Spring Data with JPA

Hitting the database with Spring and JDBC

Spring´s data-access philosophy

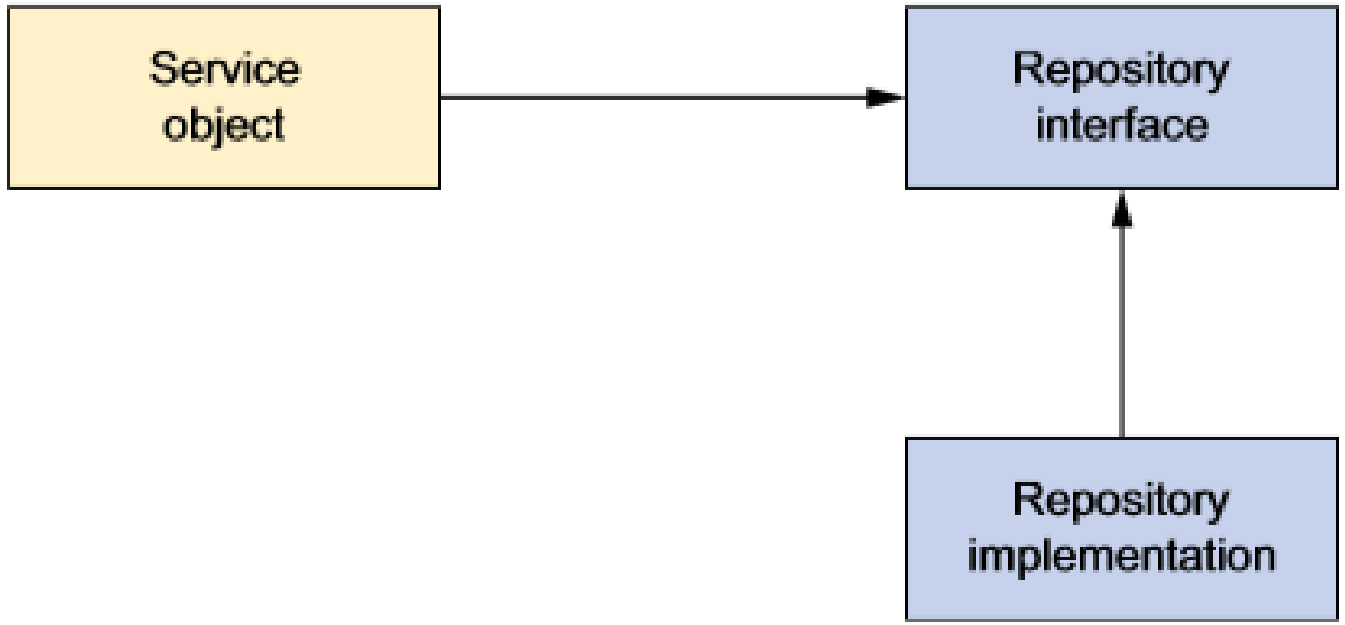

Figure 10.1 Service objects don't handle their own data access. Instead, they delegate data access to repositories. The repository's interface keeps it loosely coupled to the service object.

JDBC's exceptions	Spring's data-access exceptions
BatchUpdateException	BadSqlGrammarException
DataTruncation	CannotAcquireLockException
SQLException	CannotSerializeTransactionException
SQLWarning	CannotGetJdbcConnectionException
	CleanupFailureDataAccessException
	ConcurrencyFailureException
	DataAccessException
	DataAccessResourceFailureException
	DataIntegrityViolationException
	DataRetrievalFailureException
	DataSourceLookupApiUsageException
	DeadlockLoserDataAccessException
	DuplicateKeyException
	EmptyResultDataAccessException
	IncorrectResultSizeDataAccessException
	IncorrectUpdateSemanticsDataAccessException
	InvalidDataAccessApiUsageException
	InvalidDataAccessResourceUsageException
	InvalidResultSetAccessException
	JdbcUpdateAffectedIncorrectNumberOfRowsException
	LobRetrievalFailureException
	NonTransientDataAccessResourceException
	OptimisticLockingFailureException
	PermissionDeniedDataAccessException
	PessimisticLockingFailureException
	QueryTimeoutException
	RecoverableDataAccessException
	SQLWarningException
	SqlXmlFeatureNotImplementedException
	TransientDataAccessException
	TransientDataAccessResourceException
	TypeMismatchDataAccessException
	UncategorizedDataAccessException

