Getting Started with Versant JPA

Release 0.9

VERSANT

Getting Started with Versant JPA: Release 0.9

Versant JPA
Release 0.9

Copyright © 2012 Versant Corporation. All rights reserved.

The software described in this document is subject to change without notice. This document does not represent a commitment on the part of Versant. The software is furnished under a license agreement or nondisclosure agreement and may be used or copied only in accordance with the terms of the agreement. It is against the law to copy the software on any medium except as specifically allowed in the license or nondisclosure agreement. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, or for any purpose without the express written permission of Versant.

Versant, Versant Object Database and FastObjects are either registered trademarks or trademarks of Versant Corporation in the United States and/or other countries.

Java and all Java-based marks are trademarks or registered trademarks of Oracle Corporation in the United States and other countries.

Eclipse and Built on Eclipse are trademarks of Eclipse Foundation, Inc.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States and/or other countries.

All other products are a registered trademark or trademark of their respective company in the United States and/or other countries.

2012-V21-2

Table of Contents

Contact Information for Versant vii
1. Versant JPA Introduction 1
2. A V/JPA Application Step-by-Step 3
2.1. The Database Connection 4
2.1.1. The Persistence Unit Configuration File 4
2.1.2. Create a Database 5
2.2. Let's Get Started 6
2.2.1. A Simple Entity Class 6
2.2.2. Step 1—Persisting a Single Object 7
2.2.3. Step 2—Querying for Entity Objects 9
2.2.4. Step 3—Removing Entity Objects 10
2.3. Moving On 11
2.3.1. Step 4—Collections and Cascading Persistence 12
2.3.2. Step 5—Book by Name 13
2.3.3. Step 6—Book by Author 14
2.4. More Queries 15
2.4.1. Step 7—Predicate “Wildcards” 15
2.4.2. Step 8—Predicate Parameters 16
2.5. Automatic Change Tracking 17
2.5.1. Step 9—Modifying Objects 17
2.6. Detached Objects 18
2.6.1. Step 10—Merge 19
3. V/JPA Programming 23
3.1. Persistence Unit Configuration File Settings 23
3.2. Entity Names in Versant JPA 33
3.3. Enhancing Your V/JPA Entity Classes 34
3.3.1. The Enhancement Process 34
3.3.2. Run-time Enhancement 36
3.3.3. Build-time Enhancement—The Enhancer Ant Task 37
3.3.4. Build-time Enhancement—The Java Executable Enhancer 40
3.4. Versant Administration API 41
3.5. Distributed Databases 42
3.6. Entity Identity and the V/JPA Datastore 45
3.7. Collection Optimization 47
3.8. V/JPA Indexes 49
3.9. Cursor Queries 54
3.10. V/JPA Generic API 55
A. Versant JPA and the JPA Specification 59
A.1. Versant JPA Implementations 60

A.1.1. Primary Keys and Object Identifiers 60
A.1.2. Mapping 60
A.1.3. Normalization 61
A.1.4. Flush Operations 61
A.1.5. Entity Class Enhancement 62
A.1.6. Cascading Persistence 62
A.1.7. Query Support 62
A.1.8. Object Cloning 63
A.1.9. Entity Lifecycle Listeners 63
A.1.10. Pessimistic Locking 64
A.2. Versant JPA—Currently Supported Features 64
A.2.1. JPA Features 64
A.2.2. JPQL Support 65
A.3. JPA Public API Support 68
A.3.1. Package `javax.persistence` 68
A.3.1.1. Primary Persistence Interfaces 68
A.3.1.2. Query-related Interfaces 71
A.3.2. Package `javax.persistence.spi` 74
A.3.3. Package `javax.persistence.criteria` 75
A.3.4. Package `javax.persistence.metamodel` 75
A.3.5. Annotations 75
A.3.5.1. V/JPA Vendor-specific Annotations 75
A.3.5.2. JPA Annotation Support 76

List of Tables

3.1. V/JPA Enhancer ant Task Attributes & Elements 38
3.2. V/JPA Enhancer Class `EnhancerMain` Agruments 40
3.3. @EqualsById—Supported Classes and Methods 48
A.1. JPQL—Supported Expressions in V/JPA 66
A.2. JPQL—Unsupported Expressions in V/JPA 67
A.3. Interface `EntityManager` 68
A.4. Interface `EntityManagerFactory` 70
A.5. Interface `EntityTransaction` 70
A.6. Interface `Cache` 70
A.7. Interface `PersistenceUnitUtil` 71
A.8. Interface `PersistenceUtil` 71
A.9. Class `Persistence` 71
A.10. Interface `Query` 71
A.11. Interface `TypedQuery<X>` 73
A.12. Interface `Parameter<T>` 73
A.13. Interface `Tuple` 74
A.14. Interface `TupleElement<X>` 74
A.15. Interface Summary—`javax.persistence.spi` 74
A.16. Class Summary—`javax.persistence.spi` 75
A.17. Annotations 76

Contact Information for Versant

Versant Email Addresses

Versant JPA Support

If you have questions or need help using V/JPA, you can get quick answers. Contact the development staff at:

<mailto:jpa@versant.com>

About Versant

For general inquiries about Versant products and services, you can contact Versant at:

<mailto:info@versant.com>

Versant Office Locations

Versant Corporation is headquartered in Redwood City, California. Versant GmbH is responsible for European operations and has headquarters in Hamburg, Germany.

Versant Corporation

255 Shoreline Drive, Suite 450
Redwood City, CA 94065
USA

1-800-VERSANT
+1.650.232.2400
+1.650.232.2401 (FAX)

Versant GmbH

Halenreie 42
D-22359 Hamburg
Germany

+49 (0)40 609 90 0
+49 (0)40 609 90 113 (FAX)

Versant Web Sites

For the latest corporate and product news, please visit the Versant web site at http://www.versant.com/.

—

Please send feedback regarding this guide to:

`<mailto:documentation@versant.com>`

Chapter 1. Versant JPA Introduction

This technical preview provides you with a first look at Versant's Java Persistence API (JPA) database solution, Versant JPA. If you already have some experience with JPA you should first read the section detailing the differences between the JPA specification and the Versant JPA implementation. V/JPA follows the specification quite closely but you will want to be aware of the differences and the areas that have yet to be fully implemented. All of this is described in Appendix A, *Versant JPA and the JPA Specification* [p. 59]. (For you JPA newbies, you may want to refer to that section when you are a bit more familiar with JPA.)

To get your feet wet right away using V/JPA, the step-by-step getting started tutorial will guide you through your first real Versant JPA application. In addition to the getting started tutorial, you will find other examples in the `sdk/examples/jpa` directory in your V/JPA installation. These include a simple “Hello World” example (`sdk/examples/jpa/jpa_hello_world`; running this is a good way to see that Versant JPA is correctly installed), an example demonstrating attibute indexing in V/JPA (`sdk/examples/jpa/jpa_indexing`) and another showing the V/JPA generic API (`sdk/examples/jpa/jpa_generic_api`) and a number of others. Just follow the directions in the readme file that accompanies each example.

For additional programming infomation, including discussions regarding indexing and the generic API interface mentioned above, refer to Chapter 3, *V/JPA Programming* [p. 23]. Also, you will find the Javadoc API reference for Versant JPA, including the complete JPA standard API, in `doc/en_US/jpa/javadoc/index.html` in your V/JPA installation.

Chapter 2. A V/JPA Application Step-by-Step

Sections

- *The Database Connection*
  - *The Persistence Unit Configuration File*
  - *Create a Database*
- *Let's Get Started*
  - *A Simple Entity Class*
  - *Step 1—Persisting a Single Object*
  - *Step 2—Querying for Entity Objects*
  - *Step 3—Removing Entity Objects*
- *Moving On*
  - *Step 4—Collections and Cascading Persistence*
  - *Step 5—Book by Name*
  - *Step 6—Book by Author*
- *More Queries*
  - *Step 7—Predicate “Wildcards”*
  - *Step 8—Predicate Parameters*
- *Automatic Change Tracking*
  - *Step 9—Modifying Objects*
- *Detached Objects*
  - *Step 10—Merge*

This tutorial runs in the Eclipse IDE. You'll need to have Eclipse version 3.7 (or later) installed. You can get the Eclipse IDE at www.eclipse.org/downloads/.

Now, just a couple of things before you begin. You'll need to install the Versant JPA plugin and import the tutorial project into Eclipse.

Install the Versant JPA plugin for Eclipse

To install the Versant JPA plugin, perform the following steps.

- Select **Help** in the Eclipse menu bar and choose **Install New Software...**
- Select **Add...** and then **Archive...** and enter `Versant JPA` as name.
- Navigate to your V/JPA installation folder and select the file `sdk/IDEIntegrations/eclipse/com.versant.jpa.eclipse.zip`

- Select `Versant JPA` from the **Work with** pulldown menu
- Choose the `Versant JPA SE Plugin` and click **Finish**
- Proceed with the default installation process and restart Eclipse

Import the tutorial project into your Eclipse workspace

To import the tutorial project, from the Eclipse **File** menu select **Import...**. In the **Import** dialog, under **General**, select **Existing Projects into Workspace** and click **Next>**. The tutorial project is in the directory `sdk/examples/jpa/jpa_tutorial` where you installed V/JPA. Supply the location of this directory in the **Select root directory:** text field. Click **Finish**.

2.1. The Database Connection

To run the tutorial, you will need a V/JPA database. In the JPA persistence unit for this tutorial, `jpa_tutorial_persistence_unit`, the database name is specified as `jpa_tutorial` with the V/JPA server running on `localhost`. In this section we'll take a brief look at the persistence unit configuration file. You'll then see how to create the database using a tutorial utility application (which, in turn, uses the V/JPA `ServerAdministration` API).

2.1.1. The Persistence Unit Configuration File

In a Java Persistence API (JPA) application a *persistence unit* defines various aspects of the configuration including, for Versant JPA, the database connection. The persistence unit is defined in an XML file named `persistence.xml`. This file is in the `META_INF` directory in the tutorial project, that is, `META_INF/persistence.xml`.

Each persistence unit has a name that is unique within the context in which it is used. For this tutorial, the name is `jpa_tutorial_persistence_unit`. Here is the `persistence.xml` file you'll be using.

persistence.xml

```
<persistence version="2.0">

   <persistence-unit name="jpa_tutorial_persistence_unit" transaction-type="RESOURCE_LOCAL">

       <class>com.versant.jpa.tutorial.model.Person</class>

       <class>com.versant.jpa.tutorial.model.Book</class>

       <properties>
```

```
            <property name="versant.connectionURL" value="jpa_tutorial@localhost" />

        </properties>

    </persistence-unit>

</persistence>
```

Notice in the `<properties>` element the property `versant.connectionURL`. This locates the V/JPA database used for the persistence unit. The value is set to `jpa_tutorial@localhost`. This indicates a database named `jpa_tutorial` and the V/JPA server running on `localhost`, i.e., your local machine. This assumes you have the V/JPA server installed on the same machine you are using to run this tutorial which is typically the case. If you have a different V/JPA installation you can edit this file to reflect it.

> The persistence unit also lists the persistence capable classes, the *entity classes*, that make up your application's data model. These are the `<class>` elements you can see in the `persistence.xml` file, above. The instances of these classes will be stored in the database. You'll learn about entity classes right after you create the tutorial database.

2.1.2. Create a Database

The tutorial includes a number of utility applications in addition to the applications for each step. One such utility application is **Utility_CreateTutorialDatabase**. As the name implies, you can use this utility to create the database you'll use for the tutorial steps. To run the **Utility_CreateTutorialDatabase** application, right click on the class in the Package Explorer tree. From the context menu select **Run As > Java Application**.

> **Utility_CreateTutorialDatabase** uses the `ServerAdministration` API. You can take a look at the utility class to see how it is used. And you can get more info about the `ServerAdministration` class in the V/JPA javadoc reference.
>
> You can also create a V/JPA database using the command line. Refer to the utilities **makedb** and **createdb** in the *Administration Manual*.

Once you have created the database, you can start right in with the tutorial steps. Let's Get Started!

2.2. Let's Get Started

2.2.1. A Simple Entity Class

In JPA a persistence capable class is called an *entity class* (a persistent instance of an entity class is an *entity*). Let's look at a simple entity class. In your project, this is the `Person.java` class file in the `com.versant.jpa.tutorial.model` package.

JPA Entity Class—Person

```
/* Copyright (C) 2012 Versant Inc.  http://www.versant.com */

package com.versant.jpa.tutorial.model;

import javax.persistence.*;

@Entity
public class Person {

    @Id
    private long id;

    private String firstName;

    private String lastName;

    private Person() {

    }

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public String toString() {
        return firstName + " " + lastName;
    }

}
```

This defines a simple `Person` class. The important things to notice are the `@Entity` class annotation and the `@Id` field annotation. The `@Entity` class annotation tells the persistence engine that the class is an entity class, a persistence capable class. The `@Id` annotation designates the field (of type `long`) to be used for the

entity object identity in the datastore. That's really all you need to make this an entity class using Versant JPA. Since V/JPA stores entity class instances *as objects*, no additional annotations specifying mapping information are needed.

> Actually, there is one other thing that an entity class needs and that's a no-argument constructor. The access mode is not important, it can be private. It is good practice to implement this yourself rather than rely on the Java compiler to create it.

