

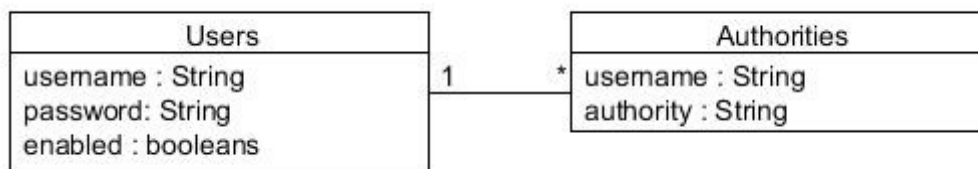
Data Access Layer Design for Java Enterprise Applications

Enterprise Application Development in Java

Java Database Connectivity (JDBC) can be used to persist Java objects to databases. However JDBC is verbose and difficult to use cleanly and therefore is not really suitable for enterprise scale applications. In this article I will demonstrate how to replace JDBC persistence code with an Object Resource Mapper to reduce its verbosity and complexity and then, through the use of the appropriate patterns, show how you might design a more complete data access layer for a Java enterprise application.

The Domain Model

In order to demonstrate the design of an Enterprise Application Data Access Layer I am going to develop a solution to manage users and their roles (authorities) for the Spring Security database tables as described in Spring in Action [SpringInAction]:



Spring Security expects two tables. The `Users` table holds a list of users consisting of a user name, password and flag to indicate whether or not the user is enabled. It has a one-to-many relationship with the `Authorities` table which holds the list of roles, stored as strings, each user has. Although Spring Security does not specify this to be enforced, it is sensible to use the `username` column as a primary key in the `Users` table and as a foreign key in the `Authorities` table (see source listing 1).

A Java abstraction of a Spring Security user might look something like this:

```

public class User
{
    private String username;
    private String password;
    private boolean enabled;
    private List<String> auths = new ArrayList<String>();

    public User(String username, String password, boolean enabled)
    {
        this.username = username;
        this.password = password;
        this.enabled = enabled;
    }

    public void addAuth(String auth)
    {
        auths.add(auth);
    }
}
  
```

```
}

public String getUsername()
{
    return username;
}

public String getPassword()
{
    return password;
}

public boolean isEnabled()
{
    return enabled;
}

public List<String> getAuths()
{
    return auths;
}
}
```

A `User` object has a user name, password, enabled flag and a list of roles. It has a constructor which initialises all of the fields, bar the roles, a method for adding individual roles and the appropriate getter for all fields.

The `User` class is the main object in our domain model.

Persisting Objects with JDBC

Once a `User` object is instantiated with a user name, password, enabled status and a list of roles it is relatively straight forward, although verbose, to persist the object using JDBC:

```
public static void save(User user, Connection con) throws SQLException
{
    try
    {
        con.setAutoCommit(false);

        final PreparedStatement userDetails
            = con.prepareStatement("{call [dbo].spSaveUser(?,?,?)}");

        final PreparedStatement deleteAuthorities
            = con.prepareStatement("{call [dbo].spDeleteAuthorities(?)}");

        final PreparedStatement saveAuthority
            = con.prepareStatement("{call [dbo].spSaveAuthority(?,?)}");

        userDetails.setString(1, user.getUsername());
        userDetails.setString(2, user.getPassword());
        userDetails.setBoolean(3, user.isEnabled());
        userDetails.execute();

        deleteAuthorities.setString(1, user.getUsername());
```

```
deleteAuthorities.execute();

saveAuthority.setString(1, user.getUsername());

for(String auth : user.getAuths())
{
    saveAuthority.setString(2,auth);
    saveAuthority.execute();
}

con.commit();
}
finally
{
    con.setAutoCommit(true);
}
}
```

The save method makes a few assumptions:

1. The `JDBC Connection` object, `con` is initialised before and cleaned up after the method call by other code.
2. The `PreparedStatement` objects can wait around to be cleaned up when the `con` object is cleaned up.
3. The stored procedures `spSaveUser`, `spDeleteAuthorities` and `spSaveAuthority` (see source listing 2) exist in the default database specified when the `con` object is initialised.

