covers Spring 3.0

Spring IN ACTION

THIRD EDITION

Craig Walls

MANNING

Table of Contents

5 Hitting the database

This chapter covers

- Defining Spring's data access support
- Configuring database resources
- Working with Spring's JDBC templates
- Using Spring with Hibernate and JPA

With the core of the Spring container now under your belt, it's time to put it to work in real applications. A perfect place to start is with a requirement of nearly any enterprise application: persisting data. Every one of us has probably dealt with database access in an application in the past. In practice, we know that data access has many pitfalls. We have to initialize our data access framework, open connections, handle various exceptions, and close connections. If we get any of this wrong, we could potentially corrupt or delete valuable company data. In case you haven't experienced the consequences of mishandled data access, it's a *Bad Thing*.

Since we strive for *Good Things*, we turn to Spring. Spring comes with a family of data access frameworks that integrate with a variety of data access technologies. Whether you're persisting your data via direct JDBC, iBATIS, or an object relational mapping (ORM) framework such as Hibernate, Spring removes the tedium of data access from your persistence code. Instead, you can lean on Spring to handle the

low-level data access work for you so that you can turn your attention to managing your application's data.

Starting in this chapter, we'll build a Twitter-like application based on Spring called Spitter. This application will be the primary example for the rest of this book. The first order of business is to develop Spitter's persistence layer.

As we develop the persistence layer, we're faced with some choices. We could use JDBC, Hibernate, the Java Persistence API (JPA), or any of a number of persistence frameworks. Fortunately, Spring supports all of those persistence mechanisms. We'll take each of them for a spin in this chapter.

But first, let's lay some groundwork by getting familiar with Spring's persistence philosophy.

5.1 Learning Spring's data access philosophy

From the previous chapters, you know that one of Spring's goals is to allow you to develop applications following the sound object-oriented (OO) principle of coding to interfaces. Spring's data access support is no exception.

DAO[1] stands for *data access object*, which perfectly describes a DAO's role in an application. DAOs exist to provide a means to read and write data to the database. They should expose this functionality through an interface by which the rest of the application will access them. Figure 5.1 shows the proper approach to designing your data access tier.

As you can see, the service objects are accessing the DAOs through interfaces. This has a couple of positive consequences. First, it makes your service objects easily testable, since they're not coupled to a specific data access implementation. In fact, you could create mock implementations of these data access interfaces. That would allow you to test your service object without ever having to connect to the database, which would significantly speed up your unit tests and rule out the chance of a test failure due to inconsistent data.

In addition, the data access tier is accessed in a persistence technology-agnostic manner. The chosen persistence approach is isolated to the DAO while only the

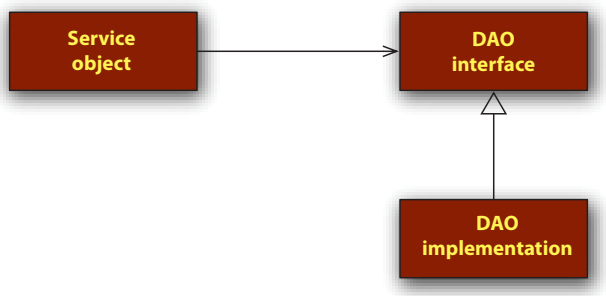

Figure 5.1 Service objects don't handle their own data access. Instead, they delegate data access to DAOs. The DAO's interface keeps it loosely coupled to the service object.

[1] Many developers, including Martin Fowler, refer to the persistence objects of an application as *repositories*. Though I appreciate the thinking that leads to the repository moniker, I believe that the word repository is already overloaded, even without adding this additional meaning. So forgive me, but I'm going to buck the popular trend—I'll continue referring to these objects as DAOs.

relevant data access methods are exposed through the interface. This makes for a flexible application design and allows the chosen persistence framework to be swapped out with minimal impact to the rest of the application. If the implementation details of the data access tier were to leak into other parts of the application, the entire application would become coupled with the data access tier, leading to a rigid application design.

> **NOTE** If after reading the last couple of paragraphs, you feel that I have a strong bias toward hiding the persistence layer behind interfaces, then I'm happy that I was able to get that point across. I believe that interfaces are key to writing loosely coupled code and that they should be used at all layers of an application, not just at the data access layer. That said, it's also important to note that though Spring encourages the use of interfaces, Spring doesn't require them—you're welcome to use Spring to wire a bean (DAO or otherwise) directly into a property of another bean without an interface between them.

One way Spring helps you insulate your data access tier from the rest of your application is by providing a consistent exception hierarchy that's used across all of its DAO frameworks.

5.1.1 Getting to know Spring's data access exception hierarchy

There's an old joke about a skydiver who's blown off course and ends up landing in a tree, dangling above the ground. After awhile someone walks by and the skydiver asks where he is.

> The passerby answers, "You're about 20 feet off the ground."
> The skydiver replies "You must be a software analyst."
> "You're right. How did you know?" asks the passerby.
> "Because what you told me was 100 percent accurate, but completely worthless."

That story has been told several times, with the profession or nationality of the passerby different each time. But the story reminds me of JDBC's `SQLException`. If you've ever written JDBC code (without Spring), you're probably keenly aware that you can't do anything with JDBC without being forced to catch `SQLException`. `SQLException` means that something went wrong while trying to access a database. But there's little about that exception that tells you what went wrong or how to deal with it.

Some common problems that might cause an `SQLException` to be thrown include

- The application is unable to connect to the database.
- The query being performed has errors in its syntax.
- The tables and/or columns referred to in the query don't exist.
- An attempt was made to insert or update values that violate a database constraint.

The big question surrounding `SQLException` is how it should be handled when it's caught. As it turns out, many of the problems that trigger an `SQLException` can't be remedied within a `catch` block. Most `SQLExceptions` that are thrown indicate a fatal

condition. If the application can't connect to the database, that usually means that the application will be unable to continue. Likewise, if there are errors in the query, little can be done about it at runtime.

If there's nothing that can be done to recover from an `SQLException`, why are we forced to catch it?

Even if you have a plan for dealing with some `SQLExceptions`, you'll have to catch the `SQLException` and dig around in its properties for more information on the nature of the problem. That's because `SQLException` is treated as a one-size-fits-all exception for problems related to data access. Rather than have a different exception type for each possible problem, `SQLException` is the exception that's thrown for all data access problems.

Some persistence frameworks offer a richer hierarchy of exceptions. Hibernate, for example, offers almost two dozen different exceptions, each targeting a specific data access problem. This makes it possible to write `catch` blocks for the exceptions that you want to deal with.

Even so, Hibernate's exceptions are specific to Hibernate. As stated before, we'd like to isolate the specifics of the persistence mechanism to the data access layer. If Hibernate-specific exceptions are being thrown, then the fact that we're dealing with Hibernate will leak into the rest of the application. Either that, or you'll be forced to catch persistence platform exceptions and rethrow them as platform-agnostic exceptions.

On one hand, JDBC's exception hierarchy is too generic—it's not really much of a hierarchy at all. On the other hand, Hibernate's exception hierarchy is proprietary to Hibernate. What we need is a hierarchy of data access exceptions that are descriptive but not directly associated with a specific persistence framework.

SPRING'S PERSISTENCE PLATFORM-AGNOSTIC EXCEPTIONS

Spring JDBC provides a hierarchy of data access exceptions that solve both problems. In contrast to JDBC, Spring provides several data access exceptions, each descriptive of the problem that they're thrown for. Table 5.1 shows some of Spring's data access exceptions lined up against the exceptions offered by JDBC.

As you can see, Spring has an exception for virtually anything that could go wrong when reading or writing to a database. And the list of Spring's data access exceptions is more vast than what's shown in table 5.1. (I would've listed them all, but I didn't want JDBC to get an inferiority complex.)

Even though Spring's exception hierarchy is far richer than JDBC's simple `SQLException`, it isn't associated with any particular persistence solution. This means that you can count on Spring to throw a consistent set of exceptions, regardless of which persistence provider you choose. This helps to keep your persistence choice confined to the data access layer.

Table 5.1 JDBC's exception hierarchy versus Spring's data access exceptions

JDBC's exceptions	Spring's data access exceptions
BatchUpdateException DataTruncation SQLException SQLWarning	CannotAcquireLockException CannotSerializeTransactionException CleanupFailureDataAccessException ConcurrencyFailureException DataAccessException DataAccessResourceFailureException DataIntegrityViolationException DataRetrievalFailureException DeadlockLoserDataAccessException EmptyResultDataAccessException IncorrectResultSizeDataAccessException IncorrectUpdateSemanticsDataAccessException InvalidDataAccessApiUsageException InvalidDataAccessResourceUsageException OptimisticLockingFailureException PermissionDeniedDataAccessException PessimisticLockingFailureException TypeMismatchDataAccessException UncategorizedDataAccessException

LOOK, MA! NO CATCH BLOCKS!

