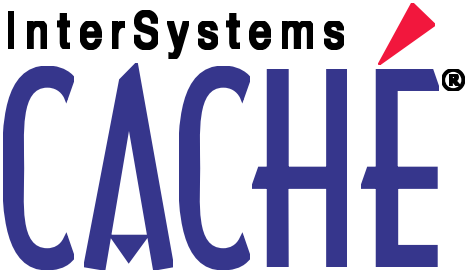


Using Caché Java with Third-party Persistence Frameworks

Version 2009.1
30 June 2009

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using Caché Java with Third-party Persistence Frameworks
Caché Version 2009.1 30 June 2009

This book was assembled and formatted in Adobe Page Description Format (PDF) using tools and information from the following sources: Sun Microsystems, RenderX, Inc., Adobe Systems, and the World Wide Web Consortium at www.w3c.org. The primary document development tools were special-purpose XML-processing applications built by InterSystems using Caché and Java.

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Customer Support
Tel: +1 617 621-0700
Fax: +1 617 374-9391
Email: support@InterSystems.com

Table of Contents

List of Figures

About This Book

This book is a guide to using Cache Java with persistence frameworks such as Enterprise Java Beans or Hibernate.

This book contains the following sections:

- Introduction
- Using Caché Java with Enterprise Java Beans
- Using the Caché Hibernate Dialect

There is also a detailed Table of Contents.

For general information, see *Using InterSystems Documentation*.

1

Introduction

This manual discusses two options for using standard Java persistence frameworks with Cache:

- *The Caché EJB Binding* — The Caché EJB binding lets J2EE Enterprise Java Beans communicate directly with objects on a Caché server. For every specified Caché class, the Caché EJB binding automatically generates an EJB Entity bean that can store and load itself within the Caché database, using an efficient, automatically generated, bean-managed persistence interface. The EJB binding automatically generates EJB "bookkeeping" (such as deployment descriptors) to reduce the amount of work involved in EJB programming.
- *The Caché Hibernate Dialect* — Hibernate is an open source utility from JBoss that generates the object-relational mapping needed to store Java objects in a relational database. Since every vendor's implementation of SQL is slightly different, Hibernate relies on vendor-provided "dialects" to customize its mappings to specific databases. The Caché dialect of Hibernate allows you to take advantage of this high performance, vendor-neutral persistence service in your Caché Java applications.

Caché also offers the *Jalapeño Persistence Library for Java* which provides a powerful, easy-to-use way to store and access Java objects without the need for object-relational mapping. Persistent objects are stored and accessed as true objects, preserving properties, relationships, and other object-oriented features. The object database schema can also be exported to a corresponding relational schema, allowing Jalapeño applications to store and access objects on a relational database.

2

The Caché EJB Binding

The Caché EJB binding allows you to use Caché with an Enterprise Java Bean (EJB) application. The Caché EJB binding offers the following advantages:

- The Caché EJB binding provides the performance of hand-written persistence code without the effort of writing and maintaining it. Caché automatically generates persistence code for EJB Entity beans using the metadata stored within the Caché class dictionary.
- The Caché database is defined in terms of objects, so there is no need for cumbersome object-relational mapping within the EJB server.
- Caché EJB offers sophisticated caching to eliminate unnecessary network traffic between the EJB server and the database server.
- Caché enables faster EJB application development by eliminating much of the associated drudge work. In addition to automatically generating EJB Entity beans, Caché generates EJB deployment descriptors, scripts for deploying your beans onto the EJB server, and Java code for testing your Entity beans.
- Caché lets your EJB application use both relational and object access to the database, allowing you choose whichever is the more appropriate technique for the task at hand.

The Caché EJB Server is described in more detail in the following sections. This chapter assumes that you are familiar with both Java and EJB.

