

# *Persistence with Spring*



Rod Johnson

Interface21

# *Spring & Persistence*

- Provide a common persistence approach across a wide range of technologies
  - JDBC
  - JDO
  - Hibernate
  - TopLink etc.
- Make persistence APIs much easier to use
  - Eliminate common causes of bugs in application code

# *Problems with persistence*

- Accessing persistent data is critical to success
- Accessing persistent data is hard
  - O/R impedance mismatch
  - Complexity of APIs
  - Inconsistent approach between different tools
- Spring can help take away the complexity issues, leaving developers free to focus on design issues

# *Persistence technologies: JDBC*

- JDBC
  - Still *very* important
  - Can't get away from a 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 databases
  - Not an API suitable for application developers

# *Persistence technologies: JDBC*

- Why you should *never* use raw JDBC directly
  - 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
    - Stored procedures returning ResultSets etc.
    - Proprietary SQL is not the main problem
- Spring provides a powerful class library above JDBC that solves all these problems, and adds further value

# *ORM: Shared Fundamental concepts: Tools differ, concepts don't*

- **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
- What is my current unit of work (for this transaction)?
- Local transaction API and JTA synchronization
- Second-level cache

# *Transaction management*

- How do we provide consistent transaction management across persistence tools
  - JDO/Hibernate + JDBC?
- Can use JTA
  - Only works in an app server
  - Adds complexity unless you want distributed transactions
- Want a consistent approach
- Service objects delimit transactions declaratively or programmatically
  - Should not work with persistence tool transaction API
  - Should work in a JTA or “local” environment, without changing application code
- Spring provides a comprehensive, consistent solution

# Goal

- Insulate business objects from persistence API
- 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

# *Solution: DAO interfaces*

- Decouple persistent API details from business logic in service objects
- Easy to mock DAO interfaces
- Service objects can be maintained without
- DAO interfaces should contain
  - Finder methods
  - Save methods
  - Aggregate function methods
- Also known as the **Repository** pattern (*Domain-Driven Development*)
- Spring is designed to facilitate this best practice

# *DAO interface design: Semantic differences*

- Transparent persistence
  - JDO, Hibernate etc
  - Persistence by reachability
  - Dirty objects automatically flushed
- JDBC
  - Finders are the same, but if we modify an object it isn't automatically saved
- Can fairly easily design interfaces portable across transparent persistence APIS
- JDBC interoperability more problematic, except for read-only

# *DAO interface example*

```
public interface ReminderDao {  
  
    public Collection  
        findRequestsEligibleForReminder  
        ()  
            throws DataAccessException;  
  
    void persist(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 persist(Reminder reminder)
        throws DataAccessException {
        getHibernateTemplate().saveOrUpdate(reminder);
    }

    ...
}
```

# ...*Hibernate DAO implementation*

- Note the code we are *not* writing:
  - How do we obtain the Hibernate Session?
  - How do we know we have the Session already associated with the thread, if there was one?
  - How do we close the Session, even in the event of errors?
  - How do we handle HibernateException?
- **Spring does this crucial work for us**
- The code we *do* write actually *does something*
- The intent of the code we write is clear

# *JDBC DAO implementation*

```
public class JdbcReminderDao extends JdbcDaoSupport
                                implements ReminderDao {

    public Collection findRequestsEligibleForReminder()
                                throws DataAccessException {
        return getJdbcTemplate().query("SELECT NAME, DATE, ... " +
            " FROM REQUEST WHERE SOMETHING = 1",
            new RowMapper() {
                public Object mapRow(ResultSet rs, int rowNum)
                                throws SQLException {
                    Request r = new Request();
                    r.setName(rs.getString("NAME"));
                    r.setDate(rs.getDate("DATE"));
                    return r;
                }
            });
    }

    public int countRequestsEligibleForReminder()
                                throws DataAccessException {
        return getJdbcTemplate.queryForInt("SELECT COUNT(*) FROM ...");
    }
}
```

# ...*JDBC DAO implementation*

- Note the code we are *not* writing
  - We don't need to write code to obtain connections
  - We don't need to write code to close connections
    - Spring will *always* close connections, even in the event of errors
  - Exceptions are translated into Spring's `DataAccessException` hierarchy, whatever persistence API we use
  - No try/catch/finally blocks
- Reduction of 60-70% in application code is typical with Spring JDBC vs raw JDBC

