Cayenne Guide

Version 3.1 (3.1.3)

Table of Contents

Chapter 1. Object Relational Mapping with Cayenne

1.1. Setup

System Requirements

- Java: Cayenne runtime framework and CayenneModeler GUI tool are written in 100% Java, and run on any Java-compatible platform. Required JDK version is 1.5 or higher. The last version of Cayenne compatible with JDK 1.4 is 1.2.x/2.0.x and JDK 1.3 is 1.1.x
- JDBC Driver: An appropriate DB-specific JDBC driver is needed to access the database. It can be included in the application or used in web container DataSource configuration.
- Third-party Libraries: Cayenne runtime framework has a minimal set of required and a few more optional dependencies on third-party open source packages. See "Including Cayenne in a Project" chapter for details.

Running CayenneModeler

CayenneModeler GUI tool is intended to work with object relational mapping projects. While you can edit your XML by hand, it is rarely needed, as the Modeler is a pretty advanced tool included in Cayenne distribution. To obtain CayenneModeler, download Cayenne distribution archive from http://cayenne.apache.org/download.html matching the OS you are using. Of course Java needs to be installed on the machine where you are going to run the Modeler.

- OS X distribution contains CayenneModeler.app at the root of the distribution disk image.
- Windows distribution contains CayenneModeler.exe file in the bin directory.
- Cross-platform distribution (targeting Linux, but as the name implies, compatible with any OS) contains a runnable CayenneModeler.jar in the bin directory. It can be executed either by double-clicking, or if the environment is not configured to execute jars, by running from command-line:

```
$ java -jar CayenneModeler.jar
```

The Modeler can also be started from Maven. While it may look like an exotic way to start a GUI application, it has its benefits - no need to download Cayenne distribution, the version of the Modeler always matches the version of the framework, the plugin can find mapping files in the project automatically. So it is an attractive option to some developers. Maven option requires a declaration in the POM:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.cayenne.plugins</groupId>
            <artifactId>maven-cayenne-modeler-plugin</artifactId>
            <version>3.1.3</version>
        </plugin>
    </plugins>
</build>
```

And then can be run as

```
$ mvn cayenne-modeler:run
```

1.2. Cayenne Mapping Structure

Cayenne Project

A Cayenne project is an XML representation of a model connecting database schema with Java classes. A project is normally created and manipulated via CayenneModeler GUI and then used to initialize Cayenne runtime. A project is made of one or more files. There's always a root project descriptor file in any valid project. It is normally called `cayenne-xyz.xml`, where "xyz" is the name of the project.

Project descriptor can reference DataMap files, one per DataMap. DataMap files are normally called `xyz.map.xml`, where "xyz" is the name of the DataMap. For legacy reasons this naming convention is different from the convention for the root project descriptor above, and we may align it in the future versions. Here is how a typical project might look on the file system:

```
~: ls -l
total 24
-rw-r--r--  1 cayenne  staff  491 Jan 28 18:25 cayenne-project.xml
-rw-r--r--  1 cayenne  staff  313 Jan 28 18:25 datamap.map.xml
```

DataMap are referenced by name in the root descriptor:

```
<map name="datamap"/>
```

Map files are resolved by Cayenne by appending ".map.xml" extension to the map name, and resolving the resulting string relative to the root descriptor URI. The following sections discuss varios ORM model objects, without regards to their XML representation. XML format details are really unimportant to the Cayenne users.

DataMap

DataMap is a container of persistent entities and other object-relational metadata. DataMap provides developers with a scope to organize their entities, but it does not provide a namespace for entities. In fact all DataMaps present in runtime are combined in a single namespace. Each DataMap must be associated with a DataNode. This is how Cayenne knows which database to use when running a query.

DataNode

DataNode is model of a database. It is actually pretty simple. It has an arbitrary user-provided name and information needed to create or locate a JDBC DataSource. Most projects only have one DataNode, though there may be any number of nodes if needed.

DbEntity

DbEntity is a model of a single DB table or view. DbEntity is made of DbAttributes that correspond to columns, and DbRelationships that map PK/FK pairs. DbRelationships are not strictly tied to FK constraints in DB, and should be mapped for all logical "relationships" between the tables.

ObjEntity

ObjEntity is a model of a single persistent Java class. ObjEntity is made of ObjAttributes and ObjRelationships. Both correspond to entity class properties. However ObjAttributes represent "simple" properties (normally things like String, numbers, dates, etc.), while ObjRelationships correspond to properties that have a type of another entity.

ObjEntity maps to one or more DbEntities. There’s always one "root" DbEntity for each ObjEntity. ObjAttribiute maps to a DbAttribute or an Embeddable. Most often mapped DbAttribute is from the root DbEntity. Sometimes mapping is done to a DbAttribute from another DbEntity somehow related to the root DbEntity. Such ObjAttribute is called "flattened". Similarly ObjRelationship maps either to a single DbRelationship, or to a chain of DbRelationships ("flattened" ObjRelationship).

ObjEntities may also contain mapping of their lifecycle callback methods.

Embeddable

Embeddable is a model of a Java class that acts as a single attribute of an ObjEntity, but maps to multiple columns in the database.

Procedure

A model of a stored procedure in the database.

Query

A model of a query. Cayenne allows queries to be mapped in Cayenne project, or created in the code. Depending on the circumstances the users may take one or the other approach.

1.3. CayenneModeler Application

Working with Mapping Projects

Reverse Engineering Database

Generating Database Schema

Migrations

Generating Java Classes

Modeling Inheritance

Modeling Generic Persistent Classes

Normally each ObjEntity is mapped to a specific Java class (such as Artist or Painting) that explicitly declare all entity properties as pairs of getters and setters. However Cayenne allows to map a completly generic class to any number of entities. The only expectation is that a generic class implements org.apache.cayenne.DataObject. So an ideal candidate for a generic class is CayenneDataObject, or some custom subclass of CayenneDataObject.

If you don't enter anything for Java Class of an ObjEntity, Cayenne assumes generic mapping and uses the following implicit rules to determine a class of a generic object. If DataMap "Custom Superclass" is set, runtime uses this class to instantiate new objects. If not, `org.apache.cayenne.CayenneDataObject` is used.

Class generation procedures (either done in the Modeler or with Ant or Maven) would skip entities that are mapped to CayenneDataObject explicitly or have no class mapping.

Mapping ObjAttributes to Custom Classes

Modeling Primary Key Generation Strategy

Chapter 2. Cayenne Framework

2.1. Including Cayenne in a Project

Jar Files and Dependencies

Cayenne distribution contains the following core runtime jars in the distribution lib directory:

- `cayenne-server-x.x.jar` - contains full Cayenne runtime (DI, adapters, DB access classes, etc.). Most applications will use only this file.
- `cayenne-client-x.x.jar` - a subset of cayenne-server.jar trimmed for use on the client in an ROP application.
- Other `cayenne-* jars` - various Cayenne extensions.

When using `cayenne-server-x.x.jar` you'll need a few third party jars (all included in `lib/third-party` directory of the distribution):

- Apache Velocity Template Engine, version 1.6.x (and all its dependencies bundled with velocity-dep)
- Apache Commons Collections, version 3.2.1
- Apache Commons Logging, version 1.1

Cayenne integrates with various caching, clustering and other frameworks. These optional integrations will require other third-party jars that the users will need to obtain on their own.

Maven Projects

If you are using Maven, you won't have to deal with figuring out the dependencies. You can simply include cayenne-server artifact in your POM:

```
<dependency>
   <groupId>org.apache.cayenne</groupId>
   <artifactId>cayenne-server</artifactId>
   <version>3.1.3</version>
</dependency>
```

Additionally Cayenne provides a Maven plugin with a set of goals to perform various project tasks, such as synching generated Java classes with the mapping, described in the following subsection. The full plugin name is `org.apache.cayenne.plugins:cayenne-maven-plugin`.

cgen

`cgen` is a `cayenne-maven-plugin` goal that generates and maintains source (.java) files of persistent objects based on a DataMap. By default, it is bound to the generate-sources phase. If "makePairs" is set to "true" (which is the recommended default), this task will generate a pair of classes (superclass/subclass) for each ObjEntity in the DataMap. Superclasses should not be changed

manually, since they are always overwritten. Subclasses are never overwritten and may be later customized by the user. If "makePairs" is set to "false", a single class will be generated for each ObjEntity.

By creating custom templates, you can use cgen to generate other output (such as web pages, reports, specialized code templates) based on DataMap information.

Table 1. cgen required parameters

Name	Type	Description
map	File	DataMap XML file which serves as a source of metadata for class generation. E.g. `${project.basedir}/src/main/resources/my.map.xml`
destDir	File	Root destination directory for Java classes (ignoring their package names).

Table 2. cgen optional parameters

Name	Type	Description
additionalMaps	File	A directory that contains additional DataMap XML files that may be needed to resolve cross-DataMap relationships for the the main DataMap, for which class generation occurs.
client	boolean	Whether we are generating classes for the client tier in a Remote Object Persistence application. "False" by default.
embeddableTemplate	String	Location of a custom Velocity template file for Embeddable class generation. If omitted, default template is used.
embeddableSuperTemplate	String	Location of a custom Velocity template file for Embeddable superclass generation. Ignored unless "makepairs" set to "true". If omitted, default template is used.
encoding	String	Generated files encoding if different from the default on current platform. Target encoding must be supported by the JVM running the build. Standard encodings supported by Java on all platforms are US-ASCII, ISO-8859-1, UTF-8, UTF-16BE, UTF-16LE, UTF-16. See javadocs for java.nio.charset.Charset for more information.
excludeEntities	String	A comma-separated list of ObjEntity patterns (expressed as a perl5 regex) to exclude from template generation. By default none of the DataMap entities are excluded.
includeEntities	String	A comma-separated list of ObjEntity patterns (expressed as a perl5 regex) to include from template generation. By default all DataMap entities are included.
makePairs	boolean	If "true" (a recommended default), will generate subclass/superclass pairs, with all generated code placed in superclass.

Name	Type	Description
mode	String	Specifies class generator iteration target. There are three possible values: "entity" (default), "datamap", "all". "entity" performs one generator iteration for each included ObjEntity, applying either standard to custom entity templates. "datamap" performs a single iteration, applying DataMap templates. "All" is a combination of entity and datamap.
overwrite	boolean	Only has effect when "makePairs" is set to "false". If "overwrite" is "true", will overwrite older versions of generated classes.
superPkg	String	Java package name of all generated superclasses. If omitted, each superclass will be placed in the subpackage of its subclass called "auto". Doesn't have any effect if either "makepairs" or "usePkgPath" are false (both are true by default).
superTemplate	String	Location of a custom Velocity template file for ObjEntity superclass generation. Only has effect if "makepairs" set to "true". If omitted, default template is used.
template	String	Location of a custom Velocity template file for ObjEntity class generation. If omitted, default template is used.
usePkgPath	boolean	If set to "true" (default), a directory tree will be generated in "destDir" corresponding to the class package structure, if set to "false", classes will be generated in "destDir" ignoring their package.

Example - a typical class generation scenario, where pairs of classes are generated with default Maven source destination and superclass package:

```
<plugin>
    <groupId>org.apache.cayenne.plugins</groupId>
    <artifactId>cayenne-maven-plugin</artifactId>
    <version>3.1.3</version>

    <!--
        There's an intermittent problem when using Maven/cgen in Eclipse with
m2eclipse plugin that
        requires placing "configuration" section at the plugin level, instead of
execution
        level.
    -->
    <configuration>
            <map>${project.basedir}/src/main/resources/my.map.xml</map>
            <destDir>${project.basedir}/src/main/java</destDir>
            <superPkg>org.example.model.auto</superPkg>
    </configuration>

    <executions>
        <execution>
            <goals>
                <goal>cgen</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

cdbgen

`cdbgen` is a `cayenne-maven-plugin` goal that drops and/or generates tables in a database on Cayenne DataMap. By default, it is bound to the pre-integration-test phase.

Table 3. cdbgen required parameters

Name	Type	Description
map	File	DataMap XML file which serves as a source of metadata for class generation. E.g. `${project.basedir}/src/main/resources/my.map.xml`
driver	String	A class of JDBC driver to use for the target database.
url	String	JDBC connection URL of a target database.

Table 4. cdbgen optional parameters

Name	Type	Description
adapter	String	Java class name implementing org.apache.cayenne.dba.DbAdapter. While this attribute is optional (a generic JdbcAdapter is used if not set), it is highly recommended to specify correct target adapter.
createFK	boolean	Indicates whether cdbgen should create foreign key constraints. Default is "true".
createPK	boolean	Indicates whether cdbgen should create Cayenne-specific auto PK objects. Default is "true".
createTables	boolean	Indicates whether cdbgen should create new tables. Default is "true".
dropPK	boolean	Indicates whether cdbgen should drop Cayenne primary key support objects. Default is "false".
dropTables	boolean	Indicates whether cdbgen should drop the tables before attempting to create new ones. Default is "false".
password	String	Database user password.
username	String	Database user name.

