Introduction to Spring 2 and JPA

Explore the Spring 2 framework and the Java Persistence API with Eclipse and DB2 Express-C

Skill Level: Introductory

Sing Li (westmakaha@yahoo.com)
Author
Wrox Press

08 Aug 2006

Java™ server applications need not be difficult and tedious to create. Now in its second generation, the lightweight Spring framework adds a large suite of features that make it simple for even new server application developers to use. One key enhancement is Spring 2's integration with the Java Persistence API (JPA), a cornerstone of the Enterprise JavaBeans (EJB) 3.0 specification. In this tutorial, learn how to create server applications from scratch using the Spring 2 framework.

Section 1. Before you start

For almost a decade, the "proper" way to build a robust and maintainable server-side Java application has been the exclusive domain of the Java 2 Enterprise Edition (J2EE) platform. J2EE applications are built using Enterprise JavaBeans (EJB) technology and run on servers that facilitate deployment and provide rich container services (such as the management of database connections and pooling). These servers also add value by providing deploy-time declarative control of important features such as security and transactions. Although versatile, the J2EE development process involves many tedious and repetitive tasks and the creation and maintenance of large numbers of source code files.

Many lightweight Java frameworks claim to simplify server application development, but none matches the Spring framework in maturity and popularity (see Resources). Now in version 2, Spring was designed from day one to simplify the server application building process. Instead of approaching development from an all-in-one container perspective, Spring aims to provide just enough support for an application's requirements without the burden of a full-fledged container
environment. Spring eliminates code bloat: you can code and test business objects completely outside of any container, letting your business-object code remain simple, testable, maintainable, and reusable.

With the arrival of Java EE 5 and EJB 3.0, the J2EE community is poised to meet the Spring developer community. EJB 3.0 supports the notion of lightweight POJOs (Plain Old Java Objects) as EJB components and introduces the Java Persistence API (JPA), a persistence mechanism that can run externally to the container. This persistence mechanism automates the movement of information between business objects and external relational databases. Version 2 of the Spring framework has continued its evolution and also leverages JPA as a persistence mechanism.

In this tutorial, you will work with Spring 2 and JPA persistence. You'll create a server application using the Spring 2 framework, complete with access to a DB2 Express-C database. The Eclipse IDE facilitates the development of the Java application and enhances your exploration of the Spring 2 framework.

About this tutorial

This tutorial aims to guide you through the use and application of the Spring 2 framework in the shortest possible time using a pure and simple learn-by-coding approach. You will build a Web application from scratch, step-by-step, assisted by the Spring 2 framework.

This tutorial does not attempt to cover all of Spring 2's features and options. Instead, it focuses narrowly on one proven approach to server application development using Spring. You are encouraged to consult other Spring 2 resources for more advanced applications and techniques related to the framework (see Resources).

You will proceed through a complete "from concept to application" cycle, including:

- Performing domain analysis
- Coding business objects and services
- Unit testing business objects
- Adding data access code painlessly to business objects using Spring JPA
- Implementing your services using Spring DAO (data access object)
- Coding integration tests for your services against DB2® Express-C
- Creating controllers for a Spring Model-View-Controller (MVC) based user interface
- Designing views for the user interface
- Creating a deployable WAR file of your application
- Configuring and deploying your application on the Apache Tomcat server
By the end of the tutorial, you'll understand how the Spring 2 framework works and how it can help you create highly componentized and maintainable Web applications. You'll gain working experience in building such an application and can apply many of the techniques you learn here to your daily development tasks.

Prerequisites

You should be familiar with basic object-oriented design concepts and Java development using Java SE 5, including generics. Relational database concepts should be familiar to you, and you should have basic knowledge of how to set up a new database in DB2 Express-C.

Familiarity with testing terminology, including unit testing and integration testing, is assumed. Working experience with test frameworks such as JUnit is desirable, but not required.

You should have working experience with Eclipse and be able to create new Java projects, compile Java code, and debug projects within Eclipse.

System requirements

The required hardware configuration for trying out the tools and example in this tutorial is a system with at least 512MB of memory (1GB recommended).

You need the following software installed:

- Sun's JDK 5.0 Update 7 or later or a version of the IBM® Developer Kit for the Java 5 platform.
- Spring framework 2.0. The code in this tutorial has been tested with Spring 2.0 RC 2. Select the "with-dependencies" version of the distribution. This will save you many additional downloads from different sites because it contains many of the required open source project binaries.
- A version of the Tomcat 5.5 server. Choose the ZIP file download for the server and unarchive it into a directory of your choice. This tutorial's code has been tested against Tomcat 5.5.17.
- DB2 Express-C (version 8.2.4 or later).
- Eclipse 3.2 or later, including the Web Tools Platform (WTP) 1.5 or later.
- The JPA reference implementation. Follow the full installation instructions in Writing DAO integration tests against an RDBMS in this tutorial.
Section 2. Overview of Spring 2 framework operations

This section introduces the Spring 2 framework and outlines its advantages over conventional server application development.

Building applications with Spring

Spring is not a programming framework in the classical API sense. In most cases, frameworks consist of APIs and a collection of code skeletons that you can use in your applications.

Spring 2 is designed to be unintrusive. In essence, it lets you code your objects and business logic as if Spring did not exist. After you code and test these objects, you can add the Spring 2 support features. In some cases, adding these features does not even require you to recompile the source code.

For example, you can create and test a Java employee object first and then add Spring 2 support to save instances of the object to a relational database. Or you can write code to update bank accounts first and then apply Spring 2 transaction capability to ensure data integrity.

Figure 1 shows a typical Web-based server application. The user interacts with the application through the user interface. The application logic is carried out as operations on a set of business objects, forming the application's domain model. The business objects are usually backed by data that is stored in a relational database.

Figure 1. Architecture of a typical Web-based application
component within Figure 1 and write custom code to enable access to the relational
database.

When using Spring to build applications, you can focus first on perfecting the domain
model. You can model the objects in the system using simple Java POJOs and
define the services in the system as standard Java interfaces. Doing your design this
way lets you create and test the domain model independently of Spring or other
frameworks/libraries.

You can then apply additional Spring features to your application, on top of your
tested domain model. For example, you can add object persistence -- the ability to
save and fetch data from relational databases -- using Spring's JPA support.