2.1 How EJB Works

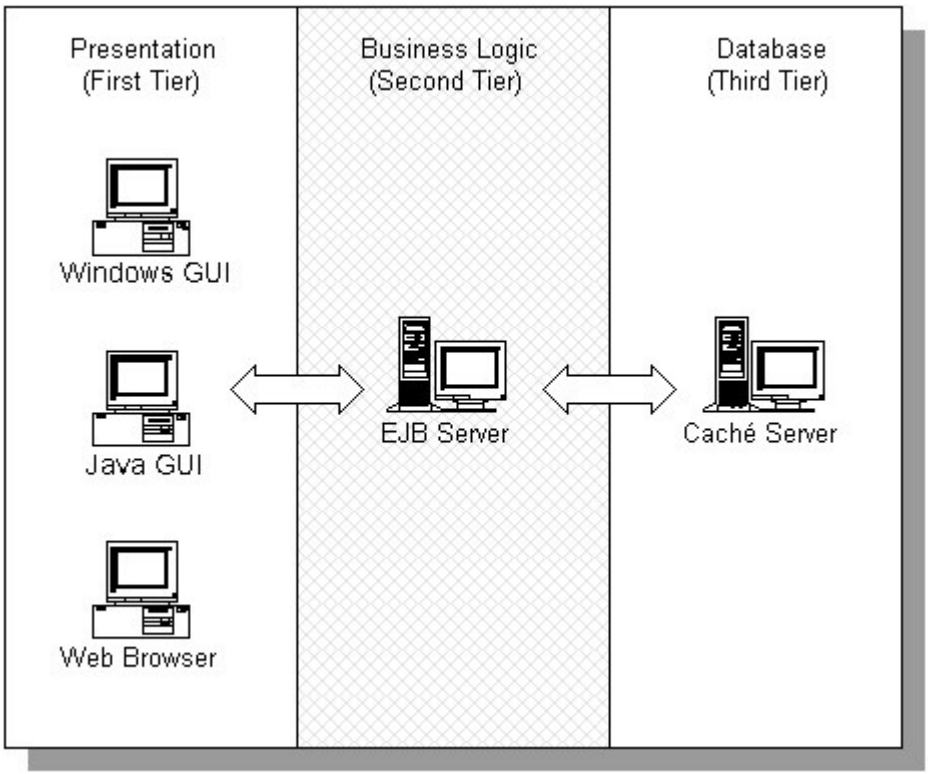
Enterprise Java Beans is a set of Java-based technologies that provide a framework for developing n-tier, distributed applications using Java components in both the presentation and business logic tiers.

EJB provides a standard framework (also referred to as a transaction monitor) for hosting components (pieces of Java code packaged as Enterprise Java Beans). This framework manages (among other things) all security, client connectivity, object lifecycles, and transaction control for the application.

2.1.1 EJB Application Architecture

The general architecture of an EJB application consists of three parts (or tiers), which can be distributed across as many machines as is appropriate:

EJB Tiers



The presentation tier is responsible for the user interface. The business tier is responsible for business logic. The database tier is responsible for all data storage and database operations.

The middle tier (business logic) of an EJB application is split into two distinct parts:

- *An EJB Server* — A Java implementation of a well-defined set of services for EJB applications. Client applications connect to the EJB Server using Java-based connectivity–typically Java RMI (Remote Method Invocation). There are several implementations of EJB server available on the market.

- *User-defined Beans* — A group of user-supplied components (beans) that either provide business logic or represent items contained in the application's database tier.

Bean Types and Persistence

Within EJB there are two types of user-defined beans or components:

- *Entity Beans* — Representations, as objects, of data stored within a database.
- *Session Beans* — Repositories for middle-tier business logic. Typically, a session bean interacts with one or more entity beans. As an alternative, many developers bypass entity beans and have their applications execute JDBC queries against the database from their session beans; this avoids the complexity and poor performance of using entity beans with relational databases.

For each type of bean, an EJB server provides a container object. A container object encapsulates all access to the entity and session beans and provides additional services (such as enforcing privileges or controlling when transactions begin and end).

