DZone

REFCARDZ
tech facts at your fingertips

CONTENTS INCLUDE:

Getting Started with JPA

By Mike Keith

GETTING STARTED WITH JPA

The Java Persistence API (JPA) is the Java standard for mapping Java objects to a relational database. Even though proprietary mapping products like Hibernate and TopLink still exist, they are now focused on providing their functionality through the JPA API, allowing all applications to be portable across JPA implementations. This refcard will give users enough to understand the basics of JPA and get started writing JPA applications. It covers entities, identifiers, O-R mappings, using an entity manager, creating and executing queries, and configuration of the `persistence.xml` file.

MAPPING AN OBJECT

The basic unit of persistence in JPA is the **entity**, which is nothing more than a regular Java class with metadata to describe how its state maps to the database tables. Metadata may be in the form of annotations on the entity class itself, or it may be an accompanying XML file, but we are using annotations since they are easier to specify and understand.

Hot Tip When used together, XML mappings can override the values specified in annotations

Every entity class should have an `@Entity` marker and an **identifier** field, indicated by `@Id`, that is mapped to the primary key column in the database. When a field contains simple data and maps to a regular column in the database we call it a basic mapping, thus an identifier field is a special kind of basic mapping. When an entity has a field that references one or more other entities, that field maps to a foreign key column, and is called a relationship field. Other than the identifier field, basic mappings do not need to be annotated, but relationships must be specified by their relationship cardinality.

Defaulting rules in JPA mean that you are not required to specify table names and column names that an entity is mapped to. If you are not happy with the JPA-assigned defaults then you can always override them through the use of additional mapping metadata. For example, by putting `@Table` on the entity class you can make the table name explicit, and by annotating a basic mapping field with `@Column` you can define the particular column that maps the state in that field. Likewise `@JoinColumn` is used to override the name of the foreign key column for relationship references.

An example of two mapped entities are the `Pet` and `Owner` classes shown in Listings 1 and 2.

Listing 1 – Pet entity class

```
@Entity
@Table(name="PET_INFO")
public class Pet {
   @Id
   @Column(name="ID")
   int licenseNumber;
   String name;
   PetType type;
   @ManyToOne
   @JoinColumn(name="OWNER_ID")
   Owner owner;
   ...
}
```

Listing 2 - Owner entity class

```
@Entity
public class Owner {
   @Id
   int id;
   String name;
   @Column(name="PHONE_NUM")
   String phoneNumber;
   @OneToOne
   Address address;
   @OneToMany(mappedBy="owner")
   List<Pet> pets;
   ...
}
```

In a bidirectional relationship pair, such as the `@OneToMany` relationship in `Owner` to `Pet` and the `@ManyToOne` relationship back from `Pet` to `Owner`, only one foreign key is required in one of the tables to manage both sides of the relationship. As a general rule, the side that does not have the foreign key in it specifies a `mappedBy` attribute in the relationship annotation and specifies the field in the related entity that maps the foreign key.

→

Mapping an Object, continued

The possible mapping annotations that can be used are:

```
@Basic          @Enumerated     @ManyToOne     @Temporal
@Embedded       @Lob            @OneToMany     @Transient
@EmbeddedId     @ManyToMany     @OneToOne
```

Annotations used to override the names of the tables or columns in the database are:

```
@AttributeOverride(s)        @PrimaryKeyJoinColumn(s)
@AssociationOverride(s)      @SecondaryTable(s)
@Column                      @SequenceGenerator
@DiscriminatorColumn         @Table
@JoinColumn(s)               @TableGenerator
@JoinTable
```

Other annotations used to indicate the type of the class or other aspects of the model are:

```
@Entity            @IdClass              @MappedSuperclass
@Embeddable        @Inheritance          @OrderBy
@GeneratedValue    @DiscriminatorValue   @Version
@Id                @MapKey
```

OBTAINING AN ENTITY MANAGER

The `EntityManager` class is the main API in JPA. It is used to create new entities, manufacture queries to return sets of existing entities, merge in the state of remotely modified entities, delete entities from the database, and more.

There are, generally speaking, two main kinds of entity managers:

container-managed	The managed entity managers may only be obtained within a container that supports the JPA Service Provider Interface (SPI).
non-managed	Non-managed entity managers may be obtained in any environment where a JPA provider is on the classpath. Listing 3 shows an example of obtaining a non-managed entity manager by first obtaining an `EntityManagerFactory` instance from the `Persistence` root class.

