This presentation provides an introduction to JPA and coverage of the methods of the EntityManager.
Purpose

1. Goals
   • Introduce the Java Persistence API and its capabilities

2. Objectives

   At the completion of this topic, the student shall
   • have an understanding of:
     • Requirements for entity classes
     • How to define a persistence unit
     • How to obtain an entity manager
     • Persistence methods available from EntityManager
   • be able to:
     • Define a Java class as an entity
     • Define a persistence unit with entity class(es)
     • Implement DAO operations in terms of javax.persistence.Persistence API
JPA Overview

1.1. Background

- Earlier versions of EJB Spec defined persistence as part of javax.ejb.EntityBean
- JavaEE 5 moved persistence to its own specification
  - Java Persistence API (JPA) version 1.0
  - javax.persistence
  - Ease of use API above JDBC
  - Fueled by the addition of @Annotations to the JavaSE language
  - Provides
    - Object/Relational Mapping (ORM) Engine
    - Query Language (JPA-QL) - SQL-like - main carryover from EJB 2.x
- JavaEE 6 advanced specification
  - Java Persistence API (JPA) version 2.0
  - More mapping capabilities
  - More entity manager capabilities
  - Standardization of properties
  - Criteria API
- JavaEE 7 enhanced specification
  - Java Persistence API (JPA) version 2.1
  - Converters - to convert attributes to/from DB types
  - Criteria API Bulk Updates - no longer tied to just queries
  - Stored Procedure Query support
  - Partial fetching of objects
  - ...
- JavaEE 8
  - Java Persistence API (JPA) version 2.2
  - Streams - to fetch data from database in a stream versus batch
  - Date and Time API Support - eliminates need for converter
  - @Repeatable - no longer have to wrap repeated annotations within single wrapper
  - ...

1.2. EntityManager

- Replaced EJB 2.x Home Functionality
- Handles O/R Mapping to the database
- Provides APIs for
  - Inserting into database
  - Updating entities in database
  - Finding and querying for entities in database
  - Removing entities from database
- Provides caching
- Integrates with JTA transactions when on server-side
Chapter 1. JPA Overview

• Tightly integrated with JavaEE and EJB, but not coupled to it

1.3. Entity

• Plain Old Java Objects (POJOs)
  • Nothing special happens when calling new()

```
Author author = new Author();
```

• From JPA perspective the above is a new/unmanaged entity

• Entity minimum requirements:

```
@javax.persistence.Entity
public class Author {
    @javax.persistence.Id
    private long id;

    private String firstName;
    private String lastName;
    private String subject;
    private Date publishDate;

    public Author() {
    }
    ...
}
```

• Annotated as an Entity or declared in an orm.xml
• Unique identity (form primary key(s))
• Public default constructor
• Persistent when associated with an entity manager/persistence context

```
em.persist(author);
```

1.4. JPA Example

```
Author author = new Author();
author.setFirstName("dr");
author.setLastName("seuss");
author.setSubject("children");
author.setPublishDate(new Date());
logger.debug("creating author: ", author);
assertEquals("unexpected initialized id", 0, author.getId());

logger.debug("em.contains(author)={}", em.contains(author));
assertFalse("author managed", em.contains(author));
em.persist(author);
logger.debug("created author: ", author);
logger.debug("em.contains(author)=[]", em.contains(author));
assertNotEquals("missing id", 0, author.getId());
```
Entity States

1.5. Entity States

1.5.1. Managed

- Associated with persistence context
- Has identity
- Changes to the entity will impact the database
- Method `em.contains(entity)` returns `true`

1.5.2. Detached

- Has identity but not associated with persistence context
- Changes to entity will not impact the database
- Method `em.contains(entity)` returns `false`
- An entity becomes detached when:
  - Has not yet been persisted
  - After a transaction-scoped transaction is committed
  - After a transaction rollback **-- remember this in your error logic**
  - Manually detaching entity from persistence context thru `em.detach()`
  - Manually clearing the persistence context thru `em.clear()`
  - Closing EntityManager
  - Serializing entity thru a remote interface

1.6. Persistence Context

- A set of managed instances managed by an EntityManager
- All entities become detached once closed
- Two types:
  - Extended

```java
Author author = new Author();
...
em.persist(author);
em.getTransaction().begin();
em.getTransaction().commit();

author.setFirstName("foo");
em.getTransaction().begin();
em.getTransaction().commit();
```
Chapter 1. JPA Overview

```java
em.getTransaction().begin();
author.setFirstName("bar");
em.getTransaction().commit();
```