For more details on JDBC resource handling see Boiler Plating Database Resource Cleanup - Part I [BPDR1].

The `save` method takes the JDBC connection out of automatic commit mode, so that all the database operations occur within a transaction. The `finally` block at the end ensures that the connection is put back into automatic commit mode regardless of whether the operations succeeded or not. This, as I'll explain in the moment, is because the `User` table and the `Authorities` tables are updated separately and the changes should only be committed if both updates are successful.

The `User` table is updated first by getting the user name, password and enabled status from the `User` object and passing them to the `spSaveUser` stored procedure. The `spDeleteAuthorities` stored procedure is then used to remove all of the existing roles for the user from the `Authorities` table and finally the `spSaveAuthority` stored procedure is used in a loop to write the new roles to the table.

The `load` method, which is only slightly less verbose than the `save` method, is used to instantiate a `User` object from the database:

```
public static User load(String username, Connection con) throws SQLException
{
    final PreparedStatement getUser
        = con.prepareStatement("{call [dbo].spGetUser(?)}");
```

```
final PreparedStatement getAuthorities
    = con.prepareStatement("{call [dbo].spGetAuthorities(?)}");

getUser.setString(1, username);
getAuthorities.setString(1, username);

ResultSet rs = getUser.executeQuery();

User user = null;

if (rs.next())
{
    user = new User(rs.getString(1), rs.getString(2), rs.getBoolean(3));

    rs = getAuthorities.executeQuery();
    while(rs.next())
    {
        user.addAuth(rs.getString(1));
    }
}

return user;
}
```

This `load` method makes the same assumptions as the `save` method, plus the existence of the `spGetUser` and `spGetAuthorities` stored procedures. There is no need for any transaction handling as the database is only being read.

The `spGetUser` stored procedure is used to get the user name, password and enabled status from the `Users` table if an entry for the specified user name exists. If it does then the `spGetAuthorities` stored procedure is used to get the user's roles and insert them into the `User` object.

On the surface the `save` and `load` methods look like simple, serviceable JDBC code, but in reality they are unnecessarily verbose and potentially difficult to maintain. An alternative is to use an Object Resource Mapper (ORM).

Object Resource Mapper

Using an ORM, such as Hibernate [Hibernate], can greatly reduce the amount of persistence code required. For example the `save` method could be reduced to:

```
public static void save(User user, SessionFactory sessions)
{
    final Session session = sessions.openSession();
    final Transaction tx = session.beginTransaction();

    try
    {
        session.saveOrUpdate(user);
        tx.commit();
    }
    finally
    {

```

```

        session.close();
    }
}

```

Of course there is slightly more to it than I have shown here, but I'll get to that shortly. The `SessionFactory` object passed into the method, among other things, manages connections to the database, which are served up as `Session` objects via the `openSession` method. The Hibernate notion of a session is somewhere between a connection and a transaction. Sessions must be closed regardless of success or failure and the most sensible place to do this is in a `finally` block. A transaction is started by calling `beginTransaction` on a `Session` object and committed by calling `commit` on the returned `Transaction` object. An object is persisted to the database by passing it to the `saveOrUpdate` method. If a row in the database with the same user name as the `User` object already exists it is updated, if one does not exist it is created.

As you can see there is no need to update the user and their roles separately, Hibernate takes care of all of that for you. There is also no need to write SQL or call stored procedures unless you want to, Hibernate does all that for you too. We'll see how after we've looked at the `load` method:

```

public static User load(String username, SessionFactory sessions)
{
    final Session session = sessions.openSession();
    final Transaction tx = session.beginTransaction();

    User user = null;

    try
    {
        user = (User) session.get(User.class, username);
        tx.commit();
    }
    finally
    {
        session.close();
    }

    return user;
}

```

The `load` method works in much the same way as the `save` method, except instead of calling `saveOrUpdate` to persist an object it calls `get` to retrieve an object. The `get` method needs to know the type of the object it is retrieving and the object's ID, in this case `username`. The returned object is then cast to the correct type (hopefully generics will appear in a later version and there'll be no need for the cast).

