Data Persistence

- Many applications must store data beyond the application's own lifetime.
- In fact, some business applications are created specifically to store, retrieve, and modify data.
- It is very important to understand how Spring can help in creating these applications.
- The storage of data, in object-oriented systems, is called persistence.
- When a data object is saved to a database, the object has been persisted.
- Of course, you can persist your data objects to disk using your own custom-written code, perhaps calling a database's published API.
- But when you use modern frameworks, Spring, there is often built-in support to make object persistence simple.

EJB Persistence

- In the world of enterprise software development, component persistence is synonymous with entity Enterprise Java Beans (EJBs).
- Entity EJBs are business data components that can be automatically stored into and retrieved from a variety of relational databases.
- Unfortunately, you need to pay a hefty price for automated EJB persistence support: you must run a large server called the Java EE EJB container.

JPA

- The “heavyweight server” requirement was such a widespread industry problem that in June 2003 a JSR 220 expert group was chartered in order to simplify EJB container-managed persistence (CMP).
- The expert group decided that a simplification of EJB CMP was not sufficient, and that what was needed was a lightweight framework that can persist plain old Java objects (POJOs).
- The effort of the expert group resulted in the Java Persistence API (JPA) specification. JPA is the official persistence mechanism of the Java 5 EE platform.

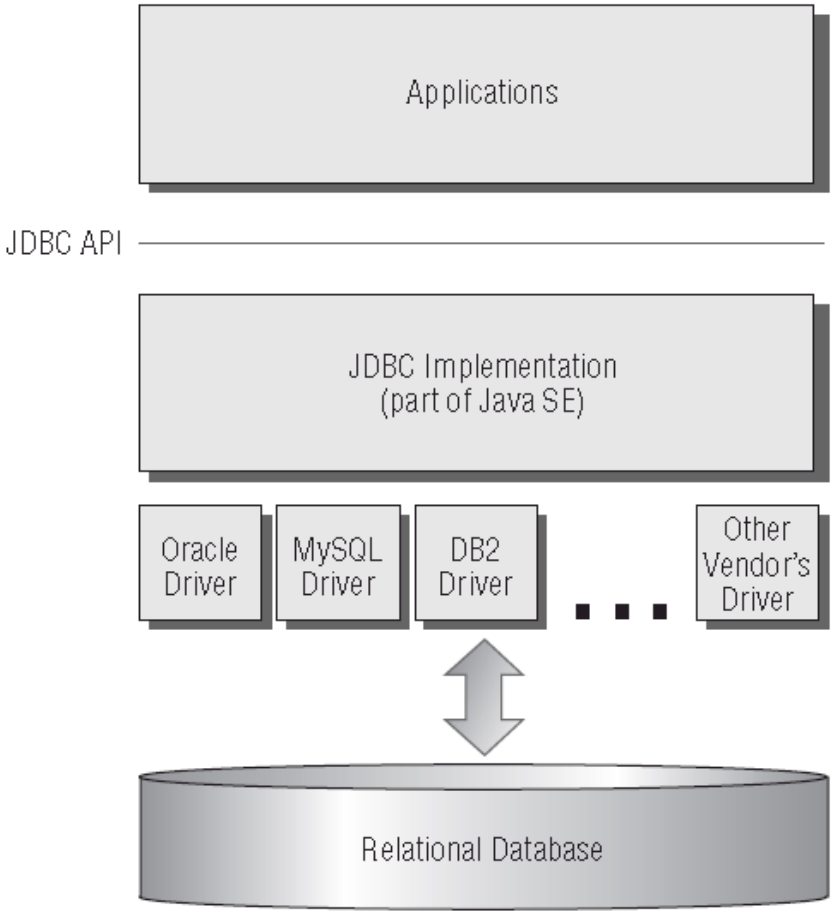
Persistence Mechanisms

- The Spring framework supports a variety of persistence mechanisms.
  - direct use of JDBC,
  - a variety of object-to-relational-mapping (ORM) technologies.
  - the implementation of a unifying data access object (DAO) abstraction layer

