Cayenne Guide

Version 5.0 (5.0-RC1)

Table of Contents

1. System Requirements . . . 2
2. Cayenne Mapping Structure . . . 3
   - Cayenne Project . . . 3
   - DataMap . . . 3
   - DataNode . . . 3
   - DbEntity . . . 4
   - ObjEntity . . . 4
   - Embeddable . . . 4
   - Procedure . . . 4
   - Query . . . 4
3. CayenneModeler Application . . . 5
   - Running CayenneModeler . . . 5
   - Reverse Engineering in Cayenne Modeler . . . 6
   - Generating Java Classes . . . 7
   - Modeling Generic Persistent Classes . . . 7
   - Modeling Primary Key Generation Strategy . . . 8
4. Including Cayenne in a Project . . . 9
   - Maven . . . 9
   - Gradle . . . 9
   - Ant, etc. . . . 9
5. Starting Cayenne . . . 10
   - Starting and Stopping CayenneRuntime . . . 10
   - Connecting to a Database . . . 11
   - Merging Multiple Projects . . . 11
6. Persistent Objects and ObjectContext . . . 13
   - ObjectContext . . . 13
   - Persistent Object and its Lifecycle . . . 13
   - ObjectContext Persistence API . . . 14
   - Cayenne Helper Class . . . 16
   - ObjectContext Nesting . . . 17
   - Generic Persistent Objects . . . 17
   - Transactions . . . 18
7. Expressions . . . 20
   - Path Expressions . . . 20
   - Creating Expressions from Strings . . . 21
   - Creating Expressions via API . . . 23
   - Evaluating Expressions in Memory . . . 24
8. Orderings . . . 26

9. Queries . . . 27
ObjectSelect . . . 27
Object Query Language . . . 31
SQLSelect and SQLExec . . . 37
Scripting SQL Queries . . . 38
MappedSelect and MappedExec . . . 43
ProcedureCall . . . 43
Custom Queries . . . 44
10. Lifecycle Events . . . 46
Types of Lifecycle Events . . . 46
Callbacks on Persistent Objects . . . 47
Callbacks on Non-Persistent Listeners . . . 48
Combining Listeners with DataChannel filters . . . 50
11. Commit Log . . . 52
Basic Usage . . . 52
Restricting to Annotated Entities . . . 53
Dispatching Events Outside the Transaction . . . 53
Custom Entity Factory . . . 54
12. Performance Tuning . . . 55
Prefetching . . . 55
Data Rows . . . 57
Iterated Queries . . . 58
Paginated Queries . . . 59
Caching and Fresh Data . . . 59
Turning off Synchronization of ObjectContexts . . . 60
13. Customizing Cayenne Runtime . . . 62
Dependency Injection Container . . . 62
Customization Strategies . . . 65
Using custom data types . . . 67
Noteworthy Built-in Services . . . 71
14. Agentic Coding . . . 72
Overview . . . 72
`apache-cayenne` Claude Code Plugin . . . 72
Cayenne MCP Server . . . 72
15. Additional Modules . . . 76
Cache Invalidation Extension . . . 76
Crypto extension . . . 77
JCache integration . . . 79
Apache Velocity Extension . . . 80
16. Build Tools . . . 82
Maven Plugin . . . 82

Gradle Plugin . . . 90
Ant Tasks . . . 93
DB Import Filtering . . . 94
Other DB Import Settings . . . 101
Appendix A. Configuration Properties . . . 103
Appendix B. Service Collections . . . 105

Copyright © 2011-2026 Apache Software Foundation and individual authors

License

Licensed to the Apache Software Foundation (ASF) under one or more contributor license agreements. See the NOTICE file distributed with this work for additional information regarding copyright ownership. The ASF licenses this file to you under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at https://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Chapter 1. System Requirements

- Java: Cayenne runtime framework and CayenneModeler GUI tool are written in 100% Java, and run on any Java-compatible platform. Minimal required JDK version depends on the version of Cayenne you are using, as shown in the following table:

Table 1. Cayenne Version History

Cayenne Version	Java Version	Status
5.0	Java 21 or newer	Development
4.2	Java 1.8 or newer	Stable
4.1	Java 1.8 — Java 17	Stable
4.0	Java 1.7 — Java 11	Aging
3.0 / 3.1	Java 1.5 — Java 1.8	Legacy
1.2 / 2.0	Java 1.4	Legacy
1.1	Java 1.3	Legacy

- JDBC Driver: An appropriate DB-specific JDBC driver is needed to access the database.
- 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.

Chapter 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 various 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 a model of a database. It is actually pretty simple. It has a name and a JDBC DataSource. Unlike the rest of the objects discussed here, DataNodes are not a part of the XML mapping project. They are defined in the code, when Cayenne runtime is created (see Starting Cayenne). Most applications 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.

Chapter 3. CayenneModeler Application

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 https://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
```

A path to an existing Cayenne project file can be passed as the only argument to open the project on startup:

```
$ java -jar CayenneModeler.jar path/to/cayenne-project.xml
```

On macOS the same applies to the `.app` bundle:

```
$ open CayenneModeler.app --args path/to/cayenne-project.xml
```

Running from Maven

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, and the version of the Modeler always matches the version of the framework. Add the `cayenne-modeler` dependency and an `exec-maven-plugin` execution to the project POM:

```
<dependencies>
    <dependency>
        <groupId>org.apache.cayenne.modeler</groupId>
        <artifactId>cayenne-modeler</artifactId>
        <version>5.0-RC1</version>
    </dependency>
</dependencies>
```

```
<build>
    <plugins>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>exec-maven-plugin</artifactId>
            <version>3.5.0</version>
            <executions>
                <execution>
                    <id>run-modeler</id>
                    <goals>
                        <goal>java</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <mainClass>org.apache.cayenne.modeler.Main</mainClass>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Then run:

```
$ mvn exec:java@run-modeler
```

Reverse Engineering in Cayenne Modeler

Alternative approach to using build tools is doing reverse engineering from CayenneModeler.

You can find the reverse engineering tool on the **DB Import** tab of a DataMap.

Reverse engineering options

[re modeler reverseengineering dialog] | *../../images/re-modeler-reverseengineering-dialog.png*

DB Import tab.

Here is a list of options to tune what will be processed by reverse engineering:

- **Add Catalog**
- **Add Schema**
- **Include Table**
- **Exclude Table**
- **Include Column**
- **Exclude Column**
- **Include Procedure**

- **Exclude Procedure**

More options are available after clicking **Advanced Options**:

- **Naming Strategy**: A strategy that generates the names of ObjEntities, attributes and relationships.
- **Meaningful PK**: Comma separated list of RegExp's for tables that you want to have meaningful primary keys. By default no meaningful PKs are created.
- **Strip From Table Names**: Regex that matches the part of the table name that needs to be stripped off generating ObjEntity name.
- **Table Types**: Table types to import. By default these are TABLE and VIEW.
- **Skip Relationships**: Whether to load relationships.
- **Skip Primary Keys**: Whether to load primary keys.
- **Force DataMap Catalog**: will set DbEntity catalog to one in the DataMap.
- **Force DataMap Schema**: will set DbEntity schema to one in the DataMap.
- **Use Java 7 Types**: Use **java.util.Date** for all columns with **DATE/TIME/TIMESTAMP** types. By default **java.time.** types will be used.

DataSource selection

Click `Configure Connection` to set a DataSource (`Run Import` would also ask for it, if there is none yet). You can pick one of the saved DataSources, or enter the connection settings right in the dialog.

[re modeler datasource select] | *../../images/re-modeler-datasource-select.png*

DataSource configuration dialog.

Then click `Continue`. Once the connection is configured, `Run Import` starts the import.

Generating Java Classes

Before using Cayenne in you code you need to generate java source code for persistent objects. This can be done with Modeler GUI or via cgen maven/ant plugin.

To generate classes in the Modeler select a DataMap, go to the **Class Generation** tab and click **Generate**.

The default settings are suitable for almost all cases. Change them only if you know what exactly you need to customize. The "Advanced options" and "Templates options" on the same tab let you control almost all aspects of code generation, including custom templates for Java code. See default Cayenne templates on GitHub as an example.

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

completely generic class to any number of entities. The only expectation is that a generic class implements `org.apache.cayenne.Persistent`. So an ideal candidate for a generic class is `GenericPersistentObject`, or some custom subclass of `GenericPersistentObject`.

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.GenericPersistentObject` is used.

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

Modeling Primary Key Generation Strategy

Cayenne supports three PK generation strategies:

1. **Cayenne Generated**. This is default strategy. Cayenne will use special table `AUTO_PK_SUPPORT` (or the DB sequences on the databases that support them) for managing primary keys.
2. **Database Generated**. Cayenne will delegate PK generation to database (e.g. auto increment fields on MySQL or `serial` type on PostgreSQL). To use it, check "Auto-Increment" for the PK column on the "Properties" tab of a DbEntity.
3. **Custom Sequence**. In this case Cayenne will use provided sequence to generate primary keys. To use it, check "Custom PK Sequence" on the "DbEntity" tab and enter the sequence name. Custom sequences are supported on Oracle and PostgreSQL.

Strategy should be set per each `DbEntity` independently.

[db entity pk] | *../../images/db-entity-pk.png*

Chapter 4. Including Cayenne in a Project

Maven

To add Cayenne to your Maven project, include `cayenne` in your POM:

```
<dependency>
   <groupId>org.apache.cayenne</groupId>
   <artifactId>cayenne</artifactId>
   <version>5.0-RC1</version>
</dependency>
```

Gradle

To add Cayenne to your Gradle project, include `cayenne` module:

```
compile 'org.apache.cayenne:cayenne:5.0-RC1'
```

Ant, etc.

If your environment requires manual dependency management (like Ant), check `lib` and `lib/third-party` folders of Cayenne distribution. It contains all Cayenne jars as well as the minimal set of third-party libraries to get you started.

Chapter 5. Starting Cayenne

Starting and Stopping CayenneRuntime

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

```
CayenneRuntime runtime = CayenneRuntime.of()
        .addConfig("com/example/cayenne-project.xml")
        .build();
```

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).

CayenneRuntime encapsulates a single Cayenne stack. Most applications will just have one CayenneRuntime using it to create as many ObjectContexts as needed, access the Dependency Injection (DI) container and work with other Cayenne features. Internally CayenneRuntime 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:

```
Module extensions = binder ->
        CoreModule.extend(binder).setProperty(RuntimeProperties.EXTERNAL_TX_PROPERTY,
"true");

CayenneRuntime runtime = CayenneRuntime.of()
        .addConfig("com/example/cayenne-project.xml")
        .addModule(extensions)
        .build();
```

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.

Connecting to a Database

The mapping project does not contain any database connection information. The database is defined in the code, when the runtime is created, as a DataNode. In the simplest case a DataNode is created from a `DataSource`. It can be any `DataSource`, e.g., the one provided by your application container, or the one built with `CayenneDataSource`:

```
DataSource dataSource = CayenneDataSource.of("jdbc:hsqldb:mem:starting")
        .userName("sa")
        .password("")
        .pool(1, 5)
        .build();

CayenneRuntime runtime = CayenneRuntime.of()
        .addConfig("cayenne-project.xml")
        .defaultDataNode(dataSource)
        .build();
```

When a DataNode needs to be customized, e.g., to let Cayenne create the DB schema on the first connection, to set a specific DbAdapter, or to give the node a name, use a `DataNodeDescriptor`:

```
DataNodeDescriptor node = DataNodeDescriptor.of("node1")
        .dataSource(dataSource)
        .createSchemaIfNeeded()
        .build();

CayenneRuntime runtime = CayenneRuntime.of()
        .addConfig("cayenne-project.xml")
        .defaultDataNode(node)
        .build();
```

There can be more than one DataNode. Each one is linked to its DataMaps by name. The "default" DataNode is optional, and is used for all the DataMaps that are not linked to a node explicitly:

```
CayenneRuntime runtime = CayenneRuntime.of()
        .addConfig("cayenne-project.xml")
        .addDataNode(ds1, "docs")
        .defaultDataNode(ds2)
        .build();
```

Merging Multiple Projects

CayenneRuntime 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 independently 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

CayenneRuntime builder:

```
CayenneRuntime runtime = CayenneRuntime.of()
        .addConfig("com/example/cayenne-project.xml")
        .addConfig("org/foo/cayenne-library1.xml")
        .addConfig("org/foo/cayenne-library2.xml")
        .build();
```

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.
- DataNodes are not a part of the projects. After the merge the DataMaps are linked to the DataNodes as described above. If any DataMaps are not explicitly linked to DataNodes, their queries will be executed via the default DataNode. This makes it possible to build mapping "libraries" that are only associated with a specific database in runtime.

Chapter 6. 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 CayenneRuntime:

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

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 multiple 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 they decide to commit them or roll them back). Even for a single user it might make sense to use multiple ObjectContexts (e.g. request-scoped contexts to allow concurrent requests from the browser that change and commit objects independently).

ObjectContext does not permanently hold 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 the `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.

The `Persistent` interface provides access to three persistence-related properties - *objectId*, *persistenceState* and *objectContext*. All three are initialized by the Cayenne runtime framework. Your 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 a quick equality check of two 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 2. 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.
COMMITTED	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 COMMITTED.
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:

```
List<Artist> artists = ObjectSelect.query(Artist.class)
        .select(context);
```

We'll discuss queries in some detail in the Queries chapter. The example above is self-explanatory - we create a `ObjectSelect` that matches all `Artist` objects present in the database, and then use `select` to get the result.

