JPA Objects 1.0 Users Guide

RDBMS Persistence for Naked Objects 4.0.x

Version 0.1

Preface ..... **v**
1. Introduction **1**
1.1. Key Concepts 1
1.2. Prerequisites 2
2. JPA Objects / Naked Objects Restrictions **3**
2.1. Annotate with `@Entity` xor `@Embeddable` 3
2.2. Annotate properties, not fields 4
2.3. Specify a primary key using `@Id` (not using `@IdClass` or `@EmbeddedId`) 4
2.4. Specify a `@GeneratedType` for the Id 4
2.5. Specify a discriminator 5
2.6. No support for Enumerated Types 6
2.7. No support for Maps 7
2.8. No support for Named Native Query (`@NamedNativeQuery`) 7
2.9. No support to access underlying JPA Persistence Context 7
3. Organizing your Project **9**
3.1. Overview 9
3.2. Update the Parent Module 10
3.3. Create a new Maven submodule for JPA Service (Repository) Implementations 11
3.4. Reference the JPA Applib from the DOM Project 12
3.5. Reference the JPA Service (Repository) Implementations from the fixture Project 12
3.6. Reference the JPA Runtime from the "App" Project 13
3.7. Alternative Approach 13
4. Annotating Domain Objects **15**
4.1. Identifying Entities 15
4.2. Basic properties 16
4.3. Many-to-One 16
4.4. One-to-One 16
4.5. One-to-Many 17
4.6. Many-to-Many 18
4.7. Embedded Objects 18
4.8. Inheritance Hierarchies 19
4.9. Polymorphic Relationships 20
5. Implementing Repositories **23**
5.1. Background 23
5.2. Annotating classes with `@NamedQuery` 25
5.3. Repository Implementation 25
5.4. Registering Repositories 26
5.5. Restrictions 26
6. Supporting Custom Value Types **27**
6.1. Naked Objects AppLib Value Types 27
6.2. Custom Value Types 29
7. Deploying JPA Objects **33**
7.1. Configure Naked Objects 33
7.2. Configure Hibernate 34
7.3. Run the Schema Manager 34
7.4. Run the Fixture Manager 35
8. Other Topics **37**
8.1. Optimistic Locking 37

8.2. Lifecycle Listeners 38
8.3. Persistence by Reachability (Cascading) 39
8.4. Lazy Loading 40
8.5. Common Properties (`@MappedSuperclass`) 41
8.6. Retrieving Collections in Order 41
8.7. Transient (non-persisted) Properties 42
8.8. Handling Exceptions 42
A. Using the Maven Archetype 43
A.1. Obtain the Claims Application 43
A.2. Set up a Relational Database 43
A.3. Run the JPA Archetype 44
A.4. Update the Existing Project's Classpath 45
A.5. Update the Configuration 46
A.6. Update the Domain Classes 47
A.7. Build the Schema 47
A.8. Install the Fixtures 48
A.9. Run the Application 48
B. Annotations Reference 49
B.1. javax.persistence 49
B.2. org.hibernate.annotations 52

Preface

JPA Objects is a sister project to Naked Objects, providing an implementation of an object store to allow Naked Objects domain models to be persisted to an RDBMS. As you might imagine from the name, the domain objects are annotated using JPA annotations (`javax.jpa.Entity` and so on). Hibernate is used as the underlying JPA provider.

This user guide describes how to annotate your domain objects for use by JPA Objects, and how to configure Naked Objects to use the objectstore provided by JPA Objects. It also describes how to use the Maven archetype provided to ease development.

JPA Objects is hosted on SourceForge, and is licensed under Apache Software License v2. Naked Objects is also hosted on SourceForge, and is also licensed under Apache Software License v2.

Chapter 1

Introduction

The Naked Objects framework provides several extension points, one of which is the Object Store API used to persist domain objects. JPA Objects provides an implementation of this API to allow Naked Objects domain models to be persisted to an RDBMS. Note that the object store is also sometimes called the "persistor". For the purpose of this document the two are interchangeable[1].

In fact, *JPA Objects* also implements another Naked Objects API, namely the reflector. This is the component that is used to build up the metamodel. *JPA Objects* provides an extended version of the standard reflector, allowing selected JPA annotations to be identified as declarative business rules. This also allows *JPA Objects* to apply some validations on these annotations.

The project also provides a Maven archetype that includes a set of annotated classes and repository implementations for the example 'claims' application that ships with Naked Objects. A run through of using this archetype is given in Appendix A, *Using the Maven Archetype*.

1.1. Key Concepts

Annotating Domain Classes

The most notable aspect of using *JPA Objects* is that we annotate our domain classes using the JPA annotations (`javax.jpa.Entity` and so on). Because JPA 1.0 does not capture enough semantics for our purposes (eg there is no support for polymorphic "any" relationships), we also use Hibernate's annotations in some circumstances, exposing the fact that Hibernate is the underlying JPA provider. We hope to remove this dependency on Hibernate in a future release when we migrate to JPA 2.0.

Repositories Implementations

Implementations of repositories that are suitable for prototyping (that is, as used by in-memory object store) are naive, because they iterate over all instances. As such, they not suitable for use by the *JPA*

[1] In fact, it is the persistor API that JPA Objects actually implements. Under client/server remoting there is an alternative persistor API that abstracts away the network.

Objects; doing so would be equivalent to performing a `select * from some_table` - with no where clause - and then filtering client-side in Java.

Therefore, using *JPA Objects* requires us to provide implementations of repository interfaces. We use named queries (`@javax.jpa.NamedQuery`) to simplify this task.

The JPA Objects AppLib

Like Naked Objects itself, *JPA Objects* also provides an application library (or applib). And the intent is the same: to minimize the coupling from your domain objects to the framework.

In the current release of *JPA Objects* the only classes in the applib are adapters to allow Naked Objects value types to be persisted as user-defined types. This is discussed further in Chapter 6, *Supporting Custom Value Types*.

Bootstrapping

JPA Objects identifies the set of entities that make up the domain model by walking the graph from the repositories. These are registered as services in Naked Objects' `nakedobjects.properties` configuration file.

> **Important**
>
> If you have inheritance hierarchies then it may be necessary to create dummy actions on your repositories so that all concrete subclasses are registered. The subclasses can appear either as parameters or as return types, and the action can be annotated as `@Hidden` so that it does not appear in the user interface.

In addition, `nakedobjects.properties` is typically used to specify the persistor implementation, ie JPA Objects' own persistor.

There are further details on configuring Naked Objects and *JPA Objects* in Chapter 7, *Deploying JPA Objects*.

1.2. Prerequisites

JPA Objects is intended to be used with Maven-based projects (hence the provision of the Maven archetype). Prerequisites are therefore Maven2 and (highly recommended) an IDE with Maven support. We use Eclipse + the m2eclipse plugin; both NetBeans and IntelliJ have built-in support for Maven.

Chapter 2

JPA Objects / Naked Objects Restrictions

The intent of *JPA Objects* is to allow you to exploit the knowledge you (may already) have of JPA to set up persistence for a Naked Objects applications. As such, most of the annotations of JPA can be used "in the regular way". That said, *JPA Objects* does not support a number of annotations, either:

- by design, to enforce "best practice" for domain-driven applications;
- by necessity, because Naked Objects itself does not support the annotation;
- by happenstance; because *JPA Objects* implementation happens not to (though it could in the future).

Those in the first category are enforced through validation provided by the metamodel (provided by an implementation of Naked Objects' `org.nakedobjects.metamodel.specloader.validator.MetaModelValidator` interface).

This chapter describes these restrictions.

2.1. Annotate with `@Entity` xor `@Embeddable`

Most domain classes will be annotated either as `@Entity` or, if an aggregated object, then as `@Embeddable`. The principle exceptions to this will be classes that factor out common properties (see Section 8.5, “Common Properties (`@MappedSuperclass`)”) and also value types (see Chapter 6, *Supporting Custom Value Types*).

It doesn't make sense for a domain class to be annotated using both `@Entity` and `@Embeddable`. *JPA Objects*' metamodel validator will therefore reject any domain model (that is, Naked Objects will fail to boot) where a domain class is annotated with both.

JPA Objects also ensures that at least one classes is loaded annotated with `@Entity`.

2.2. Annotate properties, not fields

Although JPA itself allows either fields or properties to be annotated, *JPA Objects* requires that only properties are annotated. This is because the Naked Objects metamodel is built up from domain classes by identifying methods only; fields are never used.

2.3. Specify a primary key using `@Id` (not using `@IdClass` or `@EmbeddedId`)

JPA Objects requires that all entities have an `@Id` property (or inherit one from a superclass). If this isn't the case then JPA Objects' metamodel validator will throw an exception, and the application won't boot.

JPA Objects currently does *not* support composite primary keys. Specifically, the `@IdClass` annotation (which specifies a composite primary key class mapped to multiple properties of the entity) is not allowed. The `@EmbeddedId` annotation (which is applied to a persistent property of an entity class or mapped superclass to denote a composite primary key that is an embeddable class) is also not allowed.

This restriction is by design. Coupled with discriminators, *JPA Objects* can guarantee that every entity instance can be identified in a standard fashion. See Section 2.5, “Specify a discriminator” for further discussion.

In addition, the `@Id` property should be a wrapper class, eg `java.lang.Integer` rather than `int`. This allows the JPA provider (Hibernate) to distinguish between transient and persisted objects if no `@Version` property has been specified; transient objects have a `null` id.

2.4. Specify a `@GeneratedType` for the Id

The `@Id` property should be automated populated by the database, rather than by Java code. What this means that a `@GeneratedValue` should be specified.

> ### Note
>
> This restriction may be lifted in future versions of JPA Objects.

For example:

```
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import org.nakedobjects.applib.annotation.Hidden;
...

public class Customer {

    // {{ Id
    private Long id;
    @Hidden
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE)
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
```

```
        this.id = id;
    }
    // }}

    ...
}
```

Because the Id key is automatically generated it generally has no meaning to the end-user, and therefore should also be annotated as `@Hidden`. This isn't a requirement of *JPA Objects*, however.

The value of the `@GeneratedValue`'s strategy is a `GenerationType`:

- `TABLE`

  Indicates that the persistence provider must assign primary keys for the entity using an underlying database table to ensure uniqueness.

  For this generation type, the `@TableGenerator` can also be specified. This annotation defines a primary key generator that may be referenced by name when a generator element is specified for the GeneratedValue annotation. A table generator may be specified on the entity class or on the primary key field or property. The scope of the generator name is global to the persistence unit (across all generator types).

- `SEQUENCE`

  Indicates that the persistence provider must assign primary keys for the entity using database sequence column.

  For this generation type, the `@SequenceGenerator` can also be specified. This annotation defines a primary key generator that may be referenced by name when a generator element is specified for the GeneratedValue annotation. A sequence generator may be specified on the entity class or on the primary key field or property. The scope of the generator name is global to the persistence unit (across all generator types).

- `IDENTITY`

  Indicates that the persistence provider must assign primary keys for the entity using database identity column.