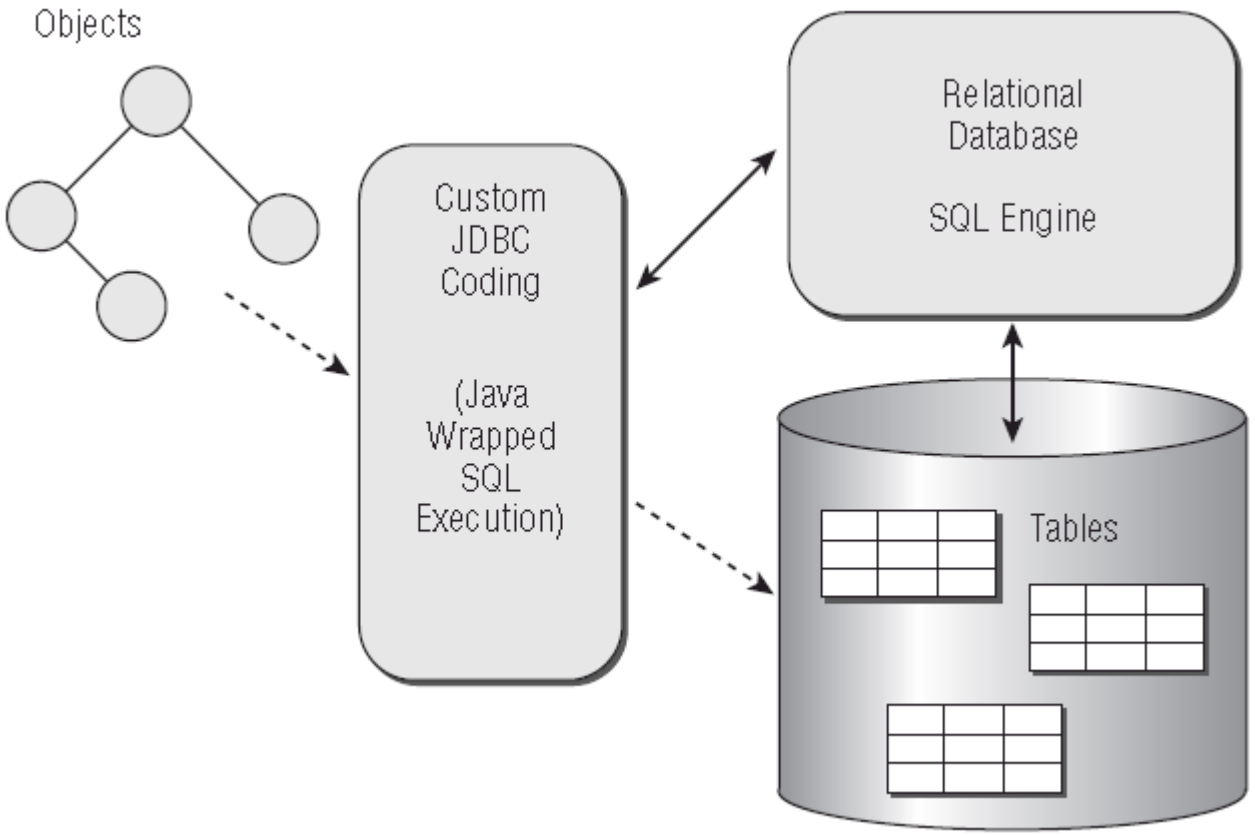
JDBC Architecture

- The JDBC API is a low-level API that interfaces with the relational database.
- Each relational database vendor provides a database driver that is JDBC-compliant.
- The driver is what interacts directly with the database, as shown in the JDBC architecture diagram in the next slide.
- As the figure demonstrates, developers can write their database access code using the JDBC APIs.
- This API implementation is built in to the Java SE platform and does not require additional installation of any type.
- To access relational databases from a different vendor, a vendor-specific driver is required.
- All modern relational database engines provide an interface to its features through Structured Query Language (SQL), and JDBC's operation is based primarily on sending SQL statements to a relational database and then retrieving the results.
- In effect, when writing JDBC code, you need to create SQL statements, execute them on the server, and then work with the tabular result set that the server returns (in the case of queries).

JDBC API


Applications
JDBC API
JDBC Implementation
(part of Java SE)
Oracle
Driver
MySQL
Driver
DB2
Driver
Other
Vendor's
Driver
Relational Database

How JDBC code can persist object data to a relational database



Traditional JDBC Approach

- The JDBC API gives programmers direct control over DB access and cache management.
- But there are some characteristics that make it rather tedious to use:
  - *Cumbersome connection and resource handling: The developer needs to deal with a lot of infrastructure* code involving connection and resource management operations such as opening and closing connections, and try-catch-finally blocks for transaction commit and rollback.
  - *The procedural nature of JDBC doesn't support Java objects: The JDBC API does not support the persistence* of Java objects directly. This procedural API does not lend itself to easy object-oriented persistence. In fact, you must possess an intimate knowledge of the relational model when programming JDBC. You need to map your tree of application objects first into tables and then into rows and columns.
  - *No exception hierarchy: JDBC uses the rather uninformative SQLException for all its exceptions,* which is rather difficult to interpret. The following is an example of a standard database violation exception:

SQLException java.sql.BatchUpdateException: DB2-008: unique constraint (DB2_CS003) violated Such an exception needs to be interpreted based on the vendor-specific database error code — a mundane task to be sure.