Templating data access

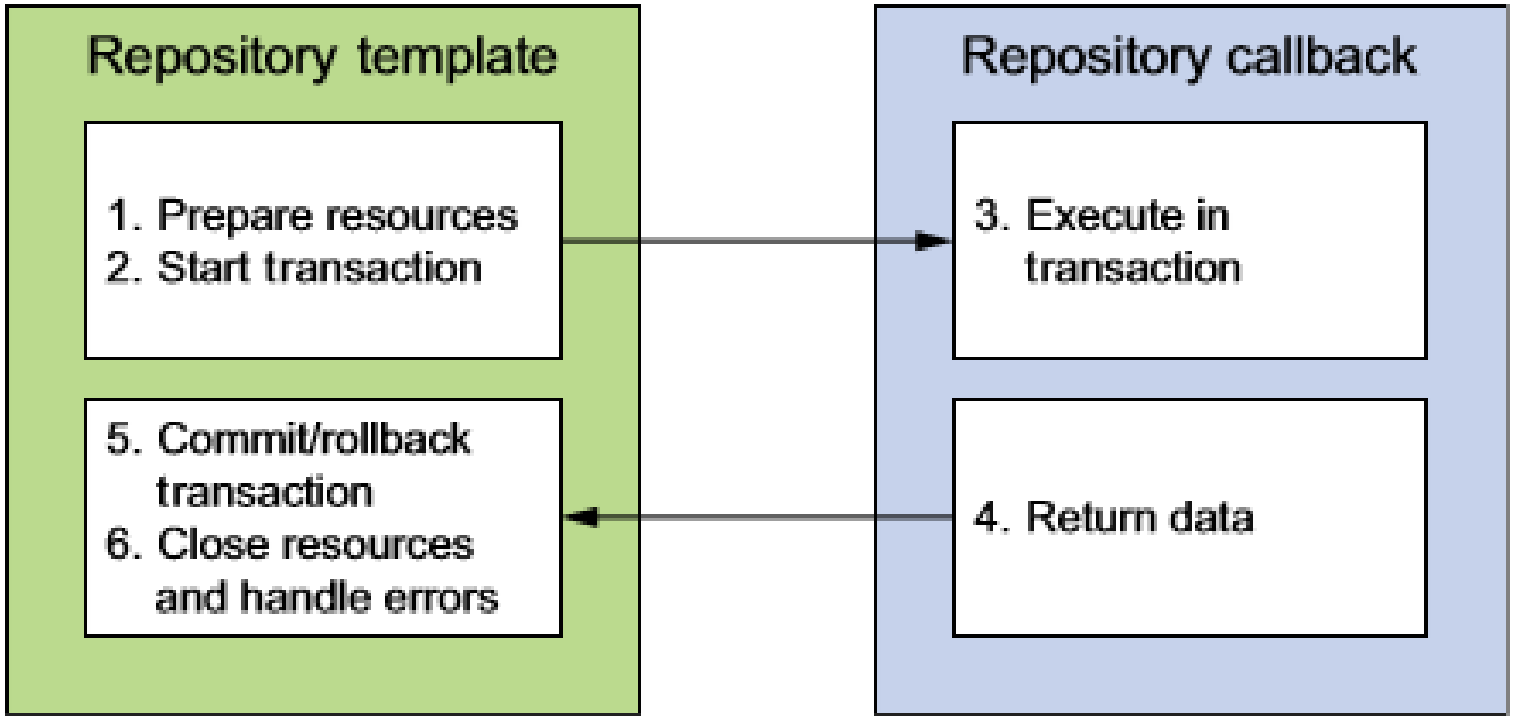

Figure 10.2 Spring's data-access template classes take responsibility for common data-access duties. For application-specific tasks, it calls back into a custom callback object.