- Live beyond a single transaction
- Allow long-lived algorithms to process without tying up a database transaction
- Transaction Scoped

```java
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public Author createAuthor(...) {
    Author author = new Author();
    ...
    em.persist(author);
    return author;
}
```

- Begin/end at transaction boundaries
- Injected by containers

### 1.7. Persistence Unit

- A set of classes that are mapped to the database
- Defined in META-INF/persistence.xml
- Entity classes may be named in persistence.xml or searched for
- Entity mapping may be provided, augmented, or overridden with orm.xml mapping file

### 1.8. Example Layout

```
|-- ejava
|`-- examples
|   |-- daoex
|       |-- bo
|       | `-- Author.class
|       |-- dao
|       | `-- AuthorDAO.class
|       |-- `-- DAOException.class
|       |    `-- jpa
|       |       `-- JPAAuthorDAO.class
|-- META-INF
  |-- `-- orm.xml
     `-- persistence.xml
```

### 1.9. Persistence.xml

#### 1.9.1. Application Example

```xml
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
```
The above example:
• defines properties for EntityManager to establish physical connections to database
• this form of connection management only occurs for 2-tier thin clients (e.g., unit tests)
• not a good place for potentially changing or sensitive property values

Persistence Unit properties can be supplied at runtime
We can alternatively supply and override all the properties from the META-INF/persistence.xml at runtime if we control the creation of the EntityManagerFactory

1.9.2. Server Example
Chapter 1. JPA Overview

The above example:

- uses a DataSource from JNDI tree to obtain connections to database
- references entity classes in a separate archive through a relative path using jar-file
- supplies references to resources that are fairly stable in value and non-sensitive

1.9.3. Optional hibernate.properties

- Can be used to specify connection properties outside of persistence.xml
- Useful in separating production mapping information from runtime connection properties

```xml
#hibernate-specific alternate source of persistence.xml properties
hibernate.connection.url=jdbc:h2:./target/h2db/ejava
hibernate.connection.driver_class=org.h2.Driver
hibernate.connection.password=
hibernate.connection.username=sa
hibernate.dialect=org.hibernate.dialect.H2Dialect
hibernate.show_sql=true
hibernate.format_sql=true
#hibernate.jdbc.batch_size=0
```

Use the hibernate form of properties in hibernate.properties

Even though one can specify properties within the persistence.xml#persistence-unit or hibernate.properties file, they are not always the same. If you use the hibernate.properties file to specify the url, driver, and credentials -- always use the hibernate form of the name to avoid stumbling on the cases where hibernate fails to look for the javax.persistence form of the name for properties coming from the hibernate.properties file.

1.9.4. Sample orm.xml

- referenced by the persistence.xml
- supplies mapping metadata for the entities that are either not annotated or overriding the annotations
- each entity or group of entities within the same persistence unit can be defined in separate files

```xml
<?xml version="1.0" encoding="UTF-8"?>
```
1.9.5. persistence.xml Elements

name
Identity used to reference persistence unit

provider
Fully qualified name of javax.persistence.PersistenceProvider
  • Not needed if provider is in classpath

mapping-file
Path reference to an orm.xml mapping file

jta-data-source
JNDI path of a JTA javax.sql.DataSource

non-jta-datasource
JNDI path of a RESOURCE_LOCAL javax.sql.DataSource

jarfile
Reference to an archive with entity classes

class
Fully qualified package name of entity class
  • One source of entity information

exclude-unlisted-classes
If set, provider will not scan to discover entity classes

properties
name/value property pairs to express additional configuration info
1.9.6. Entity Discovery

- Classes with @Entity annotation
  - In persistence.xml’s JAR file
  - Contained in any JAR listed in persistence.xml#jar-file element
- Classes mapped
  - With a META-INF/orm.xml file
  - With custom mapping files
- Classes listed in persistence.xml#class element

1.10. Schema Generation

Database schema generation
  - Live database used by EntityManager has been prepared for persistence at runtime
  - Useful for demos and tutorials
  - Not an option for production

Script schema generation
  - Database schema commands written without changing the database
  - Useful in debugging “what does the provider think I am modeling?”
  - Good starting source script to author database migration scripts from

Vendor-specific option
  - Persistence provider is required to provide schema-generation capability
  - Likely has for years through a proprietary interface

javax.persistence option
  - Standard interface added in JPA 2.1 for requesting schema generation