Example - creating a DB schema on a local HSQLDB database:

```
<plugin>
    <groupId>org.apache.cayenne.plugins</groupId>
    <artifactId>maven-cayenne-plugin</artifactId>
    <version>3.1.3</version>

    <executions>
        <execution>
            <configuration>
                <map>${project.basedir}/src/main/resources/my.map.xml</map>
                <url>jdbc:hsqldb:hsql://localhost/testdb</url>
                <adapter>org.apache.cayenne.dba.hsqldb.HSQLDBAdapter</adapter>
                <driver>org.hsqldb.jdbcDriver</driver>
                <username>sa</username>
            </configuration>
            <goals>
                <goal>cdbgen</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

cdbimport

`cdbimport` is a `maven-cayenne-plugin` goal that generates a DataMap based on an existing database schema. By default, it is bound to the generate-sources phase. This allows you to generate your DataMap prior to building your project, which may be necessary if you are also using the cgen task.

Table 5. cdbimport parameters

Name	Type	Required	Description
map	File	Yes	DataMap XML file which is the destination of the schema import. Can be an existing file. If this file does not exist, it is created when cdbimport is executed. E.g. `${project.basedir}/src/main/resources/my.map.xml`. If "overwrite" is true (the default), an existing DataMap will be used as a template for the new imported DataMap, i.e. all its entities will be cleared and recreated, but its common settings, such as default Java package, will be preserved (unless changed explicitly in the plugin configuration).
driver	String	Yes	A class of JDBC driver to use for the target database.
url	String	Yes	JDBC connection URL of a target database.
adapter	String	No	A Java class name implementing org.apache.cayenne.dba.DbAdapter. This attribute is optional. If not specified, AutoAdapter is used, which will attempt to guess the DB type.
importProcedures	boolean	No	Indicates whether stored procedures should be imported from the database. Default is false.
meaningfulPk	boolean	No	Indicates whether primary keys should be mapped as attributes of the ObjEntity. Default is false.
namingStrategy	String	No	The naming strategy used for mapping database names to object entity names. Default is `org.apache.cayenne.map.naming.SmartNamingStrategy`.
overwriteExisting	boolean	No	Indicates whether existing DB and object entities should be overwritten. This is an all-or-nothing setting. If you need finer granularity, use the CayenneModeler. Default is "true".
password	String	No	Database user password.
procedurePattern	String	No	Pattern to match stored procedure names against for import. Default is to match all stored procedures. This value is only meaningful if importProcedures is true.
schemaName	String	No	Database schema to import tables/stored procedures from.
tablePattern	String	No	Pattern to match table names against for import. Default is to match all tables.
username	String	No	Database user name.

Example - loading a DB schema from a local HSQLDB database (essentially a reverse operation

compared to the cdbgen example above) :

```
<plugin>
    <groupId>org.apache.cayenne.plugins</groupId>
    <artifactId>cayenne-maven-plugin</artifactId>
    <version>3.1.3</version>

    <executions>
        <execution>
            <configuration>
                <map>${project.basedir}/src/main/resources/my.map.xml</map>
                <url>jdbc:mysql://127.0.0.1/mydb</url>
                <adapter>org.apache.cayenne.dba.hsqldb.HSQLDBAdapter</adapter>
                <driver>com.mysql.jdbc.Driver</driver>
                <username>sa</username>
            </configuration>
            <goals>
                <goal>cdbimport</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Ant Projects

cgen

cdbgen

cdbimport

This is an Ant counterpart of "cdbimport" goal of cayenne-maven-plugin described above. It has exactly the same properties. Here is a usage example:

```
 <cdbimport map="${context.dir}/WEB-INF/my.map.xml"
     driver="com.mysql.jdbc.Driver"
     url="jdbc:mysql://127.0.0.1/mydb"
     username="sa"/>
```

cdataport

2.2. Starting Cayenne

Starting and Stopping ServerRuntime

In runtime Cayenne is accessed via `org.apache.cayenne.configuration.server.ServerRuntime`. ServerRuntime is created by calling a convenient builder:

```
ServerRuntime runtime = new ServerRuntime("com/example/cayenne-project.xml");
```

The parameter you pass to the builder is a location of the main project file. Location is a '/'-separated path (same path separator is used on UNIX and Windows) that is resolved relative to the application classpath. The project file can be placed in the root package or in a subpackage (e.g. in the code above it is in "com/example" subpackage).

ServerRuntime encapsulates a single Cayenne stack. Most applications will just have one ServerRuntime using it to create as many ObjectContexts as needed, access the Dependency Injection (DI) container and work with other Cayenne features. Internally ServerRuntime is just a thin wrapper around the DI container. Detailed features of the container are discussed in "Customizing Cayenne Runtime" chapter. Here we'll just show an example of how an application might turn on external transactions:

```
public class MyExtensionsModule implements Module {
    public void configure(Binder binder) {
        binder.bind(QueryCache.class).to(EhCacheQueryCache.class);
    }
}
```

```
Module extensions = new MyExtensionsModule();
ServerRuntime runtime = new ServerRuntime("com/example/cayenne-project.xml",
extensions);
```

It is a good idea to shut down the runtime when it is no longer needed, usually before the application itself is shutdown:

```
runtime.shutdown();
```

When a runtime object has the same scope as the application, this may not be always necessary, however in some cases it is essential, and is generally considered a good practice. E.g. in a web container hot redeploy of a webapp will cause resource leaks and eventual OutOfMemoryError if the application fails to shutdown CayenneRuntime.

Merging Multiple Projects

ServerRuntime requires at least one mapping project to run. But it can also take multiple projects and merge them together in a single configuration. This way different parts of a database can be mapped independenlty from each other (even by different software providers), and combined in runtime when assembling an application. Doing it is as easy as passing multiple project locations to ServerRuntime constructor:

```
ServerRuntime runtime = new ServerRuntime(new String[] {
        "com/example/cayenne-project.xml",
        "org/foo/cayenne-library1.xml",
        "org/foo/cayenne-library2.xml"
    }
);
```

When the projects are merged, the following rules are applied:

- The order of projects matters during merge. If there are two conflicting metadata objects belonging to two projects, an object from the last project takes precedence over the object from the first one. This makes possible to override pieces of metadata. This is also similar to how DI modules are merged in Cayenne.
- Runtime DataDomain name is set to the name of the last project in the list.
- Runtime DataDomain properties are the same as the properties of the last project in the list. I.e. properties are not merged to avoid invalid combinations and unexpected runtime behavior.
- If there are two or more DataMaps with the same name, only one DataMap is used in the merged project, the rest are discarded. Same precedence rules apply - DataMap from the project with the highest index in the project list overrides all other DataMaps with the same name.
- If there are two or more DataNodes with the same name, only one DataNode is used in the merged project, the rest are discarded. DataNode coming from project with the highest index in the project list is chosen per precedence rule above.
- There is a notion of "default" DataNode. After the merge if any DataMaps are not explicitly linked to DataNodes, their queries will be executed via a default DataNode. This makes it possible to build mapping "libraries" that are only associated with a specific database in runtime. If there's only one DataNode in the merged project, it will be automatically chosen as default. A possible way to explicitly designate a specific node as default is to override `DataDomainProvider.createAndInitDataDomain()`.

Web Applications

Web applications can use a variety of mechanisms to configure and start the "services" they need, Cayenne being one of such services. Configuration can be done within standard Servlet specification objects like Servlets, Filters, or ServletContextListeners, or can use Spring, JEE CDI, etc. This is a user's architectural choice and Cayenne is agnostic to it and will happily work in any environment. As described above, all that is needed is to create an instance of ServerRuntime somewhere and provide the application code with means to access it. And shut it down when the application ends to avoid container leaks.

Still Cayenne includes a piece of web app configuration code that can assist in quickly setting up simple Cayenne-enabled web applications. We are talking about CayenneFilter. It is declared in web.xml:

```
<web-app>
    ...
    <filter>
        <filter-name>cayenne-project</filter-name>
        <filter-class>org.apache.cayenne.configuration.web.CayenneFilter</filter-
class>
    </filter>
     <filter-mapping>
        <filter-name>cayenne-project</filter-name>
        <url-pattern>/*</url-pattern>
     </filter-mapping>
    ...
 </web-app>
```

When started by the web container, it creates a instance of ServerRuntime and stores it in the ServletContext. Note that the name of Cayenne XML project file is derived from the "filter-name". In the example above CayenneFilter will look for an XML file "cayenne-project.xml". This can be overridden with "configuration-location" init parameter.

When the application runs, all HTTP requests matching the filter url-pattern will have access to a session-scoped ObjectContext like this:

```
ObjectContext context = BaseContext.getThreadObjectContext();
```

Of course the ObjectContext scope, and other behavior of the Cayenne runtime can be customized via dependency injection. For this another filter init parameter called "extra-modules" is used. "extra-modules" is a comma or space-separated list of class names, with each class implementing Module interface. These optional custom modules are loaded after the the standard ones, which allows users to override all standard definitions.

For those interested in the DI container contents of the runtime created by CayenneFilter, it is the same ServerRuntime as would've been created by other means, but with an extra `org.apache.cayenne.configuration.web.WebModule` module that provides `org.apache.cayenne.configuration.web.RequestHandler` service. This is the service to override in the custom modules if you need to provide a different ObjectContext scope, etc.

You should not think of CayenneFilter as the only way to start and use Cayenne in a web application. In fact CayenneFilter is entirely optional. Use it if you don't have any special design for application service management. If you do, simply integrate Cayenne into that design.

2.3. Persistent Objects and ObjectContext

ObjectContext

ObjectContext is an interface that users normally work with to access the database. It provides the

API to execute database operations and to manage persistent objects. A context is obtained from the ServerRuntime:

```
ObjectContext context = runtime.getContext();
```

The call above creates a new instance of ObjectContext that can access the database via this runtime. ObjectContext is a single "work area" in Cayenne, storing persistent objects. ObjectContext guarantees that for each database row with a unique ID it will contain at most one instance of an object, thus ensuring object graph consistency between multiple selects (a feature called "uniquing"). At the same time different ObjectContexts will have independent copies of objects for each unique database row. This allows users to isolate object changes from one another by using separate ObjectContexts.

These properties directly affect the strategies for scoping and sharing (or not sharing) ObjectContexts. Contexts that are only used to fetch objects from the database and whose objects are never modified by the application can be shared between mutliple users (and multiple threads). Contexts that store modified objects should be accessed only by a single user (e.g. a web application user might reuse a context instance between multiple web requests in the same HttpSession, thus carrying uncommitted changes to objects from request to request, until he decides to commit or rollback them). Even for a single user it might make sense to use mutliple ObjectContexts (e.g. request-scoped contexts to allow concurrent requests from the browser that change and commit objects independently).

ObjectContext is serializable and does not permanently hold to any of the application resources. So it does not have to be closed. If the context is not used anymore, it should simply be allowed to go out of scope and get garbage collected, just like any other Java object.

Persistent Object and its Lifecycle

Cayenne can persist Java objects that implement `org.apache.cayenne.Persistent` interface. Generally persistent classes are generated from the model as described above, so users do not have to worry about superclass and property implementation details.

Persistent interface provides access to 3 persistence-related properties - objectId, persistenceState and objectContext. All 3 are initialized by Cayenne runtime framework. Application code should not attempt to change them. However it is allowed to read them, which provides valuable runtime information. E.g. ObjectId can be used for quick equality check of 2 objects, knowing persistence state would allow highlighting changed objects, etc.

Each persistent object belongs to a single ObjectContext, and can be in one of the following persistence states (as defined in `org.apache.cayenne.PersistenceState`) :

Table 6. Persistence States

TRANSIENT	The object is not registered with an ObjectContext and will not be persisted.
NEW	The object is freshly registered in an ObjectContext, but has not been saved to the database yet and there is no matching database row.

COMMITED	The object is registered in an ObjectContext, there is a row in the database corresponding to this object, and the object state corresponds to the last known state of the matching database row.
MODIFIED	The object is registered in an ObjectContext, there is a row in the database corresponding to this object, but the object in-memory state has diverged from the last known state of the matching database row.
HOLLOW	The object is registered in an ObjectContext, there is a row in the database corresponding to this object, but the object state is unknown. Whenever an application tries to access a property of such object, Cayenne attempts reading its values from the database and "inflate" the object, turning it to COMMITED.
DELETED	The object is registered in an ObjectContext and has been marked for deletion in-memory. The corresponding row in the database will get deleted upon ObjectContext commit, and the object state will be turned into TRANSIENT.

ObjectContext Persistence API

One of the first things users usually want to do with an ObjectContext is to select some objects from a database. This is done by calling "performQuery" method:

```
SelectQuery query = new SelectQuery(Artist.class);
List<Artist> artists = context.performQuery(query);
```

We'll discuss queries in some detail in the following chapters. The example above is self-explanatory - we create a SelectQuery that matches all Artist objects present in the database, and then call "performQuery", getting a list of Artist objects.

Some queries can be quite complex, returning multiple result sets or even updating the database. For such queries ObjectContext provides "performGenericQuery"method. While not nearly as commonly-used as "performQuery", it is nevertheless important in some situations. E.g.:

```
Collection<Query> queries = ... // multiple queries that need to be run together
QueryChain query = new QueryChain(queries);

QueryResponse response = context.performGenericQuery(query);
```

An application might modify selected objects. E.g.:

```
Artist selectedArtist = artists.get(0);
selectedArtist.setName("Dali");
```

The first time the object property is changed, the object's state is automatically set to "MODIFIED" by Cayenne. Cayenne tracks all in-memory changes until a user calls "commitChanges":

```
context.commitChanges();
```

At this point all in-memory changes are analyzed and a minimal set of SQL statements is issued in a single transaction to synchronize the database with the in-memory state. In our example "commitChanges" commits just one object, but generally it can be any number of objects.

If instead of commit, we wanted to reset all changed objects to the previously committed state, we'd call `rollbackChanges` instead:

```
context.rollbackChanges();
```

"newObject" method call creates a persistent object and sets its state to "NEW":

```
Artist newArtist = context.newObject(Artist.class);
newArtist.setName("Picasso");
```

It will only exist in memory until "commitChanges" is issued. On commit Cayenne might generate a new primary key (unless a user set it explicitly, or a PK was inferred from a relationship) and issue an INSERT SQL statement to permanently store the object.

`deleteObjects` method takes one or more Persistent objects and marks them as "DELETED":

```
context.deleteObjects(artist1);
context.deleteObjects(artist2, artist3, artist4);
```

Additionally "deleteObjects" processes all delete rules modeled for the affected objects. This may result in implicitly deleting or modifying extra related objects. Same as insert and update, delete operations are sent to the database only when "commitChanges" is called. Similarly "rollbackChanges" will undo the effect of "newObject" and "deleteObjects".

`localObject` returns a copy of a given persistent object that is "local" to a given ObjectContext:

Since an application often works with more than one context, "localObject" is a rather common operation. E.g. to improve performance a user might utilize a single shared context to select and cache data, and then occasionally transfer some selected objects to another context to modify and commit them:

```
ObjectContext editingContext = runtime.newContext();
Artist localArtist = editingContext.localObject(artist);
```

Often an appliction needs to inspect mapping metadata. This information is stored in the EntityResolver object, accessible via the ObjectContext:

```
EntityResolver resolver = objectContext.getEntityResolver();
```

Here we discussed the most commonly used subset of the ObjectContext API. There are other useful methods, e.g. those allowing to inspect registered objects state in bulk, etc. Check the latest JavaDocs

for details.

Cayenne Helper Class

There is a useful helper class called "Cayenne" (fully-qualified name `"org.apache.cayenne.Cayenne"`) that builds on ObjectContext API to provide a number of very common operations. E.g. get a primary key (most entities do not model PK as an object property) :

```
long pk = Cayenne.longPKForObject(artist);
```

It also provides the reverse operation - finding an object given a known PK:

```
Artist artist = Cayenne.objectForPK(context, Artist.class, 34579);
```

If a query is expected to return 0 or 1 object, Cayenne helper class can be used to find this object. It throws an exception if more than one object matched the query:

```
Artist artist = (Artist) Cayenne.objectForQuery(context, new SelectQuery(Artist.class
));
```

Feel free to explore Cayenne class API for other useful methods.

ObjectContext Nesting

In all the examples shown so far an ObjectContext would directly connect to a database to select data or synchronize its state (either via commit or rollback). However another context can be used in all these scenarios instead of a database. This concept is called ObjectContext "nesting". Nesting is a parent/child relationship between two contexts, where child is a nested context and selects or commits its objects via a parent.

Nesting is useful to create isolated object editing areas (child contexts) that need to all be committed to an intermediate in-memory store (parent context), or rolled back without affecting changes already recorded in the parent. Think cascading GUI dialogs, or parallel AJAX requests coming to the same session.