What isn't evident from table 5.1 is that all of those exceptions are rooted with `DataAccessException`. What makes `DataAccessException` special is that it's an unchecked exception. In other words, you don't have to catch any of the data access exceptions thrown from Spring (although you're perfectly welcome to if you'd like).

`DataAccessException` is just one example of Spring's across-the-board philosophy of checked versus unchecked exceptions. Spring takes the stance that many exceptions are the result of problems that can't be addressed in a `catch` block. Instead of forcing developers to write `catch` blocks (which are often left empty), Spring promotes the use of unchecked exceptions. This leaves the decision of whether to catch an exception in the developer's hands.

To take advantage of Spring's data access exceptions, you must use one of Spring's supported data access templates. Let's look at how Spring templates can greatly simplify data access.

5.1.2 *Templating data access*

You've probably traveled by plane before. If so, you'll surely agree that one of the most important parts of traveling is getting your luggage from point A to point B. There are many steps to this process. When you arrive at the terminal, your first stop will be at the counter to check your luggage. Next, security will scan it to ensure the safety of the flight. Then it takes a ride on the "luggage train" on its way to being placed on the plane. If you need to catch a connecting flight, your luggage needs to be moved as well. When you arrive at your final destination, the luggage has to be removed from

the plane and placed on the carousel. Finally, you go down to the baggage claim area and pick it up.

Even though there are many steps to this process, you're only actively involved in a couple of those steps. The carrier itself is responsible for driving the process. You're only involved when you need to be; the rest is taken care of. This mirrors a powerful design pattern: the Template Method pattern.

A template method defines the skeleton of a process. In our example, the process is moving luggage from departure city to arrival city. The process itself is fixed; it never changes. The overall sequence of events for handling luggage occurs the same way every time: luggage is checked in, luggage is loaded onto the plane, and so forth. Some steps of the process are fixed as well—some steps happen the same every time. When the plane arrives at its destination, every piece of luggage is unloaded one at a time and placed on a carousel to be taken to baggage claim.

At certain points, the process delegates its work to a subclass to fill in some implementation-specific details. This is the variable part of the process. For example, the handling of luggage starts with a passenger checking in the luggage at the counter. This part of the process always has to happen at the beginning, so its sequence in the process is fixed. Because each passenger's luggage check-in is different, the implementation of this part of the process is determined by the passenger. In software terms, a template method delegates the implementation-specific portions of the process to an interface. Different implementations of this interface define specific implementations of this portion of the process.

This is the same pattern that Spring applies to data access. No matter what technology we're using, certain data access steps are required. For example, we always need to obtain a connection to our data store and clean up resources when we're done. These are the fixed steps in a data access process. But each data access method we write is slightly different. We query for different objects and update the data in different ways. These are the variable steps in the data access process.

Spring separates the fixed and variable parts of the data access process into two distinct classes: *templates* and *callbacks*. Templates manage the fixed part of the process, whereas your custom data access code is handled in the callbacks. Figure 5.2 shows the responsibilities of both of these classes.

As you can see in figure 5.2, Spring's template classes handle the fixed parts of data access—controlling transactions, managing resources, and handling exceptions.

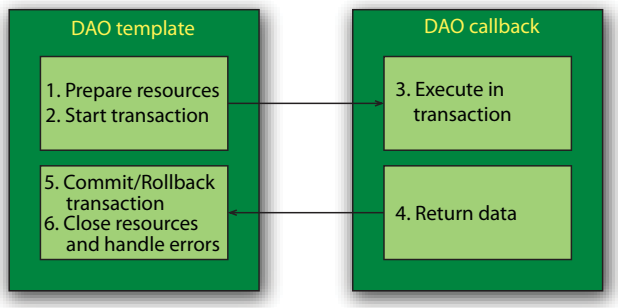

Figure 5.2 Spring's DAO template classes take responsibility for the common data access duties. For the application-specific tasks, it calls back into a custom DAO callback object.

Meanwhile, the specifics of data access as they pertain to your application—creating statements, binding parameters, and marshaling result sets—are handled in the callback implementation. In practice, this makes for an elegant framework because all you have to worry about is your data access logic.

Spring comes with several templates to choose from, depending on your persistence platform choice. If you're using straight JDBC, then you'll want to use `JdbcTemplate`. But if you favor one of the object-relational mapping frameworks, then perhaps `HibernateTemplate` or `JpaTemplate` is more suitable. Table 5.2 lists all of Spring's data access templates and their purposes.

Table 5.2 Spring comes with several data access templates, each suitable for a different persistence mechanism.

Template class (`org.springframework.*`)	Used to template. . .
`jca.cci.core.CciTemplate`	JCA CCI connections
`jdbc.core.JdbcTemplate`	JDBC connections
`jdbc.core.namedparam.NamedParameterJdbcTemplate`	JDBC connections with support for named parameters
`jdbc.core.simple.SimpleJdbcTemplate`	JDBC connections, simplified with Java 5 constructs
`orm.hibernate.HibernateTemplate`	Hibernate 2.*x* sessions
`orm.hibernate3.HibernateTemplate`	Hibernate 3.*x* sessions
`orm.ibatis.SqlMapClientTemplate`	iBATIS SqlMap clients
`orm.jdo.JdoTemplate`	Java Data Object implementations
`orm.jpa.JpaTemplate`	Java Persistence API entity managers

As you'll see, using a data access template simply involves configuring it as a bean in the Spring context and then wiring it into your application DAO. Or you can take advantage of Spring's DAO support classes to further simplify configuration of your application DAOs. Direct wiring of the templates is fine, but Spring also provides a set of convenient DAO base classes that can manage templates for you. Let's see how these template-based DAO classes work.

5.1.3 Using DAO support classes

The data access templates aren't all there is to Spring's data access framework. Each template also provides convenience methods that simplify data access without the need to create an explicit callback implementation. Furthermore, on top of the template-callback design, Spring provides DAO support classes that are meant to be subclassed by your own DAO classes. Figure 5.3 illustrates the relationship between a template class, a DAO support class, and your own custom DAO implementation.

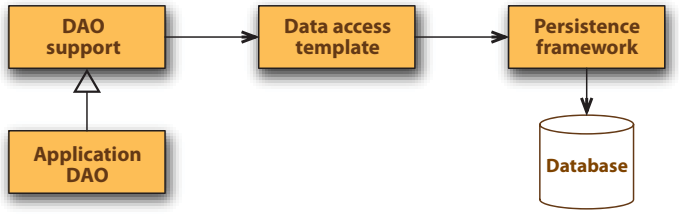

Figure 5.3 The relationship between an application DAO and Spring's DAO support and template classes

Later, as we examine Spring's individual data access support options, we'll see how the DAO support classes provide convenient access to the template class that they support. When writing your application DAO implementation, you can subclass a DAO support class and call a template retrieval method to have direct access to the underlying data access template. For example, if your application DAO subclasses `JdbcDaoSupport`, then you only need to call `getJdbcTemplate()` to get a `JdbcTemplate` to work with.

Plus, if you need access to the underlying persistence platform, each of the DAO support classes provides access to whatever class it uses to communicate with the database. For instance, the `JdbcDaoSupport` class contains a `getConnection()` method for dealing directly with the JDBC connection.

Just as Spring provides several data access template implementations, it also provides several DAO support classes—one for each template. Table 5.3 lists the DAO support classes that come with Spring.

Even though Spring provides support for several persistence frameworks, there isn't enough space to cover them all in this chapter. Therefore, we're going to focus on what I believe are the most beneficial persistence options and the ones that you'll most likely be using.

Table 5.3 Spring's DAO support classes provide convenient access to their corresponding data access template.

DAO support class (`org.springframework.*`)	Provides DAO support for. . .
`jca.cci.support.CciDaoSupport`	JCA CCI connections
`jdbc.core.support.JdbcDaoSupport`	JDBC connections
`jdbc.core.namedparam.NamedParameterJdbcDaoSupport`	JDBC connections with support for named parameters
`jdbc.core.simple.SimpleJdbcDaoSupport`	JDBC connections, simplified with Java 5 constructs
`orm.hibernate.support.HibernateDaoSupport`	Hibernate 2.*x* sessions
`orm.hibernate3.support.HibernateDaoSupport`	Hibernate 3.*x* sessions
`orm.ibatis.support.SqlMapClientDaoSupport`	iBATIS SqlMap clients
`orm.jdo.support.JdoDaoSupport`	Java Data Object implementations
`orm.jpa.support.JpaDaoSupport`	Java Persistence API entity managers

We'll start with basic JDBC access, as it's the most basic way to read and write data from a database. Then we'll look at Hibernate and JPA, two of the most popular POJO-based ORM solutions.

