Caché: Flexible, High-performance Persistence for Java Applications

Introduction

Java is indisputably one of the workhorse technologies for application development. Its "write once, deploy anywhere" nature makes the Java environment a natural choice for creating Web-based and integrated applications and, as an object-oriented language, Java enables rapid development. But it is Java's object orientation that has historically made providing data persistence to Java applications a challenging task.

Two basic approaches have evolved. One is to model information as relational tables, using JDBC to store and retrieve data. This has the advantage of making Java applications compatible with relational databases, which are still widely used. However, the "mapping" that is required to translate between complex objects and two-dimensional tables exacts a toll, both in developer productivity and application performance.

The other basic approach for providing data persistence within Java-based applications is to store and retrieve data as objects. This approach avoids the productivity and performance losses caused by object-relational mapping, but it assumes the use of an object database. The drawback is that many organizations do not currently use object databases.

The same choice (store data as tables, or store data as objects) applies to providing persistence for Enterprise Java Beans (EJB), with the added complication of deciding whether to use container-managed persistence (CMP) or bean-managed persistence (BMP). The latter usually results in better application performance, but the former has historically been quicker and easier to implement.

This paper will discuss how Caché simultaneously and seamlessly allows data to be accessed as both relational tables and objects. It will detail the various ways Caché can be used within the Java environment, with particular emphasis on Caché's EJB projection that allows the automatic generation of J2EE entity beans that use high-performance bean-managed persistence.

Multidimensional Flexibility – Caché's Unified Data Architecture

With Caché, Java developers are not locked into modeling data either as relational tables or as objects. That's because Caché is neither a relational nor a "pure object" database. Rather, its underlying data engine stores information as sparse multidimensional arrays. Multidimensional structures are rich enough to represent the complexity of data objects, but they are also easily projected as two-dimensional (i.e.: tabular) structures. Thus, Caché's efficient multidimensional arrays can be simultaneously presented to developers as relational table or as objects.

This dual nature is referred to as Caché's Unified Data Architecture. It allows Java developers to choose how they prefer to provide persistence for their applications.

Treating Data as Relational Tables – Caché SQL and JDBC

Some Java developers will choose to treat Caché like a relational database, using SQL and JDBC to manipulate data. This approach is particularly useful for developers who want to use Caché with existing Java applications that already include SQL calls to a relational database. Because Caché supports relational data access, such applications can run against Caché with only the slightest of changes.

On the Java client side, the only changes are those needed to connect to the Caché data server. First, the system environment variable CLASSPATH must be modified to include the location the

---

[1] Actually, a *triple* nature, since Caché also allows direct manipulation of its multidimensional arrays. However, direct data access is probably of less interest to Java developers than Caché's SQL and object access.

[2] See the Caché documentation for details.

CacheDB.jar file. CacheDB.jar (which is provided by InterSystems and included in the default Caché installation) contains "pure Java" versions of all the Caché system classes needed for Java support.

Next, several packages from CacheDB.jar must be imported into the Java application:

- *com.intersys.objects* implements the connection and caching mechanisms needed to communicate with a Caché server.
- *com.intersys.cobjects.reflect* allows applications to take advantage of Java reflection functionality.
- *com.intersys.jdbc* supplies JDBC connectivity.

Once these packages have been imported, Java developers can use standard methods to connect to the Caché server and run SQL queries or commands. Caché includes a Type 4 JDBC driver that converts JDBC calls into a Caché native protocol format for direction communication to the Caché database. Caché's JDBC driver is JDBC 2.0 compliant with extensions.

Of course, the target Caché database needs to contain the appropriate data tables. For the case of an existing Java application being converted to Caché, there must be a relational data schema already in use. Those DDL table definitions can be imported into Caché and compiled. (Whereupon Caché's Unified Data Architecture will automatically make those relational data structures available as objects as well.) For new development, Caché comes with a wizard-driven IDE that enables the quick and easy creation of the necessary data structures.

What are the advantages of using SQL and JDBC to access a Caché database? For existing Java applications that currently access a relational database, the answer is better performance and system management characteristics. Although Caché can be accessed as though it was a relational database,

information is actually stored in efficient multidimensional arrays. The use of multidimensional data structures eliminates the "joins" and "table hopping" that are typical of relational technology, thus allowing faster response to SQL queries. In fact, Caché's SQL response is typically five times faster than a relational database.