In theory Cayenne supports any number of nesting levels, however applications should generally stay with one or two, as deep hierarchies will most certainly degrade the performance of the deeply nested child contexts. This is due to the fact that each context in a nesting chain has to update its own objects during most operations.

Cayenne ROP is an extreme case of nesting when a child context is located in a separate JVM and communicates with its parent via a web service. ROP is discussed in details in the following chapters. Here we concentrate on the same-VM nesting.

To create a nested context, use an instance of ServerRuntime, passing it the desired parent:

```
ObjectContext parent = runtime.getContext();
ObjectContext nested = runtime.getContext((DataChannel) parent);
```

From here a nested context operates just like a regular context (you can perform queries, create and delete objects, etc.). The only difference is that commit and rollback operations can either be limited to synchronization with the parent, or cascade all the way to the database:

```
// merges nested context changes into the parent context
nested.commitChangesToParent();

// regular 'commitChanges' cascades commit through the chain
// of parent contexts all the way to the database
nested.commitChanges();
```

```
// unrolls all local changes, getting context in a state identical to parent
nested.rollbackChangesLocally();

// regular 'rollbackChanges' cascades rollback through the chain of contexts
// all the way to the topmost parent
nested.rollbackChanges();
```

Generic Persistent Objects

As described in the CayenneModeler chapter, Cayenne supports mapping of completely generic classes to specific entities. Although for conveniece most applications should stick with entity-specific class mappings, the generic feature offers some interesting possibilities, such as creating mappings completely on the fly in a running application, etc.

Generic objects are first class citizens in Cayenne, and all common persistent operations apply to them as well. There are some pecularities however, described below.

When creating a new generic object, either cast your ObjectContext to DataContext (that provides "newObject(String)" API), or provide your object with an explicit ObjectId:

```
DataObject generic = ((DataContext) context).newObject("GenericEntity");
```

```
DataObject generic = new CayenneDataObject();
generic.setObjectId(new ObjectId("GenericEntity"));
context.registerNewObject(generic);
```

SelectQuery for generic object should be created passing entity name String in constructor, instead of a Java class:

```
SelectQuery query = new SelectQuery("GenericEntity");
```

Use DataObject API to access and modify properties of a generic object:

```
String name = (String) generic.readProperty("name");
generic.writeProperty("name", "New Name");
```

This is how an application can obtain entity name of a generic object:

```
String entityName = generic.getObjectId().getEntityName();
```

Transactions

Considering how much attention is given to managing transactions in most other ORMs, transactions have been conspicuously absent from the ObjectContext discussion till now. The reason is that transactions are seamless in Cayenne in all but a few special cases. ObjectContext is an in-memory container of objects that is disconnected from the database, except when it needs to run an operation. So it does not care about any surrounding transaction scope. Sure enough all database operations are transactional, so when an application does a commit, all SQL execution is wrapped in a database transaction. But this is done behind the scenes and is rarely a concern to the application code.

Two cases where transactions need to be taken into consideration are container-managed and application-managed transactions.

If you are using an EJB container (or some other JTA environment), you'll likely need to switch Cayenne runtime into "external transactions mode". This is either done in the Modeler (check DataDomain > 'Container-Managed Transactions' checkbox), or in the code:

```
runtime.getDataDomain().setUsingExternalTransactions(true);
```

In this case Cayenne assumes that JDBC Connections obtained by runtime whenever that might happen are all coming from a transactional DataSource managed by the container. In this case Cayenne does not attempt to commit or rollback the connections, leaving it up to the container to do that when appropriate.

In the second scenario, an application might need to define its own transaction scope that spans more than one Cayenne operation. E.g. two sequential commits that need to be rolled back together in case of failure. This can be done with an explicit thread-bound transaction that surrounds a set of operations. Application is responsible for committing or rolling it back:

```
Transaction tx = runtime.getDataDomain().createTransaction();
Transaction.bindThreadTransaction(tx);

try {
    // commit one or more contexts
    context1.commitChanges();
    context2.commitChanges();
    ....
    // after changing some objects in context1, commit again
    context1.commitChnages();
    ....
    // if no failures, commit
    tx.commit();
}
catch (Exception ex) {
    tx.setRollbackOnly();
}
finally {
    Transaction.bindThreadTransaction(null);

    if (tx.getStatus() == Transaction.STATUS_MARKED_ROLLEDBACK) {
        try {
           tx.rollback();
        }
        catch (Exception rollbackEx) {
        }
    }
}
```

2.4. Expressions

Expressions Overview

Cayenne provides a simple yet powerful object-based expression language. The most common use of expressions are to build qualifiers and orderings of queries that are later converted to SQL by Cayenne and to evaluate in-memory against specific objects (to access certain values in the object graph or to perform in-memory object filtering and sorting). Cayenne provides API to build expressions in the code and a parser to create expressions from strings.

Path Expressions

Before discussing how to build expressions, it is important to understand one group of expressions widely used in Cayenne - path expressions. There are two types of path expressions - object and database, used for navigating graphs of connected objects or joined DB tables respectively. Object paths are much more commonly used, as after all Cayenne is supposed to provide a degree of isolation of the object model from the database. However database paths are helpful in certain situations. General structure of path expressions is the following:

```
[db:]segment[+][.segment[+]...]
```

- `db:` is an optional prefix indicating that the following path is a DB path. Otherwise it is an object path.
- `segment` is a name of a property (relationship or attribute in Cayenne terms) in the path. Path must have at least one segment; segments are separated by dot (".").
- `+` An "OUTER JOIN" path component. Currently "+" only has effect when translated to SQL as OUTER JOIN. When evaluating expressions in memory, it is ignored.

An object path expression represents a chain of property names rooted in a certain (unspecified during expression creation) object and "navigating" to its related value. E.g. a path expression `artist.name` might be a property path starting from a Painting object, pointing to the related Artist object, and then to its name attribute. A few more examples:

- `name` - can be used to navigate (read) the "name" property of a Person (or any other type of object that has a "name" property).
- `artist.exhibits.closingDate` - can be used to navigate to a closing date of any of the exhibits of a Painting's Artist object.
- `artist.exhibits+.closingDate` - same as the previous example, but when translated into SQL, an OUTER JOIN will be used for "exhibits".

Similarly a database path expression is a dot-separated path through DB table joins and columns. In Cayenne joins are mapped as DbRelationships with some symbolic names (the closest concept to DbRelationship name in the DB world is a named foreign key constraint. But DbRelationship names are usually chosen arbitrarily, without regard to constraints naming or even constraints presence). A database path therefore might look like this - `db:dbrelationshipX.dbrelationshipY.COLUMN_Z`. More specific examples:

- `db:NAME` - can be used to navigate to the value of "NAME" column of some unspecified table.
- `db:artist.artistExhibits.exhibit.CLOSING_DATE` - can be used to match a closing date of any of the exhibits of a related artist record.

Cayenne supports "aliases" in path Expressions. E.g. the same expression can be written using explicit path or an alias:

- `artist.exhibits.closingDate` - full path
- `e.closingDate` - alias `e` is used for `artist.exhibits`.

SelectQuery using the second form of the path expression must be made aware of the alias via `SelectQuery.aliasPathSplits(..)`, otherwise an Exception will be thrown. The main use of aliases is to allow users to control how SQL joins are generated if the same path is encountered more than once in any given Expression. Each alias for any given path would result in a separate join. Without aliases, a single join will be used for a group of matching paths.

Creating Expressions from Strings

While in most cases users are likely to rely on API from the following section for expression creation, we'll start by showing String expressions, as this will help to understand the semantics. A Cayenne expression can be represented as a String, which can be converted to an expression object using `ExpressionFactory.exp` static method. Here is an example:

```
String expString = "name like 'A%' and price < 1000";
Expression exp = ExpressionFactory.exp(expString);
```

This particular expression may be used to match Paintings whose names that start with "A" and whose price is less than $1000. While this example is pretty self-explanatory, there are a few points worth mentioning. "name" and "price" here are object paths discussed earlier. As always, paths themselves are not attached to a specific root entity and can be applied to any entity that has similarly named attributes or relationships. So when we are saying that this expression "may be used to match Paintings", we are implying that there may be other entities, for which this expression is valid. Now the expression details...

Character constants that are not paths or numeric values should be enclosed in single or double quotes. Two of the expressions below are equivalent:

```
name = 'ABC'

// double quotes are escaped inside Java Strings of course
name = \"ABC\"
```

Case sensitivity. Expression operators are case sensitive and are usually lowercase. Complex words follow the Java camel-case style:

```
// valid
name likeIgnoreCase 'A%'

// invalid - will throw a parse exception
name LIKEIGNORECASE 'A%'
```

Grouping with parenthesis:

```
value = (price + 250.00) * 3
```

Path prefixes. Object expressions are unquoted strings, optionally prefixed by `obj:` (usually they are not prefixed at all actually). Database expressions are always prefixed with `db:`. A special kind of prefix, not discussed yet is `enum:` that prefixes an enumeration constant:

```
// object path
name = 'Salvador Dali'

// same object path - a rarely used form
obj:name = 'Salvador Dali'

// multi-segment object path
artist.name = 'Salvador Dali'

// db path
db:NAME = 'Salvador Dali'

// enumeration constant
name = enum:org.foo.EnumClass.VALUE1
```

Binary conditions are expressions that contain a path on the left, a value on the right, and some operation between them, such as equals, like, etc. They can be used as qualifiers in SelectQueries:

```
name like 'A%'
```

Named parameters. Expressions can have named parameters (names that start with "$"). Parameterized expressions allow to create reusable expression templates. Also if an Expression contains a complex object that doesn't have a simple String representation (e.g. a Date, a DataObject, an ObjectId), parameterizing such expression is the only way to represent it as String. Here are some examples:

```
Expression template = Expression.fromString("name = $name");
...
Map p1 = Collections.singletonMap("name", "Salvador Dali");
Expression qualifier1 = template.expWithParameters(p1);
...
Map p2 = Collections.singletonMap("name", "Monet");
Expression qualifier2 = template.expWithParameters(p2);
```

To create a named parameterized expression with a LIKE clause, SQL wildcards must be part of the values in the Map and not the expression string itself:

```
Expression template = Expression.fromString("name like $name");
...
Map p1 = Collections.singletonMap("name", "Salvador%");
Expression qualifier1 = template.expWithParameters(p1);
```

When matching on a relationship, parameters can be Persistent objects or ObjectIds:

```
Expression template = Expression.fromString("artist = $artist");
...
Artist dali = // asume we fetched this one already
Map p1 = Collections.singletonMap("artist", dali);
Expression qualifier1 = template.expWithParameters(p1);
```

Uninitialized parameters will be automatically pruned from expressions, so a user can omit some parameters when creating an expression from a parameterized template:

```
Expression template = Expression.fromString("name like $name and dateOfBirth > $date"
);
...
Map p1 = Collections.singletonMap("name", "Salvador%");
Expression qualifier1 = template.expWithParameters(p1);

// qualifier1 is now equals to "name like 'Salvador%'", the 'dateOfBirth' condition
was
// pruned, as no value was specified for the $date parameter
```

Null handling. Handling of Java nulls as operands is no different from normal values. Instead of using special conditional operators, like SQL does (IS NULL, IS NOT NULL), "=" and "!=" expressions are used directly with null values. It is up to Cayenne to translate expressions with nulls to the valid SQL.

A formal definition of all possible valid expressions in a form of JavaCC grammar is provided in Appendix C

Creating Expressions via API

Creating expressions from Strings is a powerful and dynamic approach, however a safer alternative is to use Java API. It provides some degree of compile-time checking of expressions validity. The API is cenetred around ExpressionFactory class, and the Expression class. ExpressionFactory contains a number of rather self-explanatory factory methods. We won't be going over all of them in detail, but will rather show a few general examples and some gotchas.

The following code recreates the expression from the previous chapter, but now using expression API:

```
// String expression: name like 'A%' and price < 1000
Expression e1 = ExpressionFactory.likeExp("name", "A%");
Expression e2 = ExpressionFactory.lessExp("price", 1000);
Expression finalExp = e1.andExp(e2);
```

This is more verbose than creating it from String, but it is also more resilient to the entity properties renaming and precludes semantic errors in the expression String.

The last line in the example above shows how to create a new expression by "chaining" two other epxressions. A common error when chaining expressions is to assume that "andExp" and "orExp" append another expression to the current expression. In fact a new expression is created. I.e. Expression API treats existing expressions as immutable.

As discussed earlier, Cayenne supports aliases in path Expressions, allowing to control how SQL joins are generated if the same path is encountered more than once in the same Expression. Two ExpressionFactory methods allow to implicitly generate aliases to "split" match paths into individual joins if needed:

```
Expression matchAllExp(String path, Collection values)
Expression matchAllExp(String path, Object... values)
```

"Path" argument to both of these methods can use a split character (a pipe symbol '|') instead of dot to indicate that relationship following a path should be split into a separate set of joins, one per collection value. There can only be one split at most in any given path. Split must always precede a relationship. E.g. `|exhibits.paintings`, `exhibits|paintings`, etc. Internally Cayenne would generate distinct aliases for each of the split expressions, forcing separate joins.

Evaluating Expressions in Memory

When used in a query, an expression is converted to SQL WHERE clause (or ORDER BY clause) by Cayenne during query execution. Thus the actual evaluation against the data is done by the database engine. However the same expressions can also be used for accessing object properties, calculating values, in-memory filtering.

Checking whether an object satisfies an expression:

```
Expression e = Artist.NAME.in("John", "Bob");
Artist artist = ...
if(e.match(artist)) {
   ...
}
```

Reading property value:

```
Expression e = Expression.fromString(User.NAME_PROPERTY);
String name = e.evaluate(user);
```

Filtering a list of objects:

```
Expression e = Artist.NAME.in("John", "Bob");
List<Artist> unfiltered = ...
List<Artist> filtered = e.filterObjects(unfiltered);
```

Current limitation of in-memory expressions is that no collections are permitted in the property path.

2.5. Orderings

An Ordering object defines how a list of objects should be ordered. Orderings are essentially path expressions combined with a sorting strategy. Creating an Ordering:

```
Ordering o = new Ordering(Painting.NAME_PROPERTY, SortOrder.ASCENDING);
```

Like expressions, orderings are translated into SQL as parts of queries (and the sorting occurs in the database). Also like expressions, orderings can be used in memory, naturally - to sort objects:

```
Ordering o = new Ordering(Painting.NAME_PROPERTY, SortOrder.ASCENDING_INSENSITIVE);
List<Painting> list = ...
o.orderList(list);
```

Note that unlike filtering with Expressions, ordering is performed in-place. This list object is reordered and no new list is created.

2.6. Queries

Queries are Java objects used by the application to communicate with the database. Cayenne knows how to translate queries into SQL statements appropriate for a particular database engine. Most often queries are used to find objects matching certain criteria, but there are other types of queries too. E.g. those allowing to run native SQL, call DB stored procedures, etc. When committing objects, Cayenne itself creates special queries to insert/update/delete rows in the database.

