Data Access Objects (DAOs)

An Introduction to DAOs, DB Schema, and SQL

Revision: v2014-08-14
Built on: 2018-12-05 08:18 EST
Copyright © 2018 jim stafford (jim.stafford@jhu.edu)

This presentation provides an introduction to the Data Access Object (DAO) pattern to encapsulate access to data, database schema used to design data organization in RDBMS systems, and SQL to access the data.
Purpose

1. Goals
   • Introduce DAO Pattern
   • Introduce RDBMS Schema
   • Introduce SQL

2. Objectives

At the completion of this topic, the student shall
• have an understanding of:
  • The role played by a DAO and business tier as it relates to accessing data
  • Core concepts to relational tables
  • How to define database tables and relationships using RDBMS schema
  • How to access data within the database using SQL
• be able to:
  • Design the interface for a DAO
  • Create a schema in the database
  • Implement a basic DAO using EntityManager and native SQL commands
Chapter 1.

Data Access Object (DAO) Pattern

1.1. Context
• Access to data varies depending on source of data

1.2. Problem
• Interfaces to data sources vary
  • Relational Database Management Systems (RDBMS)
  • NoSQL Solutions
  • Flat Files
  • Backend Systems
• Even standard RDBMS/SQL interfaces can vary

1.3. Forces
• Many components within application need access to data
• Interfaces to data vary by technology and vendor
  • Least common denominator option for portability may not be feasible in all cases
  • May make use of vendor extensions
• Impact of unique interfaces significant when exposed to many components and component types
  • Components need more abstraction and shielding from the details of the persistent store

1.4. Solution
Use a Data Access Object (DAO) to abstract and encapsulate access to business objects in the data source

Figure 1.1. DAO Pattern Players

Business Logic
• Object within the business domain that needs access to data (e.g., session bean)
• Knows when/why data is needed, but not where or how to access it
Chapter 1. Data Access Object...

Data Access Object

- Abstracts the access details from the business object
- Knows where/how data is accessed, but not when or why to access it

Business Object (Entity)

- An entity within the business logic
- Encapsulates information and data business rules within application
- A data carrier of information to/from the DAO

Data Source

- Physically stores the data (e.g., database)

Figure 1.2. Interactions: DAO Create

Figure 1.3. Interactions: DAO Read/Get
Figure 1.4. Interactions: DAO Update

Figure 1.5. Interactions: DAO Delete

1.5. DAO Implementation Structure

Key point: hide details from business logic and other interfacing components
1.6. Consequences

- Centralizes All Data Access into a Separate Layer
  - Easier to maintain
  - Enables Transparency
    - Access to implementation details hidden within DAO
  - Enables Easier Migration
    - Client layers encapsulated from changes
- Reduces Code Complexity in Business Logic
  - No details, such as SQL, in business logic
- Was hard to abstract with EJB 2.x Container Managed Persistence (CMP) frameworks
- EJB3 Java Persistence API provides a significant amount of abstraction
DAO Interface

2.1. DAO Interface

- Technology agnostic and business object-focused
- No mention of Connection or EntityManager in methods
- Ability to at least CRUD (with possible options)
- Aggregate data functions added when behavior better performed at data source
- Extensions added to address data access details for specific use cases (e.g., LAZY/EAGER load)

```java
import javax.persistence.PersistenceException;

public interface BookDAO {
    Book create(Book book) throws PersistenceException;
    Book update(Book book) throws PersistenceException;
    Book get(long id) throws PersistenceException;
    void remove(Book book) throws PersistenceException;
    List<Book> findAll(int start, int count) throws PersistenceException;
}
```

Note

The declaration of the unchecked/runtime exception PersistenceException is not required and is only being done here for extra clarity.

2.2. DAO Exceptions

Runtime Exceptions

- Used to report unexpected issues (e.g., no connection)
- Extends java.lang.RuntimeException
- ex. javax.persistence.PersistenceException

Checked Exceptions

- Used to report anticipated errors mostly having to do with input
- Extends java.lang.Exception

