Distributed Client-Server Persistence with the Java™ Persistence API

Alexander Snaps, Experienced Consultant, Axen.

TS-5969
Persist your Swing application's domain model using the Java™ Persistence API and Java DB;

Distribute your server-side domain model to offline mode enabled rich clients.
Agenda

- RDBMS enable your application using Java DB
- Java Persistence API within Swing applications
- HMVC Demo

- Distribute your persistent domain model
- Hölchoko
- Demo

- Conclusion
- Q&A
Agenda

- RDBMS enable your application using Java DB
- Java Persistence API within Swing applications
- HMVC Demo
- Distribute your persistent domain model
- Hölchoko
- Demo

- Conclusion
- Q&A
Java DB aka Apache Derby

- 100% Java SQL database server
- Bundled within your current Java Development Kit (JDK™) version 6!
- Small footprint: 2 megabytes
- Enables to embed a RDBMS within your desktop application
Setting up an embedded Java DB instance

```java
System.setProperty("derby.system.home",
    System.getProperty("user.home"));

Class.forName(
    "org.apache.derby.jdbc.EmbeddedDriver");

connection = DriverManager.getConnection(
    "jdbc:derby:MyData;create=true");
```
The joy of Java DataBase Connectivity (JDBC™) API

```java
try {
    Connection connection =
        DriverManager.getConnection("jdbc:derby:Console;create=true");
    Statement statement = null;
    try {
        statement = connection.createStatement();
        statement.executeUpdate("SELECT first_name, last_name FROM persons");
        ResultSet resultSet = null;
        try {
            resultSet = statement.getResultSet();
            while(resultSet.next()) {
                String fName = resultSet.getString("first_name");
                System.out.println(resultSet.wasNull ? "(null)" : fName);
            }
        } catch (SQLException e) {
            // Handle exception thrown while retrieving the result
        } finally {
            if(resultSet != null)
                resultSet.close();
        }
    } catch (SQLException e) {
        // Handle exception thrown while trying to get to the database
    } finally {
        if(statement != null)
            statement.close();
        connection.close();
    }
} catch (SQLException e) {
    // Handle exception thrown while trying to get a connection
}
```
Agenda

- RDBMS enable your application using Java DB
- Java Persistence API within Swing applications
- HMVC Demo

- Distribute your persistent domain model
- Hölchoko
- Demo

- Conclusion
- Q&A
Object Relational Mapping

- Eliminates the need for JDBC
  - CRUD
  - Querying
- Object identity management
  - ≠ row id
- Inheritance strategies
  - Class hierarchy to single or multiple tables
- Associations, Composition
  - Lazy navigation
  - Fetching strategies
- Transparent
Introducing Java Persistence API (JPA)

- Vendor independent ORM solution

- Easily configurable
  - Configuration directly in code using Java 5 annotations
  - Configuration fine tunable, overriding annotations using XML

- Available outside Java Platform, Enterprise Edition (Java EE) containers

- Dedicated Java Specification Request (JSR) 317 as of JPA 2.0
(Almost) transparent Plain Old Java Object (POJO) persistence

- Non-final class or methods
- Constructor with no argument
- Collections typed to interfaces
- Associations aren’t managed for you
- Database identifier field
Entity sample code

@Entity
public class Person {

@Id
private Long id;
private String firstName;
private String lastName;

@ManyToOne
private Company company;

@OneToMany(mappedBy="resident")
private Set<Address> addresses;

private Person() {}
public Person(String firstName, String lastName) {
...
}
Managing persistence