1.10.1. Hibernate DDL Generation Example

hibernate.hbm2ddl.auto
  - Controls hibernate schema generation
    - create, drop, create-drop, update, validate, and none

hibernate.hbm2ddl.import_files
  - Allows for self-authored files to be used as well
  - Expressed thru hibernate.properties using hibernate.hbm2ddl properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>hibernate.hbm2ddl.auto</td>
<td>create-drop</td>
</tr>
<tr>
<td>hibernate.hbm2ddl.import_files</td>
<td>dddl/mydb-tuningdrop.ddl, dddl/mydb-tuning.ddl</td>
</tr>
<tr>
<td>hibernate.connection.url</td>
<td>${jdbc.url}</td>
</tr>
<tr>
<td>hibernate.connection.driver_class</td>
<td>${jdbc.driver}</td>
</tr>
<tr>
<td>hibernate.connection.username</td>
<td>${jdbc.user}</td>
</tr>
</tbody>
</table>

1 [HBM2DDL Docs](https://docs.jboss.org/hibernate/orm/5.0/manual/en-US/html/ch03.html#configuration-optional)
• Can be expressed thru persistence.xml but use standard properties instead to avoid confusion

1.10.2. javax.persistence Schema Generation

javax.persistence.schema-generation.database.action
Values: drop-and-create, drop, create, none

javax.persistence.schema-generation.create-source
Values: metadata (default), script, metadata-then-script, script-then-metadata
javax.persistence.schema-generation.create-script-source - provides reference to create source
javax.persistence.schema-generation.drop-script-source - provides reference to drop source

javax.persistence.sql-load-script-source
Provides reference to script to execute after schema initialized
Useful in Pre-populating sample database tables with content

• Database schema generation example

```xml
<persistence-unit name="jpa-schemagen-test">
  <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
  <properties>
    <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>
    <!-- a database connection definition is required for database actions -->
    <property name="javax.persistence.jdbc.url" value="${jdbc.url}"/>
    <property name="javax.persistence.jdbc.driver" value="${jdbc.driver}"/>
    <property name="javax.persistence.jdbc.user" value="${jdbc.user}"/>
    <property name="javax.persistence.jdbc.password" value="${jdbc.password}"/>
  </properties>
</persistence-unit>
```

• Script schema generation example

```xml
<persistence-unit name="jpa-schemagen-test">
  <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
  <properties>
    <!-- a script file containing create and drop commands will be generated without interacting with database -->
    <property name="javax.persistence.schema-generation.scripts.action" value="drop-and-create"/>
    <property name="javax.persistence.schema-generation.scripts.create-target" value="${project.build.outputDirectory}/ddl/${project.artifactId}-create.ddl"/>
    <property name="javax.persistence.schema-generation.scripts.drop-target" value="${project.build.outputDirectory}/ddl/${project.artifactId}-drop.ddl"/>
    <!-- otherwise we would get 1 line per statement without standard delimiter character -->
    <property name="hibernate.format_sql" value="true"/>
    <property name="hibernate.hbm2ddl.delimiter" value=";"/>
    <!-- required when no database connection specified -->
  </properties>
</persistence-unit>
```

2 javalex.persistence schema generation docs [https://docs.oracle.com/javaee/7/tutorial/persistence-intro005.htm]
1.11. Basic Testing Usage Steps

- Test Setup -- These actions occur outside of the individual @Test methods and normally not within the business and DAO logic. This is the kind of activity the server-side container will primarily take care of.
  - Create single EntityManagerFactory to share across multiple EntityManagers

```java
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class MyJPATest {
    private static EntityManagerFactory emf;
    private EntityManager em;

    @BeforeClass
    public static void setUpClass() {
        emf = Persistence.createEntityManagerFactory("jpaDemo");
    }

    @Before
    public void setUp() throws Exception {
        em = emf.createEntityManager();
    }

    // Other test methods...
}
```

- Create EntityManager before each test
- Runtime -- These actions occur either within the @Test or in the DAO under test. However, business and DAO logic rarely interact with the transaction calls. Transaction control is the job...
Basic Testing Usage Steps

of the container and under the configuration of the server-side EJB. Think of a JUnit TestCase
and @Test as being a simulated replacement for the container and EJB configuration so that
we can test the business and DAO logic.