2.3. DAO Implementation

Adds implementation out-of-band from DAO interface

```java
public class JPABookDAOImpl implements BookDAO {
    private EntityManager em;
    public void setEntityManager(EntityManager em) {
        this.em = em;
    }
}```
2.4. DAO Test

Demonstrates how technology-neutral DAO clients can be when dao implementation is injected into client.
Data Transfer Object (DTO) Pattern

Relating Business Objects to DTOs

3.1. Context
- Business Objects represent too much information or behavior to transfer to remote client

3.2. Problem
- Client may get information they don't need
- Client may get information they can't handle
- Client may get information they are not authorized to use
- Client may get too much information/behavior to be useful (e.g., entire database serialized to client)

3.3. Forces
- Some clients are local and can share object references with business logic
- Handling specifics of remote clients outside of core scope of business logic

3.4. Solution
- Layer a Remote Facade over Business Logic
- Remote Facade constructs Data Transfer Objects (DTOs) from Business Objects that are appropriate for remote client view
- Remote Facade uses DTOs to construct or locate Business Objects to communicate with Business Logic

Figure 3.1. DAO Pattern Players

Data Transfer Object
- Represents a subset of the state of the application at a point in time
- Not dependent on Business Objects or server-side technologies
- Doing so would require sending Business Objects to client
- XML and JSON provide the “ultimate isolation” in DTO implementation/isolation
Remote Facade

- Uses Business Logic to perform core business logic
- Layered on top of Business Logic to translate between Business Objects and DTOs

Business Logic

- Continues to perform core duties as described in DAO Pattern

Business Object (Entity)

- Continues to perform core duties as described in DAO Pattern
- May have more server-side-specific logic when DTOs are present in the design

3.5. Consequences

- Clients only get what they need
- Clients only get what they understand
- Clients only get what they are authorized to use
- Remote and Local interfaces to services are different
  - Makes it harder to provide location transparency
- Lightweight Business Objects can be used as DTOs
  - Remote Facade must make sure they are “pruned” of excess related items before transferring to client
  - Remote Facade must make sure they are “cleaned” of DAO persistence classes before transferring to client
Chapter 4.

RDBMS Schema

4.1. RDBMS

4.1.1. Background

- DBMS based on a relational model
- Introduced by E. F. Codd in 1970s
- Some challenges by other forms but still remains a standard for corporate data stores

4.1.2. Tables/Columns

**Table**
- Group of columns
- Represents a type
- Commonly mapped to a Java class

**Column**
- Single piece of data
- Represents a property
- Commonly mapped to a Java class attribute

Figure 4.1. Tables and Columns

Sample (H2) Column Types

- INTEGER
- DECIMAL
- TIME
- DATE
- TIMESTAMP
- VARCHAR

---

2. [H2 Data Types](http://www.h2database.com/html/datatypes.html)
4.1.3. Constraints

**NOT NULL**
Row cannot exist without this column value supplied

**UNIQUE**
No other row may have a column with this value

**FOREIGN KEY**
If supplied, must reference matching column(s) of existing row

4.1.4. Relationships

4.1.4.1. Relationship Expression

**Foreign Key Join**
Column within child table references parent

**Primary Key Join**
Foreign key column within child table is child's primary key column. Parent and child table primary keys must match.

**Link Table Join**
Foreign keys to parent/child expressed in separate table

4.1.4.2. Relationship Rules

- Foreign keys may be defined in child table or link table
- Link tables can always be used (at an extra cost) no matter the cardinality
- Foreign keys cannot be defined on "one side" of a one-to-many relationship
  - Link table must be used if foreign key cannot be placed on many side
- Link tables must be used in many-to-many relationships

4.1.5. Indexes

- Optionally defined
- Used to more quickly locate values within table rows

**Types**

- **Non-unique**
  Index with possible repeated values

- **Unique**
  Index over a unique (set of) column(s)

- **Simple**
  Single column index
Composite
  Multiple columns stored in index

Cluster
  Parent/child share common location

4.2. Data Definition Language (DDL)

Used to manipulate schema in RDBMS

4.2.1. Create Table

create table JPADAO_AUTHOR ( 
  ID integer generated by default as identity,
  FIRST_NAME varchar(16) not null,
  LAST_NAME varchar(32) not null,
  primary key (ID)
);
create table JPADAO_BOOK ( 
  ID bigint generated by default as identity,
  DESCRIPTION varchar(1000),
  PAGES integer,
  TITLE varchar(32) not null,
  AUTHOR_ID integer,
  primary key (ID)
);

4.2.2. Drop Table

drop table JPADAO_AUTHOR if exists;
drop table JPADAO_BOOK if exists;

4.2.3. Create Foreign Key Constraint

alter table JPADAO_BOOK
  add constraint JPADAO_BOOK_AUTHOR_FK
  foreign key (AUTHOR_ID)
  references JPADAO_AUTHOR

4.2.4. Drop Foreign Key Constraint

alter table JPADAO_BOOK drop constraint JPADAO_BOOK_AUTHOR_FK;

4.2.5. Create Index

create index JPADAO_BOOK_AUTHOR_FKX on JPADAO_BOOK(AUTHOR_ID);
create unique index JPADAO_BOOKITLE_IDX on JPADAO_BOOK(TITLE);
4.2.6. Drop Index

drop index JPADAO_BOOK_AUTHOR_FKX if exists;
drop index JPADAO_BOOK_TITLE_IDX if exists;

4.3. DDL Files in Maven Module