Object-Relational Mapping (ORM)

- Relational databases are the most common type of databases in a majority of organizations today when compared to other formats (for example, object-oriented, hierarchical, network).
- On the computer languages side of things, object-oriented (OO) programming has become the norm. Languages such as Java, C#, C++, and even OO scripting languages are common discussion topics among developers.
- A majority of the software applications that use relational database and OO languages end up writing code to map the relational model to the OO model.
- This can involve anywhere from cumbersome mapping code (because of the use of embedded SQL or stored procedure calls) to heavy-handed technology, such as EJB's entity beans.

ORM

- Because most of us seem to like both relational databases and OO, Object-Relational Mapping (ORM) has become a natural choice for working with POJOs (plain old Java objects), especially if you don't need the distributed and secure execution of EJB's entity beans (which also map object attributes to relational database fields).
- Although you still need to map the relational model to the OO model, the mapping is typically done outside of the programming language, such as in XML files.
- Also, once this mapping is done for a given class, you can use instances of this class throughout your applications as POJOs.
- For example, you can use a save method for a given object and the underlying ORM framework will persist the data for you instead of you having to write tedious INSERT or UPDATE statements using JDBC, for example.
- Hibernate is one such ORM framework and given its popularity in the world of Java today, we will use it for Time Expression.

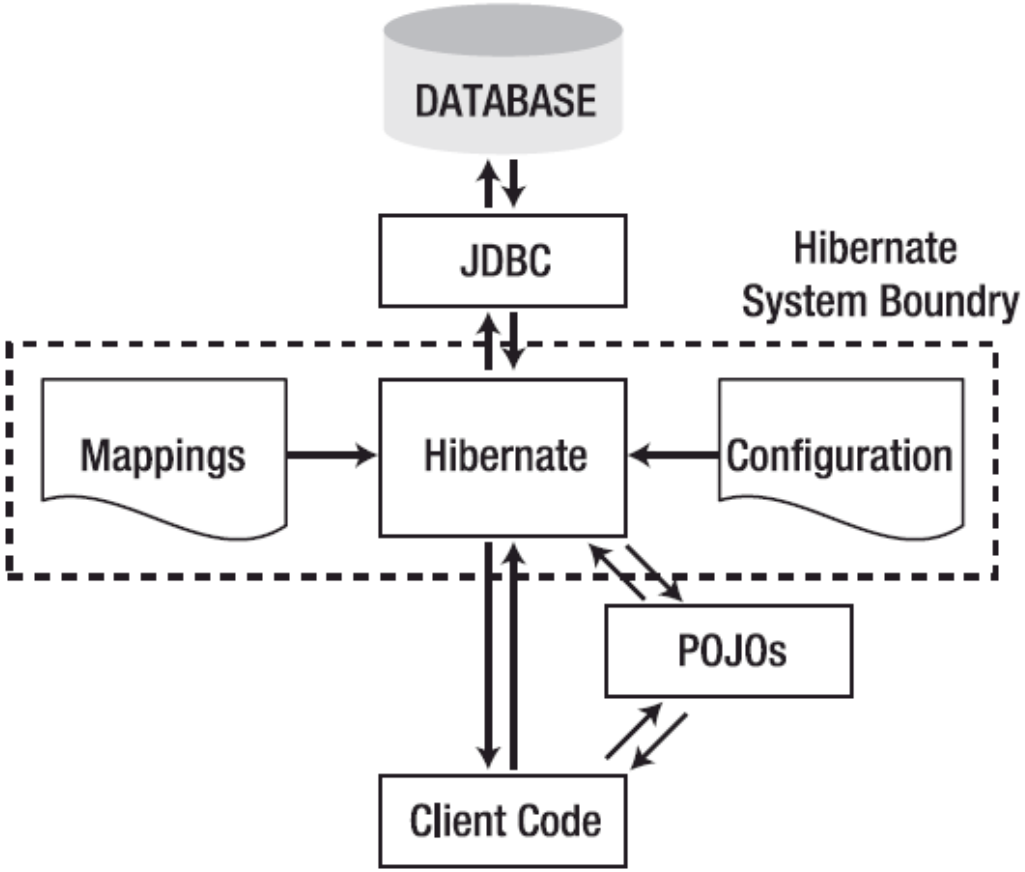
Plain Old Java Objects (POJOs)

- In our ideal world, it would be trivial to take any Java object and persist it to the database.
- No special coding would be required to achieve this, no performance penalty would ensue, and the result would be totally portable.
- In this ideal world, we would perhaps perform such an operation in a manner like that shown below:

```
POJO pojo = new POJO();
ORMSolution magic = ORMSolution.getInstance();
magic.save(pojo);
```

- There would be no nasty surprises, no additional work to correlate the class with tables in the database, and no performance problems.

