Eclipse Web Tools Platform – Uncovered: Java Persistence Development with Dali and EclipseLink

Neil Hauge
niel.hauge@oracle.com

Tom Ware
tom.ware@oracle.com
Tutorial Goals

- Review core Java Persistence API (JPA) features
- Learn how to use Dali to build JPA applications
- Become familiar with the benefits of EclipseLink and its extended functionality
Agenda

- Introduction to EclipseLink and Dali
- Essential JPA and Dali
- Queries and Testing
- Adding Relationships, Inheritance
- Advanced Functionality with EclipseLink
- Development Scenarios
- EJB 3.0 and JPA with WTP
Eclipse Persistence Services - EclipseLink

- Complete functionality of Oracle TopLink
  - Object Relational Engine
  - JPA Implementation
  - MOXy – Mapping Objects to XML
  - SDO
  - DBWS
  - Mapping Workbench
- JPA implementation based on TopLink Essentials, the JPA 1.0 Reference Implementation
- OSGI bundles planned for 1.0 Release
EclipseLink

- Code base very mature
- Incubation Technology Project
- Planning Graduation and 1.0 release this summer
- Monthly milestone builds
Dali JPA Tools Project of WTP

- Support for the definition, editing, and deployment of Object-Relational (O/R) mappings for JPA Entities

- Provides generic functionality for any JPA 1.0 compliant implementation

- 1.0.2 Release currently available, bundled with WTP/Europa

- Integrated with WTP to provide easier use of JPA in JEE applications

- Adopted by numerous JPA tooling providers
Dali JPT - Goals

- Simplicity
  - Intelligent mapping assistance and automated generation
- Intuitiveness
  - Use existing and consistent modeling and tooling practices in Eclipse
  - Light-weight views offer assistance but don’t get in the way of power users
- Compliance
  - Support any and all EJB 3.0 compliant runtime implementations
  - Test using EJB 3.0 Reference Implementation
- Extensibility
  - Provide the ability for vendors and open source projects to seamlessly add their own value-add features
Dali JPT - Features

Major features include

- Rich UI for configuring entities
- JPA Validation
- Entity generation from tables
- JPA annotation code completion
- JPA annotation and XML mapping file support

- New features planned for 2.0
  - Persistence.xml editing
  - Project Explorer contribution
  - UI improvements
Why do you need JPA Tools?

- JPA runtime combines:
  - Java Classes
  - Mapping Metadata
  - Database schema
Why do you need JPA Tools?

- How can you tell if they all match?
  - Deploy and run tests?
    - slow
    - find one problem at a time (fix, run, fix, ...)
  - definitive
- Design time validation?
  - quick
  - finds all issues
  - validates against spec
  - runtime may not match spec
JPA—in a Nutshell

- A Java standard that defines:
  - how Java objects are stored in relational databases (specified using a standard set of mappings)
  - a programmer API for reading, writing, and querying persistent Java objects (“Entities”)
  - a full featured query language
  - a container contract that supports plugging any JPA runtime in to any compliant container.
JPA—POJO Entities

- Concrete classes
- No required interfaces
  - No required business interfaces
  - No required callback interfaces
- `new()` for instance creation
- Direct access or getter/setter methods
  - Can contain logic (e.g. for validation, etc.)
- “Managed” by an EntityManager
- Can leave the Container (become “detached”)
Object-Relational Mappings

- Core JPA Mappings
  - Id
  - Basic
  - Relationships
    - OneToOne
    - OneToMany/ManyToOne
    - ManyToMany
  - And more…
- Can be specified using Annotations or XML
Java POJO

```java
public class Address {

  private int id;
  private String street;
  private String city;
  private String state;
  private String country;

  ...
}
```
JPA Entity

@javax.persistence.Entity
public class Address {
    @javax.persistence.Id
    private int id;
    private String street;
    private String city;
    private String state;
    private String country;

    ...
}

ADDRESS

<table>
<thead>
<tr>
<th>ID</th>
<th>STREET</th>
<th>CITY</th>
<th>STATE</th>
<th>COUNTRY</th>
</tr>
</thead>
</table>

Java Persistence Development with Dali and EclipseLink | © 2008 by Oracle; made available under the EPL v1.0
Basic Mapping