Listing 3 – Obtaining a non-managed entity manager

```
import javax.persistence.*;
...
EntityManagerFactory emf = Persistence
   .createEntityManagerFactory("PetShop");
EntityManager em = emf.createEntityManager();
...
em.close();
```

In Listing 4 we see how a standard host container can provide a simpler way to obtain an entity manager. The only catch is that this is only supported within standard Java EE components (or containers that are compliant to the JPA container contract), so this example uses a stateless session bean.

Listing 4 – Injecting a managed entity manager

```
@Stateless
public class MyBean implements MyInterface {
   @PersistenceContext(unitName="PetShop")
   EntityManager em;
   ...
}
```

USING AN ENTITY MANAGER

The basic purpose of an entity manager is to perform create/read/update/delete (CRUD) operations on entities. Listing 5 shows methods that perform these operations.

Listing 5 – Invoking the entity manager

```
public Pet createPet(int idNum, String name, PetType
type) {
   Pet pet = new Pet(idNum, name, type);
   em.persist(pet);
   return pet;
}
public Pet findPet(int id) {
   return em.find(Pet.class, id);
}

public Pet changeName(int id, String newName) {
   Pet pet = this.findPet(id);
   pet.setName(newName);
   return pet;
}
public void deletePet(int id) {
   Pet pet = this.findPet(id);
   em.remove(pet);
}
```

Note that finding the pet is the first step to being able to perform update and delete operations on it. Also, an update does not even involve invoking the entity manager, but requires reading the pet, loading it into the entity manager and then modifying it. The modification will be reflected in the database when the transaction is committed.

Hot Tip The merge() method can also be used to create entities, but is most useful for merging in entity changes made on the client side.

TRANSACTIONS

Since we just mentioned transactions, but didn't explain them, now would be a good time to state that JPA supports two different kinds of transactions.

JTA container transactions	Used when running in container mode
resource local transactions	Typically used when running in non-container mode.

JTA transactions are started and committed using the usual container techniques, either calling the `UserTransaction` API or making use of container-managed transaction demarcation in EJB or Spring. For example, if the methods in Listing 5 were in a session bean that had a `Required` transaction attribute setting then a transaction would be started at the beginning and committed at the end of each client method invocation.

When using local transactions the transaction must be demarcated manually by invoking on the `EntityTransaction` instance accessed from the entity manager. Each of the three methods in Listing 5 that caused the database to change would need to have begin and commit calls, as shown in Listing 6 for the persist method. Methods that only read from the database do not need to occur within a transaction.

Transactions, continued

Listing 6 – Using EntityTransaction

```
public Pet createPet(int idNum, String name, PetType
type) {
   em.getTransaction().begin();
   Pet pet = new Pet(idNum, name, type);
   em.persist(pet);
   em.getTransaction().commit();
   return pet;
}
```

The complete `EntityManager` API is listed in Table 1, with a brief description of what each method does.

Method	Description
`void persist(Object entity)`	Persist an entity
`<T> T merge(T entity);`	Merge the state of an entity into the database
`void remove(Object entity);`	Remove an entity from the database
`<T> T find( Class<T> entityClass, Object primaryKey);`	Find and return an instance of an entity class
`<T> T getReference( Class<T> entityClass, Object primaryKey);`	Create a holder for the primary key of an entity
`void flush();`	Cause all changes to be written out to the database
`void setFlushMode( FlushModeType flushMode);`	Set the flushing mode for query execution
`FlushModeType getFlushMode();`	Get the flushing mode for query execution
`void lock( Object entity, LockModeType lockMode);`	Lock an object to obtain greater isolation consistency guarantees
`void refresh(Object entity);`	Update the in-memory entity with the state from the database
`void clear();`	Make all managed entities become unmanaged
`boolean contains( Object entity);`	Determine if an entity is managed
`Query createQuery( String JP QLString);`	Create a dynamic query from JP QL
`Query createNamedQuery( String name);`	Create a named query
`Query createNativeQuery( String sqlString);`	Create a query from SQL
`Query createNativeQuery( String sqlString, Class entityClass);`	Create a query from SQL that returns a given entity type
`Query createNativeQuery( String sqlString, String resultSetMapping);`	Create a query from SQL that uses a given defined mapping
`void joinTransaction();`	Join an existing JTA transaction
`Object getDelegate();`	Access the underlying EntityManager implementation
`void close();`	Close the EntityManager
`boolean isOpen();`	Determine whether the EntityManager has been closed
`EntityTransaction getTransaction();`	Access the EntityManager local transaction