- Start Transaction - this is the role of the container

```java
em.getTransaction().begin();
```

- Interact with EntityManager - this is the role of the DAO

```java
em.persist(author);
```

- Commit or Rollback Transaction - this is the role of the container

```java
em.getTransaction().commit();
-or-
em.getTransaction().rollback();
```

- Clean up

  - Close EntityManager after each test

```java
@After
public void tearDown() throws Exception {
    if (em != null) {
        em.close();
        em = null;
    }
}
```

  - Close EntityManagerFactory after all tests

```java
@AfterClass
public static void tearDownClass() {
    if (emf != null) {
        emf.close();
        emf = null;
    }
}
```

Check for initialization in @After/@AfterClass before closing

It is always good to try to keep @After or @AfterClass lifecycle methods from
throwing sloppy exceptions (e.g., NullPointerException). These lifecycle methods
get called whether the test is passing, failing, or even failed to fully initialize. If
the @After or @AfterClass lifecycle methods throw an exception during cleanup
after an exception -- their error will mask the real error and add confusion to
the test results. Start each of these methods with a check of whether the sibling
@BeforeClass or @Before lifecycle methods initialized what this method is trying
to shutdown.
Chapter 1. JPA Overview

1.12. Entity Manager Methods

```java
public interface javax.persistence.EntityManager {
  ...
}
```

1.12.1. Basic CRUD Operations

- `void persist(Object entity);`
- `<T> T find(Class<T> entityClass, Object primaryKey);`
- `<T> T find(Class<T> entityClass, Object primaryKey, Map<String, Object> properties);`
- `<T> T merge(T entity);`
- `void remove(Object entity);`
- `<T> T getReference(Class<T> entityClass, Object primaryKey);`

1.12.2. Membership Operations

- `void clear();`
- `void detach(Object entity);`
- `boolean contains(Object entity);`

1.12.3. State Synchronization Operations

- `void flush();`
- `void setFlushMode(javax.persistence.FlushModeType);`
- `javax.persistence.FlushModeType getFlushMode();`
- `void refresh(Object);`
- `void refresh(Object, java.util.Map);`

1.12.4. Locking Operations

- `void lock(Object entity, javax.persistence.LockModeType);`
- `void lock(Object entity, javax.persistence.LockModeType, Map<String, Object> properties);`
- `<T> T find(Class<T> entityClass, Object primaryKey, javax.persistence.LockModeType);`
- `<T> T find(Class<T> entityClass, Object primaryKey, javax.persistence.LockModeType, Map<String, Object> properties);`
- `void refresh(Object, javax.persistence.LockModeType);`
- `void refresh(Object, javax.persistence.LockModeType, Map<String, Object> properties);`
- `javax.persistence.LockModeType getLockMode(Object entity);`
1.12.5. Query Operations

javax.persistence.Query createQuery(String jpaql);
<T> javax.persistence.TypedQuery<T> createQuery(String jpaql, Class<T> resultClass);
javax.persistence.Query createNamedQuery(String name);
<T> javax.persistence.TypedQuery<T> createNamedQuery(String name, Class<T> resultClass);
javax.persistence.Query createNativeQuery(String sql);
javax.persistence.Query createNativeQuery(String sql, Class resultClass);
javax.persistence.Query createNativeQuery(String sql, String resultMapping);
<T> javax.persistence.TypedQuery<T> createQuery(javax.persistence.criteria.CriteriaQuery<T> criteria);
javax.persistence.criteria.CriteriaBuilder getCriteriaBuilder();

1.12.6. Other Operations

void close();
boolean isOpen();
javax.persistence.EntityManager getTransaction();
void joinTransaction();
void setProperty(String key, Object value);
java.util.Map getProperties();
<T> T unwrap(Class<T> clazz);
Object getDelegate();
javax.persistence.metamodel.Metamodel getMetamodel();
javax.persistence.EntityManagerFactory getEntityManagerFactory();
Chapter 2.