Some queries can be quite complex, returning multiple result sets or even updating the database. For such queries, ObjectContext provides the `execute()` method. While not commonly used, it is nevertheless important in some situations. E.g.:

```
List<QueryResult> result = context.execute(query);
for (QueryResult item : result) {
    switch (item) {
        case SelectResult<?> select -> process(select.objects());
        case UpdateResult update -> process(update.counts());
        case ResultIterator<?> iterator -> process(iterator);
        case OutParametersResult out -> process(out.values());
    }
```

```
}
```

When multiple such queries need to run together, execute them inside an explicit transaction:

```
Collection<Query> queries = List.of( // multiple queries that need to be run together
        SQLExec.query("DELETE FROM PAINTING"),
        SQLExec.query("DELETE FROM ARTIST"));

runtime.performInTransaction(() -> {
    queries.forEach(context::execute);
    return null;
});
```

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 only exists 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.

The `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 application needs to inspect mapping metadata. This information is stored in the `EntityResolver` object, accessible via the `ObjectContext`:

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

Here we discussed the most commonly-used subset of the ObjectContext API. There are other useful methods, e.g. those allowing you 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 the 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);
```

The reverse operation - finding an object given a known PK - is a part of the ObjectContext API. It returns an object already registered in the context without a query, and consults the caches before going to the database:

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

For more flexibility, you could select by ID with `ObjectSelect` instead.

Feel free to explore the `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 a 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 all need to 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 levels, as deep hierarchies will almost 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.

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

```
ObjectContext parent = runtime.newContext();
ObjectContext nested = runtime.newContext(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 convenience 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.

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

When creating a generic object, either use `ObjectContext.newObject(String)`, that takes an entity name, or provide your object with an explicit ObjectId:

```
Persistent generic = context.newObject("GenericEntity");
```

```
Persistent generic = new GenericPersistentObject();
generic.setObjectId(ObjectId.of("GenericEntity"));
context.registerNewObject(generic);
```

ObjectSelect for a generic object should be created by passing the entity name String, instead of just a Java class:

```
ObjectSelect<Persistent> query = ObjectSelect.query(Persistent.class, "GenericEntity"
);
```

Use Persistent 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 the 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- and application-managed transactions.

If you are using Spring, EJB or another environment that manages transactions, you'll likely need to switch the Cayenne runtime into "external transactions mode". This is done by setting the DI configuration property defined in `RuntimeProperties.EXTERNAL_TX_PROPERTY` (see Appendix A). If this property is set to "true", 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 roll back 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 via the `CayenneRuntime.performInTransaction` method:

```
Integer result = runtime.performInTransaction(() -> {
    // commit one or more contexts
    context1.commitChanges();
    context2.commitChanges();

    // after changing some objects in context1, commit again
    context1.newObject(Artist.class).setName("a3");
    context1.commitChanges();

    // return an arbitrary result or null if we don't care about the result
    return 5;
});
```

When inside a transaction, current thread Transaction object can be accessed via a static method:

```
Transaction tx = BaseTransaction.getThreadTransaction();
```

You can control transaction isolation level and propagation logic using `TransactionDescriptor`.

```
TransactionDescriptor descriptor = TransactionDescriptor.builder()
        .isolation(Connection.TRANSACTION_SERIALIZABLE)
        .propagation(TransactionPropagation.REQUIRES_NEW)
        .build();
runtime.performInTransaction(transactionalOperation, descriptor);
```

Chapter 7. Expressions

Cayenne provides a simple, yet powerful, object-based expression language. The most common uses 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 an 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. The general structure of path expressions is the following:

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

- `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. The path must have at least one segment; segments are separated by dot (".").
- `#alias` is an optional alias of a relationship segment, discussed below.
- `` 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 a value in the "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. An alias is appended to a relationship segment after the `#` character:

- `artist.exhibits.closingDate` - a path with no aliases
- `artist.exhibits#e.closingDate` - alias "e" is used for `artist.exhibits`.

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 the API from the following section for expression creation, we'll start by showing String expressions, as this will help you understand the semantics. A Cayenne expression can be represented as a String, which can be converted to an expression object using the `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 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 say 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:

```
Expression e1 = ExpressionFactory.exp("name = 'ABC'");

// double quotes are escaped inside Java Strings of course
Expression e2 = ExpressionFactory.exp("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). 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%'
```

Parameters. Expressions can contain named parameters (names that start with "$") that can be substituted with values either by name or by position. Parameterized expressions let you 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 `Persistent`, an `ObjectId`), parameterizing the expression is the only way to represent it as String. Here are examples of both positional and named parameter bindings:

```
Expression template = ExpressionFactory.exp("name = $name");

// name binding
Map<String, Object> p1 = Map.of("name", "Salvador Dali");
Expression qualifier1 = template.params(p1);

// positional binding
```

```
Expression qualifier2 = template.paramsArray("Monet");
```

Positional binding is usually shorter. You can pass positional bindings to the `exp(..)` factory method (its second argument is a params vararg):

```
Expression qualifier = ExpressionFactory.exp("name = $name", "Monet");
```

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

```
Expression qualifier = ExpressionFactory.exp("name like $name", "Salvador%");
```

When matching on a relationship, the value parameter must be either a Persistent object, an `org.apache.cayenne.ObjectId`, or a numeric ID value (for single column IDs). E.g.:

```
Artist dali = dali(); // assume we fetched this one already
Expression qualifier = ExpressionFactory.exp("artist = $artist", dali);
```

When you use positional binding, Cayenne expects values for all parameters to be present. Binding by name offers extra flexibility: sub-expressions with uninitialized parameters are automatically pruned from the expression. So, e.g., if certain parts of the expression criteria are not provided to the application, you can still build a valid expression

```
Expression template = ExpressionFactory.exp("name like $name and dateOfBirth > $date"
);

Map<String, Object> p1 = Map.of("name", "Salvador%");
Expression qualifier1 = template.params(p1);

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

Null handling. Handling of Java nulls as operands is no different handling 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.

Creating Expressions via API

Creating expressions from Strings is a powerful and dynamic approach, however a safer alternative is to use the Java API. It provides compile-time checking of the expression's validity. The API in question is provided by the `ExpressionFactory` class (that we've seen already), the Property class and the Expression class itself. `ExpressionFactory` contains a number of self-explanatory static

methods that can be used to build expressions. E.g.:

```
// 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);
```



The last line in the example above shows how to create a new expression by "chaining" two other expressions. 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. Expression API treats existing expressions as immutable.

As discussed earlier, Cayenne supports aliases in path Expressions, so you can control how SQL joins are generated if the same path is encountered more than once in the same Expression. Two `ExpressionFactory.matchAllExp(..)` methods (one taking a `Collection` of values, another - a vararg) let you implicitly generate aliases to "split" match paths into individual joins if needed:

```
// matches the artists that have both "P1" and "P2" paintings
Expression e = ExpressionFactory.matchAllExp("|paintings.title", "P1", "P2");
```

The "Path" argument to both of these methods can use a split character (a pipe symbol '|') instead of a dot to indicate that the 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. The split must always precede a relationship. E.g. `"|exhibits.paintings"`, `"exhibits|paintings"`, etc. Internally, Cayenne generates distinct aliases for each of the split expressions, forcing separate joins.

While ExpressionFactory is pretty powerful, there's an even easier way to create an expression using static Property objects generated by Cayenne for each persistent class. Some examples:

```
// Artist.NAME is generated by Cayenne and has a type of StringProperty<String>
Expression e1 = Artist.NAME.eq("Pablo");

// Chaining multiple properties into a path.
// Painting.ARTIST is generated by Cayenne and has a type of EntityProperty<Artist>
Expression e2 = Painting.ARTIST.dot(Artist.NAME).eq("Pablo");
```

Property objects provide the API mostly analogous to ExpressionFactory, though it is significantly shorter and is aware of the value types. It provides compile-time checks of both property names and types of arguments in conditions. We will use Property-based API in further examples.

Evaluating Expressions in Memory

When used in a query, an expression is converted to a SQL WHERE 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, and in-memory filtering.

Checking whether an object satisfies an expression:

```
Expression e = Artist.NAME.in("Dali", "Monet");
Artist artist = dali();
if (e.match(artist)) {
    // do something with the matching object...
}
```

Reading property value:

```
String name = (String) Artist.NAME.getExpression().evaluate(artist);
```

Filtering a list of objects:

```
Expression e = Artist.NAME.in("Dali", "Monet");
List<Artist> unfiltered = ObjectSelect.query(Artist.class).select(context);
List<Artist> filtered = e.filterObjects(unfiltered);
```



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

Chapter 8. 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 o1 = new Ordering("title", SortOrder.ASCENDING);

// the same ordering created from a property generated by Cayenne
Ordering o2 = Painting.TITLE.asc();
```

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 = Painting.TITLE.ascInsensitive();
List<Painting> list = ObjectSelect.query(Painting.class).select(context);
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.

Chapter 9. Queries

Query objects are used to communicate with the database. Most often, to select some objects matching a criteria. Cayenne knows how to translate queries to SQL statements appropriate for a particular database engine. Other types of queries allow to run native SQL, call 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. 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`) are built with abstractions originating in the object model (the "object" side in the "object-relational" divide). E.g. `ObjectSelect` consists of a Java class of 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 using SQL (`SQLSelect`, `SQLExec` query), a reference to a stored procedure (`ProcedureQuery`), etc. The results of native queries are lists of scalars, lists of `Object[]` or lists of maps (a term "data row" is often used to describe such a map). Some of them can potentially be converted to persistent objects (though usually with considerable effort). Native queries are less (if at all) portable across databases than object queries.

ObjectSelect



`ObjectSelect` supersedes older `SelectQuery`. `SelectQuery` is deprecated since 4.2 and removed in 5.0.

Selecting objects

`ObjectSelect` is the most commonly used query in Cayenne applications. This may be the only query you will ever need. It returns a list of persistent objects (or data rows) of a certain type specified in the query:

```
List<Artist> objects = ObjectSelect.query(Artist.class).select(context);
```

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

```
INFO: SELECT DATE_OF_BIRTH, NAME, ID FROM ARTIST | selected:5 time_ms:5
```

This SQL was generated by Cayenne from the `ObjectSelect` above. `ObjectSelect` can have a qualifier to select only the data matching specific criteria. Qualifier is simply an Expression (Expressions where discussed in the previous chapter), appended to the query using "where" method. If you only want artists whose name begins with 'Pablo', you might use the following qualifier expression:

```
List<Artist> objects = ObjectSelect.query(Artist.class)
        .where(Artist.NAME.like("Pablo%"))
        .select(context);
```

The SQL will look different this time:

```
INFO: SELECT DATE_OF_BIRTH, NAME, ID FROM ARTIST WHERE NAME LIKE ?
    | bind:[NAME:'Pablo%'] selected:1 time_ms:6
```

`ObjectSelect` allows to assemble qualifier from parts, using "and" and "or" method to chain them together:

```
List<Artist> objects = ObjectSelect.query(Artist.class)
        .where(Artist.NAME.like("A%"))
        .and(Artist.DATE_OF_BIRTH.gt(someDate))
        .select(context);
```

To order the results of `ObjectSelect`, one or more orderings can be applied:

```
List<Artist> objects = ObjectSelect.query(Artist.class)
        .orderBy(Artist.DATE_OF_BIRTH.desc())
        .orderBy(Artist.NAME.asc())
        .select(context);
```

There's a number of other useful methods in `ObjectSelect` 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 `ObjectSelect` features.

Selecting by ID

`ObjectSelect` can match objects of the root entity by their primary key with `byId(..)` and `byIds(..)`:

```
Artist artistWithId1 = ObjectSelect.query(Artist.class)
        .byId(1)
        .selectOne(context);

List<Artist> artists = ObjectSelect.query(Artist.class)
        .byIds(1, 2, 3)
        .select(context);
```

Both methods accept an id in any of three forms, and `byIds(..)` may mix them:

- a scalar value, for entities with a single-column primary key;

- a `Map` of DB attribute names to values, which is the form to use for a compound primary key;
- an `ObjectId`.

```
CompoundPk object = ObjectSelect.query(CompoundPk.class)
        .byId(Map.of("KEY1", "x", "KEY2", "y"))
        .selectOne(context);

Artist artist = ObjectSelect.query(Artist.class)
        .byId(objectId)
        .selectOne(context);
```

Being an ordinary qualifier, an id match combines with everything else `ObjectSelect` offers - prefetching, caching, ordering, pagination and further conditions. Under the hood these are shorthands for the "id" expressions of the entity `SELF` property. So here is a somewhat more verbose equivalent that is also more flexible (e.g. allowing to chain id qualifier with "or") :

```
List<Artist> artists = ObjectSelect.query(Artist.class)
        .where(Artist.NAME.eq("artist4"))
        .or(Artist.SELF.idsIn(1, 2, 3))
        .select(context);

List<Painting> paintings = ObjectSelect.query(Painting.class)
        .where(Painting.ARTIST.eqId(1))
        .select(context);
```

Yet another way to find a single object by PK per the ObjectContext chapter is `ObjectContext.objectForPK(..)`. It will look in the object cache before running a query.