Templating data access

Table 10.2 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 (deprecated in Spring 3.1)
`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

Configure a data source

Regardless of which form of Spring-supported data access 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 these:

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

Using JNDI data sources

```
<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 in a Java application server, 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/`.

Using JNDI data sources

Alternatively, if you're using Java configuration, you can use `JndiObjectFactoryBean` to look up the `DataSource` from JNDI:

```
@Bean
public JndiObjectFactoryBean dataSource() {
  JndiObjectFactoryBean jndiObjectFB = new JndiObjectFactoryBean();
  jndiObjectFB.setJndiName("jdbc/SpittrDS");
  jndiObjectFB.setResourceRef(true);
  jndiObjectFB.setProxyInterface(javax.sql.DataSource.class);
  return jndiObjectFB;
}
```

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, plenty of suitable ones are available, including the following open source options:

- Apache Commons DBCP (http://jakarta.apache.org/commons/dbcp)
- c3p0 (http://sourceforge.net/projects/c3p0/)
- BoneCP (http://jolbox.com/)

Using a pooled data source

Most of these connection pools can be configured as a data source in Spring in a way that resembles Spring's own `DriverManagerDataSource` or `SingleConnectionDataSource` (which we'll talk about next). For example, here's how you might configure DBCP's `BasicDataSource`:

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
  p:driverClassName="org.h2.Driver"
  p:url="jdbc:h2:tcp://localhost/~/spitter"
  p:username="sa"
  p:password=""
  p:initialSize="5"
  p:maxActive="10" />
```

Using a pooled data source

- Or in Java-Config:

```
@Bean
public BasicDataSource dataSource() {
  BasicDataSource ds = new BasicDataSource();
  ds.setDriverClassName("org.h2.Driver");
  ds.setUrl("jdbc:h2:tcp://localhost/~/spitter");
  ds.setUsername("sa");
  ds.setPassword("");
  ds.setInitialSize(5);
  ds.setMaxActive(10);
  return ds;
}
```

BasicDataSource´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 0, there's no limit.
`maxIdle`	The maximum number of connections that can be idle in the pool without extras being released. If 0, there's no limit.
`maxOpenPreparedStatements`	The maximum number of prepared statements that can be allocated from the statement pool at the same time. If 0, 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.
`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).

Using JDBC driver-based data source

Spring offers three such data-source classes to choose from (all in the `org.springframework.jdbc.datasource` package):

- `DriverManagerDataSource`—Returns a new connection every time a connection is requested. Unlike DBCP's `BasicDataSource`, the connections provided by `DriverManagerDataSource` aren't pooled.
- `SimpleDriverDataSource`—Works much the same as `DriverManagerDataSource` except that it works with the JDBC driver directly to overcome class loading issues that may arise in certain environments, such as in an OSGi container.
- `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.

Using JDBC driver-based data source

Configuring any of these data sources is similar to how you configured DBCP's `BasicDataSource`. For example, here's how you'd configure a `DriverManagerDataSource` bean:

```
@Bean
public DataSource dataSource() {
  DriverManagerDataSource ds = new DriverManagerDataSource();
  ds.setDriverClassName("org.h2.Driver");
  ds.setUrl("jdbc:h2:tcp://localhost/~/spitter");
  ds.setUsername("sa");
  ds.setPassword("");
  return ds;
}
```

Using JDBC driver-based data source

In XML, the DriverManagerDataSource can be configured as follows:

```
<bean id="dataSource"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource"
  p:driverClassName="org.h2.Driver"
  p:url="jdbc:h2:tcp://localhost/~/spitter"
  p:username="sa"
  p:password="" />
```

Using an embedded data source

Listing 10.1 Configuring an embedded database using the jdbc namespace

```
<?xml version="1.0" encoding="UTF-
     8"?> <beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:c="http://www.springframework.org/schema/c"
    xsi:schemaLocation="http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
...

  <jdbc:embedded-
     database id="dataSource" type="H2">     <jdbc:script location="com/habum
     a/spitter/db/jdbc/schema.sql"/>     <jdbc:script location="com/habuma/sp
     itter/db/jdbc/test-data.sql"/>   </jdbc:embedded-database>
...

</beans>
```

