DZone Refcardz

**CONTENTS INCLUDE:**

# What's New in JPA 2.0

*By Mike Keith*

## INTRODUCTION

The Java Persistence API is the standard for persisting Java objects to a relational database. It includes standardized metadata for defining mappings and configuration, as well as a set of Java APIs that applications can use to persist and query entities. It also includes a standard Service Provider Interface (SPI) that allows applications to easily plug in different JPA providers. If you are new to JPA, read the Refcard *Getting Started with JPA* first to get an understanding of some basic concepts. This Refcard assumes you are familiar with JPA 1.0 and that you now want to upgrade your knowledge to include JPA 2.0. This Refcard covers the primary new features introduced in the JPA 2.0 release and explains how they can be used.

## JDBC PROPERTIES

A set of four new properties were added to provide more portability when specifying the JDBC connection parameters in Java SE mode. The driver class, URL, user, and password were previously different for every provider, but now the four standard ones can be used in a persistence.xml file.

Using the standard JDBC properties:

```
<persistence-unit name=”Account”>
    …
    <properties>
        <property name=”javax.persistence.jdbc.driver”
                  value=”org.apache.derby.jdbc.ClientDriver”/>
        <property name=”javax.persistence.jdbc.url”
                  value=”jdbc:derby://dbmachine:1527/accountDB”/>
        <property name=”javax.persistence.jdbc.user” value=”app”/>
        <property name=”javax.persistence.jdbc.password”
                  value=”app”/>
    </properties>
</persistence-unit>
```

## ACCESS MODE

When the provider accesses the state of an entity, it must typically decide whether the state is in fields or in properties. Until JPA 2.0, all of the state in an entity hierarchy had to be accessed either in one mode or the other. Now, through the use of some judiciously placed metadata, some state in an object can be accessed in fields while other state can be accessed through bean property methods.

The entity must first be annotated with an **@Access** annotation to indicate the default access mode for the class. Then, to deviate from the default and specify accessing in a different mode for a given field or property, the **@Access** annotation can be placed on the field or property to be accessed.

```
@Entity @Access(FIELD)
public class Person {
    @Id
    int id;
      …
    @Access(PROPERTY)
    public String getName() { … }
    public void setName(String pName) { … }
      …
}
```

**Hot Tip** When overriding the access type for a field to its accompanying property make sure to annotate the field with **@Transient** to prevent the field from being mapped *in addition to* the property.

## MAPPINGS

The mapping arsenal grew dramatically in JPA 2.0 to include mappings that are less common, but were not defined in the first release. A number of mappings were added to provide better support for reading from pre-existing (so-called ”legacy”) database schemas. While these mappings are not hugely useful to the average application, they come in handy to those who must work with a schema that is outside the control of the Java developer.

### Element Collections

Arguably, the most useful of the new mapping types is the element collection, which allows an entity to reference a collection of objects that are of a basic type (such as **String** or **Integer**). The **@ElementCollection** annotation is used to indicate the mapping. The objects are stored in a separate table called a collection table, which defaults to be named <entityName>_<attributeName>, but can be overridden with the **@CollectionTable** annotation. The name of the column in the collection table that stores the values defaults

to the name of the attribute but is overridable with the **@Column** annotation.

Element collection of String:

```
@Entity
public class Person {
    …
    @ElementCollection
    @CollectionTable(name="NICKNAMES")
    @Column(name="NNAME")
    Collection<String> nicknames;
    …
}
```

Element collection mappings can be used in embeddables and mapped superclasses as well as entities. They can also contain embeddable objects instead of basic types.

## Adding a Join Table

Normally, a bidirectional one-to-many relationship is mapped through a foreign key on the "many" side. However, in some data schemas, the join is done using a separate join table. Similarly a unidirectional or bidirectional one-to-one relationship can use a join table instead of the typical simple foreign key. As one might expect, using a join table for these mappings is as easy as putting an additional **@JoinTable** annotation on the owning side of the mapped relationship.