Load-time enhancement

The Spring framework stays unintrusive by providing added value during a class's
load time. When you run a Java application normally, the JVM loads classes as you
need them from disk storage through a set of classloaders. This is quite transparent
and usually happens without you being aware of it. A moderately complex piece of
software, such as the Eclipse IDE, can load thousands of classes under the hood.
With the Spring framework, you program to the framework by specifying to the
Spring engine (also called the Spring container) how you want your classes to fit
together and which features you want added to these classes. The Spring engine
constructs your classes according to your recipe. Figure 2 shows this process
schematically:

**Figure 2. Spring framework operation**
Figure 2 shows that because the Spring engine has low-level access to your classes, it can enhance them with additional features according to your specific configuration. Also shown in Figure 2 is a means by which you can provide instructions to the Spring engine. Typically, these instructions exist as XML beans descriptor files or in some cases, Java 5 annotations in-line with the source code.

For example, you can create simple Java objects that represent employees and then have the Spring engine enhance the classes so that these objects can be automatically stored and fetched from relational database tables. You will see how to do this in this tutorial.

**Classic API support**

In addition to class load-time enhancement, Spring 2 also provides classic API support to encapsulate complex and tedious operations. Figure 3 shows the Spring support libraries that this tutorial's example application uses. Contrast Figure 3 with Figure 1:
You'll use Spring 2's JPA support in conjunction with Spring DAO APIs to simplify access to relational databases. You'll use Spring MVC to add a Web-based user interface easily to the application.

Section 3. Preparing for Spring

In this section, you'll start to build a sample employee-information application that takes advantage of the Spring 2 framework. You'll identify and code the application's business objects, generate setters and getters, code a service interface, and unit-test your classes. You perform this phase of the application's development cycle independent of the Spring framework.

Identifying the POJOs in domain model analysis

The first step in this tutorial's approach to application design is *domain model analysis* -- often called "identifying the POJOs." POJOs, in this case, refers to Plain Old Java Objects. They also represent the *business objects* in the application, and this tutorial uses the two terms interchangeably.

Identifying the business objects in the application domain is the same exercise you performed when you first learned object-oriented design. The goal is to identify the objects and their interactions in the system that you are attempting to create. More specifically, you want to discover:
• Objects that maintain states (and the states you need to maintain)
• Relationships among these objects
• Interactions (if any) among these objects
• Operations that you need to perform on these objects within your application

Take the simple server application that you build in this tutorial. It is a system that displays employee information. As the name implies, an object in this system is an employee in the company. Employee information can be extensive, but in this simple application, you need to know only the following:

• Employee number
• First name
• Middle initial
• Last name
• Phone extension number
• Job
• Education level
• Sex
• Salary
• Bonus
• Commission
• Address
• Date of hire
• Date of birth

Another business object that can be identified in this domain is an object that represents an address. The address is separated from the employee information because other entities in the system (if you were to expand it) might also have an address, and this way you can keep all the addresses together. For simplicity, assume that the address object needs to contain only a street number and street name.

In this simple system, every employee has an address, and every address belongs to only one employee. This is called a one-to-one relationship. Other possible relationships include one-to-many (such as a project-to-employees relationship), many-to-one (such as an employees-in-department relationship), and many-to-many (such as an employee-to-HR-benefits relationship). These other relationships can all be modeled in Spring 2 applications but are beyond this this beginner-level tutorial’s scope.
In this system, the following operations are required on the employee and address objects:

- Create a new employee (and a new address object)
- Remove an employee (and the associated address object)
- Update the information on an employee
- Find all the employees in the company
- Find a specific employee in the company by his or her employee number
- Find employees in the company by last name
- Find employees in the company living on a certain street
- Find employees with a salary greater than a specified amount
- Find employees with commissions greater than a specified amount

Typically, you can identify these operations by thinking about the user interface required for the system and the business logic that you need to implement in the system.

The operations in this example are purposely kept simple to keep the tutorial focused. In a typical production scenario, you are likely to find more complex operations that operate across multiple sets of identified objects, as well as direct interactions among objects.

Once you have the business objects, interactions, and operations identified, you are ready to code and test.

**Coding the business objects**

The first set of objects that you code are the POJOs that make up the system. You can create these objects using your favorite development editor or IDE. The instructions in this tutorial assume that you are using Eclipse.

Create a new Java project in Eclipse called Spring2Tutorial and then a new Java class called `com.ibm.dw.spring2.Employee`, as shown in Listing 1:

```
package com.ibm.dw.spring2;
import java.util.Date;

public class Employee {
    private long empid;
    private String empno;
    private String firstName;
    private String midInitial;
    private String lastName;
    private String phoneNumber;
    private String job;
```

Listing 1. Coding the employee's data fields
private int educationLevel;
private char sex;
private double salary;
private double bonus;
private double commission;
private Address addr;
private Date hiredate;
private Date birthdate;
}

Listing 1 defines all of the information that is kept in an instance of the Employee object. These fields are all defined to be private, so you need to create some getter and setter methods to allow external access to the information.

Generating getters, setters, and constructors with Eclipse

Generate the getter and setter methods in Eclipse. Right-click inside the editor and select Source->Generate Getter and Setters.... In the pop-up dialog, shown in Figure 4, click Select All and then click OK:

Figure 4. Generating getters and setters in Eclipse
Finally, you must create a couple of constructors for the objects. In Eclipse, you can just right-click and select Source-> Generate Constructor. Then you need to edit the generated constructor. Take a look at Listing 2:

Listing 2. Constructors for the Employee POJO

```java
public Employee(String empno, String firstName, String midInitial, String lastName,
                 String birthdate, String bonus, String commission, String educationLevel,
                 String empid, String empno, String firstName, String hiredate, String job,
                 String lastName, String midInitial, String phoneNumber, String salary, String sex,
                 String addr)
```
String phoneNumber, String job, int educationLevel, char sex, double salary,
double bonus, double commission, Address addr, Date hiredate, Date birthdate) {
    this.empno = empno;
    this.firstName = firstName;
    this.midInitial = midInitial;
    this.lastName = lastName;
    this.phoneNumber = phoneNumber;
    this.job = job;
    this.educationLevel = educationLevel;
    this.sex = sex;
    this.salary = salary;
    this.bonus = bonus;
    this.commission = commission;
    this.addr = addr;
    this.hiredate = hiredate;
    this.birthdate = birthdate;
}

public Employee() {}

You need to add the trivial constructor shown in Listing 2 because you use it later in
the unit-test stage. This completes the coding for the Employee business object.