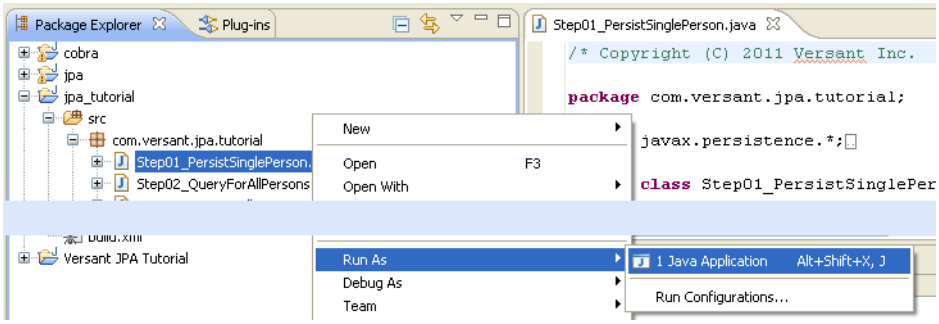
The Persistence Unit

As you learned above, a JPA application uses configuration information as defined in a persistence unit. Part of this configuration is a list of the entity classes you have defined for your application. Each entity class has a `<class>` element in the XML persistence unit file that gives the fully qualified name of the class. Here is the `<class>` element for the `Person` entity class.

```
<class>com.versant.jpa.tutorial.model.Person</class>
```

2.2.2. Step 1—Persisting a Single Object

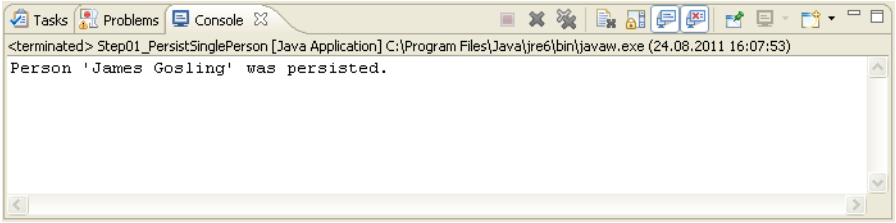
The first application will store a single `Person` object in the database. To run the application, right-click on the Java file `Step01_PersistSinglePerson` and select **Run As > Java Application**.



> **Launching Applications**
>
> To launch any of the tutorial applications, just right click the Java source file in the editor or **Package Explorer** tree and select **Run As > Java Application** from the context menu as you did for Step 1, above. The Versant JPA plugin will automatically add the required VM arguments to the run configuration.

> The project needs to have the **JPA Nature** configured. The tutorial project already comes pre-configured with this nature. To enable this nature for any other Java project, right click on the project name in the **Package Explorer** tree and choose **Versant JPA Nature > Add JPA Nature**. You can remove the nature from any project through the same context menu.

The Step 1 application connects to the `jpa_tutorial` database on your local V/JPA server. Then, within a *transaction*, it creates and persists a single `Person` object. Here is the output in the Eclipse console window.



What you should know...

With JPA, the entity class objects in your application are managed by an *entity manager*. An *entity manager factory* is responsible for the entity managers (there can be more than one). It is the factory that is associated with a persistence unit and, in turn, a V/JPA database. The first thing, then, is to create a factory and tell it which persistence unit to use for its configuration. (Remember, it is the persistence unit that defines the database used and the entity classes.)

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("jpa_tutorial_persistence_unit");
```

Here, the factory will use the `jpa_tutorial_persistence_unit` persistence unit which is defined in the `src/Meta-INF/persistence.xml` file—and connects to the `jpa_tutorial` database that you created earlier.

The factory can now be called on to provide you with an entity manager.

```
EntityManager em = emf.createEntityManager();
```

All modifications on persistent objects, including creating new ones, are made in a *transaction*. A transaction is a unit of work performed by the client, your application, and the database server. The entity manager is responsible for the transaction. Access to the transaction is with the manager's `getTransaction()` method. You mark the begin and end points of the transaction with the methods `begin()` and `commit()`.

```
em.getTransaction().begin();
// do stuff with persistent objects . . .
em.getTransaction().commit();
```

Any changes to the entity objects in the transaction are written to the database (committed) when you call the transaction `commit()` method. Transactions also give you the opportunity to say “nevermind”. Instead of `commit()`, you can call the transaction `rollback()` method to undo any changes you made.

When you create a new instance of an entity class, the entity manager doesn't know if the instance should be persistent or not. So, you have to tell the entity manager that you intend that the instance be persisted in the database by calling the entity manager `persist()` method for the new object instance.

```
Person person = new Person("James", "Gosling");
em.persist(person);
```

You only need to call `persist()` once. The next time you work with that object, any changes will be written automatically when the transaction commits.

One more thing you should know...

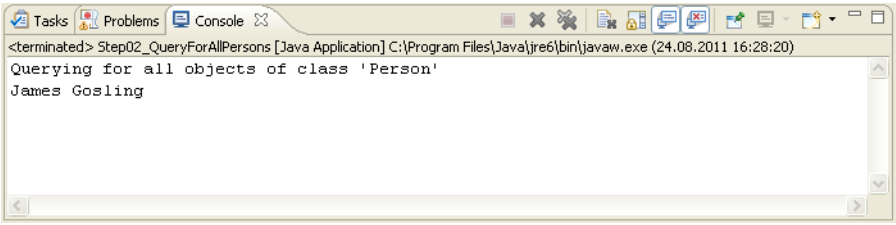
Entity classes need to be *enhanced* so that methods can be introduced to track modifications of fields and properties. The tutorial uses *run-time enhancement*. The Java agent that performs run-time enhancement is pre-configured for the tutorial project. (This is a part of the JPA Nature mentioned earlier in the Launching Applications [p. 7] tip.)

You can learn more about Versant JPA enhancement in Section 3.3, “Enhancing Your V/JPA Entity Classes” [p. 34].

2.2.3. Step 2—Querying for Entity Objects

Storing objects is of no value if you can't find them again. The next example will use a query to find all of the objects of a given class. In this case all `Person` objects. Run the application **Step02_QueryForAllPersons**. (Right-click the `Step02_QueryForAllPersons` Java source file and select **Run As > Java Application** as you did in the previous step.)

The console prints this.



What you should know...

The query is performed by asking the entity manager for a `Query` instance. The desired query criteria, the predicate, is provided to the manager's `createQuery()` method.

```
Query query = em.createQuery("select p from Person p");
```

The query predicate is expressed in JPQL (Java Persistence Query Language) and simply selects all members from the entity class `Person`.

The query is sent to the V/JPA server and the result collection is returned in the next line.

```
List<Person> resultList = query.getResultList();
```

A loop is used to print the results.

```
for (Person person : resultList) {
    System.out.println(person);
}
```

2.2.4. Step 3—Removing Entity Objects

In an application with non-persistent objects, those objects are gone when the application ends. The point of entities and persistence is that the objects, in the last saved state, are available again the next time the application is run. This implies that removing an object from persistent storage requires an explicit request.

Run the **Step03_RemoveAllPersons** application. (Right-click the `Step03_RemoveAllPersons` file; select **Run As > Java Application**.) This will remove the `Person` object you persisted in the first step and found and read in the last step.

You can re-run Step 2 to verify that the object is no longer in the database.

Removing all objects

A utility application, **Utility_RemoveAll**, is provided in the tutorial project. As the name implies, running it removes all tutorial objects in the database. You can use it at any time to remove all of objects created by the tutorial steps. Run it in the usual way using the **Run As > Java Application** on the `Utility_RemoveAll` Java source file's context menu.

What you should know...

If you compare the code for this application with the code for the previous, **Step02_QueryForAllPersons**, application, you'll see that it is almost identical. The same query is used to return a collection, a `List`, of the `Person` entity objects. This time, however, the loop is used to call the entity manager's `remove()` method for each `Person` instance.

```
for (Person person : resultList) {
    em.remove(person);
}
```

2.3. Moving On

We'll continue with some more complex examples. We'll also add to the entity class data model. Open the file `Book.java` in the `com.versant.jpa.tutorial.model` package.

Just like in the `Person` entity class, the `Book` is annotated as an entity, a persistence capable class, with the `@Enity` annotation. A field, `private long id`, carries the `@Id` annotation. And a no-arguments constructor is also defined. Great, we have an entity class!

A collection field and cascading persistence

A book can have more than one author, so, for the `authors` field, we use a collection.

```
@OneToMany(cascade=CascadeType.PERSIST)
private Collection<Person> authors;
```

Remember from the first step that a new entity class instance must be told to persist. (With the entity manager `persist()` method.) When you create a new persistent `Book` instance, you have to call `persist(`*`book`*`)`. And, of course, you want any newly created `Person` instances that are the book's authors to be persistent as well.

JPA provides a language element, an annotation, `@OneToMany` that marks the collection as having a one-to-many relationship from `Book` to `Person` (authors). This is extended with the `cascade` element, specifically `CascadeType.PERSIST`, indicating that a persist operation applied to the `Book` instance should also cascade, that is, be applied to, the collection elements. Make a `Book` instance persistent and the `Person` instances in the `authors` collection are made persistent as well. This cascading persistence is also referred to as *persistence-by-reachability*.

OK, now you're ready to create and “read” some books.

2.3.1. Step 4—Collections and Cascading Persistence

Run the **Step04_PersistBooks** application. (Right-click the `Step04_PersistBooks` file; select **Run As > Java Application**.) This creates three `Book` objects in the database, each with multiple authors (`Person` objects). To see them, run the utilty application **Utility_PrintAllBooks**. Here is the console output.

```
Printing all books:
--------------------------------------
Book 'Java Concurrency in Practice'
Authors:
  Brian Goetz
  Tim Peierls
  Joshua Bloch
  Joseph Bowbeer
  David Holmes
  Doug Lea
Book Price:
0.0
--------------------------------------
Book 'Programming in Scala'
Authors:
  Martin Odersky
  Lex Spoon
  Bill Venners
Book Price:
0.0
--------------------------------------
Book 'The Pragmatic Programmer'
Authors:
  Andrew Hunt
  David Thomas
Book Price:
0.0
```

(Don't worry that the books are free. You'll update the prices in a later step.)

What you should know...

In the code there are methods for creating each new book. The method also creates the `Person` instances to fill the `authors` collection. Here, for example, is the method for *The Pragmatic Programmer*.

```
private static Book thePragmaticProgrammer(){
    Book book = new Book("The Pragmatic Programmer");
    book.addAuthor(new Person("Andrew", "Hunt"));
    book.addAuthor(new Person("David", "Thomas"));
    return book;
}
```

Notice that nowhere in the Step 4 code is an explicit call to `persist()` made *for a `Person` object*. However, by using cascading persistence (`cascade=CascadeType.PERSIST`) the book's authors are made to persist as well.

In the next two steps you'll see how to retrieve the `Book` object you want based on the book's name or author.

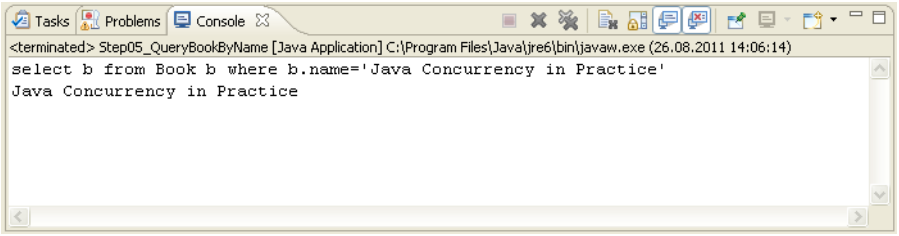
Something else you should know...

The utility application, **Utility_PrintAllBooks**, prints the name of the books and their authors to the console. You saw the output earlier in this step. And you have already seen how you can get all of the `Book` instances. Just use the same type of query you used to get all of the `Person` objects back in Step 2. And once you have a `Book` instance, you can follow the references in the `authors` list to get the `Person` instances.

With a relational datastore, retrieving a `Person` instance is performed, under-the-hood, by another query. Not so with V/JPA where resolving a reference is a simple request for that specific object. This will mean a considerable performance boost for any application that navigates object networks, resolving references, to retrieve objects.

2.3.2. Step 5—Book by Name

In this step you'll use a query to retrieve a specific book from the database by specifying the book's name. Run the **Step05_BookByName** application. The application displays the JPQL query predicate and the name field of the `Book` object found by the query.



What you should know...

In Step 2 you used a query that returned all objects of the specified class. Here is the query predicate from that earlier step.

```
select p from Person p
```

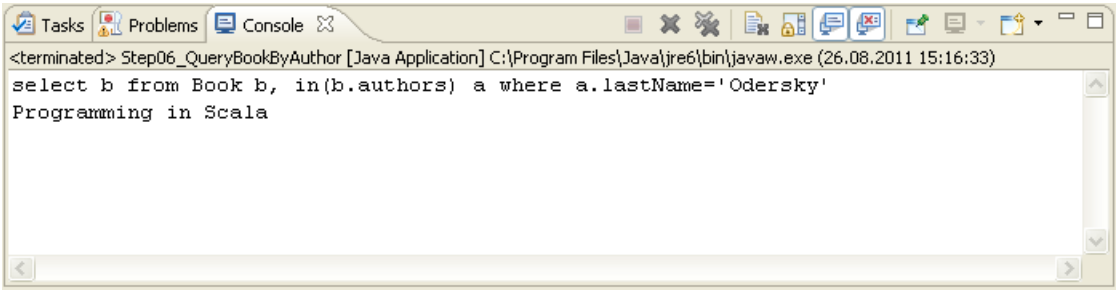
In the new query, the search criteria is narrowed with a *where* clause. In this case, *where* the *name* field of the `Book` matches a specific value.

```
select b from Book b where b.name='Java Concurrency in Practice'
```

2.3.3. Step 6—Book by Author

This step uses a query to retrieve any `Book` objects that have a specified person in their list of authors. Run **Step06_BookByAuthor**.

As in the previous step, the application prints the query predicate and the result, the book's name, to the console.



What you should know...

You can see that the query predicate is more complex.

```
select b from Book b, in(b.authors) a where a.lastName='Odersky'
```

But some pieces should already be familiar. The first part is the *select* clause.

```
select b from Book b
```

By itself, it would select all books. (Remember the query predicate in Step 2.)

The next part is an *in* clause and is new.

```
in(b.authors) a
```

It specifies the referenced part of the book's data to examine. In this case, each `Person` object in the book's list of authors.

Finally, the *where* clause specifies the search criterion, namely, the author's last name.

```
where a.lastName='Odersky'
```

2.4. More Queries

This section presents a couple of additional query constructs: “wildcard” queries and parameterized queries. These steps use the objects you created and stored in Step 4. If you've already deleted them, run Step 4 again.

Entity Retrieval Without a Query

Just so you know, even though queries are important for retrieving the objects you want from the database, they're not the only way. You read earlier that object navigation in V/JPA doesn't use a query but retrieves the entity object directly (Step 4, Something else you should know...).

You can also directly retrieve an entity object using the entity's ID. The ID is the value of the field marked with `@Id` annotation. The V/JPA implementation of the `EntityManger.find()` method uses the direct access mechanism provided by Versant JPA. Here is a short example.