## Unidirectional One-to-Many With No Join Table

In JPA 1.0, a unidirectional one-to-many relationships required a join table. However, you were stuck if the schema was fixed and had a foreign key in the target table and the target entity was not able to be modified to have a reference back to the source entity.

The ability to do this was added in JPA 2.0 by permitting a **@JoinColumn** annotation to accompany a **@OneToMany** annotation that was unidirectional (did not contain a **mappedBy** element). The join column refers to a foreign key column in the target table, which points to the primary key of the source table.

Unidirectional one-to-many relationship with target foreign key:

```
@Entity
public class Customer {
    …
    @OneToMany
    @JoinColumn(name="CUST_ID")
    List<Purchase> purchases;
    …
}
```

Hot Tip

A unidirectional one-to-many target foreign key mapping may seem to make life easier in Java but performs worse than using a join table.

## Orphan Removal

A parent-child relationship is a one-to-many or one-to-one relationship in which the target entity (the child) is owned by the source entity (the parent) or relies upon it for its existence. If either the parent gets deleted or the relationship from the parent to the child gets severed, then the child should be deleted. Previous support for cascading delete offered a solution for the first case; but to solve the second case when the child is left an orphan, the new **orphanRemoval** element of the **@OneToMany** or **@OneToOne** annotations can be set to true to cause the child object to get removed automatically by the provider.

Configuring and using orphan removal:

```
@Entity public class Library {
    …
    @OneToMany(mappedBy="library", orphanRemoval=true)
    List<Magazine> magazines;
    …
}
```

In the application code, when a magazine is torn and is to be discarded from the library, the act of removing the magazine from the library causes the magazine to automatically be deleted:

```
library.getMagazines().remove(magazine);
```

## Persistently Ordered Lists

When a many-valued relationship causes fetching of the related entities in a **List** and the order of the entities must be determined by sorting one or more attributes of the target entity type, the relationship mapping needs to be annotated with **@OrderBy**. However, if the order is determined solely by the position of the entity in the in-memory list at the time the list was last persisted, then an additional column is required to store the position. This column is called an *order column* and is specified on the mapping by the presence of **@OrderColumn**. The entity's position in the list is read from and written to this column.

Ordering without using an object attribute:

```
@Entity public class WaitList {
    …
    @OneToMany
    @OrderColumn(name="POSITION")
    List<Customer> customer;
    …
}
```

Every time a transaction causes a customer to be added to or removed from the waiting list, the updated list ordering will be written to the order column. Because the example is using a unidirectional one-to-many relationship that defaults to using a join table the order column will be in the join table.

## Maps

Using a **Map** for a many-valued relationship traditionally meant that entities were the values and some attribute of the entities was the key. The new **Map** features allow **Map** types to be used with keys or values that can be basic types, entities, or embeddables. When the value is an entity type, then the mapping is a relationship. When the value is a basic or embeddable type, the mapping is an element collection. Two examples of using a **Map** are shown. The **deliverySchedule** attribute is an element collection of dates on which deliveries are made to given address strings. The **@MapKeyColumn** annotation indicates the column in the collection table where the keys (address strings) are stored, and **@Column** references the column in which the date values are stored. The **@Temporal** annotation is used because the values are **Date** objects.

The **suppliers** attribute illustrates a unidirectional one-to-many relationship that has a **Map** of **Supplier** entities keyed by **Part** entities. The **@MapKeyJoinColumn** annotation indicates the join column in the join table referring to the **Part** entity table.

Element collection Map and one-to-many Map:

```
@Entity public class Assembly {
    …
    @ElementCollection
    @CollectionTable(name="ASSY_DLRVY")
    @MapKeyColumn(name="ADDR")
    @Column(name="DLRVY_DATE")
    @Temporal(TemporalType.DATE)
    Map<String, Date> deliverySchedule;
    …
    @OneToMany
    @JoinTable(name="PART_SUPP")
    @MapKeyJoinColumn(name="PART_NO")
    Map<Part, Supplier> suppliers;
    …
}
```

The main lesson is that different annotations are useful and applicable depending upon the key and value types in the Map.