```
src/main/resources/
  |-- ddl
  |   |-- book-create.ddl
  |   |-- book-drop.ddl
  |   |-- book-tuningadd.ddl
  |   `-- book-tuningremove.ddl
target/classes/
  |-- ddl
  |   |-- book-create.ddl
  |   |-- book-drop.ddl
  |   |-- book-tuningadd.ddl
  |   `-- book-tuningremove.ddl
```
SQL Basics

5.1. Create/INSERT

```sql
insert
into
    JPADAO_BOOK
    (ID, DESCRIPTION, PAGES, TITLE)
values
    (null, 'this and that', 1037, 'gww')
```

Get the generated primary key value

call IDENTITY()

5.2. Read/SELECT

```sql
select
    book.ID,
    book.DESCRIPTION,
    book.PAGES,
    book.TITLE
from
    JPADAO_BOOK book
```

5.3. Update/UPDATE

```sql
update
    JPADAO_BOOK
set
    DESCRIPTION='this, that, and the other thing'
where
    ID=3
```

5.4. Delete/DELETE

```sql
delete
from
    JPADAO_BOOK
where
    ID=?
```
public class JDBCBookDAOImpl implements BookDAO {
    private Connection connection;

    public void setConnection(Connection connection) {
        this.connection = connection;
    }

    @Override
    public Book create(Book book) throws PersistenceException {
        try (PreparedStatement insertStatement = getInsertPreparedStatement(connection, book);
             PreparedStatement idStatement = getIdentityStatement(connection)) {
            insertStatement.execute();

            try (ResultSet rs = idStatement.executeQuery()) {
                if (rs.next()) {
                    Field id = Book.class.getDeclaredField("id");
                    id.setAccessible(true);
                    id.set(book, rs.getLong(1));
                } else {
                    throw new PersistenceException("no identity returned from database");
                }
            } catch (NoSuchFieldException ex) {
                throw new PersistenceException("Error locating id field", ex);
            } catch (IllegalAccessException ex) {
                throw new PersistenceException("Access error setting id", ex);
            }

            return book;
        } catch (SQLException ex) {
            throw new PersistenceException("SQL error creating book", ex);
        }
    }

    private PreparedStatement getInsertPreparedStatement(Connection c, Book book) throws SQLException {
        PreparedStatement statement = connection.prepareStatement(
            "insert into JPADAO_BOOK (ID, DESCRIPTION, PAGES, TITLE) " +
            "values (null, ?, ?, ?)"
        );
        statement.setString(1, book.getDescription());
        statement.setInt(2, book.getPages());
        statement.setString(3, book.getTitle());
        return statement;
    }

    private PreparedStatement getIdentityStatement(Connection c) throws SQLException {
        PreparedStatement statement = connection.prepareStatement("call identity()");
    }
}
return statement;
}

6.2. Java Persistence API (JPA) Native SQL

```java
public class JPANativeSQLBookDAO implements BookDAO {
    private EntityManager em;

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

    @Override
    public Book create(Book book) throws PersistenceException {
        em.createNativeQuery(
            "insert into JPADAO_BOOK (ID, DESCRIPTION, PAGES, TITLE) " +
            "values (null, ?1, ?2, ?3)"
        ).setParameter(1, book.getDescription())
        .setParameter(2, book.getPages())
        .setParameter(3, book.getTitle())
        .executeUpdate();

        int idVal = ((Number)em.createNativeQuery("call identity()")
            .getSingleResult()).intValue();
        try {
            Field id = Book.class.getDeclaredField("id");
            id.setAccessible(true);
            id.set(book, idVal);
        } catch (Exception ex) {
            throw new RuntimeException("Error setting id", ex);
        }

        return book;
    }
...
}
```