```
em.getTransaction().begin();
Book b =  em.find(Book.class, 205476732998809741);
// do something with the entity . . .
em.getTransaction().commit();
```

A couple of things to keep in mind if you want to use `find()` and direct access.

- The ID of an entity in V/JPA is always type `long`
- You cannot set the ID yourself, V/JPA does that
- The ID assigned to an entity does not change for the life of the object in the database
- In order to get the ID of an entity object, you will need to make the ID field public or implement a “getter” method

2.4.1. Step 7—Predicate “Wildcards”

Run the **Step07_QueryBookByNameLike** application.

Here is the console output.

```
Tasks  Problems  Console
<terminated> Step07_QueryBookByNameLike [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (29.08.2011 15:01:49)
select b from Book b where b.name like '%Program%'
Programming in Scala
The Pragmatic Programmer
```

What you should know...

In the output you see that two books were found. A quick look at the query predicate shows why.

```
select b from Book b where b.name like '%Program%'
```

The query uses the `like` operator to specify a pattern search. It will find any `Book` object with a name that contains the string “Program”. The wildcard character % means any number of characters, including none. So, the “Program” string can have any (or no) characters before it or after it and the title will still match the search criterion.

The JPQL query language defines an additional wildcard, _ (underscore), that means exactly one character. Try changing the query predicate in `Step07_QueryBookByNameLike.java` and rerunning the query to see the effect of different wildcard patterns.

2.4.2. Step 8—Predicate Parameters

So far, all the queries we've seen used a fixed query string. In other words, the string is hard-coded in the application. But in many instances you want to execute a query based on criteria you may not have until run-time, say, for example, a value from the user. This is where a *parameterized* query predicate comes in handy. You create a query string where the value of a criterion is substituted by a parameter that you can fill in when you run the query.

Run the **Step08_QueryBookByNameParameterized** application.

The console ouput looks like this.

```
Tasks  Problems  Console
<terminated> Step08_QueryBookByNameParameterized [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (29.08.2011 17:48:32)
select b from Book b where b.name = :nameParam
nameParam: Java Concurrency in Practice
Java Concurrency in Practice
nameParam: Programming in Scala
Programming in Scala
```

What you should know...

Let's examine the query predicate.

```
select b from Book b where b.name = :nameParam
```

Notice that instead of a fixed value for the book's name the predicate has a place holder defined using the `:replaceableParameter` syntax (that's a `:` character followed immediately by the parameter name you choose). In the predicate above, it's `:nameParam` and that will be filled in before the query is executed.

To give the parameter a value, you call the `Query` class method `setParameter()` with two arguments; the parameter name (without the `:`) and the value. Here is one of the calls in the `Step08_QueryBookByNameParameterized.java` file.

```
query.setParameter("nameParam", "Java Concurrency in Practice");
```

The query will run as if you specified the predicate like this.

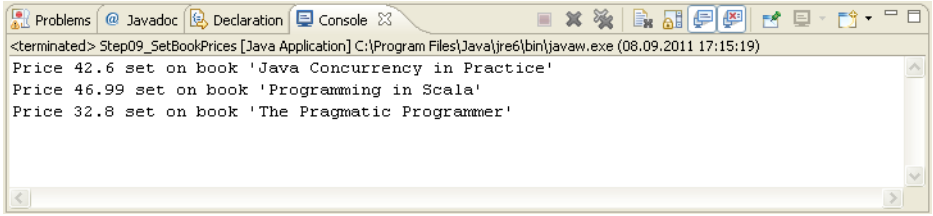
```
select b from Book b where b.name = 'Java Concurrency in Practice'
```

2.5. Automatic Change Tracking

You've now seen how to create new persistent objects and how to use a query to locate an object and bring it back into a transaction. You've also seen how to remove a persistent object from the database. Now it's time to see how modifications to persistent objects are handled.

2.5.1. Step 9—Modifying Objects

Run **Step09_SetBookPrices** to modify the `Book` objects you created in Step 4 by assigning prices to the books. Here is the console output.



What you should know...

In the `main()` routine you create an entity manager factory. The factory knows which database to use from the persistence unit you specify when creating the factory. You use the factory to get an entity manager which will provide a transaction in which to work with your persistent objects. So far, nothing new. Each of the tutorial applications begins with these operations.

No real work with persistent objects can be done until you begin the transaction. Once you do, your application and the V/JPA server are working together to track any changes you make to the objects you retrieve from the database. In this step, the method `setBookPrice()` is called with a book name and a price. In the method, a query is used to retrieve the `Book` object with the specified name and the book's price field is updated. (Up to now the books were free. When you created the `Book` instances back in Step 4, their price was `0.0`.)

When the `price` field in the `Book` is updated, that is, its value is changed, that update is recognized by the V/JPA entity manager and the object is marked as modified. When the transaction commit operation is called, any modified objects are automatically written to the database. That's automatic change tracking.

> If you make changes to objects and later decide not to commit the changes, you can end the transaction with the `rollback()` method. Nothing is written to the database. Using transaction `begin()/rollback()` is also appropriate for “read-only” transactions. Take a look at `Step02_QueryForAllPersons.java`, for example, or the code for any of the “Query” steps.

Of course, V/JPA is doing much more under-the-hood to insure that any modifications to the database are performed in a consistent manner. By default, V/JPA uses an optimistic concurrency control mechanism. This will be appropriate for many applications. Depending on your specific application needs, you can configure the type of concurrency control you use with the available JPA control mechanisms.

2.6. Detached Objects

In the previous step, you read that “No real work with persistent objects can be done until you begin the transaction.” Well, that is not strictly true. You can copy the state of a persistent object in an entity manager—*detach*—to work with it outside of a transaction and return the updated object to the datastore—*merge*—at a later time. In this step, you'll see how to detach objects and merge any changes you make back into persistent storage.

2.6.1. Step 10—Merge

Run the application **Step10_MergeBooksWithChangedPrices**. (This application needs the `Book` objects you created in Step 4 and modified in Step 9.)

Here is the console output for the **Step10_MergeBooksWithChangedPrices** run.

```
Problems  Javadoc  Declaration  Console
<terminated> Step10_MergeBooksWithChangedPrices [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (12.09.2011 16:16:21)
1) Retrieve the books and detach them.
2) Raise the book price of all books by 1
3) Merge the updated books.
4) Retrieve the books and list them.
Java Concurrency in Practice, price: 45.6
Programming in Scala, price: 49.99
The Pragmatic Programmer, price: 35.8
```

The application actually performs four operations. The first uses a query to find all of the `Book` objects, detach them from the entity manager, and return them as a collection. The second updates the prices of the books in the detached objects. In the third operation, the detached books, with their updated prices, are returned to an entity manager to be merged back into persistent storage. And finally, the newly updated and merged persistent `Book` objects are again retrieved from the database for printing to the console.

What you should know...

The first operation, detaching the objects, is performed by the method `detachBooks()`. An entity manager factory is called on to provide an entity manager and a transaction is started.

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT);
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
```

So far, nothing new.

A simple query is used to retrieve the `Book` objects in the database.

```
Query query = em.createQuery("select b from Book b");
List<Book> resultList = query.getResultList();
```

At this point the objects are the responsibility of the entity manager and automatic change tracking will insure that any changes made to the objects are written when you commit the transaction. (You saw this in Step 9.)

However, this time each book is detached from the entity manager (and transaction) and the detached objects are returned in a collection.

```
Collection<Book> books = new ArrayList<Book>();
. . .
for (Book book: resultList) {
    em.detach(book);
    books.add(book);
}
```

The detached books, in a collection, are returned to the main routine. Since no changes were made to the objects, the transaction is rolled back, not commited. It is important to understand, however, that if you had made changes to an object before detaching it, these changes would not have been written to the database with a commit. But any changes made prior to detaching an object are reflected in the detached version of that object.

Editing the detached objects

OK, now you have a collection of books in your main routine that are not associated with any entity manager. In the second operation, `raiseBookPrices()`, the prices of these books are increased by one. (Think of it as correction for inflation.) Here is the complete `raiseBookPrices()` method.

```
private static void raiseBookPrices(Collection<Book> books) {
    System.out.println("2) Raise the book price of all books by 1");
    for(Book book : books){
        book.setPrice(book.getPrice() + 1);
    }
}
```

The important thing to notice here is that no entity manager is created and no transaction is started. The detached objects are like any other transient object in your application. You could even copy them to a web-server or a smartphone for editing.

Merging the changes

But these represent persistent objects and, at some point, you will likely want to update their persistent counterparts in the database. That happens in the next operation, `mergeBooks()`. The collection of detached, and updated, books is passed to the routine. This time, you'll be updating the objects in the database and you'll need an entity manager and transaction. The detached objects are merged, one-by-one, in a simple loop.

```
for(Book book : books){
    em.merge(book);
}
```

A detached object maintains its identity in the database. When you call `merge()` the object is added to the objects being managed by the entity manager and is marked as modified. It is as if you had read the object from the database and modified it in the current transcation. Now, when you commit the transaction, the updated objects are written to the database, again, as if the changes had been made in that transaction.

But what if a detached object had been modified by some other activity in the meantime? V/JPA uses, by default, an optimistic concurrency stategy as mentioned in Step 9. If an object was updated in the database after the time you detached it, the merge will fail.

The last operation, `printBooks()`, simply lists the books, with their updated prices, to the console. It uses a query—in a transaction!—to retrieve the books once again.

Chapter 3. V/JPA Programming

Sections

- *Persistence Unit Configuration File Settings*
- *Entity Names in Versant JPA*
- *Enhancing Your V/JPA Entity Classes*
  - *The Enhancement Process*
  - *Run-time Enhancement*
  - *Build-time Enhancement—The Enhancer Ant Task*
  - *Build-time Enhancement—The Java Executable Enhancer*
- *Versant Administration API*
- *Distributed Databases*
- *Entity Identity and the V/JPA Datastore*
- *Collection Optimization*
- *V/JPA Indexes*
- *Cursor Queries*
- *V/JPA Generic API*

This section discusses additional features and programming support provided by Versant JPA.

API Reference—JPA and V/JPA

The complete Javadoc API reference for JPA and V/JPA is provided in your V/JPA installation. Open the file `doc/en_US/jpa/javadoc/index.html` in your browser.

3.1. Persistence Unit Configuration File Settings

The persistence unit configuration is defined in a file named `persistence.xml`. This section describes the currently available settings.

Versant JPA currently interprets only a sub-set of the possible settings in the persistence unit configuration file (`persistence.xml`). V/JPA will ignore settings that it doesn't understand, so long as they are syntactically correct.

The following shows a sample persistence unit configuration file as it might look for V/JPA.

persistence.xml

```
<persistence version="2.0">

    <persistence-unit name="jpa_tutorial_persistence_unit" transaction-type="RESOURCE_LOCAL">

        <class>com.versant.jpa.tutorial.model.Person</class>

        <class>com.versant.jpa.tutorial.model.Book</class>

        <properties>

            <property name="versant.connectionURL" value="jpa_tutorial@localhost" />

        </properties>

    </persistence-unit>

</persistence>
```

JPA Persistence Unit Settings—V/JPA Restrictions

- A `<class>` element must be specified, with fully-qualified package names, for every entity class.
- In the `<persistence-unit>` element, the attribute `transaction-type` is restricted to `RESOURCE_LOCAL`
- The vendor-specific property `versant.connectionURL` must be set and the specified database must exist. (Refer to `versant.connectionURL` [p. 25].)

Versant JPA Vendor-specific Properties

The following lists and describes the V/JPA vendor-specific properties.

`versant.connectionIdleTimeout`

If a connection is returned to the connection pool, all available connections (i.e., connections that are not currently in use) are examined to determine when they were last used, based on their timestamp, beginning

with the longest unused connection. If any of these available connections has not been used for longer than the specified time, it will be closed (default is 60 seconds).

Connections are closed until the number of remaining available connections is the value set for the `versant.coreConnectionPoolSize` property (or all connections older than the specified time are closed).

allowed values:

integer value (milliseconds)

default:

`60000`

`versant.connectionPoolSize`

A connection is taken from the pool for the duration of the database transaction. For example, you could create any number of entity managers without impact on the connection pool but only have active transactions on the number specified for the pool.

allowed values:

integer

default:

`10`

`versant.connectionURL`

The name of the database to connect to.

The URL should be of the form `versant:`*`database`*`@`*`host`*`:`*`port`*. The URL scheme `versant:` is optional. The parts of the URL are described below.

`database`

This is the name of the database. This is required.

`host`

This is the host name of the database server.

port

This is the port to be used to establish a connection to the database. This is optional. If not specified it defaults to `5019`, the port typically used by Versant JPA.

allowed values:

integer

default:

`5019`

`versant.coreConnectionPoolSize`

This property determines the minimum number of available connections to keep in the connection pool when removing connections that have not been used based on the value of `versant.connectionIdleTimeout`. The value of this property defaults to the value of `versant.connectionPoolSize`.

allowed values:

integer value between `1` and the value of `versant.connectionPoolSize`, inclusive

default:

value of `versant.connectionPoolSize`

`versant.defaultEntityNames`

The JPA default is to use short class names in, for example, query strings. You can change this behavior to use fully-qualified names (with package). Refer to Section 3.2, "Entity Names in Versant JPA" [p. 33] for more information about entity naming in V/JPA.

allowed values:

`shortClassName` (equivalent to JPA default)
`fullyQualifiedClassName`

default:

`shortClassName`

versant.defaultLoidBatchSize

The Versant JPA datastore uses a logical object identifier, LOID, to uniquely identify the entity object. The LOID value is also the entity primary key, the `@Id` field. Identifiers are allocated by the V/JPA datastore and supplied to the entity manager factory to be assigned to newly created entities. The datastore allocates the identifiers in batches of a specified size. The default batch size is 1024. This property allows you to adjust the batch size to better suit the requirements of your application. For example, if your application consistently creates very large numbers of new entities in a given transaction, that can lead to many datastore requests for new batches of identifiers. Adjusting the LOID batch size to a higher number may increase performance by reducing the amount of requests sent to the V/JPA server.

> Use care when adjusting the LOID batch size. Small values can result in excessive numbers of requests and consequent server traffic. Very large values can lead to depletion of available identifiers in the datastore.