## Derived Identifiers

New allowances were made in JPA 2.0 for the case when an entity has a compound primary key that includes a foreign key. Multiple **@Id** annotations can be specified, and they can be on an owned one-to-one or many-to-one mapping. The foreign key of that mapping will be included in the id class, as shown in the example. The id class must still have a field for each of the primary key components and be named the same as the entity attributes, but the types of the foreign key-based id class fields must match the identifier types of the target entity of the relationship. For example, the **dept** field in the **ProjectId** class is not of type **Department** as it is in **Project**, but is of type **int**, which is the identifier type of the **Department** entity.

Derived Identifier:

```
@IdClass(ProjectId.class)
@Entity public class Project {
    …
    @Id
    String projectName;

    @Id @ManyToOne
    Department dept;
    …
}
@Entity public class Department {
    …
    @Id
    int id;
    …
}
public class ProjectId {
    String projectName;
    int dept;
    …
}
```

Compound primary keys may also be composed of multiple foreign keys or relationships by adding **@Id** annotations to additional relationships in the entity and adjusting the id class accordingly.

**Hot Tip** Derived identifiers also support a multitude of additional identifier combinations involving embedded identifiers, multiple levels of compounding, and shared relationship primary keys.

## Java Persistence Query Language

A number of additional JP QL features were added to support the new mappings, while other features were added simply as improvements to the query language. The table summarizes the changes.

New features of JP QL:

| Feature Name | Description | Example |
|---|---|---|
| Date, time, and timestamp literals | JDBC syntax was adopted: {d 'yyyy-mm-dd'} {t 'hh-mm-ss'} {ts 'yyyy-mm-dd hh-mm-ss'} | SELECT c FROM Customer c WHERE c.birthdate < {d '1946-01-01'} |
| Non-polymorphic queries – TYPE | Can query across specific subclasses of a superclass | SELECT p FROM Project p WHERE TYPE(p) = DesignProject OR TYPE(p) = QualityProject |
| Map support - KEY, VALUE, ENTRY | Allow comparison and selection of keys and values and selection of entries | SELECT e.name, KEY(p), VALUE(p) FROM Employee e JOIN e.phones p WHERE KEY(p) IN ('Work', 'Cell') |
| Collection input parameters | Allow parameter arguments to be collections | SELECT e FROM Employee e WHERE e.lastName IN :names |
| CASE statement | Can be in either of two forms: 1) CASE {WHEN conditional THEN scalarExpr}+ ELSE scalarExpr END<br><br>2) CASE pathExpr {WHEN scalarExpr THEN scalarExpr}+ ELSE scalarExpr END | UPDATE Employee e SET e.salary = CASE WHEN e.rating = 1 THEN e.salary * 1.1 WHEN e.rating = 2 THEN e.salary * 1.05 ELSE e.salary * 1.01 END |
| NULLIF, COALESCE | Additional CASE variants: COALESCE(scalarExpr {, scalarExpr}+)<br><br>NULLIF(scalarExpr, scalarExpr) | SELECT COALESCE(d.name, d.id) FROM Department d |
| Scalar expressions in the SELECT clause | Return the result of performing a scalar operation on a selected term | SELECT LENGTH(e.name) FROM Employee e |
| INDEX in a List | Refer to an item's position index in a list | SELECT p FROM Flight f JOIN f.upgradeList p WHERE f.num = 861 AND INDEX(p) = 0 |
| Variables in SELECT constructors | Constructors in SELECT clause can contain identification vars | SELECT new CustInfo(c.name, a) FROM Customer c JOIN c.address a |