- Persist primitive attribute types.
- Default attribute mapping is ‘Basic’
Basic Mapping

```java
@Entity
public class Address {
    @Id
    private int id;
    @Basic private String street;
    @Basic private String city;
    @Basic private String state;
    @Basic private String country;

    ...
}
```

- Don’t need @Basic since it is the default
Default Basic Mapping

```java
@Entity
public class Address {
    @Id
    private int id;
    private String street;
    private String city;
    private String state;
    private String country;

    ...
}
```
Physical Mapping

```java
@Entity
public class Address {
    @Id
    @Column(name="ID") private int id;
    @Column(name="STREET") private String street;
    @Column(name="CITY") private String city;
    @Column(name="STATE") private String state;
    @Column(name="COUNTRY") private String country;
}
```

- Don’t need @Column if column and attribute have the same name
Default Physical Mapping

```java
@Entity
public class Address {
    @Id
    private int id;
    private String street;
    private String city;
    private String state;
    private String country;

    ...
}
```
Dali Mapping Validation

- Java Source Editor enhancements
- Mapping Problem Markers

```
@Entity
public class Address {
    @Id
    private int id;
    private String street;
    private String city;
    private String province;
    private String country;
}
```

**ADDRESS**

<table>
<thead>
<tr>
<th>ID</th>
<th>STREET</th>
<th>...</th>
<th>STATE</th>
</tr>
</thead>
</table>

Default mapping won’t work!

- The column province cannot be found on table Address

Address.java
Dali Mapping Assistance for Basic Mapping

No Mapping Errors!
Dali Mapping Assistance

- JPA Details View

```java
@Entity
public class Address {
    @Id
    private int id;
    private String street;
    private String city;
    private String province;
    private String country;
}
```

**ADDRESS**

<table>
<thead>
<tr>
<th>ID</th>
<th>STREET</th>
<th>...</th>
<th>STATE</th>
</tr>
</thead>
</table>

**Persistence Properties**

- Map As: Default (Basic)
- Column:
  - Name: Default (province)
  - Table: Default (province)
  - Insertable: Country
  - Updatable: Default (true)

- Errors (1 item):
  - The column province cannot be found on table Address

Java Persistence Development with Dali and EclipseLink | © 2008 by Oracle; made available under the EPL v1.0
Persistence Unit

- Entities
  - Mapped Java classes
- Persistence XML
  - Configuration XML that describes the Persistence Units
    - Name
    - List of Entities
    - Database connection info
    - How it is managed
    - Provider specific information
- Mapping XML (optional)
  - XML file that describes how persistence unit is mapped
Deployment Configuration—persistence.xml

- META-INF/persistence.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence ...
  <persistence-unit name="default">
    <class>org.eclipse.dali.example.tracerbullet.Address</class>
    <properties>
      ...
      <property name="eclipselink.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver"/>
      <property name="eclipselink.jdbc.url" value="jdbc:derby://localhost:1527/dali;create=true"/>
      <property name="eclipselink.jdbc.user" value="dali"/>
      <property name="eclipselink.jdbc.password" value="dali"/>
    </properties>
    </persistence-unit>
  </persistence>
```
JPA Structure View

- Provides JPA Entity specific view of Java Class
- A thumbnail sketch of how an Entity is mapped
- Supports navigation between mappings
- Automatically adjusts to either property or field mapping
JPA Perspective—all your JPA Views

```java
@Entity
@Table(name="PETS")
public class Pet extends NamedEntity {

    @ManyToOne(fetch=LAZY)
    private Owner owner;

