

Evaluating Options for Persisting Java Objects

Hibernate, DB4O, and Caché Database with Jalapeño

by Richard Conway



We live in a relational world – which is too bad since we develop with objects. Since most non-trivial applications require information to be persisted and retrieved in what is generically called a database, we need to find efficient methods for persisting our objects and retrieving them. Historically, this has been done with relational databases and lots of code that flattens the objects and maps them to the relational tables. This can be done in Java or with object-relational mapping tools like Hibernate.

While most books and articles about object-oriented software development discuss the benefits of using objects to describe the problem space, currently most development is done in a more convoluted manner, working from both the Java and database ends and using various technologies to bridge the gap between them. Hibernate has emerged as one of the most popular ways to address this development challenge.

Instead of starting with a database schema and building objects from the tables and data therein, this article will focus on starting with the Java objects and evaluate a few of the many options for persisting them. We will compare and contrast the methods, as well as discuss challenges and problems specific to object persistence management (see Table 1).

Options for Persisting Java Objects

There are many options for persisting your Java objects. Some are best used when your persistence needs are simple (such as saving program state between sessions) and some are better when your needs are complex (such as saving lots of data over long periods of time). Some applications simply need to load data for use in configuring the application, while others require sophisticated tools for searching and filtering object sets. The latter is typical for applications that use a database for persistence and will be the primary focus here. For each project you work on, you should evaluate which persistence strategy best fits your needs based on the project requirements. The following persistence mechanisms will be discussed in this article:

- **Hibernate**
 - The most commonly used object-relational mapping tool
- **DB4Objects DB4O**
 - A small simple embeddable object database
- **InterSystems' Caché Database with Jalapeño**
 - An enterprise database that lets you store POJOs via its Jalapeño technology – but also provides a JDBC/SQL interface to the objects stored in the database

Please note that there are many other object-relational mapping

solutions and object databases available, the purpose of this article is simply to provide some insight as to the benefits of using these technologies versus a JDBC/DAO implementation, how they compare and how you can use them for your projects.

Definition: DataStore – A system for storing and retrieving data. Can be relational or object-oriented.

Object Persistence Mechanism Considerations

For each of the persistence strategies outlined above, I will review the following:

- **Ease of Implementation**
 - How much preparation and/or configuration is required to persist your objects?
- **Ease of Persisting Objects**
 - How do you persist an object or objects?
 - Is it straightforward and intuitive?
 - Is it better/simpler/faster than using SQL and writing DAOs?
- **Ease of Retrieving Objects**
 - What mechanisms are available for finding and retrieving the objects in the datastore?
- **Control over Object Depth**
 - How many objects do you want to save or retrieve at the same time?
- **Control over Object Property Breadth**
 - How many properties do you need access to? Can you only return those properties?

	Typical/Object Relational	Desired/Object Oriented
1	Define the problem space using Nouns and Verbs to identify Objects and Methods	Define the problem space using Nouns and Verbs to identify Objects and Methods
2	Identify all the data that must be persisted as Objects and/or Tables, and document the relationships between the OBJECTS	Identify all the data that must be persisted AS OBJECTS and document the relationships between the OBJECTS
3	Develop a relational database schema to contain the data, representing the Objects as one or more Tables in a database. Work towards 3rd Normal Form	Create the database based on the Object Schema. Ideally, you have a one-to-one mapping between your business objects and your database object store
4	Write SQL Code, create Data Access Objects, or use JDO to persist and load data between the POJOs and the database	Avoid writing Data Access Objects or the equivalent. Deal with objects exclusively

Table 1 The software development process



Richard Conway is a software developer and technology consultant with more than 15 years of technology, project management, and information services experience. He has extensive experience developing Java/Struts-based web applications. He started focusing more on Swing based developments at the beginning of 2005 and has just finished a Swing-based client/server asset management project. He lives in Miami with his wife Patricia, is currently working on an EMR application, and plays sand volleyball in his spare time.
reconway@egrok.com

- Object Tree Traversal
 - Can you access all related objects and their properties simply by traversing the object tree?
- Enforcing Referential Integrity
 - How do you ensure you don't delete an object that other objects depend on?
 - * For example, can you delete a department if employees still exist?
 - Does it support cascade deletes/updates?
- Enforcing Uniqueness
 - How do you ensure that specific properties are unique in the database, such as Social Security numbers?
- Support for Indices
 - Can you define indices that will enhance the performance of your queries?
- Property Constraints
 - Can you control/limit the values that will be entered into the datastore?
- Security and Access Control
 - How do you control who has access to the data and what they can do with it?