For new Java development, the main advantage of using SQL and JDBC to provide persistence is database independence. The application may need to be able to run against relational databases as well as Caché. Also, some developers who are well versed in JDBC may prefer it to other approaches. But for the most part, if Java developers are given the choice, they will store and manipulate information as objects.

Treating Data as Objects – Caché's Java Binding

By using Caché as an object database, Java developers do not have to endure the decrease in productivity and performance caused by object-relational mapping. This approach will most likely be used for new Java applications, when developers have a choice of how they want to model data.

Caché fully implements object modeling concepts that will be familiar to Java developers, including: encapsulation, multiple inheritance, polymorphism, referenced and embedded objects, collections, and relationships. Caché classes are easily defined and edited in the Caché Studio, a graphical IDE that includes wizards for the quick completion of common development tasks. One of these wizards is the Projection Wizard, which is a quick way to project Caché classes as (among other things) Java classes or Enterprise Java Beans.

If a class definition contains a Java projection, Caché will, upon compilation, generate both the Caché class and a corresponding Java proxy class.

The proxy class will be written in pure Java. It will inherit its persistence methods (to store, retrieve, and delete the class) from the pure Java system class included in CacheDB.jar. Being a Java class, all properties of the proxy class are considered to be private. Accessor methods are automatically generated and included in the class definition by Caché.

The proxy class (being a true Java class) must itself be compiled before it can be run. Once the proxy class has been compiled it can be used in any Java application that can connect to the Caché server. The client-side requirements for making a connection are the same as for using JDBC, namely, the location of CacheDB.jar must be added to the application's CLASSPATH declaration, and the appropriate packages from CacheDB.jar must be imported. A Caché server connection can simultaneously support both JDBC and object access to the Caché database.

By giving Java programmers true object access to the Caché database engine, a much better distribution of work between Java and Caché can be implemented, In general, any method that needs to access the database will run faster on the Caché server. However, the Caché run-time environment does not currently include a Java virtual machine. Caché-server side methods must be written in either Basic or Caché ObjectScript.

If developers wish to add client-side business logic to a Java proxy class derived from a Caché class, it is recommended that they first create a new class that inherits all its properties and methods from the Java proxy, and then make any alterations or additions to the new class. In that way, if any edits are made to the Caché class on the data server, and a new version of the corresponding Java proxy is created, the client-side alterations will not have to be repeated.

There is another option for adding client-side business logic. Caché allows Java code to be includ-ed in Caché class definitions, but it will not be compiled into the executable form of the Caché class. Instead, that code will be included (exactly as written) in the Java proxy class. Once the Java proxy is compiled, the code will run on the Java client.
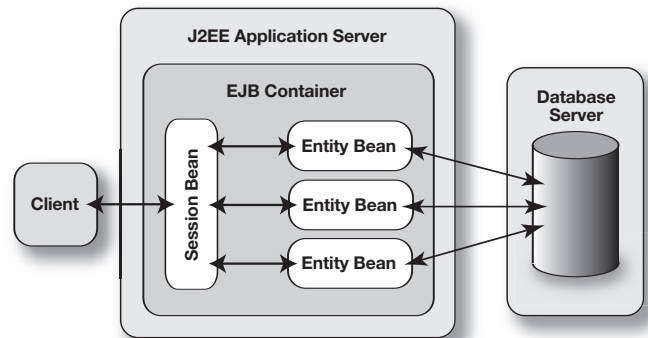
Working with Application Servers – Enterprise Java Beans

Enterprise Java Beans are Java classes designed to run on any EJB-compliant application server as part of an n-tier, distributed application. Application servers provide system-level services such as transaction monitoring, security, threading – and an interface to a persistence engine.

A generic EJB architecture is shown in Figure #1. There are two basic kinds of beans:

■ Session beans are not persistent and have no state. They generally contain business logic and interact with one or more entity beans. Caché can project classes as session beans. (See the discussion of how to implement business logic written in Java, above.)

■ Entity beans are persistent and therefore must interact with a database. As with any Java class, entity beans can access Caché using JDBC or Caché's Java binding.

Figure #1: Typical J2EE Architecture

One further consideration for entity beans is whether the application will use container-manage persistence (CMP) or bean-managed persistence (BMP).