Using an embedded data source

When you configure an embedded database in Java configuration, there isn't the convenience of the `jdbc` namespace. Instead, you can use `EmbeddedDatabaseBuilder` to construct the `DataSource`:

```
@Bean
public DataSource dataSource() {
  return new EmbeddedDatabaseBuilder()
       .setType(EmbeddedDatabaseType.H2)
       .addScript("classpath:schema.sql")
       .addScript("classpath:test-data.sql")
       .build();
}
```

Using profiles to select a data source

Listing 10.2 Spring profiles enabling selection of a data source at runtime

```
package com.habuma.spittr.config;
import org.apache.commons.dbcp.BasicDataSource;
import javax.sql.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import
  org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import
  org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
import org.springframework.jndi.JndiObjectFactoryBean;

@Configuration
public class DataSourceConfiguration {

  @Profile("development")
  @Bean
  public DataSource embeddedDataSource() {
    return new EmbeddedDatabaseBuilder()
                .setType(EmbeddedDatabaseType.H2)
                .addScript("classpath:schema.sql")
                .addScript("classpath:test-data.sql")
                .build();
  }
```

Development data source

```
@Profile("qa")                                  <-- QA data source
@Bean
public DataSource Data() {
  BasicDataSource ds = new BasicDataSource();
  ds.setDriverClassName("org.h2.Driver");
  ds.setUrl("jdbc:h2:tcp://localhost/~/spitter");
  ds.setUsername("sa");
  ds.setPassword("");
  ds.setInitialSize(5);
  ds.setMaxActive(10);
  return ds;
}

@Profile("production")                          <-- Production data source
@Bean
public DataSource dataSource() {
  JndiObjectFactoryBean jndiObjectFactoryBean
                                    = new JndiObjectFactoryBean();
  jndiObjectFactoryBean.setJndiName("jdbc/SpittrDS");
  jndiObjectFactoryBean.setResourceRef(true);
  jndiObjectFactoryBean.setProxyInterface(javax.sql.DataSource.class);
  return (DataSource) jndiObjectFactoryBean.getObject();
}
}
```

Using profiles to select a data source

Listing 10.3 Configuring profile-selected data sources in XML

```
<?xml version="1.0" encoding="UTF-
     8"?> <beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/jdbc
      http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd
    http://www.springframework.org/schema/jee
    http://www.springframework.org/schema/jee/spring-jee-3.1.xsd
    http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
  <beans profile="development">
     <jdbc:embedded-
      database id="dataSource" type="H2">       <jdbc:script location="com/hab
      uma/spitter/db/jdbc/schema.sql"/>       <jdbc:script location="com/habum
      a/spitter/db/jdbc/test-data.sql"/>     </jdbc:embedded-
      database>   </beans>
    <beans profile="qa">                              <-- QA data source
      <bean id="dataSource"
            class="org.apache.commons.dbcp.BasicDataSource"
       p:driverClassName="org.h2.Driver"
       p:url="jdbc:h2:tcp://localhost/~/spitter"
       p:username="sa"
       p:password=""
       p:initialSize="5"
       p:maxActive="10" />   </beans>
    <beans profile="production">                          <-- Production data source
      <jee:jndi-lookup id="dataSource"
                       jndi-name="/jdbc/SpitterDS"
                       resource-ref="true" />   </beans>
```

Development data source

CIIT

Using JDBC with Spring

Listing 10.5 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();
    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) {
```

Get connection

Create statement

Bind parameters

Execute statement

Handle exceptions (somehow)

Clean up

Using JDBC with Spring

```
private static final String SQL_SELECT_SPITTER =
      "select id, username, fullname from spitter where id = ?";
public Spitter findOne(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"));
      spitter.setPassword(rs.getString("password"));
      spitter.setFullName(rs.getString("fullname"));
    }
    return spitter;
```

Get connection

Create statement

Execute query

Bind parameters

Process results

Using JDBC with Spring

```
    } 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

JDBC templates

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 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.

