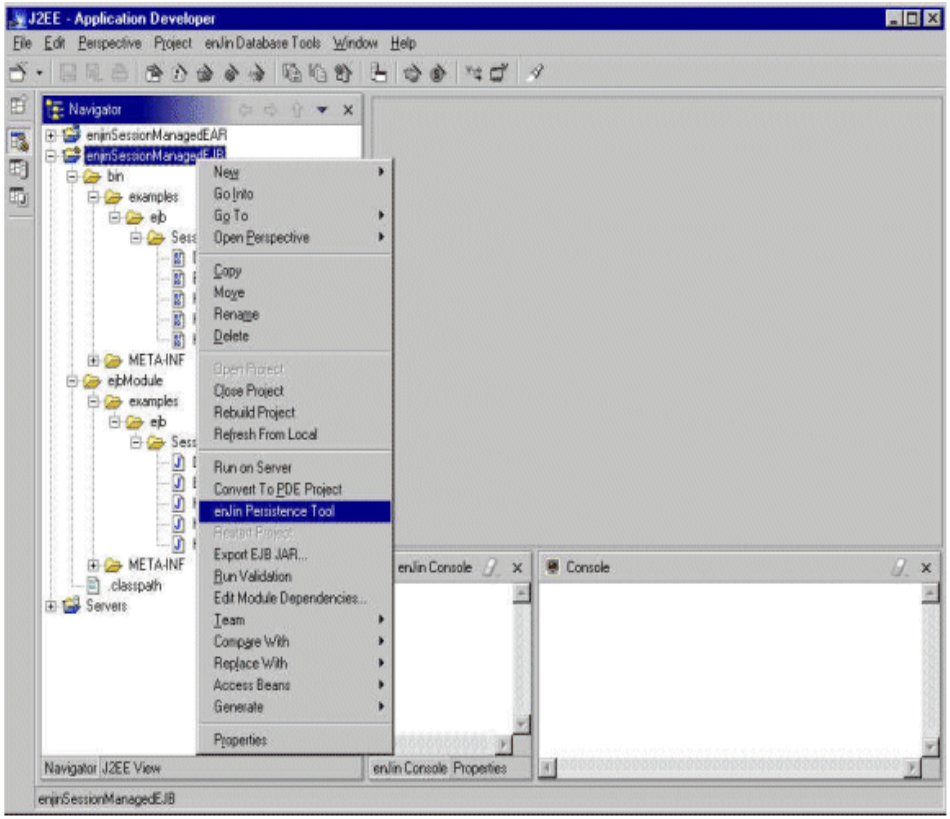


Figure 4-8 enJin Persistence Tool

You will now see the basic enJin Persistence Tool dialog box, which allows you to categorize your classes in the basic categories as shown below.

The basic screen presents two options: Not Persistent or Persistent. The basic screen's Persistent option maps to the Persistent Capable option in the Advanced screen (as well as in Versant's categorization system).

This categorization will suffice for the project we have been building thus far. Simply select the Department and Employee class and move them to the Persistent section by using the arrows, as shown in Figure 4-9.

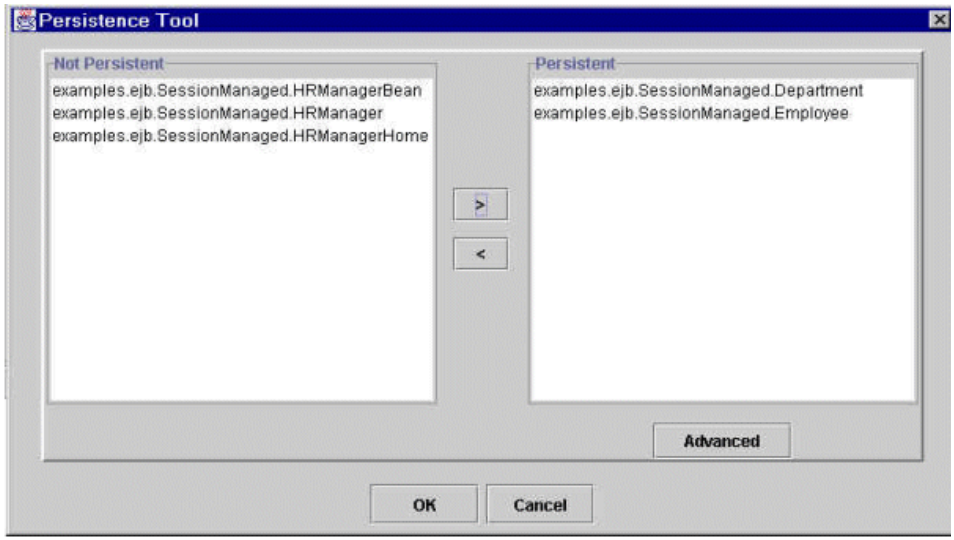


Figure 4-9 Persistence Tool

The other classes, are automatically categorized as Not Persistent. Click **OK** after the categorization.

Upon identifying the categories of your classes and clicking OK, the Persistence Tool generates a config.jvi file, which is then used as the input in the next step. For the purpose of this example, save this file in the same default directory. You will now notice that the warning regarding the enhancer not being executed has disappeared. This is because, upon categorization, the config.jvi file gets generated, and the enhancer utility gets automatically executed.

If you select the enJin Console window, you will see the categorization of the classes as seen in Figure 4-10.

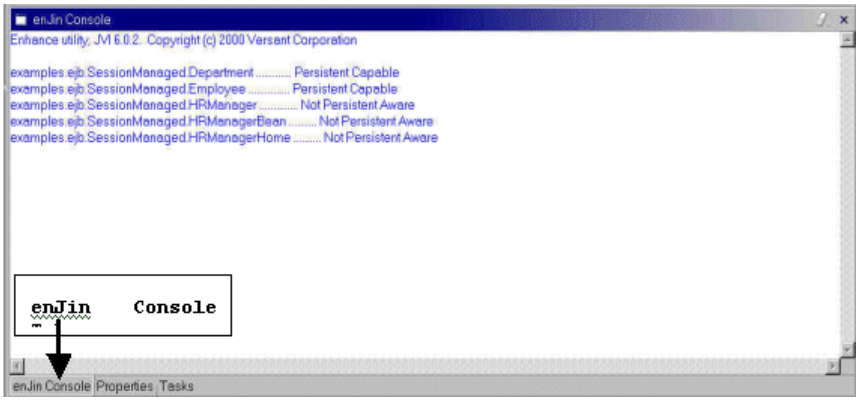


Figure 4-10 enJin console

Also, in the Navigator view, as shown in Figure 4-11, you will notice two new class files as seen below: Department_Pickler_Vj and Employee_Pickler_Vj - These are used by the Versant system to provide transparency of access and updates to persistent objects.

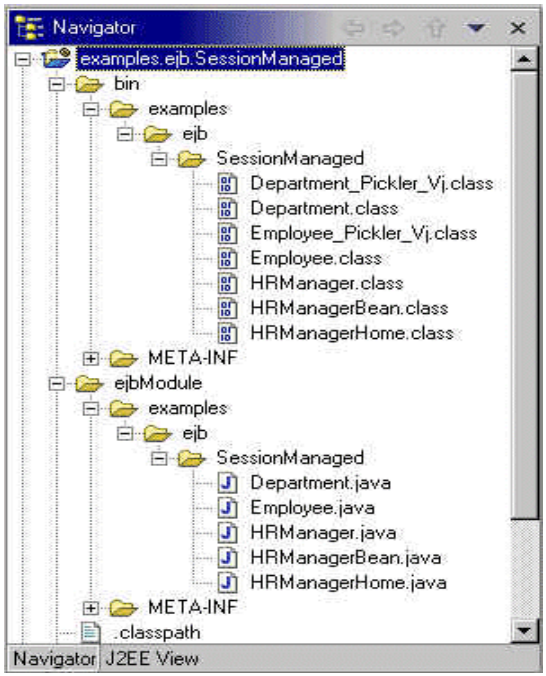


Figure 4-11 Navigator

The next step involves creating a database, where the objects that you have identified that are persistent will be stored.

You are just a few minutes away from deploying and testing your application.

4.2.4 Create the database

To create the database, run the DBAdministrator tool by either clicking the DBAdministrator icon, or select **enJin DatabaseTools -> DBAdministrator** as shown in Figure 4-12.

Figure 4-12 enJin Database Tools

Doing so, launches a new window, as seen in Figure 4-13. Select **File -> Create Database**, and this will display a new dialogue box.

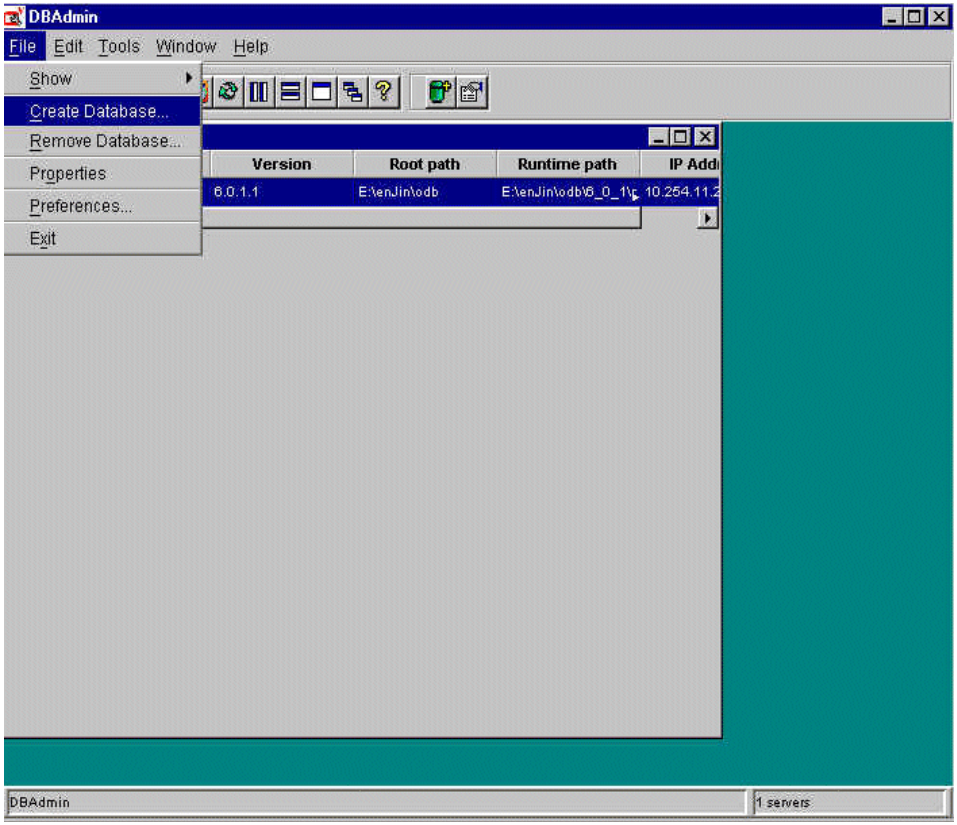


Figure 4-13 Database create

Please perform the steps in sequential order as demonstrated, and you will have a database where the objects that you create will be stored, as shown in Figure 4-14. Ensure that the server on which you to create your database has been selected.