- `AUTO`

  Indicates that the persistence provider should pick an appropriate strategy for the particular database. The `AUTO` generation strategy may expect a database resource to exist, or it may attempt to create one. A vendor may provide documentation on how to create such resources in the event that it does not support schema generation or cannot create the schema resource at runtime.

To capture the ID for use within the domain object, see Section 8.2, “Lifecycle Listeners”.

2.5. Specify a discriminator

JPA Objects requires that all entities are annotated with `@DiscriminatorValue`. The *JPA Objects*' metamodel validator will ensure this is unique (though if it isn't, then the JPA provider itself - ie Hibernate - would also throw up an exception).

In "standard" JPA the `@DiscriminatorValue` is only used to distinguish between concrete subclasses within inheritance hierarchies. JPA Objects makes this a mandatory requirement for all entities so that every entity instance can be identified using the (discriminator, id) tuple.

This tuple is valuable for two reasons:

- from a persistence viewpoint this tuple can be reused for polymorphic relations, that is, as defined using `@Any` or `@ManyToAny`.

  Naked Objects applications tend to be rather "purer" object models than domain models you might have used in non-NO applications, and - following SOLID principles - are likely to use interfaces to decouple classes in different modules. This does cause us to hit the object/relational mismatch though: a decoupled object model cannot rely on an RDBMS to enforce referential integrity, hence the use of polymorphic relations. See Section 2.5, “Specify a discriminator” for more discussion on this.

- from an integration viewpoint, because this tuple is in effect a URN for each entity within the domain, then it can (in serialized form) be used for interacting to other systems (or bounded contexts, if you are use the domain-driven design jargon).

  For example, a RESTful web service (eg as provided by Restful Objects) can use this URN with the path representing a resource, eg to read an object's property or to invoke an action upon it. Or, a message can be published asynchronously, and the URN be used as a correlation Id for a response message.

The tuple is also used internally by Naked Objects, in the `org.nakedobjects.metamodel.adapter.oid.Oid` interface, used to maintain an identity map of domain objects. *JPA Objects* provides an implementation of this interface, `org.starobjects.jpa.runtime.persistence.oid.JpaOid`, which is precisely this tuple.

Going back to the `@DiscriminatorValue`, the recommended length is 3 or 4 characters. Putting this together with the `@Id`, we get something like:

```
@Entity
@DiscriminatorValue("CUS")
public class Customer {

    private Integer id;
    @Id
    public Integer getId() { return id; }
    private void setId(Integer id) { this.id = id }

    ...
}
```

So Customer with id=12345 would have a URN ("CUS", 12345). The serialized form of this (as provided by `JpaOid`) is "CUS|12345".

See Section 4.9, “Polymorphic Relationships” for further discussion on how this tuple is reused for polymorphic relations.

2.6. No support for Enumerated Types

JPA Objects does not support enumerated types, because Naked Objects itself does not support enumerated types. The JPA `@Enumerated` and `@EnumType` annotations may therefore not be used.

A good workaround is to use regular immutable ("reference data") entities, annotated with the Naked Objects' `@Bounded` annotation. `@Bounded` here means that there is a bounded, finite, set of instances; you'll find that Naked Objects viewers will provide drop-down list boxes for these classes.

2.7. No support for Maps

JPA Objects does not support maps, only lists. This is because Naked Objects itself does not support maps. The JPA `@MapKey` annotation may therefore not be used.

A workaround is for the entity to provide an action that runs a repository query to locate the required object. One of the properties of the associated class would act as its key, but this fact would not be exposed in the domain model, only in the database schema. Alternatively, your domain object can build a transient map from a persisted collection.

2.8. No support for Named Native Query (`@NamedNativeQuery`)

JPA Objects does not currently support named native queries (the `@NamedNativeQuery` annotation), only named queries (`@NamedQuery`). See Chapter 5, *Implementing Repositories* for more details.

2.9. No support to access underlying JPA Persistence Context

JPA Objects does not currently provide any support to access the underlying JPA Persistence Context (`javax.persistence.PersistenceContext`), nor to other abstractions such as the `javax.persistence.EntityManager`, `javax.persistence.EntityManagerFactory`, `javax.persistence.EntityTransaction` or `javax.persistence.Query`.

This also means that the locking modes (per `javax.persistence.LockModeType` and the `EntityManager.lock()` method) cannot currently be specified.

Chapter 3

Organizing your Project

This chapter provides advice on how to organize your project so that you can set up dependencies in Maven. Alternatively, you might want to run the Maven archetype - as described in Appendix A, *Using the Maven Archetype*. This sets up the same general project structure as described in this chapter, though some of the details as to how dependencies are resolved varies from the approach described below. See Section 3.7, “Alternative Approach” for more details.

3.1. Overview

If you ran the Naked Objects archetype then you'll have a Maven parent module with a number of child modules:

```
xxx/pom.xml
    xxx-dom/pom.xml          # domain object model
    xxx-fixture/pom.xml      # fixtures for seeding object store
    xxx-service/pom.xml      # in-memory object store implementations of repositories
    xxx-commandline/pom.xml  # for deploying as a commandline, also for prototyping
    xxx-webapp/pom.xml       # for deploying as a webapp
```

Using *JPA Objects* means writing new repository implementations which will depend on *JPA Objects*. In order to isolate those dependencies, we recommend that you create a new Maven submodule:

```
xxx/pom.xml
    xxx-dom/pom.xml
    xxx-fixture/pom.xml
    xxx-service/pom.xml
    xxx-service-jpa/pom.xml  # JPA object store implementations of repositories
    xxx-commandline/pom.xml
    xxx-webapp/pom.xml
```

In addition, the dom and fixture projects also need updating, the dom project to reference the *JPA Objects*' applib and the fixture project in order to use the new JPA service implementations.

3.2. Update the Parent Module

To start with, we're going to have dependencies on *JPA Objects*, on a JDBC driver, and on SLF4J (used by Hibernate). Add these to a `<properties>` section:

```
<properties>
  <jpaobjects.version>1.0.0</jpaobjects.version>                    <!-- or whatever -->
  <postgresql.jdbc.version>8.3-603.jdbc3</postgresql.jdbc.version>   <!-- eg for PostgreSQL
 JDBC driver -->
  <slf4j.version>1.4.3</slf4j.version>
</properties>
```

Next, in the parent module's `<dependencyManagement>` section, add in entries for:

- for *JPA Objects*:

  ```
  <dependencyManagement>
    <dependencies>
      ...
      <dependency>
        <groupId>org.starobjects.jpa</groupId>
        <artifactId>applib</artifactId>
        <version>${jpaobjects.version}</version>
      </dependency>

      <dependency>
        <groupId>org.starobjects.jpa</groupId>
        <artifactId>runtime</artifactId>
        <version>${jpaobjects.version}</version>
      </dependency>

      <dependency>
        <groupId>org.starobjects.jpa</groupId>
        <artifactId>tools</artifactId>
        <version>${jpaobjects.version}</version>
      </dependency>
      ...
    </dependencies>
  </dependencyManagement>
  ```

  Note that this will transitively bring in Hibernate

- for the JDBC driver:

  ```
  <dependencyManagement>
    <dependencies>
      ...
      <dependency>
        <groupId>postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>${postgresql.jdbc.version}</version>
      </dependency>
      ...
    </dependencies>
  </dependencyManagement>
  ```

- for SLF4J:

  ```
  <dependencyManagement>
    <dependencies>
      ...
      <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>${slf4j.version}</version>
  ```

```
      </dependency>

      <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>${slf4j.version}</version>
      </dependency>

      <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-nop</artifactId>
        <version>${slf4j.version}</version>
        <scope>test</scope>
      </dependency>
      ...
    </dependencies>
  </dependencyManagement>
```

Lastly, add a reference to the new Maven submodule that will hold the JPA repository implementations:

```
<modules>
  <module>dom</module>
  <module>fixture</module>
  <module>service</module>
  <module>service-jpa</module>  <!-- JPA implementations -->
  <module>commandline</module>
  <module>webapp</module>
</modules>
```

3.3. Create a new Maven submodule for JPA Service (Repository) Implementations

The new `xxx-service-jpa` submodule should have a `pom.xml` that looks something like:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <name>JPA Service Implementations</name>
  <artifactId>xxx-service-jpa</artifactId>  <!-- CHANGE -->

  <parent>
    <groupId>yyy</groupId>                  <!-- CHANGE -->
    <artifactId>xxx</artifactId>            <!-- CHANGE -->
    <version>zzz</version>                  <!-- CHANGE -->
  </parent>

  <dependencies>

    <!-- project modules -->
    <dependency>
      <groupId>yyy</groupId>                <!-- CHANGE -->
      <artifactId>xxx-dom</artifactId>      <!-- CHANGE -->
    </dependency>

    <!-- jpa objects -->
    <dependency>
      <groupId>org.starobjects.jpa</groupId>
      <artifactId>runtime</artifactId>
```

```
    </dependency>

    <!-- postgres JDBC driver -->
    <dependency>
      <groupId>postgresql</groupId>
      <artifactId>postgresql</artifactId>
    </dependency>

    <!-- SLF4j binding (used by Hibernate) -->
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
    </dependency>

    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
    </dependency>

    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-nop</artifactId>
      <scope>test</scope>
    </dependency>

  </dependencies>
</project>
```

3.4. Reference the JPA Applib from the DOM Project

In the `xxx-dom` project, reference the *JPA Objects*' applib in the `<dependencies>` section:

```
<dependencies>
  ...
  <dependency>
    <groupId>org.starobjects.jpa</groupId>
    <artifactId>jpa-applib</artifactId>
  </dependency>
  ...
</dependencies>
```

This will transitively bring in the JPA (`javax.persistence`) annotations.

3.5. Reference the JPA Service (Repository) Implementations from the fixture Project

In the xxx-fixture project, reference the JPA services implementations module in the `<dependencies>` section:

```
<dependencies>
  ...
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>xxx-jpa-service</artifactId>
  </dependency>
  ...
</dependencies>
```

TODO: I can't recall why this is needed; hence the lack of commentary here. There's probably a good reason, but it escapes me...

3.6. Reference the JPA Runtime from the "App" Project

Either the commandline or the webapp project, update the `<dependencies>` section to reference the new JPA implementations:

```
<dependencies>
  ...
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>xxx-jpa-service</artifactId>
  </dependency>
  ...
</dependencies>
```

This will transitively bring in the *JPA Objects* runtime libraries.

In addition, you will want to run a couple of tools that are used to deploy the application (see Chapter 7, *Deploying JPA Objects* for more details). The commandline app is probably as good a place as any to reference these tools. Therefore, also add in:

```
<dependencies>
  ...
  <dependency>
    <groupId>org.starobjects.jpa</groupId>
    <artifactId>tools</artifactId>
  </dependency>
  ...
</dependencies>
```

3.7. Alternative Approach

An alternate approach for setting up dependencies (and the one used that is used in the archetype) is to have the `xxx-jpa-service` module inherit from `org.starobjects.jpa:release` rather than the parent module. Or, you could perhaps have the parent module inherit from `org.starobjects.jpa:release`. Doing it this way means that it isn't necessary to add entries to `<dependencyManagement>` because they are inherited. However, Maven2 does only allow a single parent, so this may not be an option for you if you want to use some other POM as your parent.

Chapter 4

Annotating Domain Objects

This chapter provides an overview on how to annotate domain objects using JPA. It isn't comprehensive by any means, but should deal with most of the common cases. Hunt out a book dedicated to JPA persistence (such as like Bauer/King's Java Persistence with Hibernate) for a much more thorough treatment. See also Appendix B, *Annotations Reference* for an overview of the supported annotations.

4.1. Identifying Entities

Domain classes that are persistent entities - which is most of them - should be annotated using `javax.persistence.Entity`. In addition - as described in Chapter 2, *JPA Objects / Naked Objects Restrictions - JPA Objects* requires that all entities are annotated using `javax.persistence.@DiscriminatorValue` and must define a single property to act as the identifier, annotated using `@javax.persistence.Id`. The type of this Id property will depend on the number of instances expected, but will generally be one of `java.lang.Integer`, `java.lang.Long`.

JPA Objects itself does not care how the values for the property are generated. It's usually easiest to get the RDBMS to generate the IDs, using a sequence or identity (or perhaps a table). This is specified using the `@javax.persistence.GeneratedValue` annotation.