Table 1. EntityManager method summary

QUERYING

Dynamic queries are objects that are created from an entity manager, and then executed. The query criteria are specified at creation time as a **Java Persistence Query Language** (JP QL) string. Before executing the query a number of possible configuration method calls may be made on the query instance to configure it. Listing 7 shows an example of creating and executing a query that returns all the instances of `Pet`, or the first 100 if there are more than 100 instances.

Listing 7 – Creating and executing a dynamic query

```
Query q = em.createQuery("SELECT p FROM Pet p");
q.setMaxResults(100);
List results = q.getResultList();
```

A **named query** is a query that is defined statically and then instantiated and executed at runtime. It can be defined as an annotation on the entity class, and assigned a name that is used when the query is created. Listing 8 shows a named query defined on the `Pet` entity.

Listing 8 – Defining a named query

```
@NamedQuery(name="Pet.findByName",
   query="SELECT p FROM Pet p WHERE p.name LIKE :pname")
@Entity
public class Pet {
   ...
}
```

The last identifier is prefixed with a colon (:) character to indicate that it is a **named parameter** that must be bound at runtime before the query can be executed. Listing 9 shows a method that executes the query by first instantiating a Query object using the `createNamedQuery()` factory method, then binding the `pname` named parameter to the name that was passed into the method, and finally executing the query by invoking `getResultList()`.

Listing 9 – Executing a named query

```
public List findAllPetsByName(String petName) {
   Query q = em.createNamedQuery("Pet.findByName");
   q.setParameter("pname", petName);
   return q.getResultList();
}
```

Hot Tip Named queries are not only more efficient than dynamic queries but are also safer since they will often get pre-compiled by the persistence implementation at deployment time

The entire Query API is shown in Table 2.

Query Method	Description
`List getResultList();`	Execute the query and return the results as a List
`Object getSingleResult();`	Execute a query that returns a single result
`int executeUpdate();`	Execute an update or delete statement
`Query setMaxResults( int maxResult);`	Set the maximum number of results to retrieve
`Query setFirstResult( int startPosition);`	Set the position of the first result to retrieve
`Query setHint( String hintName, Object value);`	Set an implementation-specific query hint
`Query setParameter( String name, Object value);`	Bind an argument to a named parameter
`Query setParameter( String name, Date value, TemporalType temporalType);`	Bind an instance of java.util.Date to a named parameter

Table 2. Query method summary

Querying, continued

Query Method	Description
`Query setParameter( String name, Calendar value, TemporalType temporalType);`	Bind an instance of java.util.Calendar to a named parameter
`Query setParameter( int position, Object value);`	Bind a parameter by position
`Query setParameter( int position, Date value, TemporalType temporalType);`	Bind an instance of java.util.Date to a positional parameter
`Query setParameter( int position, Calendar value, TemporalType temporalType);`	Bind an instance of java.util.Calendar to a positional parameter
`Query setFlushMode( FlushModeType flushMode);`	Set the flush mode for the query

Table 2. Query method summary, continued

JAVA PERSISTENCE QUERY LANGUAGE

The Java Persistence Query Language is SQL-like, but operates over the entities and their mapped persistent attributes instead of the SQL schema. Many of the SQL functions and even reserved words are supported in JP QL.

There are three basic types of JP QL statements, of which the first is monstrously the most popular and useful: selects, bulk updates and bulk deletes.

1. select_clause from_clause [where_clause] [groupby_clause] [having_clause] [orderby_clause]
2. update_clause [where_clause]
3. delete_clause [where_clause]

Hot Tip Bulk deletes are useful for doing test clean-up and clearing all of the data from the entity tables without having to revert to SQL.

A simplified table of most of the supported syntax is in Table 4. For complete and precise grammar, consult the JPA specification at http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html. The primitive terms are shown in Table 3.

