Hibernate

An Object-Relational mapping framework

U11CO025

What Hibernate Does

- Object - Relational mapping
- Transparent persistence & retrieval of objects
- Persistence of associations and collections
- Guaranteed uniqueness of an object (within a session)

Hibernate Features

- O-R mapping using ordinary JavaBeans
- Can set attributes using private fields or private setter methods
- Lazy instantiation of collections (configurable)
- Polymorphic queries, object-oriented query language
- Cascading persist & retrieve for associations, including collections and many-to-many
- Transaction management with rollback
- Can integrate with other container-provided services

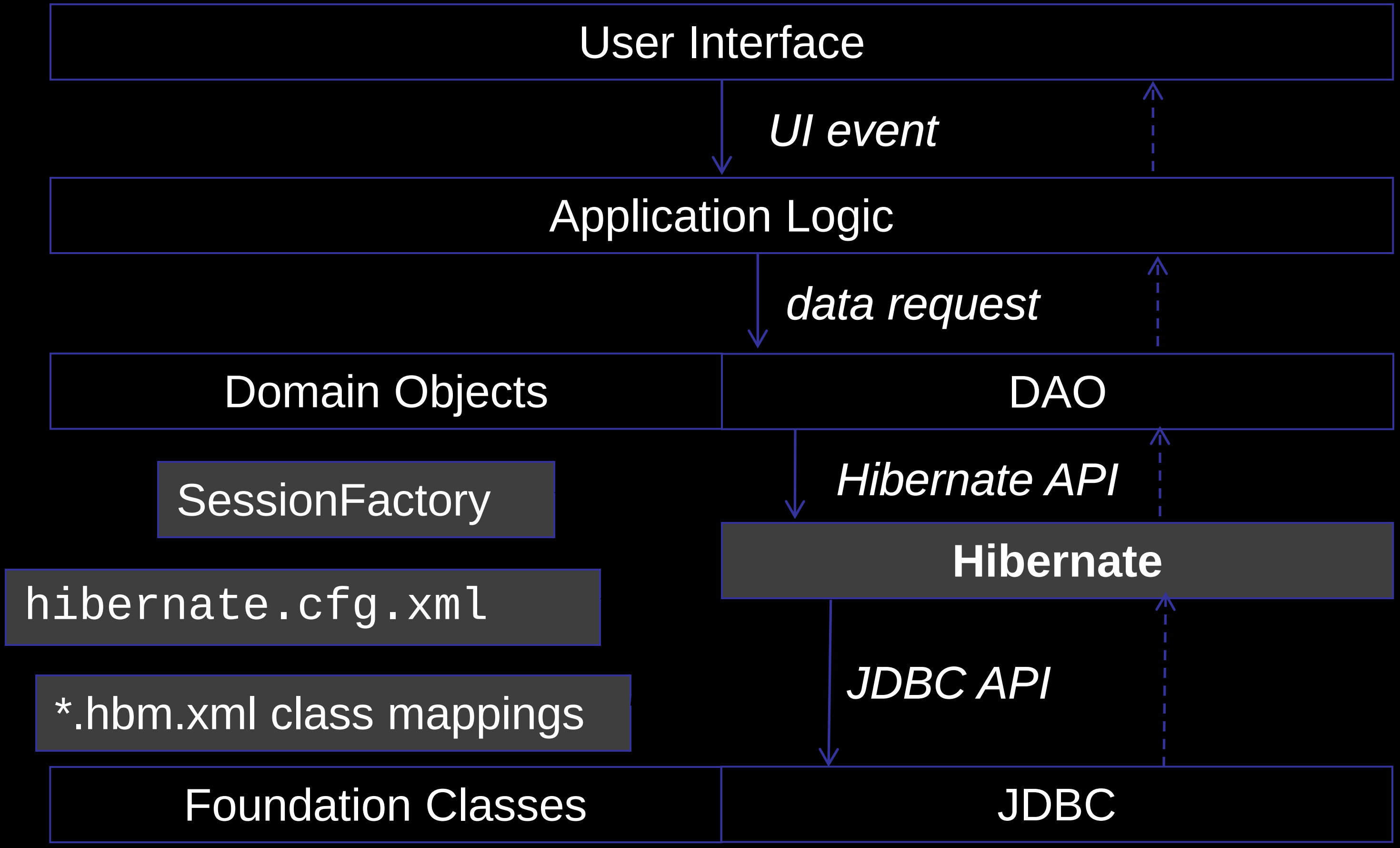
Why Use Hibernate?