```
@Entity
@DiscriminatorValue("CUS")
public class Customer {

    private Integer id;
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public Integer getId() { ... }
    private void setId(Integer id) { ... }

    ...
}
```

Other generation strategies include SEQUENCE, IDENTITY and TABLE. Note that it isn't necessary for the setter to have public visibility.

For very large ranges it is also possible to use `java.lang.BigInteger`. This isn't a datatype supported out-of-the-box by JPA, but Hibernate can be made to support it. See for example this blog post for details on how.

Alternatively, you can take control yourself, and hook into JPA's `@PrePersist` lifecycle method.

4.2. Basic properties

Value properties are mapped in JPA using the `javax.persistence.@Basic` annotation. In Naked Objects' terms, this means properties whose return type is a built-in value type: specifically primitives, `String`, `java.util.Date` (and its subclasses), `java.sql.Timestamp`, `java.math.BigDecimal`, `java.math.BigInteger`. It is also possible to map custom value types (that is, annotated with `org.nakedobjects.applib.annotation.@Value`); see Chapter 6, *Supporting Custom Value Types* for more details.

A property annotated with `@Basic` will correspond to a column in a database table. JPA will map - to an appropriate SQL type - all of the built-in value types supported by Naked Objects. The other types supported by JPA - `java.util.Calendar`, and arrays of primitives and wrappers, and enums - are not supported by Naked Objects as value types.

JPA does also allow a property whose type is serializable to be annotated with `@Basic`. However, it will be stored in the database as a blob. This is therefore not a recommended approach; better is to implement a custom value type.

Properties of type `java.util.Date` can be annotated using `@javax.persistence.Temporal`. This provides a hint as to the SQL type to use: DATE, TIME or TIMESTAMP.

4.3. Many-to-One

The `@javax.persistence.ManyToOne` annotation defines a single-valued association to another entity class that has many-to-one multiplicity. In Naked Objects' terms this is a property whose return type is another entity. In database terms, this is like a foreign key. The `@ManyToOne` annotation goes in the "child" entity, specifically on the property that points to the "parent" entity.

It is not normally necessary to specify the target parent entity explicitly since it can usually be inferred from the type of the object being referenced.

The `@ManyToOne` sometimes combines with the `@OneToMany` annotation in the parent entity, making the association bidirectional. See Section 4.5, “One-to-Many” for more details.

4.4. One-to-One

The `@javax.persistence.OneToOne` annotation defines a single-valued association to another entity that has one-to-one multiplicity. In Naked Objects' terms this is again a property whose return type is another entity.

In fact, this is just a special case of `@ManyToOne`. In the database we again end up with a foreign key in the referencing entity to the referenced entity.

It is not normally necessary to specify the associated referenced entity explicitly since it can usually be inferred from the type of the object being referenced.

4.5. One-to-Many

The `@javax.persistence.OneToMany` annotation defines a many-valued association with one-to-many multiplicity. In Naked Objects' terms this is a collection (or any of its subtypes). As discussed in Section 2.7, “No support for Maps”, Naked Objects does not support `java.util.Maps`.

It's very common for collections relationships to be bidirectional, with that the referenced "child" entity having a corresponding back reference to its owning parent entity using `@ManyToOne`. In this case the foreign key that the `@ManyToOne` requires can also be used to navigate from the parent to the child. We indicate this bidirectionality using the mappedBy attribute. In the parent we have:

```
@Entity
@DiscriminatorValue("ORD")
public class Order {

    private List<OrderItem> items = new ArrayList<OrderItem>;
    @OneToMany(mappedBy="order")
    public List<OrderItem> getItems() { ... }
    private void setItems(List<OrderItem> items) { ... }

    ...
}
```

and in the child we have:

```
@Entity
@DiscriminatorValue("ORI")
public class OrderItem {

    private Order order;
    @ManyToOne
    public Order getOrder() { ... }
    public void setOrder(Order order) { ... }

    ...
}
```

If the collection is defined using generics to specify the element type (as above), the associated entity type need not be specified; otherwise the entity class must be specified.

The above code isn't complete by the way; for bidirectional relationships you'll need to write code to keep the links in sync. The mutual registration pattern for this; see for example Dan's DDD book for more details if you aren't familiar with it.

On occasion you may have a requirement only for the one-to-many collection in the parent, and no back reference from the child back. From an annotations perspective just omit the `mappedBy` attribute. However, it's worth being aware that the database schema will be quite different. Because there is no foreign key in the child table to use, Hibernate will create a link table consisting of the (parent_id, child_id) tuples. Retrieving the children of a collection means Hibernate will need to this query this link table and join across to the child entity table.

4.6. Many-to-Many

The `@javax.persistence.ManyToMany` annotation defines a many-valued association with many-to-many multiplicity. In Naked Objects' terms this means two entities each referencing a collection of the other.

As you may well know, many-to-many associations in a relational database always require a link (or associative table), consisting of the primary keys of each of the entities. Note the similarity to the mapping of a unidirectional `@OneToMany` association (see Section 4.5, “One-to-Many”), which in this light can be thought of as a special case of `@ManyToMany`. Indeed the same annotation elements for the `@OneToMany` annotation apply to the c annotation.

Every many-to-many association has two sides, the owning side and the non-owning, or inverse, side. The join table is specified on the owning side. If the association is bidirectional, either side may be designated as the owning side. If the Collection is defined using generics to specify the element type, the associated entity class does not need to be specified; otherwise it must be specified.

4.7. Embedded Objects

An embedded object is one that is wholly owned by a parent entity, in the UML compositional ("colour in the black diamond") sense. Using domain-driven design terminology it is an aggregated object (with its owning entity the aggregate root).

The `@javax.persistence.Embeddable` annotation defines an embedded object, that is, a class whose instances are stored as an intrinsic part of an owning entity and share the identity of the entity. Note that `@Embeddable` entities must not also be annotated with `@Entity`. They also do not have an `@Id`. That's because each of the persistent properties of the embedded object is mapped to the database table for the entity. JPA restricts the embedded objects to only consisting of `@Basic` (value) properties.

Having defined an embeddable class, the owning entity defines an association using the `@javax.persistence.Embeddable` annotation. In Naked Objects terms this is a property whose return type is the `@Embeddable` class. For example, consider an `Address` class:

```
@Embeddable
public class Address {

    private Integer houseNumber;
    @Basic
    public Integer getHouseNumber() { ... }
    public void setHouseNumber(Integer houseNumber) { ... }

    private String streetName;
    @Basic
    public String getStreetName() { ... }
    public void setStreetName(Integer streetName) { ... }

    ...
}
```

The owning root `Customer` entity might look like:

```
@Entity
@DiscriminatorValue("CUS")
public class Customer {
```

```
    private Integer id;
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public Integer getId() { ... }
    private void setId(Integer id) { ... }

    private Address address;
    @Embedded
    public Address getAddress() { ... }
    ...
}
```

Although JPA defines the `@EmbeddedId` annotation (to indicate a property whose type represents a composite primary key that is an embeddable class), *JPA Objects* does not support this (because in *JPA Objects* all primary keys must be simple, not composite).

4.8. Inheritance Hierarchies

JPA supports (implementation) inheritance using the `@javax.persistence.Inheritance` annotation, and this is fully supported by *JPA Objects*. The annotation is used to determine how to map clases, using the strategy attribute to specify the `InheritanceType` enum:

- `InheritanceType.SINGLE_TABLE`

  A single table per class hierarchy

- `InheritanceType.TABLE_PER_CLASS`

  A table per concrete entity class

- `InheritanceType.JOINED`

  A strategy in which fields that are specific to a subclass are mapped to a separate * table than the fields that are common to the parent class, and a join is performed * to instantiate the subclass.

The `@Inheritance` annotation goes on the superclass.

For the `SINGLE_TABLE` and `JOINED` inheritance mapping strategies a further column is added to the table which is used to specify the concrete subclass that the data held in the table corresponds. For the `JOINED` mapping it also effectively specifies the subclass table to join to. Typically the column name and type is specified explicitly on the superclass using the `@javax.persistance.DiscriminatorColumn` annotation; the `@DiscriminatorValue` annotation is then used on the subclasses to specify the values that go into this column.

As we've already noted though in Section 2.5, “Specify a discriminator”, *JPA Objects* always requires an `@DiscriminatorValue` to be specified. Rather than invent adhoc discriminators for each inheritance hierarchy, *JPA Objects* in effect forces us to standardize the values across the entire domain model.

What this also means is that, if a `@DiscriminatorColumn` has been specified, then the `discriminatorType` (if specified) must be `DiscriminatorType.STRING`. The length should be long enough for the longest discriminator; a length of 3 is generally sufficient.

4.9. Polymorphic Relationships

A polymorphic relationship is one where the referenced type could be any entity. For example, a class that has a property of type java.lang.Object would be polymorphic; this property - if persisted - could point to any object in the database.

More typically though polymorphic relationships arise when we have decoupled two classes so that one references the other through an interface. Since there could be many implementors of the interface, we again have a polymorphic relationship.

This is one area where we hit a limitation of relational databases. A foreign key requires a primary key to reference. However, it isn't practical to map every domain class as a member of an inheritance hierarchy with `java.lang.Object` as its root. Consider: if we used the `JOINED` strategy, every retrieval would involve a query against the `object` table and then down to the subclass' table.

Instead we abandon the use of referential integrity in the database. Instead, we just store the object reference in two parts: by identifying the concrete class of the referenced object, and the identifier of the referenced object. Note how this is the same tuple discussed in Section 2.5, “Specify a discriminator”.

The approach mandated by *JPA Objects* unifies two relating requirements: that of discriminating concrete classes within an inheritance hierarchy, and of discriminating instances throughout the entire database.

In practical terms, we map a polymorphic relationship using the `@org,hibernate.annotations.Any` for a reference to a single object, and `@javax.persistence.ManyToAny` annotation for a colleciton of references. For example, suppose we have a `Vehicle` can be owned either by an individual `Person` or be owned by a `Company` (such as a fleet car). In the domain model both `Person` and `Company` implement `VehicleOwner`:

```
@Entity
@DiscriminatorValue("PRS")
public class Person implements VehicleOwner { ... }
```

and

```
@Entity
@DiscriminatorValue("CPY")
public class Company implements VehicleOwner { ... }
```

In the `Vehicle` class we use `@Any` along with `@AnyMetaDef` to identify these concrete implementations:

```
@Entity
@DiscriminatorValue("VEH")
public class Vehicle {
    ...