Entity Manager CRUD Methods

2.1. persist()

```java
Author author = new Author();
author.setFirstName("dr");
author.setLastName("seuss");
author.setSubject("children");
author.setPublishDate(new Date());

logger.debug("creating author: {}", author);
em.persist(author);
logger.debug("created author: {}", author);
```

- Creating author: ejava.examples.daoex.bo.Author@17e7691, id=0, fn=dr, ln=seuss, subject=children, pdate=Sun Sep 16 10:14:32 EDT 2012, version=0

- Created author: ejava.examples.daoex.bo.Author@17e7691, id=50, fn=dr, ln=seuss, subject=children, pdate=Sun Sep 16 10:14:32 EDT 2012, version=0

- Creating author: ejava.examples.daoex.bo.Author@17e7691, id=0, fn=dr, ln=seuss, subject=children, pdate=Sun Sep 16 10:14:32 EDT 2012, version=0

- Created author: ejava.examples.daoex.bo.Author@17e7691, id=50, fn=dr, ln=seuss, subject=children, pdate=Sun Sep 16 10:14:32 EDT 2012, version=0

- Created author: ejava.examples.daoex.bo.Author@17e7691, id=50, fn=dr, ln=seuss, subject=children, pdate=Sun Sep 16 10:14:32 EDT 2012, version=0

- Created author: ejava.examples.daoex.bo.Author@17e7691, id=50, fn=dr, ln=seuss, subject=children, pdate=Sun Sep 16 10:14:32 EDT 2012, version=0

- Created author: ejava.examples.daoex.bo.Author@17e7691, id=50, fn=dr, ln=seuss, subject=children, pdate=Sun Sep 16 10:14:32 EDT 2012, version=0

- Created author: ejava.examples.daoex.bo.Author@17e7691, id=50, fn=dr, ln=seuss, subject=children, pdate=Sun Sep 16 10:14:32 EDT 2012, version=0

• Inserts entity into database

• Actual insert time depends on transaction active and FlushMode
  • Extended Persistence Context - queues insert until transaction active
  • Transaction-Scoped Persistence Context - always has transaction active
  • Flush occurs automatically prior to or during commit()
  • Flush can be forced with manual `em.flush()` call

• New or removed entity enters managed state
  • All further changes are watched and will update database

```java
logger.debug("em.contains(author)={}", em.contains(author));
em.persist(author);
logger.debug("created author: {}", author);
logger.debug("em.contains(author)={}", em.contains(author));
```

- em.contains(author)=false
  - Created author:
    - em.contains(author)=true

• Existing entity is ignored
  • Cascades will still occur for CascadeType.PERSIST and ALL relationships

```java
em.persist(author);
... 
em.persist(author);
```

15
Calling persist applies cascades to new relationships

It is quite legal to call persist() with an already managed entity. This is how one can get cascade actions to be applied to new relationships formed after the entity is originally managed.

• Detached entities will be rejected
• Entities with an identity but not associated with a persistence context

```java
Author author = new Author(1);
author.setFirstName("dr");
...
logger.debug("creating author: {}", author);
logger.debug("em.contains(author)={}", em.contains(author));
try {
    em.persist(author);
    fail("did not detect detached entity");
} catch (PersistenceException ex) {
    logger.debug("caught expected exception:" + ex.getLocalizedMessage(), ex);
}
logger.debug("em.contains(author)={}", em.contains(author));
```

-creating author:ejava.examples.daoex.bo.Author@ad339b, id=1, ...
-caught expected exception:javax.persistence.PersistanceException: org.hibernate.PersistentObjectException: detached entity passed to persist: ejava.examples.daoex.bo.Author
-em.contains(author)=false

2.2. find()

```java
<T> T find(Class<T> entityClass, Object primaryKey);
<T> T find(Class<T> entityClass, Object primaryKey, Map<String, Object> properties);
```

• Searches for the entity by primary key value
• Returns managed entity if found, else null
• Properties permitted for vendor-specific values and ignored if not understood

2.3. getReference()

```java
<T> T getReference(Class<T> entityClass, Object primaryKey);
```

• Searches for the entity by primary key
• Returns a LAZY load reference if found
2.4. merge()

merge()

- Throws EntityNotFoundException if not exist

```
//create initial author
Author author = new Author();
...
em.persist(author);

//create detached author with changes
Author author2 = new Author(author.getId());
author2.setFirstName("updated " + author.getFirstName());
...
//merge changes
Author tmp = em.merge(author2);
em.getTransaction().begin();
em.getTransaction().commit();
...
//verify results
assertFalse("author2 is managed", em.contains(author2));
assertTrue("tmp Author is not managed", em.contains(tmp));
assertSame("merged result not existing managed", author, tmp);
...
//verify changes were made to the DB
Author author3 = em.find(Author.class, author.getId());
assertEquals("updated " + firstName, author3.getFirstName());
```

