Getting Started with Cayenne

Version 4.0 (4.0)

Table of Contents

1. Setting up the environment . . . 2
    1.1. Install Java . . . 2
    1.2. Install IntelliJ IDEA . . . 2
2. Learning mapping basics . . . 3
    2.1. Starting a project . . . 3
    2.2. Getting started with Object Relational Mapping (ORM) . . . 7
    2.3. Creating Java Classes . . . 10
3. Learning Cayenne API . . . 13
    3.1. Getting started with ObjectContext . . . 13
    3.2. Getting started with persistent objects . . . 14
    3.3. Selecting Objects . . . 17
    3.4. Deleting Objects . . . 18
4. Converting to Web Application . . . 21
    4.1. Converting Tutorial to a Web Application . . . 21

Chapter 1. Setting up the environment

The goal of this chapter of the tutorial is to install (or check that you already have installed) a minimally needed set of software to build a Cayenne application.

1.1. Install Java

Obviously, JDK has to be installed. Cayenne 4.0 requires JDK 1.7 or newer.

1.2. Install IntelliJ IDEA

Download and install IntelliJ IDEA Community Edition. This tutorial is based on version 2016.3, still it should work with any recent IntelliJ IDEA version.

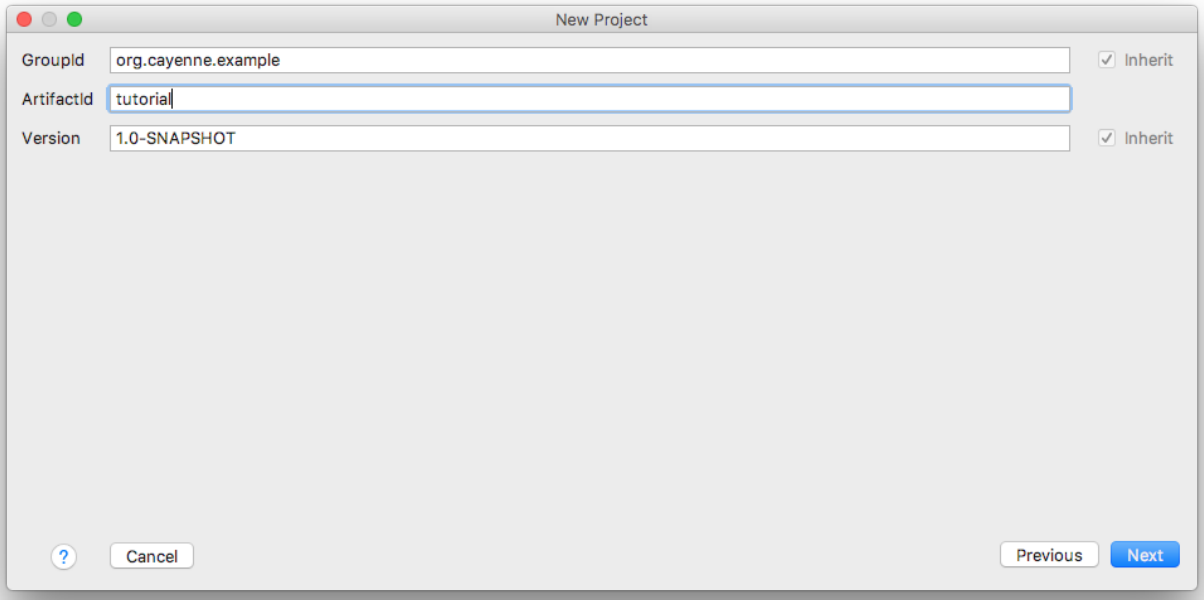
Chapter 2. Learning mapping basics

2.1. Starting a project

The goal of this chapter is to create a new Java project in IntelliJ IDEA containing a basic Cayenne mapping. It presents an introduction to CayenneModeler GUI tool, showing how to create the initial mapping objects: `DataDomain`, `DataNode`, `DataMap`.

Create a new Project in IntelliJ IDEA

In IntelliJ IDEA select `File > New > Project..` and then select `Maven` and click `Next`. In the dialog shown on the screenshot below, fill the `Group Id` and `Artifact Id` fields and click `Next`.



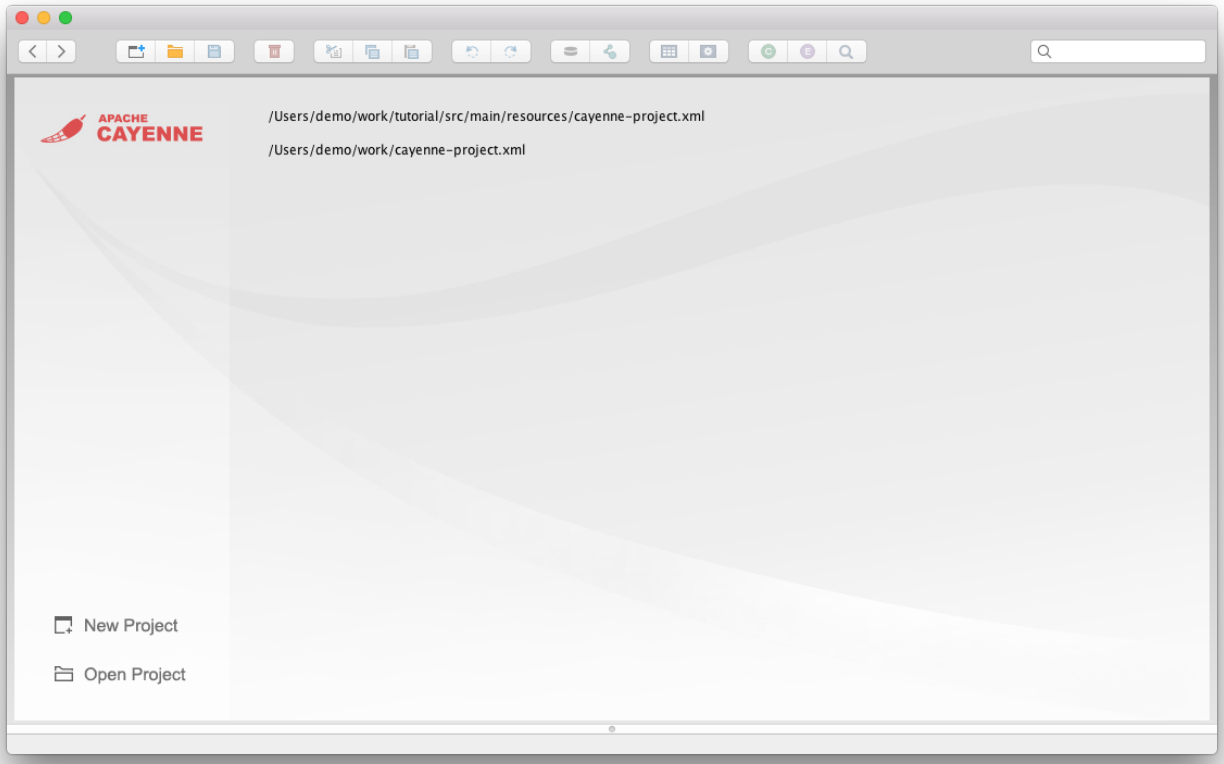
On next dialog screen you can customize directory for your project and click `Finish`. Now you should have a new empty project.

Download and Start CayenneModeler

Although later in this tutorial we'll be using Maven to include Cayenne runtime jars in the project, you'll still need to download Cayenne to get access to the CayenneModeler tool.

If you are really into Maven, you can start CayenneModeler from Maven too. We'll do it in a more traditional way here.

Download the latest release. Unpack the distribution somewhere in the file system and start CayenneModeler, following platform-specific instructions. On most platforms it is done simply by doubleclicking the Modeler icon. The welcome screen of the Modeler looks like this:



Create a New Mapping Project in CayenneModeler

Click on the `New Project` button on Welcome screen. A new mapping project will appear that contains a single **DataDomain**. The meaning of a DataDomain is explained elsewhere in the User Guide. For now it is sufficient to understand that DataDomain is the root of your mapping project.