    @Any(
       metaColumn=@Column(name="owner_type" , length=3),
       fetch=FetchType.LAZY
    )
    @AnyMetaDef(
       idType="long", metaType="string" ,
       metaValues={
          @MetaValue(targetEntity=Person.class, value="PRS" ),
          @MetaValue(targetEntity=Company.class, value="CPY" )
       }
    )
    @JoinColumn(name="owner_id" )
    public VehicleOwner getOwner() { ... }
```

```
    public void setOwner(VehicleOwner owner) { ... }

    ...
}
```

In the `vehicle` table (for `Vehicle` class) this will give rise to a two-part tuple (owner_type, owner_id), that collectively identifies the object that is acting as a `VehicleOwner`. The owner_type takes the value "PRS" for `Person`, in which case the owner_id contains a person Id from the `person` table; if it takes "CPY" then owner_id contains the company Id from the `company` table. Note that this tuple is, in effect, the `JpaOid` for the object (again, see Section 2.5, “Specify a discriminator”).

The `@org.hibernate.annotations.ManyToAny` similarly has a slew of annotions. If for example a `Vehicle` could have multiple owners, we would have:

```
@Entity
@DiscriminatorValue("VEH")
public class Vehicle {
    ...

    @ManyToAny(
        metaColumn = @Column( name = "owner_type" )
    )
    @AnyMetaDef(
        idType = "integer", metaType = "string",
        metaValues = {
            @MetaValue( targetEntity = Person.class, value="PRS" ),
            @MetaValue( targetEntity = Company.class, value="CPY" )
        }
    )
    @Cascade( { org.hibernate.annotations.CascadeType.ALL } )
    @JoinTable(
        name = "vehicle_owners",
        joinColumns = @JoinColumn( name = "vehicle_id" ),
        inverseJoinColumns = @JoinColumn( name = "owner_id" )
    )
    public List<Property> getOwners() { ... }
    private void setOwners(List<VehicleOwner> owners) { ... }

    ...
}
```

This would give rise to a `vehicle_owners` link table, whose columns would be (vehicle_id, owner_type, owner_id). The vehicle_id identifies the `Vehicle` whose owners we are interested in; the owner_type, owner_id together identify the owner (either `Person` or `Company`).

Chapter 5

Implementing Repositories

Repositories are used to obtain references to (persisted) entities. The implementation of these repositories depends on the object store in use; those for the in-memory objectstore won't be suitable for use with *JPA Objects* because they naively iterate over all instances. You'll therefore need to re-implement your repositories. And for this reason it's best practice to define an interface for your repositories (or indeed any domain service). You can then switch in different implementations just by editing `nakedobjects.properties` (see Section 7.1, “Configure Naked Objects”).

It's good practice to put your JPA implementations into a separate Maven module. That way, you can isolate the dependencies on JPA Objects itself just to the code that needs it. The Maven archetype that comes with JPA Objects is designed to work this way; see Appendix A, *Using the Maven Archetype* for more details.

To understand what goes into the *JPA Objects* repository implementations, let's start with a little background.

5.1. Background

Naked Objects provides generic repository support through its convenience adapter classes in the Naked Objects applib. Although not mandatory, it's easiest to have domain objects inherit from `org.nakedobjects.applib.AbstractDomainObject` and repositories - while prototyping at least - inherit from `org.nakedobjects.applib.AbstractFactoryAndRepository`. These both inherit from `org.nakedobjects.applib.AbstractContainedObject`, which in turn provides a Container property to allow the `DomainObjectContainer` to be injected. The `DomainObjectContainer` in effect *is* the generic repository.

Also in `AbstractContainedObject` are some convenience methods for searching, each of which just delegates to similarly named methods in `DomainObjectContainer`. First and most straightforwardly we can request all instances of a given entity class:

- `allInstances(Class<T> ofClass): List<T>`

But it is also possible to request all instances matching either a condition. This can be expressed in one of four ways:

- `allMatches(Class<T> ofClass, String title): List<T>`

  With this method we search by title. This makes sense for those classes where the title is relatively small and known to be unique. However, Naked Objects does not itself mandate unique titles; they are just labels that are unique "enough" for the objects being viewed by the end-user. This is also not a good option if the title changes, eg reflecting the state of the object

- `allMatches(Class<T> ofClass, T pattern): List<T>`

  Here we search using a pattern object, sometimes called query-by-example. Only instances whose values match those of the (set) properties of the pattern instance are returned.

- `allMatches(Class<T> ofClass, Filter<T> filter): List<T>`

  The `Filter<T>` interface (defined in the Naked Objects applib) acts as a predicate, so the method returns only those instances that meet the filter.

- `allMatches(Class<T> ofClass, Query<T> query): List<T>`

  This method is similar to the one for filtering, returning those instances that meet the query specification. (Again, `Query<T>` is defined in the Naked Objects applib).

There are similar methods to find the first instance:

- `firstMatch(Class<T> ofClass, String title): T`
- `firstMatch(Class<T> ofClass, T pattern): T`
- `firstMatch(Class<T> ofClass, Filter<T> filter): T`
- `firstMatch(Class<T> ofClass, Query<T> query): T`

There are also methods to find the first and only (unique) instance:

- `uniqueMatch(Class<T> ofClass, String title): T`
- `uniqueMatch(Class<T> ofClass, T pattern): T`
- `uniqueMatch(Class<T> ofClass, Filter<T> filter): T`
- `uniqueMatch(Class<T> ofClass, Query<T> query): T`

The difference between `Filter<T>` and `Query<T>` in these methods comes down to where the predicate is evaluated. With `Filter<T>`, the evaluation is in Java. What that means is that all instances are returned from the object store. In contrast `Query<T>` the evaluation is performed by the object store implementation. For *JPA Objects*, it ultimately corresponds to the "WHERE" clause in a SQL SELECT statement.

For prototyping you'll find that the first three of these are supported by both the in-memory object store and also by the XML object store. Indeed, every object store is likely to support these, because all they simply require that the object store can return all instances of a class. However, the version accepting `Query<T>` is different; because the `Query<T>` is evaluated in the object store, its implementation will in general be specific to the object store.

That said, there is in fact a default implementation of `Query<T>`, namely `QueryDefault<T>` (in the Naked Objects applib, again). This implementation simply holds onto a query name and a set of parameter/argument pairs.

The in-memory object store and XML object store do not support Query<T> in any way. *JPA Objects* does support Query<T>, through QueryDefault<T>. And this is what we use in our repository implementations.

5.2. Annotating classes with @NamedQuery

JPA already defines a mechanism for defining queries, with the @javax.persistence.NamedQuery and @javax.persistence.NamedQueries annotations. In theory these can be annotated on any class, but by convention they go on the class being returned.

For example, we might want to search for Customers by their id, or by their surname (family name):

```
@NamedQueries({
    @NamedQuery(name="findCustomerById", query="from Customer where id=:id"),
    @NamedQuery(name="findCustomersBySurname", query="from Customer where surname=:surname"),
})
@Entity
@DiscriminatorValue("CUS")
public class Customer {

    ...
    @Id
    public Integer getId() { ... }

    ...
    @Basic
    public String getFamilyName() { ... }

    ...
}
```

These named queries are then referenced in the repository implementations, covered next (Section 5.3, “Repository Implementation”).

5.3. Repository Implementation

The repository implementations bring together the JPA @NamedQuery annotation (see Section 5.2, “Annotating classes with @NamedQuery”) along with the QueryDefault<T> class (introduced in Section 5.1, “Background”).

For example, a CustomerRepository implementation might be:

```
public class CustomerRepositoryJpa implements CustomerRepository {

    public List<Customer> findCustomerById(Integer id) {
        return firstMatch(
            QueryDefault.create(
                "findCustomerById",
                "id", id);
    }

    public List<Customer> findCustomersBySurname(String surname) {
        return allMatches(
            QueryDefault.create(
               "findCustomersBySurname",
               "surname", surname
```

```
            ));
    }

    ...
}
```

The first argument for `QueryDefault.create()` should be the name of the query (it doesn't need to be the same as the method name, but that's a sensible convention to follow). The remaining arguments go in pairs, alternating between the parameter name and the argument value. For example, the ":id" in the `@NamedQuery` annotation corresponds to the second argument "id" in the call to `QueryDefault.create()`.

5.4. Registering Repositories

Once the repositories are written, they should be registered in `nakedobjects.properties`. See Section 7.1, “Configure Naked Objects” for more details.

5.5. Restrictions

JPA Objects currently does not support either native named queries (see Section 2.8, “No support for Named Native Query (`@NamedNativeQuery`)”) nor the creation of queries using Hibernate criteria (see Section 2.9, “No support to access underlying JPA Persistence Context”). These limitations will be addressed in a future version (most likely with an upgrade to JPA 2.0).

Chapter 6

Supporting Custom Value Types

While Naked Objects supports a similar set of value types to JPA, it also allows custom value types to be defined using the `@Value` annotation. In addition to the built-in and custom value types, Naked Objects also has its own set of its own value types, such as `Money` and `Percentage` . These reside in the Naked Objects applib, in the `org.nakedobjects.applib.value` package.

So long as a value type is serializable, then JPA will be able to save the value in the database. However, the value will be stored as a blob, meaning for example it won't be possible to query on it within repositories. And if the value type is not serializable, then JPA will not be able to save it at all.

Hibernate provides a solution to this by allowing us to write (what it calls) user-defined types, through either the `org.hibernate.usertype.UserType` interface (for values that are persisted in a single column) or the `org.hiberate.usertype.CompositeUserType` (for more complex values that are persisted to multiple columns). These are analogous to the `org.nakedobjects.applib.adapters.ValueSemanticsProviders` that accompany Naked Objects' own `@Value` annotation: a `ValueSemanticsProvider` instructs Naked Objects viewers how to interact with a custom value, while a `UserType` instructs Hibernate how to persist/retrieve a value into a database table.

The *JPA Objects*' Application Library (or AppLib, see the section called “The JPA Objects AppLib”) defines several convenience superclasses to help write these `UserType` implementations. There are also out-of-the-box implementations to support Naked Objects' own value types (such as `org.nakedobjects.applib.value.Money`). Let's start off with these.

6.1. Naked Objects AppLib Value Types

Suppose we want to capture a `Person`'s favorite colour, and choose to do this using Naked Objects built-in `org.nakedobjects.applib.value.Color` value type:

```
import org.nakedobjects.applib.value.Color;

@Entity
@DiscriminatorValue("PRS")
public class Person {
```

```
    ...
    public Color getFavoriteColor() { ... }
    ...
}
```

Only a single column is needed to encode the value, so the *JPA Objects* applib provides `org.starobjects.jpa.applib.usertypes.ColorType`, a `UserType` implementation to persist `Color`s. Here's how we use it:

```
import org.nakedobjects.applib.value.Color;
import org.starobjects.jpa.applib.usertypes.ColorType;
import org.hibernate.annotations.Type;
import org.hibernate.annotations.TypeDef;
import org.hibernate.annotations.TypeDefs;
...

