Cayenne New Features and Upgrade Guide

Guide to 3.1 Features 1
1. Distribution Contents Structure 1
2. Cayenne Configuration 1
3. Framework API 2
4. CayenneModeler 3
5. Lifecycle Extensions 3

Guide to 3.1 Features

This guide highlights the new features and changes introduced in 3.1 release. It is a high-level overview. For more details consult **RELEASE-NOTES.txt** file included in each release for the full list of changes, and **UPGRADE.txt** for the release-specific upgrade instructions.

1. Distribution Contents Structure

Cayenne distribution is made leaner and more modular:

- "cayenne-modeler.jar" is no longer included in the "lib" folder, as it is no longer used for loading local JNDI overrides. Of course "CayenneModeler-the-app" is still included.
- Ashwood library used for commit operation sorting is no longer a third-party dependency. Instead a small subset of the relevant Ashwood classes got included in Cayenne core.
- The following helper modules are split away from Cayenne core: "cayenne-project" and "cayenne-wocompat". They are bundled in CayenneModeler, and are available from the source distribution. They are not included as standalone jars in the binary distribution.

2. Cayenne Configuration

> **Note**
>
> The new DI-based bootstrap and configuration approach is not API-compatible with earlier versions of Cayenne. Make sure you read the UPGRADE.txt file for instructions how to upgrade the existing projects.

2.1. Dependency Injection Container

Cayenne 3.1 runtime stack is built around the ideas of Dependency Injection (DI), making it extremely flexible and easy to extend. It bundles a small, flexible annotations-based DI container to configure its services. The container provides DI services and exposes Cayenne extension points, but does not interfere with other DI containers that may be present in the application. I.e. it is invisible to the users who do not care about advanced Cayenne customization.

2.2. Bootstrapping Cayenne in Various Environments

Here is a simple example of starting a server-side Cayenne stack:

```
ServerRuntime runtime = new ServerRuntime("cayenne-UntitledDomain.xml");
```

For more detailed examples check the tutorials and other documentation.

2.3. Configuring Local DataSources, Removal of JNDI Hack

Cayenne 3.1 provides a property-based mechanism to override Modeler DataSource definitions, regardless of whether they are driver configurations, JNDI, DBCP, etc. A quick configuration example is shown below:

```
-Dcayenne.jdbc.driver=com.mysql.jdbc.Driver -Dcayenne.jdbc.url=jdbc:mysql://localhost/mydb \
-Dcayenne.jdbc.username=user -Dcayenne.jdbc.password=password
```

For more details and configuration options see javadocs of org.apache.cayenne.configuration.server.PropertyDataSourceFactory.

This feature supersedes what was formerly known as "JNDI hack", i.e. JNDI DataSource failover load strategy based on CayenneModeler preferences database. The problem with JNDI hack was unstable and frequently corrupted preferences database, and the need to include hsqldb and cayenne-modeler jars in the runtime.

3. Framework API

See UPGRADE.txt for the full list of changes

3.1. Lifecycle Listener Annotations

Cayenne 3.1 features support for annotations on lifecycle listeners (but not yet on entity callback methods) that simplifies registering listeners via API. Our experience with Cayenne 3.0 shows that mapping listeners in the Modeler doesn't scale well to complex applications, and 3.0 API for mapping the listeners is hard to use. In 3.1 you can annotate listener methods and register multiple callback methods with a single call.

```
// declare a listener with annotated methods
class MyListener {
    @PostLoad(Entity1.class)
    @PostPersist(Entity1.class)
    void postLoad(Object object) {
        ....
    }
}

// register a listener
ServerRuntime runtime = ..
MyListener listener = new MyListener();
runtime.getChannel().getEntityResolver().getCallbackRegistry().addListener(listener);
```

Moreover, unlike JPA annotations, Cayenne allows to attach a listener to a set of entities not known to the listener upfront, but that are all annotated with some custom annotation:

```
class MyListener {
    @PostLoad(entityAnnotations = CustomAnnotation.class)
    void postLoad(Object object) {
        ....
    }
}
```

3.2. DataChannelFilter for Intercepting DataDomain Operations

Cayenne now features a DataChannelFilter interface that allows to intercept and alter all DataChannel traffic (i.e. selects and commits between a DataContext and DataDomain). It provides a chain of command API very similar to servlet filters. Filters are widely used by "cayenne-lifecyle" extensions and allow to build powerful custom object lifecycle-aware code. To install a filter, the following API is used:

```
class MyFilter implement DataChannelFilter { .. }

MyFilter filter = new MyFilter();
ServerRuntime runtime = ..
runtime.getDataDomain().addFilter(filter);
```

Very often filters mark some of their own methods with lifecycle annotations so that certain operations can be triggered by Cayenne inside the scope of filter's onQuery() or onSync() methods. To ensure annotated methods are invoked, filter registration should be combined with listener registration:

```
MyFilter filter = new MyFilter();
ServerRuntime runtime = ..
runtime.getDataDomain().addFilter(filter);
runtime.getDataDomain().getEntityResolver().getCallbackRegistry().addListener(filter);
// noticed that by default runtime.getDataDomain() is equivalent to runtime.getChannel()
```

4. CayenneModeler

4.1. Java Preferences API

We got rid of HSQLDB-based preferences storage, and are using standard Java Preferences API for the Modeler preferences. This solved a long-standing stability issue with Modeler preferences. So no more lost user preferences.

5. Lifecycle Extensions

Cayenne 3.1 includes an optional cayenne-lifecyle module that implements a few useful extensions based on DataChannelFilters and lifecycle annotations. Those include a concept of a String ID (which is a String URL-friendly representation of ObjectId), support for (de)referencing objects by String ID, String ID-based relationships, annotation-based cache groups invalidation, annotation-based audit of object changes, etc.