Selecting individual columns

`ObjectSelect` query can be used to fetch individual properties of objects via type-safe API:

```
List<String> names = ObjectSelect.columnQuery(Artist.class, Artist.NAME)
        .select(context);
```

And here is an example of selecting several properties. The result is a list of `Object[]`:

```
List<Object[]> nameAndDate = ObjectSelect
        .columnQuery(Artist.class, Artist.NAME, Artist.DATE_OF_BIRTH)
        .select(context);
```

Selecting using aggregate functions

ObjectSelect query supports usage of aggregate functions. Most common variant of aggregation is selecting count of records, this can be done really easy:

```
long count = ObjectSelect.query(Artist.class).selectCount(context);
```

But you can use aggregates in more cases, even combine selecting individual properties and aggregates:

```
// Artist.SELF - is a special property that denotes a full object in this case
List<Object[]> artistAndPaintingCount = ObjectSelect.columnQuery(Artist.class,
                Artist.SELF,
                Artist.PAINTINGS.count())
        .where(Artist.NAME.like("P%"))
        .having(Artist.PAINTINGS.count().lt(5L))
        .orderBy(Artist.PAINTINGS.count().desc(), Artist.NAME.asc())
        .select(context);

for (Object[] next : artistAndPaintingCount) {
    Artist artist = (Artist) next[0];
    long paintings = (Long) next[1];
    System.out.println(artist.getName() + " has " + paintings + " paintings");
}
```

Here is generated SQL for this query:

```
SELECT DISTINCT a.DATE_OF_BIRTH, a.NAME, a.ID, COUNT(p.ID)
FROM ARTIST a JOIN PAINTING p ON a.ID = p.ARTIST_ID
WHERE a.NAME LIKE ?
GROUP BY a.DATE_OF_BIRTH, a.NAME, a.ID
HAVING (COUNT(p.ID) < ?)
ORDER BY COUNT(p.ID) DESC, a.NAME
```

Subqueries

`ObjectSelect` supports subqueries. Here is an example of a basic `NOT EXISTS` subquery:

```
long count = ObjectSelect.query(Artist.class)
        .where(Artist.PAINTINGS.notExists())
        .selectCount(context);
```

And here's a generated SQL:

```
SELECT COUNT( *)
FROM ARTIST a
WHERE
    NOT EXISTS (SELECT p.ID
                FROM PAINTING p
                WHERE p.ARTIST_ID = a.ID
```

```
    )
```

Object Query Language

An `ObjectSelect` or a `ColumnSelect` can be created from a String. This is an alternative to the fluent API. E.g.:

```
List<Artist> artists = ObjectSelect
        .parse(Artist.class, "from Artist where name like $name order by dateOfBirth
desc limit 10", "A%")
        .select(context);
```

The parser produces the same query object that the fluent API would, so everything said about ObjectSelect above applies. The parsed query can be customized further with the fluent API before it is executed:

```
List<Artist> artists = ObjectSelect
        .parse(Artist.class, "from Artist where name like $name", "A%")
        .localCache("artists")
        .pageSize(50)
        .select(context);
```

Parameters

The values of the `$name` parameters of a query are bound either positionally, in the order of the first occurrence of the names in the String, as in the examples above, or by name from a `Map`. When bound by name, a parameter that has no value is dropped from the query together with the smallest condition that contains it, in the subqueries alike. So a single String can serve a search form with optional filters:

```
// "$born" has no value, so its condition is dropped from the query
List<Artist> artists = ObjectSelect
        .parse(Artist.class, "from Artist where name like $name and dateOfBirth >
$born",
                Map.of("name", "A%"))
        .select(context);
```

With no values at all the parameters are preserved in the parsed query, to be bound later.

Query Structure

```
[ select [ distinct ] columns ]
from Entity
[ where     expression ]
[ having    expression ]
```

```
[ order by  orderings ]
[ limit n ]
[ offset m ]
[ prefetch  paths ]
```

The clauses must follow in this order, and all of them, except for `from`, are optional (even `select`!). Here are the rules of the language:

- There is a single root entity, named in the `from` clause. All other entities are reached via the relationship paths starting at the root (`paintings.gallery.name`), so there are no entity aliases and no explicit joins. Just like in expressions, a join is made outer by appending `` ` `` to a path component (`` `paintings ``), and a join is split from the other joins over the same path with an alias (`paintings#p1`).
- Every `expression` above, as well as every column and ordering, follows the syntax of Cayenne expressions, the same one that `ExpressionFactory.exp(..)` takes. Anything that is valid in an expression is valid in a query: comparisons, `like`, `in`, `between`, arithmetics, functions, `db:` paths, etc.
- The keywords are case-sensitive, and must be spelled exactly as shown here, which is also true for the expressions. The words that start the clauses are not reserved, and can be used as property names. The only reserved word is `self`, that denotes the root object. A property called "self" has to be written as `obj:self`.
- A query with no `select` clause, or the one with `select self`, fetches the root objects and is an `ObjectSelect`. Any other `select` clause results in a `ColumnSelect`.
- There is no `group by` clause. Just like with the fluent `ColumnSelect`, when any of the columns is an aggregate function, all the other columns become the grouping columns.
- Only selecting queries are supported. There is no `update` or `delete`.

Parameters

A parameter is a name prefixed with `$`, the same as in expressions. The values are passed to the `parse(..)` method and are bound in the order in which the parameters first appear in the String. If the same name is used more than once, it takes a single value that is bound in all those places. A value can be anything that is a valid expression operand: a scalar, a collection for `in`, a persistent object to match a relationship, etc.:

```
List<Painting> paintings = ObjectSelect
        .parse(Painting.class,
                "from Painting where artist = $artist and estimatedPrice between $low
and $high",
                artist, 10, 500)
        .select(context);
```

The parameters can be used anywhere in the expressions, including the nested selects. `limit` and `offset` only take integer literals.

Java API

There are four methods to parse a query, that differ in the type of the result:

```
// any query. Returns either an ObjectSelect or a ColumnSelect
FluentSelect<?, ?> q1 = ObjectSelect.parse("from Artist");

// a query of the root objects. "DataRow.class" is used to fetch the data rows instead
of objects
ObjectSelect<Artist> q2 = ObjectSelect.parse(Artist.class, "from Artist");
ObjectSelect<DataRow> q3 = ObjectSelect.parse(DataRow.class, "from Artist");

// a query with a single column
ColumnSelect<String> q4 = ObjectSelect.parseColumn(String.class, "select name from
Artist");

// a query with multiple columns
ColumnSelect<Object[]> q5 = ObjectSelect.parseColumns("select name, dateOfBirth from
Artist");
```

The last three are there to avoid the type casts. They throw an exception if the query is of a different kind, e.g. if a `select` clause is passed to `parse(Artist.class, ..)`.

The parser needs no mapping information, so the names of entities and properties, as well as the types of the columns, are only checked when the query is executed.

Examples

Each example shows a query String and the same query built with the fluent API.

Objects with a filter, ordering and a limit

```
from Artist where name like 'A%' order by dateOfBirth desc, name asc insensitive limit
10 offset 20
```

```
ObjectSelect<Artist> query = ObjectSelect.query(Artist.class)
        .where(Artist.NAME.like("A%"))
        .orderBy(Artist.DATE_OF_BIRTH.desc(), Artist.NAME.ascInsensitive())
        .limit(10)
        .offset(20);
```

An ordering is an expression, optionally followed by `asc` (the default) or `desc`, and then optionally by `insensitive`.

Joins

The joins are implied by the paths. Here is an outer join, matching the artists with no paintings:

```
from Artist where paintings+ = null
```

```
ObjectSelect<Artist> query = ObjectSelect.query(Artist.class).where(Artist.PAINTINGS
.outer().isNull());
```

And here are two separate joins over the same relationship, matching the artists that have both paintings:

```
from Artist where paintings#p1.title = 'P1' and paintings#p2.title = 'P2'
```

```
ObjectSelect<Artist> query = ObjectSelect.query(Artist.class)
        .where(Artist.PAINTINGS.alias("p1").dot(Painting.TITLE).eq("P1"))
        .and(Artist.PAINTINGS.alias("p2").dot(Painting.TITLE).eq("P2"));
```

Columns

```
select name, dateOfBirth from Artist order by name
```

```
ColumnSelect<Object[]> query = ObjectSelect
        .columnQuery(Artist.class, Artist.NAME, Artist.DATE_OF_BIRTH)
        .orderBy(Artist.NAME.asc());
```

The result is a `List<Object[]>`. With a single column it is a list of values:

```
select distinct gallery.name from Painting
```

```
ColumnSelect<String> query = ObjectSelect
        .columnQuery(Painting.class, Painting.GALLERY.dot(Gallery.NAME))
        .distinct();
```

A column can be any expression, and it can have an alias (the same as `BaseProperty.alias(..)`):

```
select upper(name) as upperName, length(trim(name)) as nameLength from Artist
```

Aggregates

```
select artist.name, sum(estimatedPrice), count(*)
from Painting
having sum(estimatedPrice) > $min
```

```
order by artist.name
```

```
ColumnSelect<Object[]> query = ObjectSelect
        .columnQuery(Painting.class,
                Painting.ARTIST.dot(Artist.NAME),
                Painting.ESTIMATED_PRICE.sum(),
                PropertyFactory.COUNT)
        .having(Painting.ESTIMATED_PRICE.sum().gt(min))
        .orderBy(Painting.ARTIST.dot(Artist.NAME).asc());
```

`artist.name` becomes a grouping column here, as the other columns are aggregates.

Objects as columns

`self` is the root object, and a relationship path is a related object. Here each row is an `Object[]` with an `Artist` and a `Long`:

```
select self, count(paintings+) from Artist order by count(paintings+) desc
```

```
ColumnSelect<Object[]> query = ObjectSelect
        .columnQuery(Artist.class, Artist.SELF, Artist.PAINTINGS.outer().count())
        .orderBy(Artist.PAINTINGS.outer().count().desc());
```

A to-one relationship gives a related object in each row:

```
select title, artist, gallery from Painting
```

```
ColumnSelect<Object[]> query = ObjectSelect
        .columnQuery(Painting.class, Painting.TITLE, Painting.ARTIST, Painting.
GALLERY);
```

With a to-many relationship a row is repeated for each related object. This is the same as `flat()` in the fluent API:

```
select name, paintings from Artist
```

```
ColumnSelect<Object[]> query = ObjectSelect
        .columnQuery(Artist.class, Artist.NAME, Artist.PAINTINGS.flat());
```

In both cases a row with no related object is still present in the result, with a `null` in place of the object. To exclude such rows, add a condition to the `where` clause, e.g. `where exists paintings`.

Subqueries

A nested select in parentheses can be used with `exists`, `not exists`, `in`, `not in`, `all` and `any`. It has the same structure as the main query, except for `prefetch`. Its paths are relative to its own root, and `enclosing(..)` is used to refer to the main query:

```
from Artist where exists (
    from Painting where artist = enclosing(self) and estimatedPrice > $price)
```

```
ObjectSelect<Artist> query = ObjectSelect.query(Artist.class)
        .where(ExpressionFactory.exists(ObjectSelect.query(Painting.class)
                .where(Painting.ARTIST.eq(Artist.SELF.enclosing()))
                .and(Painting.ESTIMATED_PRICE.gt(price))));
```

`enclosing(self)` is the root object of the enclosing query, `enclosing(name)` is its property, and `enclosing(enclosing(self))` goes two levels up.

`in`, `all` and `any` take a nested select with a single column:

```
from Painting where artist.name in (select name from Artist where dateOfBirth < $date)

from Painting where estimatedPrice > all (select estimatedPrice from Painting where
title = 'P2')

from Painting where estimatedPrice = any (select max(estimatedPrice) from Painting)
```

```
ObjectSelect<Painting> query = ObjectSelect.query(Painting.class)
        .where(Painting.ARTIST.dot(Artist.NAME).in(
                ObjectSelect.columnQuery(Artist.class, Artist.NAME)
                        .where(Artist.DATE_OF_BIRTH.lt(date))));
```



When there is no need for a nested select with its own conditions,`exists` over a path is shorter: `from Artist where not exists paintings`.

DbEntity as a root

A root prefixed with `db:` is a name of a DbEntity. Such a query fetches the data rows, and all its paths are the DB paths, with no need for a `db:` prefix:

```
from db:ARTIST where NAME like 'A%' order by ID desc
```

```
ObjectSelect<DataRow> query = ObjectSelect.dbQuery("ARTIST")
        .where(ExpressionFactory.likeDbExp("NAME", "A%"))
        .orderBy(new Ordering(ExpressionFactory.dbPathExp("ID"), SortOrder.DESCENDING
```

```
));
```

Prefetching

`prefetch` is a comma-separated list of relationship paths, each optionally followed by the prefetch semantics - `joint`, `disjoint` or `disjointById`. If omitted, Cayenne picks the default semantics, the same as with `prefetch(String, int)`:

```
from Artist where name = $name prefetch paintings joint, paintings.gallery
```

```
ObjectSelect<Artist> query = ObjectSelect.query(Artist.class)
        .where(Artist.NAME.eq(name))
        .prefetch(Artist.PAINTINGS.joint())
        .prefetch("paintings.gallery", PrefetchTreeNode.UNDEFINED_SEMANTICS);
```

SQLSelect and SQLExec

SQL is very powerful and allows to manipulate data in ways that can not always be described as a graph of related entities. Cayenne acknowledges this fact and provides a facility to execute SQL, sometimes allowing to map results back to persistent objects. `SQLSelect` and `SQLExec` are a pair of queries that allow to run native SQL. `SQLSelect` can be used (as the name suggests) to select custom data in form of entities, separate columns, collection of `DataRow` or `Object[]`. `SQLExec` is designed to execute any SQL (e.g. updates, deletes, DDLs, etc.).

