Java Persistence API

Agenda

- The Object/Relational paradigm mismatch
- Lifecycle of an entity
- Developing simple entities
  - Persistent identity (primary key)
  - Persistence Context
  - EntityManager API
  - Entity life cycle callbacks
  - Basic relational mapping

The Object/Relational paradigm mismatch

- Granularity
- Subtypes
- Identity
- Associations
- Object Graph Navigation

The DAO pattern

Java application

Data Access Object

JDBC

The DAO pattern

Java application

- Contains SQL as Strings
  - Hard to maintain
  - Database specific
  - A lot of "dumb" code

Java application

Data Access Object

JPA

JDBC

Java Persistence API

- Map Java classes to tables using annotations
  - Including relations and inheritance
- Use a OO query language
  - abstraction on database specific sql
- An API to persist, update, delete and get Entities

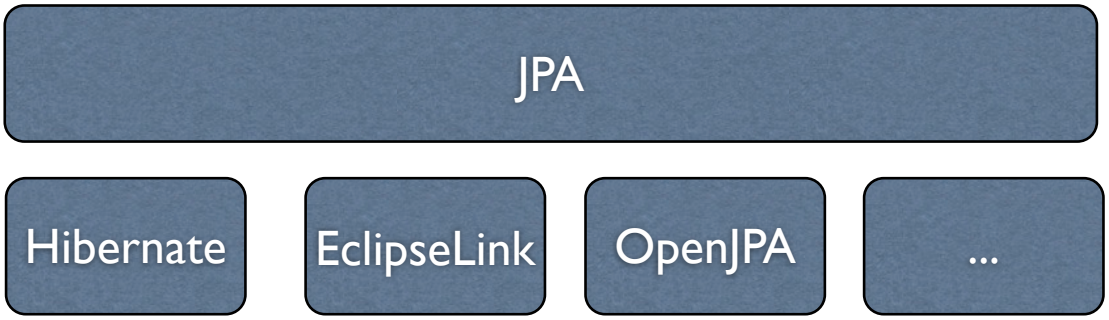
Java Persistence API

- Standardize ORM into single Java Persistence API
  - Usable in both Java SE and Java EE
  - Based on best practices from EJB 2.x, Hibernate, JDO, TopLink, etc.
- Support for pluggable, third-party persistence providers

Entities, reborn!

- Persistent objects
  - Entities, not Entity Beans
  - Java objects, not 'components'
  - Concrete classes
    - Support use of new keyword
  - Indicated by @Entity annotation

JPA Implementations



- Use standardized API for most tasks
- Use provider specific API for advanced features

Basic mapping

- @Entity to make a class an Entity
- @Id to configure Primary Key
  - required for each Entity
- @GeneratedValue to let the database generate keys
- By default each field is persisted

```
@Entity
public class Contact {
    @Id
    @GeneratedValue
    private long id;

    private String firstname;
    private Date birthDate;

    //Getters and setters
```

Persistent Identity

- Define generator strategy type
  - AUTO, IDENTITY, TABLE, SEQUENCE
  - Depends on the underlying database

```
@Entity
public class Contact {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
```