# *Callbacks*

- Spring “templates” use callbacks to move resource handling and error handling into framework
  - It is impossible to write application code that will cause a connection leak
- Count doesn't require a callback to be implemented
  - JDO and Hibernate operations also typically don't need callbacks in application code

# *How does business object get DAO implementation?*

```
public class DefaultReminderGenerator implements
    ReminderGenerator {
    private ReminderDao reminderDao;

    public void setReminderDao(ReminderDao reminderDao) {
        reminderDao = reminderDao;
    }

    ... business methods
}

<bean id="reminderGenerator"
    class="com.mycompany.service.DefaultReminderGenerator">
    <property name=" reminderDao">
        <ref local="reminderDao "/>
    </property>
</bean>
```

# *Integration with IoC*

- Spring DAO is separate from IoC container
  - You don't need to use Spring IoC container to use DAO features
  - JDBC library is a popular first usage of Spring
- ...but naturally Spring DAO integrates well with Spring IoC

# *DAO portability*

- Decrease lock-in to Hibernate or another vendor API
- Allow use of vendor-specific features—so long as they are mirrored by other vendors—without tying application code to the vendor's API
- Switch between Hibernate, JDO and other transparent persistence technologies without changing DAO interfaces
  - Examples of this in Spring sample applications
- Can even switch to JDBC where transparent update is not implied

# *Lock in*

- Why do we care about lock-in to my persistence tool?
  - Uncertainty in O/R mapping space
    - With JSR-220 persistence coming, you may want to change persistence API
    - May keep the same tool, but access it through a different API
- But aren't we locked in to Spring instead?
  - **No** -- Exceptions are just a class library: You'd need to implement them yourself otherwise. Spring just saves you a lot of work.
  - You are not locked in at an architectural level, you are purely choosing to use Spring conveniences in implementing DAO interfaces.

# *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
    - Leaky abstraction
  - Catch/wrap leads to huge redundancy
    - Catch SQLException, throw MyFunnyDaoException
    - Pointless code to write and maintain
  - Need **meaningful** exceptions
    - Not just one SQLException
    - Need to be able to catch at different levels
  - Data access exceptions should be unchecked

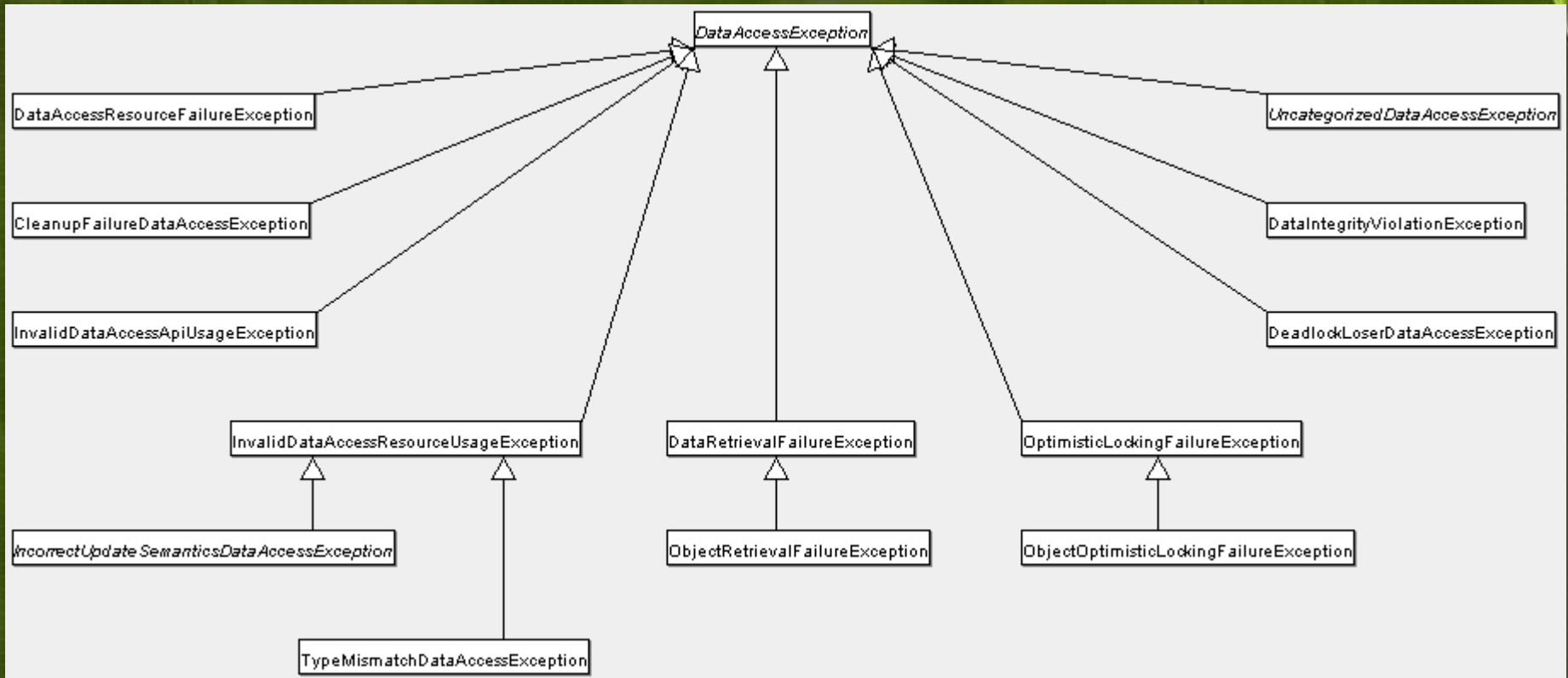
*With raw JDBC, how do we find out whether we've encountered a unique constraint violation?*