Both queries support advanced SQL templating, with variable substitution and special directives as described in the next chapter. Here we'll just provide a few simple examples:

```
// Selecting objects
List<Painting> paintings = SQLSelect
        .query(Painting.class, "SELECT * FROM PAINTING WHERE TITLE LIKE 'G%'")
        .upperColumnNames()
        .localCache()
        .limit(100)
        .select(context);

// Selecting scalar values
List<String> paintingNames = SQLSelect
        .scalarQuery("SELECT TITLE FROM PAINTING WHERE ESTIMATED_PRICE > 100000",
String.class)
        .select(context);

// Selecting DataRow with predefined types
List<DataRow> rows = SQLSelect
        .dataRowQuery("SELECT * FROM ARTIST", Integer.class, String.class, LocalDate
.class)
        .select(context);
```

```
// Selecting Object[] with predefined types
List<Object[]> arrays = SQLSelect
        .columnQuery("SELECT * FROM ARTIST", Integer.class, String.class, LocalDate
.class)
        .select(context);
```

And here is an example of how to use `SQLExec`:

```
int inserted = SQLExec
        .query("INSERT INTO ARTIST (ID, NAME) VALUES (55, 'Picasso')")
        .update(context);
```

Scripting SQL Queries

A powerful feature of `SQLSelect` and `SQLExec` is that SQL string is treated by Cayenne as a dynamic template. Before creating a PreparedStatement, the String is evaluated, resolving its dynamic parts. The two main scripting elements are "variables" (that look like `$var`) and "directives" (that look like `#directive(p1 p2 p3)`). In the discussion below we'll use both selecting and updating examples, as scripting works the same way for both `SQLSelect` and `SQLExec`.

Variable Substitution

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

```
// this will generate SQL like this: "delete from PUBLIC.PAINTING"
SQLExec query = SQLExec.query("delete from $tableName")
        .params("tableName", "PUBLIC.PAINTING");
```

Variable substitution within the text uses `object.toString()` method to replace the variable value. 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 DB SQL parser. In such cases variable should be wrapped in `#bind` directive as described below.

Directives

"Directives" look like `#directive(p1 p2 p3)` (notice the absence of comma between the arguments). The following directives are supported in SQL templates:

#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 some flavor 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(..)"`. Motivation for this directive is to handle NULL SQL syntax. If the value is not null, `= ?` is generated, but if it is, the resulting SQL would look like `IS NULL`, which is compliant 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 "!=" 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($name) where ID #bindNotEqual($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 an 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' 'ID'
)";
Artist artistParam = dali;

SQLSelect<Painting> select = SQLSelect.query(Painting.class, sql)
        .param("a", artistParam);
```

#bindObjectNotEqual

Same as `#bindObjectEqual` above, only generates `!=` 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'
'ID' )";
Artist artistParam = dali;

SQLSelect<Painting> select = SQLSelect.query(Painting.class, sql)
        .param("a", artistParam);
```

#result

Used around a column in `SELECT` clause to define the type conversion of the column value (e.g. it may force a conversion from Integer to Long) and/or define column name in the result (useful when fetching objects or DataRows).



You don't have to use `#result` for any given query if the default data types and column names coming from the query suit your needs. But if you do, you have to provide `#result` for every single result column, otherwise such column will be ignored.

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
```



For advanced features you may look at the Apache Velocity Extension

MappedSelect and MappedExec

`MappedSelect` and `MappedExec` is a queries that are just a reference to another queries stored in the DataMap. The actual stored query can be ObjectSelect, SQLTemplate or ProcedureQuery. Difference between `MappedSelect` and `MappedExec` is (as reflected in their names) whether underlying query intended to select data or just to perform some generic SQL code.

An ObjectSelect stored in the DataMap is defined by a String in the Object Query Language, so it can be any query that the language supports, including the column queries and the queries rooted in a DbEntity. Its "$name" parameters are bound by name when the query is executed. A condition with a parameter that has no value is dropped from the query, so the same mapped query can be used both with and without its optional parameters.

Here is example of how to use `MappedSelect`:

```
List<Artist> results = MappedSelect.query("artistsByName", Artist.class)
        .param("name", "Dali")
        .select(context);
```

And here is example of `MappedExec`:

```
List<QueryResult> result = MappedExec.query("updatePaintingTitle")
        .param("title", "Untitled")
        .execute(context);
UpdateResult update = (UpdateResult) result.getFirst();
System.out.println("Rows updated: " + update.count());
```

ProcedureCall

Stored procedures are mapped as separate objects in CayenneModeler. `ProcedureCall` provides a way to execute them with a certain set of parameters. This query is a "fluent" version of older `ProcedureQuery`. Just like with `SQLTemplate`, the outcome of a procedure can be anything - a single result set, multiple result sets, some data modification (returned as an update count), or a combination of these. So use root class to get a single result set, and use only procedure name for anything else:

```
List<Artist> result = ProcedureCall.query("my_procedure", Artist.class)
        .param("p1", "abc")
        .param("p2", 3000)
        .select(context);
```

```
// here we do not bother with root class.
```

```
// Procedure name gives us needed routing information
List<QueryResult> result = ProcedureCall.query("my_procedure")
        .param("p1", "abc")
        .param("p2", 3000)
        .call(context);
```

`QueryResult` is a sealed interface with `SelectResult`, `UpdateResult`, `ResultIterator` and `OutParametersResult` variants, listed in the order the procedure produced them, so the items can be accessed by index or scanned with a pattern-matching switch. A stored procedure can return data back to the application as result sets or via OUT parameters. The values of OUT and INOUT parameters are reported as a separate `OutParametersResult` item of the result, a map keyed by parameter name:

```
for (QueryResult item : ProcedureCall.query("my_procedure").call(context)) {
    switch (item) {
        case SelectResult<?> select -> process(select.objects());
        case UpdateResult update -> process(update.counts());
        case OutParametersResult out -> process(out.values().get("out_param"));
        default -> {
        }
    }
}
```

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.

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. E.g. to do something directly in the JDBC layer, you might implement `Query` like this:

```
public class MyQuery implements Query {

    @Override
    public QueryMetadata getMetaData(EntityResolver resolver) {
        // the metadata tells Cayenne which DataNode to run the query on. Here we
simply borrow it from a standard query
        return ObjectSelect.query(Artist.class).getMetaData(resolver);
    }

    @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(Artist.class, generateRawSQL());
        delegate.setFetchingDataRows(true);
        return delegate;
    }

    protected String generateRawSQL() {
        // build some SQL string
        return "SELECT * FROM ARTIST";
    }
}
```

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

Chapter 10. 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 environment 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 registered:

Table 3. 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()`.

Event	Occurs…
PostRemove	right after a commit of a deleted object is done, inside `ObjectContext.commitChanges()`.
PostLoad	• After an object is fetched inside ObjectContext.select(). • 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

Apache Cayenne provides two main methods to set up callbacks for Persistent objects: using annotated methods or configuring them via Cayenne Modeler. 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 `PersistentObject.validateForX()` methods. In the future validation may be completely superseded 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.

Annotated Callbacks Methods

You can define callback methods directly in your Persistent object class using annotations. These methods are invoked automatically by Cayenne during the lifecycle of the object.

```
public class Order extends _Order {

    @PostAdd
    public void onNewOrder() {
        // Custom logic before inserting
    }
}
```

Cayenne Modeler Callbacks Configuration

Alternatively, you can configure callbacks using the Cayenne Modeler 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 'postAdd' callback called 'onNewOrder' in ObjEntity 'Order', the following code will be generated:

```
public abstract class _Order extends PersistentObject {
    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.

Callbacks on Non-Persistent Listeners

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 CayenneRuntime 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:

```
CayenneRuntime runtime = CayenneRuntime.of()
        // ..
        .addModule(binder ->
                CoreModule.extend(binder)
                        .addListenerType(Listener1.class)
                        .addListener(new Listener2())
        )
        // ..
        .build();
```

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 multiple 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 DataChannel filters

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 `DataChannelSyncFilter` (and/or `DataChannelQueryFilter`), 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 DataChannelSyncFilter {

    private ThreadLocal<int[]> counter = new ThreadLocal<int[]>();

    @Override
    public GraphDiff onSync(ObjectContext originatingContext, GraphDiff changes, int
syncType,
                            DataChannelSyncFilterChain 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:

```
// this will also add filter as a listener
CayenneRuntime runtime = CayenneRuntime.of()
        // ..
        .addModule(b ->
                CoreModule.extend(b)
                        .addSyncFilter(CommittedObjectCounter.class)
        )
        // ..
        .build();
```

Chapter 11. Commit Log

Cayenne includes built-in support for capturing commit changes and delivering them to application-defined listeners. This lets you build audit trails, propagate changes to external systems, or enforce post-commit business rules without coupling that logic to individual entity classes.

Basic Usage

To receive commit notifications, follow three steps.

1. Annotate entities

Mark every entity whose changes you want to capture with `@CommitLog`:

```
@CommitLog
public class Order extends _Order {
    // ...
}
```

By default every committed entity is included. To limit auditing to annotated entities only, call `commitLogAnnotationEntitiesOnly()` during configuration (see below).

The annotation accepts optional attributes:

```
@CommitLog(
        ignoredProperties = {"internalState"},         // exclude from change records
        ignoreAttributes = false,                       // set true to skip all
attributes
        ignoreToOneRelationships = false,               // set true to skip all to-one
relationships
        ignoreToManyRelationships = false               // set true to skip all to-
many relationships
)
public class Order extends _Order {
    // ...
}
```

For properties containing sensitive data, annotate the getter with `@Confidential` instead of `ignoredProperties`. The value will appear in the `ChangeMap` as `*` rather than the real value.

2. Implement CommitLogListener

```
public class AuditListener implements CommitLogListener {

    @Override
    public void onPostCommit(ObjectContext originatingContext, ChangeMap changes) {
```

```
        for (ObjectChange change : changes.getUniqueChanges()) {
            switch (change.getType()) {
                case INSERT -> System.out.println("Inserted: " + change
.getPostCommitId());
                case UPDATE -> System.out.println("Updated:  " + change
.getPostCommitId());
                case DELETE -> System.out.println("Deleted:  " + change.
getPreCommitId());
            }
        }
    }
}
```

`ChangeMap` gives access to the full set of changes in one commit. Each `ObjectChange` carries:

- `getType()` — `INSERT`, `UPDATE`, or `DELETE`
- `getPreCommitId()` / `getPostCommitId()` — the object identity before and after the commit
- `getAttributeChanges()` — map of attribute name → `AttributeChange` (old and new value)
- `getToOneRelationshipChanges()` / `getToManyRelationshipChanges()` — relationship-level changes

Multiple listeners may be registered; all receive identical change data for the same commit.

3. Register the listener

```
CayenneRuntime runtime = CayenneRuntime.of()
        .addConfig("cayenne-project.xml")
        .addModule(binder -> CoreModule.extend(binder)
                .addCommitLogListener(AuditListener.class))
        .build();
```

Restricting to Annotated Entities

To audit only entities marked with `@CommitLog`:

```
CoreModule.extend(binder)
        .commitLogAnnotationEntitiesOnly()
        .addCommitLogListener(AuditListener.class);
```

Dispatching Events Outside the Transaction

By default, listeners are called within the database transaction, so a listener can write additional rows as part of the same commit. If your listener performs its own transaction (for example writing to a different database), call `excludeCommitLogFromTransaction()` to move the notification outside the main transaction:

```
CoreModule.extend(binder)
        .excludeCommitLogFromTransaction()
        .addCommitLogListener(AuditListener.class);
```



`excludeCommitLogFromTransaction()` must be called **before** `addCommitLogListener()`.

Custom Entity Factory

For full control over which entities are included and how their metadata is computed, provide a custom `CommitLogEntityFactory`:

```
public class MyEntityFactory implements CommitLogEntityFactory {

    @Override
    public CommitLogEntity getEntity(ObjectId id) {

        // the returned CommitLogEntity defines whether the entity and its individual
properties are included
        boolean entityIncluded = !"Painting".equals(id.getEntityName());

        return new CommitLogEntity() {

            @Override
            public boolean isIncluded() {
                return entityIncluded;
            }

            @Override
            public boolean isIncluded(String property) {
                // all the properties of the excluded entity must be excluded as well
                return entityIncluded;
            }

            @Override
            public boolean isConfidential(String property) {
                return false;
            }
        };
    }
}
```

```
CoreModule.extend(binder)
        .commitLogEntityFactory(MyEntityFactory.class)
        .addCommitLogListener(AuditListener.class);
```

Chapter 12. 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:

```
List<Artist> artists = ObjectSelect
        .query(Artist.class)
        .prefetch(Artist.PAINTINGS.disjoint()) ①
        .select(context); ②
```

① Instructs Cayenne to prefetch one of Artist's relationships.