Ease of Implementation

One of the things we're trying to get away from is the effort to write and maintain DAOs. So let's look at what's involved in setting up your datastore and prepping your POJOs to be persisted for each of the persistence mechanisms identified.

Hibernate

Hibernate has the most complex setup of the solutions discussed here, but it's not that bad and Hibernate does provide a lot of tools to make your life easier. There are multiple ways to implement Hibernate, but since we are starting the POJOs, we'll only consider the case of adding annotations to the Java class to support Hibernate persistence. If you're using Hibernate 3.2 with Java 1.5 or above, you can use annotations to map your POJO properties to your Table columns. Using annotations is much less verbose than defining your mappings in XML files and has the additional benefit of reducing the number of files you must keep track of. In addition to annotating your POJOs you need to provide the fully qualified name of the annotated class as a <mapping> element in the hibernate.cfg.xml file.

The minimal steps for preparing to persist your objects with Hibernate are:

- 1) Set up a database and create a username and password for Hibernate to access it with
- 2) Annotate your Java Classes
- 3) Add your class mappings, database, and user/login information to the hibernate.cfg.xml file
- 4) Run the Hibernate hbm2ddl to create the schema in the database

An example of mapping in the hibernate.cfg.xml file:

```
<mapping class="com.egrok.hibernate.Person"/>
```

Since with Hibernate, you're typically mapping to a relational database, complex objects may need to be persisted to multiple tables, adding some complexity to the implementation.

To save you the effort of annotating everything, Hibernate provides many default behaviors. For example, Hibernate by default will

map your class to a Table of the same name so you don't have to use the @Table annotation unless the class name and table name differ. The same goes for class properties. They will be mapped to columns of the same name in the Table. If you want to override this behavior, use the @Basic annotation. This type of intelligent behavior minimizes the work you have to do to persist a class. For our Person class example, this means that it will be persisted to the database Table named "Person," which will have three columns (id, firstName, and lastName) and the id column will be the primary key. Hibernate provides excellent control over how the primary key is assigned, but that's beyond the scope of this article.

An example of a minimally annotated POJO:

```
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class Person{
    @Id
    public Integer id;
    public String firstName;
    public String lastName;
}
```

An example of how a one-to-many relationship is defined:

```
// In the Department class -
@OneToMany(cascade=ALL,mappedBy = "department_id")// Requires the
foreign key "department_id" in the Person class
public ArrayList<Person> getEmployees(){// Method to return all
employees
return employees;
}

// In the Person class
@ManyToOne
@JoinColumn(name = "department_id")
public Department getDepartment(){ // Method to return the department
return department;
}
```

As a final time and labor saver, you can now use Hibernate's hbm2ddl tool to generate the database schema for you. While you may be splitting objects to save them in multiple tables, which is not required for the object databases, you do get very good control over the mapping process. And as we'll see later, there are also circumstances where the relational approach has advantages over a pure object approach.

DB4O

Implementing persistence with DB4O is dead easy. You don't need to create a database ahead of time or prep your Java Classes at all. Simply call Db4o.openFile() and provide the path to your database file as the parameter. If the database doesn't exist, it will be created for you. Then instantiate an Object and call db.Save(object) to persist it. It doesn't get any simpler.

```
ObjectContainer db=Db4o.openFile("C:\db4o\test.yapp");
try {
    Person person = new Person("James", "Hogan");
    db.set(person); // It's now saved!
```

```

}
finally {
db.close();
}

```

Since you're storing actual Java objects, there's no mapping required. There's also no need to annotate relationships, but you must provide properties in the objects on both sides of a relationship if you want to be able to traverse the object tree bidirectionally.

```

// In the Department class
public ArrayList<Person> employees;

// In the Person class
public Department department;

```

Caché

Setting up persistence with Caché and Jalapeño falls between Hibernate and DB4O in complexity. No mapping is required, however, as InterSystems correctly points out in their Jalapeño documentation: "Databases define concepts such as constraints, relationships (with referential integrity), and indices, which have no equivalent within a Java class definition." To address and support these concepts, InterSystems' Jalapeño provides database operation-specific annotations for use in your Java Classes, much as Hibernate does. However, they are only used by the Jalapeño SchemaBuilder to create the Object Storage in the Caché database, and aren't required (other than for documentation purposes) after the Object Storage has been created.