*Must look at database-specific error  
code in `SQLException`*

# *The Spring solution*

- Catch `DataIntegrityViolationException`
- Works in all databases
- Works even through an ORM tool
- Code is self-documenting
  - Reliance on `SQLException` error codes is not necessarily clear without documentation
  - Will fail silently if migrated to another database
- Exception categorization applies to whatever ORM tool or database we use

# Spring DAO: Consistent exception hierarchy



# *Exception translation summary*

- Whatever persistence tool you use, its exceptions will be translated into Spring's hierarchy
- Spring's hierarchy preserves stack traces for logging etc.
- You can even extend the mappings without changing Spring code:
  - Define a mapping from a custom PL/SQL error code raised in an Oracle trigger or stored procedure to a custom subclass of `DataAccessException`
  - Map from a specific error condition unique to your ORM tool

# *Checked vs unchecked*

- Persistence exceptions should be unchecked
  - Not usually recoverable
  - Checked exceptions lose value of exception hierarchy
  - Exception handling becomes verbose and obfuscates code
    - Look at code using CMP or JDBC
    - Catch and wrap and rethrow
  - Checked exceptions for persistence operations sound good in theory, but practice shows differently

# *Supporting services needed*

- Must be able to implement DAO interfaces easily
  - Avoid causes of common errors
  - Provide a consistent approach for multiple APIs
  - Translate all exceptions into a meaningful hierarchy
- Should be able to use different persistence technologies in the one application
- Must provide a consistent solution to transaction management

# *Spring DAO: simplification*

- Spring supports
  - JDBC
  - Hibernate
  - JDO
  - iBATIS SQL Maps
  - Apache OJB
  - TopLink
  - Cayenne
- Greatly reduces plumbing code
- Don't need to write in-house frameworks around persistence APIs

# *Spring DAO concepts*

- **Templates**
  - Callback methods allow resource acquisition/release
    - “Do with a JDO PersistenceManager”
    - Spring opens and closes resources and maps exceptions
  - One-liners for many operations
- **XXXXDaoSupport**
  - Convenient superclasses providing templates to application subclasses
- **Exception mapping**
- **Unit of work management**

# *Spring DAO concepts*

- Common concepts across all tools help to bring consistency to application code across large projects
- New developers familiar with Spring will become productive right away
- Concepts are well-documented