Create a DataNode

The next project object you will create is a **DataNode**. DataNode is a descriptor of a single database your application will connect to. Cayenne mapping project can use more than one database, but for now, we'll only use one. With "project" selected on the left, click on `Create DataNode` button on the toolbar (or select `Project > Create DataNode` from the menu).

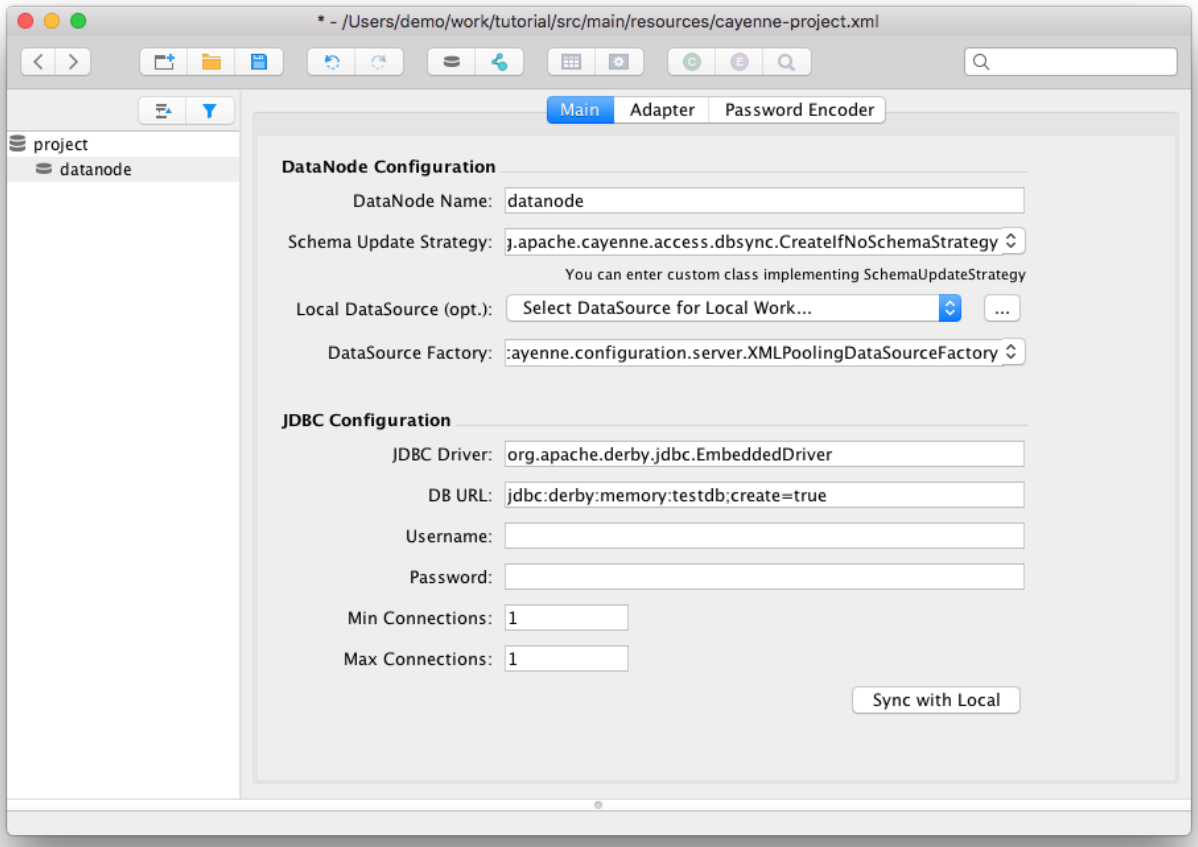
A new DataNode is displayed. Now you need to specify JDBC connection parameters. For an in-memory Derby database you can enter the following settings:

- JDBC Driver: org.apache.derby.jdbc.EmbeddedDriver
- DB URL: jdbc:derby:memory:testdb;create=true

We are creating an in-memory database here. So when you stop your application, all the data will be lost. In most real-life cases you'll be connecting to a database that actually persists its data on disk, but an in-memory DB will do for the simple tutorial.

Also you will need to change "Schema Update Strategy". Select `org.apache.cayenne.access.dbsync.CreateIfNoSchemaStrategy` from the dropdown, so that Cayenne

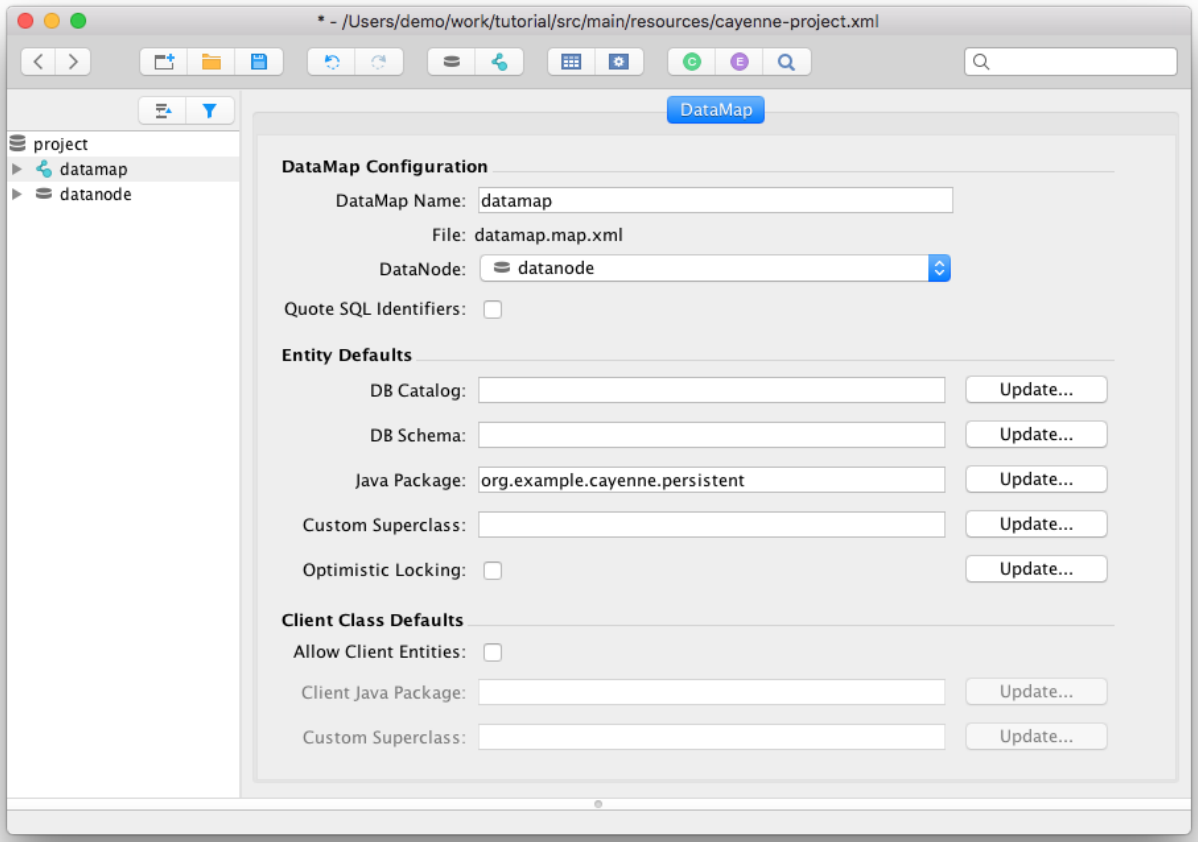
creates a new schema on Derby based on the ORM mapping when the application starts.