The missing empid field

You may notice that the empid field is missing from the constructor in Listing 2. This
field is used later to contain a JPA-generated primary key associated with an
Employee instance. This key is managed by JPA and should not be modified by the
application. You should now remove the setter method, setEmpid(), from the
source code.

Coding the Address POJO

An Employee object references an Address object. The Address object also
needs to be saved whenever an Employee object is saved to the database.

The code for the Address object is shown in Listing 3, including getters, setters,
and constructors:

Listing 3. Code for the Address POJO

package com.ibm.dw.spring2;

public class Address {
    private long id;
    private int number;
    private String street;

    public long getId() {
        return id;
    }
    public int getNumber() {
        return number;
    }
    public void setNumber(int number) {
        this.number = number;
    }
    public String getStreet() {
        return street;
    }

}
Now that you’ve finished coding the POJOs in the domain model, you need to implement operations on the POJOs in the application in terms of a service interface.

Creating a service interface

From the domain analysis you performed earlier, you have the set of operations on the business objects, required by the application.

Translating the set of requirements to code -- more precisely, to a Java interface -- you end up with something similar to Listing 4. Each of the operations on the Employee object becomes a method in the EmployeeService interface.

Listing 4. The EmployeeService interface

```java
package com.ibm.dw.spring2;
import java.util.List;
public interface EmployeeService {
    // create a new employee
    public Employee save(Employee emp);
    // removing an employee
    public void delete(Employee emp);
    // update the information on an employee
    public Employee update(Employee emp);
    // find all the employees in the company
    public List<Employee> findAll();
    // find an employee by the employee number
    public List<Employee> findByEmployeeNumber(String empno);
    // find an employee by his name
    public List<Employee> findByEmployeeLastName(String lastName);
    // find an employees living on a street
    public List<Employee> findByAddressStreetName(String streetName);
    // find an employee by the internal unique id
    public Employee findById(long id);
    // find employee over a certain salary
    public List<Employee> findEmployeeWithSalaryOver(double sal);
    // find employee with a certain commission income
    public List<Employee> findEmployeeWithCommissionOver(double comm);
}
```
Note that in Listing 4, the code is a literally direct translation of the required operations list from the domain analysis. This interface contains no Spring-specific coding. You know what you need to do with the objects in this interface but so far, you don't have a clue how to do it. You'll see how to implement the methods in this interface very shortly, but first: POJO unit tests.

Testing business objects outside of any container

After coding the independent POJOs and services in your domain model, you can write unit tests for the POJOs. Up to this point, you still haven't performed any Spring-specific steps. In fact, you could use the tested POJOs that you end up with in other applications (or other parts of the same application) without change.

Listing 5 shows POJOUnitTest.java. This is a JUnit test case that tests the Employee and Address POJOs.

**Listing 5. Independent unit test for employee and address POJOs**

```java
package com.ibm.dw.spring2;
import java.text.SimpleDateFormat;
import java.util.Date;
import junit.framework.TestCase;
public class POJOUnitTest extends TestCase {
    private Employee empl, emp2;
    private Address addr1, addr2;

    protected void setUp() throws Exception {
        addr1 = new Address(10, "Walker Street");
        addr2 = new Address();
        addr2.setNumber(20);
        addr2.setStreet("Walker Street");
        emp1 = new Employee("0001", "Joe", "R","Smith","4853","Engineer", 3, 'M',
                20000.00, 0.00, 0.00, addr1,
                makeDate("08/08/2006") , makeDate("02/04/1972"));

        emp2 = new Employee();
        emp2.setEmpno("0002");
        emp2.setFirstName("John");
        emp2.setMidInitial("T");
        emp2.setLastName("Lockheed");
        emp2.setPhoneNumber("4333");
        emp2.setAddr(addr2);
        emp2.setJob("Sales");
        emp2.setHiredate(makeDate("01/01/2005"));
        emp2.setBirthdate(makeDate("10/8/1966"));
    }

    public void testEmployee() throws java.text.ParseException {
        assertEquals("0001", emp1.getEmpno());
        assertEquals("Joe", emp1.getFirstName());
        assertEquals("R", emp1.getMidInitial());
        assertEquals("Smith", emp1.getLastName());
    }
}
```

A test-focused design culture

When you study the available Spring literature, you'll discover a number of methodologies and philosophies applicable to Spring-based system development. One common element across all of these methodologies is a test-centric culture. Because Spring lets you test the domain model POJOs rapidly outside of the container, frequent unit testing (and integration testing) can become a cornerstone of the design process.

The testing code in Listing 5 should be straightforward to follow. This test sets the fields of Employee and Address using first the constructor, then on a per-field basis, and finally verifying the value. These tests do not provide complete coverage; not every field is tested, but Listing 5 does illustrate how you can unit test the model outside of any cumbersome container.

To run the unit test inside Eclipse, right-click the POJOUnitTest.java file in the Navigator view and then select Run as... -> JUnit Test.

POJO-centric development in Spring lets you test business objects in isolation, outside of any container. Unlike other approaches, you can execute the unit tests from POJO-centric design very quickly, and as a result, you can execute them frequently (for example, as part of the project build procedure).

Section 4. Database access via Spring 2's JPA support

In this section, you add database access to the Employee and Address POJOs by leveraging Spring 2's support for the Java Persistence API (JPA).
Object persistence through JPA ORM

To provide an implementation for the service interface that you have defined, you need to draw on some of Spring 2's features and support libraries. Alternatives exist, of course. You could start implementing the interface, method by method, using standard Java database access technology such as JDBC. However, after you see how Spring 2 does this with JPA, you'll appreciate that it is substantially easier to delegate the job to Spring.

With Spring 2, the JPA persistence stack from the EJB 3.0 and Java EE 5 specifications (see Resources) is integrated, making it one of the most simple yet standard ways of enabling database access for Spring.

The Spring framework has always supported persistence through other object-to-relational mapping (ORM) technologies, but such mapping tasks require some fairly tricky and in-depth knowledge of third-party nonstandard persistence libraries. With the arrival of JPA, and a large vendor base supporting the JPA standard, the support for nonstandard third-party persistence libraries may become less important.