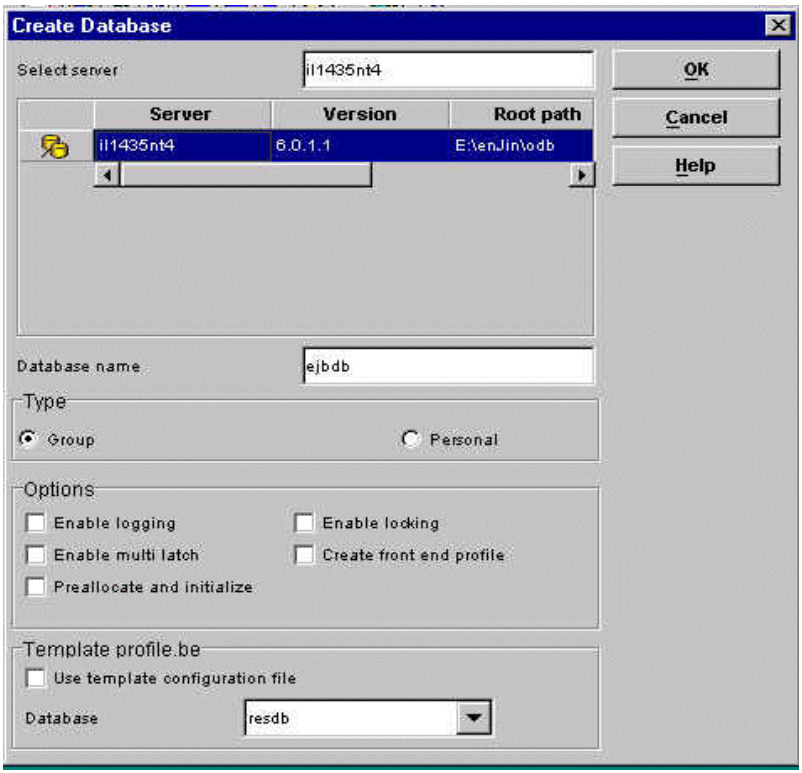


Figure 4-14 Create Database panel

Note: To eliminate any error please restrict the name of your database to ejbdb. This will allow the example that we have provided to be deployed and run without any error.

4.2.5 Deploying and testing the application

Within the WSAD environment deployment and testing your application is relatively effortless. You are just a few clicks away from running the Session Managed Example we have provided with Versant enJin.

Without further comment, let us commence the process of deployment and testing.

There are four steps involved in this process. They include:

1. Deploy code.
2. Modify server configuration file.

3. Run on server.
4. Test application.

Deploy code

Select the project we have been building in the navigator view. Right-click with the project selected, and select **Generate** from the choices presented. Upon doing so, you will be presented with three choices: select **Deploy** and **RMIC Tie code** as shown in Figure 4-15.

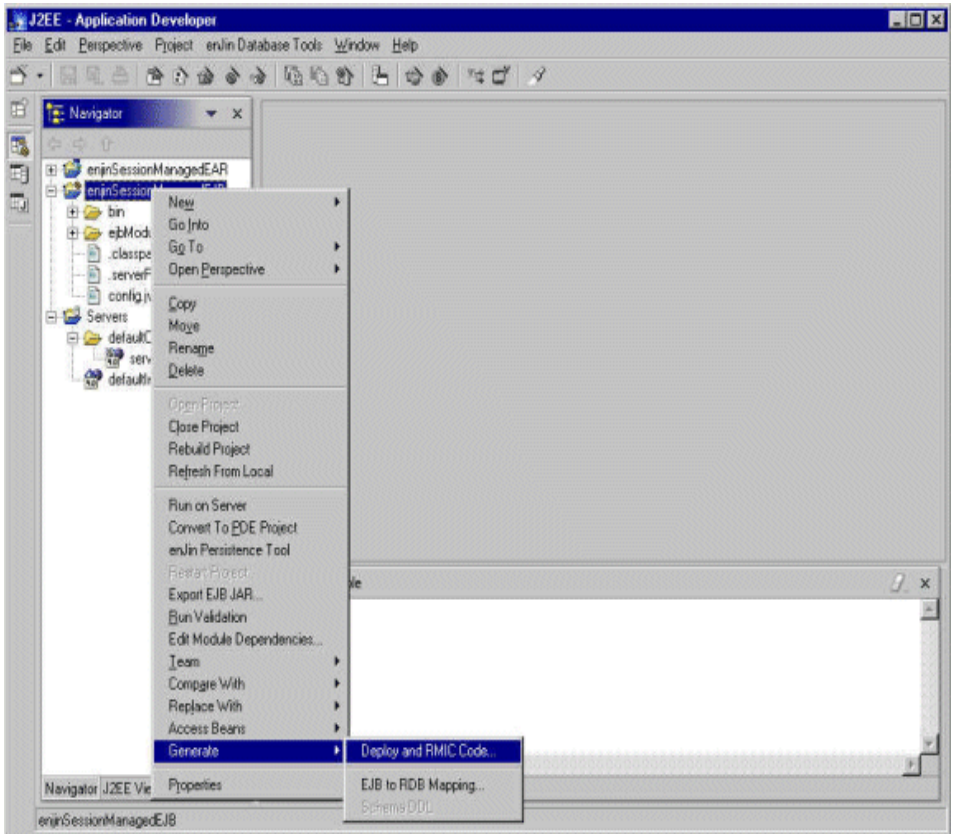


Figure 4-15 Deploy code

This brings up another dialog box that allows you to select the bean you want to deploy. Select **HRManager,** as depicted in Figure 4-16.

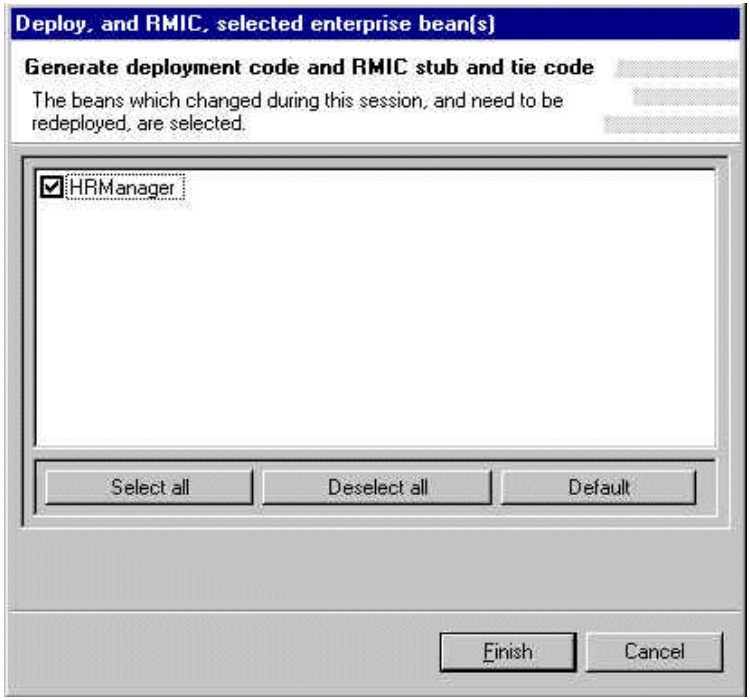


Figure 4-16 Generate

Select **Finish**. Upon doing so, WSAD performs numerous tasks for you behind the scenes. The work that WSAD does allows you to now run and test your application.

Modify server configuration file

You will need to add the following within the servers tag to your server-cfg.xml file for successful deployment. This must be done prior to running the HRManagerBean on the server.

```
<customServices
xmi:id=" enJinStartupPool_1" enable="true" externalConfigURL="
<wsadtoolszip_dir>\examples\vec-config.xml "
classname="com.versant.ejb.websphere.VECInitializer"
displayName="VEC Startup Pool" classpath="<ENJIN_ROOT>/lib/enjin.jar"/>
```

Note: Please ensure that you specify the configuration for your installation of enJin. For example, if you have installed enJin on your D drive, substitute <ENJIN_ROOT> with d:/enjin. Similarly, < wsadtoolszip_dir > refers to the directory where downloaded zip file contents have been placed.

If you do not have an instance of the server, you will have to create one. One way of doing this is by running the server and then stopping it. This can be done by from the J2EE view, by right-clicking after the bean had been selected from the project, and selecting **Run on Server** as shown in Figure 4-17.