## TYPED QUERY

One of the simplest but most useful query features added in JPA 2.0 is the TypedQuery interface. When you create a query you can pass in the type for the query result and get back a typed query object that, when executed, will return the correctly typed objects without having to cast. It works for both dynamic queries and named queries.

```
TypedQuery<Employee> q = em.createNamedQuery("Employee.findByName",
                                              Employee.class);
q.setParameter("empName", "Smith");
List<Employee> emps = q.getResultList();
```

## SHARED CACHE

Entities regularly get cached in a persistence context, but they will also typically get cached by the persistence provider in a longer-lived cache in the entity manager factory. This cache is called a shared cache because the entity data is shared across multiple persistence contexts. Although there are multiple different strategies and approaches to caching, a standard API can be used to operate on it. The standard **Cache** interface can be obtained from the **EntityManagerFactory.getCache()** method.

```
public interface Cache {
    public boolean contains(Class cls, Object primaryKey);
    public void evict(Class cls, Object primaryKey);
    public void evict(Class cls);
    public void evictAll();
}
```

**Hot Tip** It is not a good idea to manipulate the cache as part of the regular application execution. The API is most valuable for testing to clear the cache between test cases/runs.

A **shared-cache-mode** element can be configured in the persistence.xml file for a given persistence unit. It is used in conjunction with the **@Cacheable** annotation on entity classes to designate whether instances of entity classes are allowed (or intended) to be cached in the shared cache.

| Option | Description |
|---|---|
| ALL | All entities in the persistence unit are cached |
| NONE | Caching is disabled for all entities in the persistence unit |
| ENABLE_SELECTIVE | Caching is disabled, except for entities annotated with @Cacheable or @Cacheable(true) |
| DISABLE_SELECTIVE | Caching is enabled, except for entities annotated with @Cacheable(false) |
| UNSPECIFIED | Caching reverts to the default caching option for the current provider |

**Hot Tip** Providers aren't strictly required to implement shared caches. If a provider has no cache the cache API is not going to have an effect.

Two additional properties can influence what gets cached at the operational level. The **javax.persistence.cache.retrieveMode** and **javax.persistence.cache.storeMode** properties can be set at the level of the entity manager using a **setProperty()** method, an entity manager **find()** or **refresh()** operation, or on an individual query **setHint()** call. The enumerated **javax.persistence.CacheRetrieveMode** type contains the valid values for the retrieve mode, and the enumerated **javax.persistence.CacheStoreMode** type defines the valid store mode values.

| CacheRetrieveMode Value | Description |
|---|---|
| USE | Read entity data from the cache |
| BYPASS | Don't use the cache; read entity data from the database |

| CacheStoreMode Value | Description |
|---|---|
| USE | Insert entity data into cache when reading from/ writing to the database |
| BYPASS | Don't insert entity data into the cache |
| REFRESH | Same as USE, but refresh cached version of entity data when reading from database |

## ADDITIONAL API

Some of the primary API classes have had additional methods added to them to provide more functionality and offer more flexibility, and most of the pertinent ones are listed in the Tables. Some new API classes have also been added.

Additional EntityManager methods:

| Method | Description |
|---|---|
| find(Class entityClass, Object pk [,LockModeType lockMode] [,Map<String,Object> properties]) | Overloaded find methods allowing optional lockmode and properties parameters. |
| lock(Object entity, Map<String,Object> properties) | Overloaded lock method allowing properties parameter |
| refresh(Object entity [,LockModeType lockMode] [,Map<String,Object> properties]) | Overloaded refresh methods allowing optional lockmode and properties parameters |
| detach(Object entity) | Remove the entity from the persistence context |
| createQuery(String qlString, Class<T> resultClass) | Create a dynamic query and return a TypedQuery<T> |
| createQuery(CriteriaQuery<T> criteriaQuery) | Create a query using the Criteria API and return a TypedQuery<T> |
| createNamedQuery(String queryName, Class<T> resultClass) | Create a named query and return a TypedQuery<T> |
| unwrap(Class<T> cls) | Return an instance of the (often proprietary) specified class associated with the entity manager |
| getEntityManagerFactory() | Return the entity manager factory associated with the entity manager |
| getCriteriaBuilder() | Return a criteria builder for building criteria queries |
| getMetamodel() | Return the metamodel for introspecting the entity structure |