All of this relies on a properly configured `SessionFactory` object. Hibernate uses its own `Configuration` object which itself can be configured in lots of different ways, to create `SessionFactory` objects. The most straight forward way to configure it is with a `hibernate.properties` file and a Hibernate XML mapping file:

```
final SessionFactory sessions = new Configuration()
    .addClass(User.class)
    .buildSessionFactory();
```

Unless specified otherwise the configuration object looks for the `hibernate.properties` file in the classpath. A basic `hibernate.properties` file for a Microsoft SQL Server database looks something like this:

```
hibernate.connection.url=jdbc:sqlserver://localhost;DatabaseName=DataAccessLayer
hibernate.connection.username=user
hibernate.connection.password=secret
hibernate.connection.driver_class=com.microsoft.sqlserver.jdbc.SQLServerDriver
hibernate.dialect = org.hibernate.dialect.SQLServerDialect
```

The connection url, username, password and driver are all self explanatory and the same as used by JDBC. The setting that is new is the dialect. Hibernate needs to know what sort of database it is connecting to so that it can generate the appropriate SQL and take advantage of any customisations. The dialect is set by specifying one of a number of different dialect objects supplied by Hibernate. It is also possible to write custom dialect objects for any database not supported.

The `Configuration` object also needs to know about the classes you want to persist to the database. Again there are lots of different ways to do this, but the simplest is to have a Hibernate XML Mapping file for each class along side it in the package. Using the `addClass` method to tell the `Configuration` object about the `User` class tells it to look for `User.hbm.xml` in `classpath:/uk/co/marauder/dataaccesslayer/model`. `User.hbm.xml` looks like this:

```
<?xml version="1.0"?>
<!DOCTYPE
    hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="uk.co.marauder.dataaccesslayer.model.User" table="Users">
        <id name="username" column="username">
            <generator class="assigned"/>
        </id>

        <property name = "password" column = "password"/>
        <property name = "enabled" column = "enabled"/>

        <bag name="auths" table="Authorities" lazy="false">
            <key column="username" />
            <element type="java.lang.String" column="authority"/>
        </bag>
    </class>
</hibernate-mapping>
```

Hibernate XML mapping files are very easy to understand. As you can see the Java `User` class is mapped to the `Users` database table. All Hibernate persistable objects need an ID, which is specified by the `ID` tag. In this case `username` is used as the ID and is

mapped to the `username` column in the `Users` table. Hibernate will use the `name` attribute to work out what the getter on the `User` object is called, in this case `getUsername`. The `generator` tag is used to specify if and how Hibernate should generate the ID for objects being saved for the first time. In this case we want to specify the ID ourselves, so the `generator` type is `assigned` and the ID is given to the `User` object before it is persisted. The `property` tag is used to map the other fields to table columns. The `bag` tag tells Hibernate that the `User` class has a list of strings that should be written to the `authority` column in the `Authorities` table and that the foreign key relating them to the `User` object is `username`.

Obviously this is quite a simple mapping. Hibernate is capable of much more complicated object relationships including inheritance. Hibernate XML mapping files are not the only way to tell Hibernate about the classes you want to persist. Hibernate also provides some annotations, but I prefer to keep domain objects decoupled from the persistence mechanism as much as possible.

We're almost there. Unfortunately Hibernate needs a default constructor and won't use `User`'s current constructor. When loading objects it default constructs them and uses setters to initialise them. `User` doesn't have setters so they need to be added:

```
public class User
{
    private String username;
    private String password;
    private boolean enabled;
    private List<String> auths = new ArrayList<String>();

    @SuppressWarnings("unused")
    private User()
    {}

    ...

    @SuppressWarnings("unused")
    private void setUsername(String username)
    {
        this.username = username;
    }

    @SuppressWarnings("unused")
    private void setPassword(String password)
    {
        this.password = password;
    }

    @SuppressWarnings("unused")
    private void setEnabled(boolean enabled)
    {
        this.enabled = enabled;
    }

    @SuppressWarnings("unused")
    private void setAuths(List<String> auths)
    {
```

```
        this.auths = auths;
    }
}
```

Hibernate is quite clever, it can find and call the constructors and methods it needs by reflection, even if they are private. So if it is undesirable for your domain objects to have a default constructor or particular methods that are needed by Hibernate they can be made private.