The minimal steps for preparing to persist your objects with Jalapeño/Caché are as follows:

- 1) Create a Namespace and Empty Database using the Caché System Management Portal
- 2) Run the Jalapeño SchemaBuilder to create the Object Storage in the database

An example of a minimally annotated POJO that can be persisted using Jalapeño/Caché:

```

public class Person{
    public String firstName;
    public String lastName;
}

```

Like DB4O, since Caché stores objects, there's a one-to-one relationship between your POJOs and the Caché Object Classes defined in the database – so no mapping is required.

Note: For any non-trivial object schema you'll have to add annotations to support database specific functionality – such as relationships and indices. Annotating your Java Classes for Jalapeño works like the way it works for Hibernate. InterSystems is releasing NetBeans, Eclipse, and IntelliJ plug-ins for Caché 2007 that make the process of adding annotations a point-and-click operation. Thus you can quickly define or modify the Object Storage in the Caché database.

An example of how a relationship is defined:

```

// In the Department class
@Relationship(type=RelationshipType.ONE_TO_MANY,inverseClass="Person")

```

```

public ArrayList<Person> getEmployees(); // Method to return all employees

```

```

// In the Person class
@Relationship(type=RelationshipType.MANY_TO_ONE,inverseClass="Department")

```

```

public Department getDepartment(); // Method to return the department

```

Ease of Persisting Objects

Once you have your databases created and configured, persisting your objects with any of these methods is very simple. Any of these is a distinct improvement over using JDBC/SQL and DAOs.

Hibernate

Once your Hibernate mappings are configured, persisting an object is very straightforward:

```

try{
SessionFactory factory = new Configuration().configure().buildSessionFactory();
Session session = factory.openSession();
Transaction tx = session.beginTransaction(); //optional
Person person = new Person("James","Hogan");
Long personID = (Long) session.save(person);
tx.commit();
session.close(); //optional
}catch(Exception e) {

}

```

DB4O

Once again, DB4O makes it extremely easy to persist your objects. Simply instantiate an object and then call db.set() on it.

```

ObjectContainer db=Db4o.openFile("C:\db4o\test.yapp");
try {
    Person person = new Person("James","Hogan");
    db.set(person); // It's now saved!
    // commit is called implicitly when you close the container,
    // so you don't really need to call it here.
    db.commit(); //optional
}finally {
    db.close();
}

```

Caché

Caché also makes it extremely simple to save your objects.

```

try {
/* Connect to this machine, in the SAMPLES namespace */
ObjectManager objectManager = connect (connectiontype, host, username,
password);
    Person person = new Person("James","Hogan");
        Object id = objectManager.save (person, false);
        objectManager.close ();
} catch (Exception ex) {

}

```

Ease of Retrieving Objects

Storing data is only half the equation. It must be easy to access your objects as well. Once again all three solutions simplify the process of getting an object.

Hibernate

You can use HQL or SQL to retrieve objects using Hibernate. For example:

```
package hello;
import java.util.*;
import org.hibernate.*;
import persistence.*;

public class HibernateExample {
    public static void main(String[] args) {
        Session session = HibernateUtil.getSessionFactory().openSession();
        Transaction newTransaction = session.beginTransaction();
        List people = newSession.createQuery("from Person m order by m.name asc").list();
        System.out.println( people.size() + " people found:" );

        for ( Iterator iter = people.iterator();
            iter.hasNext(); ) {
            Person person = (Person) iter.next();
            System.out.println( person.getName() );
        }
        newTransaction.commit();
        newSession.close();
        // Shutting down the application
        HibernateUtil.shutdown();
    }
}
```

DB4O

DB4O provides three different ways to retrieve your objects: Query By Example, Native Queries, and SODA Queries. Each has its pros and cons.

The simplest and most limited is QBE. With this method, you create a prototype of the object you're looking for using one of the object's constructors, and DB4O returns the matching records.

```
// QBE Example - Retrieve Person By Name
Person proto=new Person("Ben","Franklin",0);
ObjectSet result=db.get(proto);
listResult(result);
```

Native Queries are the preferred query method and are type-safe, compile-time checked, and refactorable.