- Update database with state of detached object
- State of detached object copied into managed entity or new instance created
- Merging a managed entity is ignored. CascadeType.MERGE and ALL relationships are propagated
- Merging a removed entity instance results in an IllegalArgumentException
- Includes automatic checking of @Version property
- Could be replaced with manual merge

```
public Author update(Author author) {
    Author dbAuthor = em.find(Author.class, author.getId());
    dbAuthor.setFirstName(author.getFirstName());
    dbAuthor.setLastName(author.getLastName());
    dbAuthor.setSubject(author.getSubject());
    dbAuthor.setPublishDate(author.getPublishDate());
    return dbAuthor;
}
```

Note

The above manual merge will successfully merge a transient entity -- but what errors and conditions does it not account for?
2.5. remove()

void remove(Object entity);

- Managed entity is removed from database
- Actual delete time depends on transaction active and FlushMode
  - Extended Persistence Context - queues delete until transaction active
  - New entities are ignored but CascadeType.REMOVE and ALL relationships are cascaded

logger.debug("em.contains(author)={}", em.contains(author));
em.remove(author);
logger.debug("em.contains(author)={}", em.contains(author));

-em.contains(author)=false
-HHH000114: Handling transient entity in delete processing
-em.contains(author)=false

- Detached entities are rejected

Author author = new Author(1);
...
try {
  em.remove(author);
  fail("did not reject removal of detached object");
} catch (IllegalArgumentException ex) {
  logger.debug("caught expected exception:" + ex);
}

-caught expected exception:java.lang.IllegalArgumentException:
  Removing a detached instance ejava.examples.daoex.bo.Author#1

- Removed entities are ignored

Author author = new Author();
...
logger.debug("persisted: {}", author);

logger.debug("em.contains(author)={}", em.contains(author));
em.remove(author);
logger.debug("em.contains(author)={}", em.contains(author));
//entity managers will ignore the removal of a removed entity
em.remove(author);
logger.debug("em.contains(author)={}", em.contains(author));
- persisted: ejava.examples.daoex.bo.Author@6c5356, id=50, ...
- em.contains(author)=true
- em.contains(author)=false
- em.contains(author)=false
Chapter 3.

Entity Manager Membership Methods

3.1. contains()

```java
boolean contains(Object entity);
```

- Returns true if object is managed in the persistence context

3.2. clear()

```java
void clear();
```

- Clears all entities and queued changes from the persistence context
- All entities become detached

3.3. detach()

```java
em.persist(author);

// callers can detach entity from persistence context
logger.debug("em.contains(author)=[{}", em.contains(author));
logger.debug("detaching author");
em.getTransaction().begin();
em.flush();
em.detach(author);
logger.debug("em.contains(author)=[{}", em.contains(author));
em.getTransaction().commit();

// changes to detached entities do not change database
author.setFirstName("foo");
em.getTransaction().begin();
em.getTransaction().commit();
Author author2 = em.find(Author.class, author.getId());
logger.debug("author2.firstName=[{}", author2.getFirstName());
logger.debug("author22.firstName=[{}", author2.getFirstName());
assertNotEquals("unexpected name change", author2.getFirstName(), author22.getFirstName());
```

- `em.contains(author)=true`
- `detaching author`
- `em.contains(author)=false`
- `author.firstName=foo`
- `author2.firstName=dr`

- Detaches existing entity from persistence context
Chapter 3. Entity Manager Mem...

- Detach cascaded to CascadeType.DETACH and ALL relationships
- Subsequent changes to entity will not change database
- Portable use requires call to flush() prior to detach()
- New entities are ignored

```java
Author author = new Author();

logger.debug("em.contains(author)={}, em.contains(author)");
logger.debug("detaching author");
em.detach(author);
logger.debug("em.contains(author)={}, em.contains(author)");
```

- em.contains(author)=false
- detaching author
- em.contains(author)=false

- Detached entities are ignored

```java
Author author = new Author();
...
em.persist(author);
em.getTransaction().begin();
em.getTransaction().commit();

//detaching detached entity will be ignored
Author detached = new Author(author.getId());
logger.debug("em.contains(author)={}, em.contains(detached)");
logger.debug("detaching detached author");
em.detach(detached);
logger.debug("em.contains(author)={}, em.contains(detached)");
```

- em.contains(author)=false
- detaching detached author
- em.contains(author)=false
Chapter 4.