② Query is executed as usual, but the resulting Artists will have their paintings "inflated"

All types of relationships can be prefetched - to-one, to-many, flattened. A prefetch can span multiple relationships:

```
query.prefetch(Artist.PAINTINGS.dot(Painting.GALLERY).disjoint());
```

A query can have multiple prefetches:

```
query.prefetch(Artist.PAINTINGS.disjoint())
        .prefetch(Artist.PAINTINGS.dot(Painting.GALLERY).disjoint());
```

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).

A strategy to prefetch relationships is defined by prefetch "semantics". Depending on semantics, 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 performance, in some cases significantly. There are 3 types of prefetch semantics defined as constants in `org.apache.cayenne.query.PrefetchTreeNode`:

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

Disjoint prefetch semantics results in Cayenne generating 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 prefetch 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 prefetch semantics 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 `RuntimeProperties.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 prefetch 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.

`ObjectSelect` query supports all three types of semantics. You can mix and match them in the same query for different prefetches.

`SQLSelect` query supports "JOINT" and "DISJOINT_BY_ID". It does not work with "DISJOINT", as the query does not provide enough information to Cayenne to build dependent prefetch queries. So "DISJOINT" will result in exception. "JOINT" prefetching requires a bit of effort shaping the SQL to include the right columns in the result and label them properly to be convertible into object properties. The main rules to follow are:

- Include *all* columns from the root entity and every prefetched entity.
- Label each prefetched entity columns as "dbRelationship.column".

E.g.:

```
List<Artist> objects = SQLSelect.query(Artist.class, "SELECT "
                + "#result('ESTIMATED_PRICE' 'BigDecimal' ''
'paintings.ESTIMATED_PRICE'), "
                + "#result('TITLE' 'String' '' 'paintings.TITLE'), "
                + "#result('GALLERY_ID' 'int' '' 'paintings.GALLERY_ID'), "
                + "#result('t1.ID' 'int' '' 'paintings.ID'), "
                + "#result('t1.ARTIST_ID' 'int' '' 'paintings.ARTIST_ID'), "
                + "#result('NAME' 'String'), "
                + "#result('DATE_OF_BIRTH' 'java.time.LocalDate'), "
                + "#result('t0.ID' 'int' '' 'ID') "
                + "FROM ARTIST t0, PAINTING t1 "
                + "WHERE t0.ID = t1.ARTIST_ID")
        .addPrefetch(Artist.PAINTINGS.joint())
        .select(context);
```

Data Rows

Converting result set data to Persistent objects and registering these objects in the ObjectContext can be an expensive operation comparable 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 ubiquitous 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.:

```
ObjectSelect<DataRow> query = ObjectSelect.dataRowQuery(Artist.class);

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

```
SQLSelect<DataRow> query = SQLSelect.dataRowQuery("SELECT * FROM ARTIST");
List<DataRow> rows = query.select(context);
```

Individual DataRows may be converted to Persistent objects 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 an ObjectContext. ObjectContext creates an object called `ResultIterator` that is backed by an open ResultSet. Iterator 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.

Data is read from ResultIterator one row/object at a time until it is exhausted. There are two styles of accessing ResultIterator - direct access which requires explicit closing to avoid JDBC resources leak, or a callback that lets Cayenne handle resource management. In both cases iteration can be performed using "for" loop, as ResultIterator is "Iterable".

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

```
try (ResultIterator<Artist> it = ObjectSelect.query(Artist.class).iterator(context)) {
    for (Artist a : it) {
        // do something with the object...
    }
}
```

Same thing with a callback:

```
ObjectSelect.query(Artist.class).iterate(context, (Artist a) -> {
    // do something with the object...
});
```

Another example is a batch iterator that allows to process more than one object in each iteration. This is a common scenario in various data processing jobs - read a batch of objects, process them, commit the results, and then repeat. This allows to further optimize processing (e.g. by avoiding frequent commits).

```
try (ResultBatchIterator<Artist> it = ObjectSelect.query(Artist.class).batchIterator
(context, 100)) {
    for (List<Artist> list : it) {
        // do something with each list
```

```
        // possibly commit your changes
        context.commitChanges();
    }
}
```

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:

```
// the fact that result is paginated is transparent
List<Artist> artists =
        ObjectSelect.query(Artist.class).pageSize(50).select(context);
```

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

```
List<DataRow> rows =
        ObjectSelect.dataRowQuery(Artist.class).pageSize(50).select(context);
```

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

Cayenne supports mostly transparent caching of the query results. There are two levels of the cache: local (i.e. results cached by the ObjectContext) and shared (i.e. the results cached at the stack level and shared between all contexts). Local cache is much faster then the shared one, but is limited to a single context. It is often used with a shared read-only ObjectContext.

To take advantage of query result caching, the first step is to mark your queries appropriately. Here is an example for ObjectSelect query. Other types of queries have similar API:

```
ObjectSelect<Artist> query = ObjectSelect.query(Artist.class).localCache("artists");
```

This tells Cayenne that the query created here would like to use local cache of the context it is executed against. A vararg parameter to `localCache()` (or `sharedCache()`) method contains so called "cache groups". Those are arbitrary names that allow to categorize queries for the purpose of setting cache policies or explicit invalidation of the cache. More on that below.

The above API is enough for the caching to work, but by default your cache is an unmanaged LRU map. You can't control its size, expiration policies, etc. For the managed cache, you will need to explicitly use one of the more advanced cache providers. Use can use JCache integration module to enable any of JCache API compatible caching providers.

Often "passive" cache expiration policies used by caching providers are not sufficient, and the users want real-time cache invalidation when the data changes. So in addition to those policies, the app can invalidate individual cache groups explicitly via `QueryCache`:

```
QueryCache cache = runtime.getDataDomain().getQueryCache();
cache.removeGroup("artists");
```

The above can be used e.g. to build UI for manual cache invalidation. It is also possible to automate cache refresh when certain entities are committed. This can be done with the help of Cache invalidation extension.

Finally you may cluster cache group events. They are very small and can be efficiently sent over the wire to other JVMs running Cayenne. An example of Cayenne setup with event clustering is available on GitHub.

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 - `RuntimeProperties.CONTEXTS_SYNC_PROPERTY`, using one of the standard Cayenne DI approaches. E.g. from command line:

```
$ java -Dcayenne.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) {
        CoreModule.extend(binder)
                .setProperty(RuntimeProperties.CONTEXTS_SYNC_PROPERTY, "false");
    }
}
```

Chapter 13. 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 CayenneRuntime. See "Starting and Stopping CayenneRuntime" 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 its 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 dependencies. Let's start with service interfaces and classes:

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

```
public interface Service2 {
    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.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:

```
S1 String: 0_Service1Impl
S1 String: 1_Service1Impl
S1 String: 2_Service1Impl
S1 String: 3_Service1Impl
S1 String: 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 configuration 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 `CayenneRuntime.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 defined 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 CayenneRuntime, it is important to understand how to build custom extensions to a preconfigured container. As shown in "Starting and Stopping CayenneRuntime" chapter, custom extensions are done by writing an application 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();
CayenneRuntime runtime = CayenneRuntime.of()
        .addConfig("com/example/cayenne-mydomain.xml")
        .addModule(extensions)
        .build();
```

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.contexts_sync_strategy=false ...
```

A second one is to contribute a property to `o.a.c.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, overriding 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 `o.a.c.configuration.runtime.DefaultDbAdapterFactory.detectors` list:

```
public class MyDbAdapterDetector implements DbAdapterDetector {
    public DbAdapter createAdapter(DatabaseMetaData md) throws SQLException {
```

```
        // check if we support this database and return custom adapter,
        // or return null to let the other detectors handle it
        return null;
    }
}
```

```
CoreModule.extend(binder)
        .addAdapterDetector(MyDbAdapterDetector.class);
```

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

Alternative Service Implementations

As mentioned above, custom modules are loaded by CayenneRuntime 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.CoreModule` - the main built-in module in Cayenne.

Using custom data types

Value object type

`ValueObjectType` is a new and lightweight alternative to the Extended Types API described in the following section. In most cases is should be preferred as is it easier to understand and use. Currently only one case is known when `ExtendedType` should be used: when your value object can be mapped on different JDBC types.

In order to use your custom data type you should implement `ValueObjectType` describing it in terms of some type already known to Cayenne (e.g. backed by system or user ExtendedType). Let's assume we want to support some data type called `Money`:

```
public class Money {
    private BigDecimal value;

    public Money(BigDecimal value) {
        this.value = value;
    }

    public BigDecimal getValue() {
        return value;
    }

    // .. some other business logic ..
}
```

Here is how `ValueObjectType` that will allow to store our `Money` class as `BigDecimal` can be implemented:

```
public class MoneyValueObjectType implements ValueObjectType<Money, BigDecimal> {

    @Override
    public Class<BigDecimal> getTargetType() {
        return BigDecimal.class;
    }

    @Override
    public Class<Money> getValueType() {
        return Money.class;
    }

    @Override
    public Money toJavaObject(BigDecimal value) {
        return new Money(value);
    }

    @Override
    public BigDecimal fromJavaObject(Money object) {
        return object.getValue();
    }

    @Override
    public String toCacheKey(Money object) {
        return object.getValue().toString();
    }
}
```

Last step is to register this new type in `CayenneRuntime`:

```
CayenneRuntime runtime = CayenneRuntime.of()
        .addConfig("cayenne-project.xml")
        .addModule(binder ->
                CoreModule.extend(binder)
                        .addValueObjectType(MoneyValueObjectType.class))
        .build();
```

More examples of implementation you can find in cayenne core.

Extended Types

JDBC specification defines a set of "standard" database column types (defined in java.sql.Types class) and a very specific mapping of these types to Java Object Types, such as java.lang.String, java.math.BigDecimal, etc. Sometimes there is a need to use a custom Java type not known to JDBC driver and Cayenne allows to configure it. For this Cayenne needs to know how to instantiate this

type from a database "primitive" value, and conversely, how to transform an object of the custom type to a JDBC-compatible object.

Supporting Non-Standard Types

For supporting non-standard type you should define it via an interface `org.apache.cayenne.access.types.ExtendedType`. An implementation must provide `ExtendedType.getClassName()` method that returns a fully qualified Java class name for the supported custom type, and a number of methods that convert data between JDBC and custom type. The following example demonstrates how to add a custom DoubleArrayType to store `java.lang.Double[]` as a custom string in a database:

```
/**
 * Defines methods to read Java objects from JDBC ResultSets and write as parameters
of
 * PreparedStatements.
 */
public class DoubleArrayType implements ExtendedType<Double[]> {

    private final String SEPARATOR = ",";

    /**
     * Returns a full name of Java class that this ExtendedType supports.
     */
    @Override
    public String getClassName() {
        return Double[].class.getCanonicalName();
    }

    /**
     * Initializes a single parameter of a PreparedStatement with object value.
     */
    @Override
    public void setJdbcObject(PreparedStatement statement, Double[] value,
                              int pos, int type, int scale) throws Exception {

        statement.setString(pos, toString(value));
    }

    /**
     * Reads an object from JDBC ResultSet column, converting it to class returned by
     * 'getClassName' method.
     *
     * @throws Exception if read error occurred, or an object can't be converted to a
     *                   target Java class.
     */
    @Override
    public Double[] materializeObject(ResultSet rs, int index, int type) throws
Exception {
        return fromString(rs.getString(index));
```

```
    }

    /**
     * Reads an object from a stored procedure OUT parameter, converting it to class
     * returned by 'getClassName' method.
     *
     * @throws Exception if read error occurred, or an object can't be converted to a
     *                   target Java class.
     */
    @Override
    public Double[] materializeObject(CallableStatement rs, int index, int type)
throws Exception {
        return fromString(rs.getString(index));
    }

    /**
     * Converts the value to a String. Used for the database write, as well as by
Cayenne for logging.
     */
    @Override
    public String toString(Double[] value) {
        return Arrays.stream(value).map(String::valueOf).collect(Collectors.joining
(SEPARATOR));
    }

    private Double[] fromString(String string) {
        return Arrays.stream(string.split(SEPARATOR)).map(Double::valueOf).toArray
(Double[]::new);
    }
}
```

```
// add DoubleArrayType to list of user types
CayenneRuntime runtime = CayenneRuntime.of()
        .addConfig("cayenne-project.xml")
        .addModule(binder ->
                CoreModule.extend(binder)
                        .addUserExtendedType(new DoubleArrayType()))
        .build();
```

DbAdapters and Extended Types

As shown in the example above, ExtendedTypes are stored by DbAdapter. In fact DbAdapters often install their own extended types to address incompatibilities, incompleteness and differences between JDBC drivers in handling "standard" JDBC types. For instance some drivers support reading large character columns (CLOB) as java.sql.Clob, but some other - as "character stream", etc. Adapters provided with Cayenne override `configureExtendedTypes()` method to install their own types, possibly substituting Cayenne defaults. Custom DbAdapters can use the same technique.

Noteworthy Built-in Services

SQLLogger

`org.apache.cayenne.log.SQLLogger` is the service that defines the SQL logging API for Cayenne internals. It logs each executed statement as a single compact line combining the SQL, its bindings, the result count and the execution time, e.g. (long lines are wrapped here for readability):