@Entity
@DiscriminatorValue("PRS")
@TypeDefs({
    @TypeDef(name="nofcolor", typeClass=ColorType.class)
})
public class Person {

    ...
    @Type(type="nofcolor")
    public Color getFavoriteColor() { ... }
    ...
}
```

This sets up the "nofcolor" as an alias to the `ColorType`, and then says to use this alias for the `favoriteColor` property.

We'll have a look at the `ColorType` implementation in the below; if you skip ahead you'll see that the value stored is in fact an integer (corresponding to `Color#intValue()` method and the `#Color(int)` constructor).

Most of the other Naked Objects value types are also mapped using simple `UserType`s. The exception is `Money`, which is mapped as a composite (a string column for the currency iso code, and a numeric amount). For example, if there is also a property of type `Money` for our fictitious `Person` class, then we would have:

```
import org.nakedobjects.applib.value.Money;
import org.starobjects.jpa.applib.usertypes.MoneyType;
import org.hibernate.annotations.Type;
import org.hibernate.annotations.TypeDef;
import org.hibernate.annotations.TypeDefs;
...

@Entity
@DiscriminatorValue("PRS")
@TypeDefs({
    @TypeDef(name="nofcolor", typeClass=ColorType.class),
    @TypeDef(name="nofmoney", typeClass=MoneyType.class)
})
public class Person {

    ...
    @Type(type="nofmoney")
    public Money getSalary() { ... }
    ...
}
```

Now we've seen how to use *JPA Objects*' predefined user types, let's see how to write them for our own value types.

6.2. Custom Value Types

Simple Types (`ImmutableUserType`)

Simple types are those that can be mapped using a single column (such as `Color`, above). For these we subclass from `org.starobjects.jpa.applib.usertypes.ImmutableUserType`. Here's the implementation of `ColorType`:

```
package org.starobjects.jpa.applib.usertypes;

import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import org.hibernate.Hibernate;
import org.nakedobjects.applib.value.Color;

public class ColorType extends ImmutableUserType {

    public Object nullSafeGet(
            final ResultSet rs,
            final String[] names,
            final Object owner) throws SQLException {
        final int color = rs.getInt(names[0]);
        if (rs.wasNull()) {
            return null;
        }
        return new Color(color);
    }

    public void nullSafeSet(
            final PreparedStatement st,
            final Object value,
            final int index) throws SQLException {
        if (value == null) {
            st.setNull(index, Hibernate.INTEGER.sqlType());
        } else {
            st.setInt(index, ((Color) value).intValue());
        }
    }

    public Class<Color> returnedClass() {
        return Color.class;
    }

    public int[] sqlTypes() {
        return new int[] { Hibernate.INTEGER.sqlType() };
    }
}
```

The `nullSafeGet()` method is used to extract the value from the SQL `ResultSet` and instantiate the `Color` object. Conversely the `nullSafeSet()` method is used to read data from the provided Color and set up the SQL `PreparedStatement` so the value can be inserted or updated. The other two methods describe the structure of the data being read/written.

Composite Types (`ImmutableCompositeUserType`)

Composite types are those that are mapped using more than one column (such as `Money`, above). For these we subclass from `org.starobjects.jpa.applib.usertypes.ImmutableCompositeType`. Here's the implementation of `MoneyType`:

```
package org.starobjects.jpa.applib.usertypes;

import java.math.BigDecimal;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import org.hibernate.Hibernate;
import org.hibernate.engine.SessionImplementor;
import org.hibernate.type.Type;
import org.nakedobjects.applib.value.Money;

public class MoneyType extends ImmutableCompositeType {

    public Class<Money> returnedClass() {
        return Money.class;
    }

    public Object nullSafeGet(
            final ResultSet resultSet,
            final String[] names,
            final SessionImplementor session,
            final Object owner) throws SQLException {
        final BigDecimal amount = resultSet.getBigDecimal(names[0]);
        if (resultSet.wasNull()) {
            return null;
        }
        final String currency = resultSet.getString(names[1]);
        return new Money(amount.doubleValue(), currency);
    }

    public void nullSafeSet(
            final PreparedStatement statement,
            final Object value,
            final int index,
            final SessionImplementor session) throws SQLException {
        if (value == null) {
            statement.setNull(index, Hibernate.BIG_DECIMAL.sqlType());
            statement.setNull(index + 1, Hibernate.STRING.sqlType());
        } else {
            final Money amount = (Money) value;
            statement.setBigDecimal(index, amount.getAmount());
            statement.setString(index + 1, amount.getCurrency());
        }
    }

    public String[] getPropertyNames() {
        return new String[] { "amount", "currency" };
    }

    public Type[] getPropertyTypes() {
        return new Type[] { Hibernate.BIG_DECIMAL, Hibernate.STRING };
    }

    public Object getPropertyValue(final Object component, final int property) {
        final Money monetaryAmount = (Money) component;
        if (property == 0) {
            return monetaryAmount.getAmount();
```

```
        } else {
            return monetaryAmount.getCurrency();
        }
    }

    public void setPropertyValue(final Object component, final int property, final Object
 value) {
        throw new UnsupportedOperationException("Money is immutable");
    }
}
```

This works in broadly the same way, with `nullSafeGet()` and `nullSafeSet()` used to read values from SQL/write values to SQL. The `getPropertyNames()` and `getPropertyTypes()` again describe the structure of the value. The `getPropertyValue()` allow specific properties of the value (such as the `Money`'s currency) to be read; like reading a single column in the database. The `setPropertyValue()` meanwhile should always thrown an exception because value types should be immutable (replaced in their entirety rather than modified in-situ).

Chapter 7
Deploying JPA Objects

Once you have annotated your domain objects (Chapter 4, *Annotating Domain Objects*), re-implemented your repositories for JPA (Chapter 5, *Implementing Repositories*) and - if necessary - written `UserTypes` for any custom value types (Chapter 6, *Supporting Custom Value Types*), then you are ready to deploy your application. This consists of configuring both Naked Objects (telling it to use *JPA Objects*), and then configuring Hibernate as the underlying JPA provider (telling it where the database is). After that, you (or your DBA) is ready to install the database schema and (if necessary) to populate the database using the fixtures you'll have developed for prototyping.

7.1. Configure Naked Objects

The Naked Objects framework is configured using the `nakedobjects.properties` file. There are three main property settings that need specifying:

- specify the *JPA Objects* persistor

  ```
  nakedobjects.persistor=org.starobjects.jpa.runtime.persistence.JpaPersistenceMechanismInstaller
  ```

  This specifies the class to acts as the "installer" (basically, a factory) for the persistence mechanism.

- specify the *JPA Objects* reflector

  ```
  nakedobjects.reflector=org.starobjects.jpa.metamodel.specloader.JpaJavaReflectorInstaller
  ```

  This specifies the class to acts as the "installer" (basically, a factory) for an updated version of the reflector; the component that builds the Naked Objects metamodel. This is needed so that *JPA Objects* can interrogate the Naked Objects metamodel for JPA annotations when persisting domain objects.

- disable bytecode providers

  ```
  nakedobjects.reflector.class-
  substitutor=org.nakedobjects.metamodel.specloader.classsubstitutor.ClassSubstitutorIdentity
  nakedobjects.persistor.object-
  factory=org.nakedobjects.runtime.persistence.objectfactory.ObjectFactoryBasic
  ```

  The first of these components is the bytecode provider, the second is a related component that is normally used to filter out the bytecode-generated subclasses.

- register repositories

  The JPA repository implementations should be specified for nakedobjects.services key. For example:

```
nakedobjects.services.prefix = org.starobjects.jpa.testapp
nakedobjects.services = service.employee.EmployeeRepositoryJpa,
 service.claim.ClaimRepositoryJpa
```

7.2. Configure Hibernate

JPA Objects (currently) uses Hibernate's native configuration mechanism, `hibernate.cfg.xml`, to specify the location of the database. Before you wade in, you can confirm the settings and that you have connectivity using the very simple `org.starobjects.jpa.tools.Ping` utility. This takes the following arguments:

- -v (value of connection.driver_class)
- -u (value of connection.url)
- -U (value of connection.username)
- -P (value of connection.password)

When you're happy this is okay, then go ahead and update `hibernate.cfg.xml`.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!--  debugging -->
    <property name="show_sql">false</property>
    <property name="bytecode.use_reflection_optimizer">true</property>
    <property name="hibernate.query.substitutions">true 1, false 0</property>

    <property name="connection.driver_class">org.postgresql.Driver</property>
    <property name="connection.username">myapp</property>
    <property name="connection.password">myapp_pwd</property>
    <property name="connection.url">jdbc:postgresql://localhost/myapp_db</property>
    <property name="dialect">org.hibernate.dialect.PostgreSQLDialect</property>
  </session-factory>
</hibernate-configuration>
```

This file should reside in the root directory of the classpath (eg in `src/main/resources`). A good place for it is alongside your JPA repository implementations.

7.3. Run the Schema Manager

To create the database schema (using `create table` SQL commands) we can use the schema manager tool (`org.starobjects.jpa.tools.SchemaManager`) that comes with *JPA Objects*. This is really just a wrapper around Hibernate's equivalent **hbm2ddl** tool. It takes the following arguments:

- -g <config directory containing nakedobjects.properties>

  Depending on where you run the utility from, this is typically something like `../commandline/config`. This is read for the repositories, from which the Naked Objects and Hibernate metamodels are built up. After that, the hbm2ddl utility is run.

- -c

  create database, no initial drop. If omitted will do a drop followed by create (ie a recreate)

- -d

  drop database, no initial create. If omitted will do a drop followed by create (ie a recreate)

- -x

  execute. If omitted, then does nothing, ie just previews the commands by writing to stdout

7.4. Run the Fixture Manager

The way that different object stores handle fixtures depends on their implementation. The in-memory object store always loads fixtures, the XML object store loads fixtures only if they haven't been run but (by detecting the `xml` directory that it creates) ignores them second time around.

JPA Objects though is more conservative: it *never* loads up fixtures in runtime. The rationale is that seeding the database with initial data is an activity that you/your DBA should have visibility over.

Most likely though, you will have some fixtures (those for immutable reference data/static data objects) which should be loaded in. To support this, you can run the fixture manager tool (`org.starobjects.jpa.tools.FixturesManager`) that comes with *JPA Objects*. It takes the following arguments:

- -g <config directory containing nakedobjects.properties>

  Depending on where you run the utility from, this is typically something like `../commandline/config`. This is used to bootstrap Naked Objects and to run in the fixtures. (It sets a special flag which instructs *JPA Objects* object store to read in the fixtures.

After this, you should have a populated database. If you mess up, you can always recreate the database using the schema manager (see Section 7.3, “Run the Schema Manager”).

Chapter 8

Other Topics

This chapter contains an assortment of other topics worth knowing about when using *JPA Objects*.