Entity Manager State
Synchronization Methods

4.1. flush()

```java
void flush();
```

- Synchronizes cached changes with underlying database
- Requires active transaction

4.2. FlushMode

```java
void setFlushMode(javax.persistence.FlushModeType);
javax.persistence.FlushModeType getFlushMode();
```

- AUTO (default) - unspecified
- COMMIT - flush only happens during commit

4.3. refresh()

```java
em.persist(author);
em.getTransaction().begin();
em.getTransaction().commit();

//change DB state out-of-band from the cache
em.getTransaction().begin();
String newName="foo";
int count=em.createQuery(
    "update jpaAuthor a set a.firstName=:name where a.id=:id"
).setParameter("id", author.getId())
.setParameter("name", newName)
.executeUpdate();
em.getTransaction().commit();
assertEquals("unexpected count", 1, count);

//object state becomes stale when DB changed out-of-band
logger.debug("author.firstName={}", author.getFirstName());
assertNotEquals("unexpected name", newName, author.getFirstName());

//get the cached object back in sync
logger.debug("calling refresh");
em.refresh(author);
```
logger.debug("author.firstName=" + author.getFirstName());
assertEquals("unexpected name", newName, author.getFirstName());

- author.firstName=dr
- calling refresh
- author.firstName=foo

- Updates/overwrites cached entity state with database state
- Refresh of new entity results in java.lang.IllegalArgumentException

Author author = new Author();
author.setFirstName("test");
author.setLastName("new");

try {
    em.refresh(author);
    fail("refresh of new entity not detected");
} catch (IllegalArgumentException ex) {
    logger.debug("caught expected exception:" + ex);
}

-caught expected exception:java.lang.IllegalArgumentException: Entity not managed

- Refresh of detached entity results in java.lang.IllegalArgumentException

em.persist(author);
em.getTransaction().begin();
em.getTransaction().commit();

//refreshing a detached entity will get rejected
Author detached = new Author(author.getId());
em.refresh(author);
logger.debug("refreshed managed entity");
try {
    em.refresh(detached);
    fail("refresh of detached entity not detected");
} catch (IllegalArgumentException ex) {
    logger.debug("caught expected exception:" + ex);
}

-refreshed managed entity
-caught expected exception:java.lang.IllegalArgumentException: Entity not managed
Entity Manager Locking Methods

5.1. Primary Lock Types

- Optimistic
  - Entity assumed not to have concurrent access
  - No active database locks are obtained at start
  - Success judged based on entity state at end
  - State tracked in entity @Version field
- Pessimistic
  - Entity requires mitigation for concurrent access
  - Active database locks are obtained at start of transaction

5.2. LockModeType

- NONE
  - No locks
- OPTIMISTIC (was READ)
  - Requires entity to have a @Version property
  - Prevent dirty reads
  - Prevent non-repeatable reads
- OPTIMISTIC_FORCE_INCREMENT (was WRITE)
  - Requires entity to have a @Version property
  - Update only occurs if change has proper version
  - Version is incremented upon update
  - Incorrect version results in OptimisticLockException
- PESSIMISTIC_READ
  - Supported with or without @Version property
  - Obtains active database lock
  - Provides repeatable reads
  - Does not block other reads
- PESSIMISTIC_WRITE
  - Supported with or without @Version property
  - Forces serialization of entity updates among transactions
- PESSIMISTIC_FORCE_INCREMENT
  - Requires entity to have a @Version property
  - PESSIMISTIC_WRITE lock with increment of @Version

5.3. lock()

```java
void lock(Object entity, javax.persistence.LockModeType);
void lock(Object entity, javax.persistence.LockModeType, Map<String, Object> properties);
```
Chapter 5. Entity Manager Locking

• Requests lock on entity

5.4. find(lock)

\[
<T> \text{find(Class}\langle T \rangle \text{ entityClass, Object primaryKey, javax.persistence.LockModeType);}
\]

\[
<T> \text{find(Class}\langle T \rangle \text{ entityClass, Object primaryKey, javax.persistence.LockModeType, Map<String, Object> properties);}
\]

• Find object by primary key and lock

5.5. refresh(lock)

\[
\text{void refresh(Object, javax.persistence.LockModeType);}
\]

\[
\text{void refresh(Object, javax.persistence.LockModeType, Map<String, Object> properties);}
\]

• Refresh entity state and obtain lock

5.6. getLockMode()

\[
\text{javax.persistence.LockModeType getLockMode(Object entity);}
\]

• Get lock mode for entity
Chapter 6.