Caché

Caché provides an object oriented query mechanism that uses SQL for selection and which returns an iterator you can use to traverse the returned objects. You also have the option of using JDBC and SQL to perform complex queries over multiple related objects (SQL JOINS) or VIEWS. The results of these complex queries can also be accessed as objects, as long as the Object ID is returned as part of the query.

```
import com.intersys.pojo.ApplicationContext;
import com.intersys.pojo.ObjectManager;
```

```
try {
    ObjectManager objectManager;
    String url="jdbc:Caché ://localhost:1972/" + namespace;
    objectManager = ApplicationContext.createObjectManager (url,
        username, password);

    String sql = "Name %startsWith ?"; // Search for people by
    name
    if ("null".equalsIgnoreCase (query)){
        query = null;
    }
    String[] qargs = {query};
    Iterator people = objectManager.openByQuery (Person.class,
    sql, qargs);
    while (people.hasNext()){
        IPerson person = (Person) people.next();
        System.out.print ("Name: " + person.getName() );
    }
    objectManager.close ();
} catch (Exception ex) {
    System.out.println( "Caught exception: " + ex.getClass().get-
    Name() + ": " + ex.getMessage() );
    ex.printStackTrace();
}
```

Controlling Object Depth

One of the challenges of storing objects is to control how many objects are stored and retrieved at one time. For example, consider a company that contains 100 departments and each department contains 25 to 50 employees. Using Java Serialization, serializing the Company object would also persist all the Department and Employee objects – and instantiating the Company would also instantiate the related Department and Employee objects. If all you need to persist is the Company Object, you've done a lot of unnecessary work! There's considerable impact on performance and memory requirements for every object instantiated. Furthermore, if you send this object over the network, you want to be as efficient as possible – and therefore only send the Company Object.

What's needed is a mechanism for controlling how deeply you traverse the object tree when instantiating objects.

Hibernate

Hibernate handles this well because it's based on a relational database structure. It is easy to control the depth of the data (or the depth of the objects) returned by specifying LAZY or EAGER loading and if you switch to JDBC, you have total control using SQL JOIN statements.

When you specify LAZY loading, the POJO is actually replaced with a proxy object that will load properties and collections via the Hibernate session as needed.

DB4O

DB4O provides excellent control over object depth. You can easily specify exactly how many levels you want to retrieve or update. You can also turn on cascading updates/deletion – which traverses the entire object graph:

```
Db4o.configure().objectClass(Car.class).cascadeOnUpdate(true);
```