Spring 2's JPA support makes the tedious tasks of writing, reading, searching, updating, and deleting objects (POJOs) to and from relational databases transparent. You can continue to use the Java language's object-oriented syntax to work with the POJOs, and the JPA ORM layer takes care of the database table creation, queries, update code, and deletion code.

In addition to transparent database operations, Spring 2's JPA support also converts a potpourri of database-vendor-specific exceptions to a set of well-defined exceptions to make exception-handling code a lot simpler. Figure 5 illustrates the Spring 2 JPA support:

Figure 5. Spring 2 JPA support
Along with Java SE 5 annotations, JPA also supports the specification of mapping hints via an external XML file. This technique is necessary if you are not using Java SE 5 and is beyond this tutorial's scope.

In Figure 5, you feed the Spring engine your objects, together with some hints (metadata) on how you want them mapped to the relational database tables. The Spring JPA support takes care of the rest. You can supply the mapping hints to the JPA engine as Java 5 annotations or as an external XML definition file (for backward compatibility with JDK 1.4).

Because JPA implementations exist for a variety of ORM products and databases, your implementation code can be portable (if necessary) across different vendors' solutions.

Operations on the mapped objects are performed through a JPA entity manager. For example, to write a tree of related objects to a relational database with an entity manager called `em`, the code might be:

```java
em.persists(myObjectTree);
```

The JPA entity manager then examines the mapping hints that you have supplied and walks through `myObjectTree` to store all of the mapped fields from the object tree into the relational database.
As you will see shortly (in Implementing domain services using Spring DAO), Spring goes even further and simplifies the task of working with JPA entity managers.

Providing JPA ORM mapping metadata

To provide hints on how to save the Employee objects to the database, you can add Java SE 5 annotations to the Employee.java source code. These hints are frequently called metadata because they are data that describes data.

Listing 6 shows the annotated version of the Employee object, with annotations highlighted in bold:

**Listing 6. Adding JPA annotations to the Employee POJO**

```java
package com.ibm.dw.spring2;
import java.util.Date;
import javax.persistence.CascadeType;
...
import javax.persistence.TemporalType;

@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private long empid;
    @Column(length = 6)
    private String empno;
    @Column(name = "FIRSTNME")
    private String firstName;
    @Column(name = "MIDINIT")
    private String midInitial;
    private String lastName;
    @Column(name = "PHONENO")
    private String phoneNumber;
    @Column(length = 8)
    private String job;
    @Column(name = "EDLEVEL")
    private int educationLevel;
    @Column(length = 1)
    private char sex;
    @Column(precision=12, scale=2)
    private double salary;
    @Column(precision=12, scale=2)
    private double bonus;
    @Column(name = "COMM", precision=12, scale=2)
    private double commission;
    @OneToOne(cascade = CascadeType.ALL)
    private Address addr;
    @Temporal(TemporalType.DATE)
    private Date hiredate;
```
@Temporal(TemporalType.DATE)
private Date birthdate;
...

All the annotations in this case are on the field level. This is the most frequent use of JPA annotations. You can also annotate the getter method corresponding to a field. This may be necessary in cases where the value you want stored into the database is computed instead of being a field of the object.

JPA annotations for the Employee POJO

Table 1 describes the annotation on each of the fields in Listing 6 and the persistence hint that is given to the Spring 2 engine:

Table 1. JPA annotations for the Employee POJO

<table>
<thead>
<tr>
<th>Field/element</th>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee</td>
<td>@Entity</td>
<td>Indicates this is a class to be saved to the database. The class name is used as the table name by default.</td>
</tr>
<tr>
<td>empid</td>
<td>@Id</td>
<td>Indicates that this is the primary key field for the table.</td>
</tr>
<tr>
<td>empid</td>
<td>@GeneratedValue(strategy = GenerationType.TABLE)</td>
<td>Specifies the strategy to be used by the persistence engine to assign the unique primary key ID. GenerationType.TABLE indicates that a portable, table-based sequencing of unique ID should be used. Other database-specific options are available but may not work across multiple databases.</td>
</tr>
<tr>
<td>empno</td>
<td>@Column(length = 6)</td>
<td>This field contains the employee number, assigned by the company. Note that this is not the primary key. In this application, the primary key is generated and managed by the engine. The @Column() tag specifies that the field should be six characters in length. Specifying field lengths where applicable can help keep down the resulting table size.</td>
</tr>
<tr>
<td>firstName</td>
<td>@Column(name = &quot;FIRSTNME&quot;)</td>
<td>Specifies the field name that will be used for this field in the database table.</td>
</tr>
<tr>
<td>midInitial</td>
<td>@Column(name = &quot;MIDINIT&quot;)</td>
<td>Specifies the field name that will be used for this field in the database table. Note that it is different from the Java field</td>
</tr>
<tr>
<td>Field</td>
<td>Annotation/Description</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>lastName</td>
<td>No annotations, so the field name &quot;LASTNAME&quot; will be used, matching the Java field name.</td>
<td></td>
</tr>
<tr>
<td>phoneNumber</td>
<td>Specifies the field name that will be used in the database table.</td>
<td></td>
</tr>
<tr>
<td>job</td>
<td>Specifies the length of the database field.</td>
<td></td>
</tr>
<tr>
<td>educationLevel</td>
<td>Specifies the database field name.</td>
<td></td>
</tr>
<tr>
<td>sex</td>
<td>Specifies the length of the database field.</td>
<td></td>
</tr>
<tr>
<td>salary</td>
<td>Specifies the decimal precision of the floating point database field.</td>
<td></td>
</tr>
<tr>
<td>bonus</td>
<td>Specifies the decimal precision of the floating point database field.</td>
<td></td>
</tr>
<tr>
<td>commission</td>
<td>Specifies the decimal precision of the floating point database field.</td>
<td></td>
</tr>
<tr>
<td>addr</td>
<td>Specifies the relationship between this table and the Address object in another mapped table. cascade=ALL indicates that add, modify, delete, and refresh should all cascade to the related table.</td>
<td></td>
</tr>
<tr>
<td>hiredate</td>
<td>Specifies the field is a date field (instead of a time or timestamp field).</td>
<td></td>
</tr>
<tr>
<td>birthdate</td>
<td>Specifies the field is a date field (instead of a time or timestamp field).</td>
<td></td>
</tr>
</tbody>
</table>

Note that the Employee and Address instances are related one-to-one (specified by the @OneToOne(cascade=CascadeType.ALL) annotation). This annotation specifies that all entity manager operations on the Employee object should cascade to its related Address object. This means that any addition of an Employee record creates a corresponding Address record in the RDBMS. It also means that any update or deletes on the Employee record also cascades to the related Address record. This is an extension of the familiar cascading delete integrity constraint, often found in RDBMSes. In practice, you will find that Java coding is substantially simplified when you perform cascaded operations: you no longer need to coordinate the operation across multiple tables.