EJB does not define what type of database is to be used by an application; it only defines the entity bean interface. Each entity bean knows how to persist (store and retrieve) its contents into a database. EJB defines two forms of entity bean persistence:

- *Container Managed Persistence (CMP)* — This is storage, typically to a relational database, that is automatically provided by the EJB Server's container objects. The advantage of CMP is that it keeps the developer from having to write specific persistence code for each entity bean. The downside to CMP is that it typically requires the definition of a complex object-relational mapping and that it offers relatively poor performance, as all object access is shoehorned into generic JDBC-based SQL queries.
- *Bean Managed Persistence (BMP)* — In this case, each entity bean is responsible for its own storage (by implementing ejbLoad and ejbStore methods). BMP can have much higher performance than CMP, as it is custom-written, but it has the downside of requiring code to be written for each bean.

The Caché EJB binding offers the best of both of these: Caché creates entity beans with Bean Managed Persistence (BMP) that is automatically generated. You get high-performance, object-based access to objects within the database with no object/relational mapping and no generic SQL queries. If you prefer, you can also use Caché with CMP via the Caché JDBC driver.

Supported EJB Servers

The generated code is for EJB 2.0. For a list of supported servers, see Supported Enterprise Java Beans Servers.

Finding Out More About EJB

There are many web sites and books available on Enterprise Java Beans including:

- http://java.sun.com/products/ejb/index.html

- *Enterprise Java Beans* by Richard Monson — Haefel, 2001, O'Reilly & Associates.

2.2 Caché EJB Binding Architecture

The Caché EJB binding is very similar to the Caché Java binding. The main difference is that Caché classes are generated as EJB Entity Beans instead of normal Java classes. In addition, the Caché EJB binding generates an additional set of classes and other files as required by EJB.

The Caché EJB binding consists of the following components:

- *The Caché Java Class Generator* — An extension to the Caché Class Compiler that generates EJB classes as well as other related files from classes defined in the Caché Class Dictionary.
- *The Caché Java Package* — A package of pure Java classes that works in conjunction with the Java classes generated by the Caché Java Class Generator and provides them with transparent connectivity to the objects stored in the Caché database. The majority of these classes are the same as those used with the Caché Java binding.
- *The Caché Object Server* — A high performance server process that manages communication between Java clients and a Caché database server using standard networking protocols (TCP/IP). This server is the same as is used for the Caché Java binding.

The Caché Class Compiler can automatically create EJB Entity beans for any classes contained within the Caché Class Dictionary. These generated beans communicate at runtime with their corresponding Caché class on a Caché server. The generated Java classes contain only pure Java code and are automatically synchronized with the master class definition (see Java Binding Architecture for a diagram of the synchronization process).

The basic mechanism works as follows:

- You define one or more classes within Caché. Typically, these are persistent objects that are stored within the Caché database.
- The EJB Wizard generates EJB entity beans that correspond to your Caché classes. These beans include accessor (get and set) methods for all of an object's properties.
- The EJB Wizard also generates additional helper Java classes (such as a Primary Key class), a deployment descriptor, and, optionally, scripts for deploying the beans and code for testing the bean.
- At run time, the EJB server connects to a Caché server. When it needs to create an instance of an entity bean, it invokes a generated method that loads its data from the Caché server. Similarly, there is a generated method for storing a modified bean back into the database.

The runtime architecture consists of the following:

- A Caché database server (or servers).
- An EJB Server on which the Enterprise Java beans run.
- A Java client application that communicates with the EJB Server.

At runtime, the EJB server connects to Caché as it would any standard JDBC data source. As Caché offers simultaneous JDBC and object access, the EJB server uses the standard JDBC interfaces for controlling connections and transactions with the Caché server.

2.3 Installation and Configuration

This section describes how to set up WebLogic to run EJB with Caché. It assumes that you have followed all the defaults when installing both Caché and WebLogic, and that Caché is already running on your system.