Inserting data using JDBC template

```
@Bean
public JdbcTemplate jdbcTemplate(DataSource dataSource) {
  return new JdbcTemplate(dataSource);
}
```

```
@Repository
public class JdbcSpitterRepository implements SpitterRepository {

  private JdbcOperations jdbcOperations;

  @Inject
  public JdbcSpitterRepository(JdbcOperations jdbcOperations) {
    this.jdbcOperations = jdbcOperations;
  }

...

}
```

Inserting data using JDBC template

As an alternative to component-scanning and autowiring, you could explicitly declare `JdbcSpitterRepository` as a bean in Spring, like this:

```
@Bean
public SpitterRepository spitterRepository(JdbcTemplate jdbcTemplate) {
  return new JdbcSpitterRepository(jdbcTemplate);
}
```

With a `JdbcTemplate` at your repository's disposal, you can greatly simplify the `addSpitter()` method from listing 10.4. The new `JdbcTemplate`-based `addSpitter()` method is as follows.

Listing 10.7 JdbcTemplate-based addSpitter() method

```
public void addSpitter(Spitter spitter) {
    jdbcOperations.update(INSERT_SPITTER,                    <-- Insert Spitter
        spitter.getUsername(),
        spitter.getPassword(),
        spitter.getFullName(),
        spitter.getEmail(),
        spitter.isUpdateByEmail());
}
```

Reading data with JDBC template

Listing 10.8 Querying for a `Spitter` using `JdbcTemplate`

```
public Spitter findOne(long id) {
    return jdbcOperations.queryForObject(                    <-- Query for Spitter
            SELECT_SPITTER_BY_ID, new SpitterRowMapper(),
            id                          <-- Map results to object
            );
}

...

private static final class SpitterRowMapper
                      implements RowMapper<Spitter> {
  public Spitter mapRow(ResultSet rs, int rowNum)
                                    throws SQLException {
    return new Spitter(                      <-- Bind parameters
        rs.getLong("id"),
        rs.getString("username"),
        rs.getString("password"),
        rs.getString("fullName"),
        rs.getString("email"),
        rs.getBoolean("updateByEmail"));
  }
}
```

Using Java 8 Lambdas

For example, the `findOne()` method in listing 10.8 can be rewritten using Java 8 lambdas like this:

```
public Spitter findOne(long id) {
  return jdbcOperations.queryForObject(
    SELECT_SPITTER_BY_ID,
    (rs, rowNum) -&gt; {
      return new Spitter(
        rs.getLong("id"),
        rs.getString("username"),
        rs.getString("password"),
        rs.getString("fullName"),
        rs.getString("email"),
        rs.getBoolean("updateByEmail"));
    },
    id);
}
```

Using Java 8 Lambdas

Alternatively, you can use Java 8 method references to define the mapping in a separate method:

```
public Spitter findOne(long id) {
  return jdbcOperations.queryForObject(
    SELECT_SPITTER_BY_ID, this::mapSpitter, id);
}

private Spitter mapSpitter(ResultSet rs, int row) throws SQLException {
    return new Spitter(
      rs.getLong("id"),
      rs.getString("username"),
      rs.getString("password"),
      rs.getString("fullName"),
      rs.getString("email"),
      rs.getBoolean("updateByEmail"));
}
```

Using named parameters

```
@Bean
public NamedParameterJdbcTemplate jdbcTemplate(DataSource dataSource) {
  return new NamedParameterJdbcTemplate(dataSource);
}
```

Listing 10.9 Using named parameters with Spring JDBC templates

```
private static final String INSERT_SPITTER =
        "insert into Spitter " +
        "    (username, password, fullname, email, updateByEmail) " +
        "values " +
        "    (:username, :password, :fullname, :email, :updateByEmail)";

public void addSpitter(Spitter spitter) {
    Map<String, Object> paramMap = new HashMap<String, Object>();
    paramMap.put("username", spitter.getUsername());      <-- Bind parameters
    paramMap.put("password", spitter.getPassword());
    paramMap.put("fullname", spitter.getFullName());
    paramMap.put("email", spitter.getEmail());
    paramMap.put("updateByEmail", spitter.isUpdateByEmail());

    jdbcOperations.update(INSERT_SPITTER, paramMap);      <-- Perform insert
}
```

Persisting data with object-relational mapping (ORM)

Used to create features that are more sophisticated such as:

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

Hibernate

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

Spring and the Java Persistence API

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 in a Java EE container.