The annotated Employee source code is a blueprint for the JPA entity manager to manage persistent instances of Employee or Address objects. You should find this
annotation significantly simpler than the tedious and error-prone act of writing physical code to create and work with actual RDBMS tables.

The database table for the Employee object, optionally generated based on the above annotation, has a schema similar to Listing 7:

Listing 7. The equivalent database schema for employee POJOs

```
CREATE TABLE EMPLOYEE (  
    EMPID INTEGER NOT NULL,  
    EDLEVEL INTEGER,  
    SEX CHAR(1),  
    FIRSTNAME VARCHAR(255),  
    SALARY DOUBLE,  
    LASTNAME VARCHAR(255),  
    BONUS DOUBLE,  
    JOB VARCHAR(8),  
    COMM DOUBLE,  
    MIDINIT VARCHAR(255),  
    HIREDATE DATE,  
    EMPNO VARCHAR(6),  
    BIRTHDATE DATE,  
    PHONENO VARCHAR(255),  
    ADDR_ID INTEGER,  
    PRIMARY KEY (EMPID),  
    FOREIGN KEY (ADDR_ID)  
) ;
```

Length of string-based fields
Note that any String fields that you have not annotated with @Column(length=?) default to VARCHAR(255). This can represent wasted storage allocation, per row, for short fields. In production scenarios, you'll probably want tighter control over the space allocation of the underlying managed table.

Compare Listing 7 to the annotated Employee class in Listing 6 to see the effect of the annotations on the creation of the table by the Spring engine.

If you want a detailed explanation and description of all the available JPA annotations, consult JSR 220, the Enterprise JavaBeans 3.0 specification Final Release document (see Resources).

JPA annotations for the Address object

The Address POJO is annotated in a similar manner, shown in Listing 8:

Listing 8. JPA annotations for the Address POJO

```
package com.ibm.dw.spring2;
import javax.persistence.Column;
...
@Entity  
public class Address {  
  ...
```
By now, all of the annotations should make sense to you. Not surprisingly, the table that the annotations in Listing 8 generate has the schema shown in Listing 9:

### Listing 9. Equivalent database schema for Address POJO

```sql
CREATE TABLE ADDRESS (
    ID INTEGER NOT NULL,
    NUM INTEGER,
    STNAME VARCHAR(25),
    PRIMARY KEY (ID)
)
```

**Relationship between Spring 2 and Java EE 5**

JPA persistence is part of EJB 3.0, which in turn is part of the Java EE 5 specification, implying that all compatible Java EE 5 servers (commercial, open source, or otherwise) will have compliant implementations. This practically guarantees the availability of robust, high-quality JPA implementations in the near future.

Note that although Spring 2 leverages JPA persistence from the EJB 3.0 specification, users of Spring 2 are under no further obligation to employ any other elements of the EJB 3.0 or Java EE 5 specifications.

JPA was designed from the outset to be usable independently, outside a conventional EJB container. As a concrete example, the application in this tutorial benefits from JPA but is definitely not an EJB 3.0 or a Java EE 5 application.

---

**Section 5. Implementing domain services using Spring DAO**

In this section, you use the Spring DAO (data access object) API to implement the employee information application’s service interface.
Implementing the EmployeeService interface

Once the Spring 2 engine knows how to persist instances of the Employee and Address objects, the task of implementing the EmployeeService interface becomes substantially simpler.

You can draw on the Spring DAO API in your service implementation. Spring DAO implements the well-known DAO design pattern (see Resources). In this pattern, the DAO provides a consistent facade for access to data. Data fetch and modification are performed through a transfer object. The DAO encapsulates the actual data source and provides methods to work with the transfer object.

Architecturally, the DAO APIs shield you from the complexity of working with actual data persistence API calls. (In addition to JPA, Spring also supports other ORM technologies such as JDO, Hibernate, iBATIS SQL Maps, and Apache OJB.) Using Spring's DAO, you can write data access code that can be easily adapted to any of these persistence APIs.

In addition to abstraction from data persistence APIs, Spring's DAO support maps a variety of vendor-specific data access exceptions to a set of well-documented Spring data access exceptions.

The Spring DAO API also provides support classes that you can readily extend. By extending them, you can free yourself from the tedious and error prone task of writing ORM data access code. All of the required coding is encapsulated in the base classes and support library classes, and they are fully tested. These classes encapsulate the connection and transaction-management code that is often found intermixed with application logic. In the case of JPA support classes, the use of a JPA entity manager is completely encapsulated within the support class, freeing you from the concern of working with entity managers and an entity manager factory.

Some real code will convince you of the versatility of Spring's DAO API. Listing 10 is the implementation of the EmployeeService interface, called EmployeeDAO, using the Spring 2's JpaDaoSupport class:

Listing 10. Implementation of the EmployeeService interface using Spring 2's JPA support

```java
import java.util.List;
import org.springframework.orm.jpa.support.JpaDaoSupport;

public class EmployeeDAO extends JpaDaoSupport implements EmployeeService {
    public Employee findById(long id) {
        return getJpaTemplate().find(Employee.class, id);
    }
    public List<Employee> findAll() {
        return getJpaTemplate().find("select e from Employee e");
    }
    public List<Employee> findByEmployeeNumber(String empno) {
        return getJpaTemplate().find("select e from Employee e where e.empno = ?1", empno);
    }
}
```
In Listing 10, the first thing to note is the simplicity of coding in each of the method implementations. The JpaDaoSupport class handles much of the tedious housekeeping for you. The JpaTemplate helper class:

- Hides the underlying API differences
- Translates exceptions
- Manages a JPA entity manager
- Wraps the transaction handling
- Standardizes data access to a few consistent (across all Spring DAO implementation) and well-defined methods

Table 2 summarizes frequently used JpaTemplate methods that appear in Listing 10:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>find(Class &lt;T&gt; cls, Object id);</td>
<td>Finds a persisted instance by its primary key.</td>
</tr>
<tr>
<td>find(String query);</td>
<td>Finds persisted objects using a query string. The powerful query language,</td>
</tr>
<tr>
<td></td>
<td>an extended version of EJB QL, is fully described in JSR-220 (see</td>
</tr>
</tbody>
</table>
Resources).