	Serialization	Hibernate	DB4 Objects	Caché/Jalapeño
Pros	<ul style="list-style-type: none"> • Applies to any object • Simple to implement 	<ul style="list-style-type: none"> • Widely used and thus well documented and supported • Easy for developers experienced with relational databases to grasp • Since this solution typically uses a relational database for storage, it provides good control over the breadth and depth of the data persisted and retrieved. 	<ul style="list-style-type: none"> • Extremely simple to implement • Good option for embedded projects • Good option when developers are not familiar with relational database technology • Since this solution typically uses a relational database for storage, it provides good control over the breadth and depth of the data persisted and retrieved. 	<ul style="list-style-type: none"> • Easy to implement • Scales from embedded to clustered applications • Provides both Object and Relational/SQL access to data • Support available 24x7x365
Cons	<ul style="list-style-type: none"> • No control over object depth or breadth persisted • No control over uniqueness • Not very scalable/performant 	<ul style="list-style-type: none"> • Requires more effort than the other solutions to map your objects to relational tables (the Object-Relational Impedance Mismatch). • Adds some processing overhead • Can be expensive if based on a commercial database. 	<ul style="list-style-type: none"> • No enforcement of referential integrity • No enforcement over uniqueness • No user access controls. If you have access to the database - you have access to the entire database. • More expensive than Hibernate solutions backed by open source/free relational databases. 	<ul style="list-style-type: none"> • More expensive than DB4O or Hibernate solutions backed by open source/free relational databases. • Currently cannot store Enums, although that should be part of the next release.
Ease of Implementation	Just make Class Serializable and write the associate method.	Rating = C <ul style="list-style-type: none"> • Set up database • Set up hibernate.cfg.xml • Annotate POJOs to map the properties to database table columns • Run the Hibernate hbm2ddl tool to generate the database schema 	Rating = A <ul style="list-style-type: none"> • Extremely Simple - No Mapping Required! • Call Db4o.openFile() to open or create the database. 	Rating = B+ <ul style="list-style-type: none"> • No Mapping Required! • Set up database • Annotate POJOs to support database specific features (indexes, relationships, uniqueness, referential integrity) • Run the Jalapeño SchemaBuilder to create the associated Object Storage in the database. • <i>Note:</i> IDE plugins simplify annotation of POJOs • <i>Note:</i> You can also define your objects directly in Cache and then project them as Java objects.
Ease of Persisting An Object	Rating = A	Rating = B	Rating = A	Rating = B
Ease of Retrieving An Object	Rating = F No integral for search. Need to traverse the object tree and perform your own comparisons.	Rating = A You can use either SQL or HSQL to find and retrieve objects.	Rating = B You can use any of the following methods to find and retrieve objects: <ul style="list-style-type: none"> • Query By Example • Native • SODA 	Rating = A You can use SQL to find and retrieve objects.
Control Over Object Depth	None built in	Rating = B	Rating = A	Rating = B
Control Over Property Breadth	None built in	Rating = A You can access as many or as few of an object's properties as you wish	Rating = F No control here - you can only retrieve the full object	Rating = B Can't control this directly through Jalapeno, but you can switch to JDBC or even use Hibernate to provide this functionality.
Object Tree Traversal	Full/Integral	Full/Integral	Full/Integral	Full/Integral
Referential Integrity	None built in	Rating = A Excellent, but depends on the database.	Rating = F None built in.	Rating = A Excellent. Full support for maintaining referential integrity, as well as for cascade updates and deletes.
Uniqueness	None built in	Rating = A or B Depends on the underlying database	Rating = F <ul style="list-style-type: none"> • Currently there is no support for enforcing uniqueness, other than through your application. • <i>Note:</i> DB4O is working on this feature and currently expect it to be released in mid-summer 2007. 	Rating = A Any number of properties can be marked as unique.
Indexing	None built in	Rating = A/B Depends on the underlying database	Rating = B <ul style="list-style-type: none"> • Limited indexing options • Indexing has performance considerations 	Rating = A <ul style="list-style-type: none"> • Many indexing options available • BitMap/BitSlice indexing are extremely performant • Adding indices makes little or no impact on insert/update performance.
Property Constraints	None built in	Depends on database.	None	Full/Integral You can constrain a property to fall within a specific range of values, exist within a predefined list, not be null, etc...
Access Control	None built in	Excellent to Non-Existent: Depends on the database.	Can encrypt and password protect the database, but there are no mechanisms for limiting access to specific object types within the database. If you have access, you have access to everything.	Excellent. You can control the objects a user has access to, as well as control read/write/update and grant privileges.
Scalability	<ul style="list-style-type: none"> • Limited due to the fact that there is no control over the depth of the object tree instantiated. • Fine for in-memory solutions 	<ul style="list-style-type: none"> • Depends on the underlying database and the overhead incurred by the Hibernate mapping layer. 	<ul style="list-style-type: none"> • Unknown 	<ul style="list-style-type: none"> • Excellent. Can run on a laptop or on clustered servers.
Supportability	Good due to the extreme simplicity of the implementation	Depends on database vendor and support contract.	Good	Excellent admin tools and 24x7x365 support
Miscellaneous				Schema design supports inheritance and composition Can use Jalapeño to persist your objects to relational databases as well as to Caché

Or you can specify the activation depth when selecting/updating/deleting:

```
SensorReadout readout=car.getHistory();
while(readout!=null) {
db.activate(readout,2); // Activate the next 2 object levels!!!
System.out.println(readout);
readout=readout.getNext();
}
```

Thus you can specify: “Get the next three levels only” if desired or “Get everything!”

CAUTION: DB4O doesn't enforce referential integrity, so be very careful when deleting with cascade delete enabled. You can delete objects that are still pointed to by other objects in the database.

Caché

Caché provides good control over object depth as well. You can specify FetchType = Eager or Lazy like Hibernate. Calling the “detach(Object)” method ensures that all data in the given object (including all objects reachable from it by following references) can be accessed without a connection to the database.

Once again, if you switch to JDBC/SQL, you have total control over object depth via SQL JOIN statements, but can still open the objects referenced in the resultset.