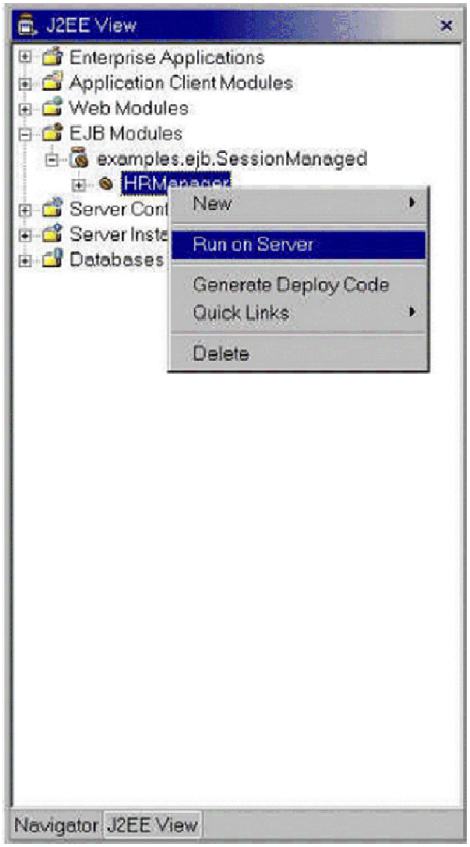


Figure 4-17 Run on Server

Upon doing so, WSAD will automatically change your perspective to the server perspective. Of course, you will not be successful in running the application since you have not configured your server yet. Select the Server Control Panel tab, and select the **Server Instance** that has just been created. Now, right-click with the instance selected, and select the **Stop** option.

Run on server

You are now ready to run your application on the server. To do this, please ensure that you are in the J2EE view, select the **HRMangerBean** that we have been building, right-click and select **Run on Server,** as shown in Figure 4-18.

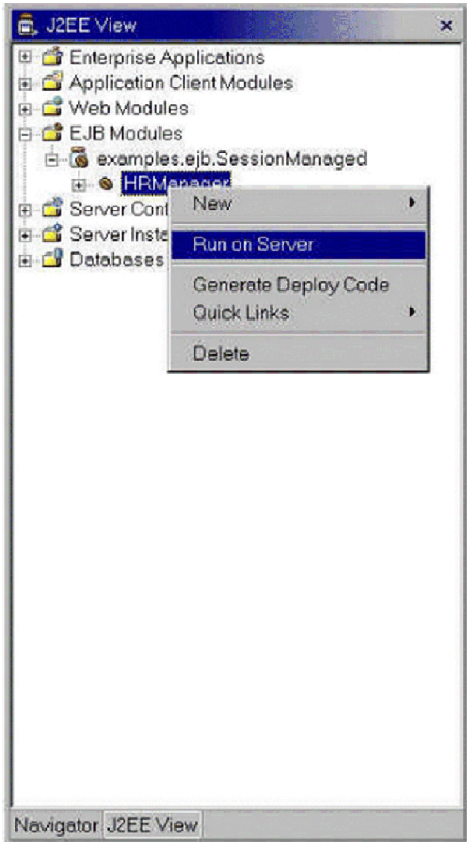


Figure 4-18 Run on Server

A test client will now appear upon successful deployment on the server. Please proceed to view the test client, as well as view the actual testing.

Testing the application

You will now be presented with a test an EJB Test client with which you can test your application, as shown in Figure 4-19. There are several steps involved in proceeding. We will step through each of these:

The first screen informs you that a constructor has not been selected.

1. Expand the HRManager and click on HRManager create() as the constructor.

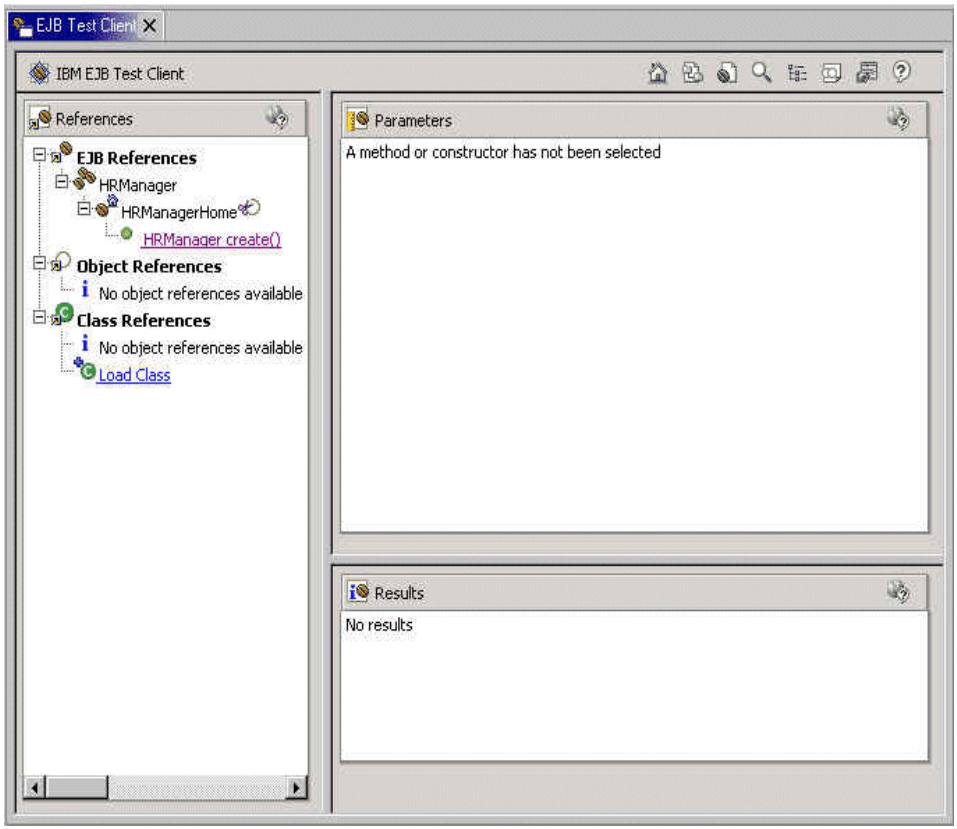


Figure 4-19 EJB test

2. Upon doing so, the right pane of your test client changes as displayed below. Click on the Invoke button as shown in Figure 4-20.

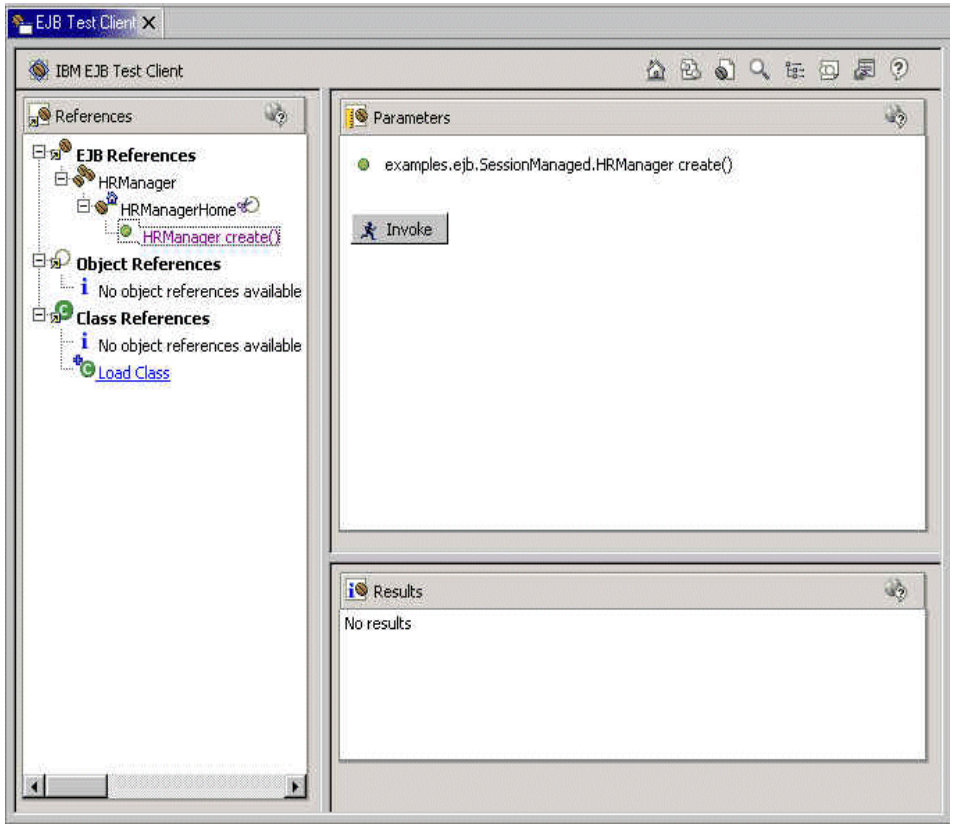


Figure 4-20 EJB test

3. Upon invoking the method, in the Parameters pane, a Work with Object button will appear as shown in Figure 4-21. Please click this button.

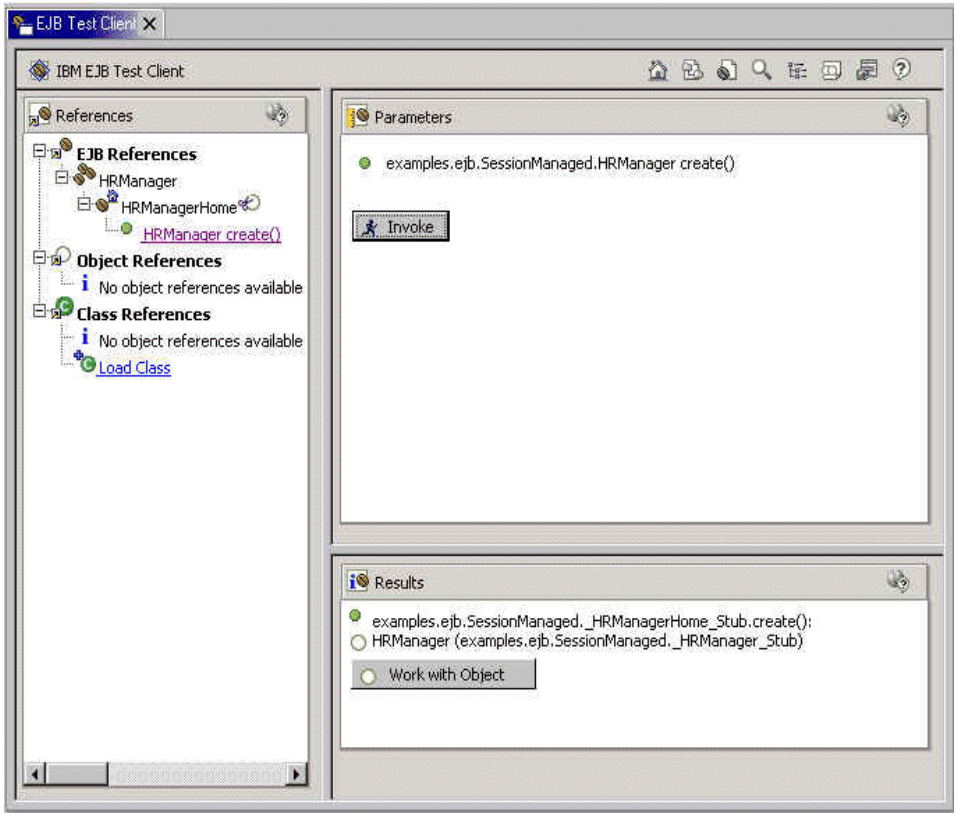


Figure 4-21 EJB test

You will now notice that References pane, on the left side of your screen will have changed. Upon expanding the HRManager, you will be able to call upon the methods. Of course, the first method would be to CreateDepartment, in which you can then add employees (newEmployees). Enter a department name, and click the **Invoke** button again.