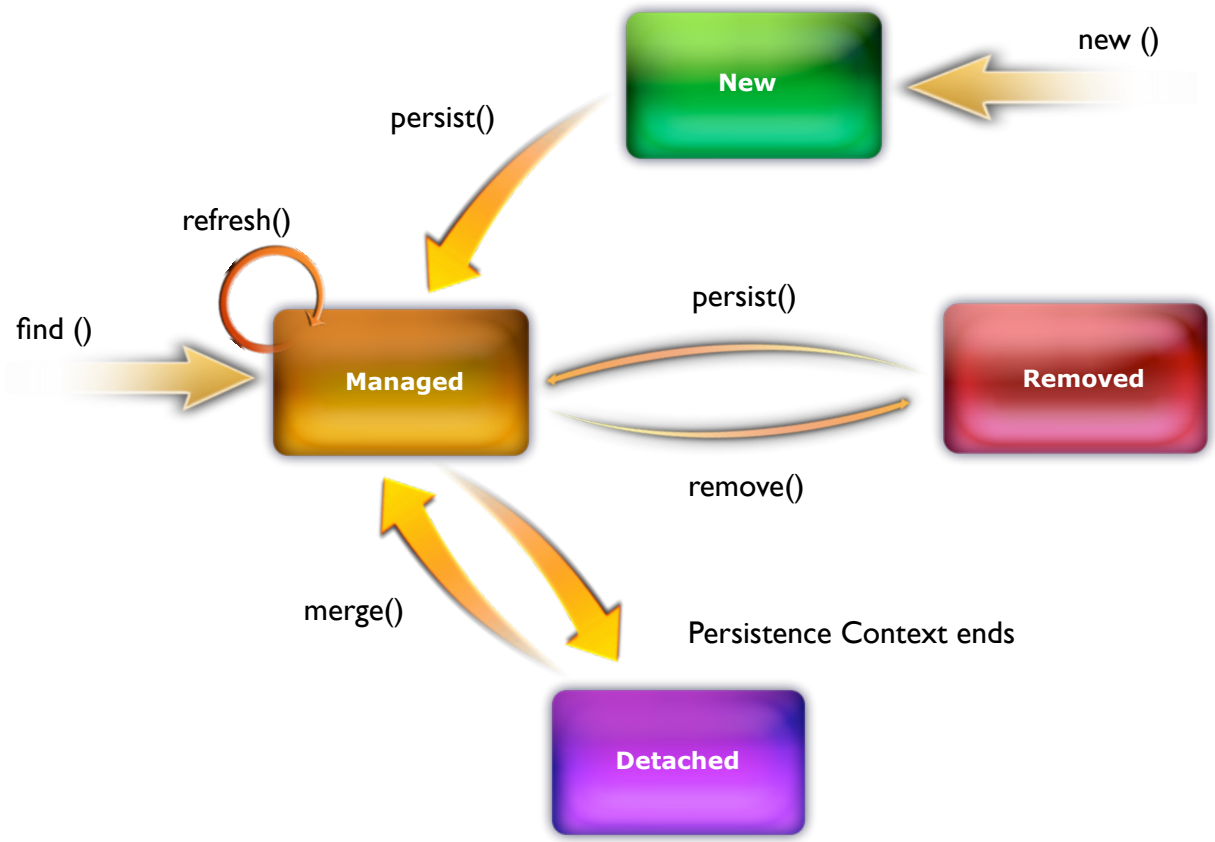
Synchronizing entities with the database

- EntityManager
  - API for object/relational mapping (ORM)
  - Inject with @PersistenceContext
- Persistence Context
  - Set of "managed" entities (at runtime)

Example: Persist

- Insert a new instance of the entity into the database
- The entity instance becomes managed in the Persistence Context

```
@PersistenceContext
EntityManager em;

public void saveContact(Contact contact) {
    em.persist(contact);
}
```



Example: Merge

- State of detached entity gets merged into a managed copy of the entity
  - merge() returns managed entity with different Java identity than detached entity

```
@PersistenceContext
EntityManager em;

public Contact saveUpdatedContact(Contact contact) {
    return em.merge(contact);
}
```

Example: Find

- Find on primary key
- EntityManager returns a managed entity
- Returns null when key does not exists

```
public Contact changeContactName(long id, String newName) {
    Contact c = em.find(Contact.class,  id);
    c.setName(newName);
    return c;
}
```

Removing entities

- Removal via EntityManager API
- Entities must be managed to be removed
  - E.g. do a find() first
- Use EJB-QL for batch deletes

```
public void removeContact(long id) {
    Contact c = em.find(Contact.class,  id);
    em.remove(c);
}
```

Basic relational mapping

- JPA provides enough flexibility to start from either direction:
  - Database > Entities
  - Entities > Database
- Elementary schema mappings:
  - Table and column mappings:
    - @Table
    - @Column

Mapping example

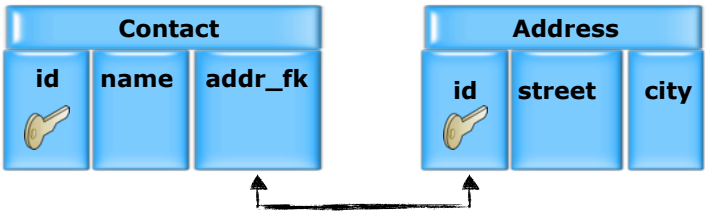
```
@Entity
@Table(name = "CONTACTS")
public class Contact {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(name = "C_NAME", length = 50, nullable = false)
    private String name;

    @Column(unique = true)
    private String email;

    @Temporal(value = TemporalType.TIMESTAMP)
    private Date birthDate;
```

Relationships

- Common relationships supported:
  - @ManyToOne, @OneToOne, @OneToMany, @ManyToMany
  - Unidirectional or bidirectional
- Owning side of relationship can specify physical mapping
  - @JoinColumn
  - @JoinTable