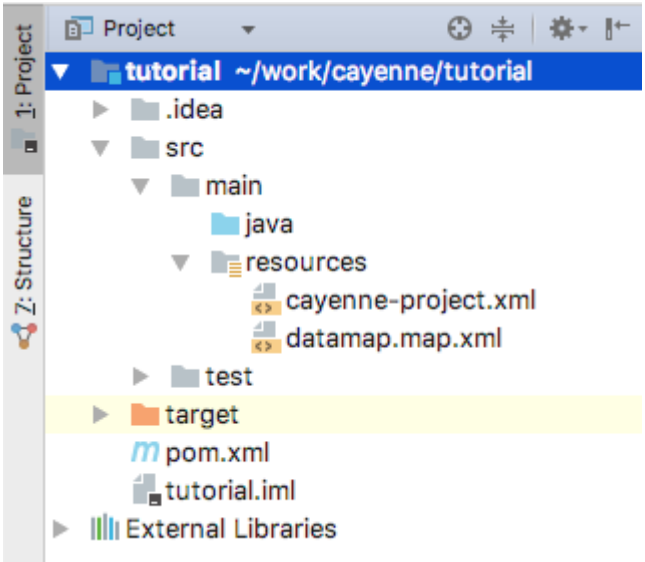
Create a DataMap

Now you will create a **DataMap**. DataMap is an object that holds all the mapping information. To create it, click on "Create DataMap" button (or select a corresponding menu item). Note that the newly created DataMap is automatically linked to the DataNode that you created in the previous step. If there is more than one DataNode, you may need to link a DataMap to the correct node manually. In other words a DataMap within DataDomain must point to a database described by the map.

You can leave all the DataMap defaults unchanged except for one - "Java Package". Enter `org.example.cayenne.persistent`. This name will later be used for all persistent classes.



Save the Project



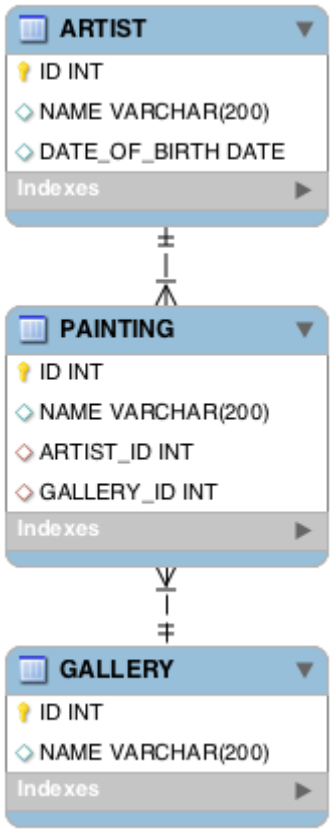
Before you proceed with the actual mapping, let's save the project. Click on "Save" button in the toolbar and navigate to the `tutorial` IDEA project folder that was created earlier in this section and its `src/main/resources` subfolder and save the project there. Now go back to IDEA and you will see two Cayenne XML files.

Note that the location of the XML files is not coincidental. Cayenne runtime looks for `cayenne-*.xml` file in the application `CLASSPATH` and `src/main/resources` folder should already be a "class folder" in IDEA for our project (and is also a standard location that Maven would copy to a jar file, if we were

using Maven from command-line).

2.2. Getting started with Object Relational Mapping (ORM)

The goal of this section is to learn how to create a simple Object-Relational model with CayenneModeler. We will create a complete ORM model for the following database schema:

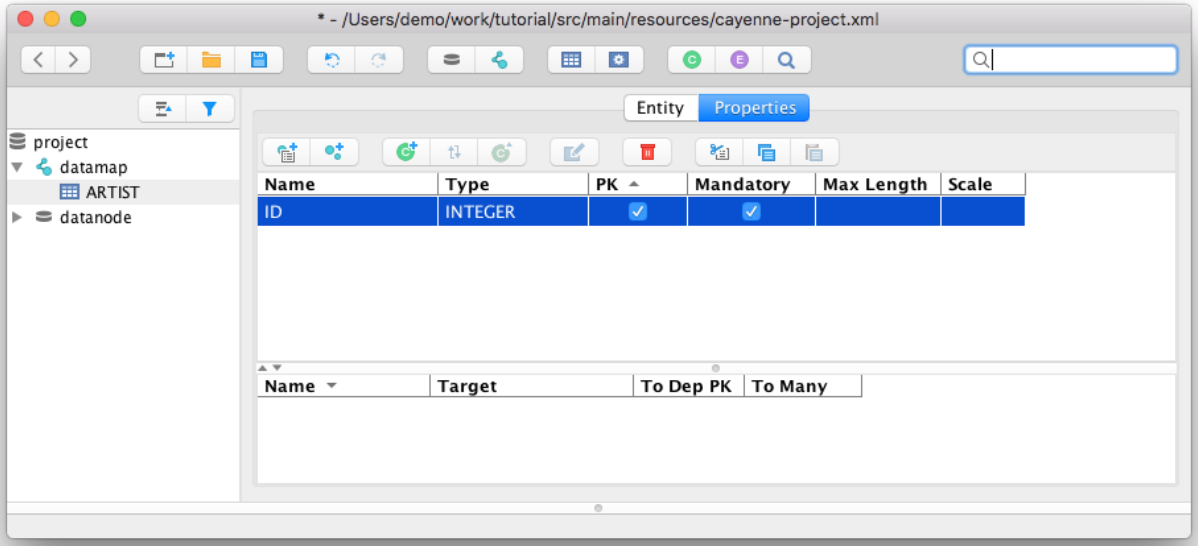


Very often you'd have an existing database already, and it can be quickly imported in Cayenne via "Tools > Reengineer Database Schema". This will save you lots of time compared to manual mapping. However understanding how to create the mapping by hand is important, so we are showing the "manual" approach below.

Mapping Database Tables and Columns

Lets go back to CayenneModeler where we have the newly created project open and start by adding the ARTIST table. Database tables are called **DbEntities** in Cayenne mapping (those can be actual tables or database views).

Select "datamap" on the left-hand side project tree and click "Create DbEntity" button  (or use "Project > Create DbEntity" menu). A new DbEntity is created. In "DbEntity Name" field enter "ARTIST". Then click on "Create Attribute" button  on the entity toolbar. This action changes the view to the "Attribute" tab and adds a new attribute (attribute means a "table column" in this case) called "untitledAttr". Let's rename it to ID, make it an `INTEGER` and make it a PK:

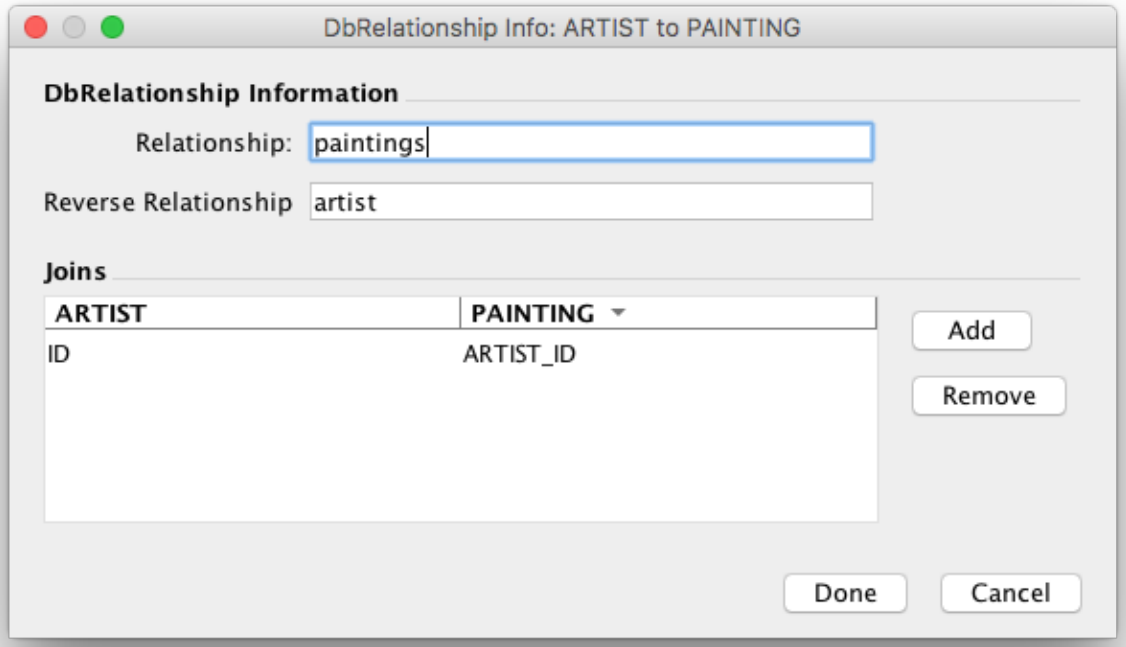


Similarly add NAME `VARCHAR(200)` and DATE_OF_BIRTH `DATE` attributes. After that repeat this procedure for PAINTING and GALLERY entities to match DB schema shown above.