You will see in the Results pane (as shown in Figure 4-22) showing that the method was created successfully. You may now add other departments, employees, or execute any of the other methods that are visible in the EJB References pane.

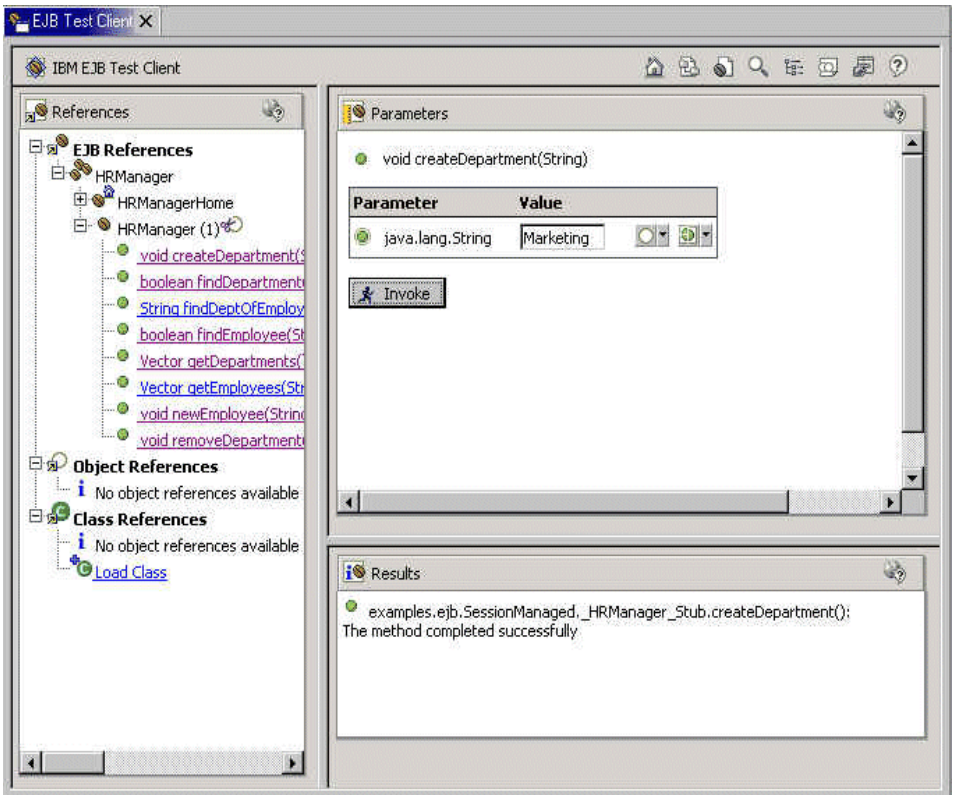


Figure 4-22 EJB test

In the lower half of the Parameters pane in the test client, you will notice that the method was created successfully. You may now add other departments, employees, or execute any of the other methods that are visible in the EJB References pane.

4.2.6 View the results in the database

Till now you have experienced the ease of development, deployment, and testing with the enJin tools integrated with WSAD. To continue to allow you to have the same ease with viewing the results of your EJB applications, enJin provides powerful database utilities. You have already used the DBAdmin to create a database. Now you will use the other DB tool, the DBInspector, which can be launched either by clicking the DBInspector icon, or selecting **enJin Database Tools -> DBInspector** as shown in Figure 4-23.

Figure 4-23 enJin database inspector

This will launch the DBInspector tool, with a list of databases as seen in Figure 4-24. Select the database we have created for this example (**ejbdb**) and select **OK**.

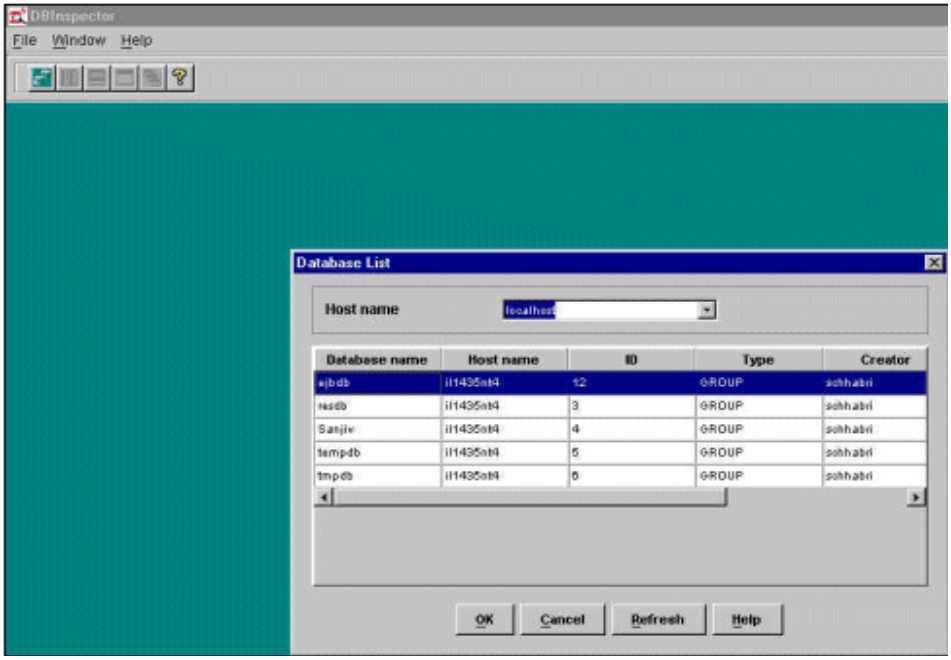


Figure 4-24 Database list

A connection will be made to the database, and you will be able to view the department we have just created. Simply double-click the class (in this case examples.ejb.SessionManaged.Department), and you will see the department we just created (Marketing) as shown in Figure 4-25.

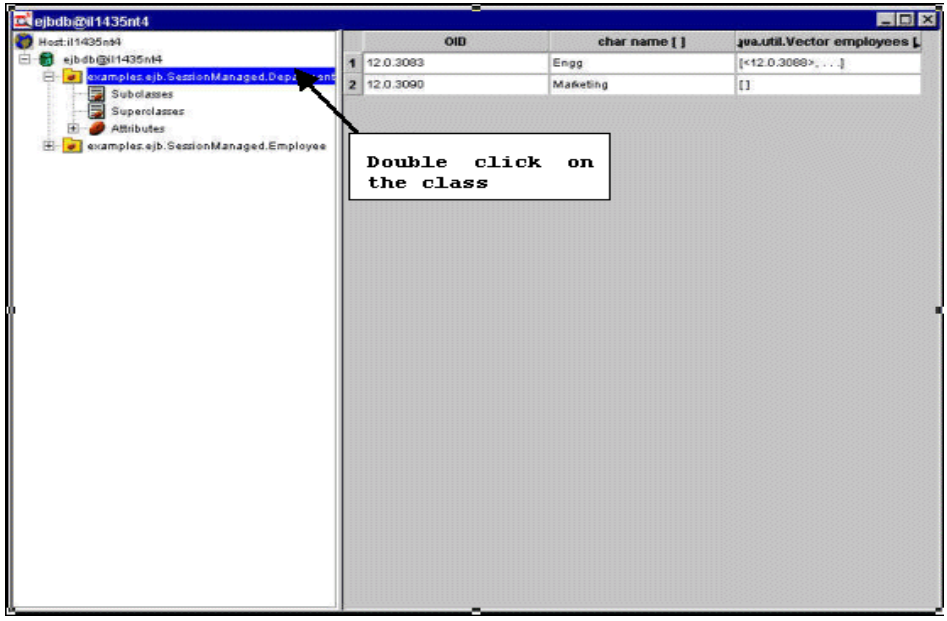


Figure 4-25 Database example

You can now continue to test your application. Create as many departments and employees as you like.

Part 3

The benchmarking project

In this part we will share with the user the details of the benchmarking project that was done, including the methodology, the application architecture, the performance metrics, and hardware configuration. The final part of this section is dedicated to present the users with the test results and analysis, as well as a summary of the project.

5

Benchmark methodology

The benchmark testing was performed using IBM's WebSphere Application Server and DB2 RDBMS, but since the performance bottleneck that enJin overcomes is merely the latency between middle-tier business objects and back-end tier relational data, these performance benefits apply equally well to all enterprise-class J2EE platforms.

This chapter is dedicated to describing in detail the methodology of the benchmark project. Specific topics that will be covered include:

- The Benchmark Application
- Performance metrics
- WebSphere configuration
- Hardware configuration

5.1 Benchmark application

The trade management application used for the benchmarking project was designed to reflect the performance metrics is intended to model the basic services required of a common financial application that manages account positions on investment. The basic use cases surround a theoretical company representing a brokerage whose employees manage corporate 401K plans. A particular employee of the brokerage will act as the custodian of several corporate portfolios, and make trades to change the position of the corporation within several market segments.