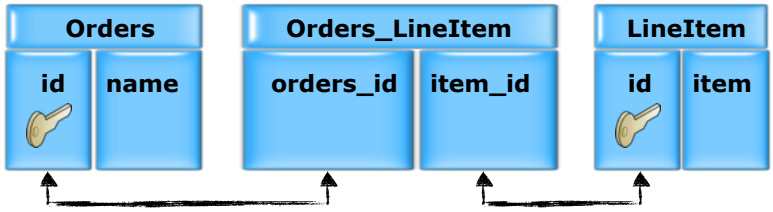
One-to-One

```
@Entity
public class Contact {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @OneToOne
    Address address;
```




```
@Entity
public class Address {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    long id;

    @OneToOne(mappedBy = "address")
    Contact contact;
```

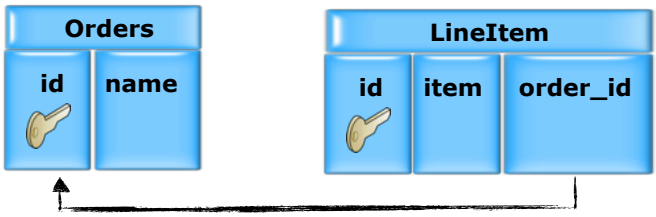
One-to-Many unidirectional




```java
@Entity
@Table(name = "orders")
public class Order {
    @Id @GeneratedValue
    private long id;

    private Date orderDate;

    @OneToMany
    private List<LineItem> items;

    //Getters and Setters
```

```java
@Entity
public class LineItem {
    @Id
    @GeneratedValue
    private long id;

    private String item;

    //Getters and Setters
```

One-to-Many bidirectional




```java
@Entity
@Table(name = "orders")
public class Order {
    @Id @GeneratedValue
    private long id;

    private Date orderDate;

    @OneToMany(mappedBy = "order")
    private List<LineItem> items;
```

passive

```java
@Entity
public class LineItem {
    @Id
    @GeneratedValue
    private long id;

    private String item;

    @ManyToOne
    private Order order;
```

Lazy loading

- Multi-value associations are by default loaded lazily
  - prevents loading the whole database...
- Collections are proxied and will be loaded when used
- An entity must be managed to load associations!
  - LazyInitializationException

Bidirectional mappings

- One side is the "owning" side
- The other side is "passive"
  - the passive side does not synchronize changes

works

```
Order order = new Order();
em.persist(order);

LineItem item = new LineItem();
item.setOrder(order);
em.persist(item);
```

Doesn't work; Relation is not set!

```
LineItem item = new LineItem();
em.persist(item);

Order order = new Order();
order.getItems().add(item);
em.persist(order);
```

Fixing the passive side

- Common trick to work with passive relations

```
@OneToMany(mappedBy = "order")
private List<LineItem> items;

public void addItem(LineItem item) {
    items.add(item);
    item.setOrder(this);
}
```

Set the owning side

```
LineItem item = new LineItem();
em.persist(item);

Order order = new Order();
order.addItem(item);
em.persist(order);
```

Cascading

```
Order order = new Order();
LineItem item = new LineItem();
item.setOrder(order);
em.persist(item);
```

TransientObjectException

The referenced order is not persisted yet

Cascading operations

- Cascading can be set on all relationship annotations
  - Default: no cascading
  - Values: PERSIST, MERGE, REMOVE, REFRESH, ALL

```
@ManyToOne(cascade = CascadeType.PERSIST)
private Order order;
```

```
@ManyToOne(cascade =
        {CascadeType.PERSIST, CascadeType.MERGE})
private Order order;
```

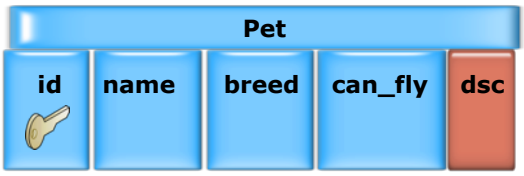
- Decide on entity-by-entity the most appropriate cascading setting

Inheritance

- Entities can extend:
  - Other entities
  - Other plain Java classes
- Mapping inheritance hierarchy to:
  - Single table: everything in one table
    - Requires discriminator value
  - Joined: each class in a separate table
  - Table per concrete class