The LOID batch size can also be adjusted at run time for a given entity manager factory. Refer to Section 3.6, “Entity Identity and the V/JPA Datastore” [p. 45] for more information about object identifiers and adjusting the LOID batch size.

allowed values:

positive integer >= `3`

default:

`1024`

versant.genericAccess

Specifies if access to the database entity objects using the V/JPA Generic API is allowed. Refer to Section 3.10, “V/JPA Generic API” [p. 55] for more information.

allowed values:

`true`, `false`

default:

`false`

`versant.indexesDefinitionFile`

Specifies the index definition file, if used, for the persistence unit. Refer to Section 3.8, “V/JPA Indexes” [p. 49] for more information.

allowed values:

valid file path

default:

none

`versant.inheritIndexes`

Specifies if index definitions are inherited by sub-classes of the class for which the index is defined. Refer to Section 3.8, “V/JPA Indexes” [p. 49] for more information.

allowed values:

`true`, `false`

default:

`true`

`versant.queryBatchSize`

When a standard query is executed, the V/JPA server first transfers only the identifiers of all entities in the result set to the client. The entities themselves (the field values) are *lazy-loaded* in batches, as they are required, when iterating over the result set. The default batch size. i.e., the number of entities loaded per fetch, is `10000`. You can use this property to adjust the fetched batch size to better suit the requirements of your application.

> The `versant.queryBatchSize` property does not apply to projection queries or cursor-queries which use different loading mechanisms. For a description of cursor queries, refer to Section 3.9, “Cursor Queries” [p. 54].

allowed values:

positive integer > `0`

default:

`10000`

versant.queryInterleaveResults

When you perform a standard query, the V/JPA server first transfers only the identifiers of the result set entities to the client. The result `List` collection initially only contains the identifiers and the entities themselves are lazy-loaded, in batches, as they are required. (The batch size is adjustable. Refer to `versant.queryBatchSize`.)

If you are using distributed databases (i.e., your application connects to multiple databases, refer to Section 3.5, "Distributed Databases" [p. 42]) the `versant.queryInterleaveResults` property determines whether the result set of a query is *interleaved* (setting is `true`) or *ordered* (`false`, the default) with respect to the corresponding databases.

When using interleaved results (`versant.queryInterleaveResults` set to `true`) all databases are queried concurrently when getting the next batch of entities while iterating over the result set. Using ordered results (the default) means that the databases are queried sequentially, one after the other.

Using interleaved results may have a small performance impact when getting the initial result set, but a performance improvement when loading the entities themselves, depending on the distribution of the entities across the databases.

> The `versant.queryInterleaveResults` property only applies to queries performed on multiple databases. The property does not apply to projection queries or cursor-queries which use different loading mechanisms. For a description of cursor queries, refer to Section 3.9, "Cursor Queries" [p. 54].

allowed values:

`true`, `false`

default:

`false`

versant.schemaDefine

This property determines whether new schema classes will be defined as needed at runtime by V/JPA. The default value is `true`.

Before the instances of a entity class can be stored in a V/JPA database, a schema for the class must first be defined in the database.

This schema definition process is executed at runtime when instantiating a V/JPA entity manager factory (by calling `Persistence.createEntityManagerFactory()`).

The schema definition process is not thread-safe. You must insure that multiple user threads do not try to define the schema for the same classes simultaneously (i.e., by synchronizing your `createEntityManagerFactory()` calls).

allowed values:

`true`, `false`

default:

`true`

`versant.schemaEvolve`

Determines whether datastore schema will be evolved if required. The default value for the flag is `false`, i.e., schema will not be evolved in the case of a mismatch.

allowed values:

`true`, `false`

default:

`false`

`versant.useSoftReferences`

Each entity manager maintains a map of oject identifiers to entities. This map is used as a cache to ensure that there is, at most, one entity instance for a given object identifier in the entity manager instance. By default, this map uses `java.lang.ref.WeakReference` instances so that the garbage collector can collect otherwise non-referenced objects. If `versant.useSoftReferences` is set to `true` the map will use `java.lang.ref.SoftReference` instances.

Regardless the setting of `versant.useSoftReferences`, the entity manager will hold strong references to all objects that have been inserted, updated or removed until they are flushed.

allowed values:

`true`, `false`

default:

`false`

`versant.strictXmlValidation`

Determines whether an optional, strict XML validation (setting is `true`) or a more relaxed validation (`false`, the default) is performed for the persistence unit file and referenced mapping files. Strict validation uses the JPA standard XML schema which defines specific ordering for the elements in the `persistence.xml` file. The relaxed schema is less strict regarding element order.

allowed values:

`true`, `false`

default:

`false`

In-memory XML Metadata

The following properties allow you to over-ride the metadata defined in the various XML metadata files with in-memory representations.

The XML metadata files are normally resolved in the classpath/file system based on the following rules.

- The `persistence.xml` file must this name and reside in the `/META-INF` directory in the class path
- Mapping files can be declared in the persistence.xml, providing a path relative to the persistence.xml's URL. In this case, it can have an arbitrary, user-defined name/path
- If no mapping file declarations are found, then default location /META-INF/orm.xml (side by side to the persistence.xml) is tried

However, there may be scenarios where you want to generate or extend this metadata information from your application prior to creating the EntityManagerFactory instance. You can pass in-memory representations of the persistence/mapping files along with the optional properties map passed into javax.persistence.Persistence.createEntityManagerFactory(String, Map) using the following properties.

`versant.persistence.xml`	`persistence.xml` override
`versant.orm.xml`	mapping file override
`versant.indexes.xml`	index definition file override

The values for these properties should be valid XML strings conforming to the corresponding schemata. These are described in more detail, below.

> The overriding process is subject to the same set of constraints as any changes to the mapping information applied after the bytecode enhancement step. In particular, no information that affects the enhanced bytecode may be different in the in-memory version than was defined in the original mapping information used for the enhancement step. Refer to Section 3.3.1, "Metadata Consistency" [p. 35] for more information.

XML Metadata Override Properties

`versant.indexes.xml`

If set, provides an in-memory string version containing the content of a Versant JPA index definition file used for creating an entity manager factory. Any index definition file declared in the `persistence.xml` will be ignored.

allowed values:

a well-formed XML string that conforms to the `http://versant.com/jpa/xml/indexes` schema

default:

none (`null`)

`versant.orm.xml`

If set, provides an in-memory string containing the content of a mapping file (orm.xml) used for creating an entity manager factory. Any mapping file declared in the `persistence.xml` or present in the default location in the class path/file system will be ignored. Note that the settings in this mapping file version must not conflict with the settings used when enhancing the class files.

allowed values:

a well-formed XML string that conforms to the `http://java.sun.com/xml/ns/persistence` schema

default:

none (`null`)

`versant.persistence.xml`

If set, provides an in-memory string containing the content of the persistence unit (persistence.xml) used for creating an entity manager factory. Any `persistence.xml` file on the class path/file system will be ignored. Note that the settings in this persistence.xml version must not conflict with the settings used when enhancing the class files. Declaring references to mapping/index files on the class path/file system is not supported.

allowed values:

a well-formed XML string that conforms to the `http://java.sun.com/xml/ns/persistence` schema

default:

none (`null`)

3.2. Entity Names in Versant JPA

Versant JPA uses the *entity name* associated with your Java class to identify the entity classes in the database. For example, when performing JPQL queries you specify the entity name in the query predicate.

In V/JPA there are three ways to determine the JPA entity name of a class.

- Use the unqualified name of the class. For example, for the class `com.company.datamodel.Book`, just the class name without the qualifying package information, `Book`, is used. This is the global default in V/JPA and follows the JPA specification.
- Use the fully qualified name, including the qualifying package information. For the example above the entity name is `com.company.datamodel.Book`. You can set this behavior in the persistence unit with the V/JPA property `versant.defaultEntityNames` using the value `fullyQualifiedClassName`.
- Use the name defined in the `@Entity` annotaton or in the `<name>` element in the `<entity>` element in the persistence unit (ORM.XML file).

  For example, the following shows setting the entity name of the `Book` class to `Paperback`.

  ```
  package com.company.datamodel;
  @Entity(name="Paperback")
  class Book { . . . }
  ```

This shows setting the entity name in an ORM file.

```
<entity class="com.company.datamodel.Book">
    <name>Paperback</name>
    . . .
</entity>
```

Once you have begun using a particular entity naming configuration, you should avoid changing it. Changing the entity names and connecting to an already populated V/JPA database can result in undesirable behavior. Simply renaming an entity class has the result that entities stored in the database with the previously defined name will not be found using the new name.

Renaming an entity class to a name that is already in use by another class will cause the entities of the newly renamed class to be handled as if they are entities of the previously existing class. If the two entity classes have different structures (different fields, field layout) this can result in application crashes and/or loss of data. Having two different entity classes sharing a common entity name could occur if, e.g., you are using more than one persistence unit or versions of a persistence unit and switch between them.

Another potential risk when defining your entity names in multiple persistence units is the same class having more than one name. In the database, these will be treated as different entity classes and only the name defined in the persistence unit you use at a given time will be seen.

3.3. Enhancing Your V/JPA Entity Classes

In order for Versant JPA to properly track modifications of fields and properties in your persistent objects, additional functionality must be added to the definitions of your entity classes. For this reason, the byte-code representations of your entity classes (the `.class` files) need to be *enhanced* to add the appropriate methods.

With V/JPA this enhancement can be done at run-time using a Java agent or as part of the build process for your application using an **ant** task or with a Java executable. The following sections describe the enhancement process and the available enhancement procedures.

3.3.1. The Enhancement Process

The work that the enhancer needs to do is determined by the metadata associated with the classes. This metadata can come from annotations or from a presistence unit (and any ORM XML files referenced by the persistence unit). For example, whether a class is an entity class, or not, may be specified with the `@Entity` annotation in your Java source file or by specifying the class as an entity in an ORM file with an `<entity>`

element. At enhancement time, the metadata specified in the persistence unit has precedence and will override any metadata from annotations.

Metadata Consistency

When you enhance your application classes, the metadata information, whether from annotations or from the persistence unit, becomes associated with the class files. When you create an `EntityManagerFactory` V/JPA checks that the information in the class matches the information in the persistence unit. Problem inconsistencies will cause an exception. (Inconsistencies will not occur if you are using run-time enhancement.) The following summarizes the possible post-enhancement conflicts.

@Entity

If you have an application class (a non-entity class at enhancement time) and change the persistence unit after enhancement so that the class is now declared as an entity class, V/JPA will complain with an exception. However, an enhanced entity class that is changed to non-entity status in the persistnce unit will still be an entity class and V/JPA will not complain.

@Id
@Transient
@Version
@Access

Following enhancement, changing the status of a field results in an exception.

Consistency across multiple persistence units

For strict and lenient enhancement modes the consistency of the metadata in the various persistence units is also important. If information about an entity class is found in more than one of the processed persistence units the information must be in agreement. This applies to both run-time and build-time enhancement.

If information for the same class is found in more than one persistence unit, the following must agree. Conflicts will result in V/JPA throwing an exception when processing the metadata.

Entity level

- Access type (field vs. property)
- Attribute list

Attribute level

- Access type (field vs. property)
- ID metadata
- Version metadata
- Transient metadata

Resolving the Metadata—Enhancer Modes

For metadata other than annotations, the enhancer needs to know which persistence unit(s) to process. There are three modes the enhancer can use in determining the units to process.

single

This is the default mode. The enhancer expects a single `persistence.xml` file that defines one persistence unit. The `persistence.xml` file must be reachable by the class loader. If multiple files are found or if the file defines more than one persistence unit the enhancer will exit with the exception `CannotResolveSinglePersistenceUnitException`.

strict

In this mode you supply a list of persistence unit names to be processed. The `persistence.xml` files defining the units must be reachable by the class loader.

lenient

This directs the enhancer to scan the class loader for all reachable `persistence.xml` files and process all the persistence units defined in these files.

3.3.2. Run-time Enhancement

The entity class enhancement is done at run-time through a Java agent. That is, the enhancer runs in the JVM and the run-time entity class definitions are enhanced when a class is loaded for use in your application. To perform run-time enhancement, whenever you run your V/JPA application you must start the JVM with a **-javaagent** option specifying the `jpa-agent.jar` file as the agent. For example, for a default Windows installation the argument is as follows (single enhancement mode).

```
-javaagent:C:\Versant_Lab\8\jpa-agent.jar
```

For strict enhancement, specify the persistence unit(s) as follows.

-javaagent:C:\Versant_Lab\8\jpa-agent.jar=*Unit-1,Unit-2,Unit-n*

For lenient enhancement, specify the **-lenient** option as follows.

-javaagent:C:\Versant_Lab\8\jpa-agent.jar=-lenient

The `jpa-agent.jar` is provided by the Versant JPA plugin. It is also available for standalone use in the `lib` directory in your V/JPA installation. Adjust the path accordingly for your **java** command line calls.

> **Run-time enhancement with the V/JPA Eclipse plug-in**
>
> If you are using the Versant JPA Eclipse plugin, this argument can be automatically included into each of your run configurations. The project needs to have the **JPA Nature** configured. To enable this nature for a Java project, right click on the project name in the **Package Explorer** tree and choose **Versant JPA Nature > Add JPA Nature**. You can remove the nature from any project through the same context menu.

3.3.3. Build-time Enhancement—The Enhancer Ant Task

An **ant** task is provided for enhancing your V/JPA `.class` files as part of the application build process. The task class is `com.versant.jpa.enhancer.ant.EnhancerTask` and is contained in the `vodjpa.jar` file. The task will enhance all `.class` and `.jar` files that are contained in the specified **ant** `<fileset>` collection. All classes that are annotated as `@Entity` are enhanced, even if the class is not specified in the persistence unit (`persistence.xml` file).

Using the Enhancer Task

To use the task within an **ant** build file, declare the task as follows (you can use any name you wish for the task name).

```
<taskdef name="VJPAEnhancer"
    classname="com.versant.jpa.enhancer.ant.EnhancerTask"
    classpath="vodjpa.jar" />
```

Call the task after you have compiled your V/JPA application classes. The task copies the files you specify in a `<fileset>` element to the given destination directory and enhances the entity classes (`.class` or `.jar` files) in the destination. The basic syntax for single-mode enhancement is as follows.

```
<VJPAEnhancer destdir="resultPath">
   <fileset>...</fileset>
</VJPAEnhancer>
```

Optionally you can define or reference a classpath with the `classpath`, `classpathref` and `inheritsClasspath` attributes. For specifying strict or lenient enhancement mode use the attribute `persistenceUnit` or `lenient`, respectively. (Attributes are described below.)