In order to provide performance measures that would be able to accurately reflect the performance characteristics of real-world applications, a stock trading application was designed and implemented for the benchmark tests. This trade management application was intended to model the basic services required for a common financial application - managing account positions on investments.

Based on the application architectures shown in Figure 5-1, two separate implementations of this application were developed: one for the baseline WebSphere-RDBMS architecture, and another separate implementation for the WebSphere-enJin-RDBMS architecture.

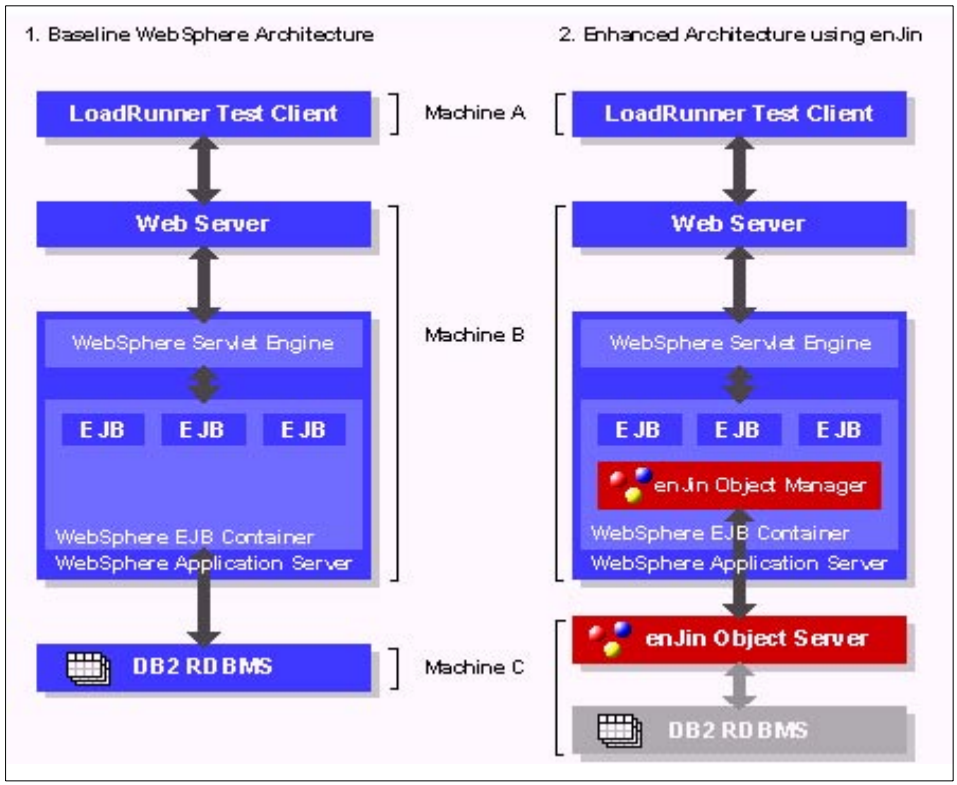


Figure 5-1 Application architecture

The representation above shows both the standard benchmark architecture, as well as the architecture enhanced by adding Versant's enJin. The standard architecture provided the baseline for measuring performance improvements due to the use of enJin's object caching facilities.

As can be seen in Configuration 1, the standard architecture consisted of IBM WebSphere Application Server running on a machine being served requests by IBM's HTTP server. WebSphere was configured to run both a servlet engine, as well as an EJB Container running several session and entity beans. The entity beans had their persistent state managed via Bean Managed Persistence code, and this persistent state was maintained in a DB2 relational database (RDBMS) on a separate machine. In order to take round-trip performance measurements, a third machine accessed the Web server and made HTTP requests for certain test actions to be made. This test client was run within the LoadRunner test harness package.

Configuration 2 shows that Versant enJin (the enJin Object Manager) has been added to WebSphere's EJB Container. In this enJin-based test architecture, Versant's enJin takes over coordination of all of the persistence and transaction management services from WebSphere's EJB Container, therefore, reads and writes are made to the objects in the persistent store provided by the enJin object server. This persistent state could then in turn be asynchronously propagated to a DB2 RDBMS, but since the asynchronous mechanism of updating such enterprise data stores does not happen in real time, these updates do not affect round-trip benchmark times, and so it was decided to simplify the benchmarking process and leave these updates out of the tests. Of course, since such updates can be configured to occur asynchronously, the performance times detailed in this document apply equally to a configuration where updates are propagated from Versant enJin to a DB2 relational database.

The baseline architecture was implemented by an independent IBM business partner using J2EE standards, IBM best-practices for implementing WebSphere applications, and using bean-managed persistence techniques in order to gain the best performance for the baseline implementation. The enJin-based implementation was completed by Versant staff using best-practices for J2EE and WebSphere, as well as best-practices for enJin-based development. This included the use of session-managed persistence techniques in order to optimize enJin performance.

5.1.1 Business domain model

The object model (shown in Figure 5-2) shows that users are managers of stock trading accounts. The users that this application models, are fund managers for corporate stock trading accounts. Each account contains all of the portfolios that are managed on behalf of a particular company by one of the fund manager users. Each stock trading account is composed of a number of portfolios, each of which represent the composition of a set of stock positions for a particular industry or market segment. A position represents a holding of shares, each share, option or other type of stock entity being modeled as a stock Instrument. Each position is also composed of a history of trades, and each instrument or stock is composed of a single specific current price quote. The dotted line boxes shown in the diagram represent the grouping of business objects into course-grained entity beans, producing a set of four entity beans for the application - user, instrument, account, and company entities.

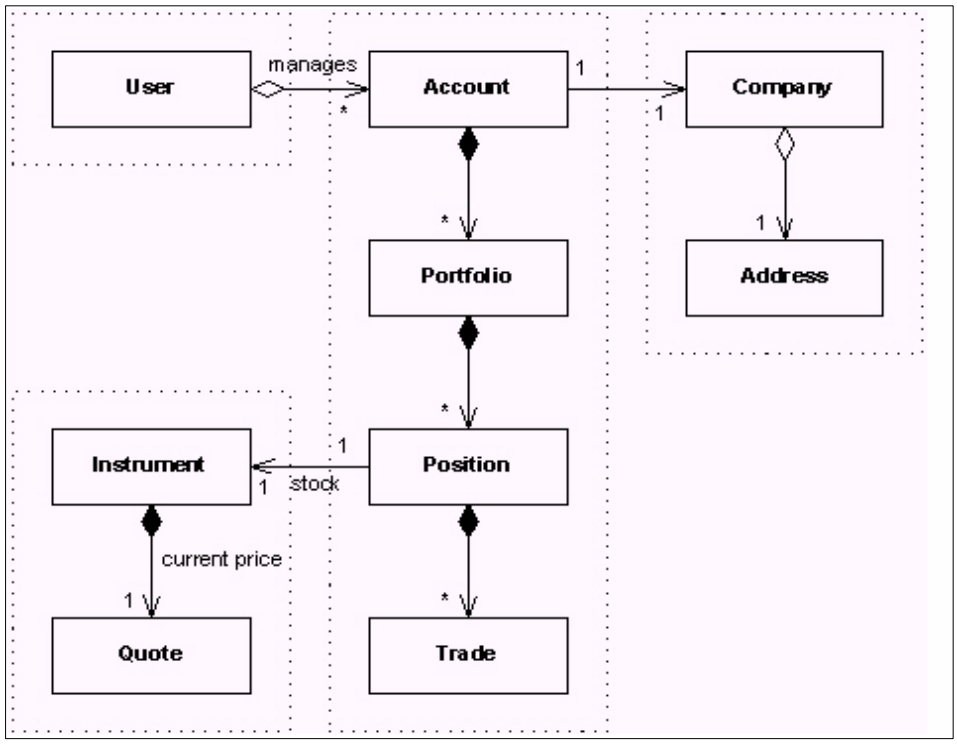


Figure 5-2 Object model

This object model was used as the basis of both implementations of the stock trading application used for the present benchmarking tests. The application is comprised of a number of different functions, some of which are administrative in nature and allow the pre-population of stock data, as well the viewing of current data in the system. For the purposes of obtaining a clear and concise set of benchmarks though, two particular functions within the application were focused upon. These functions were designed to examine the differences in performance characteristics between the traditional EJB-RDBMS architecture versus the enhanced enJin-based architecture, when these two implementations were exposed to various operational complexities. More precisely, two application functions were designed so that performance could be examined for:

- A simple operation
- A complex operation

By testing the enJin based and traditional implementations under these differing conditions, it was possible to determine the performance characteristics of the two approaches in the context of different application requirements. These two operations were:

- New trade (simple)
- New portfolio (complex)

The new trade transaction is a fairly simple operation, since it merely involves the alteration of one of the positions within an account, and the addition of a new trade entry. In contrast, the new portfolio operation involves the creation of a new portfolio, a new position, a new trade, and an association between the position and a stock Instrument, and so provides a more complex set of interactions to be tested. Therefore, performance characteristics of applications of differing complexities could be examined by benchmarking these two particular operations within the stock trading application.

5.1.2 Interactions measured by the benchmark tests

Figure 5-3 shows a high-level view of the interactions within a single end-to-end test run of the application for the baseline architecture. As shown, the standard J2EE architecture was used for the application flow, using EJBs, JSPs, and servlets to provide a model-view-controller architecture.