Inheritance mapping strategies

Single Table



Joined Tables

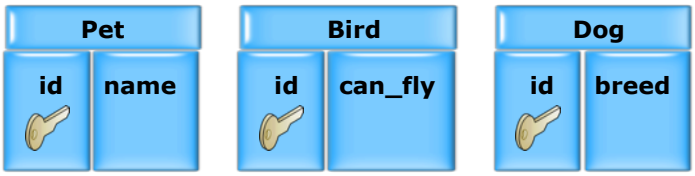


Table per concrete class

Single Table

Employee				
id	**name**	**platform**	**commision**	**DTYPE**
1	Paul	Java	*null*	Programmer
2	Jaap	*null*	50	Sales

```
@Entity
@Inheritance(strategy =
        InheritanceType.SINGLE_TABLE)
public class Employee {
    @Id
    @GeneratedValue
    private Long id;

    private String name;
```

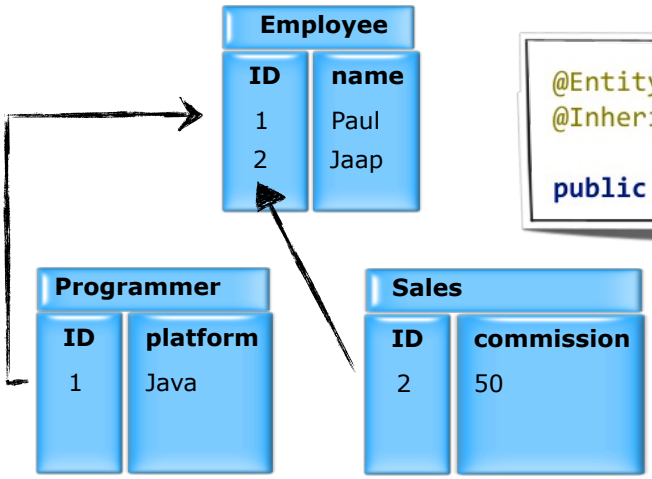
```
@Entity
public class Sales extends Employee{
    private int commission;
```

```
@Entity
public class Programmer extends Employee{
    private String platform;
```

Single Table

- Fast for each type of query
  - Always hit a single table
- Subclass fields must be nullable!

Joined table




```
@Entity
@Inheritance(
        strategy = InheritanceType.JOINED)
public class Employee {
```

- Normalized in the database
- Multiple tables for each type of query

Joined table

```
@Entity
@Inheritance
        (strategy = InheritanceType.TABLE_PER_CLASS)
public class Employee {
    @Id
    @GeneratedValue
            (strategy = GenerationType.TABLE)
    private Long id;
```

Employee	
ID	name

Programmer		
ID	name	platform
1	Paul	Java

Sales		
ID	name	commission
2	Jaap	50

- De-normalized in the database
- Single table queries for specific types
- Slow union query for querying all employees

Embedables

- Map multiple classes to a single table

```
@Entity
public class Purchase {
    @Id
    @GeneratedValue
    private Long id;

    private String orderDescription;

    @Embedded
    private Contact contact;
```

```
@Embeddable
public class Contact {
    @Column(name = "first_name")
    private String firstname;
    private String lastname;
```

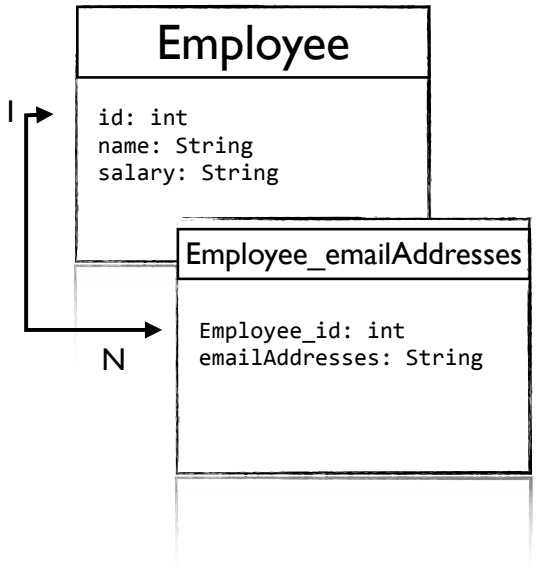
Purchase

```
id: int
orderDescription: String
first_name: String
lastname: String
```

Value collections

- Collections of simple types or embedables

```
@Entity
public class Employee {
    @Id
    private Long id;

    private String name;

    private double salary;

    @ElementCollection
    private Set<String> emailAddresses;
```



Queries

- EntityManager is a factory for Query objects
  - Using createQuery() methods
- Uses new and improved EJB-QL
- Queries can return entities, non-entities, or projections of entity data
- Native queries
  - Not portable across databases!