This is all that is needed for a fairly significant reduction in the amount of code that is needed. It also reduces the the amount of exception handling. However, it is possible to reduce it even further using the Spring Framework's [Spring] `HibernateTemplate` class:

```
public static void save(User user, HibernateTemplate hibernateTemplate)
{
    hibernateTemplate.saveOrUpdate(user);
}

public static User load(String username, HibernateTemplate hibernateTemplate)
{
    return (User) hibernateTemplate.get(User.class, username);
}
```

`HibernateTemplate` takes care of getting and releasing Hibernate sessions and some of the transaction handling. It would normally be initialised with a `DataSource` in the Spring runtime's application context, but if you're not using Spring's application context it can be initialised with a `SessionFactory`:

```
final SessionFactory sessions = new Configuration()
    .addClass(User.class)
    .buildSessionFactory();

final HibernateTemplate hibernateTemplate
    = new HibernateTemplate(sessions);
```

I have demonstrated how the `load` and `save` methods for the `User` object can be reduced from several lines of code for a JDBC solution, down to a single line each with an ORM solution. Even with the Hibernate XML mapping file this is a significant reduction in code and complexity. However, this is not enough for a real Enterprise Application Data Access Layer.

Data Mappers, Registries and Managers

The `load` and `save` methods separate the concern of persisting a `User` object from the object itself, and that is a good thing. However the user of the methods still knows they are persisting to a database and the mechanism used to persist to the database as they have to provide the appropriate `Connection`, `SessionFactory` or `HibernateTemplate` object. Worse still, when writing a unit test that wants to simulate persisting data to the database those very same `Connection`, `SessionFactory` or `HibernateTemplate` objects must be stubbed out. This is not a trivial exercise as each

has many methods and acts as a factory for other objects that must also be stubbed.

Fortunately Martin Fowler solves this problem with two patterns from his Patterns of Enterprise Application Architecture [PoEAA] book. The Data Mapper, which moves data between objects and the database while keeping them independent of each other and the mapper itself and the Registry, which is a well known object that other objects can use to find common objects and services.

The Data Mapper is an object that you can ask to load, save or perform some other sort of persistence operation on an object. In practice creating a Data Mapper for the `User` class is slightly more than refactoring the `load` and `save` methods into an class:

```
public class UserHibernateDataMapper
{
    private final HibernateTemplate hibernateTemplate;

    public UserHibernateDataMapper(HibernateTemplate hibernateTemplate)
    {
        this.hibernateTemplate = hibernateTemplate;
    }

    public void save(User user)
    {
        hibernateTemplate.saveOrUpdate(user);
    }

    public User load(String username)
    {
        return (User) hibernateTemplate.get(User.class, username);
    }
}
```

The `UserHibernateDataMapper` class is specific to `HibernateTemplate`, but similar classes could be written for a straight forward `JDBC Connection` or for a `Hibernate SessionFactory`. Once created and initialised with a `HibernateTemplate` the object can be passed around as needed and used to persist `User` objects without the user being aware of the implementation.

Spring also provides a helper class for Data Mappers called `HibernateDaoSupport` which, among other things, provides accessors for the `HibernateTemplate`:

```
public class UserHibernateDataMapper extends HibernateDaoSupport
{
    public UserHibernateDataMapper(HibernateTemplate hibernateTemplate)
    {
        setHibernateTemplate(hibernateTemplate);
    }

    public void save(User user)
    {
        getHibernateTemplate().saveOrUpdate(user);
    }
}
```

```
public User load(String username)
{
    return (User) getHibernateTemplate().get(User.class, username);
}
```

For unit testing, data mappers that talk directly to a database need to be easily interchangeable with the equivalent mock objects. Mock objects are simulated objects that mimic the behaviour of real objects.