The following example call will copy all files in *`sourcePath`*`/bin` to *`resultPath`*`/bin` and enhance the entity classes defined in any `.class` and `.jar` files. A classpath is also specified. The enhanced versions are found in the location *`resultPath`*`/bin`.

```
<VJPAEnhancer destdir="resultPath/bin" classpath="sourcePath/META-INF"
    <fileset dir="sourcePath/bin">
        <include name="**/*" />
    </fileset>
</VJPAEnhancer>
```

For `.jar` files, the enhancer creates a new `.jar` file in the result (`destdir`) directory with the same name and structure, enhancing or copying the content as appropriate.

In-place enhancing

You can also use the **ant** enhancer task to enhance `.class` files in-place by setting the `destdir` to the same directory specified for the `<fileset>`. In-place enhancment will not enhance `.jar` file contents.

Table 3.1. V/JPA Enhancer ant Task Attributes & Elements

Attribute	Description
`destdir`	Specifies the destination directory. All files specified in the `<fileset>` are copied here. Any `.class` and `.jar` (not in-place) files are enhanced in this location. Required
`classpath`	Specifies a classpath, as an **ant** path structure, to resolve dependencies and/or resources needed during enhancement, e.g., `persistence.xml`. Optional

Attribute	Description

`classpathref`	Allows you to reference a pre-defined **ant** path as the classpath for the enhancement task.
	
	`<path id="myref">`
	`  <fileset>...</fileset>`
	`  <path location="..." />`
	`</path>`
	`<VJPAEnhancer ... classpathref="myref">`
	`  ...`
	`</VJPAEnhancer>`
	
	Optional

`inheritsClasspath`	Determines whether the enhancer uses the inherited JVM classpath. Note that if set to `false` even basic Java classes will not be found unless you have specified the necessary locations with the `classpath` or `classpathref` attributes. Optional, default is `true`

`persistenceUnit`	Specifies a comma-separated list of persistence units, by name, for strict-mode enhancement.
	
	`<VJPAEnhancer destdir="resultPath"`
	`  persistenceUnit=`*`Unit-1,Unit-2,Unit-n`*`>`
	`  <fileset>...</fileset>`
	`</VJPAEnhancer>`

`lenient`	Specifies lenient enhancement mode when set to `true`
	
	`<VJPAEnhancer destdir="resultPath" lenient="true">`
	`  <fileset>...</fileset>`
	`</VJPAEnhancer>`

Element	
`<fileset>`	Specifies the files to be enhanced. All files specified are copied to the `destdir` location. Any `.class` and `.jar` files (`.jar` files are not enhanced in-place) are enhanced (in the `destdir` location). Required

3.3.4. Build-time Enhancement—The Java Executable Enhancer

An executable class, `com.versant.jpa.enhancer.main.EnhancerMain`, is also provided, in the `vodjpa.jar` file, for performing the enhancement step. The basic syntax for `EnhancerMain` is as follows.

```
EnhancerMain [-i] -cp <dependencies> -d <destdir> -b <basedir>
    [<file> [ ... <fileN>]]
```

(The arguments are described below.)

The following example call will enhance all entity classes in the `bin` folder and save the enhanced versions to the `enhanced-bin` folder.

```
java
  -cp %VERSANT_ROOT%/lib/vodjpa.jar;
     %VERSANT_ROOT%/lib/antlr-runtime-3.3.jar;
     %VERSANT_ROOT%/lib/asm-all-4.0.jar;
     %VERSANT_ROOT%/lib/javax.persistence_2.0.0.jar
  com.versant.jpa.enhancer.main.EnhancerMain -cp bin -b bin -d enhanced-bin
```

The example above uses single enhancement mode (the default). For strict or lenient enhancement mode, set the appropriate argument, **-u**/**--persistenceUnit** or **-l**/**--lenient**, respectively.

The following table describes the `EnhancerMain` arguments.

Table 3.2. V/JPA Enhancer Class `EnhancerMain` Agruments

Argument	Description
-i	Specifies *isolated mode*, i.e., the classpath will not be inherited from the JVM in which the enhancer is running. This is optional. By default, classes from the enhancer JVM will be used to resolve dependencies in the classes being enhanced.
-classpath **-cp**	Used to resolve dependencies such as, e.g., the persistence unit file, `persistence.xml` This is optional.
-b	Specifies the base directory containg the files to enhance. This is optional. The default is the current directory.

Argument	Description
-d	Specifies the destination directory. The files to be enhanced are copied to this directory and enhanced, as appropriate. This is required. **In-place enhancement** You can perform the enhancement for `.class` files in-place by specifying the same directory for the destination as for the base directory. In-place enhancement applies only for `.class` files and will not enhance `.jar` file contents.
-h	Prints a help screen and exits
<file>	A `.class` or a `.jar` file. (Wildcards are not supported.) Optional. If no files are specified, all the files in the base directory (specified by the **-b** argument) are recursively enhanced.
-u **--persistenceUnit**	Specifies a comma-separated list of persistence unit(s) for strict enhancement mode. **java -cp vodjpa.jar EnhancerMain -d *destinationDirectory* ** **-u *Unit-1,Unit-2,Unit-n***
-l **--lenient**	Specifies lenient-mode enhancement. **java -cp vodjpa.jar EnhancerMain -d *destinationDirectory* -l**

3.4. Versant Administration API

The Versant administration API is located in the package `com.versant.admin`. It allows to you create and remove Versant JPA databases from your application code. The package consists, primarily, of the `ServerAdministration` class. Use one of the methods `ServerAdministration.createDatabase(..)` to create a new database, `ServerAdministration.verifyDatabaseExists(..)` to check whether a database exists and `ServerAdministration.removeDatabase(..)` to remove an existing database.

The administration API class and methods are described in the Javadoc reference (`doc/en_US/jpa/javadoc/index.html`, class `ServerAdministration`). You can also see how

the V/JPA administation API is used to perform these server operations in the tutorial and other V/JPA examples. Refer, for example, to the tutorial application class `Utility_createTutorialDatabase`.

3.5. Distributed Databases

Versant JPA supports working with *distributed databases*, that is, the ability to store objects to and retrieve objects from multiple databases within the context of a single `EntityManagerFactory`.

Using the Generic API

If you will be using the generic API with distibuted databases, refer to Section 3.10, "The Generic API and Distributed Databases" [p. 58].

The databases to use are specified as a list of database URLs in the value of the `versant.connectionURL` property. Each URL is separated by the `|` character. For example, the following lists two databases.

```
<property name="versant.connectionURL"
   value="versant:database1@localhost|versant:database2@myOtherMachine"/>
```

The first database on the list becomes the *default database*. This default database is where new objects are persisted when a database is not otherwise specified. The *default database* can be changed by setting the entity manager property `VersantEntityManager.setDefaultDatabase(Database)`. The role of the default database is described in detail below.

To use the API for distributed databases, instances of `EntityManagerFactory`, `EntityManager`, and `Query` need to be cast to `VersantEntityManagerFactory`, `VersantEntityManager`, and `VersantQuery`, respectively.

Note that the list of databases for an `EntityManagerFactory` instance can not be changed at runtime, but it is possible to create another instance of `EntityManagerFactory` with a different set of databases.

The list of databases for a particular `VersantEntityManagerFactory` can be obtained via `getAllDatabases()`. Also, the method `getDatabase()` of `VersantEntityManager` can be used to find out the database for a given persistent object.

Atomic Commit

The default database at the time the transaction is started (when `em.getTransaction().begin()` is called) becomes the *coordinator database* of this transaction. If objects in any database other than the

coordinator database are modified within a transaction, then at transaction commit time, the *2-phase commit* protocol is used. The 2-phase commit guarantees an *atomic commit*. However, a 2-phase commit is more expensive (slower) than a simple commit, because it requires more communication and logging operations. When objects in only the coordinator database are modified, then a less expensive simple commit is executed.

The Default Database and Persistent Objects

When using distributed databases, if a database is not specified, objects that are made persistent—either implicitly, via cascading the persist operation, or explicitly with a `persist(object)` API call—are stored in the current *default database*. The default database is the first in the list of databases given for the `versant.connectionURL` persistence unit property or the database you specify with the `VersantEntityManager.setDefaultDatabase()` entity manager property.

You can override the default by specifying the database where you want the objects to be stored when calling the `persist()` method. The method to specify a database is `persist(database,object)`. If the database is specified, all objects that are *implicitly* made persistent, i.e., via cascading the persist operation, *during the make persistent operation* will also be stored in the specified database.

A few examples will help make this clear. The first example stores the objects in the default database. Assume that at the beginning of the examples, the default database is `dbOne`.

```
em.getTransaction().begin();
    // default database is dbOne
Person p = new Person();
    // Person is a persistence capable class
em.persist(p);
    // will be assigned to dbOne (the default) at commit
p.getAddresses().add(new Address());
    // Address is a persistence capable class
em.getTransaction().commit();
```

Here the default database, `dbOne`, is used for both objects in the object network. The `Person` object because a different database was not specified and the `Address` object via cascading the persist operation. The asssignment is made when the transaction commits.

The next example will store the same object network. However, the objects in the network will reside in different databases.

```
em.getTransaction().begin();
    // default database is dbOne
Person p = new Person();
em.persist(dbTwo, p);
```

```
    // the new Person object is assigned to dbTwo
p.getAddresses().add(new Address());
    // the new Address object will be assigned to dbOne
em.getTransaction().commit();
```

When `em.persist(dbTwo, p)` is called, the `Address` object is not yet a part of the object network. Once the `persist()` call finishes, the default database is again used. So, it is `dbOne` that is assigned to the address when it is made persistent, via cascading the persist operation from the `Person` object, during the commit operation.

Here is another example that stores both objects of the object network in the same database, but not the default database.

```
em.getTransaction().begin();
    // default database is dbOne
Person p = new Person();
p.getAddresses().add(new Address());
    // the new Address object is not yet assigned to a database
em.persist(dbTwo, p);
    // the new Person and Address objects will be assigned to dbTwo
em.getTransaction().commit();
```

Since the `Address` object was a part of the object network when the `Person` object is assigned to `dbTwo`, it is assigned to the same database. The point of this and the previous example is that the order of the calls can make a difference for a distributed database system.

Finally, here is an example using `setDefaultDatabase()`.

```
em.getTransaction().begin();
    // default database is dbOne
em.setDefaultDatabase(dbTwo);
    // change the default database for this pm
Person p = new Person();
em.persist(p);
    // the new Person object is assigned to dbTwo
p.getAddresses().add(new Address());
    // the new Address object will also be assigned to dbTwo
em.getTransaction().commit();
```

Note that calling `persist()` for an object which has already been written to a database (in a prior transaction or a flush operation) has no effect and will not change the database assignment for the object.

A new object that has been made persistent in the current transaction cannot have its assigned database changed. If this is attempted an exception will be thrown. This applies to cascading persistence as well. For example, if you need to persist a tree-like structure, where the root node should be in one database but its children in a different database (or different databases), either persist the children first to the desired database(s) and the root node afterwards or set the references to the children after persisting the root node.

Queries

As a default, queries are executed on all databases, however there are the following limitations:

- expressions in the filter clause (i.e., the `WHERE` part), are not evaluated across multiple databases but only within each database
- ordering conditions (i.e., the `ORDER BY` part) are ignored

To limit the number of database queries executed, the following property can be set:

```
VersantEntityManager.setDefaultQueryDatabases(Database[])
```

Alternatively, you can use the overload for `getResultSet(Database[])` provided by `VersantQuery`, which allows the explicit specification of a set of databases on which the query needs to be executed.

Merging with Distributed Databases

When you merge a detached entity (and its associated network) to an entity manager with multiple databases, any entities that already exist in a database will be returned to that database. New entities will be created in the currently defined default database (refer to Section 3.5, “The Default Database and Persistent Objects” [p. 43]). V/JPA provides two additional overloads of the `EntityManager.merge()` method in `VersantEntityManager` to allow you to specify the database for new entities during the merge operation (refer to the V/JPA API reference).

3.6. Entity Identity and the V/JPA Datastore

In the Versant JPA datastore, each entity has a unique identifier, the *LOID* or Logical Object IDentifier, that never changes for the life of the entity (and is never re-used). This LOID is also the primary key for your V/JPA entities and is the value of the `@Id` field. (Refer to Section A.1.1, “Primary Keys and Object Identifiers” [p. 60] for the rules related to the `@Id` field in V/JPA.)

> The Versant JPA datastore LOID is a 64-bit number (Java type `long` or `java.lang.Long`). The high-order 16 bits represent the database (in a distributed database environment) and the low-order 48 bits form the unique entity identity in the database.
>
> This is referred to as the *logical* identifier to differentiate it from any internal representation for the entity used by the physical datastore. Since the logical identifier is used as the primary key in your V/JPA application it satisfies the requirement that the key be a unique identifier for the entity.

Because the LOID is unique for each entity in a given database and must represent the entity for its lifetime in the database, Versant JPA is responsible for allocating these values. When you create an entity manager factory a connection to the server and database is made and the server will return a pre-allocated batch of identifiers. By default, these identifiers are allocated in blocks of 1024. When more identifiers are needed, a request is sent to the V/JPA server and a new batch of identifiers is returned.

If you are creating very large numbers of entities in the transactions associated with the entity manager factory the requests for additional batches of identifiers can result in a small, but nevertheless measurable, performance reduction. If you know that a significantly large number of identifiers will be needed, you can instruct the entity manager factory to request batches in blocks of an appropriate size.

Adjusting the LOID Batch Size

The default LOID batch size is 1024. You can change this with the persistence unit property `versant.defaultLoidBatchSize`. For example, the following would cause the entity manager factories to fetch identifiers in batches of 500000.

persistence.xml

```
<persistence version="2.0">
    <persistence-unit
            name="my_persistence_unit"
            transaction-type="RESOURCE_LOCAL">
        . . .
        <properties>
            <property
                    name="versant.defaultLoidBatchSize"
                    value="500000" />
        </properties>
    </persistence-unit>
</persistence>
```

The value specified by the property applies to all databases opened by the entity manager factories calling the persistence unt.

You can also change the batch size at runtime with the `VersantEntityManagerFactory` method `setLoidBatchSize(Database *database*, int *loidBatchSize*)`. This method adjusts the batch size for the specified database in the calling entity manager factory. You can set the *`database`* argument to `null` to refer to the first (or only) database in the list of open databases. (Refer to Section 3.5, "Distributed Databases" [p. 42] for information about using multiple databases.)