But first things first—most of Spring's persistence support options will depend on a data source. So, before we can get started with creating templates and DAOs, we need to configure Spring with a data source for the DAOs to access the database.

5.2 Configuring a data source

Regardless of which form of Spring DAO support you use, you'll likely need to configure a reference to a data source. Spring offers several options for configuring data source beans in your Spring application, including

- Data sources that are defined by a JDBC driver
- Data sources that are looked up by JNDI
- Data sources that pool connections

For production-ready applications, I recommend using a data source that draws its connections from a connection pool. When possible, I prefer to retrieve the pooled data source from an application server via JNDI. With that preference in mind, let's start by looking at how to configure Spring to retrieve a data source from JNDI.

5.2.1 Using JNDI data sources

Spring applications will often be deployed to run within a Java EE application server such as WebSphere, JBoss, or even a web container like Tomcat. These servers allow you to configure data sources to be retrieved via JNDI. The benefit of configuring data sources in this way is that they can be managed completely external to the application, allowing the application to ask for a data source when it's ready to access the database. Moreover, data sources managed in an application server are often pooled for greater performance and can be hot-swapped by system administrators.

With Spring, we can configure a reference to a data source that's kept in JNDI and wire it into the classes that need it as if it were just another Spring bean. The `<jee:jndi-lookup>` element from Spring's `jee` namespace makes it possible to retrieve any object, including data sources, from JNDI and make it available as a Spring bean. For example, if our application's data source were configured in JNDI, we might use `<jee:jndi-lookup>` like this to wire it into Spring:

```
<jee:jndi-lookup id="dataSource"
    jndi-name="/jdbc/SpitterDS"
    resource-ref="true" />
```

The `jndi-name` attribute is used to specify the name of the resource in JNDI. If only the `jndi-name` property is set, then the data source will be looked up using the name given as is. But if the application is running within a Java application server, then you'll want to set the `resource-ref` property to `true` so that the value given in `jndi-name` will be prepended with `java:comp/env/`.

5.2.2 Using a pooled data source

If you're unable to retrieve a data source from JNDI, the next best thing is to configure a pooled data source directly in Spring. Although Spring doesn't provide a pooled data source, there's a suitable one available in the Jakarta Commons Database Connection Pooling (DBCP) project (http://jakarta.apache.org/commons/dbcp).

DBCP includes several data sources that provide pooling, but the `BasicDataSource` is one that's often used because it's simple to configure in Spring and because it resembles Spring's own `DriverManagerDataSource` (which we'll talk about next).

For the Spitter application, we'll configure a `BasicDataSource` bean as follows:

```
<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
  <property name="url"
            value="jdbc:hsqldb:hsql://localhost/spitter/spitter" />
  <property name="username" value="sa" />
  <property name="password" value="" />
  <property name="initialSize" value="5" />
  <property name="maxActive" value="10" />
</bean>
```

The first four properties are elemental to configuring a `BasicDataSource`. The `driverClassName` property specifies the fully qualified name of the JDBC driver class. Here we've configured it with the JDBC driver for the Hypersonic database. The `url` property is where we set the complete JDBC URL for the database. Finally, the `username` and `password` properties are used to authenticate when we're connecting to the database.

Those four basic properties define connection information for `BasicDataSource`. In addition, several properties can be used to configure the data source pool itself. Table 5.4 lists a few of the most useful pool-configuration properties of `BasicDataSource`.

Table 5.4 's pool-configuration properties

Pool-configuration property	What it specifies
`initialSize`	The number of connections created when the pool is started.
`maxActive`	The maximum number of connections that can be allocated from the pool at the same time. If zero, there's no limit.
`maxIdle`	The maximum number of connections that can be idle in the pool without extras being released. If zero, there's no limit.
`maxOpenPreparedStatements`	The maximum number of prepared statements that can be allocated from the statement pool at the same time. If zero, there's no limit.
`maxWait`	How long the pool will wait for a connection to be returned to the pool (when there are no available connections) before an exception is thrown. If -1, wait indefinitely.

Table 5.4 's pool-configuration properties *(continued)*

Pool-configuration property	What it specifies
`minEvictableIdleTimeMillis`	How long a connection can remain idle in the pool before it's eligible for eviction.
`minIdle`	The minimum number of connections that can remain idle in the pool without new connections being created.
`poolPreparedStatements`	Whether or not to pool prepared statements (Boolean).

For our purposes, we've configured the pool to start with five connections. Should more connections be needed, `BasicDataSource` is allowed to create them, up to a maximum of ten active connections.

5.2.3 JDBC driver-based data source

The simplest data source you can configure in Spring is one that's defined through a JDBC driver. Spring offers two such data source classes to choose from (both in the `org.springframework.jdbc.datasource` package):

- `DriverManagerDataSource`—Returns a new connection every time that a connection is requested. Unlike DBCP's `BasicDataSource`, the connections provided by `DriverManagerDataSource` aren't pooled.
- `SingleConnectionDataSource`—Returns the same connection every time a connection is requested. Although `SingleConnectionDataSource` isn't exactly a pooled data source, you can think of it as a data source with a pool of exactly one connection.

Configuring either of these data sources is similar to how we configured DBCP's `BasicDataSource`:

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.
         DriverManagerDataSource">
  <property name="driverClassName"
            value="org.hsqldb.jdbcDriver" />
  <property name="url"
            value="jdbc:hsqldb:hsql://localhost/spitter/spitter" />
  <property name="username" value="sa" />
  <property name="password" value="" />
</bean>
```

The only difference is that since neither `DriverManagerDataSource` nor `SingleConnectionDataSource` provides a connection pool, there are no pool configuration properties to set.

Although `SingleConnectionDataSource` and `DriverManagerDataSource` are great for small applications and running in development, you should seriously consider the implications of using either in a production application. Because `SingleConnectionDataSource` has one and only one database connection to work with, it doesn't work

well in a multithreaded application. At the same time, even though `DriverManagerDataSource` is capable of supporting multiple threads, it incurs a performance cost for creating a new connection each time a connection is requested. Because of these limitations, I strongly recommend using pooled data sources.

Now that we've established a connection to the database through a data source, we're ready to actually access the database. As I've already mentioned, Spring affords us several options for working with databases, including JDBC, Hibernate, and the Java Persistence API (JPA). In the next section we'll see how to build the persistence layer of a Spring application using Spring's support for JDBC. But if Hibernate or JPA are more your style, then feel free to jump ahead to sections 5.4 and 5.5.

5.3 Using JDBC with Spring

There are many persistence technologies out there. Hibernate, iBATIS, and JPA are just a few. Despite this, a good number of applications are writing Java objects to a database the old-fashioned way: they earn it. No, wait—that's how people make money. The tried-and-true method for persisting data is with good old JDBC.

And why not? JDBC doesn't require mastering another framework's query language. It's built on top of SQL, which is the data access language. Plus, you can more finely tune the performance of your data access when you use JDBC than with practically any other technology. And JDBC allows you to take advantage of your database's proprietary features, where other frameworks may discourage or flat-out prohibit this.

What's more, JDBC lets you work with data at a much lower level than the persistence frameworks, allowing you to access and manipulate individual columns in a database. This fine-grained approach to data access comes in handy in applications, such as reporting applications, where it doesn't make sense to organize the data into objects, just to then unwind it back into raw data.

But all is not sunny in the world of JDBC. With its power, flexibility, and other niceties also come some not-so-niceties.

5.3.1 Tackling runaway JDBC code

Though JDBC gives you an API that works closely with your database, you're responsible for handling everything related to accessing the database. This includes managing database resources and handling exceptions.

If you've ever written JDBC that inserts data into the database, the following shouldn't be too alien to you.

Listing 5.1 Using JDBC to insert a row into a database

```
private static final String SQL_INSERT_SPITTER =
    "insert into spitter (username, password, fullname) values (?, ?, ?)";
private DataSource dataSource;
public void addSpitter(Spitter spitter) {
  Connection conn = null;
  PreparedStatement stmt = null;
```