Additional Classes:

| Class | Description |
|---|---|
| TypedQuery<T> | Subclass of Query that is typed according to the result type of the query. |
| Tuple | Representation of a row of typed data elements |
| TupleElement<T> | Representation of a single typed data element in a tuple |
| Parameter<T> | Typed query parameter |
| PersistenceUnitUtil | Class containing a group of utility methods implemented and returned by the provider of a given persistence unit |
| PersistenceUtil | Class containing a group of utility methods that will work across any and all providers |
| PersistenceProviderResolver | Pluggable class used to resolve providers in different environments. |
| PersistenceProviderResolverHolder | Static methods to set/look up the resolver |

## PESSIMISTIC LOCKING

One of the benefits of the overloaded entity manager **find()** and **refresh()** methods taking a lock mode is to enable locking at the operational level. This is most valuable when using the new **LockModeType.PESSIMISTIC_WRITE** option to pessimistically lock an entity. There is even a **javax.persistence.lock.timeout** property that can be used to put an upper bound on the wait time to acquire the lock. A **PessimisticLockException** will be thrown if the lock could not be acquired and the transaction has been rolled back.

```
Map<String,Object> props = new HashMap<String,Object>();
props.put(“javax.persistence.lock.timeout”, 5000);
try {
    Account acct = em.find(Account.class, accountId, PESSIMISTIC_WRITE,
props);
} catch (PessimisticLockException pessEx) {
    // lock not acquired, bail out and do something reasonable
}
```

The available lock modes are listed in the following table.

Lock Modes:

| | |
|---|---|
| OPTIMISTIC | New name for JPA 1.0 "READ" mode |
| OPTIMISTIC_FORCE_INCREMENT | New name for JPA 1.0 "WRITE" mode |
| PESSIMISTIC_READ | Pessimistic repeatable read isolation |
| PESSIMISTIC_WRITE | Pessimistically lock to cause write serialization |
| PESSIMISTIC_FORCE_INCREMENT | Pessimistically lock but also ensure update occurs to version field |

## VALIDATION

Validation is a new specification in its own right and can be used with virtually any Java object, not just persistent entities. However, when used with JPA, some integration was beneficial to provide the additional automatic lifecycle validation support. Validation can be configured to be automatically triggered during any of the PrePersist, PreUpdate, and PreRemove entity lifecycle events. A **validation-mode** element in the **persistence.xml** file, or the equivalent **javax.persistence.validation.mode** property passed in at entity manager factory creation time, will determine whether validation happens. A value of ***auto*** means that validation will occur if a validator is available, while a value of ***callback*** means that validation is expected to occur and will fail if no validator is available. Setting the mode to ***none*** disables validation. Because the default validation-mode is ***auto***, validation of a validation provider that exists on the classpath can be disabled by explicitly setting the mode to ***none***.

Specific validation groups can be used for validation at a given lifecycle event through the corresponding properties: **javax.persistence.validation.group.pre-persist**, **javax.persistence validation.group.pre-update**, and **javax.persistence.validation group.pre-remove**. These properties may be applied either as property elements in the **persistence.xml** file or as dynamic properties passed in at entity manager creation. For example, to cause validation of the **com.acme.validation.EntityCreation** validation group to the entities of a persistence unit at the PrePersist lifecycle event, the **javax.persistence.validation.group pre-persist** property value should be set to that fully qualified **EntityCreation** class name. Then, assuming the existence of **MyEntity** defined in the persistence unit with EntityCreation constraints on it, a call to persist an instance of **MyEntity** would cause those constraints to be validated before the instance data would be inserted to the database.

For more details about validation, consult the "Bean Validation" specification available at http://jcp.org/en/jsr/detail?id=303.