8.1. Optimistic Locking

Optimistic locking is supported using the JPA `@Version` annotation, specifying the version property of an entity class that serves as its optimistic lock value. The version is used to ensure integrity when performing the merge operation and for optimistic concurrency control. That is, if the same domain object is materialized in two concurrent sessions and modified in both, then the first update wins and the second update fails. Internally, the JPA provider (Hibernate) throws an `OptimisticLockException` and marks the current transaction for rollback only. This exception is caught by the *JPA Objects*' object store implementation.

At least, theoretically. We may need further tests here...

Only a single `@Version` property or field should be used per class (though *JPA Objects* does not enforce this). Furthermore, the `@Version` property should be mapped to the primary table for the entity class (though *JPA Objects* does not enforce this either).

The following types are supported for version properties:

- int, Integer
- short, Short
- long, Long
- Timestamp

For example:

```
import javax.persistence.Version;
import org.nakedobjects.applib.annotation.Optional;
...
```

```
public class Customer {

    ...

    // {{ Property: JPA version
    private Long version;
    @Optional
    @Version
    public Long getVersion() {
        return version;
    }
    public void setVersion(final Long version) {
        this.version = version;
    }
    // }}
}
```

The property should also be annotated as `@Optional`, because otherwise Naked Objects will provide a default value when the object is first created. Hibernate, on the other hand, relies on `null` values in the version field as one of its strategies for distinguishing transient objects from persistent ones.

8.2. Lifecycle Listeners

Naked Objects provides a number of well-defined callbacks into the domain object lifecycle. These are implemented as facets that search for well-defined methods:

- `created()`

  (transient) domain object instantiated and initialized (dependencies injected, non-optional properties set to default values)

- `persisting()`, `persisted()`

  transient domain object about to be persisted / has been persisted

- `loading()`, `loaded()`

  persistent domain object about to be loaded / has been loaded

- `updating()`, `updated()`

  persistent domain object about to be updated / has been updated

- `removing()`, `removed()`

  persistent domain object about to be removed / has been removed (that is, deleted)

These callbacks are supported by all object store implementations. A typical example is to capture the ID for use within a domain object.

In addition, the JPA specification defines a number of annotations that also define callbacks into the persistence lifecycle. Because these are implemented by the JPA provider (Hibernate), they won't be portable across different object stores. However, they may well be of use in production code.

The callbacks provided are (methods annotated using):

- `@PostLoad`
- `@PrePersist`, `@PostPersist`

- `@PreUpdate`, `@PostUpdate`
- `@PreRemove`, `@PostRemove`

For more on these callbacks, see for example this article from Oracle's technical documentation.

Alternatively, the `@EntityListeners` annotation can be used to specify another class to be notified.

8.3. Persistence by Reachability (Cascading)

Persistence by reachability is the idea that if a transient object A references another transient object B, then persisting A will also persist B.

Both Naked Objects and also JPA support this concept. The Naked Objects' support is through the `PersistAlgorithm` interface. This allows object store implementations to instruct the persistor as to how to traverse the object graph. For example, the default algorithm (used by the in-memory object store) is implemented in `DefaultPersistAlgorithm`. This queues objects for persisting bottom-up, referred to (child objects) first and the parent objects. The `TopDownAlgorithm`, on the other hand, works the other way. The ability to plug in different algorithms provides flexibility for different persistence technologies.

JPA Objects uses (a trivial subclass of) the `DefaultPersistAlgorithm`. This is sufficient because the JPA provider (Hibernate) itself will persist objects in the correct order.

If required, you can also set the `CascadeType` using the cascade attribute of `@ManyToOne` and `@OneToOne` annotations. This can be thought of as a generalization of the persistence-by-reachability idea, cascading not only persists of transient objects, but also of updates and of removals (deletions).

`CascadeType` can take the following values:

- `PERSIST`

  Cascades calls to `EntityManager#persist()`, in other words, persistence-by-reachability for transient instances referenced through the specified association.

- MERGE

  Cascades calls to `EntityManager#merge()`. This is equivalent to cascading updates .

- REMOVE

  Cascades calls to `EntityManager#remove()`. This is equivalent to cascading deletions.

- REFRESH

  Cascades calls to `EntityManager#refresh()`. This is equivalent to cascading refreshes (reloading an instance after a database change, for example a trigger running).

- ALL

  Shortcut for `cascade={PERSIST, MERGE, REMOVE, REFRESH}`

In a Naked Objects application these methods are called from *JPA Objects*, not from your domain objects. specicailly, in the implementations of `PersistenceCommand` (for example `JpaCreateCommand`). Even so, there may be occasions when these annotations will influence the normal behaviour to accomplish a specific objective.

Note

Over time a set of patterns may emerge to help define suitable CascadeType settings for various domain model relationships. If so, this guide will be updated to provide further guidance.

8.4. Lazy Loading

Very few domain objects are standalone; most have properties and/or collections that reference other objects. This creates a problem for persistence technologies: if we load (or resolve) one object from the object store, how do we prevent having to load everything it refers to (and transitively, everything they refer to, and so on).

Lazy loading provides the solution to this, by returning proxies for the referenced objects. Only when these proxies are queried do they perform a further database query. In this way we "walk the graph" piecemeal.

Different JPA providers do this in different ways, and the proxy also varies depending on whether there is lazy loading of a property (referencing a single instance of some other object) or a collection (referencing many instances). *JPA Objects* uses Hibernate, which works as follows:

- for properties, Hibernate returns a proxy to the referenced object, implemented in either CgLib or Javassist. These bytecode libraries are able to create proxies even for (non-final) domain classes.
- for collections, Hibernate replaces the original collection implementation (eg `java.util.ArrayList`) with its own implementation (`eg org.hibernate.PersistentList`).. When this proxy collection is queried, it too queries the database. The instances returned can either be resolved objects, or could themselves be proxies.

The programmer can influence lazy loading by specifying the `fetch` attribute of `@javax.persistence.OneToOne` and `@javax.persistence.ManyToOne` annotations:

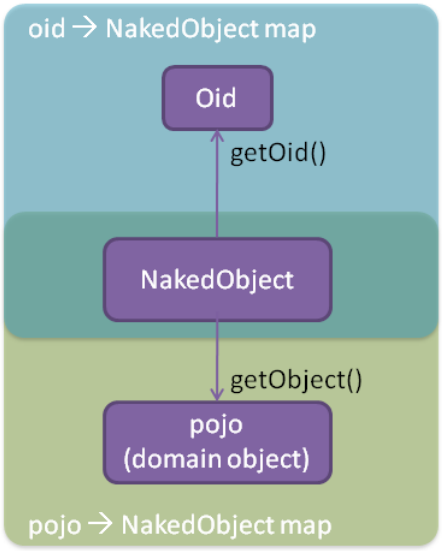
- The `EAGER` strategy is a requirement on the persistence provider runtime that data must be eagerly fetched.
- The `LAZY` strategy is a hint to the persistence provider runtime that data should be fetched lazily when it is first accessed.

The only other programming model restriction is that collections must provide a setter (even if only with `private` visibility) so that Hibernate can inject its own collection implementations.

Naked Objects itself also provides support for lazy loading, and is configured to support it out-of-the-box. This is needed in two main situations:

- for the client in client/server deployments; if a user clicks on a collection (in the DnD viewer) then Naked Objects will resolve the collection
- for either the server in client/server deployments or for webapp deployments where the object store implementation does not itself support lazy loading

Naked Objects implementation also uses either CgLib or Javassist, though the implementation differs from Hibernate. In Section 2.5, “Specify a discriminator” we discussed the idea of an `Oid` - a unique identifier for every domain object. However, this `Oid` doesn't just float around, it is actually associated with a `NakedObject`. This is a wrapper (or adapter) for the underlying domain object. The Naked Objects framework maintains mappings so anyone can be looked up from the others:



Naked Objects' lazy loading support is managed through these `NakedObject` adapters. Each adapter keeps track of the "resolve state" of its corresponding pojo. An unresolved object has not yet been loaded, so any request to view its contents forces a load (across the network, for a client, or from the objectstore, for a server). A CgLib proxy is used to trigger the request to resolve the domain object if required; ultimately this is marshalled through the `DomainObjectContainer#resolve()` method.

How does all this tie into *JPA Objects*, though? Well, because Hibernate is used under the covers , we actually switch off Naked Objects' own lazy loading for deployments running server-side or as a webapp. Details of how to do is shown in Section 7.1, “Configure Naked Objects”.

8.5. Common Properties (`@MappedSuperclass`)

JPA provides the `@MappedSuperclass` annotation as a way of factoring out common properties into a generic superclass to inherit from. For example, you might decide that every entity should have a `LastUpdatedBy` property. You might then defined an `AbstractPersistentObject` superclass, with all your entities inheriting from this. The `@MappedSuperclass` annotation indicates that this superclass if for implementation inheritance (rather than type inheritance), and therefore should not have any table defined for it. Instead, the tables for every inheriting entity should have their own columns (eg `last_updated_by`); the inherited properties are "pushed down" into the subclasses.

If necessary, inherited mapping information may be overridden in the subclasses by using the `@AttributeOverride` and `@AssociationOverride` annotations.

8.6. Retrieving Collections in Order

To specify the order of elements in a collection, use the `@OrderBy` annotation.

If the ordering element is not specified, the JPA specification says that the ordering by the primary key of the associated entity is assumed. Because *JPA Objects* requires the use of surrogate Ids (see Section 2.3, “Specify a primary key using `@Id` (not using `@IdClass` or `@EmbeddedId`)”), this basically means in insert order. For larger collections, if you want to see the most recently added objects at the top, then use `@OrderBy`.

8.7. Transient (non-persisted) Properties

To define a property as non-persistent, use the `@Transient` annotation.

Naked Objects supports the concept of derived properties, basically those that provide only a getter and no setter. In the Naked Objects viewers derived properties are in read-only (always disabled).

Because a transient property cannot be persisted, it also does not make sense for it to be set. For this reason the `@Transient` property also implies derived, and thus will be rendered as disabled.

8.8. Handling Exceptions

The JPA specification defines the following exception types:

- `EntityExistsException`

  Thrown by the persistence provider when `EntityManager#persist(Object)` is called and the entity already exists. The current transaction, if one is active, will be marked for rollback.

- `EntityNotFoundException`

  Thrown by the persistence provider when an entity reference obtained by `EntityManager#getReference(Class,Object)` is accessed but the entity does not exist. Also thrown when `EntityManager#refresh(Object)` is called and the object no longer exists in the database. The current transaction, if one is active, will be marked for rollback.

- `OptimisticLockException`

  Thrown by the persistence provider when an optimistic locking conflict occurs. This exception may be thrown as part of an API call, a flush or at commit time. The current transaction, if one is active, will be marked for rollback.

- `NonUniqueException`

  Thrown by the persistence provider when `Query#getSingleResult()` is executed on a query and there is more than one result from the query. This exception will not cause the current transaction, if one is active, to be marked for roll back.

- `NoResultException`

  Thrown by the persistence provider when `Query#getSingleResult()` is executed on a query and there is no result to return. This exception will not cause the current transaction, if one is active, to be marked for roll back.