Controlling Object Property Breadth

You may also want to limit how much data you retrieve from a given object. A complex object may consist of 20, 30, or even 50 or more properties, including embedded objects and lists or arrays of objects. What if you only need access to one or two of those properties? Isn't it overkill to instantiate the entire object, populating 50 properties in order to get two of them? If you're retrieving a list of such objects, you could end up with an array of 200 objects – along with all 50 of their properties – when all you need or want is one or two properties per object.

What's needed is a mechanism for controlling object property coverage.

One approach is to define a POJO that only defines a subset of the properties in the original object and populate it. This is where a relational database has an advantage over an object database. You can use JDBC to retrieve just the data you want and populate the POJO. “SELECT firstName, LastName FROM PERSON WHERE ID = 1”

Hibernate

Since Hibernate is typically backed by a relational database, it provides excellent control over your object property coverage. You can define a SELECT statement that only retrieves the properties you're interested in and provides them to you as a Java List or a List of Object Arrays.

DB4O

DB4O is a pure object database, so you must instantiate the object to access its properties. DB4O provides no help for you on this score.

Caché

Besides the Object interface that Caché provides, it also provides an SQL projection or interface. This lets you access objects as if

they were tables and columns in a relational database. Using this method, Caché provides excellent control over your object property breadth.

Object Tree Traversal

Once you have your objects, you want to be able to traverse the object tree. For example, if you start with an employee, you want to be able to access the company name as follows:

```
employee.getDepartment().getCompany().getName();
```

This is one of the most powerful features of object-oriented development and one of the strongest arguments for using an object database.

Hibernate

Unfortunately, relational databases provide virtually no support for this type of functionality – normally you'd have to issue SQL SELECTs to retrieve additional data as needed and create the associated POJOs. Fortunately, Hibernate provides this functionality for you by providing a proxy object to fetch additional mapped objects as needed. This works as long as you have a valid Hibernate session object available.

DB4O and Caché

Since they are object databases to begin with, DB4O and Caché handle this with aplomb. As you make calls to related objects, they're automatically retrieved from the database. Thus you can access your objects as follows:

```
employee.getDepartment().getCompany().getName();
```

```
List myEmployees = department.getEmployees();
```

And so on.

Enforcement of Referential Integrity

How can you ensure you don't delete an object that other objects depend on? For example, can you delete a department if the employees still exist? Does it support cascade deletes/updates?

Hibernate

While you can define relationships using Hibernate annotations, the actual support and enforcement of referential integrity is dependent on the database used on the back-end.

DB4O

DB4O currently doesn't enforce referential integrity.

Caché

Caché provides full support for maintaining referential integrity, as well as for performing cascade updates and deletes. This can be controlled based on the way you define the relationships. One-to-many relationships enforce referential integrity, but don't perform cascade updates and deletion. Parent-Child relationships enforce referential integrity and provide cascade update and deletion functionality as well.

Enforcement Of Uniqueness

How do you ensure that specific properties are unique in the database, such as Social Security numbers?

Hibernate

Support for marking properties unique and enforcing it is supported by most databases used as a back-end for Hibernate.

DB4O

DB4O currently doesn't provide any support for enforcing uniqueness. This feature is currently undergoing beta testing and the tentative release date is round mid-summer 2007.

Caché

You can mark as many properties unique as you want.

Support for Indices

Can you define indices that will enhance the performance of your queries?

Hibernate

Use the annotation `@Index` to cause an index to be created for the specified column or columns.

DB4O

You can define indexes in your DB4O configuration method before you open the object container. For example:

```
Db4o.configure().objectClass(Foo.class).objectField("bar").
indexed(true);
```

Caché

Caché provides powerful indexing options. Besides specifying that the property must be unique, you can specify the following index types:

```
type = "" (default standard index), bitmap, bitslice, index, and key.
For example:
```

```
@Indices({
    @Index(name="IndexOnName", columnNames={"Name"}),
    @Index(name="IndexOnSSN", type="bitmap", columnNames={"SSN"})
})
```

BitMap indices provide extremely high performance filtering for columns that have a limited number of possible values (such as categories) or which have a fixed number of characters (such as Social Security Numbers).

Enforcement of Property Constraints

Can you limit or control the values that will be saved to the database?

Hibernate

With Hibernate, you are limited to specifying that a property be NOT NULL or UNIQUE, although you may be able to specify constraints in the underlying database.