JPQL

- Query language for entities
- Vendor and implementation independent
- Similar to SQL, with slightly different syntax
- Translated to SQL using a dialect at runtime

Query examples

Select all Employees (polymorphic)

```
SELECT emp From Employee emp
```

Select only Programmers

```
SELECT p From Programmer p
```

Where clause

```
SELECT emp From Employee emp WHERE emp.salary > 3000
```

Between keyword

```
SELECT emp From Employee emp WHERE emp.salary between 2000 and 3000
```

Subquery

```
SELECT emp From Employee emp WHERE emp.salary >
  (SELECT AVG(emp.salary) FROM Employee emp)
```

Query examples

Where clause on relation

```
SELECT emp From Employee emp where emp.department.id = :id
```

Same as above

```
SELECT emp From Department department,
  IN(department.employees) emp WHERE department.id=:id
```

Executing a query

```
TypedQuery<Employee> q = em.createQuery(
        "SELECT emp From Employee emp where emp.department.id = :id",
        Employee.class);

q.setParameter("id", id);
List<Employee> employees = q.getResultList();
```

Query results

Return managed entity

```
SELECT emp From Employee emp
```

Return List<Object[]>

```
SELECT emp.name, emp.department.name From Employee emp
```

Return List<Name>

```
SELECT new demo.Name(emp.name, emp.department.name) From Employee emp
```

```
public class Name {
    private String firstname;
    private String departmentName;

    public Name(String name, String deparmentName) {
        this.firstname = name;
        this.departmentName = deparmentName;
    }
```

Join queries

- A join is generated automatically when:
  - a path expression is used in the select

```
SELECT emp.department.name, emp.name From Employee emp
```

```
select
        department1_.name as col_0_0_,
        employee0_.name as col_1_0_
    from
        Employee employee0_,
        Department department1_
    where
        employee0_.department_id=department1_.id
```

Employees without a department are excluded

Join queries

- Use the join keywords

```
SELECT department.name,
       emp.name
From Employee emp
left outer join emp.department department
```

```
select
        department1_.name as col_0_0_,
        employee0_.name as col_1_0_
    from
        Employee employee0_
    left outer join
        Department department1_
            on employee0_.department_id=department1_.id
```

Employees without a department are now included

Join queries

- Use the join keywords to return collections

```
SELECT department.name,
       emp.name
From Department department
join department.employees emp
```

```
select
        department0_.name as col_0_0_,
        employees1_.name as col_1_0_
    from
        Department department0_
    inner join
        Employee employees1_
            on department0_.id=employees1_.department_id
```

Join queries

- Use the join keywords to prefetch collections

```
SELECT department
From Department department
join fetch department.employees
```

Would normally be lazy loaded

Case expressions

- Conditional expressions

```
SELECT emp.name,
 CASE WHEN TYPE(emp) = Programmer
     THEN 'Cool dev guy'
 ELSE 'Just some guy'
 END
From Employee emp
```

```
select
        employee0_.name as col_0_0_,
        case
            when employee0_.DTYPE='Programmer' then
'Cool dev guy'
            else 'Just some guy'
        end as col_1_0_
    from
        Employee employee0_
```

Bulk updates

- Updating or deleting many entities can be done in a bulk update
  - much cheaper than iterating and updating in Java!

```
Query q = em.createQuery("update Employee e set e.salary = e.salary + 100");
int changed = q.executeUpdate();
```

```
update
        Employee
    set
        salary=salary+100
```

Named queries

- Specify re-usable queries on Entity class

```
@Entity
@NamedQuery(
        name = "findByName",
        query = "select emp from Employee emp WHERE emp.name LIKE :name")
public class Employee {
```

```
TypedQuery<Employee> q = em.createNamedQuery("findByName", Employee.class);
q.setParameter("name", name + "%");
```

Native queries

- JPQL only supports a subview of SQL
  - unsupported features examples:
    - Inline views, hierarchical queries, stored procedures and vender specific extensions
- API more friendly than JDBC
- Supports mapping to Entity classes

Native queries

Map result to Employee objects

```
Query q = em.createNativeQuery(
        "select * from employee where name like ?"
        , Employee.class);
q.setParameter(1, name + "%");

return q.getResultList();
```