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>persist(Object obj);</td>
<td>Saves the instance to the database. In JPA-speak, it persists the instance using the JPA entity manager.</td>
</tr>
<tr>
<td>merge(Object obj);</td>
<td>Updates the saved instance of the object with the information in the provided instance.</td>
</tr>
<tr>
<td>remove(Object obj);</td>
<td>Removes the persistence instance from the database.</td>
</tr>
</tbody>
</table>

Under the hood, the JpaTemplate helper class makes use of a JPA entity manager to handle all of the operations. The helper class handles the routine retrieval and closing of the entity manager during data access.

Other methods in the JpaTemplate class may be useful for your particular needs. Consult the JavaDoc for the Spring DAO API for more details (see Resources).

With the ability to persist Employee and Address instances and a concrete implementation of the EmployeeService available, it is time to test everything against a real relational database.

Wiring up Spring beans

Up to this point, it's still not clear when and how the Spring framework actually gets a chance to work with your POJOs. All the data access pieces of the puzzle are in place, but two questions linger: How does the Spring 2 engine know what to do, and how do you specify which relational database to use?

You will solve both of these mysteries immediately; you will see how to provide the beans wiring template to the Spring engine. The secret lies in an XML beans descriptor file, named dwspring-service.xml. This beans descriptor file is the wiring blueprint for the Spring engine, mentioned in Overview of Spring 2 framework operations. It describes the relationship between the various beans in a Spring application. This file is reproduced in Listing 11:

Listing 11. The dwspring-service.xml beans descriptor

```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"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="employeeService" class="com.ibm.dw.spring2.EmployeeDAO">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
  </bean>
  <bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="jpaVendorAdapter">
      <bean class="org.springframework.orm.jpa.vendor.TopLinkJpaVendorAdapter">
        <property name="showSql" value="true"/>
        <property name="generateDdl" value="true"/>
      </bean>
    </property>
  </bean>
</beans>
```
To test the EmployeeDAO implementation, you use an in-memory RDBMS called HSQLDB (see Resources). The binaries for HSQLDB is part of the "Spring 2 with dependencies" download.

In Listing 11, the lines that specifically configure an instance of HSQLDB are shown in bold. Later on (in Writing DAO integration tests against an RDBMS), you will see how to modify these lines to run the integration test against DB2 Express-C instead.

Remember that EmployeeDAO actually extends the JpaDaoSupport class. This class expects to be "injected" with a JPA EntityManagerFactory when it is loaded. It can then use this factory to obtain a JPA EntityManager for all the data access operations.

Figure 6 shows graphically how the beans are wired together within the dwspring2-service.xml file:

**Figure 6. Schematics for bean wiring**
In essence, Listing 11 is the wiring plan for the objects that need to be created by the Spring 2 engine, and Figure 6 is a graphical schematic of those objects. An important item to note in Listing 11 and Figure 6 is how an instance of the EmployeeDAO can be obtained via a bean called employeeService. This instance has its property called entityManagerFactory set to another bean called entityManagerFactory:

```xml
<bean id="employeeService" class="com.ibm.dw.spring2.EmployeeDAO">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>
```

The ref="" notation is a reference to another bean defined in the context -- usually in the same file.

**Dependency injection**

Filling a property with an externally created object, as you've just done, is called injection -- more specifically, dependency injection (DI), because the object being injected is often something that the receiving object is dependent on for proper operation. DI is used heavily throughout Spring's architecture. DI lets you write the component code without actively needing to look up or find dependent services (for example, to look up an EntityManagerFactory). Instead, you can write the component code as if one is already available, and the Spring engine injects the actual dependency into the component instance before the code is executed.

**Application of dependency injection**

If you follow through Listing 11 to the wiring of entityManagerFactory, you'll note that it has the following dependencies injected by Spring:
dataSource

jpaVendorAdapter

loadTimeWeaver

The dataSource bean is an instance of org.springframework.jdbc.datasource.DriverManagerDataSource, configured with an in-memory instance of a HSQLDB RDBMS.

The jpaVendorAdapter property is injected with a bean that connects to the actual JPA implementation for your Spring application. In this case, you are using the JPA reference implementation, accessed via the org.springframework.orm.jpa.vendor.TopLinkJpaVendorAdapter class. This class in turn needs to be configured with a databasePlatform property. This property is set to oracle.toplink.essentials.platform.database.HSQLPlatform, and this configuration supports access to the HSQLDB RDBMS. This bean’s generateDdl property controls whether Data Definition Language script is generated and executed. If this property is set to true, the schema in the database is recreated each time this bean is loaded. You should leave the property as true for integration testing purposes.

In the configuration of the dataSource bean, an instance of org.springframework.jdbc.datasource.DriverManagerDataSource is created. It is parameterized with:

- The HSQLDB database driver
- A JDBC URL that creates an in-memory database (the mem portion of the JDBC URL)
- The username and password (which default to sa and "", respectively, with HSQLDB)

The last lone transactionManager bean is configured for your later integration testing. This bean doesn't need to be wired up because the testing base class that is used later looks for this bean by type.

You should have a feel for how Spring 2 wires up beans by this time. You should also have an idea how to change the database from HSQLDB to DB2 Express-C, a step that you perform in the next section (Writing DAO integration tests against an RDBMS).

Section 6. Writing DAO integration tests against an RDBMS
In this section, you write and run an integration test to test the employee information application against a database.

Testing the EmployeeDAO implementation of EmployeeService

The only questions remaining are: How and when does the Spring 2 engine get invoked, and how does it know to use the dwspring2-service.xml configuration file?