```
SELECT id FROM my_table WHERE user_id = ? | bind:[user_id:15] selected:1 time_ms:2
INSERT INTO table1(id, name) VALUES(?, ?)
    | bind:[id:1,name:'n1'][id:2,name:'n2'] updated:2 time_ms:4
```

A statement that produces more than one result (such as a stored procedure) reports the extra results on `also …` continuation lines. The default implementation is `org.apache.cayenne.log.Slf4jSQLLogger`, which logs via the slf4j-api library under the fixed logger name `cayenne-sql`. Both the statement lines and the transaction boundaries (`tx started`, `tx committed`, `tx rolled back`) are logged at `INFO`. To silence SQL logging entirely, set the `cayenne-sql` logger level above `INFO`.

Each row of a batch is logged in its own square brackets. Once the batch exceeds `cayenne.jdbc.log.batch.threshold` rows (default 20), the bindings are truncated to that number of rows, split between the head and the tail of the batch, with the count of the elided rows in the middle. E.g., this is a batch of 5 rows logged with the threshold of 3:

```
INSERT INTO table1(id, name) VALUES(?, ?)
    | bind:[id:1,name:'n1'][id:2,name:'n2']..2..[id:5,name:'n5'] updated:5 time_ms:4
```

DataChannelSyncFilter and DataChannelQueryFilter

Interfaces of filters that allow to intercept DataChannel operations. Filters allow to implement chains of custom processors around a DataChannel, that can be used for security, monitoring, business logic, providing context to lifecycle event listeners, etc.

QueryCache

Defines API of a cache that stores query results.

Chapter 14. Agentic Coding

Overview

AI coding agents such as Claude Code, Cursor, etc. can read and write Cayenne projects like any other Java codebase. For day-to-day API work — writing queries, manipulating an `ObjectContext`, etc. — you may not need anything Cayenne-specific. Prompt the agent the way you would for any other project. But two additional elements make the agent Cayenne-aware and allow to implement full ORM workflows:

- MCP server — a small stdio-based process the agent launches on demand, exposing tools for class generation, DB import, and opening CayenneModeler. Works with any MCP-compatible client (Claude Code, Cursor, Codex, etc.).
- `apache-cayenne` Claude Code plugin — a Claude Code-specific extension that bundles pre-built skills for DataMap editing, query writing, runtime setup, and more. The skills detect the MCP server at runtime;

Both are independent but work best together. This chapter walks you through installing both and explains the available MCP tools.

`apache-cayenne` Claude Code Plugin

(Claude Code specific) Install the `apache-cayenne` plugin with Cayenne skills under Claude Code:

```
/plugin marketplace add apache/cayenne
/plugin install apache-cayenne@apache-cayenne
/reload-plugins
```

The plugin adds skills for DataMap editing, class generation, reverse engineering, query writing, and runtime setup. The skills work best with the MCP server present (see the next chapter).

Cayenne MCP Server

The MCP server is a self-contained, stdio-based process that the AI agent launches on demand. It works with most agents (not only Claude). The MCP server is bundled with the platform-specific CayenneModeler and requires Java 21 or later on the system `PATH`. So to get it, first download the Modeler at https://cayenne.apache.org/download/ .

MCP runnable jar is `CayenneMCPServer.jar`. Its location is OS-dependent (you will need this location in the next step):

Mac:

```
# <install-dir> is typically "/Applications", but it can be any other directory
<install-dir>/CayenneModeler.app/Contents/Resources/mcp/CayenneMCPServer.jar
```

Windows:

```
<install-dir>\bin\CayenneMCPServer.jar
```

Linux / cross-platform:

```
<install-dir>/bin/CayenneMCPServer.jar
```

When installing the MCP, replace `/path/to/CayenneMCPServer.jar` mentioned in the instructions with the actual jar path:

Installing on Claude Code

```
# Register the MCP server (--scope user makes it available across all projects)
claude mcp add cayenne --scope user -- java -jar /path/to/CayenneMCPServer.jar

# Verify it is connected
claude mcp list
```

Installing on Cursor

Edit `~/.cursor/mcp.json` (global, all projects) or `.cursor/mcp.json` in your project (per-project):

```
{
  "mcpServers": {
    "cayenne": {
      "command": "java",
      "args": ["-jar", "/path/to/CayenneMCPServer.jar"]
    }
  }
}
```

Installing on Codex

Add a `[mcp_servers.cayenne]` entry to `~/.codex/config.toml`:

```
[mcp_servers.cayenne]
command = "java"
args = ["-jar", "/path/to/CayenneMCPServer.jar"]
```

Available MCP Tools

The server exposes three tools. Your agent will discover them on its own, but it is useful to understand the capabilities and possible prompts. All tools operate on a Cayenne project descriptor

(a `cayenne-*.xml` file) and take absolute paths — agents should resolve relative paths against the current workspace before invoking.

If you are using the `apache-cayenne` Claude Code plugin, its skills (`cayenne-cgen`, `cayenne-modeler`, `cayenne-db-import`) call these tools automatically. You can verify the server is available with `claude mcp list` — look for an entry named `cayenne`.

open_project

Launches CayenneModeler with the given project file pre-loaded. The call is non-blocking; the server waits for the Modeler to confirm a successful project load via a short startup handshake, then returns.

Table 4. Parameters

Name	Required	Description
`projectPath`	yes	Absolute path to the top-level Cayenne project descriptor (`cayenne-*.xml`).

In most cases your agent knows where the project is, so a typical prompt that triggers this tool is just this:

> open my Cayenne project in the Modeler

Though you can be more specific and indicate the exact project to open:

> open the Cayenne project at @/path/to/cayenne-project.xml in the Modeler.

cgen_run

Runs the Cayenne class generator (`cgen`) for a single DataMap inside the project, using the `cgen` configuration stored in the DataMap XML. The tool returns a JSON report describing which files were written, which were already up-to-date, and any errors encountered.

Table 5. Parameters

Name	Required	Description
`projectPath`	yes	Absolute path to the top-level Cayenne project descriptor (`cayenne-*.xml`), not a DataMap file.
`dataMap`	yes	Name of the target DataMap as it appears in the `<map name='…'>` element of the project descriptor.

A typical agent prompt that triggers this tool:

> regenerate Java classes for the `MyDataMap` DataMap in `/path/to/cayenne-project.xml`.

Or if you only have one project and one DataMap:

> regenerate Java classes from Cayenne model

dbimport_run

Runs Cayenne reverse engineering (dbimport) for a single DataMap inside the project. The tool reads the JDBC connection from CayenneModeler preferences on the local machine (the agent doesn't see the credentials). If the DataMap has a `<reverse-engineering>` block its filters are applied; otherwise the full database schema is imported. The DataMap XML is rewritten on disk with the merged schema, and a JSON summary of what changed is returned.

If no connection has been saved yet (`dbconnector_not_configured` error), open the project in the Modeler with `open_project`, run the dialog once to save the connection, then call `dbimport_run` again.

Table 6. Parameters

Name	Required	Description
`projectPath`	yes	Absolute path to the top-level Cayenne project descriptor (`cayenne-*.xml`), not a DataMap file.
`dataMap`	yes	Name of the target DataMap as it appears in the `<map name='…'>` element of the project descriptor.

A typical agent prompt that triggers this tool:

> sync the Cayenne model with the database

Or more specifically:

> re-import the `MyDataMap` schema from the database into `/path/to/cayenne-project.xml`

Chapter 15. Additional Modules

Cache Invalidation Extension

Cache invalidation module is an extension that allows to define cache invalidation policy programmatically.

Maven

```
<dependency>
    <groupId>org.apache.cayenne</groupId>
    <artifactId>cayenne-cache-invalidation</artifactId>
    <version>5.0-RC1</version>
</dependency>
```

Gradle

```
compile 'org.apache.cayenne:cayenne-cache-invalidation:5.0-RC1'
```

Usage

Module supports autoloading mechanism, so no other actions required to enable it. Just mark your entities with @CacheGroups annotation and you are ready to use it:

```
@CacheGroups("some-group")
public class MyEntity extends _MyEntity {
    // ...
}
```

After any modification of `MyEntity` objects cache group `"some-group"` will be dropped from cache automatically.



You can read more about cache and cache groups in corresponding chapter of this documentation.

In case you need some complex logic of cache invalidation you can disable default behaviour and provide your own.

To do so you need to implement `o.a.c.cache.invalidation.InvalidationHandler` interface and setup Cache Invalidation module to use it. Let's use implementation class called `CustomInvalidationHandler` that will simply match all entities' types with `"custom-group"` cache group regardless of any annotations:

```
public class CustomInvalidationHandler implements InvalidationHandler {
```

```
    @Override
    public Function<Persistent, Collection<CacheGroupDescriptor>> canHandle(Class<?
extends Persistent> type) {
        return p -> Collections.singleton(new CacheGroupDescriptor("custom-group"));
    }
}
```

Now we'll set up it's usage by `CayenneRuntime`:

```
CayenneRuntime runtime = CayenneRuntime.of()
        .addModule(binder -> CacheInvalidationModule.extend(binder)
                // optionally you can disable @CacheGroups annotation processing
                .noCacheGroupsHandler()
                .addHandler(CustomInvalidationHandler.class))
        .build();
```



You can combine as many invalidation handlers as you need.

Crypto extension

Crypto module allows encrypt and decrypt values stored in DB transparently to your Java app.

Maven

```
<dependency>
    <groupId>org.apache.cayenne</groupId>
    <artifactId>cayenne-crypto</artifactId>
    <version>5.0-RC1</version>
</dependency>
```

Gradle

```
compile 'org.apache.cayenne:cayenne-crypto:5.0-RC1'
```

Usage

Setup your model and DB

To use crypto module you must prepare your database to allow `byte[]` storage and properly name columns that will contain encrypted values.

Currently supported SQL types that can be used to store encrypted data are:

1. Binary types: `BINARY, BLOB, VARBINARY, LONGVARBINARY`. These types are preferred.

2. Character types, that will store `base64` encoded value: `CHAR, NCHAR, CLOB, NCLOB, LONGVARCHAR, LONGNVARCHAR, VARCHAR, NVARCHAR`.



Not all data types may be supported by your database.

Default naming strategy that doesn't require additional setup suggests using "CRYPTO_" prefix. You can change this default strategy by injecting you own implementation of `o.a.c.crypto.map.ColumnMapper` interface.

```
CayenneRuntime runtime = CayenneRuntime.of()
        .addModule(binder -> CryptoModule.extend(binder)
                .columnMapper(MyColumnMapper.class))
        .build();
```

Here is an example of how `ObjEntity` with two encrypted and two unencrypted properties can look like:

[ext crypto obj entity] | *../../images/ext-crypto-obj-entity.png*

Setup keystore

To perform encryption you must provide `KEYSTORE_URL` and `KEY_PASSWORD`. Currently crypto module supports only Java "jceks" KeyStore.

```
CayenneRuntime runtime = CayenneRuntime.of()
        .addModule(binder -> CryptoModule.extend(binder)
                .keyStore(
                        this.getClass().getResource("keystore.jcek"),
                        "my-password".toCharArray(),
                        "my-key-alias"))
        .build();
```

Additional settings

In addition to `ColumnMapper` mentioned above you can customize other parts of `crypto module`. You can enable `gzip` compression and `HMAC` usage (later will ensure integrity of data).

```
CayenneRuntime runtime = CayenneRuntime.of()
        .addModule(binder -> CryptoModule.extend(binder)
                .compress()
                .useHMAC())
        .build();
```

Another useful extension point is support for custom Java value types. To add support for your data type you need to implement `o.a.c.crypto.transformer.value.BytesConverter` interface that will convert required type to and from `byte[]`.

```
CayenneRuntime runtime = CayenneRuntime.of()
        .addModule(binder -> CryptoModule.extend(binder)
                .objectToBytesConverter(MyClass.class, new MyClassBytesConverter()))
        .build();
```



In addition to Java primitive types (and their object counterparts), `crypto module` supports encryption only of `java.util.Date`, `java.math.BigInteger` and `java.math.BigDecimal` types.

JCache integration

Allows to integrate any JCache (JSR 107) compatible caching provider with Cayenne.

Maven

```
<dependency>
    <groupId>org.apache.cayenne</groupId>
    <artifactId>cayenne-jcache</artifactId>
    <version>5.0-RC1</version>
</dependency>
```

Gradle

```
compile 'org.apache.cayenne:cayenne-jcache:5.0-RC1'
```

Usage

To use JCache provider in your app you need to include this module and caching provider libs (e.g. Ehcache). You can provide own implementation of `org.apache.cayenne.jcache.JCacheConfigurationFactory` to customize cache configuration if required.

For advanced configuration and management please use provider specific options and tools.

JCache module supports custom configuration files for cache managers.

```
CayenneRuntime runtime = CayenneRuntime.of()
        .addModule(binder -> JCacheModule.extend(binder)
                .setJCacheProviderConfig("cache-config.xml"))
        .build();
```

Also JCache module supports contribution of pre-configured cache manager.