Each column must be a property on the Entity

SqlResultMapping

- Specify how query results are mapped to an Entity

```
@SqlResultSetMapping(name = "employeeResult", entities = {
        @EntityResult(entityClass = Employee.class, fields = {
                @FieldResult(name = "name", column = "EMP_NAME"),
                @FieldResult(name = "id", column = "id")})})
public class Employee {
```

```
Query q = em.createNativeQuery(
        "select id, name as EMP_NAME from employee where name like ?"
        , "employeeResult");
```

Criteria API

- The criteria API is used to build queries from code
- Useful for queries that are dynamically created at runtime
  - e.g. a search screen with optional fields
- Does **not replace** JPQL

Criteria API

- CriteriaBuilder
  - contains methods to construct a query (equals, gt, max etc.)
- CriteriaQuery
  - uses a fluent API to build the query
  - the type parameter should be the type that is returned by the SELECT
- Root
  - the first Entity in the FROM

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Employee> c = cb.createQuery(Employee.class);
Root<Employee> emp = c.from(Employee.class);
c.select(emp);

List<Employee> employees = em.createQuery(c).getResultList();
```

where clause

equals operator

the argument

```
c.select(emp).where(cb.equal(emp.get("name"), name));
```

on the 'name' property

predicates and paths

define AND predicate

```
c.select(emp).
        where(
                cb.and(
                        cb.equal(emp.get("name"), name),
                        cb.equal(emp.get("department").get("id"), depId)));
```

path expression

select clause

List<Employee>

```
CriteriaQuery<Employee> c = cb.createQuery(Employee.class);
Root<Employee> emp = c.from(Employee.class);
c.select(emp);
```

List<String>

```
CriteriaQuery<String> c = cb.createQuery(String.class);
Root<Employee> emp = c.from(Employee.class);
c.select(emp.<String>get("name"));
```

select clause cont'd

List<Name>

```
CriteriaQuery<Name> c = cb.createQuery(Name.class);
Root<Employee> emp = c.from(Employee.class);
c.select(
        cb.construct(
                Name.class,
                emp.<String>get("name"),
                emp.<String>get("department").get("name")));
```

List<Object[]>

```
CriteriaQuery<Object[]> c = cb.createQuery(Object[].class);
Root<Employee> emp = c.from(Employee.class);
c.multiselect(
        emp.<String>get("name"),
        emp.<String>get("department").get("name"));
```

joins

- Join defaults to INNER
- Use explicit joins to create OUTER joins

```
Root<Employee> emp = c.from(Employee.class);
Join<Employee, Department> department = emp.join("department", JoinType.LEFT);
c.multiselect(
        emp.<String>get("name"),
        department.<String>get("name"));
```

Fetch Join

```
CriteriaQuery<Department> c = cb.createQuery(Department.class);
Root<Department> d = c.from(Department.class);
d.fetch("employees");
```

Subselect

```
Root<Employee> emp = c.from(Employee.class);
Subquery<Double> sq = c.subquery(Double.class);
Root<Employee> subEmp = sq.from(Employee.class);
sq.select(cb.avg(subEmp.<Integer>get("salary")));
c.select(emp).where(cb.gt(emp.<Integer>get("salary"), sq));
```

Type-safe Meta Model

- The criteria API so far is not type-safe

```
c.select(emp).where(cb.equal(emp.get("name"), name));
```

how do you know "name" is a valid property?

- The criteria API can be used type-safe by introducing a static meta-model

Static meta-model

example entity

```
@Entity
public class Employee {
    @Id
    @GeneratedValue
    private Long id;

    private String name;

    @ManyToOne
    private Department department;

    private double salary;
```

Static meta-model

```
@StaticMetamodel(Employee.class)
public abstract class Employee_ {

    public static volatile SingularAttribute<Employee, Long> id;
    public static volatile SingularAttribute<Employee, Department> department;
    public static volatile SingularAttribute<Employee, String> name;
    public static volatile SingularAttribute<Employee, Double> salary;

}
```

```
CriteriaQuery<Employee> c = cb.createQuery(Employee.class);
Root<Employee> emp = c.from(Employee.class);
c.select(emp).where(cb.equal(emp.get(Employee_.name), name));
```

the name property is now type-safe

Generating the meta-model

- Vendor specific
- Integration in IDEs, Maven and ANT

Optimistic locking

- Make sure two transactions don't modify the same entity without knowing about the other
- Optimistic - data can probably be updated without problems
- Use a version column in the entity
  - No real database lock required

@Version

```
@Version
private int version;
```

- Version can be an int, long or timestamp
- Automatically used by JPA during updates
- Throws `javax.persistence.OptimisticLockException`