This example sets the LOID batch size to 500000 for the first (or only) connected database in the calling entity manager factory.

```
VersantEntityManagerFactory emf = (VersantEntityManagerFactory)
        Persistence.createEntityManagerFactory("my_persistence_unit");
emf.setLoidBatchSize(null, 500000);
```

The value you specify with `setLoidBatchSize()` overrides any value set with the `versant.defaultLoidBatchSize` property (or the default if the property is not set). The specified batch size will take effect the next time the factory requests a LOID batch, i.e., when the current batch is exhausted. The value stays in effect until you specify a different value (with another call to `setLoidBatchSize()`) or the factory is closed.

If your application will only occasionally need a large number of identifiers, you should use the runtime `setLoidBatchSize()` method and be sure to return the batch size to a reasonable value when appropriate. Though 2^48 is a very large number, you could inadvertantly deplete the available identifiers if every fetch results in, for example, millions of identifiers that may not be used.

In any case, use care when adjusting the LOID batch size, especially when changing the default using the persistence unit property. Very small values can result in excessive requests for additional batches with a potential negative impact on application performance.

3.7. Collection Optimization

In order to optimize working with collection, Versant JPA provides the ability to use the entity identity when determining collection membership. The usual mechanism for determining whether, for example, a collection contains a particular entity is to use the `equals()` method. Using the entity identity (the `@Id` field value) for such comparisons is much more efficient.

When you read an entity that contains a collection field from the database, only the entity identities for the collection elements are loaded. The elements themselves are *lazy-loaded*, i.e, only loaded on-demand. (You can change this behavior with `FetchType=FetchType.EAGER`. The default is `FetchType.LAZY`) If a collection uses the `equals()` method for entity comparison, the collection elements must be loaded when performing many operations on the collection. By using the entity identity for comparisons you can get around loading the element entities.

To specify that a collection use the entity identity for comparisons, V/JPA provides a vendor-specific annotation, `@EqualsById`. The following shows an example.

```
public class Book {
    . . .
    @EqualsById
    Set<Person> authors;
    . . .
}
```

The `@EqualsById` annotation is supported for `Set` and `Map` types and applies to the methods of those classes that perform entity comparisons. (Refer to Table 3.3, "@EqualsById—Supported Classes and Methods" [p. 48] for the supported classes and methods.)

For collections using `@EqualsById`, the identity of the entity determines whether the entity matches another in the collection. If you are relying on the `equals()` method to determine if two objects are the same (e.g., objects with the same content are equal) the `@EqualsById` annotation will likely not be appropriate. Use the `@EqualsById` annotation only for collections where the entity identity comparison satisfies your requirements.

Newly-created entities added to a collection using `@EqualsById` are immediately made persistent. The persist operation assigns the entity identity which is needed for entity comparison.

Table 3.3. @EqualsById—Supported Classes and Methods

Class	Methods

`Set`	`add()`
	`addAll()`
	`contains()`
	`containsAll()`
	`remove()`
	`removeAll()`
	`retainAll()`

Class	Methods

`Map`	`put()`
	`putAll()`
	`containsKey()`
	`remove()`

3.8. V/JPA Indexes

The JPA standard does not define any indexing related constructs. As vendor-specific extensions, Versant JPA provides two mechanisms for defining class attribute indexes. You can define indexes using class-level annotations or in an *index definition file* which is referenced in the persistence unit, i.e., the `persistence.xml` file. (The latter is similar to the JPA-specified `mapping-file` property to reference an ORM file.)

Versant JPA supports non-unique BTree indexes. Indexes on `@Id` and `@Version` attributes are not supported. Indexes must be named and if a given class has more than one index the index names must be unique.

As mentioned above, an index is defined in a class annotation or in an index definition file. You can mix the two approaches with some indexes defined in the class and others in the definition file. Index annotations and index definition files are described in the following sections.

V/JPA Index Annotations

The `Index` annotation allows you to define an index for a given class. The index may be on a single class attribute or multiple class attributes. (These are often referred to as *simple* or *compound* indexes, respectively. However, V/JPA does not make that distinction.) The `Index` annotation is supplied with two annotation attributes, the name of the index and a list of the class attributes to be indexed. The following shows an example, indexing two class attributes.

```
@Entity
@Index(name = "CourseNumberDurationIndex",
    attributes = {"courseNumber", "duration"})
public class Course {
    @Id
    long id;
    String courseTitle;
    int courseNumber;
    int duration;
```

```
    . . .
}
```

The value of the `name` attribute is a `String` and specifies the index name. The `attributes` attribute value is a `String` array specifying the class attribute(s) to be indexed. The order of the class attributes is significant. The first attribute specified is the primary index.

Multiple indexes

If you want to specify more than one index for a given class you can wrap multiple `Index` annotations in an `Indexes` annotation. The value of an `Indexes` annotation is an array of `Index` annotations.

```
@Entity
@Indexes(
    {
        @Index(name = "Index_1", attributes = {"field_1"}),
        @Index(name = "Index_2", attributes = {"field_2, field_3"})
    }
)
public class ClassWithMultipleIndexes {
    @Id
    long Id;
    String field_1;
    int field_2;
    long field_3;
}
```

V/JPA Index Definition File

Indexes may also be defined using an index definition file. The following example shows such a file for the multiple indexes example described above.

```
<index-definitions version="1.0">
    <class name="com.example.ClassWithMultipleIndexes">
        <index name="Index_1">
            <attribute name="field_1"/>
        </index>
        <index name="Index_2">
            <attribute name="field_2"/>
            <attribute name="field_3"/>
```

```
        </index>
    </class>
</index-definitions>
```

In order to associate a persistence unit with an index definition file, the property `versant.indexesDefinitionFile` is used.

```
<persistence-unit ...>
    <class>...</class>
    <properties>
        <property ... />
        <property name="versant.indexesDefinitionFile"
                value="META-INF/indexes.xml" />
    </properties>
</persistence-unit>
```

The index definition file may have any name that locates the file as a resource in the classpath.

Index Inheritance

In V/JPA sub-classes inherit the indexes of their super-classes by default. Assume you have the following class definitions.

```
@Index("AaIndex", {"a"})
public class A {
    . . .
    int a;
}

public class B : extends A {
    . . .
    int b;
}
```

The class `B` also has an index on the inherited class attribute `a`.

You can disable automatic index inheritance by setting the property `versant.inheritIndexes` to `false` in the appropriate persistence unit (i.e., in the `persistence.xml` file).

```
<persistence-unit ...>
    <class>...</class>
    <properties>
```

```
        <property ... />
        <property name="versant.inheritIndexes" value="false" />
    </properties>
</persistence-unit>
```

Indexing Inherited Attributes

An index definition may contain class attributes from super-classes. For example, assume classes A with class attribute a and B with class attribute b.

```
public class A {
    . . .
    int a;
}

@Index("BbAaIndex", {"b", "a"})
public class B : extends A {
    . . .
    int b;
}
```

An index defined for class B can contain the member a from the parent class A.

If a given class does not contain the class attribute named in the index definition, the occurance of the named class attribute from the nearest related class is used. For example, consider the next class hierarchy.

```
public class A {
    . . .
    int a;
}

public class B : extends A {
    . . .
    long a;
    int b;
}

@Index("CcBaIndex", {"c", "a"})
public class C : extends B {
    . . .
```

```
    int c;
}
```

In this case, the index for class `C` has the class attributes `c` from class `C` and `a` *from class B*.

Building Index Meta Data From Annotations

The V/JPA enhancer reads the `Index` and `Indexes` annotations for all classes that are to be enhanced. It builds up the metadata that is needed to create indexes. This metadata is applied when you create an `EntityManagerFactory` instance.

The enhancer will throw an exception on the following conditions.

- The attribute annotated as `@Id` is in the indexed attribute set (exception `IndexOnIdAttributeException`)
- The attribute annotated as `@Version` is in the indexed attribute set (exception `IndexOnVersionAttributeException`)
- An attribute named in the indexed attribute set does not exist in the entity class (exception `IndexedAttributeNotFoundException`)
- The indexed attribute set is empty, i.e., no attributes to be indexed are specified (exception `IndexOnEmptyAttributeSetException`)
- The index name is `null` or empty (exception `InvalidIndexNameException`)
- The same name is used for more than one index definition (whether via annotations or in the index definition file; exception `NonUniqueIndexNameException`)

Further checks are done by the server at schema comparison time.

Building Index Meta Data From an Index Definition File

Index definition files are parsed only at runtime, at the creation of the `EntityManagerFactory` instance. Index definitions found at that point override index definitions of equally named indexes built by the enhancer from annotation. The same checks for correctness are applied to indexes defined in the index definition file(s) as for annotations.

Index Evolution

The defininition of an index, that is the fields that are indexed and/or their order in the index, can *evolve* or change over time. In V/JPA, valid index evolution can be adding fields to an index, removing fields from an index or changing the sequence of the indexed fields. These evolutionary changes are automatically propagated to the database if the property `versant.indexEvolve` is set to true.

Index evolution is needed if the metadata defining the index (`@Index`/`@Indexes` annotations and/or in the indexes definition file) has changed and there is a same-named index in the database for the same class but with different attributes or attribute sequence. In any given V/JPA database, only indexes that were, or potentially could have been, defined using the V/JPA binding can be changed, i.e.,named, non-unique, BTree indexes.

For example, assume that for a given class there is an index named `Index_1` in the database on attributes `member1` and `member2`. The index was created based on an older version of the class which contained an annotation that defined exactly the index on these two attributes. You now have evolved the definition of the class with the index, still named `Index_1`, defined on attributes `member1`, `member2` and `member3`. If index evolution is allowed (i.e., `versant.indexEvolve` is set to true), `Index_1` will be modified in the database to be in accord with the new version of the class.

3.9. Cursor Queries

Versant JPA supports *cursor queries*. A cursor query returns the result as an iterator rather than a collection. The iterator acts as your cursor that points to a specific location in the query result. The cursor (iterator) always begins at the first item in the query result and is forward-only, i.e., you can only step forwards and not backwards in the result. A cursor query is particularly useful if the query returns a very large number of results and if an iterator is appropriate for your application.

> An example demonstrating a cursor query is included in the `sdk/examples/jpa/jpa_cursor_query` directory in your Versant JPA installation.

The query itself is, in effect, executed in several batches of a configurable result size. These intermediate results are transferred to the client as the cursor traverses the results. Note that if you are specifying an “order by” condition the entire query result must be calculated on the server but the results are transferred to the client in the specified intermediate batch size.

The Versant JPA class `VersantQuery` extends the standard JPA `Query` interface to provide the cursor query functionality. To execute a cursor query, use the `VersantQuery` method `getResultCursor(int batchSize)`.

```
em.getTransaction().begin();
VersantQuery query =
    (VersantQuery) em.createQuery("select these from ThatClass these");
Iterator cursor = query.getResultCursor(1000);
// use the cursor to read and operate on your results . . .
em.getTransaction().commit();
```

The method returns a Java `Iterator` instance (*not a collection*). This iterator is the cursor for the query results in the client application. The *`batchSize`* argument (`1000` in the command above) specifies the size of the intermediate result batch that is transferred to the client.

> The intermediate result size specified by the `getResultCursor()` *`batchSize`* argument should not be confused with the batch size specified with the `versant.queryBatchSize` property.

Cursor queries must be executed within a transaction and the cursor is only valid within that transaction. Attempting to use the cursor (iterator) outside of the transaction in which it is created will result in an exception. The cursor query is effectively closed when the last element of the iterator is reached. For distributed databases, the cursor "walks" the results in a serial fashion. That is the results from the first database in the list of databases followed by the results from the second database and so on.

3.10. V/JPA Generic API

The generic API provides a freestanding mechanism for accessing the content of a V/JPA database without the need of having the JPA entity classes available. This allows you to write tools that can browse and modify the database content.

> The pre-defined internal system classes are not visible through the V/JPA Generic API.

Enabling generic access

To enable generic access for a given persistence unit, add the following property to the unit definition in the `persistence.xml` file.

persistence.xml

```
<property name="versant.genericAccess" value="true" />
```

Each V/JPA class in the database is represented by an instance of `com.versant.jpa.generic.DatabaseClass`. Any generic instance is an instance of `com.versant.jpa.generic.DatabaseObject`.

To determine the type of a `DatabaseObject` instance use the method `getType()` which returns the corresponding instance of `DatabaseClass`.

The `DatabaseClass` class defines a number of static methods for looking up instances of `DatabaseClass`, for example, `forName(String` *`className`*`, EntityManagerFactory` *`scope`*`)`. The `EntityManagerFactory` instance must be supplied in order to set the scope in which you are looking for the class.

Working with generic instances

> A simple example application using the V/JPA Generic API is provided in the `sdk/examples/jpa/jpa_generic_api` directory in your V/JPA installation. To use the example, read the instructions in the `README.txt` file in the example directory.

All entities returned through any JPA API or simply via navigation are of type `DatabaseObject`. If you need to create a new instance of a particular class, you can call `newInstance()` on the `DatabaseClass` object.

Values of an instance field are obtained or modified with the `DatabaseField` `get` and `set` methods.

```
DatabaseObject dbObj = . . .
DatabaseField field = dbObj.getType().getDeclaredField(fieldname);
Object o = field.get(dbObj);
field.set(dbObj, value);
```

To obtain a list of available fields, the method `getDeclaredFields()` can be called on a `DatabaseClass` instance. This returns the declared, but not the inherited fields, of this class. The fields

are represented as instances of `com.versant.jpa.generic.DatabaseField`, and `getType()` returns the declared type of this field, which can be a `java.lang.Class` instance (for primitive fields) or a `com.versant.jpa.generic.DatabaseClass` instance (for a single reference to another persistent class). To get the full type information, the method `getTypeName()` can be used. For example, if `getType()` returned `List.class`, then `getTypeName()` also contains the generic arguments of this `List` (e.g., `java.lang.List<PersClass>`).

Type mappings

Many Java types are mapped to the same database types, and the generic API does not have the full information available, what the original declared type in the Java Entity class has been. For the following (non-collection) field types, the generic API provides the type exactly as originally declared.