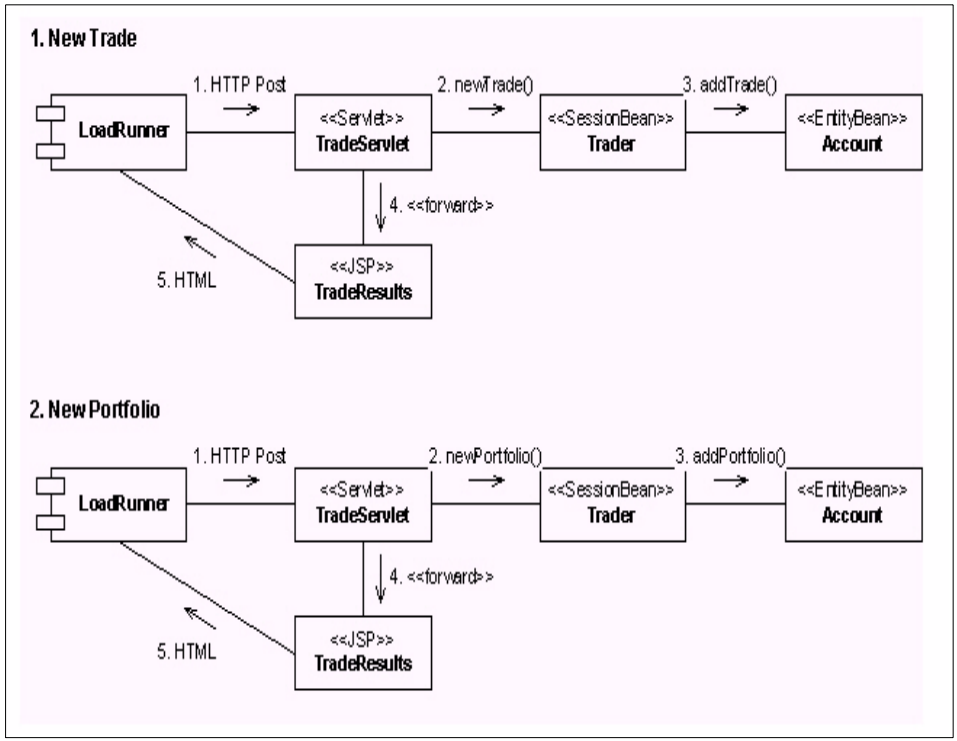


Figure 5-3 Interactions

The collaboration diagrams shown represent a simplified view of the interactions that were measured for the purposes of these benchmarks - these diagrams represent the baseline architecture, where the enJin object caching service was not used. Value objects, data access objects, Helper classes, and container-generated classes have been omitted in order to provide a clear overview of the interactions that were measured.

The first of the diagrams shows the new trade use case - the simple transaction condition. In this interaction, the LoadRunner test-harness makes a request on the TradeServlet, which takes parameters present in the HTTP request, and builds a trade value-object (a simple JavaBean) encapsulating the data to be used for creation of the new trade. The type of the value object created is determined by a parameter of the HTTP post request, which informs the servlet of which of the use cases to execute - here the *NewTrade* use case. The trade servlet then looks up the trader stateless session bean, and invokes the newTrade(Trade) method of the session bean's interface, passing the newly created instance of the trade object. The trader session bean then invokes one of the finder methods of the account entity bean's home interface, (based on information in the trade value object) in order to locate the appropriate entity bean to be updated.

The trader session bean then invokes the addTrade(Trade) method, passing the trade value object. The entity bean uses a Data Access Object Helper class, which uses SQL and JDBC to flatten the object and save it to the RDBMS. The flow of control then returns to the servlet, which forwards to a Java server page to generate an HTML response.

The new portfolio use case has similar interactions; they simply create different value objects, and invoke different methods of the trader session bean and account entity bean. The primary difference between the two scenarios in terms of complexity, is contained entirely within the entity bean's access to the database. Both scenarios use the same fairly simple and very standard J2EE sequence of interactions. They merely differ in regards to the complexity and number of objects that are updated, and so, differ in the complexity of the JDBC/SQL calls required to implement the updates.

For the enJin-based implementation, the two use cases have similar interactions. When using enJin however, rather than having the trader session bean invoke methods on an entity bean, and the entity bean persist the object via a Data Access Object Helper class, the enJin architecture instead treats the value objects as persistence-capable classes themselves, and so is able to bypass the entity bean and data access object layers. This *Session Bean Managed Persistence* approach allows us to persist the objects directly to the enJin object cache, using enJin's Java Versant Interface (JVI) API. This requires no SQL update statement to map the objects to relational tables. Java methods in enJin's

Java API are simply invoked to tell enJin to persist the entire object as an object. All other aspects of the interactions are the same. The trader session bean returns control back to the trader servlet, which then forwards control to a trade results Java server page, which then generates an HTML response to be returned to the LoadRunner client.

5.2 Performance metrics

So, for the baseline application, this sequence of interactions was the basis of all measurements taken. The round-trip response times detailed in the benchmark results represent all of the interactions above, from the LoadRunner test harness making a request upon the servlet controller, through to the EJB execution, back to the JSP generating an HTML page, and it being received by the LoadRunner test harness.

The measures of throughput are also based purely upon this measure. Throughput is measured in terms of the average number of transactions per second that the WebSphere platform can handle in either the baseline or enJin scenarios. Essentially, what this measures is the number of client requests that can be completely processed and replied to in a given second on average. It is important to emphasize that this is an average measure, so that even if the average response time is more than 1 second, if, say, 50 of the above interactions can be processed and responded to within 5 seconds, then the average throughput measure will be 10 transactions per second.

In addition to response times and throughput, CPU utilization and memory use were measured for each of these test conditions. This allowed for the estimation of the resources both used and available to the WebSphere platform to handle greater workloads.

5.3 WebSphere deployment configurations

The following list shows all of the parameters that were used to configure the EJBs, servlets, and JSPs for both the baseline test runs as well as the enJin test runs. Since WebSphere provides a vast array of configuration parameters for application tuning, only the primary configuration options, as well as all of the properties, which may have had any impact on performance have been shown.

DB2 Datasource

- JDBC Driver - com.ibm.db2.jdbc.app.DB2Driver (type 2 JDBC Driver)
- Minimum pool size - 10
- Maximum pool size - 500

- Connection timeout - 180 milliseconds
- Idle timeout - 1800 milliseconds
- Orphan timeout - 1800 milliseconds

Application Server (running within WebSphere)

- Tracing - none
- Object level tracing enabled - False
- Debugging enabled - False
- Transaction timeout - 2 minutes
- Transaction inactivity timeout - 1 minute
- Thread pool size - 20
- Security enabled - False
- EJB container
  - Cache size - 2 MB
  - Trader <Session Bean>
    - State Management: Stateless
    - JNDI Home Name - com/versant/benchmarkbmp/ejb/Trader
    - Minimum Pool Size - 2
    - Maximum Pool Size - 100
    - Transaction Attributes:TX_NOT_SUPPORTED (all methods of remote interface)
    - Transaction Isolation: SERIALIZABLE (for all methods of remote interface)
  - Account <Entity Bean>
    - JNDI home name - com/versant/benchmarkbmp/ejb/AccountWrapper
    - Find for update - False
    - Minimum pool size - 2
    - Maximum pool size - 100
    - DB exclusive access - False
    - Transaction attributes: TX_REQUIRED (for all methods of remote interface)
    - Transaction Isolation: READ_COMMITTED (all methods of remote interface)
    - Read only methods: none used
  - Servlet engine
    - Maximum connections - 100
    - Transport type - OSE (Open Servlet Engine)
    - OSE transport - INET Sockets (since running on SunOS)
  - Web application
    - Auto reload - True
    - Reload interval - 9 seconds
    - Shared context - False

- TraderServlet <Servlet>
  - Code - versant.benchmark.servlets.Controller
  - Load at startup - True
  - Debug mode - False
- JSP 1.1 Compiler <Servlet>
  - Code - org.apache.jasper.runtime.JspServlet
  - Load at startup - True
  - Debug mode - False
- Session manager
  - Enable sessions - True
  - Enable URL Rewriting - False
  - Enable cookies - True
  - Shared context - False

All of the configuration parameters shown above were kept the same between both the baseline architecture, as well as the enJin-based test architecture. All of the WebSphere configuration parameters not shown have little or no impact on the performance characteristics of the test application, but were also kept the same between both the baseline and enJin deployments in order to ensure accurate benchmarking.

5.4 Other runtime parameters

Before running any of the benchmark tests, the application database was pre-configured with data for 500 different users as well as other context data including companies, accounts, and other related information. This context data totaled approximately 10 MB of pre-test data.

5.5 Hardware configuration

The hardware configuration is as follows:

LoadRunner Client:

- IBM Netfinity 8500R
  - 4 x PIII Xeon 550Mhz
  - 2560 MB SDRAM ECC
  - 10/100 Mbps Ethernet
  - Microsoft Windows NT 4.0 Service Pack 6
  - LoadRunner 6.5 client software

Web & WebSphere Server:

- Sun Ultra 80:
  - 4 x UltraSparc-II 450 Mhz,
  - 4MB Level 2 cache
  - 4096 MB RAM
  - 10/100 Mbps Ethernet
  - SunOS 5.8
  - IBM HTTP Server v1.3.19
  - IBM WebSphere Application Server Advanced Edition v3.5.3
  - Versant enJin 2.1

DB2 RDBMS Server:

- Sun Ultra 80:
  - 4 x UltraSparc-II 450 Mhz,
  - 4MB Level 2 Cache
  - 4096 MB RAM
  - 10/100 Mbps Ethernet
  - SunOS 5.8
  - IBM DB2 6.1 fixpack6
  - Versant enJin 2.1

Network:

- All machines on 100 Mb/s switched fast Ethernet

6

Benchmark results

Versant's enJin provides a unique way of managing such new demands by providing facilities for high-speed persistence of intermediate application data, which may not need to be stored in backend systems, as well as persistence of middle-tier-only data. However, enJin also provides a middle-tier object cache for information that is stored in backend enterprise information systems such as relational databases: a cache which provides tremendous performance advantages. This object cache provides transaction integrity and persistence capabilities to ensure consistency between data that is stored in this cache and the data contained in backend information systems. It also ensures the integrity of transactions in cases of server failures, and ensures consistency between multiple caches in an environment of clustered J2EE application servers.

As shown by the benchmarks presented within this chapter, the addition of Versant's enJin to a standard Enterprise JavaBeans, servlets, and Java server pages application provides the following performance benefits:

- Elimination of between 77% and 89% of the delay between a user making a request and receiving the response. In other words, application response times that are up to 9.6 times faster.
- The ability to handle between 3.6 and 8.5 times the number of concurrent transactions per second.