2.3.1 Installing and Configuring the EJB Server

To use EJB with Caché, begin by installing and configuring the EJB server (in this case, WebLogic):

- Install WebLogic. InterSystems recommends that you follow all the defaults during the installation process; this includes that you install WebLogic as a stand-alone application, not as a service.
- After installing WebLogic, perform the following edits on the file startWebLogic.cmd, which, by default, is located in the c:\bea\wlserver6.1\config\mydomain directory.
- Change the line that sets CLASSPATH environment variable from:

  ```
  set CLASSPATH=.;.\lib\weblogic_sp.jar;.\lib\weblogic.jar
  ```

  to:

  ```
  set CACHEPATH=c:\program files\cachesys\dev\java\lib\CacheDB.jar
  set CLASSPATH=.;.\lib\weblogic_sp.jar;.\lib\weblogic.jar;%CACHEPATH%
  ```

- In the lines regarding production and development mode, make sure that the STARTMODE variable is set to false, as in the code below:

  ```
  @rem Set Production Mode. When set to true, the server starts up
  @rem in production mode. When set to false, the server starts up
  @rem in development mode. If not set, it is defaulted to false.
  set STARTMODE=false
  ```

  This variable specifies that WebLogic starts in what it calls “developer mode.” In this mode, WebLogic is set up for automatic deployment of beans; in production mode, deployment requires manual intervention and that you provide a password.

2.3.2 Creating an EJB Projection

Once you have installed and configured the EJB server, the next step is to create an EJB projection for an existing class. When you compile this class, Caché generates content required for configuring the Caché JDBC driver.

- Begin by loading the class that you want to project, such as Sample.Person, in Caché Studio.
- Still in Studio, select `New Projection` from the `Add` submenu of the `Class` menu; this displays the New Class Projection Wizard.
- In the New Class Projection Wizard, specify the projection's name (for example, `PersonEJBProjection`).
- On the wizard's second screen, under `Projection Type`, click the `...` button and select `EJB` from the available list; this displays a list of EJB projection parameters. For this list of parameters, enter values for at least the following items:
  - `APPSERVERHOME` — The path to where WebLogic is installed, which is c:\bea\wlserver6.1 by default. This parameter must have a value specified for the class to compile.
  - `BEANNAME` — The name of the bean to be generated for the projected class. If more than one class is specified in `CLASSLIST`, each must have its own bean specified. As with `CLASSLIST`, the list must be comma-separated. If no value is specified for this parameter, Caché uses the string `"Package_Class_"` for bean-related names that include the package and simply `"Class"` for bean-related names that mention only the class. Hence, there is a Sample_Person_ directory containing bean information, including a EJBPerson.java file and a TestSample_Person_Client directory containing a TestSample_Person__Bean.java file.
  - `CLASSLIST` — The current class, in the form `Package.Class`, and, optionally, a comma-separated list of any other classes that it does not automatically include. It does automatically include superclasses and classes associated with object-valued properties. If no value is specified for this parameter, Caché uses the current class.
  - `JAVAHOME` — The location where Java is installed. This parameter must have a value specified for the class to compile. If you have installed WebLogic according to the defaults, this is C:\bea\jdk131.
  - `ROOTDIR` — The path and directory where Caché places generated EJB content.
- Once you have specified all the necessary parameters in the wizard, select `Finish`.
- Compile the class. This creates all files necessary to use the Caché class as an enterprise Java bean. See Testing a Bean for information on using these files. For configuration purposes, the most significant of these is the <BeanName>ConnectionPool.xml file, where <BeanName> is the value specified by the `BEANNAME` parameter above. This file is located in the directory specified above by `ROOTDIR`.

2.3.3 Configuring the Caché JDBC Driver

Once you have projected a class, you can use data from that projection to configure the Caché JDBC driver:

- While WebLogic is not running, copy the contents of <BeanName>ConnectionPool.xml and paste it into config.xml (located, by default, in c:\bea\wlserver6.1\config\mydomain\) immediately before the final `</Domain>` tag. The content that you paste will be of the form:

```
<JDBCConnectionPool
CapacityIncrement="2"
DriverName="com.intersys.jdbc.CacheDriver"
InitialCapacity="2"
MaxCapacity="255"
Name="cachePool<NS>"
Properties="user=_SYSTEM;password=sys;TCP_NODELAY=true"
RefreshMinutes="10"
Targets="myserver"
TestConnectionsOnReserve="false"
TestTableName="ISC.TestTable"
URL="jdbc:Cache://127.0.0.1:1972/<NS>/ejbjdbc.log"/>

<JDBCTxDataSource JNDIName="weblogic.jdbc.jts.cachePool<NS>"
Name="cachePool<NS>" PoolName="cachePool<NS>" Targets="myserver"/>
```

  where each instance of `<NS>` is the Namespace containing the projected class. The content of these elements is all in standard WebLogic form. The only Caché-specific information is the content of the `"JDBCConnectionPool"` element's URL attribute. For details on these elements and their attributes, consult the WebLogic documentation.