There is a number of built-in queries in Cayenne, described later in this chapter. Most of the newer queries use fluent API and can be created and executed as easy-to-read one-liners. Users can define their own query types to abstract certain DB interactions that for whatever reason can not be adequately described by the built-in set.

Queries can be roughly categorized as "object" and "native". Object queries (most notably ObjectSelect, SelectById, and EJBQLQuery) are built with abstractions originating in the object model (the "object" side in the "object-relational" divide). E.g. ObjectSelect is assembled from a Java class of the objects to fetch, a qualifier expression, orderings, etc. - all of this expressed in terms of the object model.

Native queries describe a desired DB operation as SQL code (SQLSelect, SQLTemplate query) or a reference to a stored procedure (ProcedureQuery), etc. The results of native queries are usually presented as Lists of Maps, with each map representing a row in the DB (a term "data row" is often used to describe such a map). They can potentially be converted to objects, however it may take a considerable effort to do so. Native queries are also less (if at all) portable across databases than object queries.

SelectQuery

SelectQuery is the most commonly used query in user applications. This may be the only query you will need in most appplications. It returns a list of persistent objects of a certain type specified in the query:

```
SelectQuery query = new SelectQuery(Artist.class);
List<Artist> objects = context.performQuery(query);
```

This returned all rows in the "ARTIST" table. If the logs were turned on, you might see the following SQL printed:

```
INFO: SELECT t0.DATE_OF_BIRTH, t0.NAME, t0.ID FROM ARTIST t0
INFO: === returned 5 row. - took 5 ms.
```

This SQL was generated by Cayenne from the SelectQuery above. SelectQuery can have a qualifier to select only the data that you care about. Qualifier is simply an Expression (Expressions where discussed in the previous chapter). If you only want artists whose name begins with 'Pablo', you might use the following qualifier expression:

```
SelectQuery query = new SelectQuery(Artist.class,
        ExpressionFactory.likeExp(Artist.NAME_PROPERTY, "Pablo%"));
List<Artist> objects = context.performQuery(query);
```

The SQL will look different this time:

```
INFO: SELECT t0.DATE_OF_BIRTH, t0.NAME, t0.ID FROM ARTIST t0 WHERE t0.NAME LIKE ?
[bind: 1->NAME:'Pablo%']
INFO: === returned 1 row. - took 6 ms.
```

SelectQuery allows to append parts of qualifier to self:

```
SelectQuery query = new SelectQuery(Artist.class);
query.setQualifier(ExpressionFactory.likeExp(Artist.NAME_PROPERTY, "A%"));
query.andQualifier(ExpressionFactory.greaterExp(Artist.DATE_OF_BIRTH_PROPERTY,
someDate));
```

To order the results of SelectQuery, one or more Orderings can be applied. Ordering were already discussed earlier. E.g.:

```
SelectQuery query = new SelectQuery(Artist.class);

// create Ordering object explicitly
query.addOrdering(new Ordering(Artist.DATE_OF_BIRTH_PROPERTY, SortOrder.DESCENDING));

// or let SelectQuery create it behind the scenes
query.addOrdering(Artist.NAME_PROPERTY, SortOrder.ASCENDING);
```

There's a number of other useful properties in SelectQuery that define what to select and how to optimize database interaction (prefetching, caching, fetch offset and limit, pagination, etc.). Some of them are discussed in separate chapters on caching and performance optimization. Others are fairly self-explanatory. Please check the API docs for the full extent of the SelectQuery features.

EJBQLQuery

EJBQLQuery was created as a part of an experiment in adopting some of Java Persistence API (JPA) approaches in Cayenne. It is a parameterized object query that is created from query String. A String used to build EJBQLQuery must conform to JPQL (JPA query language):

```
EJBQLQuery query = new EJBQLQuery("select a FROM Artist a");
```

JPQL details can be found in any JPA manual. Here we'll mention only how this fits into Cayenne and what are the differences between EJBQL and other Cayenne queries.

Although most frequently EJBQLQuery is used as an alternative to SelectQuery, there are also DELETE and UPDATE varieties available.

As of this version of Cayenne, DELETE and UPDATE do not change the state of objects in the ObjectContext. They are run directly against the database instead.

```
EJBQLQuery select = new EJBQLQuery("select a FROM Artist a WHERE a.name = 'Salvador
Dali'");
List<Artist> artists = context.performQuery(select);
```

```
EJBQLQuery delete = new EJBQLQuery("delete from Painting");
context.performGenericQuery(delete);
```

```
EJBQLQuery update = new EJBQLQuery("UPDATE Painting AS p SET p.name = 'P2' WHERE
p.name = 'P1'");
context.performGenericQuery(update);
```

In most cases SelectQuery is preferred to EJBQLQuery, as it is API-based, and provides you with better compile-time checks. However sometimes you may want a completely scriptable object

query. This is when you might prefer EJBQL. A more practical reason for picking EJBQL over SelectQuery though is that the former offers some extra selecting capabilities, namely aggregate functions and subqueries:

```
EJBQLQuery query = new EJBQLQuery("select a, COUNT(p) FROM Artist a JOIN a.paintings p
GROUP BY a");
List<Object[]> result = context.performQuery(query);
for(Object[] artistWithCount : result) {
    Artist a = (Artist) artistWithCount[0];
    int hasPaintings = (Integer) artistWithCount[1];
}
```

This also demonstrates a previously unseen type of select result - a List of Object[] elements, where each entry in an Object[] is either a DataObject or a scalar, depending on the query SELECT clause. A result can also be a list of scalars:

```
EJBQLQuery query = new EJBQLQuery("select a.name FROM Artist a");
List<String> names = context.performQuery(query);
```

While Cayenne Expressions discussed previously can be thought of as identical to JPQL WHERE clause, and indeed they are very close, there are a few noteable differences:

- Null handling: SelectQuery would translate the expressions matching NULL values to the corresponding "X IS NULL" or "X IS NOT NULL" SQL syntax. EJBQLQuery on the other hand requires explicit "IS NULL" (or "IS NOT NULL") syntax to be used, otherwise the generated SQL will look like "X = NULL" (or "X <> NULL"), which will evaluate differently.
- Expression Parameters: SelectQuery uses "$" to denote named parameters (e.g. "$myParam"), while EJBQL uses ":" (e.g. ":myParam"). Also EJBQL supports positional parameters denoted by the question mark: "?3".

SQLTemplate

SQLTemplate is a query that allows to run native SQL from a Cayenne application. It comes handy when the standard ORM concepts are not sufficient for a given query or an update. SQL is too powerful and allows to manipulate data in ways that are not easily described as a graph of related entities. Cayenne acknowledges this fact and provides this facility to execute SQL, mapping the result to objects when possible. Here are examples of selecting and non-selecting SQLTemplates:

```
SQLTemplate select = new SQLTemplate(Artist.class, "select * from ARTIST");
List<Artist> result = context.performQuery(select);
```

```
SQLTemplate update = new SQLTemplate(Artist.class, "delete from ARTIST");
QueryResponse response = context.performGenericQuery(update);
```

Cayenne doesn't make any attempt to make sense of the SQL semantics, so it doesn't know whether

a given query is performing a select or update, etc. It is the the user's decision to run a given query as a selecting or "generic".

Any data modifications done to DB as a result of SQLTemplate execution do not change the state of objects in the ObjectContext. So some objects in the context may become stale as a result.

Another point to note is that the first argument to the SQLTemplate constructor - the Java class - has the same meaning as in SelectQuery only when the result can be converted to objects (e.g. when this is a selecting query and it is selecting all columns from one table). In this case it denotes the "root" entity of this query result. If the query does not denote a single entity result, this argument is only used for query routing, i.e. determining which database it should be run against. You are free to use any persistent class or even a DataMap instance in such situation. It will work as long as the passed "root" maps to the same database as the current query.

To achieve interoperability between mutliple RDBMS a user can specify multiple SQL statements for the same SQLTemplate, each corresponding to a native SQL dialect. A key used to look up the right dialect during execution is a fully qualified class name of the corresponding DbAdapter. If no DB-specific statement is present for a given DB, a default generic statement is used. E.g. in all the examples above a default statement will be used regardless of the runtime database. So in most cases you won't need to explicitly "translate" your SQL to all possible dialects. Here is how this works in practice:

```
SQLTemplate select = new SQLTemplate(Artist.class, "select * from ARTIST");

// For Postgres it would be nice to trim padding of all CHAR columns.
// Otherwise those will be returned with whitespace on the right.
// assuming "NAME" is defined as CHAR...
String pgSQL = "SELECT ARTIST_ID, RTRIM(NAME), DATE_OF_BIRTH FROM ARTIST";
query.setTemplate(PostgresAdapter.class.getName(), pgSQL);
```

Scripting SQLTemplate with Velocity

The most interesting aspect of SQLTemplate (and the reason why it is called a "template") is that a SQL string is treated by Cayenne as an Apache Velocity template. Before sending it to DB as a PreparedStatement, the String is evaluated in the Velocity context, that does variable substitutions, and performs special callbacks in response to various directives, thus controlling query interaction with the JDBC layer.

Check Velocity docs for the syntax details. Here we'll just mention the two main scripting elements - "variables" (that look like `$var`) and "directives" (that look like `#directive(p1 p2 p3)`). All built-in Velocity directives are supported. Additionally Cayenne defines a number of its own directives to bind parameters to PreparedStatements and to control the structure of the ResultSet. These directives are described in the following sections.

Variable Substitution

All variables in the template string are replaced from query parameters:

```
SQLTemplate query = new SQLTemplate(Artist.class, "delete from $tableName");
query.setParameters(Collections.singletonMap("tableName", "mydb.PAINTING"));

// this will generate SQL like this: "delete from mydb.PAINTING"
```

The example above demonstrates the point made earlier in this chapter - even if we don't know upfront which table the query will run against, we can still use a fixed "root" in constructor (`Artist.class` in this case) , as we are not planning on converting the result to objects.

Variable substitution within the text uses `"object.toString()"` method to replace the variable value. Keep in mind that this may not be appropriate in all situations. E.g. passing a date object in a WHERE clause expression may be converted to a String not understood by the target RDBMS SQL parser. In such cases variable should be wrapped in `#bind` directive as described below.

Directives

These are the Cayenne directives used to customize SQLTemplate parsing and integrate it with the JDBC layer:

#bind

Creates a PreparedStatement positional parameter in place of the directive, binding the value to it before statement execution. `#bind` is allowed in places where a "?" would be allowed in a PreparedStatement. And in such places it almost always makes sense to pass objects to the template via this or other forms of #bind instead of inserting them inline.

Semantics:

```
#bind(value)
#bind(value jdbcType)
#bind(value jdbcType scale)
```

Arguments:

- `value` - can either be a char constant or a variable that is resolved from the query parameters. Note that the variable can be a collection, that will be automatically expanded into a list of individual value bindings. This is useful for instance to build IN conditions.
- `jdbcType` - is a JDBC data type of the parameter as defined in `java.sql.Types`.
- `scale` - An optional scale of the numeric value. Same as "scale" in PreparedStatement.

Usage:

```
#bind($xyz)
#bind('str')
#bind($xyz 'VARCHAR')
#bind($xyz 'DECIMAL' 2)
```

Full example:

```
update ARTIST set NAME = #bind($name) where ID = #bind($id)
```

#bindEqual

Same as #bind, but also includes the "=" sign in front of the value binding. Look at the example below - we took the #bind example and replaced `"ID = #bind(..)"` with `"ID #bindEqual(..)"`. While it looks like a clumsy shortcut to eliminate the equal sign, the actual reason why this is useful is that it allows the value to be null. If the value is not null, `"= ?"` is generated, but if it is, the resulting chunk of the SQL would look like `"IS NULL"` and will be compilant with what the DB expects.

Semantics:

```
#bindEqual(value)
#bindEqual(value jdbcType)
#bindEqual(value jdbcType scale)
```

Arguments: (same as #bind)

Usage:

```
#bindEqual($xyz)
#bindEqual('str')
#bindEqual($xyz 'VARCHAR')
#bindEqual($xyz 'DECIMAL' 2)
```

Full example:

```
update ARTIST set NAME = #bind($name) where ID #bindEqual($id)
```

#bindNotEqual

This directive deals with the same issue as `#bindEqual` above, only it generates "not equal" in front of the value (or IS NOT NULL).

Semantics:

```
#bindNotEqual(value)
#bindNotEqual(value jdbcType)
#bindNotEqual(value jdbcType scale)
```

Arguments: (same as #bind)

Usage:

```
#bindNotEqual($xyz)
#bindNotEqual('str')
#bindNotEqual($xyz 'VARCHAR')
#bindNotEqual($xyz 'DECIMAL' 2)
```

Full example:

```
update ARTIST set NAME = #bind($name) where ID #bindEqual($id)
```

#bindObjectEqual

It can be tricky to use a Persistent object or an ObjectId in a binding, especially for tables with compound primary keys. This directive helps to handle such binding. It maps columns in the query to the names of Persistent object ID columns, extracts ID values from the object, and generates SQL like "COL1 = ? AND COL2 = ? ..." , binding positional parameters to ID values. It can also correctly handle null object. Also notice how we are specifying a Velocity array for multi-column PK.

Semantics:

```
#bindObjectEqual(value columns idColumns)
```

Arguments:

- `value` - must be a variable that is resolved from the query parameters to a Persistent or ObjectId.
- `columns` - the names of the columns to generate in the SQL.
- `idColumn` - the names of the ID columns for a given entity. Must match the order of "columns" to match against.

Usage:

```
#bindObjectEqual($a 't0.ID' 'ID')
#bindObjectEqual($b ['t0.FK1', 't0.FK2'] ['PK1', 'PK2'])
```

Full example:

```
String sql = "SELECT * FROM PAINTING t0 WHERE #bindObjectEqual($a 't0.ARTIST_ID'
'ARTIST_ID' ) ORDER BY PAINTING_ID";
SQLTemplate select = new SQLTemplate(Artist.class, sql);

Artist a = ....
select.setParameters(Collections.singletonMap("a", a));
```

#bindObjectNotEqual

Same as #bindObjectEqual above, only generates "not equal" operator for value comparison (or IS NOT NULL).

Semantics:

```
#bindObjectNotEqual(value columns idColumns)
```

Arguments: (same as #bindObjectEqual)

Usage:

```
#bindObjectNotEqual($a 't0.ID' 'ID')
#bindObjectNotEqual($b ['t0.FK1', 't0.FK2'] ['PK1', 'PK2'])
```

Full example:

```
String sql = "SELECT * FROM PAINTING t0 WHERE #bindObjectNotEqual($a 't0.ARTIST_ID'
'ARTIST_ID' ) ORDER BY PAINTING_ID";
SQLTemplate select = new SQLTemplate(Artist.class, sql);

Artist a = ....
select.setParameters(Collections.singletonMap("a", a));
```

#result

Renders a column in SELECT clause of a query and maps it to a key in the result DataRow. Also ensures the value read is of the correct type. This allows to create a DataRow (and ultimately - a persistent object) from an arbitrary ResultSet.