```
update
        Contact
    set
        version=?
    where
        id=?
        and version=?
```

Advanced Lock Modes

- A lock mode can be set using
  - EntityManager.lock()
  - EntityManager.refresh()
  - EntityManager.find()
  - Query.setLockMode()
- `LockModeType.OPTIMISTIC`
- `LockModeType.OPTIMISTIC_FORCE_INCREMENT`

Optimistic Read lock

- Prevent an entity to be changed during a transaction
  - optimistic approach to Repeatable Read isolation

```
List<Employee> emps = d.getEmployees();
for (Employee emp : emps) {
    em.lock(emp, LockModeType.OPTIMISTIC);

    totalSalary += emp.getSalary();
}
```

transactions fails if another transaction changed the Employee

Optimistic Write Lock

- Force incrementing the version number
  - even when the entity is not updated!
- Useful for updating the version of a root entity if related entities change

Pessimistic Locking

- Locks rows in the database directly
  - SELECT FOR UPDATE
- Use with care, causes scaling problems easily
  - only use when write concurrency is very high

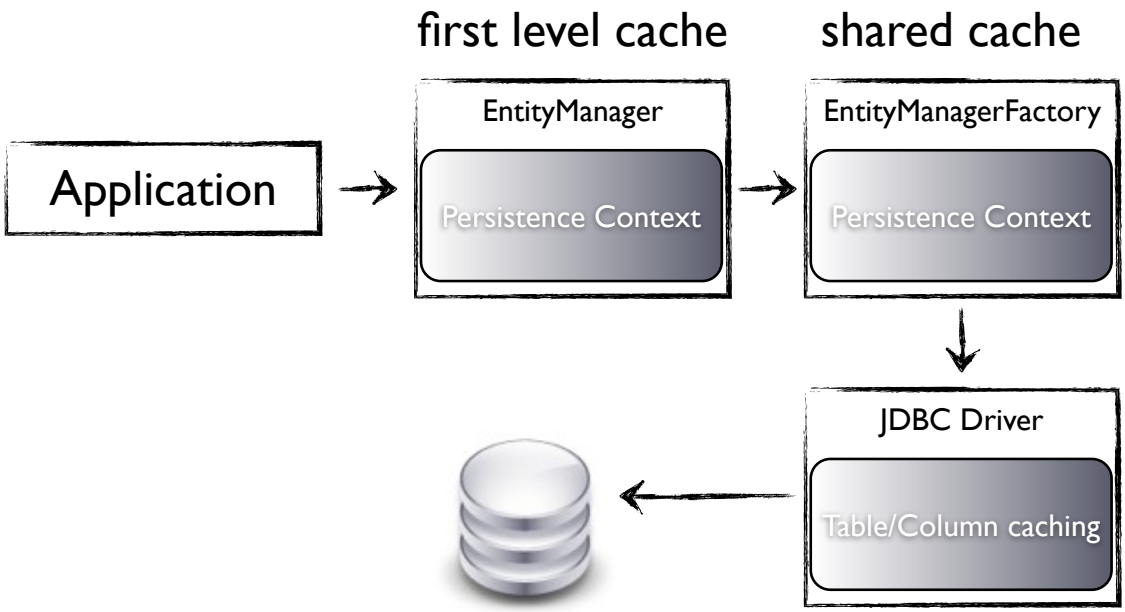
```
for (Employee emp : emps) {
    em.lock(emp, LockModeType.PESSIMISTIC_WRITE);

    totalSalary += emp.getSalary();
}
```

Pessimistic Timeouts

- JPA doesn't describe how timeouts must be implemented
- There is a hint however

```
Map<String, Object> props = new HashMap<String,  Object>();
props.put("javax.persistence.lock.timeout", 5000);
```

Caching



Using the shared cache

- Providers may choose how to use a shared cache
  - Hibernate offers advanced tuning but is disabled by default
  - EclipseLink is simpler but works out-of-the-box

Using the shared cache

- Enable or disable caching for an entity using `@Cacheable(true/false)`
- Default cache usage is configured in persistence.xml

```
<property name="javax.persistence.sharedCache.mode"
          value="ENABLE_SELECTIVE"/>
```

Hibernate example configuration

```
<property name="hibernate.hbm2ddl.auto" value="create"/>

<property name="hibernate.cache.provider_class"
          value="org.hibernate.cache.EhCacheProvider"/>

<property name="hibernate.ejb.classcache.demo.entities.Employee"
          value="read-write"/>
```

ehcache.xml