- Eliminate need for repetitive SQL
- Work with classes and objects instead of queries and result sets
  - More OO, less procedural
- Mapping approach can resist changes in object/data model more easily
- Strong support for caching

Why Use Hibernate?

- Handles all create-read-update-delete (CRUD) operations using simple API; no SQL
- Generates DDL scripts to create DB schema
- (tables, constraints, sequences)
- Flexibility to hand-tune SQL and call stored procedures to optimize performance
- Supports over 20 RDBMS; change the database by tweaking configuration files

What Hibernate Does



- **SessionFactory**(org.hibernate.SessionFactory)
  - A threadsafe (immutable) cache of compiled mappings for a single database.
  - A factory for Session and a client of ConnectionProvider.
  - Holds an optional (second-level) cache of data that is reusable between transactions, at a process level.
- Session (org.hibernate.Session)
  - A single-threaded, short-lived object representing a conversation between the application and the persistent store.
  - Wraps a JDBC connection. Factory for Transaction. Holds a mandatory (first-level) cache of persistent objects, used when navigating the object graph or looking up objects by identifier.

- Persistent objects and collections
  - Short-lived, single threaded objects containing persistent state and business function.
  - These might be ordinary JavaBeans/POJOs, the only special thing about them is that they are currently associated with (exactly one) Session. As soon as the Session is closed, they will be detached and free to use in any application layer.

- Transaction (org.hibernate.Transaction)
  - A single-threaded, short-lived object used by the application to specify atomic units of work.
  - Abstracts application from underlying JDBC, JTA or CORBA transaction.
  - A Session might span several Transactions.
- ConnectionProvider (org.hibernate.connection.ConnectionProvider)
  - A factory for (and pool of) JDBC connections. Abstracts application from underlying Datasource or DriverManager. Not exposed to application, but can be extended/implemented by the developer.

- TransactionFactory (org.hibernate.TransactionFactory)(Optional)
  - A factory for Transaction instances. Not exposed to the application, but can be extended/implemented by the developer.

What is the JPA?

- Developed under JSR-220
  - Initial goal was to simplify EJB CMP
- JSR-220 segmented into two specifications:
  - EJB 3.0
  - Java Persistence API
    - Complete, standalone ORM solution for both Java EE
- and Java SE environments
- Significant Community Involvement:
  - Leverages best ideas from Hibernate, Toplink, and JDO

Why should I care?

- Why not just use JDBC?
  - Low level API
  - Simple to use, but can be error prone
- Why not just use [INSERT ORM HERE]?
  - Standardized API leveraging best ideas from ORM community
  - Better for developers - one API to learn and use
  - Can choose between competing implementations
  - Vendor independence

Goals

- Provide complete ORM solution for Java SE and Java EE environments
- Easy to use
  - Standard POJOs - no framework interfaces or classes to implement or extend
  - Facilitate test-driven development
- Annotation driven, no XML mappings required.
- Configuration By Exception
  - Sensible defaults

JPA Features

- Simple POJO Persistence
  - No vendor-specific interfaces or classes
- Supports rich domain models
  - No more anemic domain models
  - Multiple inheritance strategies
  - Polymorphic Queries
  - Lazy loading of associations
- Rich Annotation Support
- Pluggable persistence providers