In all cases these methods are called internally by *JPA Objects*, and will be trapped and handled as necessary, with exceptions converted into Object store exceptions to be handled by the Naked Objects framework itself.

TODO: further tests may be required to verify correct behaviour in all these cases.

Appendix A. Using the Maven Archetype

Typically we advise that you don't integrate your domain model through to a relational DBMS in the early exploration/prototyping stages of development, because it's then easier to refactor and rework the model as your understanding of the complexities of the domain deepens. During this time, just use the in-memory object store and use fixtures to populate the object store for demoing. (You might also want to check out the FitNesse integration provided by the Tested Objects sister project).

However, there will come a point (eg after a couple of iterations) when the domain model is stabilizing. At this point you'll want to start integrating your new/updated domain objects into an RDBMS. The *JPA Objects*' Maven archetype is designed to help you complete this integration work.

When run the archetype creates a parent module along with some child modules, the idea being that you incorporate these modules into your own parent module (by editing the `<modules>` element). However, because most of the complexity involved is in adding the JPA annotations, what the archetype also does is supply examples of annotated domain objects for the "claims" applications that ships with Naked Objects. What you might therefore want to do is to gain experience by running the Maven archetype against this claims application. When you've seen how it works, you can run it again for your own application.

The following sections provide a workthrough of how to run the Maven archetype and show how to get the claims application running with it.

A.1. Obtain the Claims Application

In this walk-through we want to see how the example domain objects and repositories for the "claims" application can be used. So, first we should load up this "claims" application.

If you haven't done this previously, go to Naked Objects' Sourceforge project and download the `nakedobjects-4.0.x-for-maven.zip` archive. Unzip this to a convenient location. The parent module `pom.xml` application is in `~/examples/claims`.

In Eclipse (with m2eclipse installed, see Section 1.2, “Prerequisites”), see the starobjects developers guide if required), switch to a new workspace, and then import the "claims" application using File > Import > General > Maven Projects. When done you should end up with 6 projects (claims, claims-commandline, claims-dom, claims-fixture, claims-service and claims-webapp).

Before continuing, check that you can run up the application using the DnD viewer or the HTML viewer, using the `.launch` scripts in claims-commandline project's `eclipse_launch` folder.

A.2. Set up a Relational Database

If you already have a database setup, know its JDBC URL and have a JDBC driver for it, then skip onto the next section, JPA Objects 1.0 Users Guide.

Otherwise though we need to install a database server. There are a number of very good open source relational database around; the one I've chosen to use here is PostgreSQL. It's cross platform, easy to setup, has Java JDBC drivers, and comes with a nice GUI administration tool so that we can see what's going on.

So, go to the PostgreSQL website and download the pre0buit binary install (I selected PostgreSQL 8.3.7) for your operating system. Do the install, remembering the password for the *postgres* superuser.

We could leave it at that, but it's bad practice to use superuser accounts and built-in databases. So, let's also create our own login and database:

- on the top-level PostgreSQL server node, right click and then select New Object > New Login Role.
- in the resultant dialog, enter a role name of "claims" and a password of "claims" also. Hit OK.
- next, right click on the Databases node (under the server node), and select New Database
- in the resultant diag, enter the name "claims_db", and specify the owner as "claims"

Double check that everything is setup correctly by logging out and reconnecting as the new "claims" login.

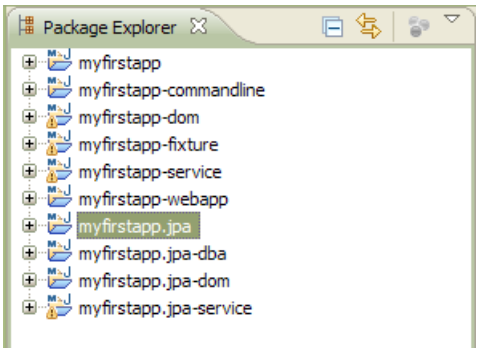
We're now ready to run the Maven archetype.

A.3. Run the JPA Archetype

We can run the archetype either using a wizard within Eclipse, or outside Eclipse and then import (similar to the steps in Section A.1, “Obtain the Claims Application”). To use the wizard:

- use File>New>Maven Project to bring up the wizard dialog
- skip the first page of the wizard; on the second page, select the catalog "Nexus Indexer" and then select the latest version of the jpaobjects-integration archetype
- on the final page, enter:
  - `group Id` of org.nakedobjects.examples
  - `artifact Id` of claims-jpa
  - `package` of org.nakedobjects.examples.claims-jpa
  - the additional `appRootArtifactId` variable to claims (that is, the `artifact Id` of the example application)
- Hit Finish

Maven will now generate four new projects in your workspace, as shown below:



Running through these in detail:

- the claims-jpa-dom project holds a copy of the original classes in claims-dom, updated to include the JPA annotations. We'll be copying these source files over the original versions in just a minute
- the claims-jpa-service project provides implementation of the two repositories. We'll use this project as a complete replacement for the original claims-service.
- The claims-jpa-dba has no code in it but instead has Eclipse launch configuration to allow us (or our DBA) to create the database schema and to seed the database with our fixtures
- The claims-jpa project is just a parent for the above three projects.

Before we go any further, this would be a good time to check database connectivity.

Now, use Run > Run Configuration... to bring up the list of configurations, and navigate to the `dba - Ping - PostgreSQL` launch configuration (from the claims-jpa-dba project). Inspect the command line arguments. If you are using the PostgreSQL database (as set up in Section A.2, “Set up a Relational Database”) then the arguments should be correct, but if using your own database then you'll need to adjust them.

When ready, choose Run to run the configuration. In the Console view an "OK" message should appear. If you don't, then you'll need to investigate further (`Ping` is a very simple class so the problem will almost certainly be in your setup somewhere).

Now let's update the existing projects to reference the newly generated projects.

A.4. Update the Existing Project's Classpath

Updates are required to several of the existing `pom.xml` files.

Parent Module

In the parent (claims) project's pom.xml, locate the `<modules>` element and update to include the claims-jpa project (which in turn includes the other new projects):

```
<modules>
    <module>dom</module>
    <module>fixture</module>
    <module>service</module>
    <module>commandline</module>
    <module>webapp</module>
    <module>../claims-jpa</module>
</modules>
```

Next, move the salient configuration from the claims-jpa container project's `pom.xml` to the parent claims project's `pom.xml`:

- under the `<properties>` section there are elements defining versions for JPA Objects, PostgreSQL, jTDS and SL4J. Cut-n-paste all of these to the corresponding `<properties>` section in the parent `pom.xml`;
- similarly, cut-n-paste the entire contents under the `<dependencyManagement>` element from claims-jpa's `pom.xml` into corresponding section in claims' `pom.xml`.

Fixture Module

Now, in the claims-fixture project's `pom.xml`, update its `<dependencies>` section to add in a new `<dependency>` to claims-jpa-service (you can leave in the reference to the original claims-service or remove it, it doesn't matter):

```
<dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>claims-jpa-service</artifactId>
</dependency>
```

DOM Module

Finally, copy the `<dependency>` section from claims-jpa-dom `pom.xml` (referencing the JPA Objects' own applib) to the corresponding section in claims-dom:

```
<dependency>
    <groupId>org.starobjects.jpa</groupId>
    <artifactId>jpa-applib</artifactId>
</dependency>
```

JDBC Drivers

If you set up the PostgreSQL database (as described in Section A.2, “Set up a Relational Database”) then you should be done. If you are using your own database then you will also need to add `<dependency>` entries to your JDBC driver in the Maven `pom.xml` (search out the existing PostgreSQL `<dependency>`s and add your own alongside).

A.5. Update the Configuration

The main changes we need to make are for Naked Objects' configuration (as discussed in Section 7.1, “Configure Naked Objects”) and for Hibernate itself (as discussed in Section 7.2, “Configure Hibernate”). Follow the steps described there, configuring (a) the persistor key, (b) the reflector key, (c) disable the bytecode providers and (d) registering the JPA repository implementations. For the last of these, the service implementations to specify should be:

```
nakedobjects.services=jpa.service.employee.EmployeeRepositoryJpa,\
                      jpa.service.claim.ClaimRepositoryJpa
```

For Hibernate, make sure that the JDBC driver, username, password and URL are correct, as well as the dialect. If using the PostgreSQL database (as described earlier in Section A.2, “Set up a Relational Database”) then the settings to use are:

```
<property name="connection.driver_class">org.postgresql.Driver</property>
<property name="connection.username">claims</property>
<property name="connection.password">claims</property>
<property name="connection.url">jdbc:postgresql://localhost/claims_db</property>
<property name="dialect">org.hibernate.dialect.PostgreSQLDialect</property>
```

A.6. Update the Domain Classes

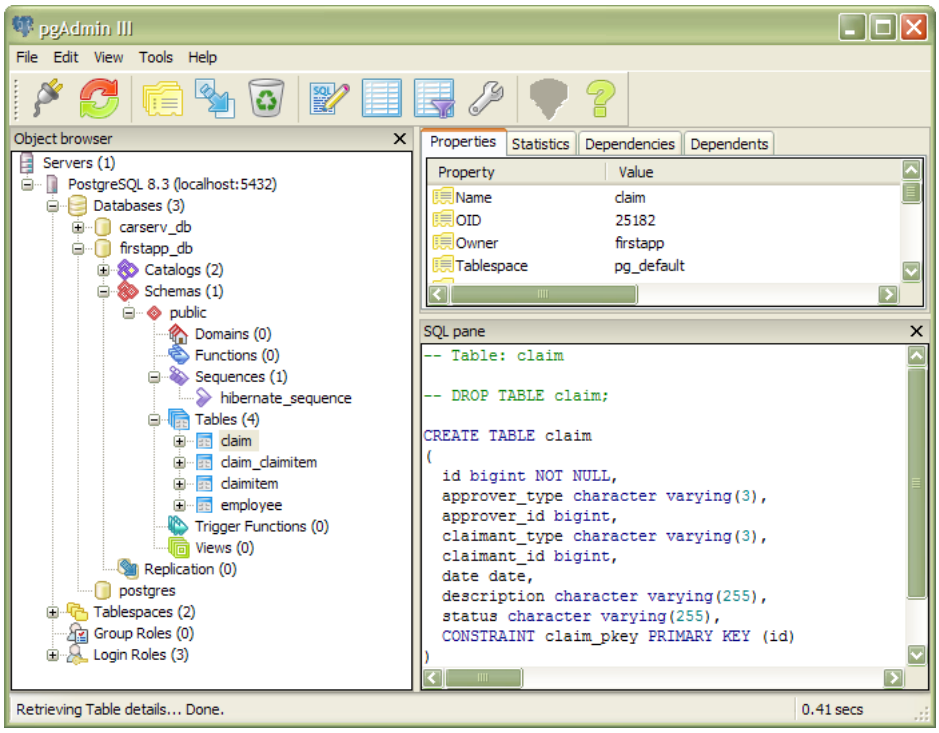
The main work involved with integrating JPA Objects is in annotating the domain classes, as described in detail in Chapter 4, *Annotating Domain Objects*. For us, though, we can use the pre-annotated versions of the claims domain classes. These reside in the claims-jpa-dom project, in the `jpasrc` directory.