```
<ehcache>
    <defaultCache
            maxElementsInMemory="10000"/>
    <cache name="demo.entities.Employee"
            maxElementsInMemory="50"/>
    <cache name="org.hibernate.cache.QueryCache"
            maxElementsInMemory="500"/>
</ehcache>
```

Cache properties

- Cache properties can be passed to most EntityManager methods

```
Map<String, Object> props = new HashMap<String, Object>();
props.put("javax.persistence.cache.retrieveMode",
        CacheRetrieveMode.BYPASS);
props.put("javax.persistence.cache.storeMode",
        CacheStoreMode.REFRESH);

em.find(Employee.class, 1L, props);
```

Lifecycle Events

- Create life-cycle event listeners
  - @PrePersist / @PostPersist
  - @PreUpdate / @PostUpdate
  - @PreRemove / @PostRemove
  - @PostLoad

```
@PrePersist
public void prePersistHook() {
    System.out.println("Hi, I'm going to be persisted");
}
```

Persistence Unit

- Set of entities and related classes that share the same configuration
  - Each unit must have unique name
  - Empty string is also considered unique
- Packaging and deployment unit
  - Standard jar file
  - Contains persistence.xml file in META-INF/
  - Optionally, contains orm.xml file in META-INF/

persistence.xml

- Contains one or more <persistence-unit> elements

Out-of-container configuration

```
<persistence-unit name="testPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
        <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver"/>
        <property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:mem:test"/>
        <property name="javax.persistence.jdbc.user" value="sa"/>
        <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>

        <property name="hibernate.archive.autodetection" value="class"/>
        <property name="hibernate.show_sql" value="true"/>
        <property name="hibernate.format_sql" value="true"/>

        <property name="hibernate.hbm2ddl.auto" value="create"/>
    </properties>
</persistence-unit>
```

persistence.xml

EJB3 configuration

```
<persistence-unit name="mainPU">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/jpa_examples</jta-data-source>
    <properties>
        <property name="hibernate.show_sql" value="true"/>
        <property name="hibernate.format_sql" value="true"/>
        <property name="hibernate.dialect"
                  value="org.hibernate.dialect.MySQLDialect"/>
        <property name="hibernate.hbm2ddl.auto" value="update"/>
    </properties>
</persistence-unit>
```

Validation

- Bean Validation API (JSR-303)
- Define field constraints on Entities
- Constraints are validated on persist
- Many other frameworks integrate with Bean Validation
  - e.g. JSF 2.0 and Spring 3

Validation Example

```
@Entity
public class Employee {
    @Id
    private Long id;

    @NotNull @Size(min = 2, max = 20)
    private String name;

    @Past
    private Date birthDate;

    @Min(value = 1000)
    private double salary;
```

- ☐ @Null
- ☐ @NotNull
- ☐ AssertTrue
- ☐ AssertFalse
- ☐ @Min
- ☐ @Max
- ☐ @DecimalMin
- ☐ @DecimalMax
- ☐ @Size
- ☐ @Digits
- ☐ @Past
- ☐ @Future
- ☐ @Pattern

Invoking Validation

```
validator.validate(Object instance, Class<?>... groups)
```

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();

Set<ConstraintViolation<Employee>> violations =
        validator.validate(emp1, Communicating.class);

for (ConstraintViolation<Employee> violation : violations) {
    System.out.println("Error: " + violation.getMessage());
}
```

Creating constraints

- Create annotation
- Implement validator class

```
@ElementCollection
@NonEmptyCollection
private Set<String> emailAddresses;
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.FIELD})
@Constraint(validatedBy = NonEmptyCollectionValidator.class)
public @interface NonEmptyCollection {
    String message() default "Collection may not be empty";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Creating constraints

```
public class NonEmptyCollectionValidator implements
        ConstraintValidator<NonEmptyCollection, Collection<?>> {
    @Override
    public void initialize(NonEmptyCollection nonEmptyCollection) {

    }

    @Override
    public boolean isValid(Collection<?> objects,
                           ConstraintValidatorContext
                                   constraintValidatorContext) {
        return objects != null && objects.size() > 0;
    }
}
```

Validation Groups

- Define different sets of constraints for different situations
- Trigger validation only for a certain constraint

```
@ElementCollection
@NonEmptyCollection(groups = Communicating.class)
private Set<String> emailAddresses;
```

```
public interface Communicating {
}
```

```
validator.validate(emp1, Communicating.class);
```