Mapping Database Relationships

Now we need to specify relationships between ARTIST, PAINTING and GALLERY tables. Start by creating a one-to-many ARTIST/PAINTING relationship:

- Select the ARTIST DbEntity on the left and click on the "Properties" tab.
- Click on "Create Relationship" button on the entity toolbar - a relationship called "untitledRel" is created.
- Choose the "Target" to be "Painting".
- Click on the "Database Mapping" button - relationship configuration dialog is presented. Here you can assign a name to the relationship and also its complimentary reverse relationship. This name can be anything (this is really a symbolic name of the database referential constraint), but it is recommended to use a valid Java identifier, as this will save some typing later. We'll call the relationship "paintings" and reverse relationship "artist".
- Click on "Add" button on the right to add a join
- Select "ID" column for the "Source" and "ARTIST_ID" column for the target.
- Relationship information should now look like this:



- Click "Done" to confirm the changes and close the dialog.
- Two complimentary relationships have been created - from ARTIST to PAINTING and back. Still you may have noticed one thing is missing - "paintings" relationship should be to-many, but "To Many" checkbox is not checked. Let's change that - check the checkbox for "paintings" relationship, and then click on PAINTING DbEntity, and uncheck "artist" relationship "To Many" to make the reverse relationship "to-one" as it should be.
- Repeat the steps above to create a many-to-one relationship from PAINTING to GALLERY, calling the relationships pair "gallery" and "paintings".

Mapping Java Classes

Now that the database schema mapping is complete, CayenneModeler can create mappings of Java classes (aka "ObjEntities") by deriving everything from DbEntities. At present there is no way to do it for the entire DataMap in one click, so we'll do it for each table individually.

- Select "ARTIST" DbEntity and click on "Create ObjEntity" button either on the entity toolbar or on the main toolbar. An ObjEntity called "Artist" is created with a Java class field set to "org.example.cayenne.persistent.Artist". The modeler transformed the database names to the Java-friendly names (e.g., if you click on the "Attributes" tab, you'll see that "DATE_OF_BIRTH" column was converted to "dateOfBirth" Java class attribute).
- Select "GALLERY" DbEntity and click on "Create ObjEntity" button again - you'll see a "Gallery" ObjEntity created.
- Finally, do the same thing for "PAINTING".

Now you need to synchronize relationships. Artist and Gallery entities were created when there was no related "Painting" entity, so their relationships were not set.

- Click on the "Artist" ObjEntity. Now click on "Sync ObjEntity with DbEntity" button on the toolbar - you will see the "paintings" relationship appear.
- Do the same for the "Gallery" entity.

Unless you want to customize the Java class and property names (which you can do easily) the mapping is complete.

2.3. Creating Java Classes

Here we'll generate the Java classes from the model that was created in the previous section. CayenneModeler can be used to also generate the database schema, but since we specified "CreateIfNoSchemaStrategy" earlier when we created a DataNode, we'll skip the database schema step. Still be aware that you can do it if you need to via "Tools > Create Database Schema".

Creating Java Classes

- Select "Tools > Generate Classes" menu.
- For "Type" select "Standard Persistent Objects", if it is not already selected.
- For the "Output Directory" select "src/main/java" folder under your IDEA project folder (this is a "peer" location to the `cayenne-*.xml` location we selected before).
- Click on "Classes" tab and check the "Check All Classes" checkbox (unless it is already checked and reads "Uncheck all Classes").
- Click "Generate"

Now go back to IDEA - you should see pairs of classes generated for each mapped entity. You probably also see that there's a bunch of red squiggles next to the newly generated Java classes in IDEA. This is because our project does not include Cayenne as a Maven dependency yet. Let's fix it now by adding "cayenne-server" and "cayenne-java8" artifacts in the bottom of the `pom.xml` file. Also we should tell Maven compile plugin that our project needs Java 8. The resulting POM should look like this:

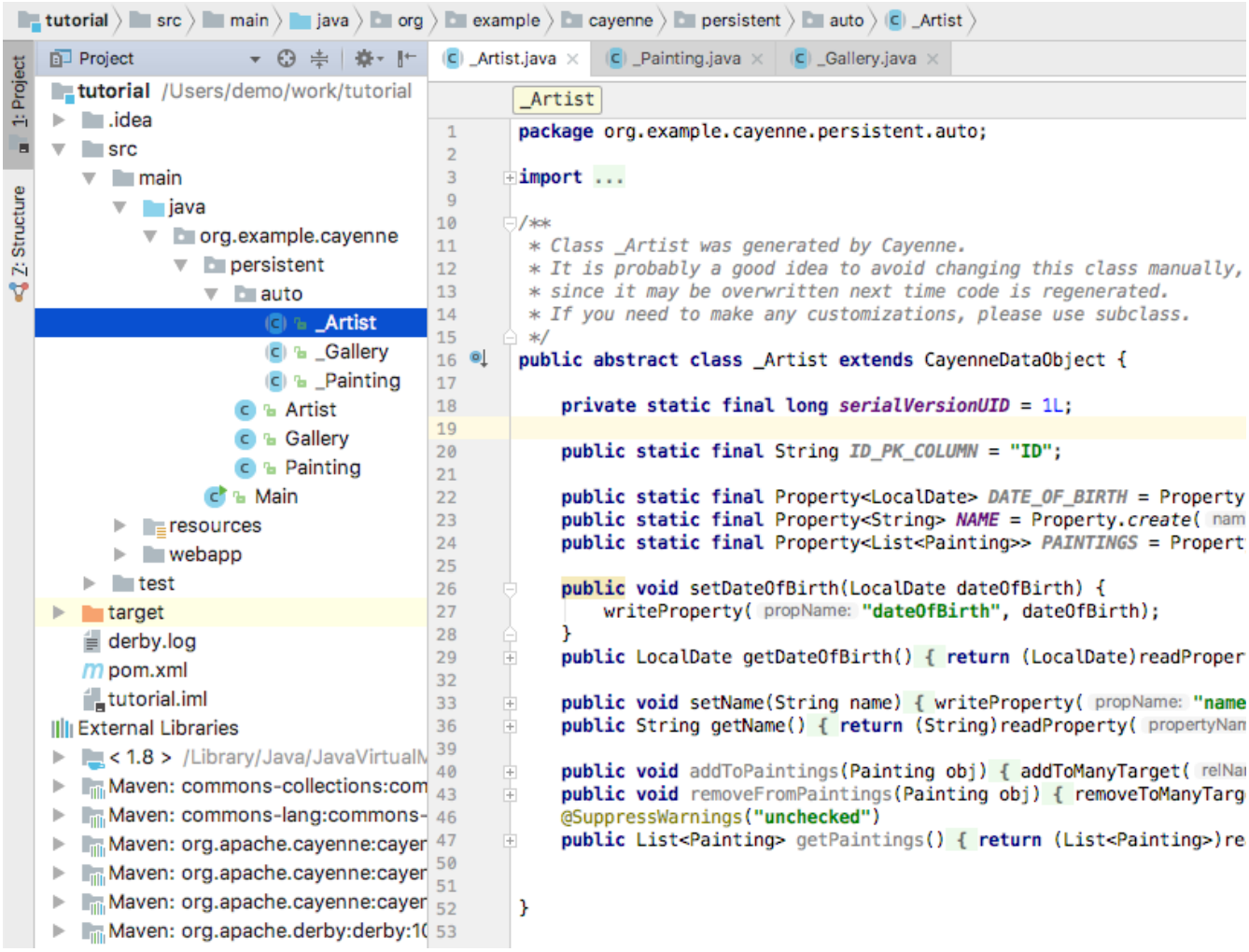
```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.example.cayenne</groupId>
    <artifactId>tutorial</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <properties>
        <cayenne.version>4.0</cayenne.version> ①
        <maven.compiler.source>1.8</maven.compiler.source> ②
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.apache.cayenne</groupId>
            <artifactId>cayenne-server</artifactId>
            <version>${cayenne.version}</version>
        </dependency>
        <!--  For java.time.* types you need to use this dependency-->
        <dependency>
            <groupId>org.apache.cayenne</groupId>
            <artifactId>cayenne-java8</artifactId>
            <version>${cayenne.version}</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
            <version>1.7.25</version>
        </dependency>
    </dependencies>
</project>
```

① Here you can specify the version of Cayenne you are actually using

② Tell Maven to support Java 8

Your computer must be connected to the internet. Once you edit the `pom.xml`, IDEA will download the needed Cayenne jar file and add it to the project build path. As a result, all the errors should disappear. In tutorial for console output we use slf4j-simple logger implementation. Due to use SLF4J logger in Apache Cayenne, you can use your custom logger (e.g. log4j or commons-logging) through bridges.