```
CayenneRuntime runtime = CayenneRuntime.of()
        .addModule(binder ->
                binder.bind(CacheManager.class).toInstance(customCacheManager))
        .build();
```



You can read about using cache in Cayenne in this chapter.

You may else be interested in Cache invalidation extension.

Ehcache setup example

Here is an example of using ehcache as cache manager.

First you need to include ehcache dependency:

```
<dependency>
    <groupId>org.ehcache</groupId>
    <artifactId>ehcache</artifactId>
    <version>3.8.1</version>
</dependency>
```

If you need custom configuration you can contribute configuration file to JCache module:

```
CayenneRuntime runtime = CayenneRuntime.of()
        .addModule(binder -> JCacheModule.extend(binder)
                .setJCacheProviderConfig("file:/ehcache.xml"))
        .build();
```

As a result you will have ehcache manager as your default cache manager.

Apache Velocity Extension

Enables usage of full featured Apache Velocity templates in SQLSelect / SQLExec queries.

Maven

```
<dependency>
    <groupId>org.apache.cayenne</groupId>
    <artifactId>cayenne-velocity</artifactId>
    <version>5.0-RC1</version>
</dependency>
```

Gradle

```
compile 'org.apache.cayenne:cayenne-velocity:5.0-RC1'
```

Usage

This module doesn't require any additional setup. In addition of directives mentioned in this chapter, this module also adds `#chain` and `#chunk` directives.

`#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($id) ID > #bind($id) #end
#end"
```

Chapter 16. Build Tools

While we encourage the use of CayenneModeler for tasks such as DB reverse-engineering and code generation, Cayenne also provides an option to execute them from your preferred build tool. It may be occasionally useful to keep them as a part of the build. This chapter shows how to use them in Maven, Gradle or Ant.

Maven Plugin

The full plugin Maven name is `org.apache.cayenne.plugins:cayenne-maven-plugin`. It can be executed as `mvn cayenne:<goal>`.

cgen

`cgen` is a 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 7. 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`

Table 8. 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.
destDir	File	Root destination directory for Java classes (ignoring their package names). The default is "src/main/java".
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.

Name	Type	Description
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.
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.
createPropertyNames	boolean	If set to "true", will generate String Property names. Default is "false"

Name	Type	Description
force	boolean	*Deprecated and ignored.* cgen always regenerates classes, so there is nothing left to force.
createPKProperties	boolean	If set to "true", will generate PK attributes as Properties. Default is "false".

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>5.0-RC1</version>

    <configuration>
        <map>${project.basedir}/src/main/resources/my.map.xml</map>
    </configuration>

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

cdbimport

`cdbimport` is a `cayenne-maven-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, possibly followed by "cgen" execution to generate the classes.

Table 9. 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. `${project.basedir}/src/main/resources/my.map.xml`. If "overwrite" is true (the default), an existing DataMap will be used as a template for the new imported DataMap, i.e. all its entities will be cleared and recreated, but its common settings, such as default Java package, will be preserved (unless changed explicitly in the plugin configuration).
cayenneProject	File	No	Project XML file which will be used. Can be an existing file, in this case data map will be added to project if it's not already there. If this file does not exist, it is created when cdbimport is executed. E.g. `${project.basedir}/src/main/resources/cayenne-project.xml`.
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.
dataSource	XML	Yes	An object that contains Data Source parameters.
dbimport	XML	No	An object that contains detailed reverse engineering rules about what DB objects should be processed. For full information about this parameter see DB Import Filtering and Other DB Import Settings.

Table 10. <dataSource> parameters

Name	Type	Required	Description
driver	String	Yes	A class of JDBC driver to use for the target database.
url	String	Yes	JDBC URL of a target database.
username	String	No	Database user name.
password	String	No	Database user password.

Table 11. <dbimport> parameters

Name	Type	Description
defaultPackage	String	A Java package that will be set as the imported DataMap default and a package of all the persistent Java classes. This is a required attribute if the "map" itself does not already contain a default package, as otherwise all the persistent classes will be mapped with no package, and will not compile.
forceDataMapCatalog	boolean	Automatically tagging each DbEntity with the actual DB catalog/schema (default behavior) may sometimes be undesirable. If this is the case then setting `forceDataMapCatalog` to `true` will set DbEntity catalog to one in the DataMap. Default value is `false`.
forceDataMapSchema	boolean	Automatically tagging each DbEntity with the actual DB catalog/schema (default behavior) may sometimes be undesirable. If this is the case then setting `forceDataMapSchema` to `true` will set DbEntity schema to one in the DataMap. Default value is `false`.
meaningfulPkTables	String	A comma-separated list of Perl5 patterns that defines which imported tables should have their primary key columns mapped as ObjAttributes. "*" would indicate all tables.
namingStrategy	String	The naming strategy used for mapping database names to object entity names. Default is `o.a.c.dbsync.naming.DefaultObjectNameGenerator`.
skipPrimaryKeyLoading	boolean	Whether to load primary keys. Default "false".
skipRelationshipsLoading	boolean	Whether to load relationships. Default "false".

Name	Type	Description
stripFromTableNames	String	Regex that matches the part of the table name that needs to be stripped off when generating ObjEntity name. Here are some examples: `<!-- Strip prefix --> <stripFromTableNames>^myt_</stripFromTableNames>  <!-- Strip suffix --> <stripFromTableNames>_s$</stripFromTableNames>  <!-- Strip multiple occurrences in the middle --> <stripFromTableNames>_abc</stripFromTableNames>`
usePrimitives	boolean	Whether numeric and boolean data types should be mapped as Java primitives or Java classes. Default is "true", i.e. primitives will be used.
useJava7Types	boolean	Whether *DATE*, *TIME* and *TIMESTAMP* data types should be mapped as `java.util.Date` or `java.time.* classes`. Default is "false", i.e. `java.time.*` will be used.
tableTypes	Collection<String>	Collection of table types to import. By default "TABLE" and "VIEW" types are used. Typical types are: • TABLE • VIEW • SYSTEM TABLE • GLOBAL TEMPORARY • LOCAL TEMPORARY • ALIAS • SYNONYM

Name	Type	Description
filters configuration	XML	Detailed reverse engineering rules about what DB objects should be processed. For full information about this parameter see DB Import Filtering. Here is some simple example: `<dbimport>   <catalog name="test_catalog">   <schema name="test_schema">   <includeTable> .*</includeTable>   <excludeTable> test_table</excludeTable>   </schema>   </catalog>    <includeProcedure pattern=".*"/> </dbimport>`

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>5.0-RC1</version>

    <executions>
        <execution>
            <configuration>
                <map>${project.basedir}/src/main/resources/my.map.xml</map>
                <dataSource>
                    <url>jdbc:mysql://127.0.0.1/mydb</url>
                    <driver>com.mysql.jdbc.Driver</driver>
                    <username>sa</username>
                </dataSource>
                <dbimport>
                    <defaultPackage>com.example.cayenne</defaultPackage>
                </dbimport>
            </configuration>
            <goals>
                <goal>cdbimport</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 12. 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`
dataSource	XML	An object that contains Data Source parameters

Table 13. <dataSource> parameters

Name	Type	Required	Description
driver	String	Yes	A class of JDBC driver to use for the target database.
url	String	Yes	JDBC URL of a target database.
username	String	No	Database user name.
password	String	No	Database user password.

Table 14. 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".

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

```
<plugin>
```

```
    <groupId>org.apache.cayenne.plugins</groupId>
    <artifactId>cayenne-maven-plugin</artifactId>
    <version>5.0-RC1</version>
    <executions>
        <execution>
            <configuration>
                <map>${project.basedir}/src/main/resources/my.map.xml</map>
                <adapter>org.apache.cayenne.dba.hsqldb.HSQLDBAdapter</adapter>
                <dataSource>
                    <url>jdbc:hsqldb:hsql://localhost/testdb</url>
                    <driver>org.hsqldb.jdbcDriver</driver>
                    <username>sa</username>
                </dataSource>
            </configuration>
            <goals>
                <goal>cdbgen</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Gradle Plugin

Cayenne Gradle plugin provides tasks similar to Maven plugin. It also provides `cayenne` extension that has some useful utility methods. Here is example of how to include Cayenne plugin into your project:

```
buildscript {
    // add Maven Central repository
    repositories {
        mavenCentral()
    }
    // add Cayenne Gradle Plugin
    dependencies {
        classpath group: 'org.apache.cayenne.plugins', name: 'cayenne-gradle-plugin',
version: '5.0-RC1'
    }
}

// apply plugin
apply plugin: 'org.apache.cayenne'

// set default DataMap
cayenne.defaultDataMap 'datamap.map.xml'

// add Cayenne dependencies to your project
dependencies {
    compile 'org.apache.cayenne:cayenne:5.0-RC1'
```

```
}
```



Cayenne Gradle plugin is experimental and it's API may still change.

cgen

Cgen task generates Java classes based on your DataMap, it has same configuration parameters as in Maven Plugin version, described in Table, "cgen required parameters".. If you provided default DataMap via `cayenne.defaultDataMap`, you can skip `cgen` configuration as default settings will suffice in common case.

Here is how you can change settings of the default `cgen` task:

```
cgen {
    mode = 'all'
    overwrite = true
    createPropertyNames = true
}
```

cdbimport

This task is for creating and synchronizing your Cayenne model from database schema. Full list of parameters are same as in Maven Plugin, with the exception that Gradle version will use Groovy instead of XML. Here is example of configuration for cdbimport task:

```
cdbimport {
    // map can be skipped if it is defined in cayenne.defaultDataMap
    map 'src/main/resources/datamap.map.xml'
    // optional project file, will be created if missing
    cayenneProject 'src/main/resources/cayenne-project.xml'

    dataSource {
        driver 'com.mysql.cj.jdbc.Driver'
        url 'jdbc:mysql://127.0.0.1:3306/test?useSSL=false'
        username 'root'
        password ''
    }

    dbImport {
        // additional settings
        usePrimitives false
        defaultPackage 'org.apache.cayenne.test'

        // DB filter configuration
        catalog 'catalog-1'
        schema 'schema-1'
```

```
        catalog {
            name 'catalog-2'

            includeTable 'table0', {
                excludeColumns '_column_'
            }

            includeTables 'table1', 'table2', 'table3'

            includeTable 'table4', {
                includeColumns 'id', 'type', 'data'
            }

            excludeTable '^GENERATED_.*'
        }

        catalog {
            name 'catalog-3'
            schema {
                name 'schema-2'
                includeTable 'test_table'
                includeTable 'test_table2', {
                    excludeColumn '__excluded'
                }
            }
        }

        includeProcedure 'procedure_test_1'

        includeColumns 'id', 'version'

        tableTypes 'TABLE', 'VIEW'
    }
}
```

cdbgen

Cdbgen task drops and/or generates tables in a database on Cayenne DataMap. Full list of parameters is same as in the Maven plugin. Here is example of how to configure default `cdbgen` task:

```
cdbgen {

    adapter 'org.apache.cayenne.dba.derby.DerbyAdapter'

    dataSource {
        driver 'org.apache.derby.jdbc.EmbeddedDriver'
        url 'jdbc:derby:build/testdb;create=true'
        username 'sa'
```

```
        password ''
    }

    dropTables true
    dropPk true

    createTables true
    createPk true
    createFk true
}
```

Link tasks to Gradle build lifecycle

You can connect Cayenne tasks to the default build lifecycle. Here is short example of how to connect defaut `cgen` and `cdbimport` tasks with `compileJava` task:

```
cgen.dependsOn cdbimport
compileJava.dependsOn cgen
```

Ant Tasks

Ant tasks are the same as Maven plugin goals described previously, namely "cgen", "cdbgen", "cdbimport". Configuration parameters are also similar (except Maven can guess many defaults that Ant can't). To include Ant tasks in the project, use the following Antlib:

```
<typedef resource="org/apache/cayenne/tools/antlib.xml">
   <classpath>
        <fileset dir="lib" >
            <include name="cayenne-*.jar" />
            <include name="cayenne-ant-*.jar" />
            <include name="cayenne-cgen-*.jar" />
            <include name="cayenne-dbsync-*.jar" />
            <include name="cayenne-di-*.jar" />
            <include name="cayenne-project-*.jar" />
            <include name="commons-collections-*.jar" />
            <include name="commons-lang-*.jar" />
            <include name="slf4j-api-*.jar" />
            <include name="velocity-*.jar" />
        </fileset>
   </classpath>
</typedef>
```

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"
   defaultPackage="com.example.cayenne"/>
```

DB Import Filtering

The first thing you usually want to control during reverse engineering is what exactly should be loaded from database and what not. One of the most common cases is excluding system tables, as you usually don't want to map them.

Briefly, you are able to include/exclude tables, columns and procedures and do it at several levels: default, catalog, schema. Although everything defined at the top level (default rules) will be applied for the nested elements, all rules from the most specific areas will override general rules (i.e. rules from schemas override rules from catalogs and even more override default rules).

The following use-cases will provide you a better understanding of how filtering works and how you could use it.

Process everything from schema/catalog

The simplest example of reverse engineering is processing tables from one schema of catalog and there are several options to do this. Basic syntax is described below:

```
<dbimport>
    <!-- Ant/Maven in case you only want to specify the schema to import -->
    <schema>SCHEMA_NAME</schema>

    <!-- Maven way in case you have nested elements in the schema  -->
    <schema>
        <name>SCHEMA_NAME</name>
        ...
    </schema>

    <!-- Ant way in case you have nested elements in the schema -->
    <schema name="SCHEMA_NAME">
        ...
    </schema>
</dbimport>
```

The same options are available for catalogs:

```
<dbimport>
    <!-- Ant/Maven in case you only want to specify the catalog to import -->
    <catalog>CATALOG_NAME</catalog>

    <!-- Maven way in case you have nested elements in the catalog -->
    <catalog>
        <name>CATALOG_NAME</name>
        ...
    </catalog>

    <!-- Ant way in case you have nested elements in the catalog -->
    <catalog name="CATALOG_NAME">
        ...
    </catalog>
</dbimport>
```



Current version of reverse engineering doesn't support catalog filtering for Postgres database.