POJO Requirements

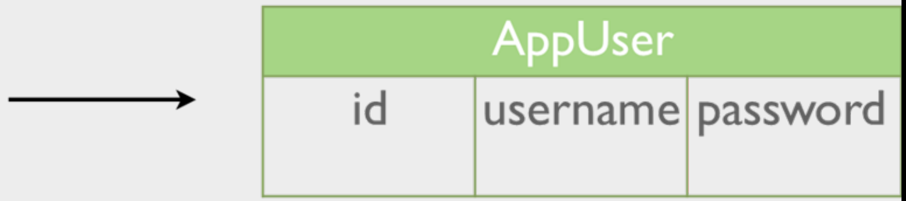
- Annotated with @Entity
- Contains a persistent @Id field
- No argument constructor (public or protected)
- Not marked final
  - Class, method, or persistent field level
- Top level class
  - Can’t be inner class, interface, or enum
- Must implement Serializable to be remotely passed by value as a detached instance

Persistent Entity Example

```
@Entity
public class AppUser {

  @Id
  private Long id;
  private String username;
  private String password;

}
```



JPA Annotations

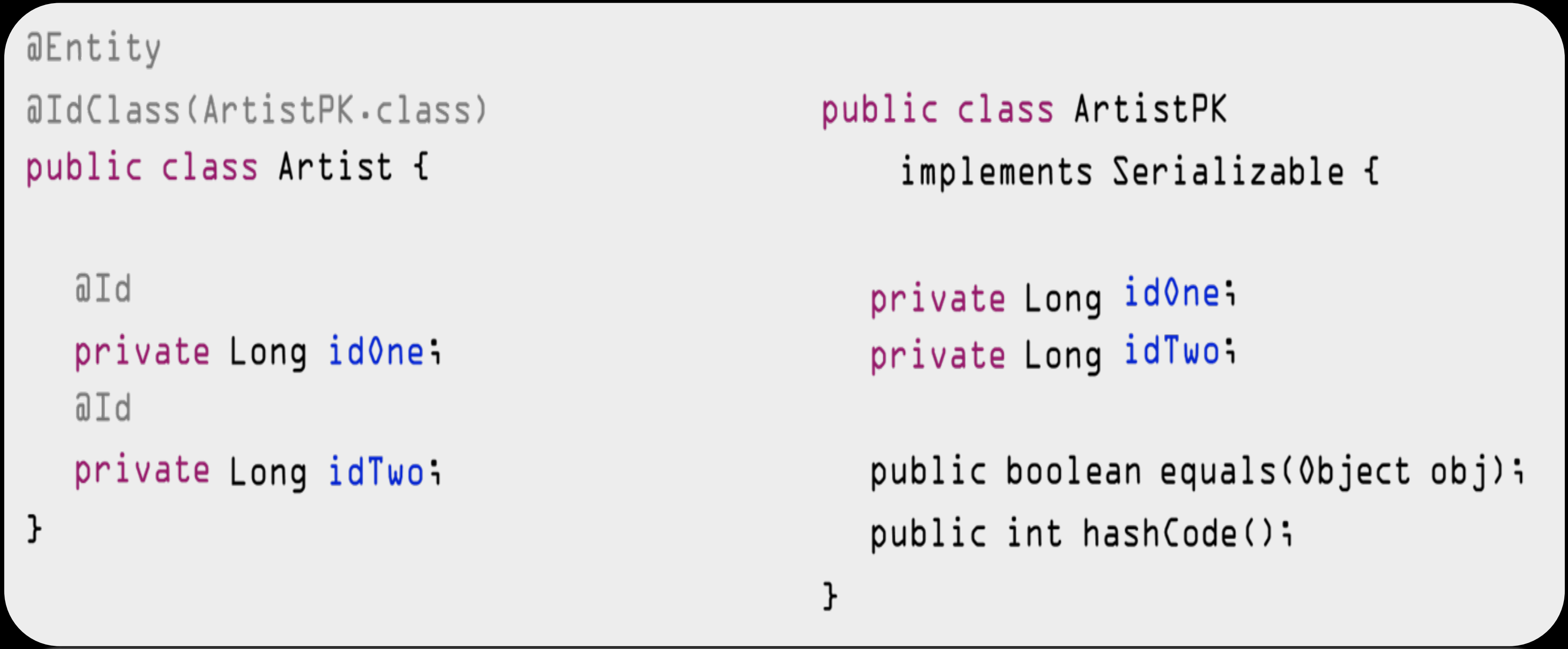
- JPA annotations are defined in the javax.persistence package:
  - http://java.sun.com/javaee/5/docs/api/javax/persistence/packagesummary.html
- Annotations can be placed on fields or properties
  - Field level access is preferred to prevent executing logic
  - Property-level annotations are applied to "getter" method
- Can’t mix style in inheritance hierarchy
  - Must decide on field OR property