Now let's check the entity class pairs. Each one is made of a superclass (e.g. `_Artist`) and a subclass (e.g. `Artist`). You **should not** modify the superclasses whose names start with "_" (underscore), as they will be replaced on subsequent generator runs. Instead all custom logic should be placed in the subclasses in `org.example.cayenne.persistent` package - those will never be overwritten by the class generator.

Class Generation Hint

Often you'd start by generating classes from the Modeler, but at the later stages of the project the generation is usually automated either via Ant cgen task or Maven cgen mojo. All three methods are interchangeable, however Ant and Maven methods would ensure that you never forget to regenerate classes on mapping changes, as they are integrated into the build cycle.

Chapter 3. Learning Cayenne API

3.1. Getting started with ObjectContext

In this section we'll write a simple main class to run our application, and get a brief introduction to Cayenne ObjectContext.

Creating the Main Class

- In IDEA create a new class called "Main" in the "org.example.cayenne" package.
- Create a standard "main" method to make it a runnable class:

```
package org.example.cayenne;

public class Main {

    public static void main(String[] args) {
    }
}
```

- The first thing you need to be able to access the database is to create a `ServerRuntime` object (which is essentially a wrapper around Cayenne stack) and use it to obtain an instance of an `ObjectContext`.

```
package org.example.cayenne;

import org.apache.cayenne.ObjectContext;
import org.apache.cayenne.configuration.server.ServerRuntime;

public class Main {

    public static void main(String[] args) {
        ServerRuntime cayenneRuntime = ServerRuntime.builder()
                        .addConfig("cayenne-project.xml")
                        .build();
        ObjectContext context = cayenneRuntime.newContext();
    }
}
```

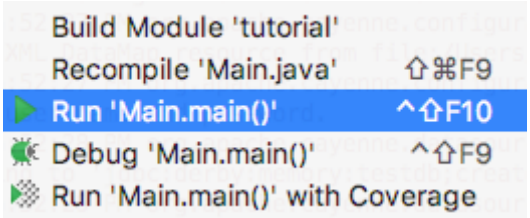
`ObjectContext` is an isolated "session" in Cayenne that provides all needed API to work with data. ObjectContext has methods to execute queries and manage persistent objects. We'll discuss them in the following sections. When the first ObjectContext is created in the application, Cayenne loads XML mapping files and creates a shared access stack that is later reused by other ObjectContexts.

Running Application

Let's check what happens when you run the application. But before we do that we need to add another dependency to the `pom.xml` - Apache Derby, our embedded database engine. The following piece of XML needs to be added to the `<dependencies>⋯</dependencies>` section, where we already have Cayenne jars:

```
<dependency>
   <groupId>org.apache.derby</groupId>
   <artifactId>derby</artifactId>
   <version>10.13.1.1</version>
</dependency>
```

Now we are ready to run. Right click the "Main" class in IDEA and select "Run 'Main.main()'".



In the console you'll see output similar to this, indicating that Cayenne stack has been started:

```
INFO: Loading XML configuration resource from file:/.../cayenne-project.xml
INFO: Loading XML DataMap resource from file:/.../datamap.map.xml
INFO: loading user name and password.
INFO: Connecting to 'jdbc:derby:memory:testdb;create=true' as 'null'
INFO: +++ Connecting: SUCCESS.
INFO: setting DataNode 'datanode' as default, used by all unlinked DataMaps</screen>
```

How to Configure Cayenne Logging

Follow the instructions in the logging chapter to tweak verbosity of the logging output.

3.2. Getting started with persistent objects

In this chapter we'll learn about persistent objects, how to customize them and how to create and save them in DB.

Inspecting and Customizing Persistent Objects

Persistent classes in Cayenne implement a DataObject interface. If you inspect any of the classes generated earlier in this tutorial (e.g. `org.example.cayenne.persistent.Artist`), you'll see that it extends a class with the name that starts with underscore (`org.example.cayenne.persistent.auto._Artist`), which in turn extends from `org.apache.cayenne.CayenneDataObject`. Splitting each persistent class into user-customizable subclass (`Xyz`) and a generated superclass (`_Xyz`) is a useful technique to avoid overwriting the

custom code when refreshing classes from the mapping model.

Let's for instance add a utility method to the Artist class that sets Artist date of birth, taking a string argument for the date. It will be preserved even if the model changes later:

```
public class Artist extends _Artist {

    static final String DEFAULT_DATE_FORMAT = "yyyyMMdd";

    /**
     * Sets date of birth using a string in format yyyyMMdd.
     */
    public void setDateOfBirthString(String yearMonthDay) {
        if (yearMonthDay == null) {
            setDateOfBirth(null);
        } else {

            LocalDate date;
            try {
                DateTimeFormatter formatter = DateTimeFormatter
                        .ofPattern(DEFAULT_DATE_FORMAT);
                date = LocalDate.parse(yearMonthDay, formatter);
            } catch (DateTimeParseException e) {
                throw new IllegalArgumentException(
                        "A date argument must be in format '"
                                + DEFAULT_DATE_FORMAT + "': " + yearMonthDay);
            }
            setDateOfBirth(date);
        }
    }
}
```

Create New Objects

Now we'll create a bunch of objects and save them to the database. An object is created and registered with `ObjectContext` using "newObject" method. Objects **must** be registered with `DataContext` to be persisted and to allow setting relationships with other objects. Add this code to the "main" method of the Main class:

```
Artist picasso = context.newObject(Artist.class);
picasso.setName("Pablo Picasso");
picasso.setDateOfBirthString("18811025");
```

Note that at this point "picasso" object is only stored in memory and is not saved in the database. Let's continue by adding a Metropolitan Museum "Gallery" object and a few Picasso "Paintings":

```
Gallery metropolitan = context.newObject(Gallery.class);
metropolitan.setName("Metropolitan Museum of Art");

Painting girl = context.newObject(Painting.class);
girl.setName("Girl Reading at a Table");

Painting stein = context.newObject(Painting.class);
stein.setName("Gertrude Stein");
```

Now we can link the objects together, establishing relationships. Note that in each case below relationships are automatically established in both directions (e.g. `picasso.addToPaintings(girl)` has exactly the same effect as `girl.setToArtist(picasso)`).

```
picasso.addToPaintings(girl);
picasso.addToPaintings(stein);

girl.setGallery(metropolitan);
stein.setGallery(metropolitan);
```

Now lets save all five new objects, in a single method call:

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

Now you can run the application again as described in the previous chapter. The new output will show a few actual DB operations:

```
...
INFO: --- transaction started.
INFO: No schema detected, will create mapped tables
INFO: CREATE TABLE GALLERY (ID INTEGER NOT NULL, NAME VARCHAR (200), PRIMARY KEY (ID))
INFO: CREATE TABLE ARTIST (DATE_OF_BIRTH DATE, ID INTEGER NOT NULL, NAME VARCHAR
(200), PRIMARY KEY (ID))
INFO: CREATE TABLE PAINTING (ARTIST_ID INTEGER, GALLERY_ID INTEGER, ID INTEGER NOT
NULL,
      NAME VARCHAR (200), PRIMARY KEY (ID))
INFO: ALTER TABLE PAINTING ADD FOREIGN KEY (ARTIST_ID) REFERENCES ARTIST (ID)
INFO: ALTER TABLE PAINTING ADD FOREIGN KEY (GALLERY_ID) REFERENCES GALLERY (ID)
INFO: CREATE TABLE AUTO_PK_SUPPORT (
      TABLE_NAME CHAR(100) NOT NULL,  NEXT_ID BIGINT NOT NULL,  PRIMARY
KEY(TABLE_NAME))
INFO: DELETE FROM AUTO_PK_SUPPORT WHERE TABLE_NAME IN ('ARTIST', 'GALLERY',
'PAINTING')
INFO: INSERT INTO AUTO_PK_SUPPORT (TABLE_NAME, NEXT_ID) VALUES ('ARTIST', 200)
INFO: INSERT INTO AUTO_PK_SUPPORT (TABLE_NAME, NEXT_ID) VALUES ('GALLERY', 200)
INFO: INSERT INTO AUTO_PK_SUPPORT (TABLE_NAME, NEXT_ID) VALUES ('PAINTING', 200)
INFO: SELECT NEXT_ID FROM AUTO_PK_SUPPORT WHERE TABLE_NAME = ? FOR UPDATE [bind:
1:'ARTIST']
INFO: SELECT NEXT_ID FROM AUTO_PK_SUPPORT WHERE TABLE_NAME = ? FOR UPDATE [bind:
1:'GALLERY']
INFO: SELECT NEXT_ID FROM AUTO_PK_SUPPORT WHERE TABLE_NAME = ? FOR UPDATE [bind:
1:'PAINTING']
INFO: INSERT INTO GALLERY (ID, NAME) VALUES (?, ?)
INFO: [batch bind: 1->ID:200, 2->NAME:'Metropolitan Museum of Art']
INFO: === updated 1 row.
INFO: INSERT INTO ARTIST (DATE_OF_BIRTH, ID, NAME) VALUES (?, ?, ?)
INFO: [batch bind: 1->DATE_OF_BIRTH:'1881-10-25 00:00:00.0', 2->ID:200, 3->NAME:'Pablo
Picasso']
INFO: === updated 1 row.
INFO: INSERT INTO PAINTING (ARTIST_ID, GALLERY_ID, ID, NAME) VALUES (?, ?, ?, ?)
INFO: [batch bind: 1->ARTIST_ID:200, 2->GALLERY_ID:200, 3->ID:200, 4->NAME:'Gertrude
Stein']
INFO: [batch bind: 1->ARTIST_ID:200, 2->GALLERY_ID:200, 3->ID:201, 4->NAME:'Girl
Reading at a Table']
INFO: === updated 2 rows.
INFO: +++ transaction committed.
```

So first Cayenne creates the needed tables (remember, we used “CreateIfNoSchemaStrategy”). Then it runs a number of inserts, generating primary keys on the fly. Not bad for just a few lines of code.

3.3. Selecting Objects

This chapter shows how to select objects from the database using `ObjectSelect` query.

Introducing ObjectSelect

It was shown before how to persist new objects. Cayenne queries are used to access already saved objects. The primary query type used for selecting objects is `ObjectSelect`. It can be mapped in CayenneModeler or created via the API. We'll use the latter approach in this section. We don't have too much data in the database yet, but we can still demonstrate the main principles below.

- Select all paintings (the code, and the log output it generates):

```
List<Painting> paintings1 = ObjectSelect.query(Painting.class).select(context);
```

```
INFO: SELECT t0.GALLERY_ID, t0.ARTIST_ID, t0.NAME, t0.ID FROM PAINTING t0
INFO: === returned 2 rows. - took 18 ms.
```

- Select paintings that start with "gi", ignoring case:

```
List<Painting> paintings2 = ObjectSelect.query(Painting.class)
        .where(Painting.NAME.likeIgnoreCase("gi%")).select(context);
```

```
INFO: SELECT t0.GALLERY_ID, t0.NAME, t0.ARTIST_ID, t0.ID FROM PAINTING t0 WHERE
UPPER(t0.NAME) LIKE UPPER(?)
  [bind: 1->NAME:'gi%'] - prepared in 6 ms.
INFO: === returned 1 row. - took 18 ms.
```

- Select all paintings done by artists who were born more than a 100 years ago:

```
List<Painting> paintings3 = ObjectSelect.query(Painting.class)
        .where(Painting.ARTIST.dot(Artist.DATE_OF_BIRTH).lt(LocalDate.of(1900,1,1)))
        .select(context);
```

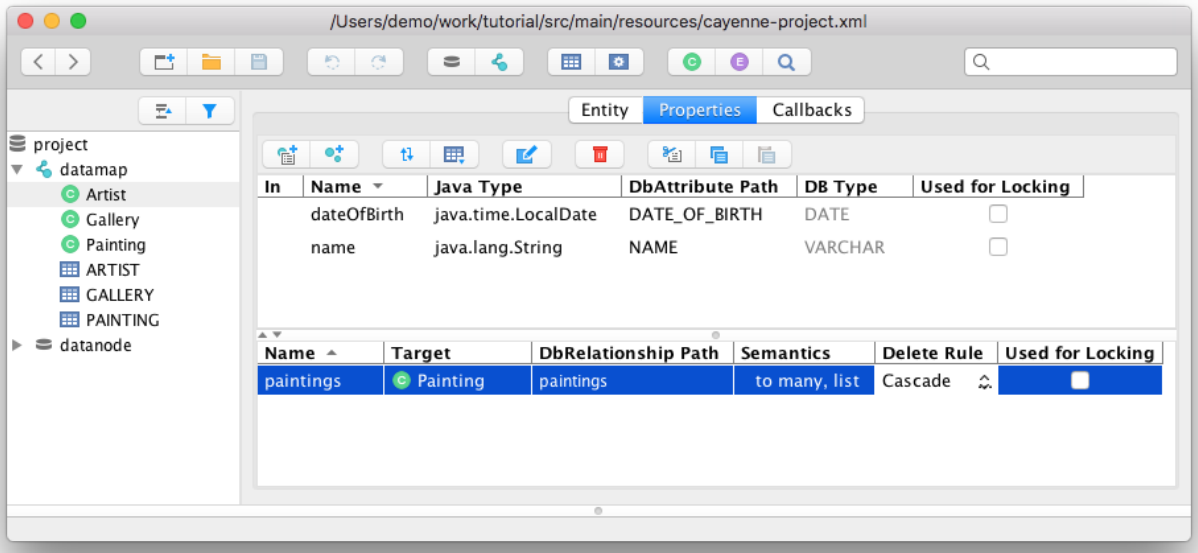
```
INFO: SELECT t0.GALLERY_ID, t0.NAME, t0.ARTIST_ID, t0.ID FROM PAINTING t0 JOIN ARTIST
t1 ON (t0.ARTIST_ID = t1.ID)
  WHERE t1.DATE_OF_BIRTH < ? [bind: 1->DATE_OF_BIRTH:'1911-01-01 00:00:00.493'] -
prepared in 7 ms.
INFO: === returned 2 rows. - took 25 ms.
```

3.4. Deleting Objects

This chapter explains how to model relationship delete rules and how to delete individual objects as well as sets of objects. Also demonstrated the use of Cayenne class to run a query.

Setting Up Delete Rules

Before we discuss the API for object deletion, lets go back to CayenneModeler and set up some delete rules. Doing this is optional but will simplify correct handling of the objects related to deleted objects. In the Modeler go to "Artist" ObjEntity, "Relationships" tab and select "Cascade" for the "paintings" relationship delete rule:



Repeat this step for other relationships:

- For Gallery set "paintings" relationship to be "Nullify", as a painting can exist without being displayed in a gallery.
- For Painting set both relationships rules to "Nullify".

Now save the mapping.