Term	Description
entityName	Name of an entity (which is defaulted to the name of the entity class)
variable	Identifier variable following Java identifier rules
state_field_exp	Term that resolves to an entity field containing simple state (e.g. if Pet is represented by variable p, then p.name or p.owner.phoneNumber)
single_rel_exp	Term that resolves to an entity field containing an one-to-one or many-to-one relationship (e.g. if Pet is represented by variable p, then p.owner or p.owner.address)
multi_rel_exp	Term that resolves to an entity field containing a one-to-many or many-to-many relationship (e.g. if Owner is represented by variable o, then o.pets)
rel_field	Term composed of a variable and one of its relationship fields, with no traversing of intermediate relationships (e.g. if Pet is represented by variable p, then p.owner)
constructor_method	Constructor for a non-entity class (i.e. the name of the class)
input_param	Variable that represents an input parameter and must be bound before the query can be executed
literal	A value of a particular type such as a string or integer (e.g. 'Iggy Pop', or 42)
pattern_value	A valid SQL pattern string (e.g. "% Smith")
escape_char	A character to be escaped

Table 3. Primitive terms for JP QL grammar

Java Persistence Query Language, continued

Clause/Term	Syntax
select_clause	**SELECT [DISTINCT]** select_exp {,select_exp}*
select_exp	variable | state_field_exp | single_rel_exp | aggregate_exp | constructor_exp
aggregate_exp	{{**AVG** | **MAX** | **MIN** | **SUM**} ([**DISTINCT**] state_field_exp)} | **COUNT** ([**DISTINCT**] variable | state_field_exp | single_rel_exp)
constructor_exp	**NEW** constructor_method (constructor_item {,constructor_item}*)
constructor_item	single_rel_exp | aggregate_exp
from_clause	**FROM** variable_decl {, {variable_decl | in_decl}}*
variable_decl	entityName [**AS**] variable {join_exp | fetch_join_exp}*
join_exp	[**LEFT** [**OUTER**] | **INNER**] **JOIN** rel_field [**AS**] variable
fetch_join_exp	[**LEFT** [**OUTER**] | **INNER**] **JOIN FETCH** rel_field
in_decl	**IN** (multi_rel_exp) [AS] variable
where_clause	**WHERE** conditional_exp
conditional_exp	{[**NOT**] conditional} | {conditional_exp {**AND** | **OR**} conditional_exp}
conditional	comparison | between_exp | like_exp | in_exp | compare_null_exp | compare_empty_exp | compare_member_exp | exists_exp
comparison	compare_string | compare_boolean | compare_enum | compare_datetime | compare_entity | compare_arithmetic
compare_string	string_exp {**=** | **>** | **>=** | **<** | **<=** | **<>**} {string_exp | all_any_subquery}
compare_boolean	boolean_exp {**=** | **<>**} {boolean_exp | all_any_subquery}
compare_enum	enum_exp {**=** | **<>**} {enum_exp | all_any_subquery}
compare_datetime	datetime_exp {**=** | **>** | **>=** | **<** | **<=** | **<>**} {datetime_exp | all_any_subquery}
compare_entity	entity_exp {**=** | **<>**} {entity_exp | all_any_subquery}
compare_arithmetic	arithmetic_exp {**=** | **>** | **>=** | **<** | **<=** | **<>**} {arithmetic_exp | all_any_subquery}
all_any_subquery	{**ALL** | **ANY** | **SOME**} (subquery)
between_exp	arithmetic_exp [**NOT**] **BETWEEN** arithmetic_exp **AND** arithmetic_exp
like_exp	string_exp [**NOT**] **LIKE** pattern_value [**ESCAPE** escape_char]
in_exp	state_field_exp [**NOT**] **IN** ({literal | input_param} {,{literal | input_param}}*)
compare_null_exp	{single_rel_exp | input_param} **IS** [**NOT**] **NULL**
compare_empty_exp	multi_rel_exp IS [**NOT**] **EMPTY**
compare_member_exp	entity_exp [**NOT**] **MEMBER** [**OF**] multi_rel_exp
exists_exp	[**NOT**] **EXISTS** (subquery)
arithmetic_exp	arithmetic | (subquery)
string_exp	string | (subquery)
entity_exp	variable | input_param | single_rel_exp
enum_exp	enum | (subquery)
datetime_ exp	datetime | (subquery)
boolean_exp	boolean | (subquery)
arithmetic	arithmetic_term | {arithmetic { ***** | **/** | **+** | **-** } arithmetic}
arithmetic_term	state_field_exp | literal | input_param | aggregate_exp | numeric_function | (arithmetic)
string	state_field_exp | literal | input_param | aggregate_exp | string_function
enum	state_field_exp | literal | input_param
datetime	state_field_exp | input_param | aggregate_exp | datetime_function
boolean	state_field_exp | literal | input_param