Therefore, copy over the files from this directory into the claim project's `src/main/java` directory. If you want to see the types of changes that are required then use a file comparison utility (such as WinMerge, Meld or FileMerge) to copy over the files.

A.7. Build the Schema

To create the database schema (using `create table` SQL commands) we can use the `SchemaManager` tool that comes with *JPA Objects*. This is really just a wrapper around Hibernate's equivalent **hbm2ddl** tool.

The steps for doing this is described in Section 7.3, “Run the Schema Manager”, but the Maven archetype goes a little further by setting up some launch configurations for you. Use Run > Run Configurations... and locate the `dba - SchemaManager - create only` launch configuration. Then Run. Inspect your database using the *PgAdmin III* GUI admin tool (or equivalent if using your own database); you should see a number of tables created, as shown below:



For convenience, there are other launch configurations for drop, to recreate or to preview. The last of these, preview, will not hit the database, instead it will just print out the SQL.

A.8. Install the Fixtures

Having built the database we need to populate the database. Unlike the in-memory object store, *JPA Objects* never loads fixtures at run time. Instead, we use the `FixtureManager`.

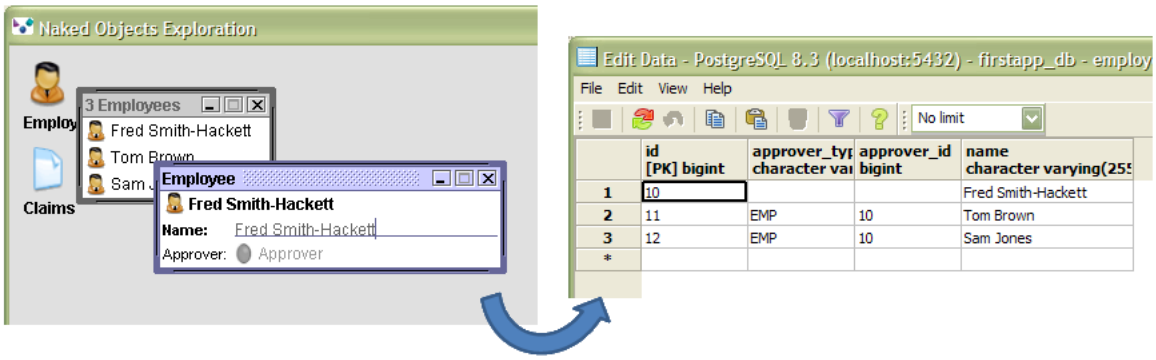
The steps for doing this is described in Section 7.4, “Run the Fixture Manager”, but the Maven archetype goes a little further by setting up some launch configurations for you. Use Run > Run Configurations... and locate the `dba - Fixture Manager` launch configuration; then Run. Rows representing the domain objects should be inserted into the database.

Note that all the `FixtureManager` does is to boot up Naked Objects with a special flag set so that *JPA Objects does* load the fixtures. The fixtures loaded will be those defined in nakedobjects.properties. If you want to load up a different set (for example, just the reference data) then provide a fixture set just for the data to be loaded.

A.9. Run the Application

Finally, we're ready to run the application. We can run it in the usual way using either the DnD viewer or the HTML viewer. There's no need to specify *JPA Objects*' persistor in the command line flags because we've added it already in `nakedobjects.properties` file.

You should be able to make changes and then use the *PgAdmin III* GUI tool (or equivalent) to see the changes being persisted:



Appendix B. Annotations Reference

This appendix describes the annotations that are identified by Naked Objects and whose semantics are captured in the Naked Objects metamodel. The presence of some of these are reflected in the Naked Objects viewers, others are required by JPA Objects itself.

Note that these are not the only annotations that can be applied; in general most JPA/Hibernate annotations will work. However, there are a number of annotations that are not supported, namely those that define composite primary keys. See Section 2.3, “Specify a primary key using `@Id` (not using `@IdClass` or `@EmbeddedId`)” for further discussion.

B.1. javax.persistence

`@javax.persistence.Basic`

Applies only to properties. Used by JPA to describe a simple value property.

For Naked Objects, it determines:

- whether the property is mandatory or not, using `@Basic(optional=...)`
- the fetch type; `@Basic(fetch=...)`. This isn't used by Naked Objects viewer but is stored in the metamodel.

`@javax.persistence.Column`

Applies only to properties. Used by JPA to specify details about the column in the database table.

For Naked Objects, it determines:

- whether the property is mandatory or not, using `@Column(nullable=...)`
- for a string property, its maximum length, using `@Column(length=...)`

`@javax.persistence.DiscriminatorValue`

Applies only to classes. Used by JPA to distinguish subclass entities (see the section called “`@javax.persistence.Entity`”) within inheritance hierarchies.

Whereas in "regular" JPA it is only necessary to annotate entities with inheritance, JPA Objects makes this a mandatory requirement for all entities that are not embeddable (see the section called “`@javax.persistence.Embeddable`”). This is for two reasons:

- first it helps enforce standards in the use of meta-def strings for polymorphic relationships (see the section called “`@org.hibernate.annotations.AnyMetaDef`”).
- second, it is used for the string form of the object identifier (see Section 2.3, “Specify a primary key using `@Id` (not using `@IdClass` or `@EmbeddedId`)”).

@javax.persistence.Embeddable

Applies to classes. Used by JPA to indicate that a class is not standalone, in other words is embeddable within another entity (see the section called "@javax.persistence.Entity") by way of an property annotated with @Embedded (see Section 4.7, "Embedded Objects").

In domain-driven design terms an embeddable object is an aggregated object, and this semantic is captured in the Naked Objects metamodel. In addition, JPA Objects ensures that every domain class is annotated as either an @Entity or @Embeddable.

@javax.persistence.Entity

Applies to classes, indicating that the class represents a persistent entity.

For Naked Objects, @javax.persistence.Entity is captured in the metamodel and is used for validation; every domain class must be annotated as either an @Entity or @Embeddable (see the section called "@javax.persistence.Embeddable"). See Chapter 2, *JPA Objects / Naked Objects Restrictions* for more details. It isn't otherwise used.

See also the Hibernate specific annotation, the section called "@org.hibernate.annotations.Entity".

@javax.persistence.FetchType

Applies to properties and collections. Used by JPA/Hibernate, to specify lazy loading characteristics.

Not used by Naked Objects, but captured in the metamodel (primarily as a side-effect of capturing semantics by other annotations that do provide other relevant info, eg @org.hibernate.annotations.CollectionOfElements, the section called "@org.hibernate.annotations.CollectionOfElements").

@javax.persistence.Id

Applies only to properties. Used by JPA to identify the property to use as the primary key of an entity (see the section called "@javax.persistence.Entity").

In JPA every entity must be identified, and while this is typically done using @Id, it is also possible to use composite keys. However, JPA Objects does *not* support composite keys; every entity must be annotated with @Id.

The reasoning is the same as that for making discriminator values (see the section called "@javax.persistence.DiscriminatorValue") mandatory:

- first, Hibernate requires that any classes engaged in polymorphic relationships must use an Id; and such relationships are common in Naked Objects applications;
- second, the Id is used for the string form of the object identifier (see Section 2.3, "Specify a primary key using @Id (not using @IdClass or @EmbeddedId)").

For Naked Objects, the @Id annotation implies that the property is both mandatory and disabled.

@javax.persistence.JoinColumn

Applies only to properties. Used by JPA to specify a mapped column for joining an entity association, in other words the name of the foreign key table for a `@ManyToOne` (the section called "`@javax.persistence.ManyToOne`") or a `@OneToOne` (the section called "`@javax.persistence.OneToOne`") association.

For Naked Objects, determines:

- whether the property is optional, using `@JoinColumn(nullable=...)`

@javax.persistence.ManyToOne

Applies only to properties. Used by JPA to represent a many-to-one (child/parent) relationship, indicating the property on a "child" entity that is a reference to the "parent" entity.

For Naked Objects, determines:

- whether the property is optional, using `@ManyToOne(optional=...)`
- fetch type, from `@ManyToOne(fetch=...)`; not used by Naked Objects but captured in the meta-model

See also the section called "`@javax.persistence.OneToOne`".

@javax.persistence.NamedQueries and @javax.persistence.NamedQuery

Both of these annotations apply only to classes, and in particular to entities (see the section called "`@javax.persistence.Entity`"). They define a named query/set of named queries) which return the annotated entity, for use in repository implementations. See Chapter 5, *Implementing Repositories* for further details.

@javax.persistence.OneToOne

Applies only to properties. Used by JPA to represent a one-to-one relationship, indicating the property on a "referencing" entity to a "referenced" entity.

For Naked Objects, determines:

- whether the property is optional, using `@OneToOne(optional=...)`
- fetch type, from `@OneToOne(fetch=...)`; not used by Naked Objects but captured in the meta-model

See also the section called "`@javax.persistence.ManyToOne`".

@javax.persistence.Transient

Applies only to properties. Used by JPA to indicate that the property is not persistent.

For Naked Objects, determines:

- that the property is derived (cf a property with only a getter and no setter)

@javax.persistence.Version

Applies only to properties. Used by JPA to specify the propert that serves as its optimistic lock value.

For Naked Objects, the `@Version` annotation implies that the property is both mandatory and disabled.

For further discussion on optimistic locking, see Section 8.1, “Optimistic Locking”.

B.2. org.hibernate.annotations

@org.hibernate.annotations.AnyMetaDef

Applies onto to properties. Used by Hibernate to captures the semantics for an "any" (polymorphic) relationship. See for more details on using this annotation.

The above notwithstanding, the `@AnyMetaDef` annotation is captured in the Naked Objects metamodel, but not otherwise used by Naked Objects.

@org.hibernate.annotations.CollectionOfElements

Applies only to collections. Used by JPA/Hibernate to allow collections of elements to be persisted in a separate table.

For Naked Objects, it determines:

- the class of collection (cf `@org.nakedobjects.applib.TypeOf` annotation)
- the fetch type (see the section called “`@javax.persistence.FetchType`”).

The class of the collection is potentially usable by Naked Objects viewers (eg to return a table of objects instead of a simple list of their titles).

@org.hibernate.annotations.Entity

Applies to classes. The Hibernate annotation captures some additional semantics not in the JPA spec.

For Naked Objects, the `@org.hibernate.annotations.Entity` annotation determines:

- immutability; `@org.hibernate.annotations.Entity(mutable=false)` implies the class is immutable (cf `@org.nakedobjects.applib.annotation.Immutable` annotation)

See also the JPA specific annotation, the section called “`@javax.persistence.Entity`”.

@org.hibernate.annotations.Immutable

Applies to classes and to properties. Used by JPA/Hibernate "to make some minor performance optimizations" (it probably does not bother to wrap in proxies).

More significantly, for Naked Objects it determines:

- on a class, that it is immutable (cf `@org.nakedobjects.applib.annotation.Immutable` annotation)

- on a property, that it is disabled (cf `@org.nakedobjects.applib.annotation.Disabled` annotation)