Berner
Fachhochschule

CAS Java Microservice Development

Java Persistence API - Entity Manager

Simon Martinelli, simon.martinelli@bfh.ch, v2025.10

Content

1. Entity Manager
2. Cascading
3. Transaction Handling

Persistence Context

- The Persistence Context defines the environment of the `EntityManager` and contains
  - set of all managed entities in the application
  - the current transaction
  - the context type

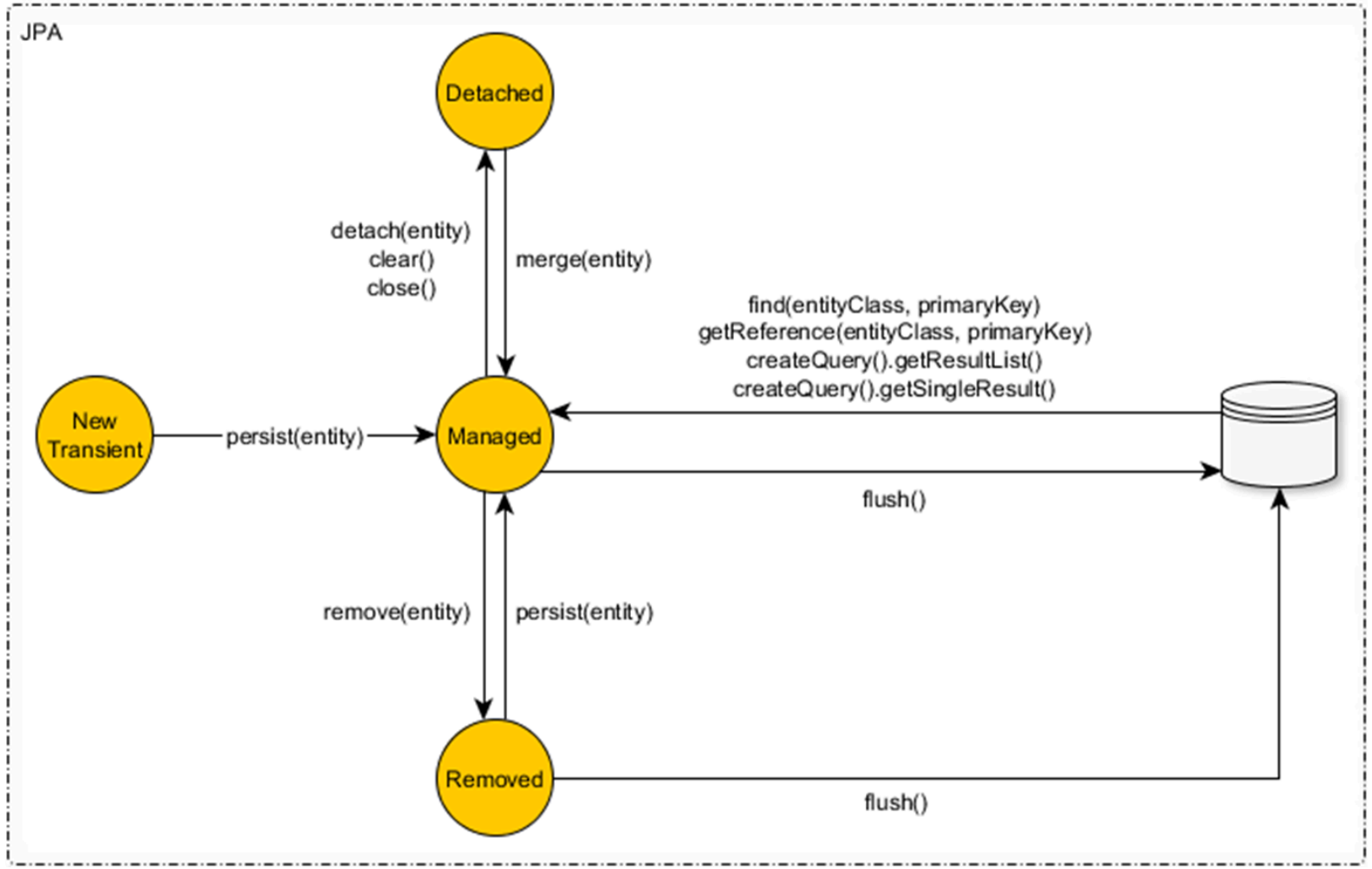
Entity Mangement

- The transfer of entities to and from the database is automatic: as late as possible
  -> Lazy Access
- The transfer of entities to and from the database can be forced manually
  -> synchronous to the call
- Of course, a transaction model applies
  Access to entities takes place at the beginning of the transaction, the synchronization with the database is completed no later than the Commit and is subject to the ACID rule
- Entities can also be accessed outside of transactions but without consistency and synchronization guarantee

Entity States

- **New**
  Entity is newly created, has no connection with the database and no valid ID
- **Managed**
  The entity exists in the database. Changes are tracked automatically by the Entity Manager and propagated to the DB
- **Detached**
  The entity exists in the database but has been decoupled. The state is not synchronized with the database
- **Removed**
  The object still exists but is marked for deletion

States and Transitions



Persist

- Calling `persist()`, a new entity becomes managed by the EntityManager

```
Department dept = em.find(Department.class, deptId);

Employee emp = new Employee();
emp.setId(empId);
emp.setName(empName);

emp.setDepartment(dept);
dept.getEmployees().add(emp);

em.persist(emp);

// The method contains() can be used to check whether an entity is managed
if (em.contains(emp)) {
}
```

Cascade Persist

- Cascaded persistence means: all the objects reachable from a single persistent object are also considered persistent

```
Employee employee = new Employee();
em.persist(emp);
Address address = new Address();
employee.setAddress(address);
```

- Cascading must be declared:

`PERSIST`, `MERGE`, `REMOVE`, `REFRESH`, `DETACH` or `ALL`

Example

```
public class Employee {

  @OneToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
  private Address address;
}
```

Orphan Removal

- Dependant child elements are to be deleted in to-many relationships
- Removing an order from the set will delete the order in the database