Table 4. Simplified JP QL Grammar

Java Persistence Query Language, continued

Clause/Term	Syntax
string_function	**CONCAT** (string , string) | **SUBSTRING** (string , arithmetic , arithmetic) | **TRIM** ([[{**LEADING** | **TRAILING** | **BOTH**}] [trim_char] **FROM**] string) | **LOWER** (string) | **UPPER** (string)
datetime_function	**CURRENT_DATE** | **CURRENT_TIME** | **CURRENT_TIMESTAMP**
numeric_function	**LENGTH** (string) | **LOCATE** (string , string [, arithmetic]) | **ABS** (arithmetic) | **SQRT** (arithmetic) | **MOD** (arithmetic , arithmetic) | **SIZE** (multi_rel_ exp)
subquery	**SELECT** [**DISTINCT**] {variable | single_rel_exp | aggregate_exp} **FROM** subquery_decl {, subquery_decl}* [where_clause]
subquery_decl	variable_decl | {single_rel_exp [**AS**] variable} | in_decl
update_clause	**UPDATE** entityName [[**AS**] variable] **SET** update_item {,{update_item}}*
update_item	{state_field_exp | single_rel_exp} = new_value
new_value	variable | input_param | arithmetic | string | boolean | datetime | enum | **NULL**
delete_clause	**DELETE FROM** entityName [[**AS**] variable]
groupby_clause	**GROUP BY** groupby_item {, groupby_item}*
groupby_item	single_rel_exp | variable
having_clause	**HAVING** conditional_exp
orderby_clause	**ORDER** BY orderby_item {, orderby_item}*
orderby_item	state_field_exp [ASC | DESC]

Table 4. Simplified JP QL Grammar, continued

Hot Tip JP QL queries can return data projections over entity attributes, averting instantiation of the actual entity objects

CONFIGURATION

Without counting the mappings from the entity to the database tables, there is really only one unit of JPA configuration needed to get your application up and running. It is based on the notion of a persistence unit, and is configured in a file called `persistence.xml`, which must always be placed in the `META-INF` directory of your deployment unit. Each persistence unit is a configuration closure over the settings necessary to run in the relevant environment. The parent element in a `persistence.xml` file is the `persistence` element and may contain one or more `persistence-unit` elements representing different execution configurations. Each one must be named using the mandatory persistence-unit `name` attribute.

There are slightly different requirements for configuring the persistence unit, depending upon whether you are deploying to a managed container environment or a non-managed one. In a managed container the target database is indicated through the `jta-data-source` element, which is the JNDI name for the managed data source describing where the entity state is stored

Configuration, continued

for that configuration unit. In a non-managed environment the target database is typically specified through the use of vendor-specific properties that describe the JDBC driver and connection properties to use. Also, in non-managed environments the entity classes must be enumerated in `class` elements, whereas in managed containers the entity classes will be automatically detected. Examples of container and non-container persistence unit elements are indicated in Listings 10 and 11, respectively.

Listing 10 – Container persistence unit configuration

```
<persistence-unit name="PetShop">
   <jta-data-source>jdbc/PetShopDB</jta-data-source>
</persistence-unit>
```

Listing 11 – Non-container persistence unit configuration

```
<persistence-unit name="PetShop">
   <class>com.acme.petshop.Pet</class>
   ...
   <class>com.acme.petshop.Owner</class>
   <properties>
      <property name="eclipselink.jdbc.driver"
        value="oracle.jdbc.OracleDriver"/>
      <property name="eclipselink.jdbc.url"
value="jdbc:oracle:thin:@localhost:1521:XE"/>
      <property name="eclipselink.jdbc.user"
value="scott"/>
      <property name="eclipselink.jdbc.password"
value="tiger"/>
   </properties>
</persistence-unit>
```

Hot Tip A provider implementation will be found by default, so avoid using the provider element and binding yourself to a specific provider unless you really are dependent upon that provider.

A hierarchical view of the possible XML elements in a persistence.xml file are shown in Figure 1. All of the elements are optional and the starred elements may be pluralized.