- Primitives: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`
- Wrappers (`java.lang`): `Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`
- Single typed reference to another persistent object (e.g., `PersClass myReference`)
- `java.sql.Timestamp`, `java.lang.String`
- arrays of primitives: `boolean[]`, `char[]`, `byte[]`, `short[]`, `int[]`, `long[]`, `float[]`, `double[]`

date substitution

Any supported date type (`java.util.Date`, `java.sql.Timestamp`, `java.sql.Time`, `java.sql.Date`) is represented as `java.sql.Timestamp`, because it offers the highest precision. This also applies if the date type is a generic argument of any collection or map.

interface substitution

Any interface is represented as `java.lang.Object`. This also applies if the interface type is a generic argument of any collection or map.

array substitution

Any array type, except arrays of primitives, is represented as appropriately typed `java.util.List`.

collection substitution

Any supported `Collection` type is represented as an appropriately typed `java.util.List`.

map collection

Any supported `Map` type is represented as an appropriately typed `java.util.Map`.

Note, that even sets are represented as `List`. Using the generic API, you can break the `Set` contract by inserting duplicates. Duplicates will be removed when you next access the set with the non-generic API, however, it can not be guaranteed which of the duplicate objects contained in the set will be removed.

Note, that the declared type might rely on the `hashCode()` and/or `equals()` implementation of the entity class. The `DatabaseObject` instance cannot reflect any such user implemented behavior. Thus, the contract of the declared field might be violated.

Note, that some collections/map types don't support adding `null`.

Generic Queries

The generic API classes do not show the ID field (`@Id`) in the the list of declared fields. (You can use the `DatabaseObject.getLoid()` method to get this value for a particular object.) Since the ID field is not directly available from the generic API you are not able to formulate a query with your ID field name in the expression. In order to allow a query to use the ID field, V/JPA provides an artificial field, `selfoid`, for use in a query expression.

```
select p.selfoid from Person p
```

(Note that if you have declared a field with the name `selfoid` in your class, that field will be used.)

Cascade and eager load settings

The original cascade and eager load settings are not known to the generic API, so they will have the field type dependent default values.

The Generic API and Distributed Databases

The generic API will only work with distributed databases if the schema information for each database is the same. If the schemas are not found to be equal, an exception will be thrown when opening the `EntityManagerFactory` instance. Schemas are considered equal if they have the same set of classes with the same inheritance hierarchy. Additionally, for each class, the lists of fields are compared and need to be equal though the order of the fields in the different database schemas need not be the same. Classes not defined by V/JPA (such as, e.g., internal system classes) and index definitions are ignored when comparing the schemas.

Appendix A. Versant JPA and the JPA Specification

Sections

- *Versant JPA Implementations*
  - *Primary Keys and Object Identifiers*
  - *Mapping*
  - *Normalization*
  - *Flush Operations*
  - *Entity Class Enhancement*
  - *Cascading Persistence*
  - *Query Support*
  - *Object Cloning*
  - *Entity Lifecycle Listeners*
  - *Pessimistic Locking*
- *Versant JPA—Currently Supported Features*
  - *JPA Features*
  - *JPQL Support*
- *JPA Public API Support*
  - *Package* `javax.persistence`
  - *Package* `javax.persistence.spi`
  - *Package* `javax.persistence.criteria`
  - *Package* `javax.persistence.metamodel`
  - *Annotations*

Versant JPA is an implementation of the Java Persistence API—as specified in JSR-000317 Java™ Persistence 2.0 of the Java Community Process Program—for Versant JPA.

This describes the JPA features that are currently supported in Versant JPA and the areas where V/JPA differs with regard to the official JPA specification.

> The official specification for JSR-000317 Java™ Persistence 2.0 can be downloaded at http://jcp.org/aboutJava/communityprocess/final/jsr317/index.html

A.1. Versant JPA Implementations

This section provides you with important information about the Versant JPA implementation of certain points of the JPA specification.

A.1.1. Primary Keys and Object Identifiers

In a JPA entity class there must be a field or property that is designated as the primary key. This is indicated by the `@Id` attribute.

```
@Enitity
class Person {
    . . .
    @Id
    long id;
    . . .
}
```

The primary key uniquely identifies the entity object in the database. In an object-oriented database this unique identifier is referred to as the *object identity*. The object indentity in a V/JPA database is represented by a 64-bit number and the value is assigned by the V/JPA server. What this means for Versant JPA is that only types `long` or `java.lang.Long` are permitted for the `@Id` attribute. Prior to the object being made persistent the value of the ID field or property is zero (`0l`). The value is assigned when performing the persist operation and/or persist cascade operations take place during a flush.

Since this object identifier is assigned by V/JPA you may not use key generators or other mechanisms to set the ID and attempting to write to the field or property will result in a runtime error. Read access to the ID field is allowed.

This is a deviation from the JPA specification.

A.1.2. Mapping

Because Versant JPA stores your objects with a similar data model in the database and in memory there is no need to map between those two models. In fact, Versant JPA will ignore any mapping information you provide. This is true for mapping of attributes as well as for mapping of the inheritance structure. Keep it simple and just don’t specify anything mapping-related for the attributes in your class. It will just work.

This is a deviation from the JPA specification.

A.1.3. Normalization

The major differences between an object storage model and a relational storage model are how relationships and inheritance are handled. The storage model for an object-oriented database is, unsurprisingly, pretty similar to that of the data model of an object-oriented programming language such as Java or C++.

The types of attributes of a class in such a database can be categorized as follows.

- Basic types such as an `int` (`java.lang.String` is also treated as a basic type)
- References to other persistent objects (conceptually equivalent to a foreign key)
- Collections of basic types or references

So data-model-wise, this looks pretty similar to a table which has some collection-valued columns. In relational lingo, this would be a Non-First-Normal-Form (NFNF) database, because apparently this contradicts the first NF of the relational data model. However object-oriented models simply don't need that, so relationships are stored directly in the owning object. Thus, it is unnecessary to specify anything regarding the mapping of the relationship. It is not even necessary to use `@OneToOne`, `@OneToMany`, `@ManyToMany` or `@ManyToOne`, unless you want to specify some behavior for cascading operations.

This is a deviation from the JPA specification.

> Because the data model in the Versant JPA database is the same as in Java, bi-directional relationships are stored on both sides of the relationship. As there is no owner of a bi-directional relationship, if you change one side of the relationship you are responsible for changing the relationship in the other side. This is also required by the JPA specification.

A.1.4. Flush Operations

In addition to all the situations where V/JPA is required by the specification to do a flush operation (e.g., during commit, flush before query, explicit flush invocation, etc.), V/JPA also flushes if the number of modified, removed or inserted objects within a transaction has hit a certain threshold. This can be configured (the default is 10,000) and the main purpose is to allow the garbage collector to reclaim memory of those objects.

Obviously, by keeping strong references to all those objects, you would effectively negate V/JPA's housekeeping efforts. For objects, which have neither been inserted, removed nor modified, V/JPA only keeps weak references.

This is not a deviation from the JPA specification and is a permitted implementation strategy.

A.1.5. Entity Class Enhancement

Versant JPA uses a so-called enhancer to modify your `@Entity` classes to make them persistence capable. This enhancement can be triggered in different ways, e.g., at runtime by specifying a JVM argument to call the Versant JPA enhancement agent. Enhancement allows persisting and retrieving entities in the most efficient way.

This is not a deviation from the JPA specification and is a permitted implementation strategy.

A.1.6. Cascading Persistence

One could say that there are three kinds of entities, those which you have explicitly made persistent, those which are implicitly made persistent via cascading persist and those which, well, you have simply forgotten. JPA will be kind enough to tell you that you've forgotten something by the means of throwing an exception at you as soon as the next flush operation is performed.

However, if you are wondering how this could actually be a useful behaviour, JPA also defines a global switch to always enable cascade on persist for all associations. Versant JPA does not even support the not-so-useful behavior. (And we here at Versant are not even feeling guilty about that.) Persist cascade is always enabled for all associations and cannot be disabled, this is also known as persistence-by-reachability. This has been done to keep the implementation fast and maintainable, especially when interactions with automatic flushing take place. And this is really useful and really what you want.

This is a deviation from the JPA specification.

A.1.7. Query Support

Focusing on the efficient support for retrieving objects via navigation, the V/JPA query engine does not support all JPA query language constructs. For details see the feature matrix below.

This is a deviation from the JPA specification.

A.1.8. Object Cloning

If you clone a persistent entity, Versant JPA needs to ensure that this newly created object is not part of the same persistent context as the original object, but is, rather, a disconnected object. To ensure this, the enhancer injects a clone method into the top-most persistent class or modifies any user-defined clone method of the top-most persistent class.

This will cover all but some extreme corner-case clone implementations. For these cases, Versant JPA imposes the following restrictions.

- A user-written clone method may not call any JPA API methods (e.g., `EntityManager.persist()`)
- A user-written clone method may not manipulate any enhancer injected fields (the field names starting with `_vjpa`) via reflection

Cloning is not discussed in detail by the JPA specification. However, this behavior is a natural requirement.

A.1.9. Entity Lifecycle Listeners

The JPA specification does not strictly mandate at what point a lifecycle listener is called, allowing a degree of flexibility for the vendor. The following describes how and when various lifecycle listeners are called in Versant JPA.

The **pre-persist** and **pre-remove** listener events are performed, first, when the respective persist and remove operations are called.

The **pre-update** listener event is performed before the update operation is flushed to the database.

The **post-persist**, **post-remove** and **post-update** listener events are performed immediately after the respective operations are executed during flush. Be aware that the commit operation implicitly triggers a flush and that the listener events are performed after the flush but before the transaction is actually committed to the database.

Lifecycle callbacks are permitted to modify the non-relationship state of an entity (for example, modifying an `int` or string field). This is not a problem for pre-persist, pre-remove, post-remove or pre-update. However, if a post-persist, post-update or post-load listener modifies an entity, the entity is then treated as a “dirty” object which will need to be flushed, again, to the database.

For an explicit flush invocation, this simply means that after flush there are still dirty objects which will be flushed again with the next flush invocation or during commit. For a commit invocation, if any dirty objects are created they are immediately flushed again before the database commit is executed. This is done in a loop

until no dirty objects are created. This implies that if your listener (e.g., post-update) always unconditionally makes your object dirty, the commit call will be an endless loop.

A.1.10. Pessimistic Locking

For pessimistic locking, the JPA specification states that the `EntityManager.lock()` methods throw an exception, `EntityNotFoundException`, if the entity to be locked does not exist in the database. However, with Versant JPA, attempting to set a lock on a non-existent entity will not generate an error and no exception is thrown.

This is a deviation from the JPA specification.

A.2. Versant JPA—Currently Supported Features

A.2.1. JPA Features

A listing of general JPA feature support in V/JPA as well as currently unsupported features and features with limited support

Supported Features

The following JPA features are supported in this release of Versant JPA.

- Java Platform, Standard Edition Persistence
- JPQL

  (Currently, only a subset of the full JPQL language is provided. Refer to Section A.2.2, “JPQL Support” [p. 65].)
- Object lifecycle mechanisms

  persist, remove, refresh, detach, merge and cascading of these operations
- Transaction management
- Runtime enhancement of entity classes

- Build-time entity class enhancement
- Distributed databases
- Automatic schema evolution

Limited Support or Unsupported

The following JPA features are not yet supported or only partially supported in this release.

- Java Platform, Enterprise Edition persistence

  Currently, no support for container-based applications or external transaction management.

- The following Java classes are currently not supported.

  ```
  java.math.BigDecimal
  java.math.BigInteger
  java.util.Calendar
  ```

  If your class contains an attribute of any of the above types, an exception, `PersistenceException`, is thrown when you create an `EntityManagerFactory` instance.

- Mapped Superclasses

  Currently, not supported.

- Criteria query language

  Currently, not supported.

- Embedded Objects

  Currently, the `@embedded` annotation is ignored; embedded obects are stored as first-class objects.

- Versant FTS (V/JPA fault-tolerant server)

  Currently, not supported.

A.2.2. JPQL Support

The following summarizes the current support for JPQL expressions in V/JPA.

Supported JPQL expressions

The following table shows the currently supported JPQL expressions.

Table A.1. JPQL—Supported Expressions in V/JPA

JPQL Expression	Example	Comment
Comparison	select o from Person where o.age > :age	
Named Parameters	select o from Person where o.age > :age	
Positional Parameters	Select o from Person where o.age > ?1	
Between Expression	select o from Person where o.age between 8 and 5	
Literal	select o from Person where o.age > 5	
In Expression	select o from Person where o.age in (5, 6)	
In Expression(param)	select o from Person where o.age in :inParams, select o from Person where o.age in (:inParam1, :inParam2)	
Like Expression	select o from Person where o.name like '%en'	
Is [not] null Expression	select o from Person where o.address is not null	only supported for single references
Collection Member Declarations	select o from Employee o, in (o.address) adds where adds.street = :name	
Collection Member Expressions	select o from Employee o where :address member of o.address	not supported for identification variables ("select p from P p where p member of p.siblings").
Order by Expression	Select o from Person where o.age > :age order by o.age asc	

Unsupported JPQL expressions

The following JPQL expressions are currently *not* supported in V/JPA.

Table A.2. JPQL—Unsupported Expressions in V/JPA

JPQL Expression	Example	Comment
In Expression(subquery)	select o from Person where o.age in (select a.age from Employee a)	Only limited support planned.
All or Any Expressions	select auth from Author auth where auth.salary >= all(select a.salary from Author a where a.magazine = auth.magazine)	Only limited support planned.
Exists	select o from Employee o where exists (select e from Employee e where e.baseField = o.baseField)	Only limited support planned.
Collection Comparison Expressions	select mag from Magazine mag where mag.articles is empty	
Subqueries		Not planned.
Functional Expressions		
Aggregate Functions	select max(o.age) from Person o	
Entity TYPE() Expressions	select c from Company c, in(c.employees) e where TYPE(e) = FulltimeEmployee	Currently, not supported.
Constructor Expressions	select new com.company.PublisherInfo(pub.id, pub.revenue, mag.price) from Publisher pub join pub.magazines mag where mag.price > 5.00	Currently, not supported.
Distinct		Not planned.
Group By/Having		Not planned.
Update Queries		Not planned.
Delete by Query		Not planned.
Join		Not planned.