Hibernate comes remarkably close to this, at least when compared with the alternatives—but alas, there are configuration files to create and subtle performance issues to consider. Hibernate does, however, achieve its fundamental aim—it allows you to store POJOs in the database.



The role of Hibernate in a Java application

The common term for the direct persistence of traditional Java objects is *object-relational mapping*—that is, mapping the objects in Java to the relational entities in a database.

A Thin Solution?

- One of the benefits often touted for Hibernate is that it is a “thin” solution.
- The problem with this description is that it is very much an informal term, so it doesn’t really tell you anything about what attributes Hibernate has that could categorize it as thin.
- Hibernate does not require an application server to operate (while EJBs do). It is therefore applicable in client-side applications in which EJBs are entirely inappropriate.
- So from this point of view, it is perhaps thin.

A Hibernate Hello World Example

The Hibernate Approach to Retrieving the POJO

```
public static List getMessages(int messageId)
   throws MessageException
{
   SessionFactory sessions =
      new Configuration().configure().buildSessionFactory();
   Session session = sessions.openSession();
   Transaction tx = null;
   try {
      tx = session.beginTransaction();

      List list = session.createQuery("from Message").list();

      tx.commit();
      tx = null;
      return list;

   } catch ( HibernateException e ) {
      if ( tx != null ) tx.rollback();
      log.log(Level.SEVERE, "Could not acquire message", e);
      throw new MotdException(
            "Failed to retrieve message from the database.",e);
   } finally {
      session.close();
   }
}
```

Mappings

As we have intimated, Hibernate needs something to tell it which tables relate to which objects (this information is usually provided in an XML mapping file). While some tools inflict vast, poorly documented XML configuration files on their users, Hibernate offers a breath of fresh air—you create and associate a small, clear mapping file with each of the POJOs that you wish to map into the database. You're permitted to use a single monolithic configuration file if you prefer, but it's neither compulsory nor encouraged.

A document type definition (DTD) is provided for all of Hibernate's configuration files, so with a good XML editor you should be able to take advantage of autocompletion and autovalidation of the files as you create them. Java 5 annotations can be used to replace them entirely.

The XML File That Maps the POJO to the Database

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
   <class name="Message" table="Message">
      <id type="int" column="id">
         <generator class="native"/>
      </id>
      <property name="message" column="message" type="string"/>
   </class>
</hibernate-mapping>
```

Complexity?

- It would be reasonable to ask if the complexity has simply been moved from the application code into the XML mapping file.
  - But, in fact, this isn't really the case for several reasons.

1. the XML file is much easier to edit than a complex population of a POJO from a result set—and far more easily changed after the fact should your mappings into the database change at a late stage of development.

2. we have still done away with the complicated error handling that was required with the JDBC approach.

3. it is of particular note that Hibernate solves the problem in which the developer needs to extract from the database a single object that is related by references to a substantial number of objects in the database. Hibernate postpones such additional extractions until they are actually accessed, generally avoiding substantial memory and performance costs.

The Steps Needed to Integrate and Configure Hibernate

1. Identify the POJOs that have a database representation.
2. Identify which properties of those POJOs need to be persisted.
3. Create Hibernate XML mapping files for each of the POJOs that map properties to columns in a table
4. Create the database schema using the schema export tool, use an existing database, or create your own database schema.
5. Add the Hibernate Java libraries to your application's classpath.
6. Create a Hibernate XML configuration file that points to your database and your XML mapping files.
7. In your Java application, create a Hibernate Configuration object that references your XML configuration file.
8. Also in your Java application, build a Hibernate SessionFactory object from the Configuration object
9. Finally, retrieve Hibernate Session objects from the SessionFactory, and write your data access logic for your application (create, retrieve, update, and delete).

Understanding Where Hibernate Fits in Your Java Application

- **You can call Hibernate from your Java application directly, or you can access Hibernate through another framework.**
- You can call Hibernate from a Swing application, a servlet, a portlet, a JSP page, or any other Java application that has access to a database.
- Typically, you would use Hibernate to either create a data access layer for an application or replace an existing data access layer.
- In addition, Hibernate uses standard Java Database Connectivity (JDBC) database drivers to access the relational database.
- **Hibernate does not replace JDBC as a database connectivity layer—Hibernate sits on a level above JDBC.**
- In addition to the standard Java APIs, many Java web and application frameworks now integrate with Hibernate.
- Hibernate's simple, clean API makes it easy for these frameworks to support Hibernate in one way or another.
- The Spring framework provides excellent Hibernate integration, including generic support for persistence objects, a generic set of persistence

Configuration Object

- An instance of org.hibernate.cfg.Configuration represents an entire set of mappings of an application's Java types to an SQL database.
- The org.hibernate.cfg.Configuration is used to build an immutable org.hibernate.SessionFactory. The mappings are compiled from various XML mapping files.
- You can obtain a org.hibernate.cfg.Configuration instance by instantiating it directly and specifying XML mapping documents. If the mapping files are in the classpath, use addResource(). For example:

```
Configuration cfg = new Configuration()
    .addResource("Item.hbm.xml")
    .addResource("Bid.hbm.xml");
```

- An alternative way is to specify the mapped class and allow Hibernate to find the mapping document for you:

```
Configuration cfg = new Configuration()
  .addClass(org.hibernate.auction.Item.class)
 .addClass(org.hibernate.auction.Bid.class);
```

- Hibernate will then search for mapping files named /org/hibernate/auction/Item.hbm.xml and /org/hibernate/auction/Bid.hbm.xml in the classpath. This approach eliminates any hardcoded filenames.

The Session Factory

- You use the Hibernate session factory to create Session objects that manage connection data, caching, and mappings.
- Your application is responsible for managing the session factory.
- You should only have one session factory unless you are using Hibernate to connect to two or more database instances with different settings, in which case you should still have one session factory for each database instance.
- In order to maintain backward compatibility with Hibernate 2, the Hibernate 3 session factory can also create org.hibernate.classic.Session session objects. These “classic” session objects implement all of the Hibernate 3 session functionality in addition to the deprecated Hibernate 2 session methods.

Session Object

- You obtain a session from the SessionFactory object using one of the four openSession() methods.
- The no-argument openSession() method opens a session, with the database connection and interceptor specified in the SessionFactory's original configuration.
- You can explicitly pass a JDBC connection to use, a Hibernate interceptor, or both as arguments to the remaining openSession() methods.

Creating a Hibernate Configuration File

There are several ways that Hibernate can be given all of the information that it needs to connect to the database and determine its mappings. For our Message example, we used the configuration file `hibernate.cfg.xml` placed in our project's `src` directory and given in Listing 3-4.

Listing 3-4. *The Message Application's Mapping File*

```
<?xml version='1.0' encoding='utf-8'?>
  <session-factory>
    <property name="hibernate.connection.url">
       jdbc:hsqldb:file:testdb;shutdown=true
    </property>
    <property name="hibernate.connection.driver_class">
       org.hsqldb.jdbcDriver
    </property>
    <property name="hibernate.connection.username">sa</property>
    <property name="hibernate.connection.password"></property>
    <property name="hibernate.connection.pool_size">0</property>
    <property name="hibernate.dialect">
       org.hibernate.dialect.HSQLDialect
    </property>
    <property name="hibernate.show_sql">false</property>

    <!-- "Import" the mapping resources here -->
    <mapping resource="sample/entity/Message.hbm.xml"/>

  </session-factory>
</hibernate-configuration>
```

Running the Message Example

With Hibernate and a database installed, and our configuration file created, all we need to do now is create the classes in full, and then build and run everything. Chapter 1 omitted the trivial parts of the required classes, so we provide them in full in Listings 3-5 through 3-7, after which we'll look at some of the details of what's being invoked.

Listing 3-5. *The* Message *POJO Class*

```
package sample.entity;

public class Message {
   private String message;

   public Message(String message) {
      this.message = message;
   }

   Message() {
   }

   public String getMessage() {
      return this.message;
   }

   public void setMessage(String message) {
      this.message = message;
   }
}
```

```
public class ListMessages {
   public static void main(String[] args)
   {
      SessionFactory factory =
         new Configuration().configure().buildSessionFactory();
      Session session = factory.openSession();

      List messages = session.createQuery("from Message").list();
      System.out.println("Found " + messages.size() + " message(s):");

      Iterator i = messages.iterator();
      while(i.hasNext()) {
         Message msg = (Message)i.next();
         System.out.println(msg.getMessage());
      }

      session.close();
   }
}
```