DB4O

DB4O doesn't provide any support for constraints.

Caché

Caché provides `@PropertyParameter` and `@PropertyParameters` annotations so you can control the values entered into a property.

You can specify a maximum value, minimum value, and even an input pattern.

```
@PropertyParameter (name = "PATTERN", value = "3N1\~\~2N1\~\~4N")
public String ssn;
```

```
@PropertyParameter (name = "MINVAL", value = "0")
public float balance;
```

Security and Access Control

How do you control who has access to the data and what they can do with it?

Hibernate

This can be performed programmatically or via the underlying database.

DB4O

You can encrypt and password-protect the database file, but there are no other user access controls. Once you have access to the database, you have access to all the data in it. If you want to control access to specific data or objects in the database, this can only be done programmatically.

Caché

This can be done programmatically or via the Caché System Management Portal. You can specify which objects the user has access to as well as the level of access (ALTER, SELECT, INSERT, UPDATE, DELETE, and REFERENCES)

Portability

How portable is the solution? Is there a vendor tie-in?

Hibernate

By definition and purpose, Hibernate helps make your application database-independent – so long as you stick to standard SQL and don't use database-specific functionality. This can be useful if you want to prototype your application on a lightweight database and move it to a more robust database later at production, however I feel that it's typically better to match your development environment to the production environment as much as possible. I've also seen very few instances where an application has been migrated to another database except in extreme legacy systems.

DB4O

With DB4O, you could say there's a vendor tie-in, but you can always add Hibernate annotations to your POJOs and run a script to read in your data from DB4O and save it to your Hibernate persistence layer.

Caché/Jalapeño

Caché stores objects using sparse arrays, so it's not your typical relational database. However, you can access data as objects or via Caché's SQL projection – which makes it look and act like a relational database (to your JDBC applications at any rate). Interestingly enough, you can also use Jalapeño to export your Caché class schema to a DDL file that can be imported into a relational database. You can then use Hibernate to map your objects to the new relational schema – or continue to use the Jalapeño Object Manager to interact with the new data source. The Object Manager automatically uses object persistence methods (Open, Save,

New, Delete) when accessing Caché, and relational persistence methods (Select, Update, Insert, Delete) when it's configured to connect to a relational database.

Conclusion

While you can eliminate mapping your objects to relational tables altogether, using DB4O or Caché, for example, it appears that some work must always be done if you want to take advantage of advanced database/datastore features such as enforcing referential integrity and uniqueness.

Hibernate has come a long way since it was first released. It has a bewildering number of options for configuring your object persistence mappings and behavior – as well as great tools to make it if not painless then at least not so painful.

If you want to quickly persist your objects for a small project and you can manage uniqueness and referential integrity within your application code - look no further than DB4O. It just doesn't get any easier.

The Caché/Jalapeño combination provides a compelling option for quickly persisting your Java objects with a minimum of effort while providing excellent control over database-specific functionality.

While you were busy programming your last tour de force, your peers and technology vendors have been busy building tools that enable you to do things that were previously impossible. You owe it to yourself and to your clients to pause once in a while and survey the state-of-the-art in databases and development tools to see where new entries can save you time and effort. For a comparison of features, go to the online version of this article at <http://jdj.sys-con.com>.

Resources

- Comparative Study of Persistence Mechanisms for the Java Platform. <http://research.sun.com/techrep/2004/abstract-136.html>
- Mark Weisfeld. *The Object-Oriented Thought Process*. SAMS Publishing. This is an excellent analysis of what object-oriented design is all about and how it compares to procedural programming.

Java Serialization:

- Discover the secrets of the Java Serialization API. <http://java.sun.com/developer/technicalArticles/Programming/serialization/>
- Bruce Eckell. *Thinking in Java*. <http://www.mindview.net/Books/TIJ/>

Hibernate:

- Web site: <http://www.hibernate.org/>
- Dave Minter and Jeff Linwood. *Beginning Hibernate*. Apress. 2006.

InterSystems' Caché Database:

- Web site: <http://www.intersystems.com/Cache>
- Jalapeño: <http://www.intersystems.com/Jalapeno>

DB4O:

- Web site: <http://www.db4o.com/>
- Simple Object Persistence with the db4o Object Database. <http://www.onjava.com/pub/a/onjava/2004/12/01/db4o.html>

INTERSYSTEMS

www.InterSystems.com