Spring and the Java Persistence API

- *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.

Spring and the Java Persistence API

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

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 in the classpath.

The purpose of the persistence.xml file is to define one or more persistence units.

Configuring application-managed JPA

Here's a typical example of a persistence.xml file as it pertains to the Spittr application:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
      version="1.0">
    <persistence-unit name="spitterPU">
      <class>com.habuma.spittr.domain.Spitter</class>
      <class>com.habuma.spittr.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>
```

Configuring application-managed JPA

The following <bean> declares a LocalEntityManagerFactoryBean in Spring:

```
@Bean
public LocalEntityManagerFactoryBean entityManagerFactoryBean() {
  LocalEntityManagerFactoryBean emfb
      = new LocalEntityManagerFactoryBean();
  emfb.setPersistenceUnitName("spitterPU");
  return emfb;
}
```

Configuring container-managed JPA

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
public LocalContainerEntityManagerFactoryBean entityManagerFactory(
        DataSource dataSource, JpaVendorAdapter jpaVendorAdapter) {
  LocalContainerEntityManagerFactoryBean emfb =
      new LocalContainerEntityManagerFactoryBean();
  emfb.setDataSource(dataSource);
  emfb.setJpaVendorAdapter(jpaVendorAdapter);
  return emfb;
}
```

Configuring container-managed JPA

Spring comes with a handful of JPA vendor adapters to choose from:

- `EclipseLinkJpaVendorAdapter`
- `HibernateJpaVendorAdapter`
- `OpenJpaVendorAdapter`
- `TopLinkJpaVendorAdapter` (deprecated in Spring 3.1)

The Hibernate JPA vendor adapter

In this case, you're using Hibernate as a JPA implementation, so you configure it with a `HibernateJpaVendorAdapter`:

```
@Bean
public JpaVendorAdapter jpaVendorAdapter() {
  HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
  adapter.setDatabase("HSQL");
  adapter.setShowSql(true);
  adapter.setGenerateDdl(false);
  adapter.setDatabasePlatform("org.hibernate.dialect.HSQLDialect");
  return adapter;
}
```

The Hibernate JPA vendor adapter

Table 11.1 The Hibernate JPA vendor adapter supports several databases. You can specify which database to use by setting its `database` property.

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`

The Hibernate JPA vendor adapter

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory(
        DataSource dataSource, JpaVendorAdapter jpaVendorAdapter) {
  LocalContainerEntityManagerFactoryBean emfb =
      new LocalContainerEntityManagerFactoryBean();
  emfb.setDataSource(dataSource);
  emfb.setJpaVendorAdapter(jpaVendorAdapter);
  emfb.setPackagesToScan("com.habuma.spittr.domain");
  return emfb;
}
```

Pulling an EntityManagerFactory from JNDI

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

You can also configure the EntityManagerFactory bean with Java configuration by using

```
@Bean
public JndiObjectFactoryBean entityManagerFactory() {}
JndiObjectFactoryBean jndiObjectFB = new JndiObjectFactoryBean();
  jndiObjectFB.setJndiName("jdbc/SpittrDS");
  return jndiObjectFB;
}
```

→ returns in an EntityManagerFactory bean

Writing a JPA-based repository

```
@Repository
@Transactional
public class JpaSpitterRepository implements SpitterRepository {

  @PersistenceUnit
  private EntityManagerFactory emf;

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

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

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

Inject EntityManagerFactory

Create and use EntityManager

Writing a JPA-based repository

```
@Repository
@Transactional
public class JpaSpitterRepository implements SpitterRepository {

  @PersistenceContext
  private EntityManager em;                          <-- Inject EntityManager

  public void addSpitter(Spitter spitter) {
    em.persist(spitter);                             <-- Use EntityManager
  }

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

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

Writing a JPA-based repository

It's important to understand that `@PersistenceUnit` and `@PersistenceContext` aren't Spring annotations; they're provided by the JPA specification. In order for Spring to understand them and inject an `EntityManagerFactory` or `EntityManager`, Spring's `PersistenceAnnotationBeanPostProcessor` must be configured. If you're already using `<context:annotation-config>` or `<context:component-scan>`, then you're good to go because those configuration elements automatically register a `PersistenceAnnotationBeanPostProcessor` bean. Otherwise, you'll need to register that bean explicitly:

```
@Bean
public PersistenceAnnotationBeanPostProcessor paPostProcessor() {
  return new PersistenceAnnotationBeanPostProcessor();
}

   }
...
}
```

Methods contained in the JPA-repository

Modifier and Type	Method and Description
`void`	`deleteAllInBatch()` Deletes all entities in a batch call.
`void`	`deleteInBatch(Iterable<T> entities)` Deletes the given entities in a batch which means it will create a single `Query`.
`List<T>`	`findAll()`
`<S extends T>` `List<S>`	`findAll(org.springframework.data.domain.Example<S> example)`
`<S extends T>` `List<S>`	`findAll(org.springframework.data.domain.Example<S> example,` `org.springframework.data.domain.Sort sort)`

Methods contained in the JPA-repository

Modifier and Type	Method and Description
`List<T>`	`findAll(org.springframework.data.domain.Sort sort)`
`List<T>`	`findAllById(Iterable<ID> ids)`
`void`	`flush()` Flushes all pending changes to the database.
`T`	`getOne(ID id)` Returns a reference to the entity with the given identifier.
`<S extends T>` `List<S>`	`saveAll(Iterable<S> entities)`
`<S extends T>` `S`	`saveAndFlush(S entity)` Saves an entity and flushes changes instantly.

Methods contained in the JPA-repository

Methods inherited from interface org.springframework.data.repository.PagingAndSortingRepository

`findAll`

Methods inherited from interface org.springframework.data.repository.CrudRepository

`count, delete, deleteAll, deleteAll, deleteById, existsById, findById, save`

Methods inherited from interface org.springframework.data.repository.query.QueryByExampleExecutor

`count, exists, findAll, findOne`

Automatic JPA repository with Spring Data

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

Listing 11.4 Creating a repository from an interface definition with Spring Data

```
public interface SpitterRepository
         extends JpaRepository<Spitter, Long> {
}
```

Automatic JPA repository with Spring Data

Listing 11.5 Configuring Spring Data JPA

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jpa="http://www.springframework.org/schema/data/jpa"
  xsi:schemaLocation="http://www.springframework.org/schema/data/jpa
    http://www.springframework.org/schema/data/jpa/spring-jpa-1.0.xsd">

  <jpa:repositories base-package="com.habuma.spittr.db" />

  ...

</beans>
```

Automatic JPA repository with Spring Data

- In Java-Configuration:

```
@Configuration
@EnableJpaRepositories(basePackages="com.habuma.spittr.db")
public class JpaConfiguration {
   ...
}
```

Defining query methods

One thing your `SpitterRepository` will need is a means of looking up a `Spitter` object given a username. For example, let's say you modify the `SpitterRepository` interface to look like this:

```
public interface SpitterRepository
        extends JpaRepository<Spitter, Long> {
    Spitter findByUsername(String username);
}
```