The easiest way to make the real data mappers and the mock data mappers interchangeable is for them both to implement the same interface. This is achieved by Extracting the Interface [Refactoring] of the real data mapper and then creating a mock implementation:

```
public interface UserDataMapper
{
    void save(User user);

    User load(String username);
}

public class UserHibernateDataMapper
    extends HibernateDaoSupport
    implements UserDataMapper
{
    public UserHibernateDataMapper(HibernateTemplate hibernateTemplate)
    {
        setHibernateTemplate(hibernateTemplate);
    }

    @Override
    public void save(User user)
    {
        getHibernateTemplate().saveOrUpdate(user);
    }

    @Override
    public User load(String username)
    {
        return (User) getHibernateTemplate().get(User.class, username);
    }
}

public class MockUserDataMapper implements UserDataMapper
{
    private Map<String,User> users = new HashMap<String,User>();

    @Override
    public User load(String username)
    {
        return users.get(username);
    }

    @Override
    public void save(User user)

```

```
{
    users.put(user.getUsername(), user);
}
```

Of course you can have as many implementations of the interface as you want, so you could have a Hibernate implementation, a JDBC implementation and a mock implementation for use in different systems if you wanted too.

Each data mapper only needs to be created once. You could create and destroy them when they are needed, but then the underlying JDBC or Hibernate object would need to be passed around instead and the user would no longer be abstracted from the particular data access implementation being used. However, passing individual data mappers to everywhere they are needed can be tedious and it is much easier to pass around a single object that can be asked for the required data mapper.

Martin Fowler's Registry pattern describes one such object. Just to remind you, the Registry is a well known object that other objects can use to find common objects and services. Patterns describe solutions to problems, not implementations. Therefore the registry used to store and retrieve data mappers can be very different to the example Fowler suggests. The implementation I have chosen is this:

```
public interface DMRegistry
{
    <T> T get(Class<T> interfaceType, Class<?> object);
}
```

Having a registry interface makes the interchanging of different implementations of registries easier and the passing round of registries less coupled. For example you might have a JDBC data mapper registry, a Hibernate data mapper registry and a mock data mapper registry. If your code using the data mappers takes a `DMRegistry` you can easily change the implementation just by passing the required registry to it.

```
public abstract class AbstractDMRegistry implements DMRegistry
{
    private final Map<Class<?>, Object> map = new HashMap<Class<?>, Object>();

    public <T> void add(final Class<?> objectType, final T dataMapper)
    {
        map.put(objectType, dataMapper);
    }

    @Override
    public <T> T get(Class<T> interfaceType, Class<?> object)
    {
        final T dataMapper = interfaceType.cast(map.get(object));
        if(dataMapper == null)
        {
            throw new IllegalStateException( object
                " not found in registry "
                + getClass().getName());
        }
    }
}
```

```

        return dataMapper;
    }
}

```

`AbstractDMRegistry` is an abstract class that holds the common implementation for all data mapper registries. All data mappers are identified by the type of the object they map. The `add` method is used to map the object type to a data mapper instance. The `get` method takes the expected interface for the data mapper and the object type, looks it up and returns the appropriate data mapper. If a data mapper for the supplied object type is not present an exception is thrown.

Finally you need a specific implementation of the data mapper registry. A `HibernateTemplate` implementation might look like this:

```

public class HibernateTemplateDMRegistry extends AbstractDMRegistry
{
    public HibernateTemplateDMRegistry(HibernateTemplate hibernateTemplate)
    {
        add(User.class, new UserHibernateDataMapper(hibernateTemplate));
    }
}

```

The data mapper for the `User` object is created using a supplied `HibernateTemplate`, in the constructor and added to the registry. Any number of data mappers can be added to and accessed from the registry. Other implementations of a registry would be very similar, but instantiate different data mappers. A `HibernateTemplateDMRegistry` is instantiated like this:

```
final DMRegistry registry = new HibernateTemplateDMRegistry(hibernateTemplate);
```

and used like this:

```

final UserHibernateDataMapper mapper = registry.get(
    UserHibernateDataMapper.class,
    User.class);
mapper.save(user);

final User newUser = mapper.load(user.getUsername());

