Cayenne 4.0 New Features and Upgrade Guide

Version 4.0 (4.0)

Table of Contents

Chapter 1. Guide to 4.0 Features

This guide highlights the new features and changes introduced in Apache Cayenne 4.0. For a full list of changes consult RELEASE-NOTES.txt included in Cayenne download. For release-specific upgrade instructions check UPGRADE.txt.

1.1. Java Version

Minimum required JDK version is 1.7 or newer. Cayenne 4.0 is fully tested with Java 1.7, 1.8.

The examples below often use Java 8 syntax. But those same examples should work without lambdas just as well.

1.2. Cayenne Configuration

ServerRuntimeBuilder

Cayenne 3.1 introduced dependency injection and ServerRuntime. 4.0 provides a very convenient utility to create a custom runtime with various extensions. This reduces the code needed to integrate Cayenne in your environment to just a few lines and no boilerplate. E.g.:

```
ServerRuntime runtime = ServerRuntime.builder("myproject")
        .addConfigs("cayenne-project1.xml", "cayenne-project2.xml")
        .addModule(binder -> binder.bind(JdbcEventLogger.class).toInstance(myLogger))
        .dataSource(myDataSource)
        .build();
```

Mapping-free ServerRuntime

ServerRuntime can now be started without any ORM mapping at all. This is useful in situations when Cayenne is used as a stack to execute raw SQL, in unit tests, etc.

DI Container Decorators

In addition to overriding services in DI container, Cayenne now allows to supply decorators. True to the "smallest-footprint" DI philosophy, decorator approach is very simple and does not require proxies or class enhancement. Just implement the decorated interface and provide a constructor that takes a delegate instance being decorated:

```
public class MyInterfaceDecorator implements MyInterface {

    private MyInterface delegate;

    public MockInterface1_Decorator3(@Inject MyInterface delegate) {
        this.delegate = delegate;
    }

    @Override
    public String getName() {
        return "<" + delegate.getName() + ">";
    }
}

Module module = binder ->
        binder.decorate(MyInterface.class).before(MyInterfaceDecorator.class);
```

1.3. Framework API

Fluent Query API

Fluent Query API is one of the most exciting new features in Cayenne 4.0. This syntax is "chainable" so you can write query assembly and execution code on one line. The most useful fluent queries are `ObjectSelect`, `SQLSelect` and `SelectById`:

ObjectSelect

A "chainable" analog of SelectQuery.

```
Artist a = ObjectSelect
     .query(Artist.class)
     .where(Artist.ARTIST_NAME.eq("Picasso"))
     .selectOne(context);
```

ColumnSelect

This query allows you directly access individual properties of Objects and use functions (including aggregate) via type-safe API.

```
List<String> names = ObjectSelect
    .columnQuery(Artist.class, Artist.ARTIST_NAME)
    .where(Artist.ARTIST_NAME.length().gt(6))
    .select(context);
```

SQLSelect

A "chainable" analog of `SQLTemplate` used specifically to run selecting raw SQL:

```
List<Long> ids = SQLSelect
     .scalarQuery(Long.class, "SELECT ARTIST_ID FROM ARTIST ORDER BY ARTIST_ID")
     .select(context);
```

SelectById

There's really no good analog of `SelectById` in Cayenne prior to 4.0. Previously available `ObjectIdQuery` didn't support half of the features of `SelectById` (e.g. caching consistent with other queries, prefetches, etc.) :

```
Artist a = SelectById
     .query(Artist.class, 3)
     .useLocalCache("g1")
     .selectOne(context)
```