Defining query methods

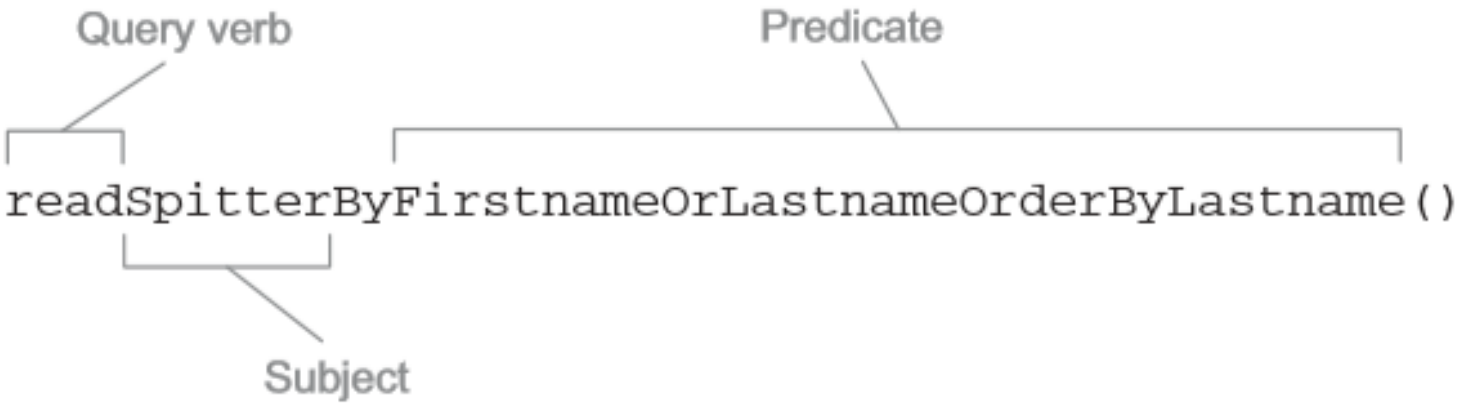

Figure 11.1 Repository methods are named following a pattern that helps Spring Data generate queries against the database.

Defining query methods

- Supported comparison operations:

- IsAfter, After, IsGreaterThan, GreaterThan
- IsGreaterThanEqual, GreaterThanEqual
- IsBefore, Before, IsLessThan, LessThan
- IsLessThanEqual, LessThanEqual
- IsBetween, Between
- IsNull, Null
- IsNotNull, NotNull
- IsIn, In
- IsNotIn, NotIn
- IsStartingWith, StartingWith, StartsWith
- IsEndingWith, EndingWith, EndsWith
- IsContaining, Containing, Contains
- IsLike, Like
- IsNotLike, NotLike
- IsTrue, True
- IsFalse, False
- Is, Equals
- IsNot, Not

Defining query methods

- More examples:

```
List<Spitter> readByFirstnameIgnoringCaseOrLastnameIgnoresCase(
                    String first, String last);
```

```
List<Spitter> readByFirstnameOrLastnameOrderByLastnameAsc(
                        String first, String last);
```

```
List<Spitter> readByFirstnameOrLastnameOrderByLastnameAscFirstnameDesc(
                    String first, String last);
```

- `List<Pet> findPetsByBreedIn(List<String> breed)`
- `int countProductsByDiscontinuedTrue()`
- `List<Order> findByShippingDateBetween(Date start, Date end)`

Declaring custom queries

In situations where the desired data can't be adequately expressed in the method name, you can use the `@Query` annotation to provide Spring Data with the query that should be performed. For the `findAllGmailSpitters()` method, you might use `@Query` like this:

```
@Query("select s from Spitter s where s.email like '%gmail.com'")
List<Spitter> findAllGmailSpitters();
```

Mixing in custom functionality

Listing 11.6 Repository that promotes frequent `Spitter` users to Elite status

```
public class SpitterRepositoryImpl implements SpitterSweeper {

  @PersistenceContext
  private EntityManager em;

  public int eliteSweep() {
    String update =
        "UPDATE Spitter spitter " +
        "SET spitter.status = 'Elite' " +
        "WHERE spitter.status = 'Newbie' " +
        "AND spitter.id IN (" +
        "SELECT s FROM Spitter s WHERE (" +
        "  SELECT COUNT(spittles) FROM s.spittles spittles) > 10000" +
        ")";
    return em.createQuery(update).executeUpdate();
  }

}
```

Mixing in custom functionality

```
public interface SpitterSweeper{
    int eliteSweep();
}
```

You should also make sure the `eliteSweep()` method is declared in the `SpitterRepository` interface. The easy way to do that and avoid duplicating code is to change `SpitterRepository` so that it extends `SpitterSweeper`:

```
public interface SpitterRepository
        extends JpaRepository<Spitter, Long>,
                SpitterSweeper {
    ....
}
```

Mixing in custom functionality

If you'd prefer to use some other postfix, you need to specify it when configuring `@EnableJpaRepositories` by setting the `repositoryImplementationPostfix` attribute:

```
@EnableJpaRepositories(
  basePackages="com.habuma.spittr.db",
  repositoryImplementationPostfix="Helper")
```

Or in xml:

```
<jpa:repositories base-package="com.habuma.spittr.db"
    repository-impl-postfix="Helper" />
```

Spring Data with JPA