Moreover, these performance benefits are obtained while actually using fewer CPU resources, and equivalent memory resources. With enJin's dramatic performance and throughput gains with zero additional resource requirements, the return on an investment in enJin for a J2EE-based application is easily shown to be on the order of 53-61% savings in hardware and software platform costs alone, based on very conservative estimates. Depending on the scale of an application's deployment, these savings are likely to translate into platform cost reductions anywhere between $175,000 for modest deployments, to around $675,000+ in savings for enterprise scale deployment topologies.

The following metrics were metrics were evaluated in this project:

- Mean Response Time - Time taken from LoadRunner making an HTTP request to LoadRunner receiving an HTML reply for that particular request
- Throughput in mean number of transactions per second - Average number of round-trip interactions that could be responded to within 1 second.
- Mean CPU utilization - Average CPU activity for each of the 4 processors in the WebSphere Application Server machine, as well as each of the 4 processors in the Data Server machine.
- Mean Memory Utilization - Average amount of memory used by the WebSphere Application Server and database server machines.

The subsequent section will detail the results of the project. Specifics that will be discussed include:

- Response-Time performance and application server throughput
- Resource Utilization
- Scalability

6.1 Response time and application server throughput

The first set of measures taken (as shown in Figure 6-1) used LoadRunner to execute requests using a single thread, 2 concurrent threads, 3 concurrent threads, and so forth, up to 10 concurrent LoadRunner threads making requests upon the application in each of the baseline and enJin conditions. Each of these LoadRunner *threads* was simply a concurrently running test-harness, which when aggregated, simulated the demand that would be expected from between 10 and 400 concurrent users, depending on the test condition. Each test harness thread waited until it had received an HTML response before making a new request.

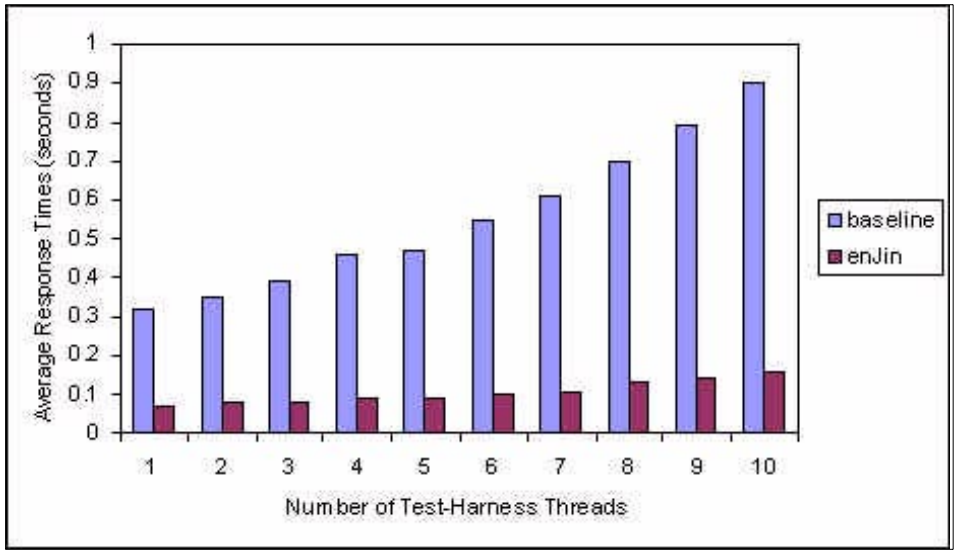


Figure 6-1 Response times for simple (new trade) transactions

As the chart in Figure 6-1 shows, as the amount of workload placed on the baseline WebSphere implementation grew, response times began to suffer, slowing down from 320 milliseconds with 1 test harness thread, to 900 milliseconds with 10 concurrent threads. In contrast, the implementation which used enJin's object cache had far better response times, averaging a mere 70 milliseconds with a single test harness thread, to 160 milliseconds with 10 concurrent test harnesses. So, not only did the architecture using enJin's object cache give much better response times, but it also showed much more stable response time performance as workload increased.

In addition to producing much faster response times, due to the nature of the LoadRunner test-harness which waits for a response from a previous request before making a subsequent request, enJin actually achieved these better response times under a greater load than the baseline architecture. Table 6-1 shows the average number of transactions completed per second for the same test runs as shown in the Figure 6-1 previously.

Table 6-1 Throughput as measured by transactions per second

Threads	1	2	3	4	5	6	7	8	9	10
Baseline	2.87	5.14	7.12	8.04	9.86	10.18	10.74	10.83	10.82	10.45
enJin	10.46	20.06	28.70	36.31	44.12	47.70	50.02	48.74	50.69	49.77

So, not only did enJin perform much better than the baseline condition in respect of response times, but it was producing these better response times under much greater workload.

The results (shown in Table 6-1) illustrate enJin's performance characteristics for a simple application scenario.

In order to gauge performance under more complex application demands, the above measures were repeated for the complex transaction condition, such as a request to add a new portfolio. These results are shown in Figure 6-2.

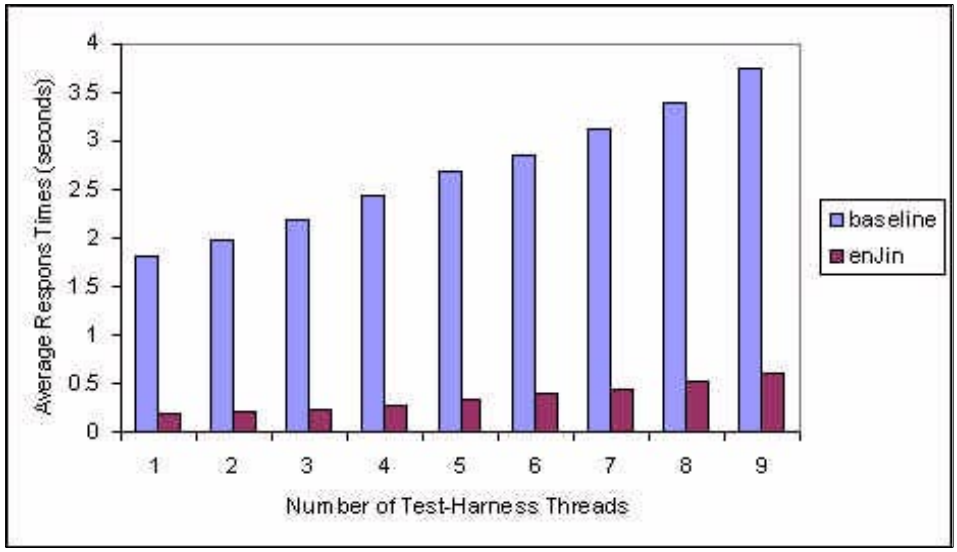


Figure 6-2 Response times for complex (new portfolio) transactions

Figure 6-2 shows the response time performance gains are even greater for more complex application scenarios, and again enJin's performance was more stable as workload increased. enJin was 9.6 times faster for a single test harness thread (190 milliseconds versus 1820 milliseconds) and 6.2 times faster with 10 concurrent test harness threads (600 milliseconds versus 3730 milliseconds).

Again, these response time gains were made even though enJin was responding to many more requests - up to 8.3 times as many requests - as shown in Table 6-2.

Table 6-2 Throughput as measured by transactions per second for complex

Threads	1	2	3	4	5	6	7	8	9
Baseline	0.53	0.98	1.33	1.58	1.81	2.04	2.19	2.3	2.34
enJin	4.53	8.29	10.82	12.42	13.49	14.00	14.15	14.16	13.54

So, for both Simple and Complex transactions, adding enJin to a standard WebSphere J2EE application can cut response times by as much as 89%, while simultaneously improving the number of requests that can be serviced by as much 755% (both of these estimates taken from complex-1 thread condition).

All of the response time data presented above is purely for round-trip response times - in other words, the response times that an end-user will experience. In order to get a better picture of the internal performance differences, log timings were also taken at key points within the internal application execution, in order to isolate just the data access times. Figure 6-3 shows the performance of the baseline and enJin enhanced implementations at the point where enJin actually targets its performance optimizations; the time taken within the application just to access the persistent store.

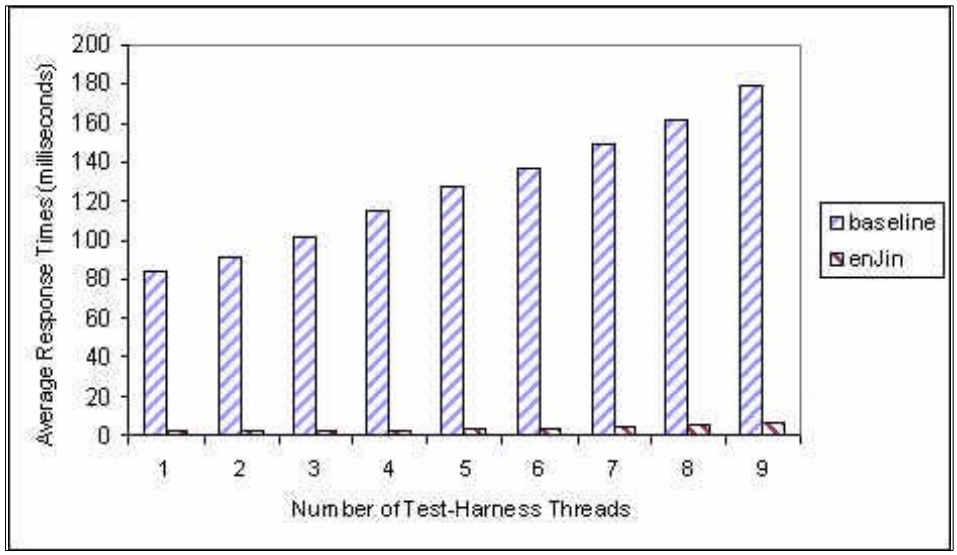


Figure 6-3 Internal timing of data access component of round-trip transaction times

As shown in Figure 6-3, when the data access component of response times is isolated from all of the other activities within a full round-trip execution of a transaction, the performance benefits of enJin are highlighted even further. This data is taken from the complex transaction executions, for the same test runs as shown in Figure 6-2.