Combine Schema and Catalog filters

Cayenne supports combination of different schemas and catalogs, and it filters data according to your requirements. You could achieve this by the following example of reverse engineering configuration:

```
<dbimport>

    <catalog>
        <name>shop_01</name>
        <schema>schema-name-01</schema>
        <schema>schema-name-02</schema>
        <schema>schema-name-03</schema>
    </catalog>

    <catalog>
        <name>shop_02</name>
        <schema>schema-name-01</schema>
    </catalog>

    <catalog>
        <name>shop_03</name>
        <schema>schema-name-01</schema>
        <schema>schema-name-02</schema>
        <schema>schema-name-03</schema>
    </catalog>

</dbimport>
```

In the example above, Cayenne reverse engineering process contains three catalogs named as shop_01, shop_02 and shop_03, each of which has their own schemas. Cayenne will load all data only from the declared catalogs and schemas.

If you want to load everything from database, you could simply declare catalog specification alone.

```
<dbimport>

    <catalog>shop_01</catalog>
    <catalog>shop_02</catalog>
    <catalog>shop_03</catalog>

</dbimport>
```

If you want to do reverse engineering for specific schemas, just remove unwanted schemas from the catalog section. For example, if you want to process schema-name-01 and schema-name-03 schemas only, then you should change reverse engineering section like this.

```
<dbimport>

    <catalog>
        <name>shop_01</name>
        <schema>schema-name-01</schema>
        <schema>schema-name-03</schema>
    </catalog>

    <catalog>
        <name>shop_02</name>
        <schema>schema-name-01</schema>
    </catalog>

    <catalog>
        <name>shop_03</name>
        <schema>schema-name-01</schema>
        <schema>schema-name-03</schema>
    </catalog>

</dbimport>
```

Including and Excluding tables, columns, procedures and relationships

Cayenne reverse engineering let you fine tune table, columns and stored procedures names that you need to import to your model file. In every filter you can use regexp syntax. Here is some examples of configuration for common tasks.

1) Include tables with 'CRM_' prefix if you are working in that domain of application:

```
<includeTable>CRM_.*</includeTable>
```

2) Include tables with '_LOOKUP' suffix

```
<includeTable>
    <pattern>.*_LOOKUP</pattern>
</includeTable>
```

3) Exclude tables with 'CRM_' prefix if you are not working only in that domain of application:

```
<excludeTable>CRM_.*</excludeTable>
```

4) Include only specific columns that follows specific naming convention:

```
<includeColumn>includeColumn01</includeColumn>
<includeColumn>includeColumn03</includeColumn>
```

5) Exclude system or obsolete columns:

```
<excludeColumn>excludeColumn01</excludeColumn>
<excludeColumn>excludeColumn03</excludeColumn>
```

6) Include/Exclude columns for particular table or group of tables:

```
<includeTable>
    <pattern>table pattern</pattern>
    <includeColumn>includeColumn01</includeColumn>
    <excludeColumn>excludeColumn01</excludeColumn>
</includeTable>
```

7) Include stored procedures:

```
<includeProcedure>includeProcedure01</includeProcedure>
<includeProcedure>
    <pattern>includeProcedure03</pattern>
</includeProcedure>
```

8) Exclude stored procedures by pattern:

```
<excludeProcedure>excludeProcedure01</excludeProcedure>
<excludeProcedure>
    <pattern>excludeProcedure03</pattern>
```

```
</excludeProcedure>
```

9) Exclude relationships:

```
<excludeRelationship>excludeRelationship01</excludeRelationship>
<excludeRelationship>
    <pattern>excludeRelationship03</pattern>
</excludeRelationship>
```

All filtering tags `<includeTable>`, `<excludeTable>`, `<includeColumn>`, `<excludeColumn>`, `<includeProcedure>`, `<excludeProcedure>` and `<excludeRelationship>` have 2 ways to pass filtering RegExp.

1) text inside tag

```
 <includeTable>CRM_.*</includeTable>
```

2) pattern inner tag

```
  <includeTable>
        <pattern>.*_LOOKUP</pattern>
    </includeTable>
```

All filtering tags can be placed inside schema and catalog tags, but also inside `<dbimport>` tag. It means that filtering rules will be applied for all schemas and catalogs.

Complete filtering example

Initially, let's make a small sample. Consider the following reverse engineering configuration.

```
<dbimport>
    <catalog>shop-01</catalog>
</dbimport>
```

In this case reverse engineering will not filter anything from the shop-01 catalog. If you really want to filter database columns, tables, stored procedures and relationships, you could do it in the following way.

```
<dbimport>
    <catalog>shop-01</catalog>
    <catalog>
        <name>shop-02</name>
        <includeTable>includeTable-01</includeTable>
    </catalog>
```

```
</dbimport>
```

Then Cayenne will do reverse engineering for both shop-01 and shop-02 catalogs. First catalog will not be processed for filtering, but the second catalog will be processed with “includeTable-01” filter.

Let’s assume you have a lot of table prefixes with the same names. Cayenne allows you to mention a pattern as regular expression. Using regular expressions is easier way to handle a big amount of database entities than writing filter config for each use-case. They make your configuration more readable, understandable and straightforward. There is not complex. Let’s see how to use patterns in reverse engineering configuration with complete example.

```
<dbimport>

    <catalog>shop-01</catalog>

    <catalog>
        <name>shop-02</name>
    </catalog>

    <catalog>
        <name>shop-03</name>
        <includeTable>includeTable-01</includeTable>

        <includeTable>
            <pattern>includeTable-02</pattern>
        </includeTable>

        <includeTable>
            <pattern>includeTable-03</pattern>
            <includeColumn>includeColumn-01</includeColumn>
            <excludeColumn>excludeColumn-01</excludeColumn>
        </includeTable>

        <excludeTable>excludeTable-01</excludeTable>

        <excludeTable>
            <pattern>excludeTable-02</pattern>
        </excludeTable>

        <includeColumn>includeColumn-01</includeColumn>

        <includeColumn>
            <pattern>includeColumn-02</pattern>
        </includeColumn>

        <excludeColumn>excludeColumn-01</excludeColumn>

        <excludeColumn>
            <pattern>excludeColumn-02</pattern>
        </excludeColumn>
```

```
        <includeProcedure>includeProcedure-01</includeProcedure>

        <includeProcedure>
            <pattern>includeProcedure-02</pattern>
        </includeProcedure>

        <excludeProcedure>excludeProcedure-01</excludeProcedure>

        <excludeProcedure>
            <pattern>excludeProcedure-02</pattern>
        </excludeProcedure>

        <excludeRelationship>excludeRelationship-01</excludeRelationship>

        <excludeRelationship>
            <pattern>excludeRelationship-02</pattern>
        </excludeRelationship>

    </catalog>
</dbimport>
```

The example above should provide you more idea about how to use filtering and patterns in Cayenne reverse engineering. You could notice that this example demonstrates you the "name" and "pattern" configurations. Yes, you could use these as separates xml element and xml attributes.

The cdbimport will execute reverse engineering task for all entities from “shop-01” and “shop-02”, including tables, views, stored procedures and table columns. As “shop-03” has variety filter tags, entities from this catalog will be filtered by cdbimport.

Ant configuration example

Here is config sample for `Ant` task:

```
<!-- inside <cdbimport> tag -->
<catalog>shop-01</catalog>

<catalog name="shop-02"/>

<catalog name="shop-03">

    <includeTable>includeTable-01</includeTable>
    <includeTable pattern="includeTable-02"/>

    <includeTable pattern="includeTable-03">
        <includeColumn>includeColumn-01</includeColumn>
        <excludeColumn>excludeColumn-01</excludeColumn>
    </includeTable>
```

```
    <excludeTable>excludeTable-01</excludeTable>
    <excludeTable pattern="excludeTable-02"/>

    <includeColumn>includeColumn-01</includeColumn>
    <includeColumn pattern="includeColumn-02"/>

    <excludeColumn>excludeColumn-01</excludeColumn>
    <excludeColumn pattern="excludeColumn-02"/>

    <includeProcedure>includeProcedure-01</includeProcedure>
    <includeProcedure pattern="includeProcedure-02"/>

    <excludeProcedure>excludeProcedure-01</excludeProcedure>
    <excludeProcedure pattern="excludeProcedure-02"/>

    <excludeRelationship>excludeRelationship-01</excludeRelationship>
    <excludeRelationship pattern="excludeRelationship-02"/>

</catalog>
```



In Ant task configuration all filter tags located inside root tag `<cdbimport>` as there is no `<dbimport>` tag.

Other DB Import Settings

In databases relations are defined via foreign keys and there are a lot of different politics according to the level of relationships and ways how those relationships could be modeled in database. Anyway, cdbimport is able to recognize basic patterns of relationships, such as OneToMany, OneToOne and ManyToMany.

Skip Relationships Loading

You are able to skip relationships loading by the `<skipRelationshipsLoading>` element.

```
<dbimport>
        <skipRelationshipsLoading>true</skipRelationshipsLoading>
</dbimport>
```

Skip Primary Keys Loading

Another useful Cayenne reverse engineering property is `<skipPrimaryKeyLoading>`. If you decide to support all relationships at the application layer and avoid their management in database, you'll find useful to turn off primary keys synchronization at all.

```
 <dbimport>
        <skipPrimaryKeyLoading>true</skipPrimaryKeyLoading>
```

```
</dbimport>
```

Table Types

By default, cdbimport imports tables and views. Some databases may support other table-like objects, e.g. `SYSTEM TABLE`, `GLOBAL TEMPORARY`, `LOCAL TEMPORARY`, `ALIAS`, `SYNONYM`, etc. To control which types should be included `<tableType></tableType>` element is used. Some examples:

Import tables only (skip views and others and other types):

```
<dbimport>
        <tableType>TABLE</tableType>
</dbimport>
```

Tables and views (the default option):

```
 <dbimport>
        <tableType>TABLE</tableType>
        <tableType>VIEWS</tableType>
</dbimport>
```

Appendix A. Configuration Properties

Note that the property names below are defined as constants in `org.apache.cayenne.RuntimeProperties` 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 applies just to this domain/node. Otherwise it applies to all domains/nodes in the system.
    - Default value: none. Only used for the DataNodes that were not given an explicit DataSource
- `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 applies just to this domain/node. Otherwise it applies to all domains/nodes in the system.
    - Default value: none. Only used for the DataNodes that were not given an explicit DataSource
- `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 applies just to this domain/node. Otherwise it applies to all domains/nodes in the system.
    - Possible values: any
    - Default value: none. Only used for the DataNodes that were not given an explicit DataSource
- `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 applies just to this domain/node. Otherwise it applies to all domains/nodes in the system
    - Default value: none. Only used for the DataNodes that were not given an explicit DataSource
- `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 applies just to this domain/node. Otherwise it applies to all domains/nodes in the system
    - Default value: none. Only used for the DataNodes that were not given an explicit DataSource
- `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 applies just to this domain/node. Otherwise it applies to all domains/nodes in the system
    - Default value: none. Only used for the DataNodes that were not given an explicit DataSource
- `cayenne.jdbc.max_wait` defines a maximum time in milliseconds that a connection request could wait in the connection queue. After this period expires, an exception will be thrown in the calling method. A value of zero will make the thread wait until a connection is available with no time out.
    - Default value: 20 seconds
- `cayenne.jdbc.validation_query` defines a SQL string that returns some result. It will be used to validate connections in the pool.
    - Default value: none
- `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.DataRowStore.snapshot.size` defines snapshot cache max size
  - Possible values: any positive int
  - Default value: 10000
- `cayenne.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: false (since 4.1)
- `cayenne.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.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.external_tx` defines whether runtime should use external transactions.
  - Possible values: true, false
  - Default value: false
- `cayenne.query_execution_time_logging_threshold` *(deprecated and ignored)* used to define the minimum number of milliseconds a query must run before it is logged. The slow-query threshold warning was removed with the compact SQL logger redesign (CAY-2912).
- `cayenne.jdbc.log.batch.threshold` defines the maximum number of batch rows whose bindings are logged in full before the compact SQL logger truncates them to "first rows .. elided count .. last rows".
  - Default value: 20
- `cayenne.domain.name` defines an optional name of the runtime DataDomain. If not specified, the name is inferred from the configuration name.
  - Default value: none

Appendix B. Service Collections

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

Table 15. Service Collection Keys Present in CayenneRuntime

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.
`cayenne.adapter_detectors`	`List<DbAdapterDetector>`	Contains objects that can discover the type of current database and install the correct DbAdapter in runtime.
`cayenne.domain_listeners`	`List<Object>`	Stores DataDomain listeners.
`cayenne.project_locations`	`List<String>`	Stores locations of the one of more project configuration files.
`cayenne.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.user_types"`).
`cayenne.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.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.