    @Temporal(TemporalType.DATE)
    private Date birthDate;
...
```
Integrated F1 Help

- In any Dali view you can hit F1 to get context specific help.

One-to-one mapping

Use a **One-to-One Mapping** to define a relationship with one-to-many multiplicity.

1. In the **Persistence Outline view**, select the field to map. The **Persistence Properties view** (for attributes) displays the properties for the selected.
2. In the Map As field, select **One-to-one**.
3. Use this table to complete the remaining fields on the **General** tab in Persistence Properties view.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Entity</td>
<td>The entity to which this attribute is mapped.</td>
<td>null</td>
</tr>
<tr>
<td>Cascade Type</td>
<td>See &quot;Cascade Type&quot; for details.</td>
<td>Default</td>
</tr>
<tr>
<td>Fetch Type</td>
<td>Defines how data is loaded from the database.</td>
<td>Eager</td>
</tr>
</tbody>
</table>

You do not need to explicitly specify the target entity, since it can be inferred from the type of object being referenced.
Creating a Database Connection

- Dali uses DTP for all DB services
Creating a Dali JPA Project

- Open the New Project Wizard
- Select the JPA Project and click "Next >"
Creating a Dali JPA Project

- Name the project
- Configure the Project
Synchronizing Entities with Persistence.xml

- In Java SE environment, persistence.xml must list the Entities—Dali offers synchronization

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com"
  <persistence-unit name="jdj"/>
</persistence>
```
Synchronizing Entities with Persistence.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xr"
    <persistence-unit name="jnj">
        <class>model.Customer</class>
        <class>model.CustomerInfo</class>
        <class>model.Invoice</class>
        <class>model.Phone</class>
    </persistence-unit>
</persistence>
```
Demo – Getting Started

- Create a Database Connection
- Create a JPA project
  - Configure Project to use EclipseLink
- Create an Entity
  - Configure Id mapping
  - Configure Basic Mappings
- Configure the Persistence XML
JPA Essentials: Querying

- Queries can be in two formats:
  - **JPQL**
    - Written against the object model, not the data model
    - Insulates you from data model changes
    - Enables query optimizations like cache usage
  - **SQL**
    - Couples your application code to the data model
    - JPA runtimes cannot leverage cache with SQL
    - Can be cumbersome mapping from SQL results to objects
JPQL Example

```java
Address actualAddress = (Address) em.createQuery("select a from Address a where a.street = :street and a.city = :city")
    .setParameter("street", expectedStreet)
    .setParameter("city", expectedCity)
    .getSingleResult();
```
JPQL Example—createQuery()

```java
Address actualAddress = (Address) em.createQuery(
    "select a from Address a where 
    "a.street = :street and " + 
    "a.city = :city"
)
```

- `EntityManager.createQuery(...)`
- `create a query object`
JPQL Example—setParameter()

Query.setParameter(name, value)

set a parameter on a query object

Address

"select a.street, a.city"

.setParameter("street", expectedStreet)
.setParameter("city", expectedCity)
.getParameter();
JPQL Example—getSingleResult()

```java
Address

Query.getSingleResult()  
➢ execute a query object and return the single expected result

"select a.street, a.city
  .setParameter("street", expectedStreet)
  .setParameter("city", expectedCity)
  .getSingleResult();
```
Hands On: Queries and JUnit

- Create a JUnit test
- Get an Entity Manager
- Run a JPQL Query
Inheritance

- Entities can extend:
  - Other Entities — concrete or abstract
  - Non-persistent classes — concrete or abstract

- Non-persistent super classes are mapped with @MappedSuperclass, not @Entity

- Inheritance Hierarchies can be mapped in three ways
  - Single table — all classes stored in the same table
  - Joined — Each class (concrete or abstract) stored in a separate table
  - Table per concrete class — Each concrete class stored in separate table (optional)
Inheritance—MappedSuperclass

- Entities may share a common parent from which they inherit persistent attributes that is itself not persistent.

```java
@MappedSuperclass
class BaseEntity {
  @Id private Integer id;
  ...

@Entity
public class Visit extends BaseEntity {
  ...
```
Inheritance—AttributeOverride

- Explicit or implicit column mappings from the MappedSuperclass are inherited by subclasses but can be overridden.

```java
@Entity
@AttributeOverrides( {
    @AttributeOverride(
        name = "id",
        column = @Column(name = "VISIT_ID"))
})
public class Visit extends BaseEntity {
...
```
Inheritance Hierarchies

```java
public abstract class Animal {
    int id;
    String name;
}

public class LandAnimal extends Animal {
    int legCount;
}

public class AirAnimal extends Animal {
    short wingSpan;
}
```
### Example Data Models

#### Single table:

<table>
<thead>
<tr>
<th>ANIMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
</tbody>
</table>

#### Joined:

<table>
<thead>
<tr>
<th>ANIMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LAND_ANML</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AIR_ANML</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
</tbody>
</table>

#### Table per Class:

<table>
<thead>
<tr>
<th>LAND_ANML</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AIR_ANML</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
</tbody>
</table>
JPA Relationship Mappings

- Single Entity
  - @OneToOne
  - @ManyToOne—inverse side of @OneToMany

- Collection of Entities
  - @OneToMany
  - @ManyToMany
ManyToMany Mapping

