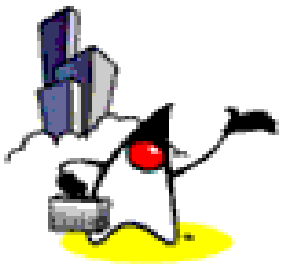
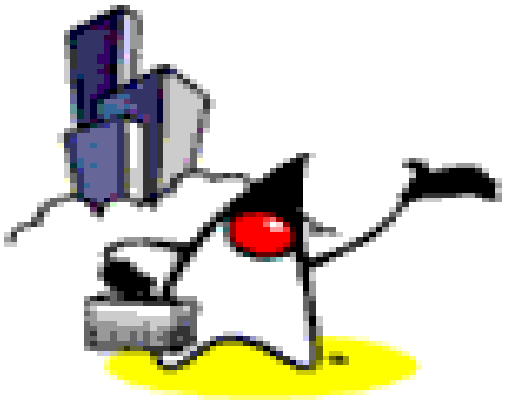
Persistence Support in Spring Framework



Topics

- Persistent technologies
- DAO interface
- Spring DAO
- Spring and Hibernate integration
  - Injecting *SessionFactory*
  - *HibernateTemplate*
- Supporting services



Persistency Technologies

Problems with Persistence

- Accessing persistent data is critical to success
- Accessing persistent data is hard
  - O/R impedance mismatch
  - Complexity of APIs
  - Performance issues

Persistence Technologies

- EJB 2.1 entity beans
  - High complexity
  - Not OO
    - Inheritance not supported
  - Tied to the EJB container
  - Not testable

Persistent Technologies: JDBC

- Still very important
- Can't get away from SQL-based approach in many scenarios
- Need to be able to mix JDBC and ORM usage
- JDBC API is fairly good at defining a standard interface to relational database
- Not an API suitable for application developers

Issues with JDBC

- Verbose: try/catch/finally
- Difficult to get correct error handling, guaranteed release of resources
- Not fully portable
  - Need to look at proprietary codes in *SQLException*
  - BLOB handling issues
  - Store procedures returning *ResultSets*

Persistent Technologies: ORM

- Transparent persistence
  - The O/R impedance mismatch can be solved
  - You can persist objects with acceptable tradeoffs
  - Partially decouples from database
    - Still must consider performance
    - Deep inheritance questionable
  - Copes better with change
  - ORM queries are less fragile than SQL queries
    - Against your domain objects, not RDBMS schema
    - Can drop down to SQL queries if necessary

ORM Solutions

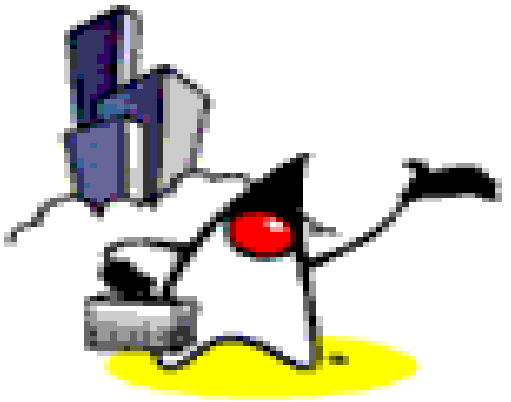
- JDO
- Hibernate
- TopLink
- EJB 3.0 Java Persistence API (JPA)

ORM: Shared Fundamental Concepts

- Unit of work
  - Transactional cache
  - JDO PersistenceManager
  - Hibernate Session
  - TopLink UnitOfWork
- How do you get units of work?
  - JDO PersistenceManagerFactory
  - Hibernate SessionFactory
  - Several ways in TopLink

Solution

- Insulate business objects from persistence API
- Do not use fake objects
  - Apply correct OO design
  - Persistent objects should contain business logic
- But how do we obtain and persist objects
  - Don't want business objects to know about particular unit of work
  - Don't want HQL, JDO QL, SQL queries in business objects



DAO Interface

DAO Interfaces

- De-couple persistence API details from business logic in service objects
- Easy to mock DAO interfaces
- DAO interfaces contain
  - Finder methods
  - Save methods
  - Count methods