```

Once the data mapper has been retrieved from the registry it is simple to use, but the code to retrieve it is verbose and could end up being repeated throughout your code. One way to get around this is to use a Facade [GoF] to “manage” the object being persisted:

```

public class UserManager
{
    private final DMRegistry registry;

    public UserManager(DMRegistry registry)
    {
        this.registry = registry;
    }

    private UserHibernateDataMapper getMapper()
    {

```

```
        return registry.get(UserHibernateDataMapper.class, User.class);
    }

    public void save(User user)
    {
        getMapper().save(user);
    }

    public User load(String username)
    {
        return getMapper().load(username);
    }
}
```

The `UserManager` uses a reference to data mapper registry to obtain the data mapper for the `User` object and uses it to save and load `User` objects. A `UserManager` can be passed around instead of a data mapper registry and reduce the verbosity and repetition of code without losing any of the flexibility and does not care what sort of data mapper registry it has been passed. More operations can be added to managers easily and if an operation becomes more complicated and, for example, requires multiple data mapper calls the manager becomes the ideal place to add things like encompassing transactions.

Finally

I have shown how to reduce the verbosity and complexity of JDBC persistence code using an ORM. I have also shown how a data access layer could be written that is suitable for an enterprise application. It allows simple interchanging of different database persistence implementations, including a mock object implementation to aid automated unit testing.

By using the basic form of the Spring Security database I have demonstrated very simple use of an ORM and object managers. ORMs can be used to persist far more complicated object relationships and managers used to do far more. Although it is only the tip of the iceberg this article should give a firm grounding for more complex enterprise application data access layers.

Source Listings

Listing 1: Spring Security Tables

```
CREATE TABLE [dbo].[Users]
(
    [username] [varchar] (50) NOT NULL UNIQUE,
    [password] [varchar] (50) ,
    [enabled] [bit] NOT NULL,

    CONSTRAINT Pk_Users PRIMARY KEY CLUSTERED ([username] ASC)
) ON [PRIMARY];

CREATE TABLE [dbo].[Authorities]
(
    [username] [varchar] (50) NOT NULL,
    [authority] [varchar] (50) NOT NULL,

    CONSTRAINT Pk_Authorities
        PRIMARY KEY CLUSTERED ([username],[authority] ASC),
    CONSTRAINT Fk_Authorities_User FOREIGN KEY ([username])
        REFERENCES [Users] ([username]),
) ON [PRIMARY];
```

Listing 2: Spring Security Stored Procedures

```
CREATE PROC [dbo].spSaveUser
(
    @username [varchar] (50),
    @password [varchar] (50),
    @enabled [bit] )
AS

IF EXISTS(SELECT [username] FROM [dbo].[Users] WHERE [username] = @username )
    UPDATE [dbo].[Users] SET [password] = @password, [enabled] = @enabled
    WHERE [username] = @username
ELSE
    INSERT INTO [dbo].[Users] ([username],[password],[enabled])
VALUES (@username,@password,@enabled)

CREATE PROC [dbo].spDeleteAuthorities
( @username varchar(50))
AS
DELETE FROM Authorities WHERE [username] = @username;

CREATE PROC [dbo].spSaveAuthority
( @username varchar(50),
  @authority varchar(50))
AS
INSERT INTO Authorities ([username],[authority]) VALUES
(@username,@authority);
```

References

[SpringInAction] Spring in Action by Craig Walls, Ryan Breidenbach, Manning Publications; 2 edition ISBN: 978-1933988139

[BPDR1] Boiler Plating Database Resource Cleanup - Part I by Paul Grenyer, http://www.marauder-consulting.co.uk/Boiler_Plating_Database_Resource_Cleanup_-_Part_I.pdf

[Hibernate] Hibernate: <https://www.hibernate.org/>

[Spring] Spring Framework: <http://www.springsource.org/>

[PoEAA] Patterns of Enterprise Application Architecture by Martin Fowler, Addison Wesley ISBN: 978-0321127426

[Refactoring] Refactoring: Improving the Design of Existing Code by Martin Fowler, Addison Wesley, ISBN: 978-0201485677

[GoF] Design patterns : elements of reusable object-oriented software by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison Wesley, ISBN: 978-0201633610