```java
@Entity
public class Invoice {
    @Id
    private int id;
    @ManyToMany
    private Customer cust;
    ...
}
```
ManyToOne Mapping

- JPA Details view can be used to select mapping type.
ManyToMany Mapping

- Target Entity defaults based on attribute Java type.
- Dali displays the JPA defined default join columns—you can override.
OneToMany Mapping

```java
@Entity
public class Invoice {
    @Id
    private int id;
    @ManyToOne
    private Customer cust;
    ...
}

@Entity
public class Customer {
    @Id
    private int id;
    @OneToMany(mappedBy="cust")
    private Set<Invoice> invoices;
    ...
}
```
OneToMany Mapping

```java
@OneToMany(mappedBy="cust")
private Set<Invoice> invoices;
```

Dali understands the JPA model (in this case the existing ManyToOne) and provides intelligent choices.
ManyToMany Mapping

```java
@Entity
public class Customer {
    @Id
    private int id;

    @ManyToMany
    private Collection<Phone> phones;
}

@Entity
public class Phone {
    @Id
    private int id;

    @ManyToMany(mappedBy="phones")
    private Collection<Customer> custs;
}
```
ManyToMany Mapping

- Dali detects if the default mapping doesn’t match the database schema.
ManyToMany Mapping

```java
@Entity
public class Customer {

    @ManyToMany
    @JoinTable(
        name = "CUST_PHONE",
        joinColumns = @JoinColumn(name = "cust_id",
                                 referencedColumnName = "id"),
        inverseJoinColumns = @JoinColumn(name = "phone_id",
                                          referencedColumnName = "id"))
    private Collection<Phone> phones;
```

```sql
<table>
<thead>
<tr>
<th>CUSTOMER</th>
<th>PHONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_PHONE</th>
<th>ID</th>
<th>PHON_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUST_ID</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
ManyToMany Mapping

- Use Dali to override Join Table with valid values.

```java
@ManyToMany
@JoinTable(
    name = "CUST_PHONE",
    joinColumns = @JoinColumn(name = "cust_id", referencedColumnName = "id"),
    inverseJoinColumns = @JoinColumn(name = "phone_id", referencedColumnName = "id")
)
private Collection<Phone> phones;
@OneToMany(mappedBy="cust")
private Set<Invoice> invoices;
```
Demo: MappedSuperclass and Relationships

- Add a OneToOne Relationship
- Add a OneToMany Relationship
- Add Inheritance
EclipseLink Caching

Entity Manager Factory → EntityManager

- UnitOfWork (TX Cache)
  - Client Session (Isolated Cache)
  - Server Session (Shared Cache)

RDBMS
EclipseLink Caching

- EntityManager contains its own cache
  - All items read into an EntityManager remain in the cache until they become detached or the EntityManager is closed
- L2 Cache on EntityManagerFactory
  - Configurable cache type and size
    - SoftWeak – default
    - HardWeak
    - Full
    - Soft
    - Weak
- EclipseLink provides a cache coordination feature
Locking types

EclipseLink Provides Additional Optimistic Locking types

- **ALL_COLUMNS**
  - Checks that no columns have changed prior to update
- **CHANGED_COLUMNS**
  - Ensures none of the columns being updated has changed prior to update
- **SELECTED_COLUMNS**
  - A specified set of columns are checked to ensure they have not been updated prior to executing an update

- Pessimistic Locking
  - Pessimistic Locking is supported through a Query Hint
Performance Features and Weaving

**Performance Features**

- Lazy Loading (defined in JPA Specification)
- Optimized Change Tracking
- Fetch Groups
- Optimization of Cache-look-ups

**Weaving**

- Weaving is used to enable these features
- Enabled by default in JavaEE 5 container
- Dynamic weaving can also be enabled with :javaagent
- Static Weaving
Additional Mapping Configuration

- Private Owned Mappings
  - Allow orphans to be removed
  - @PrivateOwned annotation on mapping
- BasicMap/BasicCollection
  - Not defined by the specification
  - Allow storing non-entity values in a map or collection
  - @BasicMap/@BasicCollection annotation
- Converter
  - Apply logic to values as they are read and written
  - @Converter and @Convert Annotations
Stored Procedure Calls