DAO Interface

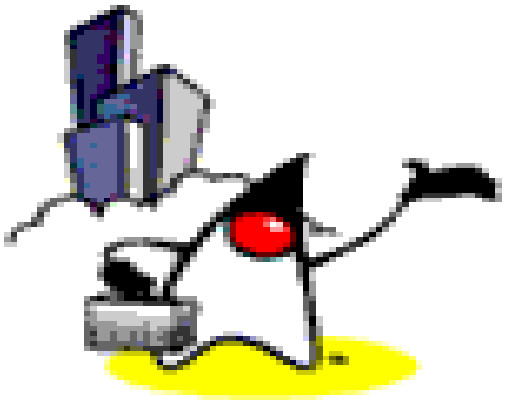
```
public interface ReminderDao {
   public Collection findRequestsEligibleForReminder()
                                        throws  DataAccessException;
   void save(Reminder reminder)
                                        throws DataAccessException;
}
```

Hibernate DAO Implementation

```
public class HibernateReminderDao extends HibernateDaoSupport
    implements ReminderDao {

   public Collection findRequestsEligibleForReminder()
            throws DataAccessException {
      getHibernateTemplate().find(
                  "from Request r where r.something =1");
   }

   public void save(Reminder reminder)
            throws DataAccessException {
      getHibernateTemplate().saveOrUpdate(reminder);
   }

}
```

DAO Portability

- Decrease lock-in to Hibernate or another vendor API
- Switch between Hibernate, JDO, and other transparent persistent technologies without changing DAO interfaces
- Can even switch to JDBC where transparent update is not implied



Spring DAO

Spring DAO

- Spring supports
  - JDBC
  - Hibernate
  - JDO
  - iBATIS SQL Maps
  - Apache OJB
- Greatly reduces plumbing code

Spring DAO Concepts: Template

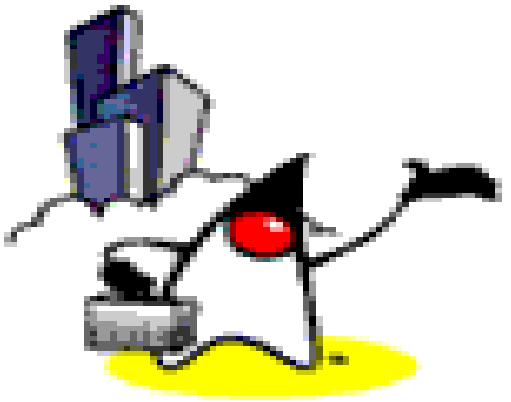
- Callback methods allow resource acquisition/release
  - Spring opens and closes resources and maps exceptions
- One-liners for many operations

Spring DAO Concepts

- XXXDaoSupport
  - Convenient
- Exception mapping
- Unit of work management

Spring DAO: JDBC

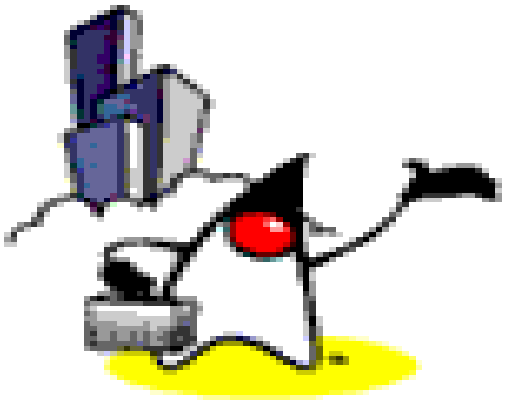
- JdbcTemplate
  - Uses callbacks to enable code to throw *SQLException*
  - Guarantees that connections and other resources will be released
- JDBC operation objects
  - An object representing a query, stored procedure or update
- Much less verbose than native JDBC
- Much less error-prone than native JDBC



Spring & Hibernate Integration

Spring & Hibernate Integration

- *SessionFactory* object is Dependency-Injected
- *HibernateTemplate* class is provided