- Persistence creates an EntityManagerFactory based on a persistence unit name
- EntityManagerFactory creates EntityManager instances
- EntityManager to handle the persistence of your entities
- Query to... query(!) them back from the database
- EntityTransaction used for transaction demarcation
Setting up the persistence unit

```xml
<persistence
xmlns="http://java.sun.com/xml/ns/persistence"
version="1.0">
  <persistence-unit name="persistenceUnitName"
       transaction-type="RESOURCE_LOCAL">
    <provider>
      oracle.toplink.essentials.PersistenceProvider
    </provider>
    <class>some.domain.Class</class>
    <properties>
      <property name="toplink.jdbc.driver"
        value="org.apache.derby.jdbc.EmbeddedDriver"/>
      <property name="toplink.jdbc.url"
        value="jdbc:derby:myData;create=true"/>
      <property name="toplink.ddl-generation"
        value="create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```
Managed vs. detached entities

> Managed entities have their state being synched back to database transparently for you
  • No need to call a persist operation on managed instances
  • State flushed to the database automatically

> Detached entities, while having been persisted, aren’t managed anymore
  • Saving the state back to the database requires a merge operation
Pitfalls

➢ Transaction management
  • No need for long living transactions

➢ Exception handling
  • As entities become detached

➢ Transaction rollbacks
  • Again, transaction rolls back

➢ Detached vs. managed entities
Java Persistence API & Swing

- MVC still is a good way to design your UI
- Layered MVC makes a even better match

- Each MVC layer can
  - have its own persistence context
  - or inherit the one from its parent controller

- Events can
  - be MVC local
  - propagate down to child
  - or to the entire tree
Layered MVC

<table>
<thead>
<tr>
<th>First name</th>
<th>Last name</th>
<th>Company</th>
<th>Hometown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brad</td>
<td>Schwarz</td>
<td>Nintendo</td>
<td>Paris</td>
</tr>
<tr>
<td>Alexander</td>
<td>Jacomet</td>
<td>Apple, Inc.</td>
<td>Tokyo</td>
</tr>
<tr>
<td>Jonathan</td>
<td>Damon</td>
<td>eBay</td>
<td>Atlanta</td>
</tr>
<tr>
<td>Jack</td>
<td>Hopkins</td>
<td>Nintendo</td>
<td>London</td>
</tr>
<tr>
<td>George</td>
<td>Douglas</td>
<td>JavaBin</td>
<td>San Francisco</td>
</tr>
<tr>
<td>Jack</td>
<td>Schwarz</td>
<td>JBoss</td>
<td>Tokyo</td>
</tr>
<tr>
<td>Quentin</td>
<td>Tarantino</td>
<td>Nintendo</td>
<td>Tampa</td>
</tr>
<tr>
<td>Quentin</td>
<td>Wall</td>
<td>JoGoSlow</td>
<td>San Francisco</td>
</tr>
<tr>
<td>Louis</td>
<td>Smith</td>
<td>JBoss</td>
<td>Atlanta</td>
</tr>
<tr>
<td>Romain</td>
<td>Polanski</td>
<td>JoGoSlow</td>
<td>Paris</td>
</tr>
<tr>
<td>Robert</td>
<td>Jacomet</td>
<td>JavaBin</td>
<td>New York</td>
</tr>
<tr>
<td>Kirk</td>
<td>Jacobi</td>
<td>Nintendo</td>
<td>Oslo</td>
</tr>
<tr>
<td>Kirk</td>
<td>Polanski</td>
<td>JoGoSlow</td>
<td>Brussels</td>
</tr>
<tr>
<td>Anthony</td>
<td>Johnson</td>
<td>JBoss</td>
<td>Paris</td>
</tr>
<tr>
<td>Quentin</td>
<td>Smith</td>
<td>JoGoSlow</td>
<td>London</td>
</tr>
<tr>
<td>John</td>
<td>Schwarz</td>
<td>JavaBin</td>
<td>Oslo</td>
</tr>
<tr>
<td>Kirk</td>
<td>Gosling</td>
<td>eBay</td>
<td>Tokyo</td>
</tr>
<tr>
<td>Julian</td>
<td>Hopkins</td>
<td>Nintendo</td>
<td>Paris</td>
</tr>
<tr>
<td>George</td>
<td>Smith</td>
<td>JoGoSlow</td>
<td>Tokyo</td>
</tr>
<tr>
<td>Quentin</td>
<td>Jacobi</td>
<td>Apple, Inc.</td>
<td>Los Angeles</td>
</tr>
<tr>
<td>Romain</td>
<td>Jacomet</td>
<td>Nintendo</td>
<td>Tokyo</td>
</tr>
<tr>
<td>Dick</td>
<td>Lucas</td>
<td>Nintendo</td>
<td>Atlanta</td>
</tr>
<tr>
<td>Alexander</td>
<td>Jacomet</td>
<td>Apple, Inc.</td>
<td>Tampa</td>
</tr>
</tbody>
</table>

Brad Damon Axen New York
Quentin Snaps JBoss Brussels

250 persons listed

Model
HMVC in Swing using JPA
Axen, Alexander Snaps
Agenda

➢ RDBMS enable your application using Java DB
➢ Java Persistence API within Swing applications
➢ HMVC Demo

➢ Distribute your persistent domain model
➢ Hölcoko
➢ Demo

➢ Conclusion
➢ Q&A
Distributing your data

- Client/server with offline mode
- Distributed data sets
- Sharing data
- What else can you think of?
What is the difference?

- Most probably the database identifier
  - and the optimistic lock version id
- Maybe the database itself
- The JPA implementation used
- The entities that can be altered on each side
Agenda

- RDBMS enable your application using Java DB
- Java Persistence API within Swing applications
- HMVC Demo

- Distribute your persistent domain model
- Hölchoko
- Demo

- Conclusion
- Q&A
Defining a client side

- Client keeps track of “pairs”
- A pair is a local and remote ids for an entity
- The id can be represented by any Serializable type
- Communicates to server through a “filter”
Synching state to clients

- Server simply sends entities to the client
- Sending entities over the wires makes them detached automatically
- Client changes id field to the local value matching the remote identity
- Client merges the detached entities with the current local persistence context
Synching state back to the server

- Client sends entities to server, after having replaced local id with remote ones

- Entities with no remote id are being referenced in value holder objects, that also hold the local id value

- Value holders are being send back to the client, holding the identity matches

- Entity matches are persisted locally for future reference
Distributed Persistence With JPA
Axen, Alexander Snaps
...not quite there yet!

- Optimistic locking
  - Only track remote version id's
  - Sync local changes back before syncing remote state

- Deletion of entities
  - Are you really sure you want to do it?
  - Really?!
  - Cascading “delete-orphans”
  - JPA event listeners

- Hibernate only...
  - ...as of now
  - TopLink support on the way
Summary

- JPA & Java DB really ease persistence ...
  ... even on the desktop

- All this abstraction pays out ...
  ... and enables distributed persistence

- Enabling you to DRY and KISS!
For More Information

➤ Java DB
  • developers.sun.com/javadb

➤ JPA
  • www.oracle.com/technology/jpa
  • www.hibernate.org

➤ HMVC & Distributed persistence frameworks
  • www.codespot.net
THANK YOU

Distributed Client-Server Persistence with the Java™ Persistence API
Alexander Snaps

TS-5969