ObjectContext

Callback-based Object Iterator

ObjectContext now features a simpler way to iterate over large result sets, based on callback interface that can be implemented with a lambda:

```
SelectQuery<Artist> q = new SelectQuery<Artist>(Artist.class);

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

Generics in Expressions and Queries

We finished the work of "genericizing" Cayenne APIs started in 3.1. Now all APIs dealing with persistent objects (Expressions, Queries, ObjectContext, etc.) support generics of Persistent object type or attribute property type.

Property API

Persistent superclasses (_MyEntity) now contain a set of static `Property<T>` variables generated from the model. These metadata objects make possible to create type-safe Expressions and other query parts.

Positional Parameter Bindings

Expressions and `SQLTemplate` traditionally supported binding of parameters by name as a map. Cayenne 4.0 introduces a very easy form of positional bindings. It works with the same named template format, only parameters are bound by position, left-to-right. Here we showing a more complex example with the same parameter name being used more than once in the query:

```
// two distinct names, 3 positional parameters.
// "price" is set to 23, "maxPrice" - to 50
Expression e = ExpressionFactory.exp(
     "price = $price or averagePrice = $price and maxPrice = $maxPrice", 23, 50);
```

This API is supported in Expressions, SQLTemplate as well as new SQLSelect and can be used interchangeably with named parameters with a single template flavor.

Improved Transaction API

Transaction factory is now setup via DI (instead of being configured in the Modeler). There's a utility method on ServerRuntime to perform multiple operations as one transaction:

```
runtime.performInTransaction(() -> {
    // ... do some changes
    context.commitChanges();

    // ... do more changes
    context.commitChanges();

    return true;
});
```

Transparent Database Cryptography with "cayenne-crypto" Module

Cayenne includes a new module called "cayenne-crypto" that enables transparent cryptography for designated data columns. This is a pretty cool feature that allows to enable encryption/decryption of your sensitive data pretty much declaratively using your regular DB storage. Encrypted values can be stored in (VAR)BINARY and (VAR)CHAR columns. Currently "cayenne-crypto" supports AES/CBC/PKCS5Padding encryption (though other cyphers can be added). It also supports encrypted data compression. Here is an example of building a crypto DI module that can be added to ServerRuntime:

```
Module cryptoExtensions = CryptoModule.extend()
    .keyStore("file:///mykeystore", "keystorepassword".toCharArray(), "keyalias")
    .compress()
    .module();
```

1.4. CayenneModeler

Improved UI

CayenneModeler features a number of UI improvements. Attributes and relationships are now edited in the same view (no need to switch between the tabs). Project tree allows to easily filter elements by type and quickly collapse the tree.

Dropped Support for Mapping Listeners

Managing listeners in the DataMap XML model is counterproductive and confusing, so support for listeners was removed from both the XML and the Modeler. If you previously had listeners mapped in the model, annotate their callback methods, and perform listener registration in the code:

```
runtime.getDataDomain().addListener(myListener);
```

or via DI:

```
Module module = binder -> ServerModule.contributeDomainListeners(myListener);
```

Entity callbacks on the other hand are managed in the Modeler as before.

1.5. Build Tools

cdbimport

`cdbimport` has evolved from an experiment to a full-featured production tool that significantly reduces (and sometimes eliminates) the need for manual maintenance of the DataMaps in CayenneModeler. Two improvements made this possible. First, smart merge algorithm will ensure that custom changes to the model are not overridden on subsequent runs of "cdbimport". Second, the mechanism for specifing DB reverse-engineering parameters, such as name filtering, is made much more powerful with many new options. E.g. we started supporting filters by catalogs and schemas, table name filters can be added per catalog/schema or at the top level, etc.

cgen

As was mentioned above, `cgen` now includes `Property<T>` static variables for expression building. It is also made smarter about its defaults for "destDir" and "superPkg".

Gradle Plugin

Cayenne now provides it's own Gradle Plugin that allows you easily integrate `cdbimport` and `cgen` tools into your build process. Here is example of it's usage:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath group: 'org.apache.cayenne.plugins', name: 'cayenne-gradle-plugin',
version: '4.0'
    }
}

apply plugin: 'org.apache.cayenne'

cayenne.defaultDataMap 'datamap.map.xml'

dependencies {
    compile cayenne.dependency('server')
    compile cayenne.dependency('java8')
}
```