The answer is obvious when you take a look at the source code of the integration test in EmployeeServiceIntegrationTest.java. This integration test tests the EmployeeDAO implementation of EmployeeService against an actual RDBMS. See the code fragment in Listing 12:

Listing 12. Integration test in EmployeeServiceIntegrationTest (part 1)

```java
package com.ibm.dw.spring2;
import java.util.Date;
...
import org.springframework.test.jpa.AbstractJpaTests;
public class EmployeeServiceIntegrationTest extends AbstractJpaTests {
    private EmployeeService employeeService;
    private long JoeSmithId = 99999;
    public void setEmployeeService(EmployeeService employeeService) {
        this.employeeService = employeeService;
    }
    protected String[] getConfigLocations() {
        return new String[] {
            "classpath:/com/ibm/dw/spring2/dwspring2-service.xml" ];
        }
    }
```

This set of integration tests is written with the help of the AbstractJpaTests class in the Spring 2 library. By implementing the getConfigLocations() method, highlighted in Listing 12, you can provide one or more beans configuration files to be parsed by the Spring 2 engine. You may have multiple configuration files because it is a common practice to separate the back-end and the user-interface bean configuration files.

The setEmployeeService() method in Listing 12 is a good example of dependency injection. When the Spring 2 engine loads this EmployeeServiceIntegrationTest class (derived from AbstractJpaTests), it discovers an unfulfilled dependency -- a property of type EmployeeService. The engine looks through the dwspring2-service.xml file for a configured bean of type EmployeeService and injects it via the setEmployeeService() method.

Autowiring by type
You may notice the lack of explicit wiring instructions for the injection of `employeeService` in the dwspring2-service.xml file. In fact, this injection happened automatically. This is called **autowiring** in Spring terminology.

The `AbstractJpaTests` base class is derived from the `AbstractDependencyInjectionSpringContextTests` class. `AbstractDependencyInjectionSpringContextTests` makes testing simple by configuring Spring's autowire-by-type feature by default. Any dependencies (public properties) of its subclass -- and `EmployeeServiceIntegrationTest` is such a subclass -- is automatically injected if a bean of the same type is found in the application context (configured by the dwspring2-service.xml file in this case).

### Integration test setup

A useful feature of `AbstractJpaTests` for testing is its ability to perform the tests in a transaction and then roll back all its effects after the test. This substantially speeds up the tests because it's not necessary to delete and recreate data in between each test run.

Listing 13 shows the code that performs the initial setup for each test. This setup code populates the database with three `Employee`s and their associated `Address`es. You must execute this code within the same transaction as each of the tests. Otherwise, you will not see the data inserted because the transaction is always rolled back. To perform setup within the same transaction, you override the `onSetUpInTransaction()` method, as shown in Listing 13:

#### Listing 13. Integration test in EmployeeServiceIntegrationTest (part 2)

```java
protected void onSetUpInTransaction() throws Exception {
    Employee emp1 = new Employee("0001", "Joe", "R", "Smith",
        "4853", "Engineer", 3, 'M',
        20000.00, 0.00, 0.00,
        new Address(10, "Walker Street")
    , new Date(), new Date());
    Employee emp2 = new Employee("0002", "John", "T", "Lockheed",
        "4333", "Sales", 2, 'M',
        40000.00, 0.00, 5000.00,
        new Address(20, "Walker Street")
    , new Date(), new Date());
    Employee emp3 = new Employee("0003", "Mary", "M", "Johnson",
        "4383", "Admin", 3, 'F',
        60000.00, 0.00, 390.00,
        new Address(123, "Booth Ave")
    , new Date(), new Date());

    employeeService.save(emp1);
    employeeService.save(emp2);
    employeeService.save(emp3);

    JoeSmithId = emp1.getEmpid();
}
```

Notice how straightforward it is to create an `Employee` instance and persist it through the JPA-based `employeeService`. Because the `save()` method calls JPA's `persist()` operation and the operation cascades from the `Employee` to the
Address object (you specified this in the Employee POJO’s JPA annotations), you can count on JPA to create a new record in the address table for you as well.

The setup code in onSetUpInTransaction() is performed for each and every test case. This ensures that the three employees are persisted before each test.

As an example of a test, Listing 14 shows the testModifyEmployee() test method for the EmployeeDAO's update() method:

### Listing 14. Integration test for EmployeeServiceIntegrationTest (part 3)

```java
public void testModifyEmployee() {
    String oldLastName = "Lockheed";
    String newLastName = "Williams";
    Employee emp = employeeService
        .findByEmployeeLastName(oldLastName).get(0);
    emp.setLastName(newLastName);
    Employee emp2 = employeeService.update(emp);
    assertEquals(newLastName, emp2.getLastName());

    List<Employee> results = employeeService
        .findByEmployeeLastName(oldLastName);
    assertEquals(0, results.size());

    results = employeeService
        .findByEmployeeLastName(newLastName);
    assertEquals(1, results.size());
}
```

The testModifyEmployee() test case in Listing 14 changes the last name of the employee "John Lockheed" from "Lockheed" to "Williams" using the EmployeeService's update() method. It then verifies that there is now an employee with the "Williams" last name by calling findByEmployeeLastName("Williams"). It also verifies that no employee with the last name "Lockheed" remains.

There are many other tests in EmployeeServiceIntegrationTest. You can study these tests to see how you can use the methods on the EmployeeService implementation to manipulate actual data (see Download).

Next, you set up the environment to run these integration tests against the RDBMS.

### Downloading and installing the reference JPA implementation

JPA is part of the EJB 3.0 specification. In turn, EJB 3.0 is a core component of the Java EE 5 specification. Because of this, an open source reference implementation of JPA is a part of the GlassFish Java EE 5 reference server.

If you haven't already done so, you should download and install this JPA implementation now to try out the integration tests (see Resources).

**Preparing your classpath**
To access the JPA capabilities in your code, you must have the JPA reference implementation JAR in your build and runtime classpaths:

1. In Eclipse, right-click the project name in the Navigator pane and select Properties.
2. Select Java Build Path and then in the dialog choose the Libraries tab.
3. Click the Add External JARs button and make sure the toplink-essentials.jar from the JPA download is included.