Comparing just the data access times then, enJin cuts between 96.4% and 97.4% of the data access latency for complex transactions. Similar data was found for simple transactions (not shown), where data access times within the application were cut by between 90.4% and 94.4% when comparing enJin to the baseline RDBMS-only architecture. So at a minimum, data access latency within an applications execution is only 9.6% the amount of time needed for an update or query to a relational database, and may be as little as 2.6% the time taken to query or update a relational database, rather than query or update enJin's object cache (and have the relational database updated asynchronously).

6.2 Resource utilization

In addition to measuring improvements in response times and transaction throughput, utilization statistics for application server and database server resources were also collected. This data was captured in order to determine what effect, if any, would occur in regards to the availability of system resources.

Does enJin place a greater burden on system resources, or does it actually free up system resources because transactions are executed so much more quickly? CPU utilization and available system memory were measured to investigate enJin's resource utilization characteristics.

6.2.1 CPU utilization

Figure 6-4 displays CPU utilization for simple and complex transaction conditions, for the same test runs as presented in the analysis of response times and throughput.

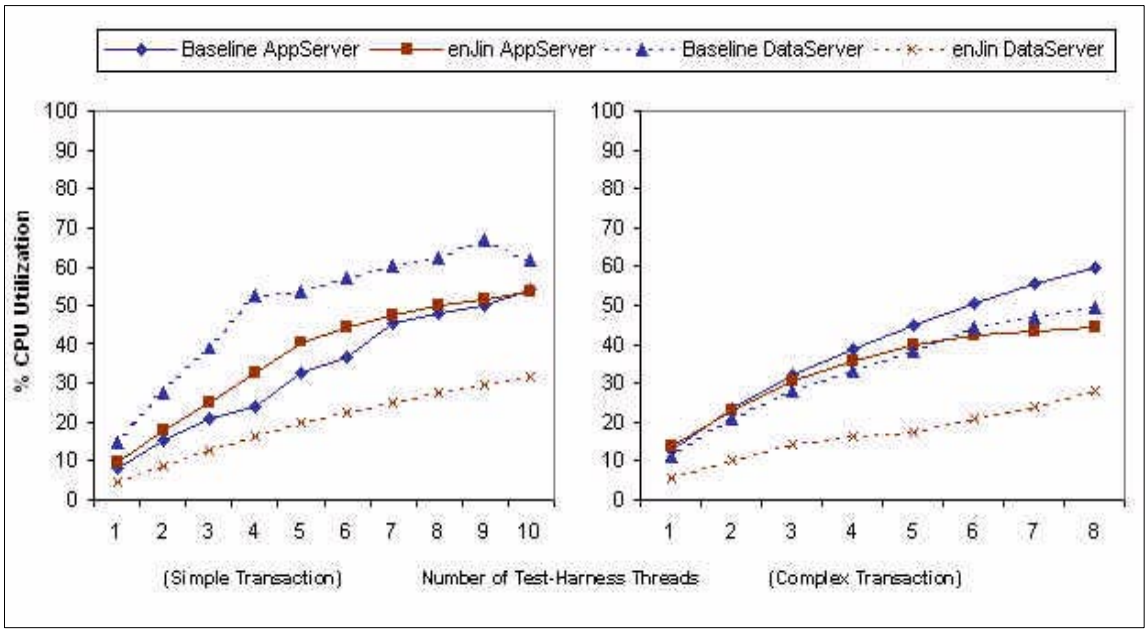


Figure 6-4 CPU utilization for simple (new trade) and complex (new portfolio) transactions

In Figure 6-4 we can see in the simple transaction scenario that enJin uses slightly more application server CPU time than the baseline in some conditions, but overall, there was no significant difference, with the 10 threads condition actually using marginally less CPU time with enJin. For the data server machine however, much less processor time was used in the enJin condition, since we were not performing updates to the RDBMS in real time. The complex transaction scenario, however, shows that for both the application server and data server, fewer CPU cycles were needed to service the complex transaction requests when using enJin rather than the baseline architecture.

So again, we can see from Figure 6-4 that even though the enJin-based architecture was servicing between 4 and 9 times as many requests, and responding to each of these requests much faster than the baseline WebSphere architecture (as shown in Figure 5-1), it produced this performance improvement while actually using generally fewer application server and data server CPU cycles.

6.2.2 Memory utilization

Figure 6-5 displays memory utilization for the same test runs as presented above:

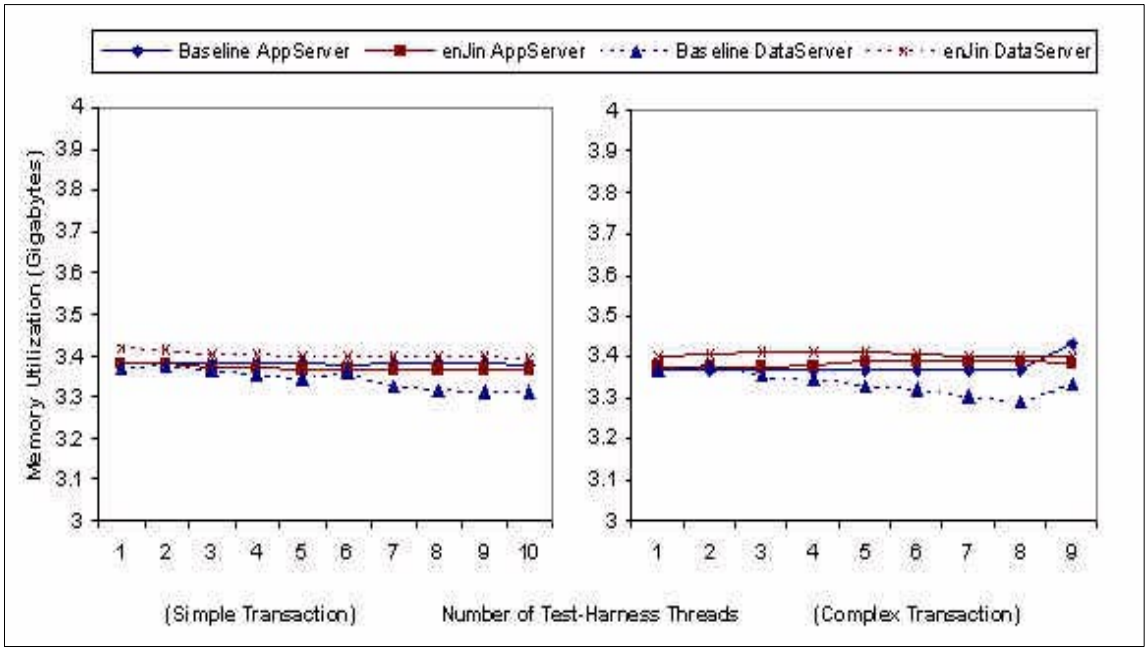


Figure 6-5 Memory utilization for simple (New Trade) and complex (New Portfolio) transactions

Figure 6-5 showing the simple and complex transactions clearly show that memory utilization was not at all a factor in the benchmark results. The scales for Figure 6-5 range from 3 GB to 4 GB of in-use memory, and so the slight differences that can be seen are visible primarily due to the expanded scale. For both the application server and data server, memory use ranged from 3.31 to 3.43 Gigabytes with a mere 3% variation. So, the use of enJin requires no additional memory resources when compared to a traditional architecture.

6.3 Scalability

The benchmarks show that by adding enJin to the WebSphere platform we can achieve much faster response times, and much higher throughput. The application server spends fewer CPU resources executing each transaction, and so it can make itself available for further work more quickly.

This does not actually make our application more scalable; the application's scalability is still guaranteed by WebSphere's ability to cluster many application servers to share increased workloads as user demands increase. Even though enJin's performance characteristics do not make our application more scalable, they do provide our application the ability to scale to meet a given demand with fewer resources.

A

Versant WebSphere Consulting Practice

Versant's core competency in object-oriented applications and expertise in Java for the application server has lead to a strategic partnership with IBM to deliver WebSphere Consulting Services. The Versant WebSphere Consulting Practice consists of WebSphere experts in EJB application architecture, WebSphere back-end developers, VisualAge for Java consultants, and WebSphere systems administration experts. Through this consulting practice, Versant supports IBM's explosive growth in the application server business by helping their customers design, architect, implement and roll-out WebSphere applications.

Some examples of the type of WebSphere Java application projects where these consultants help our clients:

- Application architecture/design
- Object-oriented analysis and design
- e-business application development using Java 2 technologies such as EJB, JSP, and Servlet technologies
- WebSphere performance and tuning consulting
- Application project management
- VisualAge for Java mentoring
- WebSphere deployment

- WebSphere installation, setup, and systems administration
- Managing roll-out of WebSphere systems for production implementation

For more information please see the following URL for the WebSphere consulting practice at Versant is:

http://www.versant.com/services/profservices/index.html

Index

S

T

U

V

W

X

(0.2”spine)
0.17”<->0.473”
90<->249 pages

Accelerating IBM WebSphere Application Server Performance with Versant enJin

Improve data access by more than 50 times

Speed transaction throughput by 10 times

Eliminate object-to-relational mapping code

This IBM Redbook is an in-depth guide for implementation of Versant enJin methodologies and practices for development and deployment of your J2EE applications using IBM WebSphere Application Server.

Versant enJin is a flexible infrastructure platform that persists Java objects and EJBs within the middle-tier without overloading your existing database systems.

Details of the benchmarking project that was a joint undertaking with IBM and Versant conducted at the IBM Silicon Valley Lab in San Jose are covered.

Part 1 describes the methodology used in developing J2EE applications using Versant enJin.

Part 2 covers how to implement the methodology into developing and deploying a typical J2EE application using IBM WebSphere and enJin.

Part 3 describes the details of the benchmarking project that was done including the methodology, the application architecture, the performance metrics, and hardware configuration.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-6561-00 ISBN 0738425109