Entity Manager Query Methods

6.1. JPA Queries

- Create query based on JPA Query Language (JPAQL)

```
javax.persistence.Query createQuery(String jpaql);
<T> javax.persistence.TypedQuery<T> createQuery(String jpaql, Class<T> resultClass);
javax.persistence.Query createNamedQuery(String name);
<T> javax.persistence.TypedQuery<T> createNamedQuery(String name, Class<T> resultClass);
```

6.2. Native Queries

- Create query based on SQL

```
javax.persistence.Query createNativeQuery(String sql);
javax.persistence.Query createNativeQuery(String sql, Class resultClass);
javax.persistence.Query createNativeQuery(String sql, String resultMapping);
```

6.3. Criteria Queries

- Create query based on typed criteria API

```
<T> javax.persistence.TypedQuery<T> createQuery(javax.persistence.criteria.CriteriaQuery<T> criteria);
javax.persistence.criteria.CriteriaBuilder getCriteriaBuilder();
```
Other Entity Manager Methods

7.1. isOpen()

boolean isOpen();

- Returns true if persistence context is open

7.2. close()

void close();

- Closes persistence context
- All contained entities become detached

7.3. getTransaction()

javax.persistence.EntityManager getTransaction();

- Returns transaction object for inspection and control
- Invalid call for transaction-scoped persistence contexts

7.4. joinTransaction()

void joinTransaction();

- Associate a persistence context with a JTA transaction

7.5. unwrap()

<T> T unwrap(Class<T> clazz);

- Return object of specified type
- Provides access to underlying implementation classes
Chapter 7. Other Entity Management

7.6. getDelegate()

Object getDelegate();

• Returns vendor-specific object backing EntityManager

7.7. getMetaModel()

javax.persistence.metamodel.Metamodel getMetamodel();

• Returns metamodel object for persistence context

7.8. getEntityManagerFactory()

javax.persistence.EntityManagerFactory getEntityManagerFactory();

• Returns EntityManagerFactory associated with EntityManager

7.9. setProperties()/getProperties()

void setProperty(String key, Object value);
java.util.Map getProperties();

• Access EntityManager properties
**JPA Maven Environment**

8.1. JPA Maven Dependencies

8.1.1. JPA API classes

```xml
<dependency>
  <groupId>javax.persistence</groupId>
  <artifactId>javax.persistence-api</artifactId>
  <version>2.2</version>
  <scope>provided</scope>
</dependency>
```

8.1.2. JPA Provider classes

```xml
<dependency>
</dependency>
```
Chapter 8. JPA Maven Environment

8.1.3. Database

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <version>1.4.197</version>
  <scope>test</scope>
</dependency>
```

8.2. Supplying Runtime Properties

8.2.1. Turn on Resource Filtering in pom.xml

```
<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
    </resource>
    <resources>
    <testResources>
      <testResource>
        <directory>src/test/resources</directory>
        <filtering>true</filtering>
      </testResource>
      <testResources>
    ... 
  </build>
```

Above will depend on whether you use src/main or src/test for resource file(s)

8.2.2. Use \${variable} References in Resource Files

```
<properties>
  <!-- standard properties -->
  <property name="javax.persistence.jdbc.url" value="\${jdbc.url}"/>
  <property name="javax.persistence.jdbc.driver" value="\${jdbc.driver}"/>
  <property name="javax.persistence.jdbc.user" value="\${jdbc.user}"/>
  <property name="javax.persistence.jdbc.password" value="\${jdbc.password}"/>

  <!-- hibernate-specific properties -->
  <property name="hibernate.dialect" value="\${hibernate.dialect}"/>
  ... 
</properties>
```
8.2.3. Define Property Values in Parent pom.xml

```xml
<properties>
  <jdbc.driver>org.h2.Driver</jdbc.driver>
  <jdbc.url>jdbc:h2:${basedir}/target/h2db/ejava</jdbc.url>
  <jdbc.user>sa</jdbc.user>
  <jdbc.password/>
  <hibernate.dialect>org.hibernate.dialect.H2Dialect</hibernate.dialect>
</properties>
```

8.2.4. Run with Filtered Values

```xml
<properties>
 <!-- standard properties -->
  <property name="javax.persistence.jdbc.url" value="jdbc:h2:/home/jcstaff/proj/ejava-javaee/git/pjDAO/target/h2db/ejava"/>
  <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
  <property name="javax.persistence.jdbc.user" value="sa"/>
  <property name="javax.persistence.jdbc.password" value=""/>

  <!-- hibernate-specific properties -->
  <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>

  ...
</properties>
```