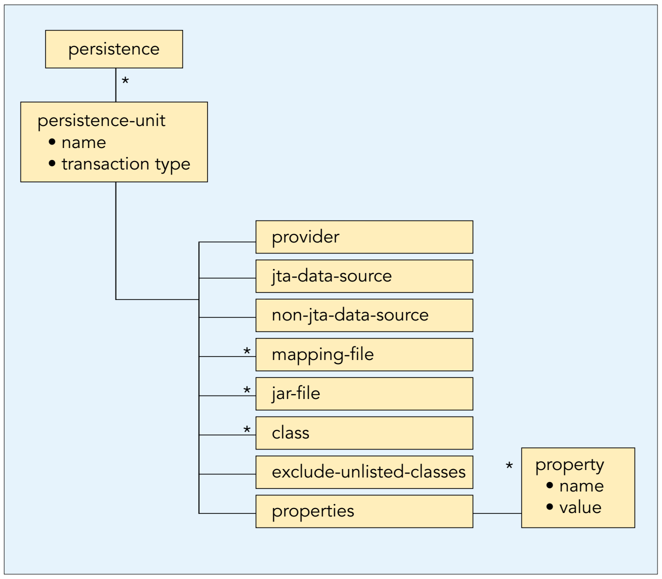


Figure 1. XML elements in `persistence.xml` file

MOVING ON

As you may have noticed, using JPA is not that hard, and you win the big prizes of portability and compatibility going forward. Hopefully you are now feeling ready to cut your teeth on a JPA application of your very own. The next step is to download the open source JPA 1.0 Reference Implementation (TopLink Essentials) and start it up. It is available at https://glassfish.dev.java.net/downloads/persistence/JavaPersistence.html and is trivial to install and configure. Happy persisting!

RESOURCES

Resource	Source
Glassfish Persistence Page	https://glassfish.dev.java.net/javaee5/persistence/entity-persistence-support.html
Oracle Technology Network JPA resources	http://www.oracle.com/technology/products/ias/toplink/jpa/index.html
Eclipse JPA (part of Eclipse Persistence Services Project)	http://www.eclipse.org/eclipselink
Pro EJB 3: Java Persistence API	By Mike Keith and Merrick Schincariol Apress, 2006 books.dzone.com/books/java-persistence

ABOUT THE AUTHOR

Mike Keith

Mike Keith was a co-lead of the EJB 3.0 and JPA 1.0 specifications and co-authored the premier JPA reference book called *Pro EJB 3: Java Persistence API*. He has 18 years of teaching, research and development experience in object-oriented and distributed systems, specializing in object persistence. He currently works as an architect for Java and persistence strategies at Oracle and represents Oracle on the JPA 2.0 and Java EE 6 expert groups. He has authored a host of articles and papers and is a popular speaker at numerous conferences and events around the world.

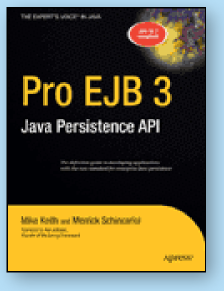
Publications

- Pro EJB 3: Java Persistence API, Mike Keith and Merrick Schincariol, Apress, May 2006
- ACM Queue, May/June 2008, Exposing the ORM Cache
- JavaLobby, June 2008, Looking Forward to JPA 2.0–Part 2
- JavaLobby, April 2008, Looking Forward to JPA 2.0–Part 1

Current projects:

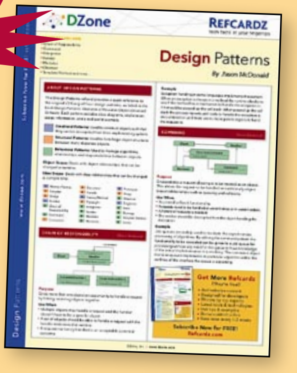
- Committer on Eclipse Persistence Project
- Member of JSR 317–JPA 2.0
- Member of JSR 316–Java EE 6
- Member of JSR 318–EJB 3.1
- Member of Java EE subcommittee of OSGi Alliance EEG

RECOMMENDED BOOK



Assuming a basic knowledge of Java, SQL and JDBC, this book will teach you the Java Persistence API from the ground up. After reading it, you will have an in-depth understanding of JPA and learn many tips for using it in your applications.

BUY NOW
books.dzone.com/books/java-persistence



DZone, Inc.
1251 NW Maynard
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-24-0
ISBN-10: 1-934238-24-4
50795
9 781934 238240
$7.95

Version 1.0