Persistent Identifiers

- Entities must define an id field/fields corresponding the the database primary key
- The id can either be simple or composite value
- Strategies:
  - @Id: single valued type - most common
  - @IdClass: map multiple fields to table PK
  - @EmbeddedId map PK class to table PK
- Composite PK classes must:
  - implement Serializable
  - override equals() and hashCode()

@IdClass

- Maps multiple fields of persistent entity to PK class



```java
@Entity
@IdClass(ArtistPK.class)
public class Artist {

  @Id
  private Long idOne;
  @Id
  private Long idTwo;
}
```

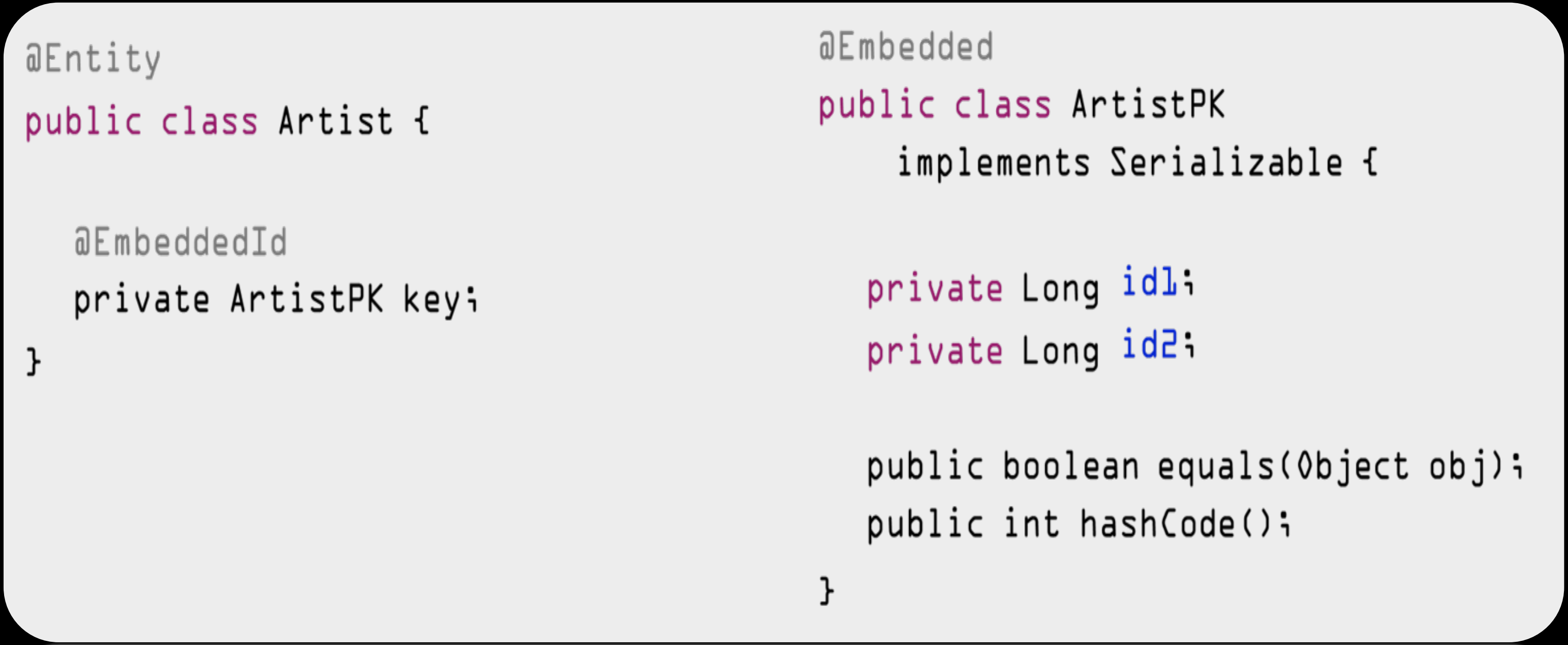
```java
public class ArtistPK
    implements Serializable {

  private Long idOne;
  private Long idTwo;

  public boolean equals(Object obj);
  public int hashCode();
}
```