```
  try {
    conn = dataSource.getConnection();

    stmt = conn.prepareStatement(SQL_INSERT_SPITTER);

    stmt.setString(1, spitter.getUsername());
    stmt.setString(2, spitter.getPassword());
    stmt.setString(3, spitter.getFullName());

    stmt.execute();

  } catch (SQLException e) {
    // do something...not sure what, though

  } finally {
    try {
      if (stmt != null) {
        stmt.close();
      }
      if (conn != null) {
        conn.close();
      }
    } catch (SQLException e) {
      // I'm even less sure about what to do here
    }
  }
}
```

Get connection

Create statement

Bind parameters

Execute statement

Handle exceptions (somehow)

Clean up

Holy runaway code, Batman! That's more than 20 lines of code to insert a simple object into a database. As far as JDBC operations go, this is about as simple as it gets. So why does it take this many lines to do something so simple? Actually, it doesn't. Only a handful of lines actually do the insert. But JDBC requires that you properly manage connections and statements and somehow handle the `SQLException` that may be thrown.

Speaking of that `SQLException`: not only is it not clear how you should handle it (because it's not clear what went wrong), but you're forced to catch it twice! You must catch it if something goes wrong while inserting a record, and you have to catch it again if something goes wrong when closing the statement and connection. Seems like a lot of work to handle something that usually can't be handled programmatically.

Now look at the following listing, where we use traditional JDBC to update a row in the Spitter table in the database.

Listing 5.2 Using JDBC to update a row in a database

```
private static final String SQL_UPDATE_SPITTER =
      "update spitter set username = ?, password = ?, fullname = ?"
      + "where id = ?";
public void saveSpitter(Spitter spitter) {
  Connection conn = null;
  PreparedStatement stmt = null;
  try {
    conn = dataSource.getConnection();
```

Get connection

```
    stmt = conn.prepareStatement(SQL_UPDATE_SPITTER);

    stmt.setString(1, spitter.getUsername());
    stmt.setString(2, spitter.getPassword());
    stmt.setString(3, spitter.getFullName());
    stmt.setLong(4, spitter.getId());

    stmt.execute();
  } catch (SQLException e) {
    // Still not sure what I'm supposed to do here
  } finally {
    try {
      if (stmt != null) {
        stmt.close();
      }
      if (conn != null) {
        conn.close();
      }
    } catch (SQLException e) {
      // or here
    }
  }
}
```

Create statement

Bind parameters

Execute statement

Handle exceptions (somehow)

Clean up

At first glance, listing 5.2 may appear to be identical to listing 5.1. In fact, disregarding the SQL `String` and the line where the statement is created, they're identical. Again, that's a lot of code to do something as simple as update a single row in a database. What's more, that's a lot of repeated code. Ideally, we'd only have to write the lines that are specific to the task at hand. After all, those are the only lines that distinguish listing 5.2 from listing 5.1. The rest is just boilerplate code.

To round out our tour of conventional JDBC, let's see how you might retrieve data out of the database. As you can see in the following, that's not pretty, either.

Listing 5.3 Using JDBC to query a row from a database

```
private static final String SQL_SELECT_SPITTER =
      "select id, username, fullname from spitter where id = ?";
public Spitter getSpitterById(long id) {
  Connection conn = null;
  PreparedStatement stmt = null;
  ResultSet rs = null;
  try {
    conn = dataSource.getConnection();

    stmt = conn.prepareStatement(SQL_SELECT_SPITTER);

    stmt.setLong(1, id);

    rs = stmt.executeQuery();

    Spitter spitter = null;
    if (rs.next()) {
      spitter = new Spitter();
      spitter.setId(rs.getLong("id"));
      spitter.setUsername(rs.getString("username"));
```

Get connection

Create statement

Bind parameter

Execute query

Process results

```
      spitter.setPassword(rs.getString("password"));
      spitter.setFullName(rs.getString("fullname"));
    }
    return spitter;
  } catch (SQLException e) {

  } finally {
    if(rs != null) {
      try {
        rs.close();
      } catch(SQLException e) {}
    }

    if(stmt != null) {
      try {
      stmt.close();
      } catch(SQLException e) {}
    }

    if(conn != null) {
      try {
        conn.close();
      } catch(SQLException e) {}
    }
  }

  return null;
}
```

Handle exceptions (somehow)

Clean up

That's about as verbose as the insert and update examples—maybe more. It's like the Pareto principle[2] flipped on its head: 20 percent of the code is needed to actually query a row whereas 80 percent is boilerplate code.

By now you should see that much of JDBC code is boilerplate code for creating connections and statements and exception handling. With my point made, I'll end the torture here and not make you look at any more of this nasty code.

But the fact is that this boilerplate code is important. Cleaning up resources and handling errors is what makes data access robust. Without it, errors would go undetected and resources would be left open, leading to unpredictable code and resource leaks. So not only do we need this code, we also need to make sure that it's correct. This is all the more reason to let a framework deal with the boilerplate code so that we know that it's written once and written right.

5.3.2 Working with JDBC templates

Spring's JDBC framework will clean up your JDBC code by shouldering the burden of resource management and exception handling. This leaves you free to write only the code necessary to move data to and from the database.

[2] http://en.wikipedia.org/wiki/Pareto%27s_principle

As I explained in section 5.3.1, Spring abstracts away the boilerplate data access code behind template classes. For JDBC, Spring comes with three template classes to choose from:

- `JdbcTemplate`—The most basic of Spring's JDBC templates, this class provides simple access to a database through JDBC and simple indexed-parameter queries.
- `NamedParameterJdbcTemplate`—This JDBC template class enables you to perform queries where values are bound to named parameters in SQL, rather than indexed parameters.
- `SimpleJdbcTemplate`—This version of the JDBC template takes advantage of Java 5 features such as autoboxing, generics, and variable parameter lists to simplify how a JDBC template is used.

At one time, you had to weigh your choice of JDBC template carefully. But as of the most recent versions of Spring, the decision is much easier. In Spring 2.5, the named parameter features of `NamedParameterJdbcTemplate` were merged into `SimpleJdbcTemplate`. And as of Spring 3.0, support for older versions of Java (prior to Java 5) has been dropped—so there's almost no reason to choose the plain `JdbcTemplate` over `SimpleJdbcTemplate`. In light of these changes, we'll focus solely on `SimpleJdbcTemplate` in this chapter.

ACCESSING DATA USING SIMPLEJDBCTEMPLATE

All that a `SimpleJdbcTemplate` needs to do its work is a `DataSource`. This makes it easy enough to configure a `SimpleJdbcTemplate` bean in Spring with the following XML:

```
<bean id="jdbcTemplate"
      class="org.springframework.jdbc.core.simple.SimpleJdbcTemplate">
   <constructor-arg ref="dataSource" />
</bean>
```

The actual `DataSource` being referred to by the `dataSource` property can be any implementation of `javax.sql.DataSource`, including those we created in section 5.2.

Now we can wire the `jdbcTemplate` bean into our DAO and use it to access the database. For example, suppose that the Spitter DAO is written to use `SimpleJdbcTemplate`:

```
public class JdbcSpitterDAO implements SpitterDAO {
...
  private SimpleJdbcTemplate jdbcTemplate;
  public void setJdbcTemplate(SimpleJdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
  }
}
```

You'd then wire the `jdbcTemplate` property of `JdbcSpitterDAO` as follows:

```
<bean id="spitterDao"
      class="com.habuma.spitter.persistence.SimpleJdbcTemplateSpitterDao">
  <property name="jdbcTemplate" ref="jdbcTemplate" />
</bean>
```

With a `SimpleJdbcTemplate` at our DAO's disposal, we can greatly simplify the `addSpitter()` method from listing 5.1. The new `SimpleJdbcTemplate`-based `addSpitter()` method is shown next.

Listing 5.4 A `SimpleJdbcTemplate`-based `addSpitter()` method

```
public void addSpitter(Spitter spitter) {
  jdbcTemplate.update(SQL_INSERT_SPITTER,
          spitter.getUsername(),
          spitter.getPassword(),
          spitter.getFullName(),
          spitter.getEmail(),
          spitter.isUpdateByEmail());
  spitter.setId(queryForIdentity());
}
```

Update Spitter

I think you'll agree that this version of `addSpitter()` is significantly simpler. There's no more connection or statement creation code—and no more exception-handling code. There's nothing but pure data insertion code.

Just because you don't see a lot of boilerplate code, that doesn't mean it's not there. It's cleverly hidden inside of the JDBC template class. When the `update()` method is called, `SimpleJdbcTemplate` will get a connection, create a statement, and execute the insert SQL.

What you also don't see is how the `SQLException` is handled. Internally, `SimpleJdbcTemplate` will catch any `SQLExceptions` that are thrown. It'll then translate the generic `SQLException` into one of the more specific data access exceptions from table 5.1 and rethrow it. Because Spring's data access exceptions are all runtime exceptions, we didn't have to catch it in the `addSpitter()` method.

Reading data is also simplified with `JdbcTemplate`. The following shows a new version of `getSpitterById()` that uses `SimpleJdbcTemplate` callbacks to map a result set to domain objects.

Listing 5.5 Querying for a `Spitter` using `SimpleJdbcTemplate`