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "customer", orphanRemoval = true)
private Set<Order> orders;
```

Finding Entities

- With `find()`, an entity can be found through its primary key
- The found Entity automatically comes into the managed state
- Since `find()` searches through the primary key, this method can be optimized by the persistence provider and the database access may be ommited
- If references of existing entities should be used as relationships in an existing entity, `getReference()` may be used to prevent the full loading of the referenced entity

Read

- The object state is read at the time of first access to the object
- If `FetchType.EAGER` is set, referenced objects are also loaded as well
- If `FetchType.LAZY` is set, referenced objects are read on first use
- The object state is never automatically refreshed but rather only via the `EntityManager.refresh()` method
- A new transaction does NOT automatically lead to the re-reading of existing objects

Spring Transaction Handling

- Transactions are set to `@Transactional(readOnly = true)` on class level
- Writer transactional are set on all modifying methods:
  - `save*`
  - `delete*`
  - `flush()`
- This has a different semantic than JPA because JPA doesn't know a save method
- Also the implementation of `save()` is controversial: https://vladmihalcea.com/best-spring-data-jparepository/

Entity State after Commit

- Object is in detached state after commit
- Changes must be passed to the entity manager with `EntityManager.merge()` at the time of the next transaction

Entity State after Rollback

- After a rollback, the entities are the detached state
- The state of the entities is not changed by the rollback, but the state in the database is changed
- **Caution**: Possible inconsistencies! Entities should be reloaded from the database

Clean up Persistence Context

- In some situations, the Persistence Context must be cleared
- This can be achieved with the `clear()` method of the `EntityManager`
- All entities change into the detached state
- **Caution!** If the Persistence Context contains changes that are not saved with `commit()`, these will be lost

Exercise: Entity Manager

1. Define the cascading with the relationships in the model
2. Remove the unneeded `save()` calls
3. Define the loading behavior (`EAGER` or `LAZY`) in your relationships
4. Have a look at the Lazy Loading behavior in the debugger
5. Repeat the tests and check the generated SQL statements.
   How has the fetching process changed?