  config.xml may include multiple sets of entries — such as for multiple namespaces or servers. If there are multiple entries, make sure that there are no conflicting listings, such as two for the same namespace (specified by the URL attribute of `"JDBCConnectionPool"`) that have different passwords or test parameters).

- Next, you can confirm that WebLogic is properly configured to work with Caché. To do this, look at the **[Home] > [Processes]** page of the System Management Portal, and note how many processes are running.
- Finally, start WebLogic by selecting the `Start Default Server` choice from the appropriate Bea submenu from the `Programs` menu. Once the server starts running, check how many processes are running. The number should have increased by the value specified by the `InitialCapacity` attribute of the `"JDBCConnectionPool"` attribute (second line in the example above). You can then shut down the WebLogic server if you want to.

At this point, you have configured WebLogic so that it can connect to Caché. You can then use this functioning connection to run one of the automatically generated bean test programs described in the next section.

2.4 Testing a Bean

In the process of configuring WebLogic, you have created the beans for a Caché. When you compile the class with the EJB projection, Caché creates a set of beans and bean-related files in the location specified by ROOT_DIR and using the bean name specified by the BEANNAME parameter. For example, for a class called MyClass with a bean called MyBean, these are:

- MyBean — A directory containing bean-related files.
- MyBeanObj — A second directory containing bean-related files.
- testMyBeanClient — A command-line sample application using the bean (for the Windows command prompt, not the Caché Terminal).
- testMyBeanServlet — A Web-based sample application using the bean.
- runtestclient.bat — A batch file for invoking testMyClassServlet (below).
- MyBeanConnectionPool.xml — A file containing generic EJB connection information (not class-specific) and already used previously.
- deployear.bat — A batch file that creates the set of Web service EJB JAR (Java archive) and EAR (Enterprise Archive) files, so that you can run applications using the newly generated beans (including the test applications)

You can test the beans using either of the sample applications provided. The procedure for this is:

- Run deployear.bat. Since WebLogic is running in developer mode, invoking this file performs the following actions uninterruptedly and without requiring any password. deployear.bat compiles the already-created beans into runnable classes and creates more files needed for compiling the bean (MyBeanDeploy.jar and MyBean.jar).
- Start the WebLogic server if it is not already running. The server indicates that it is up and running by displaying a notice such as:

```
   <Notice> <WebLogicServer>
   <Started WebLogic Admin Server "myserver" for domain "mydomain"
running in Development Mode>.
```

You can then invoke either of the test programs. Your choices are:

- Invoke the command-line test program located in the Test<BeanName>Client directory by running runtestclient.bat at the Windows command prompt.
- Invoke the Web-based test program located in the Test<BeanName>Servlet directory by loading the test page in your Web browser:

  http://localhost:7001/Test<BeanName>Servlet/Test<BeanName>.html

3

The Caché Hibernate Dialect

Hibernate is an open source utility from JBoss that generates the object-relational mapping needed to store Java objects in a relational database. Since every vendor's implementation of SQL is slightly different, Hibernate relies on vendor-provided "dialects" to customize its mappings to specific databases. The Caché dialect of Hibernate allows you to take advantage of this high performance, vendor-neutral persistence service in your Caché Java applications.

Why Caché Hibernate?

For many applications, it would be pointless to add an external persistence service to Caché. Although Caché can expose data as relational tables, it also allows object access to data. Caché can automatically project its classes to Java proxy classes, providing object oriented data persistence without a mapping layer. Caché classes can also be projected as EJB for high performance bean-managed persistence.