EclipseLink supports Calling Stored Procedures in JPA

@Named StoredProcedureQuery(
    name="callStoredProc"
    resultClass=test.Address.class
    procedureName="STORED_PROC"

    procedureParameters={
        @StoredProcedureParameter(procedureParameterDirection=IN_OUT,
        name="address_id_v", queryParameter="ADDRESS_ID", type=Long.class)

        @StoredProcedureParameter(procedureParameterDirection=OUT,
        name="street_v", queryParameter="STREET", type=String.class)
    }
)
Additional Functionality

- Query Hints
  - Caching
  - JDBC settings
  - Etc…
- Custom Annotations
  - Allow configuration of advanced features with annotations
- Customizers
  - Allow advanced session configuration
- More…
Demo: Extended Functionality

- Locking Configuration
- Private Owned
- Custom Converter
- Fetch Join on Mapping
Development Scenarios

Possible Approaches

- Meet in the Middle
  - existing object and data models
- Bottom Up
  - generate mapped object model from data model
- Top Down
  - generate data model from mapped object model
  - requires a JPA implementation in 1.0
Meet In the Middle (MITM)

- The advantage of MITM is that you can focus on getting your object model and data models correct.
  - Table != Class
    - 1 Table could be N classes (using embedded)
    - N tables could be 1 class (with secondary table)
  - Use Java language features like inheritance not present in relational model
- Dali’s validation makes MITM practical
  - Avoids map, deploy, debug cycle
  - Provides access to database schema to provide valid choices
Bottom Up

- Generate Entities from Tables
  - Great way to bootstrap a JPA application from an existing database
  - Uses an Entity == Table approach
  - Do it once and then modify the generated Entities
    - Dali mapping validation will help identify issues resulting from modifications to Entities
Top Down

- Generate DDL from Entities
  - Some support in Dali 0.5 release
  - Deprecated in favor of using full featured support implemented by JPA runtimes.
  - Extension point available to allow for plugging in runtime DDL generation.
Demo: Bottom up with Dali
WTP Development with Dali

- Possible Deployment Target Environments
  - Java SE
    - We’ve done this in our hands on labs.
    - Use JPA Bootstrap API
    - Use JPA Transaction API
    - persistence.xml has <class>es
  - Java 5 Web Application (not EE)
    - Same as SE
  - Java EE 5
    - Container integrates with JPA to simplify development
JPA in Java EE 5

- EJB 3 Session Beans support JPA
  - Dependency Injection of EntityManager
  - Declarative Transactions
    - You don’t use the EntityManager transaction API
- Container Finds Entities (no <class>es in persistence.xml)
- Can use data sources provided by container
EJB 3.0 Dependency Injection

```java
@Stateless public class CustomerAccessBean {
    private EntityManager em;

    @PersistenceContext void setEntityManager(EntityManager em) {
        this.em = em;
    }

    public void updateAccountAccess(String custNm) {
        Customer c = em.find(Customer.class, custNm);
        c.getAccount().setLastAccessTime();
    }
}
```
EJB 3.0/JPA apps with WTP

- EJB 3.0 applications can be built with WTP 1.5 and up
  - WTP 2.0 (Europa) provides some support for EJB 3.0
  - 1.5 won’t help you with development of EJB 3.0 Session Beans but it will support packaging and deployment.
- You need a application server runtime that supports EJB 3.0 and JPA (e.g., Oracle application server 10.1.3.1)
- Package JPA Entities either in an EJB project or Utility project.
- “Run on Server” to deploy
Demo: EJB 3.0 and JPA with WTP
Summary

- JPA—the Java EE standard for object-relational mapping
- EclipseLink – Advanced Persistence Library
  - Includes JPA Provider and other functionality
  - Based on Oracle TopLink and the JPA 1.0 Reference Implementation
- Dali—the WTP project bringing developer productivity to JPA
  - Mapping validation to avoid the map, deploy, debug cycle
  - Intelligent mapping assistance to avoid problems
  - Integrated with WTP to support development for Java SE and EE
Resources

- EclipseLink
  - [http://www.eclipse.org/eclipselink](http://www.eclipse.org/eclipselink)

- Dali Homepage
  - [http://www.eclipse.org/dali](http://www.eclipse.org/dali)
Q & A