Semantics:

```
#result(column)
#result(column javaType)
#result(column javaType alias)
#result(column javaType alias dataRowKey)
```

Arguments:

- `column` - the name of the column to render in SQL SELECT clause.
- `javaType` - a fully-qualified Java class name for a given result column. For simplicity most common Java types used in JDBC can be specified without a package. These include all numeric types, primitives, String, SQL dates, BigDecimal and BigInteger. So `"#result('A' 'String')"`, `"#result('B' 'java.lang.String')"` and `"#result('C' 'int')"` are all valid
- `alias` - specifies both the SQL alias of the column and the value key in the DataRow. If omitted,

"column" value is used.

- `dataRowKey` - needed if SQL 'alias' is not appropriate as a DataRow key on the Cayenne side. One common case when this happens is when a DataRow retrieved from a query is mapped using joint prefetch keys (see below). In this case DataRow must use database path expressions for joint column keys, and their format is incompatible with most databases alias format.

Usage:

```
#result('NAME')
#result('DATE_OF_BIRTH' 'java.util.Date')
#result('DOB' 'java.util.Date' 'DATE_OF_BIRTH')
#result('DOB' 'java.util.Date' '' 'artist.DATE_OF_BIRTH')
#result('SALARY' 'float')
```

Full example:

```
SELECT #result('ID' 'int'), #result('NAME' 'String'), #result('DATE_OF_BIRTH'
'java.util.Date') FROM ARTIST
```

#chain and #chunk

`#chain` and `#chunk` directives are used for conditional inclusion of SQL code. They are used together with `#chain` wrapping multiple `#chunks`. A chunk evaluates its parameter expression and if it is NULL suppresses rendering of the enclosed SQL block. A chain renders its prefix and its chunks joined by the operator. If all the chunks are suppressed, the chain itself is suppressed. This allows to work with otherwise hard to script SQL semantics. E.g. a WHERE clause can contain multiple conditions joined with AND or OR. Application code would like to exclude a condition if its right-hand parameter is not present (similar to Expression pruning discussed above). If all conditions are excluded, the entire WHERE clause should be excluded. chain/chunk allows to do that.

Semantics:

```
#chain(operator) ... #end
#chain(operator prefix) ... #end
#chunk() ... #end
#chunk(param) ... #end
```

Full example:

```
#chain('OR' 'WHERE')
    #chunk($name) NAME LIKE #bind($name) #end"
    #chunk($id) ARTIST_ID > #bind($id) #end"
#end"
```

Mapping SQLTemplate Results

Here we'll discuss how to convert the data selected via SQLTemplate to some useable format, compatible with other query results. It can either be very simple or very complex, depending on the structure of the SQL, JDBC driver nature and the desired result structure. This section presents various tips and tricks dealing with result mapping.

By default SQLTemplate is expected to return a List of Persistent objects of its root type. This is the simple case:

```
SQLTemplate query = new SQLTemplate(Artist.class, "SELECT * FROM ARTIST");

// List of Artists
List<Artist> artists = context.performQuery(query);
```

Just like SelectQuery, SQLTemplate can fetch DataRows. In fact DataRows option is very useful with SQLTemplate, as the result type most often than not does not represent a Cayenne entity, but instead may be some aggregated report or any other data whose object structure is opaque to Cayenne:

```
String sql = "SELECT t0.NAME, COUNT(1) FROM ARTIST t0 JOIN PAINTING t1 ON (t0.ID =
t1.ARTIST_ID) "
    + "GROUP BY t0.NAME ORDER BY COUNT(1)";
SQLTemplate query = new SQLTemplate(Artist.class, sql);

// ensure we are fetching DataRows
query.setFetchingDataRows(true);

// List of DataRow
List<DataRow> rows = context.performQuery(query);
```

In the example above, even though the query root is Artist. the result is a list of artist names with painting counts (as mentioned before in such case "root" is only used to find the DB to fetch against, but has no bearning on the result). The DataRows here are the most appropriate and desired result type.

In a more advanced case you may decide to fetch a list of scalars or a list of Object[] with each array entry being either an entity or a scalar. You probably won't be doing this too often and it requires quite a lot of work to setup, but if you want your SQLTemplate to return results similar to EJBQLQuery, it is doable using SQLResult as described below:

```
SQLTemplate query = new SQLTemplate(Painting.class, "SELECT ESTIMATED_PRICE P FROM
PAINTING");

// let Cayenne know that result is a scalar
SQLResult resultDescriptor = new SQLResult();
resultDescriptor.addColumnResult("P");
query.setResult(resultDescriptor);

// List of BigDecimals
List<BigDecimal> prices = context.performQuery(query);
```

```
SQLTemplate query = new SQLTemplate(Artist.class, "SELECT t0.ID, t0.NAME,
t0.DATE_OF_BIRTH, COUNT(t1.PAINTING_ID) C " +
      "FROM ARTIST t0 LEFT JOIN PAINTING t1 ON (t0.ID = t1.ARTIST_ID) " +
      "GROUP BY t0.ID, t0.NAME, t0.DATE_OF_BIRTH");

// let Cayenne know that result is a mix of Artist objects and the count of their
paintings
EntityResult artistResult = new EntityResult(Artist.class);
artistResult.addDbField(Artist.ID_PK_COLUMN, "ARTIST_ID");
artistResult.addObjectField(Artist.NAME_PROPERTY, "NAME");
artistResult.addObjectField(Artist.DATE_OF_BIRTH_PROPERTY, "DATE_OF_BIRTH");

SQLResult resultDescriptor = new SQLResult();
resultDescriptor.addEntityResult(artistResult);
resultDescriptor.addColumnResult("C");
query.setResult(resultDescriptor);

// List of Object[]
List<Object[]> data = context.performQuery(query);
```

Another trick related to mapping result sets is making Cayenne recognize prefetched entities in the result set. This emulates "joint" prefetching of SelectQuery, and is achieved by special column naming. Columns belonging to the "root" entity of the query should use unqualified names corresponding to the root DbEntity columns. For each related entity column names must be prefixed with relationship name and a dot (e.g. "toArtist.ID"). Column naming can be controlled with "#result" directive:

```
String sql = "SELECT distinct "
    + "#result('t1.ESTIMATED_PRICE' 'BigDecimal' '' 'paintings.ESTIMATED_PRICE'), "
    + "#result('t1.PAINTING_TITLE' 'String' '' 'paintings.PAINTING_TITLE'), "
    + "#result('t1.GALLERY_ID' 'int' '' 'paintings.GALLERY_ID'), "
    + "#result('t1.ID' 'int' '' 'paintings.ID'), "
    + "#result('NAME' 'String'), "
    + "#result('DATE_OF_BIRTH' 'java.util.Date'), "
    + "#result('t0.ID' 'int' '' 'ID') "
    + "FROM ARTIST t0, PAINTING t1 "
    + "WHERE t0.ID = t1.ARTIST_ID";

SQLTemplate q = new SQLTemplate(Artist.class, sql);
q.addPrefetch(Artist.PAINTINGS_PROPERTY)
List<Artist> objects = context.performQuery(query);
```

And the final tip deals with capitalization of the DataRow keys. Queries like `"SELECT * FROM…"` and even `"SELECT COLUMN1, COLUMN2, … FROM …"` can sometimes result in Cayenne exceptions on attempts to convert fetched DataRows to objects. Essentially any query that is not using a `#result` directive to describe the result set is prone to this problem, as different databases may produce different capitalization of the java.sql.ResultSet columns.

The most universal way to address this issue is to describe each column explicitly in the SQLTemplate via `#result`, e.g.: `"SELECT #result('column1'), #result('column2'), .."`. However this quickly becomes impractical for tables with lots of columns. For such cases Cayenne provides a shortcut based on the fact that an ORM mapping usually follows some naming convention for the column names. Simply put, for case-insensitive databases developers normally use either all lowercase or all uppercase column names. Here is the API that takes advantage of that user knowledge and forces Cayenne to follow a given naming convention for the DataRow keys (this is also available as a dropdown in the Modeler):

```
SQLTemplate query = new SQLTemplate("SELECT * FROM ARTIST");
query.setColumnNamesCapitalization(CapsStrategy.LOWER);
List objects = context.performQuery(query);
```

or

```
SQLTemplate query = new SQLTemplate("SELECT * FROM ARTIST");
query.setColumnNamesCapitalization(CapsStrategy.UPPER);
List objects = context.performQuery(query);
```

None of this affects the generated SQL, but the resulting DataRows are using correct capitalization. Note that you probably shouldn't bother with this unless you are getting CayenneRuntimeExceptions when fetching with SQLTemplate.

ProcedureQuery

Stored procedures are mapped as separate objects in CayenneModeler. ProcedureQuery provides a way to execute them with a certain set of parameters. Just like with SQLTemplate, the outcome of a procedure can be anything - a single result set, mutliple result sets, some data modification (returned as an update count), or a combination of these. So use "performQuery" to get a single result set, and use "performGenericQuery" for anything else:

```
ProcedureQuery query = new ProcedureQuery("my_procedure", Artist.class);

// Set "IN" parameter values
query.addParam("p1", "abc");
query.addParam("p2", 3000);

List<Artist> result = context.performQuery(query);
```

```
// here we do not bother with root class.
// Procedure name gives us needed routing information
ProcedureQuery query = new ProcedureQuery("my_procedure");

query.addParam("p1", "abc");
query.addParam("p2", 3000);

QueryResponse response = context.performGenericQuery(query);
```

A stored procedure can return data back to the application as result sets or via OUT parameters. To simplify the processing of the query output, QueryResponse treats OUT parameters as if it was a separate result set. If a stored procedure declares any OUT or INOUT parameters, QueryResponse will contain their returned values in the very first result list:

```
ProcedureQuery query = new ProcedureQuery("my_procedure");
QueryResponse response = context.performGenericQuery(query);

// read OUT parameters
List out = response.firstList();

if(!out.isEmpty()) {
    Map outParameterValues = (Map) outList.get(0);
}
```

There maybe a situation when a stored procedure handles its own transactions, but an application is configured to use Cayenne-managed transactions. This is obviously conflicting and undesirable behavior. In this case ProcedureQueries should be executed explicitly wrapped in an "external" Transaction. This is one of the few cases when a user should worry about transactions at all. See Transactions section for more details.

NamedQuery

NamedQuery is a query that is a reference to another query stored in the DataMap. The actual stored query can be SelectQuery, SQLTemplate, EJBQLQuery, etc. It doesn't matter - the API for calling them is the same - via a NamedQuery:

```
String[] keys = new String[] {"loginid", "password"};
Object[] values = new String[] {"joe", "secret"};

NamedQuery query = new NamedQuery("Login", keys, values);

List<User> matchingUsers = context.performQuery(query);
```

Custom Queries

If a user needs some extra functionality not addressed by the existing set of Cayenne queries, he can write his own. The only requirement is to implement `org.apache.cayenne.query.Query` interface. The easiest way to go about it is to subclass some of the base queries in Cayenne.

E.g. to do something directly in the JDBC layer, you might subclass AbstractQuery:

```
public class MyQuery extends AbstractQuery {

    @Override
    public SQLAction createSQLAction(SQLActionVisitor visitor) {
        return new SQLAction() {

            @Override
            public void performAction(Connection connection, OperationObserver
observer) throws SQLException, Exception {
                // 1. do some JDBC work using provided connection...
                // 2. push results back to Cayenne via OperationObserver
            }
        };
    }
}
```

To delegate the actual query execution to a standard Cayenne query, you may subclass IndirectQuery:

```
public class MyDelegatingQuery extends IndirectQuery {

    @Override
    protected Query createReplacementQuery(EntityResolver resolver) {
        SQLTemplate delegate = new SQLTemplate(SomeClass.class, generateRawSQL());
        delegate.setFetchingDataRows(true);
        return delegate;
    }

    protected String generateRawSQL() {
        // build some SQL string
    }
}
```

In fact many internal Cayenne queries are IndirectQueries, delegating to SelectQuery or SQLTemplate after some preprocessing.

2.7. Lifecycle Events

An application might be interested in getting notified when a Persistent object moves through its lifecycle (i.e. fetched from DB, created, modified, committed). E.g. when a new object is created, the application may want to initialize its default properties (this can't be done in constructor, as constructor is also called when an object is fetched from DB). Before save, the application may perform validation and/or set some properties (e.g. "updatedTimestamp"). After save it may want to create an audit record for each saved object, etc., etc.

All this can be achieved by declaring callback methods either in Persistent objects or in non-persistent listener classes defined by the application (further simply called "listeners"). There are eight types of lifecycle events supported by Cayenne, listed later in this chapter. When any such event occurs (e.g. an object is committed), Cayenne would invoke all appropriate callbacks. Persistent objects would receive their own events, while listeners would receive events from any objects.

Cayenne allows to build rather powerful and complex "workflows" or "processors" tied to objects lifecycle, especially with listeners, as they have full access to the application evnironment outside Cayenne. This power comes from such features as filtering which entity events are sent to a given listener and the ability to create a common operation context for multiple callback invocations. All of these are discussed later in this chapter.

Types of Lifecycle Events

Cayenne defines the following 8 types of lifecycle events for which callbacks can be regsitered:

Table 7. Lifecycle Event Types

Event	Occurs...
PostAdd	right after a new object is created inside `ObjectContext.newObject()`. When this event is fired the object is already registered with its ObjectContext and has its ObjectId and ObjectContext properties set.
PrePersist	right before a new object is committed, inside `ObjectContext.commitChanges()` and `ObjectContext.commitChangesToParent()` (and after `"validateForInsert()"`).
PreUpdate	right before a modified object is committed, inside `ObjectContext.commitChanges()` and `ObjectContext.commitChangesToParent()` (and after `"validateForUpdate()"`).
PreRemove	right before an object is deleted, inside `ObjectContext.deleteObjects()`. The event is also generated for each object indirectly deleted as a result of CASCADE delete rule.
PostPersist	right after a commit of a new object is done, inside `ObjectContext.commitChanges()`.
PostUpdate	right after a commit of a modified object is done, inside `ObjectContext.commitChanges()`.
PostRemove	right after a commit of a deleted object is done, inside `ObjectContext.commitChanges()`.
PostLoad	• After an object is fetched inside ObjectContext.performQuery(). • After an object is reverted inside ObjectContext.rollbackChanges(). • Anytime a faulted object is resolved (i.e. if a relationship is fetched).