Object persistence becomes a problem when Caché needs a simple way to exchange objects with a relational database. How can relational data be accessed and stored without giving up the advantages provided by Caché object orientation? Hibernate allows your Java application to exchange object-oriented data transparently between Caché and one or more relational databases. In a multi-database environment, Hibernate offers two major advantages:

- *Object/relational mapping* — Hibernate is essentially a wrapper around JDBC that provides object/relational mapping (ORM) services for relational databases. Although Caché JDBC itself can be used to persist objects in relational databases, this approach quickly becomes unmanageable when dealing with more than a few simple objects. Hibernate automates the process of converting between object and relational formats, relieving developers of a serious maintenance burden. For most applications, Hibernate is also significantly simpler and more flexible than EJB and similar persistence services.
- *Hibernate Query Language* — Hibernate Query Language (HQL) is a vendor-neutral, object oriented query language based on SQL. Your application can use a single query language to access both Caché and the relational databases. This reduces the potential for problems caused by variations

between SQL dialects. HQL also provides object oriented features that make it easier to query object data in a natural manner.

If your Caché Java application needs to work with relational databases, Hibernate could make development and maintenance considerably simpler.

Further reading

Hibernate comes with extensive electronic documentation. In addition, we recommend *Java Persistence with Hibernate* from Manning Press for both introductory material and in-depth discussion. This is a greatly expanded second edition of the book formerly titled *Hibernate in Action*. It describes Hibernate 3.2.1 (the first version with Caché support), and provides important new information on how Hibernate implements the Java Persistence API.

3.1 Installation and Configuration

This section provides instructions for setting up your system to use Hibernate with Caché. The instructions assume that the correct versions of both Caché and Hibernate are installed and operational.

Requirements

This document assumes that the following software is installed on your system:

- Caché 2007.1 or higher
- Hibernate 3.2.1GA or higher (any version that contains the Caché dialect extensions)
- Java JDK 1.3 or higher. InterSystems recommends using JDK 1.4 or higher. JDK 1.5 or higher is required if Hibernate annotations are used.

Directories

These instructions refer to the following directories:

- <cachesys> — the Caché installation directory (for the location of <cachesys> on your system, see Default Caché Installation Directory in the *Caché Installation Guide*).
- <hibernate_root> — the Hibernate installation directory (C:\hibernate-3.2 by default).

3.1.1 System Settings

Make the following changes to your system:

- *CacheDB.jar File*

  The CacheDB.jar file contains the Caché JDBC driver. If you haven't already done so, copy CacheDB.jar from

<cachesys>\Dev\java\lib\JDK15\CacheDB.jar

to

<hibernate_root>\lib\CacheDB.jar.

- *Java Classpath*

  Make sure the following items are on your Java classpath:

  - The jar files from <hibernate_root>\lib
  - The directory or directories where the Hibernate configuration files (hibernate.properties and hibernate.cfg.xml) are kept. By default, both files are in:

    <hibernate_root>\etc

3.1.2 Hibernate Configuration

In the Hibernate configuration files (either hibernate.properties or hibernate.cfg.xml), specify the connection information for your database, and the Caché dialect that Hibernate will use.

- *dialect* — The fully qualified name of the dialect class. The base dialect class for the Caché dialect is:

  ```
  org.hibernate.dialect.Cache71Dialect
  ```

  You can use a custom dialect class derived from this base class if you need to enable support for the Hibernate primary key generator classes (see Support for Sequences).

- *driver_class* — The fully qualified name of the Caché JDBC driver:

  ```
  com.intersys.jdbc.CacheDriver
  ```

  The JDBC driver is contained in the CacheDB.jar file (see System Settings for details).

- *username* — User name for the Caché namespace you want to access (default is _SYSTEM).
- *password* — Password for the Caché namespace (default is SYS).
- *url* — The URL for the Caché JDBC driver. The format for the URL is:

  ```
  jdbc:Cache://<host>:<port>/<namespace>
  ```

  where <host> is the host address of the machine hosting Caché, <port> is the super server port of your Caché instance, and <namespace> is the namespace that contains your Caché database data.