## ENTITY METAMODEL

Although it may be more useful for tools than for regular developers, one interesting feature that was introduced in JPA 2.0 was the ability to access a metamodel of the objects mapped in a persistence unit. The metamodel is available at runtime to query the structure of the objects and is obtained by calling **getMetamodel()** on an entity manager or entity manager factory. The example shows how the metamodel can be used to introspect the attributes of the **Account** entity.

```
Metamodel model = em.getMetamodel();
EntityType<Account>  account_ = model.entity(Account.class);
for (Attribute<? super Account, ?> attr : account_.getAttributes()) {
    System.out.println("Account attribute: " + attr.getName() +
                       "  java type = " + attr.getJavaType.getName() +
                       "  mapping type = " +
                       attr.getPersistentAttributeType());
}
```

## CRITERIA API

Perhaps the biggest new feature (certainly in terms of page count!) is the Criteria API for dynamic query creation. It comprises a Java API to incrementally add nodes to a criteria tree that can be subsequently passed to a query wrapper for evaluation. The criteria nodes are typed according to their semantics and the type of object they are being applied to.

Hot Tip

The Criteria API is not a replacement for JP QL queries but an alternative query mechanism for developers who want to use a more dynamic or more strongly typed API. Developers happy with JP QL may not ever have the need to use the Criteria API.

There are two flavors of criteria usage. The one you will use depends on the reason you are using the API, the development practices of your company, and personal preference.

## STRING-BASED CRITERIA

String-based criteria implies specifying attribute names as strings, much like many of the existing criteria APIs and expression frameworks present in Hibernate and TopLink/EclipseLink. The resulting expressions are more typed than what simple JP QL will offer but less typed than the second strongly typed flavor of the Criteria API. They may also result in "raw type" compiler warnings indicative of not declaring the type of a collection that was defined as a parameterized type.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery c = cb.createQuery(Account.class);
Root acct = c.from(Account.class);
c.select(acct)
 .where(cb.equal(acct.get("name"), "Jim Morrison"));
List result = em.createQuery(c).getResultList();
```

We start by obtaining a CriteriaBuilder, the primary factory both for the criteria query object and for most of the criteria nodes, from the entity manager. We then create the criteria object, get a root, declare our selection clause, and start adding constraint nodes. Of course, declaring the selection clause and building the constraint tree are not really order-dependent and could be done in either order. Once complete, though, the query can be executed through the traditional means of query execution just by passing the criteria object as an argument to the **EntityManager.createQuery()** method and calling **getResultList()** on the returned query.

A better typed version of the query can be achieved by adding the types but still using string-based attributes. The types make the code slightly harder to read and use but provide a measure of typing to improve the quality.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Account> c = cb.createQuery(Account.class);
Root<Account> acct = c.from(Account.class);
c.select(acct)
 .where(cb.equal(acct.get("name"), "Jim Morrison"));
List<Account> result = em.createQuery(c).getResultList();
```