```
public Spitter getSpitterById(long id) {
  return jdbcTemplate.queryForObject(
          SQL_SELECT_SPITTER_BY_ID,
      new ParameterizedRowMapper<Spitter>() {
        public Spitter mapRow(ResultSet rs, int rowNum)
            throws SQLException {
          Spitter spitter = new Spitter();
          spitter.setId(rs.getLong(1));
          spitter.setUsername(rs.getString(2));
          spitter.setPassword(rs.getString(3));
          spitter.setFullName(rs.getString(4));
          return spitter;
        }
      },
      id
      );
}
```

Queries for Spitter

Maps results to object

Binds parameters

This `getSpitterById()` method uses `SimpleJdbcTemplate`'s `queryForObject()` method to query for a `Spitter` from the database. The `queryForObject()` method takes three parameters:

- A `String` containing the SQL to be used to select the data from the database
- A `ParameterizedRowMapper` object that extracts values from a `ResultSet` and constructs a domain object (in this case a `Spitter`)
- A variable argument list of values to be bound to indexed parameters of the query

The real magic happens in the `ParameterizedRowMapper` object. For every row that results from the query, `JdbcTemplate` will call the `mapRow()` method of the `RowMapper`. Within `ParameterizedRowMapper`, we've written the code that creates a `Spitter` object and populates it with values from the `ResultSet`.

Just like `addSpitter()`, the `getSpitterById()` method is free from JDBC boilerplate code. Unlike traditional JDBC, there's no resource management or exception-handling code. Methods that use `SimpleJdbcTemplate` are laser-focused on retrieving a `Spitter` object from the database.

USING NAMED PARAMETERS

The `addSpitter()` method in listing 5.4 used indexed parameters. This meant that we had to take notice of the order of the parameters in the query and list the values in the correct order when passing them to the `update()` method. If we were to ever change the SQL in such a way that the order of the parameters would change, we'd also need to change the order of the values.

Optionally, we could use named parameters. Named parameters let us give each parameter in the SQL an explicit name and to refer to the parameter by that name when binding values to the statement. For example, suppose that the `SQL_INSERT_SPITTER` query were defined as follows:

```
private static final String SQL_INSERT_SPITTER =
      "insert into spitter (username, password, fullname) " +
      "values (:username, :password, :fullname)";
```

With named parameter queries, the order of the bound values isn't important. We can bind each value by name. If the query changes and the order of the parameters is no longer the same, we won't have to change the binding code.

In Spring 2.0 you'd have to rely on a special JDBC template class called `NamedParameterJdbcTemplate` to use named parameter queries. Prior to Spring 2.0, it wasn't even possible. But starting with Spring 2.5, the named parameter features of `NamedParameterJdbcTemplate` have been merged into `SimpleJdbcTemplate`, so you're already set to update your `addSpitter()` method to use named parameters. The following listing shows the new named-parameter version of `addSpitter()`.

Listing 5.6 Using named parameters with Spring JDBC templates

```
public void addSpitter(Spitter spitter) {
  Map<String, Object> params = new HashMap<String, Object>();
  params.put("username", spitter.getUsername());
  params.put("password", spitter.getPassword());
  params.put("fullname", spitter.getFullName());

  jdbcTemplate.update(SQL_INSERT_SPITTER, params);
  spitter.setId(queryForIdentity());
}
```

Bind parameters

Perform insert

The first thing you'll notice is that this version of `addSpitter()` is a bit longer than the previous version. That's because named parameters are bound through a `java.util.Map`. Nevertheless, every line is focused on the goal of inserting a `Spitter` object into the database. There's still no resource management or exception-handling code cluttering up the chief purpose of the method.

USING SPRING'S DAO SUPPORT CLASSES FOR JDBC

For each of our application's JDBC-backed DAO classes, we'll need to be sure to add a `SimpleJdbcTemplate` property and setter method. And we'll need to be sure to wire the `SimpleJdbcTemplate` bean into the `SimpleJdbcTemplate` property of each DAO. That's not a big deal if the application only has one DAO, but if you have multiple DAOs, that's a lot of repeated code.

One solution would be for you to create a common parent class for all your DAO objects where the `SimpleJdbcTemplate` property resides. Then all of your DAO classes would extend that class and use the parent class's `SimpleJdbcTemplate` for its data access. Figure 5.4 shows the proposed relationship between an application DAO and the base DAO class.

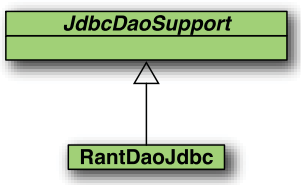

Figure 5.4 Spring's DAO support classes define a placeholder for the JDBC template objects so that subclasses won't have to manage their own JDBC templates.

The idea of creating a base DAO class that holds the JDBC template is such a good idea that Spring comes with just such a base class out of the box. Actually, it comes with three such classes—`JdbcDaoSupport`, `SimpleJdbcDaoSupport`, and `NamedParameterJdbcDaoSupport`—one to mirror each of Spring's JDBC templates. To use one of these DAO support classes, start by changing your DAO class to extend it. For example:

```
public class JdbcSpitterDao extends SimpleJdbcDaoSupport
    implements SpitterDao {
...
}
```

The `SimpleJdbcDaoSupport` provides convenient access to the `SimpleJdbcTemplate` through the `getSimpleJdbcTemplate()` method. For example, the `addSpitter()` method may be written like this:

```
public void addSpitter(Spitter spitter) {
  getSimpleJdbcTemplate().update(SQL_INSERT_SPITTER,
          spitter.getUsername(),
          spitter.getPassword(),
          spitter.getFullName(),
          spitter.getEmail(),
          spitter.isUpdateByEmail());
  spitter.setId(queryForIdentity());
}
```

When configuring your DAO class in Spring, you could directly wire a `SimpleJdbcTemplate` bean into its `jdbcTemplate` property as follows:

```
<bean id="spitterDao"
      class="com.habuma.spitter.persistence.JdbcSpitterDao">
  <property name="jdbcTemplate" ref="jdbcTemplate" />
</bean>
```

This will work, but it isn't much different from how you configured the DAO that didn't extend `SimpleJdbcDaoSupport`. Alternatively, you can skip the middleman (or middle bean, as the case may be) and wire a data source directly into the `dataSource` property that `JdbcSpitterDao` inherits from `SimpleJdbcDaoSupport`:

```
<bean id="spitterDao"
      class="com.habuma.spitter.persistence.JdbcSpitterDao">
  <property name="dataSource" ref="dataSource" />
</bean>
```

When `JdbcSpitterDao` has its `dataSource` property configured, it'll internally create a `SimpleJdbcTemplate` instance for you. This eliminates the need to explicitly declare a `SimpleJdbcTemplate` bean in Spring.

JDBC is the most basic way to access data in a relational database. Spring's JDBC templates save you the hassle of dealing with the boilerplate code that handles connection resources and exception handling, leaving you to focus on the actual work of querying and updating data.

Even though Spring takes much of the pain out of working with JDBC, it can still become cumbersome as applications grow larger and more complex. To help manage the persistence challenges of large applications, you may want to graduate to a persistence framework such as Hibernate. Let's see how to plug Hibernate in the persistence layer of a Spring application.

5.4 Integrating Hibernate with Spring

When we were kids, riding a bike was fun, wasn't it? We'd ride to school in the mornings. When school let out, we'd cruise to our best friend's house. When it got late and our parents were yelling at us for staying out past dark, we'd peddle home for the night. Gee, those days were fun.

Then we grew up, and now we need more than a bike. Sometimes we have to travel a long distance to work. Groceries have to be hauled, and ours kids need to get to soccer practice. And if you live in Texas, air conditioning is a must! Our needs have simply outgrown our bikes.

JDBC is the bike of the persistence world. It's great for what it does, and for some jobs it works fine. But as our applications become more complex, so do our persistence requirements. We need to be able to map object properties to database columns and have our statements and queries created for us, freeing us from typing an endless string of question marks. We also need features that are more sophisticated:

- *Lazy loading*—As our object graphs become more complex, we sometimes don't want to fetch entire relationships immediately. To use a typical example, suppose we're selecting a collection of `PurchaseOrder` objects, and each of these objects contains a collection of `LineItem` objects. If we're only interested in `PurchaseOrder` attributes, it makes no sense to grab the `LineItem` data. This could be expensive. Lazy loading allows us to grab data only as it's needed.
- *Eager fetching*—This is the opposite of lazy loading. Eager fetching allows you to grab an entire object graph in one query. In the cases where we know that we need a `PurchaseOrder` object and its associated `LineItems`, eager fetching lets us get this from the database in one operation, saving us from costly round-trips.
- *Cascading*—Sometimes changes to a database table should result in changes to other tables as well. Going back to our purchase order example, when an `Order` object is deleted, we also want to delete the associated `LineItems` from the database.

Several frameworks are available that provide these services. The general name for these services is *object-relational mapping (ORM)*. Using an ORM tool for your persistence layer can save you literally thousands of lines of code and hours of development time. This lets you switch your focus from writing error-prone SQL code to addressing your application requirements.

Spring provides support for several persistence frameworks, including Hibernate, iBATIS, Java Data Objects (JDO), and the Java Persistence API (JPA).

As with Spring's JDBC support, Spring's support for ORM frameworks provides integration points to the frameworks as well as some additional services:

- Integrated support for Spring declarative transactions
- Transparent exception handling
- Thread-safe, lightweight template classes
- DAO support classes
- Resource management

I don't have enough space in this chapter to cover all of the ORM frameworks that are supported by Spring. That's okay, because Spring's support for one ORM solution is similar to the next. Once you get the hang of using one ORM framework with Spring, you'll find it easy to switch to another one.

Let's get started by looking at how Spring integrates with what's perhaps the most popular ORM framework in use—Hibernate. Later in this chapter, we'll also look at how Spring integrates with JPA (in section 5.5).

Hibernate is an open source persistence framework that has gained significant popularity in the developer community. It provides not only basic object-relational mapping but also all the other sophisticated features you'd expect from a full-featured ORM tool, such as caching, lazy loading, eager fetching, and distributed caching.

In this section, we'll focus on how Spring integrates with Hibernate, without dwelling too much on the intricate details of using Hibernate. If you need to learn more about working with Hibernate, I recommend either *Java Persistence with Hibernate* (Manning, 2006) or the Hibernate website at http://www.hibernate.org.

5.4.1 A Hibernate overview

In the previous section we looked at how to work with JDBC through Spring's JDBC templates. As it turns out, Spring's support for Hibernate offers a similar template class to abstract Hibernate persistence. Historically, `HibernateTemplate` was the way to work with Hibernate in a Spring application. Like its JDBC counterpart, `HibernateTemplate` took care of the intricacies of working with Hibernate by catching Hibernate-specific exceptions and rethrowing them as one of Spring's unchecked data access exceptions.

One of the responsibilities of `HibernateTemplate` is to manage Hibernate `Sessions`. This involves opening and closing sessions as well as ensuring one session per transaction. Without `HibernateTemplate`, you'd have no choice but to clutter your DAOs with boilerplate session management code.

The downside of `HibernateTemplate` is that it's somewhat intrusive. When we use Spring's `HibernateTemplate` in a DAO (whether directly or through `HibernateDaoSupport`), the DAO class is coupled to the Spring API. Although this may not be of much concern to some developers, others may find Spring's intrusion into their DAO code undesirable.

Even though `HibernateTemplate` is still around, it's no longer considered the best way of working with Hibernate. *Contextual sessions*, introduced in Hibernate 3, are a way in which Hibernate itself manages one `Session` per transaction. There's no need for `HibernateTemplate` to ensure this behavior. This keeps your DAO classes free of Spring-specific code.

Since contextual sessions are the accepted best practice for working with Hibernate, we'll focus on them and not spend any more time on `HibernateTemplate`. If you're still curious about `HibernateTemplate` and want to see how it works, I refer you to the second edition of this book or to the example code that can be downloaded from http://www.manning.com/walls4/, where I include a `HibernateTemplate` example.

Before we dive into working with Hibernate's contextual sessions, we need to set the stage for Hibernate by configuring a Hibernate session factory in Spring.

5.4.2 Declaring a Hibernate session factory

Natively, the main interface for working with Hibernate is `org.hibernate.Session`. The `Session` interface provides basic data access functionality such as the ability to

save, update, delete, and load objects from the database. Through the Hibernate `Session`, an application's DAO will perform all of its persistence needs.

The standard way to get a reference to a Hibernate `Session` object is through an implementation of Hibernate's `SessionFactory` interface. Among other things, `SessionFactory` is responsible for opening, closing, and managing Hibernate `Session`s.

In Spring, the way to get a Hibernate `SessionFactory` is through one of Spring's Hibernate session factory beans. These session factory beans are implementations of Spring's `FactoryBean` interface that produce a Hibernate `SessionFactory` when wired into any property of type `SessionFactory`. This makes it possible to configure your Hibernate session factory alongside the other beans in your application's Spring context.

When it comes to configuring a Hibernate session factory bean, you have a choice to make. The decision hinges on whether you want to configure your persistent domain objects using Hibernate's XML mapping files or with annotations. If you choose to define your object-to-database mapping in XML, you'll need to configure `LocalSessionFactoryBean` in Spring :

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="mappingResources">
   <list>
    <value>Spitter.hbm.xml </value>
   </list>
  </property>
  <property name="hibernateProperties">
   <props>
    <prop key="dialect">org.hibernate.dialect.HSQLDialect</prop>
   </props>
  </property>
</bean>
```

`LocalSessionFactoryBean` is configured here with three properties. The `dataSource` property is wired with a reference to a `DataSource` bean. The `mappingResources` property lists one or more Hibernate mapping files that define the persistence strategy for the application. Finally, `hibernateProperties` is where we configure the minutia of how Hibernate should operate. In this case, we're saying that Hibernate will be working with a Hypersonic database and should use the `HSQLDialect` to construct SQL accordingly.

If annotation-oriented persistence is more your style, then you'll need to use `AnnotationSessionFactoryBean` instead of `LocalSessionFactoryBean`:

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.annotation.
  ➥AnnotationSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="packagesToScan"
```

```
      value="com.habuma.spitter.domain" />
  <property name="hibernateProperties">
   <props>
    <prop key="dialect">org.hibernate.dialect.HSQLDialect</prop>
   </props>
  </property>
 </bean>
```

As with `LocalSessionFactoryBean`, the `dataSource` and `hibernateProperties` properties tell where to find a database connection and what kind of database we'll be dealing with.

But instead of listing Hibernate mapping files, we can use the `packagesToScan` property to tell Spring to scan one or more packages looking for domain classes that are annotated for persistence with Hibernate. This includes classes that are annotated with JPA's `@Entity` or `@MappedSuperclass` and Hibernate's own `@Entity` annotation.

A list of one

`AnnotationSessionFactoryBean`'s `packagesToScan` property takes an array of `String`s specifying the packages to look for persistent classes in. Normally, I might specify such a list as follows:

```
<property name="packagesToScan">
    <list>
        <value>com.habuma.spitter.domain</value>
    </list>
</property>
```

But since I'm only asking it to scan a single package, I'm taking advantage of a built-in property editor that automatically converts a single `String` value into a `String` array.

If you'd prefer, you may also explicitly list out all of your application's persistent classes by specifying a list of fully qualified class names in the `annotatedClasses` property:

```
<property name="annotatedClasses">
    <list>
        <value>com.habuma.spitter.domain.Spitter</value>
        <value>com.habuma.spitter.domain.Spittle</value>
   </list>
</property>
```

The `annotatedClasses` property is fine for hand-picking a few domain classes. But `packagesToScan` is more appropriate if you have a lot of domain classes and don't want to list them all or if you want the freedom to add or remove domain classes without revisiting the Spring configuration.

With a Hibernate session factory bean declared in the Spring application context, we're ready to start creating our DAO classes.

5.4.3 Building Spring-free Hibernate

As mentioned before, without contextual sessions, Spring's Hibernate templates would handle the task of ensuring one session per transaction. But now that Hibernate manages this, there's no need for a template class. That means that you can wire a Hibernate session directly into your DAO classes.

Listing 5.7 Hibernate's contextual sessions enable Spring-free Hibernate DAOs.