A.3. JPA Public API Support

A.3.1. Package `javax.persistence`

A.3.1.1. Primary Persistence Interfaces

Table A.3. Interface `EntityManager`

Method	Comments	Status
clear()		Supported
close()		Supported
contains(entity)		Supported
createNamedQuery(name)		Supported
createNamedQuery(name, resultClass)		Supported
createNativeQuery(sqlString)	Not applicable, no native SQL queries for V/JPA	N/A
createNativeQuery(sqlString, resultClass)	Not applicable, no native SQL queries for V/JPA	N/A
createNativeQuery(sqlString, resultSetMapping)	Not applicable, no native SQL queries for V/JPA	N/A
createQuery(criteriaQuery)		Currently, not supported
createQuery(qlString)		Supported
createQuery(qlString, resultClass)		Supported
detach(entity)		Supported
find(entityClass, primaryKey)		Supported
find(entityClass, primaryKey, lockMode)		Supported
find(entityClass, primaryKey, lockMode, properties)	Restriction: Setting properties currently has no effect	Supported

Method	Comments	Status
find(entityClass, primaryKey, properties)	Restriction: Setting properties currently has no effect	Supported
flush()		Supported
getCriteriaBuilder()		Currently, not supported
getDelegate()	Deprecated, use unwrap(cls) instead	N/A
getEntityManagerFactory()		Supported
getFlushMode()		Supported
getLockMode(entity)		Supported
getMetamodel()		Currently, not supported
getProperties()		Supported
getReference(entityClass, primaryKey)		Supported
getTransaction()		Supported
isOpen()		Supported
joinTransaction()	not needed for the currently supported transaction-type RESOURCE_LOCAL	Currently, not supported
lock(entity, lockMode)	Attempting to set a (pessimistic) lock on a non-existing object does not result in an error. (Refer to Section A.1.10, “Pessimistic Locking” [p. 64].)	Supported
lock(entity, lockMode, properties)	Restriction: Setting properties currently has no effect	Supported
merge(entity)		Supported
persist(entity)		Supported
refresh(entity)		Supported
void refresh(entity, lockMode)		Supported
refresh(entity, lockMode, properties)	Restriction: Setting properties currently has no effect	Supported
refresh(entity, properties)	Restriction: Setting properties currently has no effect	Supported
remove(entity)		Supported
setFlushMode(flushMode)		Supported

Method	Comments	Status
setProperty(propertyName, value)		Supported
unwrap(cls)	use unwrap(com.versant.jpa.VersantEntityManager.class) to access Versants vendor extensions	Supported

Table A.4. Interface `EntityManagerFactory`

Method	Comments	Status
close()		Supported
createEntityManager()		Supported
createEntityManager(java.util.Map map)		Supported
getCache()		Currently, not supported
getCriteriaBuilder()		Currently, not supported
getMetamodel()		Currently, not supported
getPersistenceUnitUtil()		Supported
getProperties()		Supported
isOpen()		Supported

Table A.5. Interface `EntityTransaction`

Method	Comments	Status
begin()		Supported
commit()		Supported
getRollbackOnly()		Supported
isActive()		Supported
rollback()		Supported
setRollbackOnly()		Supported

Table A.6. Interface `Cache`

Method	Comments	Status
contains(cls, primaryKey)		Currently, not supported

Method	Comments	Status
evict(cls)		Currently, not supported
evict(cls, primaryKey)		Currently, not supported
evictAll()		Currently, not supported

Table A.7. Interface `PersistenceUnitUtil`

Method	Comments	Status
getIdentifier(entity)		Supported
isLoaded(entity)		Supported
isLoaded(entity, attributeName)		Supported

Table A.8. Interface `PersistenceUtil`

Method	Comments	Status
isLoaded(entity)		Supported
isLoaded(entity, attributeName)		Supported

Table A.9. Class `Persistence`

Method	Comments	Status
createContainerEntityManagerFactory(persistenceUnit, properties)		Supported
createEntityManagerFactory(persistenceUnitName, properties)		Supported
getProviderUtil()		Supported

A.3.1.2. Query-related Interfaces

Table A.10. Interface `Query`

Method	Comments	Status
executeUpdate()		Not planned
getFirstResult()		Currently, not supported
getFlushMode()		Supported

Method	Comments	Status
getHints()		Currently, not supported
getLockMode()		Supported
getMaxResults()		Currently, not supported
getParameter(int position)		Supported
getParameter(int position, type)		Supported
getParameter(name)		Supported
getParameter(name, type)		Supported
getParameters()		Supported
getParameterValue(int position)		Supported
getParameterValue(Parameter<T> param)		Supported
getParameterValue(name)		Supported
getResultList()		Supported
getSingleResult()		Currently, not supported
isBound(Parameter<?> param)		Supported
setFirstResult(int startPosition)		Currently, not supported
setFlushMode(FlushModeType flushMode)		Supported
setHint(hintName, value)		Currently, not supported
setLockMode(lockMode)		Supported
setMaxResults(int maxResult)		Currently, not supported
setParameter(int position, java.util.Calendar value, temporalType)		Supported
setParameter(int position, java.util.Date value, temporalType)		Supported
setParameter(int position, value)		Supported
setParameter(param, java.util.Calendar value, temporalType)		Supported
setParameter(param, java.util.Date value, temporalType)		Supported
setParameter(param, value)		Supported
setParameter(name, java.util.Calendar value, temporalType)		Supported
setParameter(name, java.util.Date value, temporalType)		Supported

Method	Comments	Status
setParameter(name, value)		Supported
unwrap(cls)		Currently, not supported

Table A.11. Interface `TypedQuery<X>`

Method	Comments	Status
getResultList()		Supported
getSingleResult()		Currently, not supported
setFirstResult(startPosition)		Currently, not supported
setFlushMode(flushMode)		Supported
setHint(hintName, value)		Currently, not supported
setLockMode(lockMode)		Supported
setMaxResults(maxResult)		Currently, not supported
setParameter(position, java.util.Calendar value, temporalType)		Supported
setParameter(position, java.util.Date value, temporalType)		Supported
setParameter(position, value)		Supported
setParameter(param, java.util.Calendar value, temporalType)		Supported
setParameter(param, java.util.Date value, temporalType)		Supported
setParameter(param, value)		Supported
setParameter(name, java.util.Calendar value, temporalType)		Supported
setParameter(name, java.util.Date value, temporalType)		Supported
setParameter(name, value)		Supported

Table A.12. Interface `Parameter<T>`

Method	Comments	Status
getName()		Supported
getParameterType()	only required for Criteria queries	Currently, not supported
getPosition()		Supported

Table A.13. Interface `Tuple`

Method	Comments	Status
get(int i)		Currently, not supported
get(int i, type)		Currently, not supported
get(alias)		Currently, not supported
get(alias, type)		Currently, not supported
get(TupleElement<X> tupleElement)		Currently, not supported
getElements()		Currently, not supported
toArray()		Currently, not supported

Table A.14. Interface `TupleElement<X>`

Method	Comments	Status
getAlias()		Currently, not supported
getJavaType()		Currently, not supported

A.3.2. Package `javax.persistence.spi`

Table A.15. Interface Summary—`javax.persistence.spi`

Interface	Comments	Status
ClassTransformer	A persistence provider supplies an instance of this interface to the PersistenceUnitInfo.addTransformer method.	interface implementation missing (Java EE related)
PersistenceProvider	Interface implemented by the persistence provider.	interface implementation complete
PersistenceProviderResolver	Determine the list of persistence providers available in the runtime environment.	(not to be implemented)
PersistenceUnitInfo	Interface implemented by the container and used by the persistence provider when creating an EntityManagerFactory.	(not to be implemented)
ProviderUtil	Utility interface implemented by the persistence provider.	interface implementation complete

Table A.16. Class Summary—`javax.persistence.spi`

Class	Comments	Status
PersistenceProviderResolverHolder	Holds the global PersistenceProviderResolver instance.	(not to be implemented)

A.3.3. Package `javax.persistence.criteria`

(Currently, package is not supported.)

A.3.4. Package `javax.persistence.metamodel`

(Currently, package is not supported.)

A.3.5. Annotations

A.3.5.1. V/JPA Vendor-specific Annotations

The following are the available vendor-specific annotations for Versant JPA.

Vendor-specific Annotations

`EqualsById`

This directs a collection to use the entity identity, rather than the `equals()` method, for operations that require entity comparisons, e.g., `add()`, `remove()`, etc. Refer to Section 3.7, “Collection Optimization” [p. 47] for information.

`Index`

Class-level annotation to define a single simple or compound index. Refer to Section 3.8, “V/JPA Indexes” [p. 49] for information.

`Indexes`

Class-level annotation to define a multiple simple or compound indexes. Refer to Section 3.8, “V/JPA Indexes” [p. 49] for information.

A.3.5.2. JPA Annotation Support

General remark on mapping

Many of the JPA annotations are used for defining object/relational mapping. The Versant JPA database is an object database. There is no need for this mapping, as V/JPA stores objects directly. Because of this, Versant JPA will ignore annotations which are only important for mapping purposes. These annotations are marked in the following list with the status “Mapping”.

Table A.17. Annotations

Annotation	Comments	Status
Access		Supported
AssociationOverride	Refer to the general remark on mapping	Mapping
AssociationOverrides	Refer to the general remark on mapping	Mapping
AttributeOverride	Refer to the general remark on mapping	Mapping
AttributeOverrides	Refer to the general remark on mapping	Mapping
Basic	V/JPA ignores the hint “optional=false”	Supported
Cacheable		Currently, not supported
CollectionTable	Refer to the general remark on mapping	Mapping
Column	Refer to the general remark on mapping	Mapping
ColumnResult	Not applicable, no native SQL queries for V/JPA	N/A
DiscriminatorColumn	Refer to the general remark on mapping	Mapping
DiscriminatorValue	Refer to the general remark on mapping	Mapping
ElementCollection	Refer to the general remark on mapping	Mapping
Embeddable		Currently, not supported
Embedded		Currently, not supported
EmbeddedId		Currently, not supported
Entity		Supported
EntityListeners		Supported
EntityResult	Not applicable, no native SQL queries for V/JPA	N/A
Enumerated		Supported

Annotation	Comments	Status
ExcludeDefaultListeners		Supported
ExcludeSuperclassListeners		Supported
FieldResult	Not applicable, no native SQL queries for V/JPA	N/A
GeneratedValue	Versant JPA supports only (server-side) generated type `long` Ids (LOIDs - Logical Object IDs)	N/A
Id	Restriction: Id has to be a `long`/`Long` attribute; setting the ID by the user is not permitted	Supported
IdClass	Versant JPA supports only (server-side) generated type `long` Ids (LOIDs - Logical Object IDs)	N/A
Inheritance	Refer to the general remark on mapping	Mapping
JoinColumn	Refer to the general remark on mapping	Mapping
JoinColumns	Refer to the general remark on mapping	Mapping
JoinTable	Refer to the general remark on mapping	Mapping
Lob		Currently, not supported
ManyToMany	Values for Elements “targetEntity” and “mappedBy” will be ignored mappedBy: Refer to the general remark on mapping	Supported
ManyToOne	V/JPA ignores the hint “optional=false” Values for Elements “targetEntity” will be ignored	Supported
MapKey	Refer to the general remark on mapping	Mapping
MapKeyClass		Currently, not supported
MapKeyColumn	Refer to the general remark on mapping	Mapping
MapKeyEnumerated		Supported
MapKeyJoinColumn	Refer to the general remark on mapping	Mapping
MapKeyJoinColumns	Refer to the general remark on mapping	Mapping
MapKeyTemporal	V/JPA does not require this to be defined for `java.util.Date` or `java.util.Calendar` (the latter is currently not supported). All date types are transparently stored in the V/JPA internal date type.	N/A

Annotation	Comments	Status
MappedSuperclass		Currently, not supported
MapsId	Refer to the general remark on mapping	Mapping
NamedNativeQueries	Not applicable, no native SQL queries for V/JPA	N/A
NamedNativeQuery	Not applicable, no native SQL queries for V/JPA	N/A
NamedQueries		Supported
NamedQuery		Supported
OneToMany	Values for Elements “targetEntity”, “mappedBy” and “orphanRemoval” will be ignored mappedBy: Refer to the general remark on mapping	Supported
OneToOne	V/JPA ignores the hint “optional=false” Values for Elements “targetEntity”, “mappedBy” and “orphanRemoval” will be ignored mappedBy: Refer to the general remark on mapping	Supported
OrderBy		Currently, not supported
OrderColumn		Currently, not supported
PersistenceContext	Only needed in Java EE context	Currently, not supported
PersistenceContexts	Only needed in Java EE context	Currently, not supported
PersistenceProperty	Only needed in Java EE context	Currently, not supported
PersistenceUnit		Currently, not supported
PersistenceUnits		Currently, not supported
PostLoad		Supported
PostPersist		Supported
PostRemove		Supported
PostUpdate		Supported
PrePersist		Supported
PreRemove		Supported
PreUpdate		Supported

Annotation	Comments	Status
PrimaryKeyJoinColumn	Refer to the general remark on mapping	Mapping
PrimaryKeyJoinColumns	Refer to the general remark on mapping	Mapping
QueryHint		Currently, not supported
SecondaryTable	Refer to the general remark on mapping	Mapping
SecondaryTables	Refer to the general remark on mapping	Mapping
SequenceGenerator	Versant JPA supports only (server-side) generated type `long` Ids (LOIDs - Logical Object IDs)	N/A
SqlResultSetMapping	Not applicable, no native SQL queries for V/JPA	N/A
SqlResultSetMappings	Not applicable, no native SQL queries for V/JPA	N/A
Table	Refer to the general remark on mapping	Mapping
TableGenerator	Versant JPA supports only (server-side) generated type `long` Ids (LOIDs - Logical Object IDs)	N/A
Temporal	V/JPA does not require this to be defined for `java.util.Date` or `java.util.Calendar` (the latter is currently not supported). All date types are transparently stored in the V/JPA internal date type.	N/A
Transient		Supported
UniqueConstraint	Refer to the general remark on mapping	Mapping
Version	Versant JPA allows up to one version field per entity class. Defining a version attribute simply makes the (always existent) V/JPA internal database timestamp visible to the application (which is of type `int`, 4-byte; the semantic is an incremented version number, not a time value). The application is not allowed to update this. The following types are supported: `int`, `Integer`, `long`, `Long`. Not supported: `short`, `Short`, `Timestamp`.	Supported