A typical entry in hibernate.properties would contain the following lines:

```
hibernate.dialect   org.hibernate.dialect.Cache71Dialect
hibernate.connection.driver_class   com.intersys.jdbc.CacheDriver
hibernate.connection.username   _SYSTEM
hibernate.connection.password   SYS
hibernate.connection.url   jdbc:Cache://127.0.0.1:1972/USER
```

The following example shows the same information as it would appear in hibernate.cfg.xml:

```
<hibernate-configuration>
  <session-factory>
    <property name="dialect">
      org.hibernate.dialect.Cache71Dialect
    </property>
    <property name="connection.driver_class">
      com.intersys.jdbc.CacheDriver</property>
    <property name="connection.username">_SYSTEM</property>
    <property name="connection.password">SYS</property>
    <property name="connection.url">
      jdbc:Cache://127.0.0.1:1972/USER
    </property>
  </session-factory>
</hibernate-configuration>
```

CAUTION: If the same property is set in both hibernate.properties and hibernate.cfg.xml, Hibernate will use the value from hibernate.cfg.xml.

3.1.3 Using Hibernate with SQL

The following SQL features are not supported by the Caché Hibernate dialect:

- The `longvarbinary` and `longvarchar` types are not supported in a `WHERE` clause.
- Row valued expressions such as `WHERE (A,B,C)=(1,2,3)` are not supported.

3.1.4 Support for Sequences

The base class for Caché dialects is Cache71Dialect in package org.hibernate.dialect. You can extend this class to enable support for the optional Hibernate primary key generator methods. This support is required if you want to enable Oracle-style sequences or use other non-standard generation methods.

Adding the Extended Class

Create the following class:

```
public Cache71SequenceDialect extends Cache71Dialect {
   public boolean supportsSequences() {
      return true;
   }
}
```

Specify the fully qualified name of your new class in the dialect property of your hibernate.properties or hibernate.cfg.xml file (see the examples in Hibernate Configuration). For example, if your package was named mySeqDialect, the entry in hibernate.properties would be:

```
hibernate.dialect  mySeqDialect.Cache71SequenceDialect
```

In hibernate.cfg.xml, the entry would be:

```
<property name="dialect">
   mySeqDialect.Cache71SequenceDialect
</property>
```

Installing and Using Sequence Dialect Support

To install Hibernate sequence dialect support, load the CacheSequences project into the namespace that your application will access:

1. In Caché Studio, change to the namespace your application will access.
2. From the Caché Studio menu, select `Tools > Import Local...`
3. In the file dialog, open the CacheSequences.xml project file:

   <hibernate_root>/etc/CacheSequences.xml.

   Studio will load and compile the Caché InterSystems.Sequences class.

The sequence generator to be used is specified in the `id` property of your Hibernate mapping. Oracle-style sequences can be generated by specifying `sequence` or `seqhilo`. If `native` is specified, the `identity` generator will be used when Caché is the underlying database. For more information on these classes, see the reference documentation that comes with Hibernate (specifically, the id section in the *Basic O/R Mapping* chapter).

The following example demonstrates the use of a sequence generator in a Hibernate mapping:

```
<id name="id" column="uid" type="long" unsaved-value="null">
   <generator class="seqhilo">
      <param name="sequence">EVENTS_SEQ</param>
      <param name="max_lo">0</param>
   </generator>
</id>
```

The sequence classes maintain a table of counters that can be incremented by calling the stored procedure just before performing the insert. For example:

```
call InterSystems.Sequences_GetNext("Name")
// perform the insert here
```

You can also increment the table with a standard SQL statement. For example:

```
SELECT InterSystems.Sequences_GetNext(sequencename)
   FROM InterSystems.Sequences
   WHERE Name='sequencename'
```

The table can be queried as table `InterSystems.Sequences`. The data is actually stored in global `^InterSystems.Sequences`. You can make the sequences system-wide by mapping `^InterSystems.Sequences*` to a location that is common to all your applications.