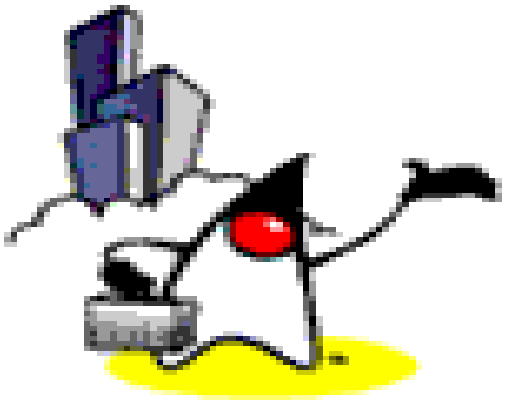
Setting up Hibernate SesssionFactory

DataSource Setup Through DI

```
<beans>

 <bean id="myDataSource"
   class="org.apache.commons.dbcp.BasicDataSource" destroy-
   method="close">
   <property name="driverClassName"
   value="org.hsqldb.jdbcDriver"/>
   <property name="url"
   value="jdbc:hsqldb:hsql://localhost:9001"/>
   <property name="username" value="sa"/>
   <property name="password" value=""/>
 </bean>
```

SessionFactory Setup Through DI

```
<bean id="mySessionFactory"
  class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource" ref="myDataSource"/>
  <property name="mappingResources">
    <list>
      <value>product.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <value>
      hibernate.dialect=org.hibernate.dialect.HSQLDialect
    </value>
  </property>
</bean>

</beans>
```



HibernateTemplate

HibernateTemplate

- Helper class that simplifies Hibernate data access code
- Automatically converts *HibernateExceptions* into *DataAccessExceptions*, following the *org.springframework.dao exception* hierarchy
- The central method is *execute*, supporting Hibernate access code implementing the *HibernateCallback* interface.

HibernateTemplate

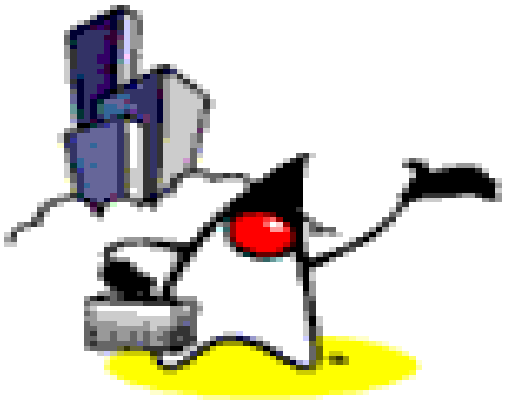
- It provides Hibernate Session handling such that neither the *HibernateCallback* implementation nor the calling code needs to explicitly care about retrieving/closing Hibernate Sessions, or handling Session lifecycle exceptions.
- For typical single step actions, there are various convenience methods (*find*, *load*, *saveOrUpdate*, *delete*).

HibernateTemplate Example

```
SessionFactory sessionFactory= HibernateFactory.getSessionFactory();
HibernateTemplate template= new HibernateTemplate(sessionFactory);

Event event1 = new Event("Event 1");
Event event2 = new Event("Event 2");
Event event3 = new Event("Event 3");

template.save(event1);
template.save(event2);
template.save(event3);
```

HibernateTemplate Example

```
template.execute(new HibernateCallback() {
    public Object doInHibernate(Session session) throws
                                                HibernateException, SQLException {
        Query query = session.createQuery("from Event");
        query.setMaxResults(2);
        List events = query.list();
        for (Iterator it = events.iterator(); it.hasNext();) {
            Event event = (Event) it.next();
            System.out.println(event.getName());
            event.setDuration(60);
        }
        return null;
    }
});
```



Supporting Services

What Supporting Services Do We Need?

- Must solve the problem of data access exceptions to have independent DAO interfaces
  - Can't throw *SQLException* or *JDOException*
  - Catch/warp leads to huge redundancy
    - Ex) Catch *SQLException* throw *MyFunnyDaoException*
  - Need meaningful exceptions
    - Not just one *SQLException*
    - Need to be able to catch at different levels
  - Data access exceptions should be unchecked

Persistence Support in Spring Framework