Callbacks on Persistent Objects

Callback methods on Persistent classes are mapped in CayenneModeler for each ObjEntity. Empty callback methods are automatically created as a part of class generation (either with Maven, Ant or the Modeler) and are later filled with appropriate logic by the programmer. E.g. assuming we mapped a 'post-add' callback called 'onNewOrder' in ObjEntity 'Order', the following code will be generated:

```
public abstract class _Order extends CayenneDataObject {
    protected abstract void onNewOrder();
}

public class Order extends _Order {

    @Override
    protected void onNewOrder() {
        //TODO: implement onNewOrder
    }
}
```

As `onNewOrder()` is already declared in the mapping, it does not need to be registered explicitly. Implementing the method in subclass to do something meaningful is all that is required at this

point.

As a rule callback methods do not have any knowledge of the outside application, and can only access the state of the object itself and possibly the state of other persistent objects via object's own ObjectContext.

Validation and callbacks: There is a clear overlap in functionality between object callbacks and `DataObject.validateForX()` methods. In the future validation may be completely superceeded by callbacks. It is a good idea to use "validateForX" strictly for validation (or not use it at all). Updating the state before commit should be done via callbacks.

Callbacks on Non-Persistent Listeners

While listener callback methods can be declared in the Modeler (at least as of this wrting), which ensures their automatic registration in runtime, there's a big downside to it. The power of the listeners lies in their complete separation from the XML mapping. The mapping once created, can be reused in different contexts each having a different set of listeners. Placing a Java class of the listener in the XML mapping, and relying on Cayenne to instantiate the listeners severly limits mapping reusability. Further down in this chapter we'll assume that the listener classes are never present in the DataMap and are registered via API.

A listener is simply some application class that has one or more annotated callback methods. A callback method signature should be `void someMethod(SomePersistentType object)`. It can be public, private, protected or use default access:

```
public class OrderListener {

  @PostAdd(Order.class)
  public void setDefaultsForNewOrder(Order o) {
     o.setCreatedOn(new Date());
  }
}
```

Notice that the example above contains an annotation on the callback method that defines the type of the event this method should be called for. Before we go into annotation details, we'll show how to create and register a listener with Cayenne. It is always a user responsibility to register desired application listeners, usually right after ServerRuntime is started. Here is an example:

First let's define 2 simple listeners.

```
public class Listener1 {

    @PostAdd(MyEntity.class)
    void postAdd(Persistent object) {
        // do something
    }
}

public class Listener2 {

    @PostRemove({ MyEntity1.class, MyEntity2.class })
    void postRemove(Persistent object) {
        // do something
    }

    @PostUpdate({ MyEntity1.class, MyEntity2.class })
    void postUpdate(Persistent object) {
        // do something
    }
}
```

Ignore the annotations for a minute. The important point here is that the listeners are arbitrary classes unmapped and unknown to Cayenne, that contain some callback methods. Now let's register them with runtime:

```
ServerRuntime runtime = ...

LifecycleCallbackRegistry registry =
    runtime.getDataDomain().getEntityResolver().getCallbackRegistry();

registry.addListener(new Listener1());
registry.addListener(new Listener2());
```

Listeners in this example are very simple. However they don't have to be. Unlike Persistent objects, normally listeners initialization is managed by the application code, not Cayenne, so listeners may have knowledge of various application services, operation transactional context, etc. Besides a single listener can apply to multiple entities. As a consequence their callbacks can do more than just access a single ObjectContext.

Now let's discuss the annotations. There are eight annotations exactly matching the names of eight lifecycle events. A callback method in a listener should be annotated with at least one, but possibly with more than one of them. Annotation itself defines what event the callback should react to. Annotation parameters are essentially an entity filter, defining a subset of ObjEntities whose events we are interested in:

```
// this callback will be invoked on PostRemove event of any object
// belonging to MyEntity1, MyEntity2 or their subclasses
@PostRemove({ MyEntity1.class, MyEntity2.class })
void postRemove(Persistent object) {
    ...
}
```

```
// similar example with multipe annotations on a single method
// each matching just one entity
@PostPersist(MyEntity1.class)
@PostRemove(MyEntity1.class)
@PostUpdate(MyEntity1.class)
void postCommit(MyEntity1 object) {
    ...
}
```

As shown above, "value" (the implicit annotation parameter) can contain one or more entity classes. Only these entities' events will result in callback invocation. There's also another way to match entities - via custom annotations. This allows to match any number of entities without even knowing what they are. Here is an example. We'll first define a custom annotation:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Tag {

}
```

Now we can define a listener that will react to events from ObjEntities annotated with this annotation:

```
public class Listener3 {

    @PostAdd(entityAnnotations = Tag.class)
    void postAdd(Persistent object) {
        // do something
    }
}
```

As you see we don't have any entities yet, still we can define a listener that does something useful. Now let's annotate some entities:

```
@Tag
public class MyEntity1 extends _MyEntity1 {

}

@Tag
public class MyEntity2 extends _MyEntity2 {

}
```

Combining Listeners with DataChannelFilters

A final touch in the listeners design is preserving the state of the listener within a single select or commit, so that events generated by multiple objects can be collected and processed all together. To do that you will need to implement a `DataChannelFilter`, and add some callback methods to it. They will store their state in a ThreadLocal variable of the filter. Here is an example filter that does something pretty meaningless - counts how many total objects were committed. However it demonstrates the important pattern of aggregating multiple events and presenting a combined result:

```
public class CommittedObjectCounter implements DataChannelFilter {

    private ThreadLocal<int[]> counter;

    @Override
    public void init(DataChannel channel) {
        counter = new ThreadLocal<int[]>();
    }

    @Override
    public QueryResponse onQuery(ObjectContext originatingContext, Query query,
DataChannelFilterChain filterChain) {
        return filterChain.onQuery(originatingContext, query);
    }

    @Override
    public GraphDiff onSync(ObjectContext originatingContext, GraphDiff changes, int
syncType,
            DataChannelFilterChain filterChain) {

        // init the counter for the current commit
        counter.set(new int[1]);

        try {
            return filterChain.onSync(originatingContext, changes, syncType);
        } finally {

            // process aggregated result and release the counter
            System.out.println("Committed " + counter.get()[0] + " object(s)");
            counter.set(null);
        }
    }

    @PostPersist(entityAnnotations = Tag.class)
    @PostUpdate(entityAnnotations = Tag.class)
    @PostRemove(entityAnnotations = Tag.class)
    void afterCommit(Persistent object) {
        counter.get()[0]++;
    }
}
```

Now since this is both a filter and a listener, it needs to be registered as such:

```
CommittedObjectCounter counter = new CommittedObjectCounter();

ServerRuntime runtime = ...
DataDomain domain = runtime.getDataDomain();

// register filter
domain.addFilter(counter);

// register listener
domain.getEntityResolver().getCallbackRegistry().addListener(counter);
```

2.8. Performance Tuning

Prefetching

Prefetching is a technique that allows to bring back in one query not only the queried objects, but also objects related to them. In other words it is a controlled eager relationship resolving mechanism. Prefetching is discussed in the "Performance Tuning" chapter, as it is a powerful performance optimization method. However another common application of prefetching is to refresh stale object relationships, so more generally it can be viewed as a technique for managing subsets of the object graph.

Prefetching example:

```
SelectQuery query = new SelectQuery(Artist.class);

// this instructs Cayenne to prefetch one of Artist's relationships
query.addPrefetch("paintings");

// query is expecuted as usual, but the resulting Artists will have
// their paintings "inflated"
List<Artist> artists = context.performQuery(query);
```

All types of relationships can be preftetched - to-one, to-many, flattened.

A prefetch can span multiple relationships:

```
 query.addPrefetch("paintings.gallery");
```

A query can have multiple prefetches:

```
query.addPrefetch("paintings");
query.addPrefetch("paintings.gallery");
```

If a query is fetching DataRows, all "disjoint" prefetches are ignored, only "joint" prefetches are

executed (see prefetching semantics discussion below for what disjoint and joint prefetches mean).

Prefetching Semantics

Prefetching semantics defines a strategy to prefetch relationships. Depending on it, Cayenne would generate different types of queries. The end result is the same - query root objects with related objects fully resolved. However semantics can affect preformance, in some cases significantly. There are 3 types of prefetch semantics, all defined as constants in `org.apache.cayenne.query.PrefetchTreeNode`:

```
PrefetchTreeNode.JOINT_PREFETCH_SEMANTICS
PrefetchTreeNode.DISJOINT_PREFETCH_SEMANTICS
PrefetchTreeNode.DISJOINT_BY_ID_PREFETCH_SEMANTICS
```

Each query has a default prefetch semantics, so generally users do not have to worry about changing it, except when performance is a concern, or a few special cases when a default sematics can't produce the correct result. SelectQuery uses DISJOINT_PREFETCH_SEMANTICS by default. Semantics can be changed as follows:

```
SelectQuery query = new SelectQuery(Artist.class);
query.addPrefetch("paintings").setSemantics(
                PrefetchTreeNode.JOINT_PREFETCH_SEMANTICS);
```

There's no limitation on mixing different types of semantics in the same SelectQuery. Multiple prefetches each can have its own semantics.

SQLTemplate and ProcedureQuery are both using JOINT_PREFETCH_SEMANTICS and it can not be changed due to the nature of these two queries.

Disjoint Prefetching Semantics

This semantics results in Cayenne generatiing one SQL statement for the main objects, and a separate statement for each prefetch path (hence "disjoint" - related objects are not fetched with the main query). Each additional SQL statement uses a qualifier of the main query plus a set of joins traversing the preftech path between the main and related entity.

This strategy has an advantage of efficient JVM memory use, and faster overall result processing by Cayenne, but it requires (1+N) SQL statements to be executed, where N is the number of prefetched relationships.

Disjoint-by-ID Prefetching Semantics

This is a variation of disjoint prefetch where related objects are matched against a set of IDs derived from the fetched main objects (or intermediate objects in a multi-step prefetch). Cayenne limits the size of the generated WHERE clause, as most DBs can't parse arbitrary large SQL. So prefetch queries are broken into smaller queries. The size of is controlled by the DI property Constants.SERVER_MAX_ID_QUALIFIER_SIZE_PROPERTY (the default number of conditions in the generated WHERE clause is 10000). Cayenne will generate (1 + N * M) SQL statements for each

query using disjoint-by-ID prefetches, where N is the number of relationships to prefetch, and M is the number of queries for a given prefetch that is dependent on the number of objects in the result (ideally M = 1).

The advantage of this type of prefetch is that matching database rows by ID may be much faster than matching the qualifier of the original query. Moreover this is **the only type of prefetch** that can handle SelectQueries with **fetch** limit. Both joint and regular disjoint prefetches may produce invalid results or generate inefficient fetch-the-entire table SQL when fetch limit is in effect.

The disadvantage is that query SQL can get unwieldy for large result sets, as each object will have to have its own condition in the WHERE clause of the generated SQL.

Joint Prefetching Semantics

Joint semantics results in a single SQL statement for root objects and any number of jointly prefetched paths. Cayenne processes in memory a cartesian product of the entities involved, converting it to an object tree. It uses OUTER joins to connect prefetched entities.

Joint is the most efficient prefetch type of the three as far as generated SQL goes. There's always just 1 SQL query generated. Its downsides are the potentially increased amount of data that needs to get across the network between the application server and the database, and more data processing that needs to be done on the Cayenne side.

Data Rows

Converting result set data to Persistent objects and registering these objects in the ObjectContext can be an expensive operation compareable to the time spent running the query (and frequently exceeding it). Internally Cayenne builds the result as a list of DataRows, that are later converted to objects. Skipping the last step and using data in the form of DataRows can significantly increase performance.

DataRow is a simply a map of values keyed by their DB column name. It is a ubiqutous representation of DB data used internally by Cayenne. And it can be quite usable as is in the application in many cases. So performance sensitive selects should consider DataRows - it saves memory and CPU cycles. All selecting queries support DataRows option, e.g.:

```
SelectQuery query = new SelectQuery(Artist.class);
query.setFetchingDataRows(true);

List<DataRow> rows = context.performQuery(query);
```

```
SQLTemplate query = new SQLTemplate(Artist.class, "SELECT * FROM ARTIST");
query.setFetchingDataRows(true);

List<DataRow> rows = context.performQuery(query);
```

Moreover DataRows may be converted to Persistent objects later as needed. So e.g. you may implement some in-memory filtering, only converting a subset of fetched objects:

```
// you need to cast ObjectContext to DataContext to get access to 'objectFromDataRow'
DataContext dataContext = (DataContext) context;

for(DataRow row : rows) {
    if(row.get("DATE_OF_BIRTH") != null) {
        Artist artist = dataContext.objectFromDataRow(Artist.class, row);
        // do something with Artist...
        ...
    }
}
```

Iterated Queries

While contemporary hardware may easily allow applications to fetch hundreds of thousands or even millions of objects into memory, it doesn't mean this is always a good idea to do so. You can optimize processing of very large result sets with two techniques discussed in this and the following chapter - iterated and paginated queries.

Iterated query is not actually a special query. Any selecting query can be executed in iterated mode by the DataContext (like in the previous example, a cast to DataContext is needed). DataContext returns an object called ResultIterator that is backed by an open ResultSet. Data is read from ResultIterator one row at a time until it is exhausted. Data comes as a DataRows regardless of whether the orginating query was configured to fetch DataRows or not. A ResultIterator must be explicitly closed to avoid JDBC resource leak.

Iterated query provides constant memory performance for arbitrarily large ResultSets. This is true at least on the Cayenne end, as JDBC driver may still decide to bring the entire ResultSet into the JVM memory.

Here is a full example:

```
// you need to cast ObjectContext to DataContext to get access to
'performIteratedQuery'
DataContext dataContext = (DataContext) context;

// create a regular query
SelectQuery q = new SelectQuery(Artist.class);

// ResultIterator operations all throw checked CayenneException
// moreover 'finally' is required to close it
try {

    ResultIterator it = dataContext.performIteratedQuery(q);

    try {
        while(it.hasNextRow()) {
            // normally we'd read a row, process its data, and throw it away
            // this gives us constant memory performance
            Map row = (Map) it.nextRow();

            // do something with the row...
            ...
        }
    }
    finally {
        it.close();
    }
}
catch(CayenneException e) {
   e.printStackTrace();
}
```

Also common sense tells us that ResultIterators should be processed and closed as soon as possible to release the DB connection. E.g. storing open iterators between HTTP requests and for unpredictable length of time would quickly exhaust the connection pool.

Paginated Queries

Enabling query pagination allows to load very large result sets in a Java app with very little memory overhead (much smaller than even the DataRows option discussed above). Moreover it is completely transparent to the application - a user gets what appears to be a list of Persistent objects - there's no iterator to close or DataRows to convert to objects:

```
SelectQuery query = new SelectQuery(Artist.class);
query.setPageSize(50);

// the fact that result is paginated is transparent
List<Artist> artists = ctxt.performQuery(query);
```

Having said that, DataRows option can be combined with pagination, providing the best of both worlds:

```
SelectQuery query = new SelectQuery(Artist.class);
query.setPageSize(50);
query.setFetchingDataRows(true);