@EmbeddedId

- Primary key is formal member of persistent entity



```
@Entity
public class Artist {

   @EmbeddedId
   private ArtistPK key;
}
```

```
@Embedded
public class ArtistPK
    implements Serializable {

   private Long id1;
   private Long id2;

   public boolean equals(Object obj);
   public int hashCode();
}
```

@GeneratedValue

- Supports auto-generated primary key values
- Strategies defined by GenerationType enum:
  - GenerationType.AUTO (preferred)
  - GenerationType.IDENTITY
  - GenerationType.SEQUENCE
  - GenerationType.TABLE

@Table and @Column

- Used to define *name* mappings between Java object and database table/columns
- @Table applied at the persistent class level
- @Column applied at the persistent field/property level

Relationships

- JPA supports all standard relationships
  - One-To-One
  - One-To-Many
  - Many-To-One
  - Many-To-Many
- Supports unidirectional and bidirectional relationships
- Supports both composite and aggregate relationships

Cascading Operations

- JPA supports multiple cascade styles
- Defined by the CascadeType enum:
  - CascadeType.PERSIST
  - CascadeType.MERGE
  - CascadeType.REMOVE
  - CascadeType.REFRESH
  - CascadeType.ALL

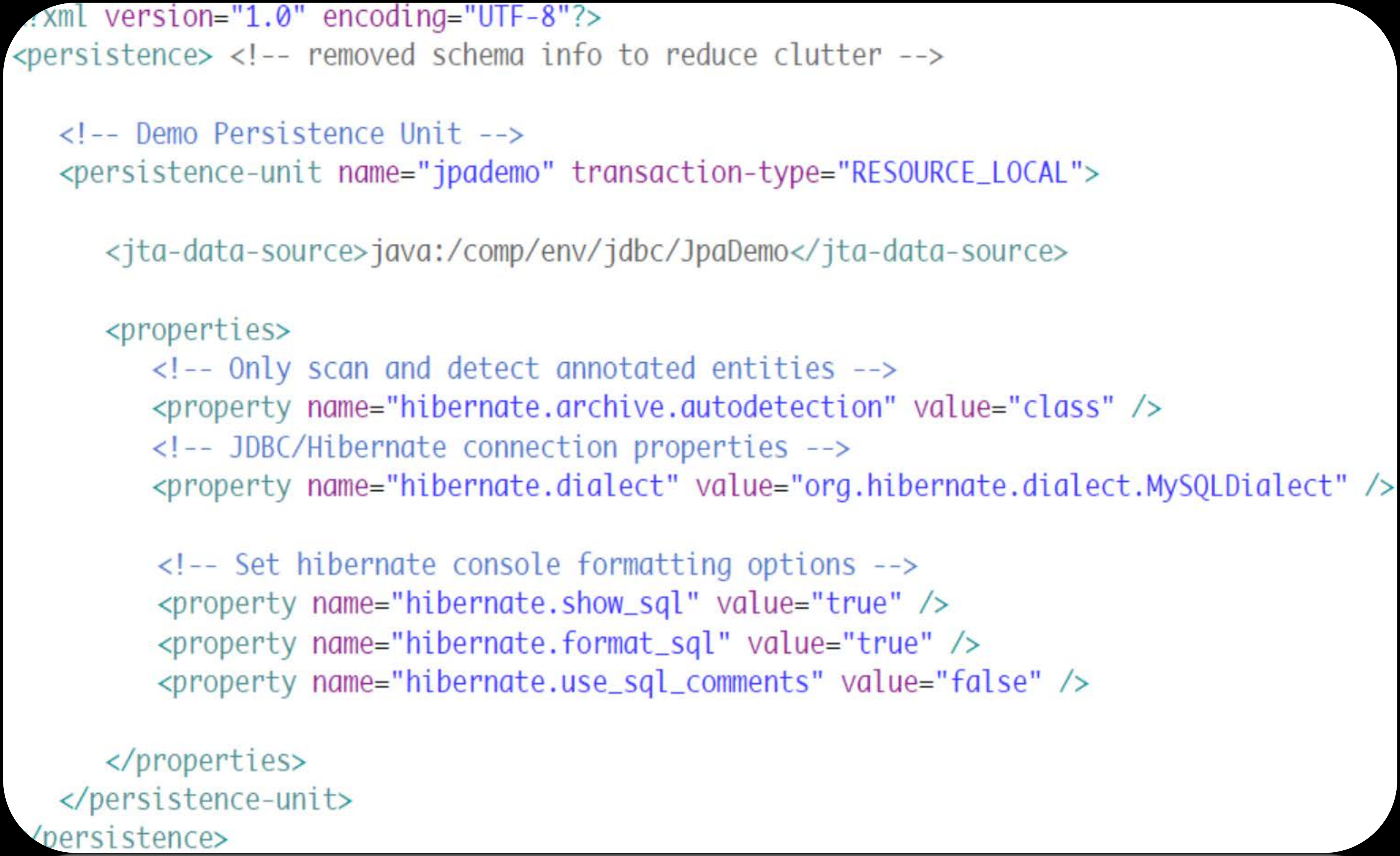
EntityManager

- Provides interface to *Persistence Context*
- Obtained from instance of EntityManagerFactory
  - Manually created in Java SE environment
  - Managed in Java EE or Spring and injects EntityManager instances where needed
- Provides core persistence operations
- Used to obtain Query interface instance
- Provides access to transaction manager for use in Java SE environments

Persistence Context & Unit

- A Persistence Context is a collection of persistent entities managed by the Entity Manager
- Persistence Unit is defined in persistence.xml
  - The only XML required by JPA!
  - Must be defined loaded from META-INF directory
- A persistence-unit defines:
  - The persistence context name
  - Data source settings
  - Vendor specific properties and configurations

Example persistence.xml



```xml
?xml version="1.0" encoding="UTF-8"?>
<persistence> <!-- removed schema info to reduce clutter -->

   <!-- Demo Persistence Unit -->
   <persistence-unit name="jpademo" transaction-type="RESOURCE_LOCAL">

      <jta-data-source>java:/comp/env/jdbc/JpaDemo</jta-data-source>

      <properties>
         <!-- Only scan and detect annotated entities -->
         <property name="hibernate.archive.autodetection" value="class" />
         <!-- JDBC/Hibernate connection properties -->
         <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />

         <!-- Set hibernate console formatting options -->
         <property name="hibernate.show_sql" value="true" />
         <property name="hibernate.format_sql" value="true" />
         <property name="hibernate.use_sql_comments" value="false" />

      </properties>
   </persistence-unit>
/persistence>
```

Query Interface

- Obtained from the EntityManager using:
  - createQuery()
  - createNamedQuery()
  - createNativeQuery()
- Supports bind parameters, both named and ordinal
- Returns query result:
  - getSingleResult()
  - getResultList()
- Pagination Support:
  - setFirstResult()
  - setMaxResults()

JPA Queries

- Supports static & dynamic queries
- Queries can be written using JPQL or SQL
- Named and positional bind parameters
- Supports both static and dynamic queries
  - Static queries are written as annotations of the entity
- Supports eager fetching using the fetch keyword

JPQL Features

- Java Persistence Query Language (JPQL)
  - Extension of EJB QL language
- SQL like syntax
  - Reference objects/properties instead of tables/columns
- Supports common SQL features:
  - Projections
  - Inner & Outer Joins - Eager fetching supported
  - Subqueries
  - Bulk operations (update and delete)