```
package com.habuma.spitter.persistence;
import java.util.List;
import org.hibernate.SessionFactory;
import org.hibernate.classic.Session;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import com.habuma.spitter.domain.Spitter;
import com.habuma.spitter.domain.Spittle;

@Repository
public class HibernateSpitterDao implements SpitterDao {
  private SessionFactory sessionFactory;

  @Autowired
  public HibernateSpitterDao(SessionFactory sessionFactory) {
    this.sessionFactory = sessionFactory;
  }

  private Session currentSession() {
    return sessionFactory.getCurrentSession();
  }

  public void addSpitter(Spitter spitter) {
    currentSession().save(spitter);
  }

  public Spitter getSpitterById(long id) {
    return (Spitter) currentSession().get(Spitter.class, id);
  }

  public void saveSpitter(Spitter spitter) {
    currentSession().update(spitter);
  }
...
}
```

Construct DAO

Retrieve current Session from SessionFactory

Use current Session

There are several things to take note of in listing 5.7. First, note that we're using Spring's `@Autowired` annotation to have Spring automatically inject a `SessionFactory` into `HibernateSpitterDao`'s `sessionFactory` property. Then, in the `currentSession()` method, we use that `SessionFactory` to get the current transaction's session.

Also note that we've annotated the class with `@Repository`. This accomplishes two things for us. First, `@Repository` is another one of Spring's stereotype annotations that, among other things, are scanned by Spring's `<context:component-scan>`. This means that we won't have to explicitly declare a `HibernateSpitterDao` bean, as long as we configure `<context:component-scan>` like so:

```
<context:component-scan
     base-package="com.habuma.spitter.persistence" />
```

In addition to helping to reduce XML-based configuration, `@Repository` serves another purpose. Recall that one of the jobs of a template class is to catch platform-specific exceptions and rethrow them as one of Spring's unified unchecked exceptions. But if we're using Hibernate contextual sessions and not a Hibernate template, then how can the exception translation take place?

To add exception translation to a template-less Hibernate DAO, we just need to add a `PersistenceExceptionTranslationPostProcessor` bean to the Spring application context:

```
<bean class="org.springframework.dao.annotation.
  ➥PersistenceExceptionTranslationPostProcessor"/>
```

`PersistenceExceptionTranslationPostProcessor` is a bean post processor which adds an advisor to any bean that's annotated with `@Repository` so that any platform-specific exceptions are caught and then rethrown as one of Spring's unchecked data access exceptions.

And now the Hibernate version of our DAO is complete. And we were about to develop it without directly depending on any Spring-specific classes (aside from the `@Repository` annotation). That same template-less approach can also be applied when developing a pure JPA-based DAO. So, let's take one more stab at developing a `SpitterDao` implementation, this time using JPA.

5.5 Spring and the Java Persistence API

From its beginning, the EJB specification has included the concept of entity beans. In EJB, *entity beans* are a type of EJB that describes business objects that are persisted in a relational database. Entity beans have undergone several tweaks over the years, including *bean-managed persistence (BMP)* entity beans and *container-managed persistence (CMP)* entity beans.

Entity beans both enjoyed the rise and suffered the fall of EJB's popularity. In recent years, developers have traded in their heavyweight EJBs for simpler POJO-based development. This presented a challenge to the Java Community Process to shape the new EJB specification around POJOs. The result is JSR-220—also known as *EJB 3*.

The Java Persistence API (JPA) emerged out of the rubble of EJB 2's entity beans as the next-generation Java persistence standard. JPA is a POJO-based persistence mechanism that draws ideas from both Hibernate and *Java Data Objects (JDO)*, and mixes Java 5 annotations in for good measure.

With the Spring 2.0 release came the premiere of Spring integration with JPA. The irony is that many blame (or credit) Spring with the demise of EJB. But now that Spring provides support for JPA, many developers are recommending JPA for persistence in Spring-based applications. In fact, some say that Spring-JPA is the dream team for POJO development.

The first step toward using JPA with Spring is to configure an entity manager factory as a bean in the Spring application context.

5.5.1 Configuring an entity manager factory

In a nutshell, JPA-based applications use an implementation of `EntityManagerFactory` to get an instance of an `EntityManager`. The JPA specification defines two kinds of entity managers:

- *Application-managed*—Entity managers are created when an application directly requests one from an entity manager factory. With application-managed entity managers, the application is responsible for opening or closing entity managers and involving the entity manager in transactions. This type of entity manager is most appropriate for use in standalone applications that don't run within a Java EE container.
- *Container-managed*—Entity managers are created and managed by a Java EE container. The application doesn't interact with the entity manager factory at all. Instead, entity managers are obtained directly through injection or from JNDI. The container is responsible for configuring the entity manager factories. This type of entity manager is most appropriate for use by a Java EE container that wants to maintain some control over JPA configuration beyond what's specified in persistence.xml.

Both kinds of entity manager implement the same `EntityManager` interface. The key difference isn't in the `EntityManager` itself, but rather in how the `EntityManager` is created and managed. Application-managed `EntityManagers` are created by an `EntityManagerFactory` obtained by calling the `createEntityManagerFactory()` method of the `PersistenceProvider`. Meanwhile, container-managed `EntityManagerFactorys` are obtained through `PersistenceProvider`'s `createContainerEntityManagerFactory()` method.

So what does this all mean for Spring developers wanting to use JPA? Not much. Regardless of which variety of `EntityManagerFactory` you want to use, Spring will take responsibility for managing `EntityManagers` for you. If using an application-managed entity manager, Spring plays the role of an application and transparently deals with the `EntityManager` on your behalf. In the container-managed scenario, Spring plays the role of the container.

Each flavor of entity manager factory is produced by a corresponding Spring factory bean:

- `LocalEntityManagerFactoryBean` produces an application-managed `EntityManagerFactory`.
- `LocalContainerEntityManagerFactoryBean` produces a container-managed `EntityManagerFactory`.

It's important to point out that the choice made between an application-managed `EntityManagerFactory` and a container-managed `EntityManagerFactory` is completely transparent to a Spring-based application. Spring's `JpaTemplate` hides the intricate details of dealing with either form of `EntityManagerFactory`, leaving your data access code to focus on its true purpose: data access.

The only real difference between application-managed and container-managed entity manager factories, as far as Spring is concerned, is how each is configured within the Spring application context. Let's start by looking at how to configure the application-managed `LocalEntityManagerFactoryBean` in Spring. Then we'll see how to configure a container-managed `LocalContainerEntityManagerFactoryBean`.

CONFIGURING APPLICATION-MANAGED JPA

Application-managed entity manager factories derive most of their configuration information from a configuration file called persistence.xml. This file must appear in the META-INF directory within the classpath.

The purpose of the persistence.xml file is to define one or more persistence units. A persistence unit is a grouping of one or more persistent classes that correspond to a single data source. In simple terms, persistence.xml enumerates one or more persistent classes along with any additional configuration such as data sources and XML-based mapping files. Here's a typical example of a persistence.xml file as it pertains to the Spitter application:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    version="1.0">
  <persistence-unit name="spitterPU">
    <class>com.habuma.spitter.domain.Spitter</class>
    <class>com.habuma.spitter.domain.Spittle</class>
    <properties>
      <property name="toplink.jdbc.driver"
          value="org.hsqldb.jdbcDriver" />
      <property name="toplink.jdbc.url" value=
          "jdbc:hsqldb:hsql://localhost/spitter/spitter" />
      <property name="toplink.jdbc.user"
          value="sa" />
      <property name="toplink.jdbc.password"
          value="" />
    </properties>
  </persistence-unit>
</persistence>
```

Because so much configuration goes into a persistence.xml file, little configuration is required (or even possible) in Spring. The following `<bean>` declares a `LocalEntityManagerFactoryBean` in Spring:

```
<bean id="emf"
      class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
  <property name="persistenceUnitName" value="spitterPU" />
</bean>
```

The value given to the `persistenceUnitName` property refers to the persistence unit name as it appears in persistence.xml.

The reason why much of what goes into creating an application-managed `EntityManagerFactory` is contained in persistence.xml has everything to do with what it means to be application managed. In the application-managed scenario (not involving Spring), an application is entirely responsible for obtaining an `EntityManagerFactory` through the JPA implementation's `PersistenceProvider`. The application

code would become incredibly bloated if it had to define the persistence unit every time it requested an `EntityManagerFactory`. By specifying it in persistence.xml, JPA can look in this well-known location for persistence unit definitions.

But with Spring's support for JPA, we'll never deal directly with the `PersistenceProvider`. Therefore, it seems silly to extract configuration information into persistence.xml. In fact, doing so prevents us from configuring the `EntityManagerFactory` in Spring (so that, for example, we can provide a Spring-configured data source).