Container-managed persistence means that the application server software handles the database interactions for all entity beans. The advantage of CMP is that Java application developers don't have to write persistence methods for each entity bean. The disadvantage is that CMP can result in poor performance – not only does CMP add an additional layer of processing between the bean and the database, but application server software invariably uses SQL and JDBC to provide persistence. Although that makes the J2EE application compatible with most commonly used databases, the processing overhead needed to overcome the "impedance mismatch" between Java objects and relational tables can have a negative effect on performance.

For J2EE applications that require high performance, many Java developers choose to use bean-managed persistence. That means writing custom persistence methods for each entity bean, so that each bean "knows how to store itself" in the most efficient way. Usually, BMP is used with a database that can store entity beans as objects, thus avoiding the performance degradation caused by impedance mismatch. The advantage of BMP is high performance. The disadvantage is that it usually requires a lot of hand coding, which may result in longer development and maintenance cycles.

Caché's EJB Binding

Caché projects classes as Enterprise Java Beans in essentially the same way it projects them as standard Java classes. All that is required is to include the projection statement, as shown in Figure #2, below. When the Caché class is compiled it will generate all the required J2EE-compliant .java bean code, all the required interfaces, a test servlet, a test Web page, and all the deployment descriptors required by the application server.

While the code generated is J2EE compliant, there are a few items which are specific to each EJB application server, for example, deployment descriptors. As of the date of this document, Caché supports the following EJBs application servers .

- BEA WebLogic Server 8.1
- JBoss 3.2
- Pramati 3.5
- WebSphere 5.1

For each EJB projection, there are a number of parameters that must be set by the developer. (The projection wizard provides a list of the information Caché needs in order to generate the EJB. More experienced developers can "hand code" the projection into the Caché class definition.) One of these is the PERSISTENCETYPE parameter, which indicates whether the EJB is to use container-managed or bean-managed persistence.

Figure #2: A Typical EJB Projection Statement

This projection statement, creating a BMP projection for BEA WebLogic 8.1, would be included in the Caché class definition.

```
Projection BeaWebLogic8 As %Projection.EJBWebLogic(APPLICATIONDIR =
"C:\bea\user_projects\domains\mydomain\applications", APPSERVERHOME =
"c:\bea\weblogic81\server", BEANNAME = "MyBean", CLASSLIST =
"MyPackage.MyClass", JAVAHOME = "c:\bea\jdk141_05", PACKAGE =
"MyPackage", PERSISTENCETYPE = "BMP", ROOTDIR =
"C:\path_to_projected_files", SERVERTYPE = "WEBLOGIC8");
```

[3] Please see www.intersystems.com for the most current list.

If CMP is chosen, Caché will generate standard JDBC-compliant Java code. For BMP, Caché automatically implements (using object data access) several persistence methods in the EJB, including:

- ejbCreate( )
- ejbLoad( )
- ejbStore( )
- ejbRemove( )
- ejbFindByPrimaryKey(string key)

These methods, which could take hours to code in other technologies, are automatically created by Caché in seconds. Using Caché, J2EE developers can quickly and easily implement bean-managed persistence. Applications benefit from high performance while avoiding the long development cycles that are typical of BMP. Caché further shortens development cycles (regardless of the PERSISTENCETYPE chosen) by automatically generating all the files needed to test and deploy each bean.

Conclusion

There is not a single "correct" way to provide data persistence to Java applications. Depending on the needs of the application and the available database technologies, Java developers may decide to model data as simple tabular structures or as complex data objects. For J2EE applications they may choose either container-managed or bean-managed persistence. Deciding how to implement persistence usually involves making trade-offs between database independence, application performance, and development time and effort.

Caché minimizes those trade-offs. Its Unified Data Architecture allows data to be accessed simultaneously as relational tables and as rich data objects, without mapping. Automated generation of Enterprise Java Beans with bean-managed persistence helps developers create high-performance J2EE applications without the need for a lot of tedious coding. Regardless of how Java developers approach the issue of implementing data persistence, Caché is designed to provide the highest possible performance for the least possible development effort.

InterSystems Corporation

World Headquarters
One Memorial Drive
Cambridge, MA 02142-1356
Tel: +1.617.621.0600
Fax: +1.617.494.1631

www.InterSystems.com

INTERSYSTEMS