Deleting Objects

While deleting objects is possible via SQL, qualifying a delete on one or more IDs, a more common way in Cayenne (or ORM in general) is to get a hold of the object first, and then delete it via the context. Let's use utility class Cayenne to find an artist:

```
Artist picasso = ObjectSelect.query(Artist.class)
            .where(Artist.NAME.eq("Pablo Picasso")).selectOne(context);
```

Now let's delete the artist:

```
if (picasso != null) {
    context.deleteObject(picasso);
    context.commitChanges();
}
```

Since we set up "Cascade" delete rule for the Artist.paintings relationships, Cayenne will automatically delete all paintings of this artist. So when your run the app you'll see this output:

```
INFO: SELECT t0.DATE_OF_BIRTH, t0.NAME, t0.ID FROM ARTIST t0
  WHERE t0.NAME = ? [bind: 1->NAME:'Pablo Picasso'] - prepared in 6 ms.
INFO: === returned 1 row. - took 18 ms.
INFO: +++ transaction committed.
INFO: --- transaction started.
INFO: DELETE FROM PAINTING WHERE ID = ?
INFO: [batch bind: 1->ID:200]
INFO: [batch bind: 1->ID:201]
INFO: === updated 2 rows.
INFO: DELETE FROM ARTIST WHERE ID = ?
INFO: [batch bind: 1->ID:200]
INFO: === updated 1 row.
INFO: +++ transaction committed.
```

Chapter 4. Converting to Web Application

This chapter shows how to work with Cayenne in a web application.

4.1. Converting Tutorial to a Web Application

The web part of the web application tutorial is done in JSP, which is the least common denominator of the Java web technologies, and is intentionally simplistic from the UI perspective, to concentrate on Cayenne integration aspect, rather than the interface. A typical Cayenne web application works like this:

- Cayenne configuration is loaded when an application context is started, using a special servlet filter.
- User requests are intercepted by the filter, and the DataContext is bound to the request thread, so the application can access it easily from anywhere.
- The same DataContext instance is reused within a single user session; different sessions use different DataContexts (and therefore different sets of objects). *The context can be scoped differently depending on the app specifics. For the tutorial we'll be using a session-scoped context.*

So let's convert the tutorial that we created to a web application:

- In IDEA under "tutorial" project folder create a new folder `src/main/webapp/WEB-INF`.
- Under `WEB-INF` create a new file `web.xml` (a standard web app descriptor):

web.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE web-app
   PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <display-name>Cayenne Tutorial</display-name>

    <!-- This filter bootstraps ServerRuntime and then provides each request thread
         with a session-bound DataContext. Note that the name of the filter is
important,
         as it points it to the right named configuration file.
    -->
    <filter>
        <filter-name>cayenne-project</filter-name>
        <filter-class>org.apache.cayenne.configuration.web.CayenneFilter</filter-
class>
    </filter>
    <filter-mapping>
        <filter-name>cayenne-project</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

- Create the artist browser page `src/main/webapp/index.jsp` file with the following contents:

webapp/index.jsp

```
<%@ page language="java" contentType="text/html" %>
<%@ page import="org.example.cayenne.persistent.*" %>
<%@ page import="org.apache.cayenne.*" %>
<%@ page import="org.apache.cayenne.query.*" %>
<%@ page import="org.apache.cayenne.exp.*" %>
<%@ page import="java.util.*" %>

<%
    ObjectContext context = BaseContext.getThreadObjectContext();
    List<Artist> artists = ObjectSelect.query(Artist.class)
                .orderBy(Artist.NAME.asc())
                .select(context);
%>

<html>
    <head>
        <title>Main</title>
    </head>
    <body>
        <h2>Artists:</h2>

        <% if(artists.isEmpty()) {%>
        <p>No artists found</p>
        <% } else {
               for(Artist a : artists) {
        %>
        <p><a href="detail.jsp?id=<%=Cayenne.intPKForObject(a)%>"> <%=a.getName()%>
</a></p>
        <%
               }
           } %>
        <hr>
        <p><a href="detail.jsp">Create new artist...</a></p>
    </body>
</html>
```

- Create the artist editor page `src/main/webapp/detail.jsp` with the following content:

webapp/detail.jsp

```
<%@ page language="java" contentType="text/html" %>
<%@ page import="org.example.cayenne.persistent.*" %>
<%@ page import="org.apache.cayenne.*" %>
<%@ page import="org.apache.cayenne.query.*" %>
<%@ page import="java.util.*" %>
<%@ page import="java.text.*" %>
<%@ page import="java.time.format.DateTimeFormatter" %>
```

```
<%
    ObjectContext context = BaseContext.getThreadObjectContext();
    String id = request.getParameter("id");

    // find artist for id
    Artist artist = null;
    if(id != null &amp;amp;&amp;amp; id.trim().length() > 0) {
        artist = SelectById.query(Artist.class,
Integer.parseInt(id)).selectOne(context);
    }

    if("POST".equals(request.getMethod())) {
        // if no id is saved in the hidden field, we are dealing with
        // create new artist request
        if(artist == null) {
            artist = context.newObject(Artist.class);
        }

        // note that in a real application we would so dome validation ...
        // here we just hope the input is correct
        artist.setName(request.getParameter("name"));
        artist.setDateOfBirthString(request.getParameter("dateOfBirth"));

        context.commitChanges();

        response.sendRedirect("index.jsp");
    }

    if(artist == null) {
        // create transient artist for the form response rendering
        artist = new Artist();
    }

    String name = artist.getName() == null ? "" : artist.getName();

    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyyMMdd");
    String dob = artist.getDateOfBirth() == null
                        ? "" :artist.getDateOfBirth().format(formatter);
%>
<html>
    <head>
        <title>Artist Details</title>
    </head>
    <body>
        <h2>Artists Details</h2>
        <form name="EditArtist" action="detail.jsp" method="POST">
            <input type="hidden" name="id" value="<%= id != null ? id : "" %>" />
            <table border="0">
                <tr>
                    <td>Name:</td>
                    <td><input type="text" name="name" value="<%= name %>"/></td>
```

```
                </tr>
                <tr>
                    <td>Date of Birth (yyyyMMdd):</td>
                    <td><input type="text" name="dateOfBirth" value="<%= dob
%>"/></td>
                </tr>
                <tr>
                    <td></td>
                    <td align="right"><input type="submit" value="Save" /></td>
                </tr>
            </table>
        </form>
    </body>
</html>
```

Running Web Application

We need to provide javax servlet-api for our application.

pom.xml

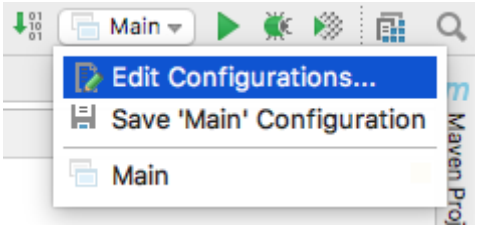
```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <scope>provided</scope>
</dependency>
```

Also to run the web application we'll use "maven-jetty-plugin". To activate it, let's add the following piece of code to the `pom.xml` file, following the "dependencies" section and save the POM:

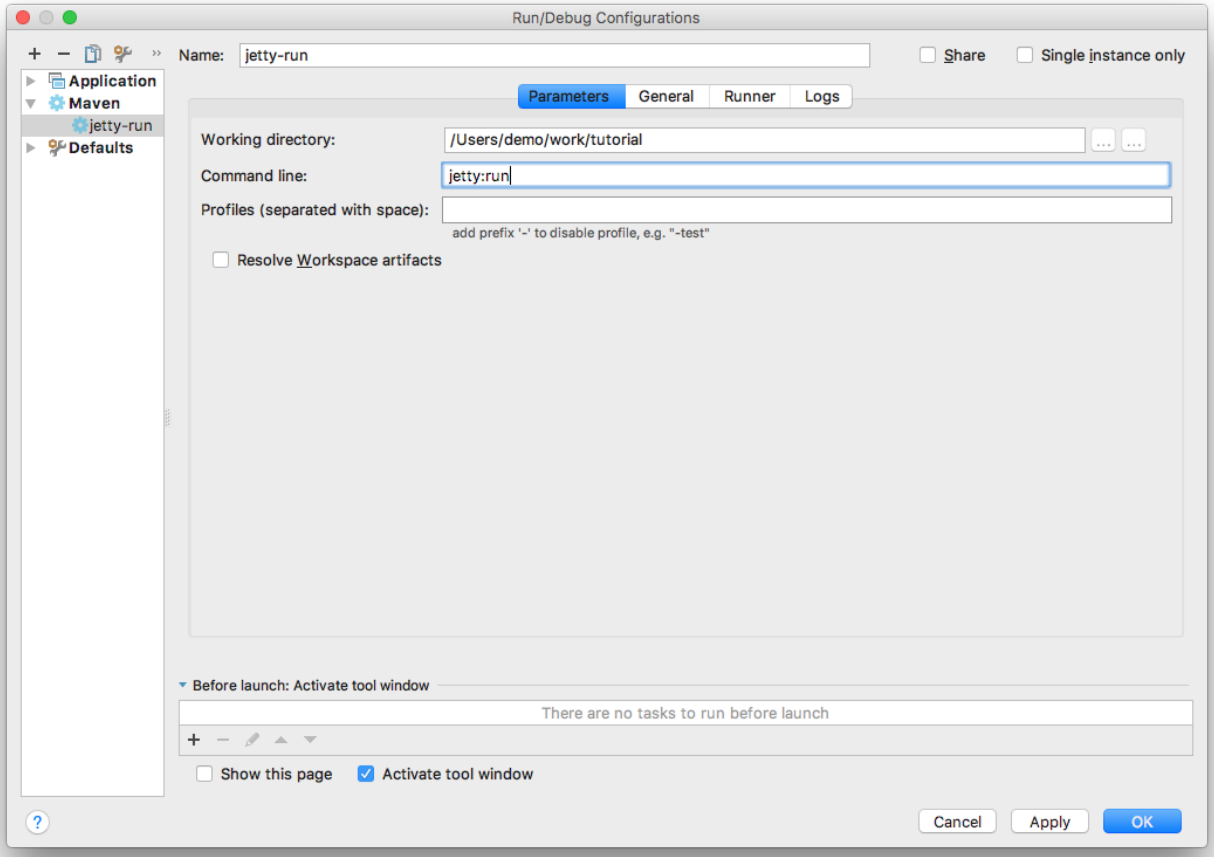
pom.xml

```
<build>
    <plugins>
        <plugin>
            <groupId>org.eclipse.jetty</groupId>
            <artifactId>jetty-maven-plugin</artifactId>
            <version>9.3.14.v20161028</version>
        </plugin>
    </plugins>
</build>
```

- Go to "Select Run/Debug Configuration" menu, and then "Edit Configuration..."



- Click + button and select "Maven". Enter "Name" and "Command line" as shown on screenshot:



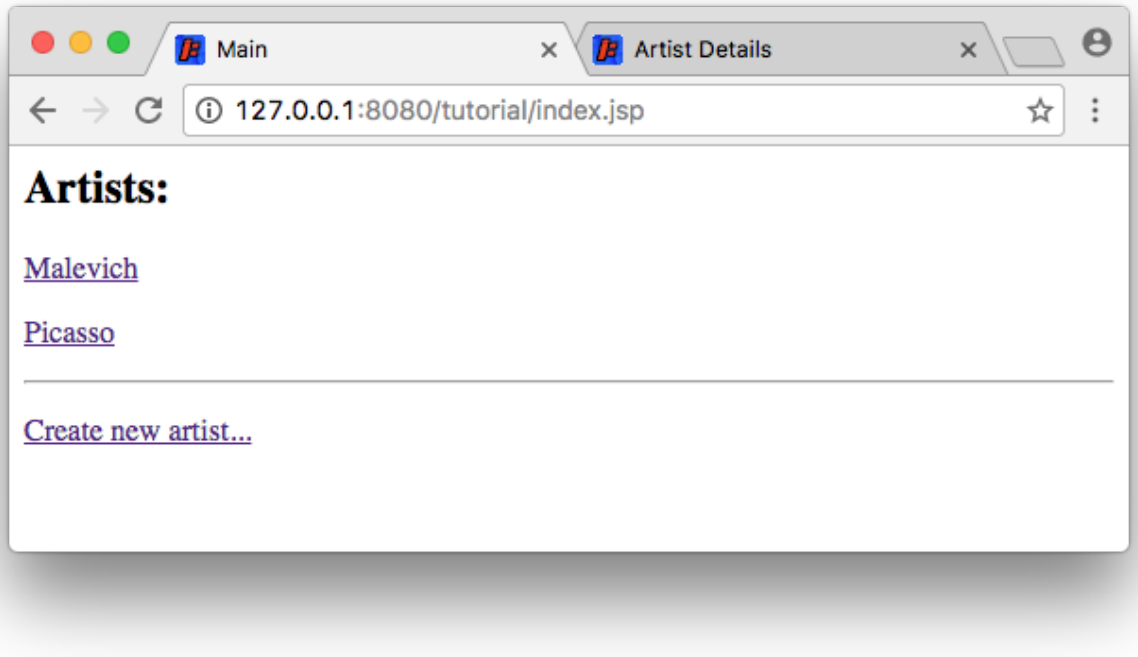
- Click "Apply" and "Run". On the first execution it may take a few minutes for Jetty plugin to download all dependencies, but eventually you'll see the logs like this:

```
[INFO] ------------------------------------------------------------------------
[INFO] Building tutorial 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
...
[INFO] Configuring Jetty for project: tutorial
[INFO] webAppSourceDirectory not set. Trying src/main/webapp
[INFO] Reload Mechanic: automatic
[INFO] Classes = /.../tutorial/target/classes
[INFO] Logging initialized @1617ms
[INFO] Context path = /
[INFO] Tmp directory = /.../tutorial/target/tmp
[INFO] Web defaults = org/eclipse/jetty/webapp/webdefault.xml
[INFO] Web overrides =  none
[INFO] web.xml file = file:/.../tutorial/src/main/webapp/WEB-INF/web.xml
[INFO] Webapp directory = /.../tutorial/src/main/webapp
[INFO] jetty-9.3.0.v20150612
[INFO] Started
o.e.j.m.p.JettyWebAppContext@6872f9c8{/,file:/.../tutorial/src/main/webapp/,AVAILAB
LE}{file:/.../tutorial/src/main/webapp/}
[INFO] Started ServerConnector@723875bc{HTTP/1.1,[http/1.1]}{0.0.0.0:8080}
[INFO] Started @2367ms
[INFO] Started Jetty Server</screen>
```

- So the Jetty container just started.
- Now go to http://localhost:8080/ URL. You should see "No artists found message" in the web browser and the following output in the IDEA console:

```
INFO: Loading XML configuration resource from
file:/.../tutorial/target/classes/cayenne-project.xml
INFO: loading user name and password.
INFO: Connecting to 'jdbc:derby:memory:testdb;create=true' as 'null'
INFO: +++ Connecting: SUCCESS.
INFO: setting DataNode 'datanode' as default, used by all unlinked DataMaps
INFO: Detected and installed adapter: org.apache.cayenne.dba.derby.DerbyAdapter
INFO: --- transaction started.
INFO: No schema detected, will create mapped tables
INFO: CREATE TABLE GALLERY (ID INTEGER NOT NULL, NAME VARCHAR (200), PRIMARY KEY
(ID))
INFO: CREATE TABLE ARTIST (DATE_OF_BIRTH DATE, ID INTEGER NOT NULL, NAME VARCHAR
(200), PRIMARY KEY (ID))
INFO: CREATE TABLE PAINTING (ARTIST_ID INTEGER, GALLERY_ID INTEGER, ID INTEGER NOT
NULL,
      NAME VARCHAR (200), PRIMARY KEY (ID))
INFO: ALTER TABLE PAINTING ADD FOREIGN KEY (ARTIST_ID) REFERENCES ARTIST (ID)
INFO: ALTER TABLE PAINTING ADD FOREIGN KEY (GALLERY_ID) REFERENCES GALLERY (ID)
INFO: CREATE TABLE AUTO_PK_SUPPORT (
      TABLE_NAME CHAR(100) NOT NULL,  NEXT_ID BIGINT NOT NULL,  PRIMARY
KEY(TABLE_NAME))
...
INFO: SELECT t0.DATE_OF_BIRTH, t0.NAME, t0.ID FROM ARTIST t0 ORDER BY t0.NAME
INFO: === returned 0 rows. - took 17 ms.
INFO: +++ transaction committed.</screen>
```

- You can click on "Create new artist" link to create artists. Existing artists can be edited by clicking on their name:



You are done with the tutorial!