For that reason, we should turn our attention to container-managed JPA.

CONFIGURING CONTAINER-MANAGED JPA

Container-managed JPA takes a different approach. When running within a container, an `EntityManagerFactory` can be produced using information provided by the container—Spring, in our case.

Instead of configuring data source details in persistence.xml, you can configure this information in the Spring application context. For example, the following `<bean>` declaration shows how to configure container-managed JPA in Spring using `LocalContainerEntityManagerFactoryBean`.

```
<bean id="emf" class=
      "org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="jpaVendorAdapter" ref="jpaVendorAdapter" />
</bean>
```

Here we've configured the `dataSource` property with a Spring-configured data source. Any implementation of `javax.sql.DataSource` is appropriate, such as those that we configured in section 5.2. Although a data source may still be configured in persistence.xml, the data source specified through this property takes precedence.

The `jpaVendorAdapter` property can be used to provide specifics about the particular JPA implementation to use. Spring comes with a handful of JPA vendor adaptors to choose from:

- `EclipseLinkJpaVendorAdapter`
- `HibernateJpaVendorAdapter`
- `OpenJpaVendorAdapter`
- `TopLinkJpaVendorAdapter`

In this case, we're using Hibernate as a JPA implementation, so we've configured it with a `HibernateJpaVendorAdapter`:

```
<bean id="jpaVendorAdapter"
      class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
  <property name="database" value="HSQL" />
  <property name="showSql" value="true"/>
  <property name="generateDdl" value="false"/>
  <property name="databasePlatform"
            value="org.hibernate.dialect.HSQLDialect" />
</bean>
```

Database platform	Value for `database` property
IBM DB2	`DB2`
Apache Derby	`DERBY`
H2	`H2`
Hypersonic	`HSQL`
Informix	`INFORMIX`
MySQL	`MYSQL`
Oracle	`ORACLE`
PostgresQL	`POSTGRESQL`
Microsoft SQL Server	`SQLSERVER`
Sybase	`SYBASE`

Table 5.5 The Hibernate JPA vendor adapter supports several databases. You can specify which database to use by setting its property.

Several properties are set on the vendor adapter, but the most important one is the `database` property, where we've specified the Hypersonic database as the database we'll be using. Other values supported for this property include those listed in table 5.5.

Certain dynamic persistence features require that the class of persistent objects be modified with instrumentation to support the feature. Objects whose properties are lazily loaded (they won't be retrieved from the database until they're accessed) must have their class instrumented with code that knows to retrieve unloaded data upon access. Some frameworks use dynamic proxies to implement lazy loading. Others, such as JDO, perform class instrumentation at compile time.

Which entity manager factory bean you choose will depend primarily on how you'll use it. For simple applications, `LocalEntityManagerFactoryBean` may be sufficient. But because `LocalContainerEntityManagerFactoryBean` enables us to configure more of JPA in Spring, it's an attractive choice and likely the one that you'll choose for production use.

PULLING AN ENTITYMANAGERFACTORY FROM JNDI

It's also worth noting that if you're deploying your Spring application in some application servers, an `EntityManagerFactory` may have already been created for you and may be waiting in JNDI to be retrieved. In that case, you can use the `<jee:jndi-lookup>` element from Spring's `jee` namespace to nab a reference to the `EntityManagerFactory`:

```
<jee:jndi-lookup id="emf" jndi-name="persistence/spitterPU" />
```

Regardless of how you get your hands on an `EntityManagerFactory`, once you have one, you're ready to start writing a DAO. Let's do that now.

5.5.2 Writing a JPA-based DAO

Just like all of Spring's other persistence integration options, Spring-JPA integration comes in template form with `JpaTemplate` and a corresponding `JpaDaoSupport` class. Nevertheless, template-based JPA has been set aside in favor of a pure JPA approach. This is analogous to the Hibernate contextual sessions that we used in section 5.4.3.

Since pure JPA is favored over template-based JPA, we'll focus on building Spring-free JPA DAOs in this section. Specifically, `JpaSpitterDao` in the following listing shows how to develop a JPA DAO without resorting to using Spring's `JpaTemplate`.

Listing 5.8 A pure JPA DAO doesn't use any Spring templates.

```
package com.habuma.spitter.persistence;
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import org.springframework.dao.DataAccessException;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import com.habuma.spitter.domain.Spitter;
import com.habuma.spitter.domain.Spittle;

@Repository("spitterDao")
@Transactional
public class JpaSpitterDao implements SpitterDao {
  private static final String RECENT_SPITTLES =
      "SELECT s FROM Spittle s";
  private static final String ALL_SPITTERS =
      "SELECT s FROM Spitter s";
  private static final String SPITTER_FOR_USERNAME =
      "SELECT s FROM Spitter s WHERE s.username = :username";
  private static final String SPITTLES_BY_USERNAME =
      "SELECT s FROM Spittle s WHERE s.spitter.username = :username";

  @PersistenceContext
  private EntityManager em;

  public void addSpitter(Spitter spitter) {
    em.persist(spitter);
  }

  public Spitter getSpitterById(long id) {
    return em.find(Spitter.class, id);
  }

  public void saveSpitter(Spitter spitter) {
    em.merge(spitter);
  }
...
}
```

Inject EntityManager

Use EntityManager

`JpaSpitterDao` uses a `EntityManager` to handle persistence. By working with a `EntityManager`, the DAO remains pure and resembles how a similar DAO may appear in a non-Spring application. But where does it get the `EntityManager`?

Note that the `em` property is annotated with `@PersistentContext`. Put plainly, that annotation indicates that an instance of `EntityManager` should be injected into `em`. To enable `EntityManager` injection in Spring, we'll need to configure a `PersistenceAnnotationBeanPostProcessor` in Spring's application context:

```
<bean class="org.springframework.orm.jpa.support.
  ➥PersistenceAnnotationBeanPostProcessor"/>
```

You may have also noticed that `JpaSpitterDao` is annotated with `@Repository` and `@Transactional`. `@Transactional` indicates that the persistence methods in this DAO will be involved in a transactional context. We'll talk more about `@Transactional` in the next chapter when we cover Spring's support for declarative transactions.

As for `@Repository`, it serves the same purpose here as it did when we developed the Hibernate contextual session version of the DAO. Without a template to handle exception translation, we need to annotate our DAO with `@Repository` so that `PersistenceExceptionTranslationPostProcessor` will know that this is one of those beans for whom exceptions should be translated into one of Spring's unified data access exceptions.

Speaking of `PersistenceExceptionTranslationPostProcessor`, we'll need to remember to wire it up as a bean in Spring just as we did for the Hibernate example:

```
<bean class="org.springframework.dao.annotation.
  ➥PersistenceExceptionTranslationPostProcessor"/>
```

Note that exception translation, whether it be with JPA or Hibernate, isn't mandatory. If you'd prefer that your DAO throw JPA-specific or Hibernate-specific exceptions, then you're welcome to forgo `PersistenceExceptionTranslationPostProcessor` and let the native exceptions flow freely. But if you do use Spring's exception translation, you'll be unifying all of your data access exceptions under Spring's exception hierarchy, which will make it easier to swap out persistence mechanisms later.

5.6 Summary

Data is the life blood of an application. Some of the data-centric among us may even contend that data *is* the application. With such significance being placed on data, it's important that we develop the data access portion of our applications in a way that's robust, simple, and clear.

Spring's support for JDBC and ORM frameworks takes the drudgery out of data access by handling common boilerplate code that exists in all persistence mechanisms, leaving you to focus on the specifics of data access as they pertain to your application.

One way that Spring simplifies data access is by managing the lifecycle of database connections and ORM framework sessions, ensuring that they're opened and closed as

necessary. In this way, management of persistence mechanisms is virtually transparent to your application code.

Also, Spring can catch framework-specific exceptions (some of which are checked exceptions) and convert them to one of a hierarchy of unchecked exceptions that are consistent among all persistence frameworks supported by Spring. This includes converting nebulous `SQLException`s thrown by JDBC into meaningful exceptions that describe the actual problem that led to the exception being thrown.

In this chapter, we saw how to build the persistence layer of a Spring application using JDBC, Hibernate, or JPA. Which you choose is largely a matter of taste, but because we developed our persistence layer behind a common Java interface, the rest of our application can remain unaware of how data is ferried to and from the database.

Transaction management is another aspect of data access that Spring can make simple and transparent. In the next chapter, we'll explore how to use Spring AOP for declarative transaction management.