## STRONGLY TYPED CRITERIA

The problem with the string-based versions above is that the "**name**" attribute can be mistyped, or the wrong attribute can be specified. The compiler will not be able to catch the error because its argument type is String. As far as the compiler is concerned, any string will do. To protect against these kinds of attribute errors at compile time, the argument must be typed very specifically to match the attribute type. The metamodel types provide the tools to do this.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Account> c = cb.createQuery(Account.class);
Root<Account> acct = c.from(Account.class);
c.select(acct)
 .where(cb.equal(acct.get(Account_.name), "Jim Morrison"));
List<Account> result = em.createQuery(c).getResultList();
```

To use the strongly typed criteria, the "**name**" string is substituted by **Account_.name**. The **Account_** class is called the *canonical metamodel* class corresponding to the **Account** entity. It is generated by the provider for the purposes of the strongly typed Criteria API and contains a static field for each of its mapped attributes. While the details are not that important, the Account_ class can be used for these kinds of strongly typed queries. The compiler will not let you mistype the name of the attribute or supply the wrong attribute because all of the objects along the way are typed strongly enough to catch these kinds of errors.

## CRITERIABUILDER

You may have noticed from the examples that if we wanted to start adding operators to the where clause, we needed to obtain them from the **CriteriaBuilder** instance. The **CriteriaBuilder** class acts as a node factory for operations, (both string-based and strongly typed), and contains most of the useful operators. The following reference table shows at a glance the operators (by category) that are available in the **CriteriaBuilder** class.

The arguments for many of these methods are overloaded for convenience.

CriteriaBuilder methods by category:

| Category | Method Names |
|---|---|
| Query Construction | createQuery(), createTupleQuery() |
| Selection Construction | construct(), tuple(), array() |
| Ordering | asc(), desc() |
| Aggregate Functions | avg(), sum(), sumAsLong(), sumAsDouble(), min(), max(), greatest(), least(), count(), countDistinct() |
| Subqueries | exists(), all(), some(), any() |
| Boolean Functions | and(), or(), not(), conjunction(), disjunction() |
| Value Testing | isTrue(), isFalse(), isNull(), isNotNull() |
| Equality | equal(), notEqual() |
| Numeric Comparison | gt(), ge(), lt(), le() |
| Comparison | greaterThan(), greaterThanOrEqualTo(), lessThan(), lessThanOrEqualTo(), between() |
| Numeric Operations | neg(), abs(), sum(), prod(), diff(), quot(), mod(), sqrt() |
| Typecasts | toLong(), toInteger(), toFloat(), toDouble(), toBigDecimal(), toBigInteger(), toString() |
| Literals | literal(), nullLiteral() |
| Parameters | parameter() |
| Collection Operations | isEmpty(), isNotEmpty(), size(), isMember(), isNotMember() |
| Map Operations | keys(), values() |
| String Operations | like(), notLike(), concat(), substring(), trim(), lower(), upper(), length(), locate() |
| Temporal Functions | currentDate(), currentTime, currentTimestamp() |
| Expression Construction | in(), selectCase() |
| Misc Operations | coalesce(), nullif(), function() |

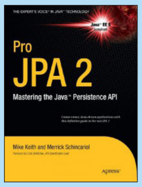
## SUMMARY

While we have not been completely exhaustive in our coverage of the new JPA 2.0 features, we have covered the vast majority and most interesting features among them. Clearly, JPA 2.0 has propelled JPA into an even more mature position in the world of persistence, causing it to be more easily applied to old and new data schemas alike.

## ABOUT THE AUTHOR

**Mike Keith** has been a distributed systems and persistence expert for over 20 years. He co-led the expert group that produced the first release of JPA and co-authored the premier JPA reference book, *Pro EJB: Java Persistence API*, followed up with *Pro JPA 2: Mastering the Java Persistence API*. He works at Oracle as an architect for enterprise Java and represents Oracle on numerous expert groups and specifications, including the Java EE platform specification, JPA, and others. He leads the Gemini Enterprise Modules project at Eclipse, and is a member of the Eclipse Runtime project PMC.

## RECOMMENDED BOOK



*Pro JPA 2* introduces, explains, and demonstrates how to use the new Java Persistence API (JPA). JPA provides Java developers with both the knowledge and insight needed to write Java applications that access relational databases through JPA.

**BUY NOW**
http://apress.com/book/view/9781430219569

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, blogs, feature articles, source code and more. **"DZone is a developer's dream,"** says PC Magazine.

DZone, Inc.
140 Preston Executive Dr.
Suite 100
Cary, NC 27513

888.678.0399
919.678.0300

**Refcardz Feedback Welcome**
refcardz@dzone.com

**Sponsorship Opportunities**
sales@dzone.com

ISBN-13: 978-1-936502-03-5
ISBN-10: 1-936502-03-8
50795
9 781936 502035
$7.95

Version 1.0