List<DataRow> rows = ctxt.performQuery(query);
```

The way pagination works internally, it first fetches a list of IDs for the root entity of the query. This is very fast and initially takes very little memory. Then when an object is requested at an arbitrary index in the list, this object and adjacent objects (a "page" of objects that is determined by the query pageSize parameter) are fetched together by ID. Subsequent requests to the objects of this "page" are served from memory.

An obvious limitation of pagination is that if you eventually access all objects in the list, the memory use will end up being the same as with no pagination. However it is still a very useful approach. With some lists (e.g. multi-page search results) only a few top objects are normally accessed. At the same time pagination allows to estimate the full list size without fetching all the objects. And again - it is completely transparent and looks like a normal query.

Caching and Fresh Data

Object Caching

Query Result Caching

Turning off Synchronization of ObjectContexts

By default when a single ObjectContext commits its changes, all other contexts in the same runtime receive an event that contains all the committed changes. This allows them to update their cached object state to match the latest committed data. There are however many problems with this ostensibly helpful feature. In short - it works well in environments with few contexts and in unclustered scenarios, such as single user desktop applications, or simple webapps with only a few users. More specifically:

- The performance of synchronization is (probably worse than) O(N) where N is the number of peer ObjectContexts in the system. In a typical webapp N can be quite large. Besides for any given context, due to locking on synchronization, context own performance will depend not only on the queries that it runs, but also on external events that it does not control. This is unacceptable in most situations.
- Commit events are untargeted - even contexts that do not hold a given updated object will receive the full event that they will have to process.
- Clustering between JVMs doesn't scale - apps with large volumes of commits will quickly saturate the network with events, while most of those will be thrown away on the receiving end as mentioned above.
- Some contexts may not want to be refreshed. A refresh in the middle of an operation may lead

to unpredictable results.

- Synchronization will interfere with optimistic locking.

So we've made a good case for disabling synchronization in most webapps. To do that, set to "false" the following DI property - `Constants.SERVER_CONTEXTS_SYNC_PROPERTY`, using one of the standard Cayenne DI approaches. E.g. from command line:

```
$ java -Dcayenne.server.contexts_sync_strategy=false
```

Or by changing the standard properties Map in a custom extensions module:

```
public class MyModule implements Module {

    @Override
    public void configure(Binder binder) {
        binder.bindMap(Constants.PROPERTIES_MAP).put(Constants
.SERVER_CONTEXTS_SYNC_PROPERTY, "false");
    }
}
```

2.9. Customizing Cayenne Runtime

Dependency Injection Container

Cayenne runtime is built around a small powerful dependency injection (DI) container. Just like other popular DI technologies, such as Spring or Guice, Cayenne DI container manages sets of interdependent objects and allows users to configure them. These objects are regular Java objects. We are calling them "services" in this document to distinguish from all other objects that are not configured in the container and are not managed. DI container is responsible for service instantiation, injecting correct dependencies, maintaining service instances scope, and dispatching scope events to services.

The services are configured in special Java classes called "modules". Each module defines binding of service interfaces to implementation instances, implementation types or providers of implementation instances. There are no XML configuration files, and all the bindings are type-safe. The container supports injection into instance variables and constructor parameters based on the `@Inject` annotation. This mechanism is very close to Google Guice.

The discussion later in this chapter demonstrates a standalone DI container. But keep in mind that Cayenne already has a built-in Injector, and a set of default modules. A Cayenne user would normally only use the API below to write custom extension modules that will be loaded in that existing container when creating ServerRuntime. See "Starting and Stopping ServerRuntime" chapter for an example of passing an extension module to Cayenne.

Cayenne DI probably has ~80% of the features expected in a DI container and has no dependency on the rest of Cayenne, so in theory can be used as an application-wide DI engine. But it's primary

purpose is still to serve Cayenne. Hence there are no plans to expand it beyond Cayenne needs. It is an ideal "embedded" DI that does not interfere with Spring, Guice or any other such framework present elsewhere in the application.

DI Bindings API

To have a working DI container, we need three things: service interfaces and classes, a module that describes service bindings, a container that loads the module, and resolves the depedencies. Let's start with service interfaces and classes:

```
public interface Service1 {
    public String getString();
}
```

```
public interface Service2 {
    public int getInt();
}
```

A service implementation using instance variable injection:

```
public class Service1Impl implements Service1 {
    @Inject
    private Service2 service2;

    public String getString() {
        return service2.getInt() + "_Service1Impl";
    }
}
```

Same thing, but using constructor injection:

```
public class Service1Impl implements Service1 {

    private Service2 service2;

    public Service1Impl(@Inject Service2 service2) {
        this.service2 = service2;
    }

    public String getString() {
        return service2.getInt() + "_Service1Impl";
    }
}
```

```
public class Service2Impl implements Service2 {
    private int i;

    public int getInt() {
        return i++;
    }
}
```

Now let's create a module implementing `org.apache.cayenne.tutorial.di.Module` interface that will contain DI configuration. A module binds service objects to keys that are reference. Binder provided by container implements fluent API to connect the key to implementation, and to configure various binding options (the options, such as scope, are demonstrated later in this chapter). The simplest form of a key is a Java Class object representing service interface. Here is a module that binds Service1 and Service2 to corresponding default implementations:

```
public class Module1 implements Module {

    public void configure(Binder binder) {
        binder.bind(Service1.class).to(Service1Impl.class);
        binder.bind(Service2.class).to(Service2Impl.class);
    }
}
```

Once we have at least one module, we can create a DI container. `org.apache.cayenne.di.Injector` is the container class in Cayenne:

```
Injector injector = DIBootstrap.createInjector(new Module1());
```

Now that we have created the container, we can obtain services from it and call their methods:

```
Service1 s1 = injector.getInstance(Service1.class);
for (int i = 0; i < 5; i++) {
    System.out.println("S1 String: " + s1.getString());
}
```

This outputs the following lines, demonstrating that s1 was Service1Impl and Service2 injected into it was Service2Impl:

```
0_Service1Impl
1_Service1Impl
2_Service1Impl
3_Service1Impl
4_Service1Impl
```

There are more flavors of bindings:

```
// binding to instance - allowing user to create and configure instance
// inside the module class
binder.bind(Service2.class).toInstance(new Service2Impl());

// binding to provider - delegating instance creation to a special
// provider class
binder.bind(Service1.class).toProvider(Service1Provider.class);

// binding to provider instance
binder.bind(Service1.class).toProviderInstance(new Service1Provider());

// multiple bindings of the same type using Key
// injection can reference the key name in annotation:
// @Inject("i1")
// private Service2 service2;
binder.bind(Key.get(Service2.class, "i1")).to(Service2Impl.class);
binder.bind(Key.get(Service2.class, "i2")).to(Service2Impl.class);
```

Another types of confiuguration that can be bound in the container are lists and maps. They will be discussed in the following chapters.

Service Lifecycle

An important feature of the Cayenne DI container is instance scope. The default scope (implicitly used in all examples above) is "singleton", meaning that a binding would result in creation of only one service instance, that will be repeatedly returned from `Injector.getInstance(..)`, as well as injected into classes that declare it as a dependency.

Singleton scope dispatches a "BeforeScopeEnd" event to interested services. This event occurs before the scope is shutdown, i.e. when `Injector.shutdown()` is called. Note that the built-in Cayenne injector is shutdown behind the scenes when `ServerRuntime.shutdown()` is invoked. Services may register as listeners for this event by annotating a no-argument method with `@BeforeScopeEnd` annotation. Such method should be implemented if a service needs to clean up some resources, stop threads, etc.

Another useful scope is "no scope", meaning that every time a container is asked to provide a service instance for a given key, a new instance will be created and returned:

```
binder.bind(Service2.class).to(Service2Impl.class).withoutScope();
```

Users can also create their own scopes, e.g. a web application request scope or a session scope. Most often than not custom scopes can be created as instances of `org.apache.cayenne.di.spi.DefaultScope` with startup and shutdown managed by the application (e.g. singleton scope is a DefaultScope managed by the Injector) .

Overriding Services

Cayenne DI allows to override services already definied in the current module, or more commonly - some other module in the the same container. Actually there's no special API to override a service, you'd just bind the service key again with a new implementation or provider. The last binding for a key takes precedence. This means that the order of modules is important when configuring a container. The built-in Cayenne injector ensures that Cayenne standard modules are loaded first, followed by optional user extension modules. This way the application can override the standard services in Cayenne.

Customization Strategies

The previous section discussed how Cayenne DI works in general terms. Since Cayenne users will mostly be dealing with an existing Injector provided by ServerRuntime, it is important to understand how to build custom extensions to a preconfigured container. As shown in "Starting and Stopping ServerRuntime" chapter, custom extensions are done by writing an aplication DI module (or multiple modules) that configures service overrides. This section shows all the configuration possibilities in detail, including changing properties of the existing services, contributing services to standard service lists and maps, and overriding service implementations. All the code examples later in this section are assumed to be placed in an application module "configure" method:

```
public class MyExtensionsModule implements Module {
    public void configure(Binder binder) {
        // customizations go here...
    }
}
```

```
Module extensions = new MyExtensionsModule();
ServerRuntime runtime =
    new ServerRuntime("com/example/cayenne-mydomain.xml", extensions);
```

Changing Properties of Existing Services

Many built-in Cayenne services change their behavior based on a value of some environment property. A user may change Cayenne behavior without even knowing which services are responsible for it, but setting a specific value of a known property. Supported property names are listed in "Appendix A".

There are two ways to set service properties. The most obvious one is to pass it to the JVM with -D flag on startup. E.g.

```
$ java -Dcayenne.server.contexts_sync_strategy=false ...
```

A second one is to contribute a property to `org.apache.cayenne.configuration.DefaultRuntimeProperties.properties` map (see the next section

on how to do that). This map contains the default property values and can accept application-specific values, overrding the defaults.

Note that if a property value is a name of a Java class, when this Java class is instantiated by Cayenne, the container performs injection of instance variables. So even the dynamically specified Java classes can use @Inject annotation to get a hold of other Cayenne services.

If the same property is specified both in the command line and in the properties map, the command-line value takes precedence. The map value will be ignored. This way Cayenne runtime can be reconfigured during deployment.

Contributing to Service Collections

Cayenne can be extended by adding custom objects to named maps or lists bound in DI. We are calling these lists/maps "service collections". A service collection allows things like appending a custom strategy to a list of built-in strategies. E.g. an application that needs to install a custom DbAdapter for some database type may contribute an instance of custom DbAdapterDetector to a `org.apache.cayenne.configuration.server.DefaultDbAdapterFactory.detectors` list:

```
public class MyDbAdapterDetector implements DbAdapterDetector {
    public DbAdapter createAdapter(DatabaseMetaData md) throws SQLException {
        // check if we support this database and retun custom adapter
        ...
    }
}
```

```
// since build-in list for this key is a singleton, repeated
// calls to 'bindList' will return the same instance
binder.bindList(DefaultDbAdapterFactory.DETECTORS_LIST)
       .add(MyDbAdapterDetector.class);
```

Maps are customized using a similar `"bindMap"` method.

The names of built-in collections are listed in "Appendix B".

Alternative Service Implementations

As mentioned above, custom modules are loaded by ServerRuntime after the built-in modules. So it is easy to redefine a built-in service in Cayenne by rebinding desired implementations or providers. To do that, first we need to know what those services to redefine are. While we describe some of them in the following sections, the best way to get a full list is to check the source code of the Cayenne version you are using and namely look in `org.apache.cayenne.configuration.server.ServerModule` - the main built-in module in Cayenne.

Now an example of overriding `QueryCache` service. The default implementation of this service is provided by `MapQueryCacheProvider`. But if we want to use `EhCacheQueryCache` (a Cayenne wrapper for the EhCache framework), we can define it like this:

```
binder.bind(QueryCache.class).to(EhCacheQueryCache.class);
```

Noteworthy Built-in Services

JdbcEventLogger

`org.apache.cayenne.log.JdbcEventLogger` is the service that defines logging API for Cayenne internals. It provides facilities for logging queries, commits, transactions, etc. The default implementation is `org.apache.cayenne.log.Slf4jJdbcEventLogger` that performs logging via slf4j-api library. Cayenne library includes another potentially useful logger - `org.apache.cayenne.log.FormattedSlf4jJdbcEventLogger` that produces formatted multiline SQL output that can be easier to read.

DataSourceFactory

DataChannelFilter

QueryCache

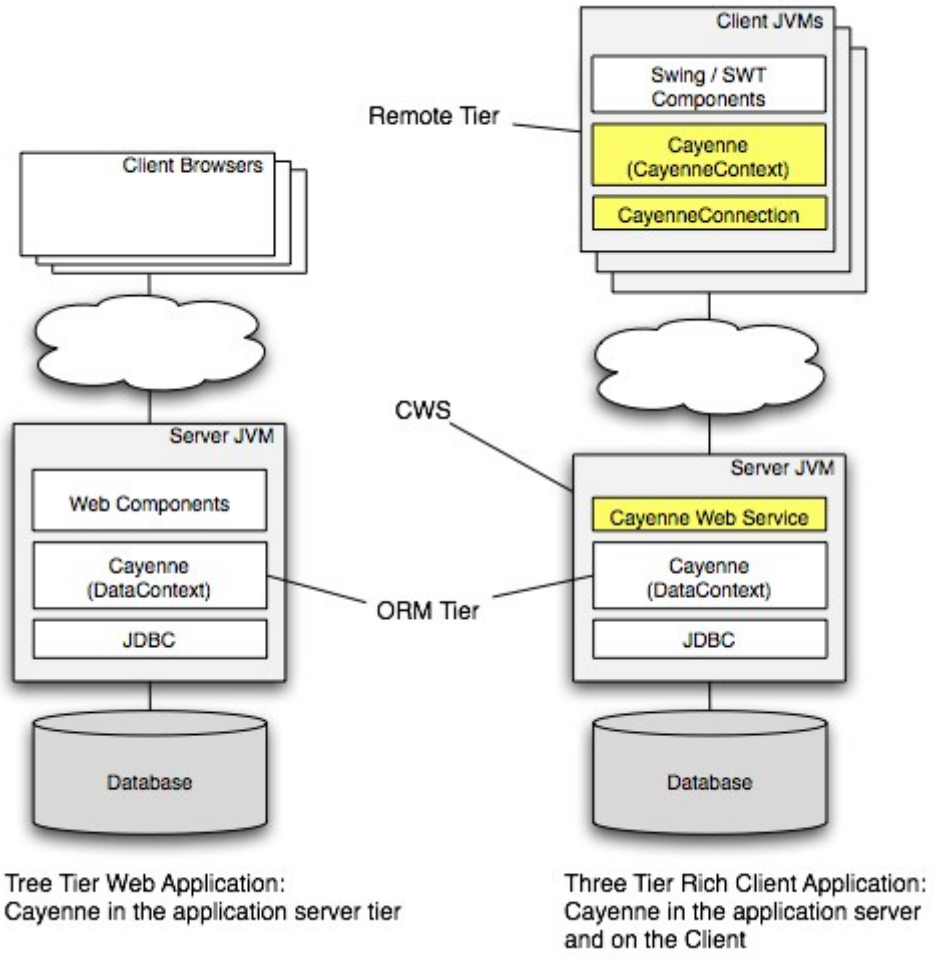
ExtebdedTypes

Chapter 3. Cayenne Framework - Remote Object Persistence

3.1. Introduction to ROP

What is ROP

"Remote Object Persistence" is a low-overhead web services-based technology that provides lightweight object persistence and query functionality to 'remote' applications. In other words it provides familiar Cayenne API to applications that do not have direct access to the database. Instead such applications would access Cayenne Web Service (CWS). A single abstract data model (expressed as Cayenne XML DataMap) is used on the server and on the client, while execution logic can be partitioned between the tiers.The following picture compares a regular Cayenne web application and a rich client application that uses remote object persistence technology:



Persistence stack above consists of the following parts:

- ORM Tier: a server-side Cayenne Java application that directly connects to the database via JDBC.
- CWS (Cayenne Web Service): A wrapper around an ORM tier that makes it accessible to remote CWS clients.

- Remote Tier (aka Client Tier): A Java application that has no direct DB connection and persists its objects by connecting to remote Cayenne Web Service (CWS). Note that CWS Client doesn't have to be a desktop application. It can be another server-side application. The word "client" means a client of Cayenne Web Service.

Main Features

- Unified approach to lightweight object persistence across multiple tiers of a distributed system.
- Same abstract object model on the server and on the client.
- Client can "bootstrap" from the server by dynamically loading persistence metadata.
- An ability to define client objects differently than the server ones, and still have seamless persistence.
- Generic web service interface that doesn't change when object model changes.
- An ability to work in two modes: dedicated session mode or shared ("chat") mode when multiple remote clients collaboratively work on the same data.
- Lazy object and collection faulting.
- Full context lifecycle
- Queries, expressions, local query caching, paginated queries.
- Validation
- Delete Rules

3.2. Implementing ROP Client

System Requirements

Jar Files and Dependencies

3.3. Implementing ROP Server

3.4. Implementing ROP Client

3.5. ROP Deployment

Deploying ROP Server

Recent versions of Tomcat and Jetty containers (e.g. Tomcat 6 and 7, Jetty 8) contain code addressing a security concern related to "session fixation problem" by resetting the existing session ID of any request that requires BASIC authentcaition. If ROP service is protected with declarative security (see the ROP tutorial and the following chapters on security), this feature prevents the ROP client from attaching to its session, resulting in MissingSessionExceptions. To solve that you will need to either switch to an alternative security mechanism, or disable "session fixation problem" protections of the container. E.g. the later can be achieved in Tomcat 7 by adding the following context.xml file to the webapp's META-INF/ directory:

```
<Context>
    <Valve className="org.apache.catalina.authenticator.BasicAuthenticator"
           changeSessionIdOnAuthentication="false" />
</Context>
```

(The <Valve> tag can also be placed within the <Context> in any other locations used by Tomcat to load context configurations)

Deploying ROP Client

Security

3.6. Current Limitations

Chapter 4. Appendix A. Configuration Properties

Note that the property names below are defined as constants in `org.apache.cayenne.configuration.Constants` interface.

- `cayenne.jdbc.driver[.domain_name.node_name]` defines a JDBC driver class to use when creating a DataSource. If domain name and optionally - node name are specified, the setting overrides DataSource info just for this domain/node. Otherwise the override is applied to all domains/nodes in the system.
    - Default value: none, project DataNode configuration is used
- `cayenne.jdbc.url[.domain_name.node_name]` defines a DB URL to use when creating a DataSource. If domain name and optionally - node name are specified, the setting overrides DataSource info just for this domain/node. Otherwise the override is applied to all domains/nodes in the system.
    - Default value: none, project DataNode configuration is used
- `cayenne.jdbc.username[.domain_name.node_name]` defines a DB user name to use when creating a DataSource. If domain name and optionally - node name are specified, the setting overrides DataSource info just for this domain/node. Otherwise the override is applied to all domains/nodes in the system.
    - Possible values: any
    - Default value: none, project DataNode configuration is used
- `cayenne.jdbc.password[.domain_name.node_name]` defines a DB password to use when creating a DataSource. If domain name and optionally - node name are specified, the setting overrides DataSource info just for this domain/node. Otherwise the override is applied to all domains/nodes in the system
    - Default value: none, project DataNode configuration is used
- `cayenne.jdbc.min_connections[.domain_name.node_name]` defines the DB connection pool minimal size. If domain name and optionally - node name are specified, the setting overrides DataSource info just for this domain/node. Otherwise the override is applied to all domains/nodes in the system
    - Default value: none, project DataNode configuration is used
- `cayenne.jdbc.max_connections[.domain_name.node_name]` defines the DB connection pool maximum size. If domain name and optionally - node name are specified, the setting overrides DataSource info just for this domain/node. Otherwise the override is applied to all domains/nodes in the system
    - Default value: none, project DataNode configuration is used
- `cayenne.querycache.size` An integer defining the maximum number of entries in the query cache. Note that not all QueryCache providers may respect this property. MapQueryCache uses it, but the rest would use alternative configuration methods.
    - Possible values: any positive int value
    - Default value: 2000

- `cayenne.server.contexts_sync_strategy` defines whether peer ObjectContexts should receive snapshot events after commits from other contexts. If true (*default*), the contexts would automatically synchronize their state with peers.
    - Possible values: true, false
    - Default value: true
- `cayenne.server.object_retain_strategy` defines fetched objects retain strategy for ObjectContexts. When weak or soft strategy is used, objects retained by ObjectContext that have no local changes can potentially get garbage collected when JVM feels like doing it.
    - Possible values: weak, soft, hard
    - Default value: weak
- `cayenne.server.max_id_qualifier_size` defines a maximum number of ID qualifiers in the WHERE clause of queries that are generated for paginated queries and for DISJOINT_BY_ID prefetch processing. This is needed to avoid hitting WHERE clause size limitations and memory usage efficiency.
    - Possible values: any positive int
    - Default value: 10000
- `cayenne.rop.service_url` defines the URL of the ROP server
    - Default value: none
- `cayenne.rop.service_username` defines the user name for an ROP client to login to an ROP server.
    - Default value: none
- `cayenne.rop.service_password` defines the password for an ROP client to login to an ROP server.
    - Default value: none
- `cayenne.rop.service.timeout` a value in milliseconds for the ROP client-server connection read operation timeout
    - Possible values: any positive long value
    - Default value: none
- `cayenne.rop.shared_session_name` defines the name of the shared session that an ROP client wants to join on an ROP server. If omitted, a dedicated session is created.
    - Default value: none
- `cayenne.rop.channel_events` defines whether client-side DataChannel should dispatch events to child ObjectContexts. If set to true, ObjectContexts will receive commit events and merge changes committed by peer contexts that passed through the common client DataChannel.
    - Possible values: true, false
    - Default value: false
- `cayenne.rop.context_change_events` defines whether object property changes in the client context result in firing events. Client UI components can listen to these events and update the UI. Disabled by default.
    - Possible values: true, false

- Default value: false

- `cayenne.rop.context_lifecycle_events` defines whether object commit and rollback operations in the client context result in firing events. Client UI components can listen to these events and update the UI. Disabled by default.
  - Possible values: true,false
  - Default value: false
- `cayenne.server.rop_event_bridge_factory` defines the name of the `org.apache.cayenne.event.EventBridgeFactory` that is passed from the ROP server to the client. I.e. server DI would provide a name of the factory, passing this name to the client via the wire. The client would instantiate it to receive events from the server. Note that this property is stored in `cayenne.server.rop_event_bridge_properties` map, not in the main `cayenne.properties`.
  - Default value: false

Chapter 5. Appendix B. Service Collections

Note that the collection keys below are defined as constants in `org.apache.cayenne.configuration.Constants` interface.

Table 8. Service Collection Keys Present in ServerRuntime and/or ClientRuntime

Collection Property	Type	Description
`cayenne.properties`	`Map<String,String>`	Properties used by built-in Cayenne services. The keys in this map are the property names from the table in Appendix A. Separate copies of this map exist on the server and ROP client.
`cayenne.server.adapter_detectors`	`List<DbAdapterDetector>`	Contains objects that can discover the type of current database and install the correct DbAdapter in runtime.
`cayenne.server.domain_filters`	`List<DataChannelFilter>`	Stores DataDomain filters.
`cayenne.server.project_locations`	`List<String>`	Stores locations of the one of more project configuration files.
`cayenne.server.default_types`	`List<ExtendedType>`	Stores default adapter-agnostic ExtendedTypes. Default ExtendedTypes can be overridden / extended by DB-specific DbAdapters as well as by user-provided types configured in another colltecion (see `"cayenne.server.user_types"`).
`cayenne.server.user_types`	`List<ExtendedType>`	Stores a user-provided ExtendedTypes. This collection will be merged into a full list of ExtendedTypes and would override any ExtendedTypes defined in a default list, or by a DbAdapter.
`cayenne.server.type_factories`	`List<ExtendedTypeFactory>`	Stores default and user-provided ExtendedTypeFactories. ExtendedTypeFactory allows to define ExtendedTypes dynamically for the whole group of Java classes. E.g. Cayenne supplies a factory to map all Enums regardless of their type.

Collection Property	Type	Description
`cayenne.server.rop_event_bridge_properties`	`Map<String, String>`	Stores event bridge properties passed to the ROP client on bootstrap. This means that the map is configured by server DI, and passed to the client via the wire. The properties in this map are specific to EventBridgeFactory implementation (e.g JMS or XMPP connection prameters). One common property is `"cayenne.server.rop_event_bridge_factory"` that defines the type of the factory.

Chapter 6. Appendix C. Expressions BNF

```
TOKENS
<DEFAULT> SKIP : {
" "
| "\t"
| "\n"
| "\r"
}

<DEFAULT> TOKEN : {
<NULL: "null" | "NULL">
| <TRUE: "true" | "TRUE">
| <FALSE: "false" | "FALSE">
}

<DEFAULT> TOKEN : {
<PROPERTY_PATH: <IDENTIFIER> ("." <IDENTIFIER>)*>
}

<DEFAULT> TOKEN : {
<IDENTIFIER: <LETTER> (<LETTER> | <DIGIT>)* (["+"])?>
| <#LETTER: ["_","a"-"z","A"-"Z"]>
| <#DIGIT: ["0"-"9"]>
}

/**
 * Quoted Strings, whose object value is stored in the token manager's
 * "literalValue" field. Both single and double qoutes are allowed
 */<DEFAULT> MORE : {
"\'" : WithinSingleQuoteLiteral
| "\"" : WithinDoubleQuoteLiteral
}

<WithinSingleQuoteLiteral> MORE : {
<ESC: "\\" (["n","r","t","b","f","\\","\'","`","\""] | (["0"-"3"])? ["0"-"7"] (["0"-
"7"])?)> : {
| <~["\'","\\"]> : {
}

<WithinSingleQuoteLiteral> TOKEN : {
<SINGLE_QUOTED_STRING: "\'"> : DEFAULT
}

<WithinDoubleQuoteLiteral> MORE : {
<STRING_ESC: <ESC>> : {
| <~["\"","\\"]> : {
}
```

```
<WithinDoubleQuoteLiteral> TOKEN : {
<DOUBLE_QUOTED_STRING: "\""> : DEFAULT
}

/**
 * Integer or real Numeric literal, whose object value is stored in the token
manager's
 * "literalValue" field.
 */<DEFAULT> TOKEN : {
<INT_LITERAL: ("0" (["0"-"7"])* | ["1"-"9"] (["0"-"9"])* | "0" ["x","X"] (["0"-
"9","a"-"f","A"-"F"])+)
        (["l","L","h","H"])?> : {
| <FLOAT_LITERAL: <DEC_FLT> (<EXPONENT>)? (<FLT_SUFF>)? | <DEC_DIGITS> <EXPONENT>
(<FLT_SUFF>)?
| <DEC_DIGITS> <FLT_SUFF>> : {
| <#DEC_FLT: (["0"-"9"])+ "." (["0"-"9"])* | "." (["0"-"9"])+>
| <#DEC_DIGITS: (["0"-"9"])+>
| <#EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+>
| <#FLT_SUFF: ["d","D","f","F","b","B"]>
}

NON-TERMINALS
    expression    :=    orCondition <EOF>
    orCondition    :=    andCondition ( "or" andCondition )*
    andCondition    :=    notCondition ( "and" notCondition )*
    notCondition    :=    ( "not" | "!" ) simpleCondition
        |    simpleCondition
    simpleCondition    :=    <TRUE>
        |    <FALSE>
        |    scalarConditionExpression
             ( simpleNotCondition
               | ( "=" | "==" ) scalarExpression
               | ( "!=" | "<>" ) scalarExpression
               | "<=" scalarExpression
               | "<" scalarExpression | ">" scalarExpression
               | ">=" scalarExpression
               | "like" scalarExpression
               | "likeIgnoreCase" scalarExpression
               | "in" ( namedParameter | "(" scalarCommaList ")" )
               | "between" scalarExpression "and" scalarExpression
             )?
    simpleNotCondition    :=    ( "not" | "!" )
             ( "like" scalarExpression
               | "likeIgnoreCase" scalarExpression
               | "in" ( namedParameter | "(" scalarCommaList ")" )
               | "between" scalarExpression "and" scalarExpression
             )
    scalarCommaList    :=    ( scalarConstExpression ( "," scalarConstExpression )* )
    scalarConditionExpression    :=    scalarNumericExpression
        |    <SINGLE_QUOTED_STRING>
        |    <DOUBLE_QUOTED_STRING>
```

```
        |    <NULL>
    scalarExpression    :=    scalarConditionExpression
        |    <TRUE>
        |    <FALSE>
    scalarConstExpression    :=    <SINGLE_QUOTED_STRING>
        |    <DOUBLE_QUOTED_STRING>
        |    namedParameter
        |    <INT_LITERAL>
        |    <FLOAT_LITERAL>
        |    <TRUE>
        |    <FALSE>
    scalarNumericExpression    :=    multiplySubtractExp
             ( "+" multiplySubtractExp | "-" multiplySubtractExp )*
    multiplySubtractExp    :=    numericTerm ( "*" numericTerm | "/" numericTerm )*
    numericTerm    :=    ( "+" )? numericPrimary
        |    "-" numericPrimary
    numericPrimary    :=    "(" orCondition ")"
        |    pathExpression
        |    namedParameter
        |    <INT_LITERAL>
        |    <FLOAT_LITERAL>
    namedParameter    :=    "$" <PROPERTY_PATH>
    pathExpression    :=    ( <PROPERTY_PATH>
                            | "obj:" <PROPERTY_PATH>
                            | "db:" <PROPERTY_PATH>
                            | "enum:" <PROPERTY_PATH>  )
```

Chapter 7. List of tables