# *Common concepts, common approach*

Purpose	JDBC	JDO	Hibernate	TopLink
Consistent error handling, resource acquisition and release	JdbcTemplate	JdoTemplate	HibernateTemplate	TopLinkTemplate
Provide resources to templates	Uses standard J2EE DataSource	LocalPersistenceManagerFactoryBean or JNDI-bound PersistenceManagerFactory	LocalSessionFactoryBean or JNDI-bound SessionFactory	SessionFactoryBean
Convenient superclass for DAO implementations	JdbcDaoSupport	JdoDaoSupport	HibernateDaoSupport	TopLinkDaoSupport
Exception translation	SQLExceptionTranslator	JdoDialect	SessionFactoryUtils	TopLinkAccessor
Transaction management	DataSourceTransactionManager or JTA	JdoTransactionManager or JTA	HibernateTransactionManager or JTA	TopLinkTransactionManager or JTA

# *Simplification: 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
- Much less error-prone

# *JDBC Template usage*

```
List l = jdbcTemplate.query("SELECT NAME, DATE, ... " +  
                            " FROM REQUEST WHERE SOMETHING = 1",  
    new RowMapper() {  
        public Object mapRow(ResultSet rs, int rowNum) throws  
                               SQLException {  
            Request r = new Request();  
            r.setName(rs.getString("NAME"));  
            r.setDate(rs.getDate("DATE"));  
            return r;  
        }  
    });
```

```
int count = jdbcTemplate.queryForInt("SELECT COUNT(*) FROM ...");
```

# *JDBC operation example*

```
ReminderGeneratorProc rg = new  
    ReminderGeneratorProc(dataSource);  
  
int result = rg.generate(25);
```

- Extend Spring StoredProcedure superclass
  - Specify SQL, DataSource, in/out bind variables
- Threadsafe object
- Also superclasses for updates, queries
  - Looks a little like JDO for querying
  - Hides SQL inside the object

## *Other value adds of Spring JDBC*

- Increase portability, not merely with respect to error codes
  - Advanced operations such as stored procedures returning result sets can be coded in the same way for all databases
  - Stored function access is portable
  - No need for RDBMS-specific code (for example, in Oracle) to perform BLOB handling

# *Spring JDBC: Summary*

- Spring JDBC support takes away much of the complexity of working with JDBC
  - You get to focus on the SQL, not the plumbing
- But will still want to use an ORM tool where possible...

# *Simplification for JDO/Hibernate and other ORM tools*

- Manage transactional units of work
  - JDO PersistenceManager
  - Hibernate Session
  - TopLink Session/UnitOfWork
- We need ThreadLocal sessions
  - Relates to transaction management
- Integrate management of unit of work with underlying JDBC connection, allowing use of JDBC in same transaction with no custom coding

# *TopLink*

- Same benefits as with JDO and Hibernate
  - User never needs to acquire TopLink UnitOfWork
  - TopLinkTemplate
  - TopLinkDaoSupport
  - TopLinkTransactionManager
- Why do Oracle want this?
  - Clients are asking for it
  - TopLink team are enthusiastic about the Spring approach to persistence
    - Encourages best practices advocated for years by TopLink Professional Services team

# *Interoperability*

- Spring allows an ORM tool to be used in the same transaction as JDBC
  - Same Connection is used for JDBC operations
- Some cautions
  - Must flush the ORM session to allow JDBC to see data
  - If JDBC code modifies the database, may invalidate ORM cache
- No issue if the two use different tables but **in the same transaction**
- Common situation:
  - JDBC for write-only data
  - JDBC for a few really complex queries that need Oracle-specific SQL after flushing session
  - ORM for everything else

# *Advanced features*

- “Open session in view” pattern
  - Allows Hibernate Session or JDO PersistenceManager to be kept open—non transactionally—in web tier
  - Sometimes useful when we want to minimize the need to disconnect objects
- Again, a consistent approach across different persistence APIs

# *Extensibility*

- LDAP (contribution)
- Documentum (integration in at least 2 projects)
- Vendor-specific value adds
  - Kodo JDO
  - Would be possible for Lido, and we have discussed it with them
- Will support JSR-220 persistence API as soon as binaries are published

# Summary

- Spring has unique persistence capabilities that can help reduce errors in application code and improve productivity
- Spring JDBC is a highly sophisticated class library
  - Reduces the volume of code required to perform JDBC operations
  - Helps to eliminate common errors such as connection leaks
- Spring provides a consistent approach across many persistence frameworks including ORM tools such as JDO and Hibernate
- Spring DAO facilitates clean, layered, design

# *Questions*