To compile and run the integration tests successfully, you must have the dependent libraries in the build path and runtime classpath. Table 3 shows the JAR files that you must have in your build and runtime class path. You can set these paths up in Eclipse:

<table>
<thead>
<tr>
<th>JAR file</th>
<th>Originating download</th>
</tr>
</thead>
<tbody>
<tr>
<td>commons-logging.jar</td>
<td>spring-framework-2.x-with-dependencies.zip</td>
</tr>
<tr>
<td>db2jcc.jar</td>
<td>IBM DB 2 Express-C distribution</td>
</tr>
<tr>
<td>db2jcc_licence_cu.jar</td>
<td>IBM DB2 Express-C distribution</td>
</tr>
<tr>
<td>hsqldb.jar</td>
<td>spring-framework-2.x-with-dependencies.zip</td>
</tr>
<tr>
<td>junit.jar</td>
<td>spring-framework-2.x-with-dependencies.zip</td>
</tr>
<tr>
<td>log4j-1.x.xx.jar</td>
<td>spring-framework-2.x-with-dependencies.zip</td>
</tr>
<tr>
<td>persistence.jar</td>
<td>spring-framework-2.x-with-dependencies.zip</td>
</tr>
<tr>
<td>spring.jar</td>
<td>spring-framework-2.x-with-dependencies.zip</td>
</tr>
<tr>
<td>spring-core.jar</td>
<td>spring-framework-2.x-with-dependencies.zip</td>
</tr>
<tr>
<td>spring-jpa.jar</td>
<td>spring-framework-2.x-with-dependencies.zip</td>
</tr>
<tr>
<td>spring-mock.jar</td>
<td>spring-framework-2.x-with-dependencies.zip</td>
</tr>
<tr>
<td>toplink-essential.jar</td>
<td>Reference JPA implementation download</td>
</tr>
</tbody>
</table>

Table 3. Open source libraries that must be in your classpath

Including a persistence.xml file

A persistence.xml file is required by the JPA specification, although it does not provide configuration information in this Spring JPA integration scenario.

The persistence.xml file is the description of a persistence unit. In JPA terminology, a persistence unit includes an EntityManagerFactory (and associated configuration information), together with the EntityManagers that it creates and the classes that those EntityManagers manage (and the JPA metadata for these classes, as either annotations or XML).

In this case, the persistence.xml file is very simple, as shown in Listing 15:
Listing 15. The META-INF/persistence.xml file

```xml
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
  <persistence-unit name="dwSpring2Jpa" transaction-type="RESOURCE_LOCAL" />
</persistence>
```

Under some vendors’ configurations, the persistence.xml file may contain relevant description information, but it doesn't for Spring/JPA integration. You just need to make sure you have a copy placed at META-INF/persistence.xml.

Running the Spring integration tests

With the build path configured, you can now run the integration tests.

To run the integration test in Eclipse, highlight the EmployeeServiceIntegrationTest.java file in the Navigator pane and then right-click and select Run As -> JUnit Test. This starts the integration test. Remember that this test runs against an in-memory database and is quite fast. During execution, you should see log messages of the table creation, SQL inserts, and queries in the Eclipse Console window. See Figure 7 for a sample run of the integration test:

![Figure 7. Integration test running in Eclipse](image)

Switching datasource from the in-memory database to DB2
Express-C

When you use Spring DAO and the DAO design pattern in general, the datasource is completely encapsulated and hidden from the application code. In fact, the coding in the example is completely independent of the relational database used, in addition to being independent of the JPA vendor used. This flexibility lets you test the same application code on a database that provides fast execution (such as an in-memory database) and then deploy it on a robust and scalable commercial-grade database (such as IBM DB2 Express-C).

To switch between different databases, you only need to make a modification in the dwspring2-service.xml beans descriptor file. Listing 16 shows the changes (highlighted in bold) required to switch the datasource from HSQLDB to IBM DB2 Express-C:

**Listing 16. Changing the datasource to DB2 Express-C**

```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"
xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="employeeService" class="com.ibm.dw.spring2.EmployeeDAO">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
  </bean>

  <bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="jpaVendorAdapter">
      <bean class="org.springframework.orm.jpa.vendor.TopLinkJpaVendorAdapter">
        <property name="showSql" value="true"/>
        <property name="generateDdl" value="true"/>
        <property name="databasePlatform" value="oracle.toplink.essentials.platform.database.DB2Platform"/>
      </bean>
    </property>
    <property name="loadTimeWeaver">
      <bean class="org.springframework.instrument.classloading.SimpleLoadTimeWeaver"/>
    </property>
  </bean>

  <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.ibm.db2.jcc.DB2Driver"/>
    <property name="url" value="jdbc:db2://192.168.23.36:50000/dwspring"/>
    <property name="username" value="bill"/>
    <property name="password" value="lotus123"/>
  </bean>

  <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
    <property name="dataSource" ref="dataSource"/>
  </bean>

</beans>
```

The modifications highlighted in Listing 16 assume that you have an instance of DB2 Express-C at 192.168.23.36 with port 50000 (default). You need to create a
database called dwspring and provide full access to the application user (bill in Listing 16; change it to your specific user).

Now run the integration tests again. The test connects to the DB2 Express-C database, creates the tables, and executes all the tests. Other than the execution speed being a little slower -- because the tests now run over the network and use on-disk data storage -- you will notice little change in its execution. Figure 8 shows a typical run against DB2 Express-C in Eclipse:

Figure 8. Running integration tests against DB2 Express-C

The data tier for your application is now completed and fully tested. This tier can be used with a variety of user interfaces. For example, you can layer a command-line interface over it, or you might want to create a GUI fat client that uses it. The decoupling between UI and data access tier is important because it enables you to repurpose and adapt independently tested data tier code easily.

For this exercise, you'll use Spring MVC to create a Web-based UI for the application.

Section 7. Understanding Spring MVC
In this section, you create a Web-based UI for the employee information application using Spring MVC.

Implementing the MVC design pattern with Spring

The Model-View-Controller (MVC) design pattern is simply a great way to connect Web tier and data tier components together to create a Web application. Almost all modern-day application frameworks provide support for building MVC-patterned applications.

One key benefit the MVC pattern delivers is a clean separation between data and presentation within an application. This clean separation enables the data and presentation to evolve independently. This is highly valuable in production systems where both data and presentation typically change, because of differing requirements, over time.

For an in-depth general description of the MVC design pattern, see Resources. The following discussion provides an introductory view, applied specifically to the example application.

The model part of your application consists of elements that encapsulate data and the operations on those data. In fact, you've already seen the model -- the domain model -- in the example application. The entire domain model is created and tested without incorporating any presentation elements (UI, reports, and so on). The MVC design pattern lets you create and test the model independently of the view.

The view part of the application is created completely using JavaServer Pages (JSP) technology and the JSP Standard Tag Library (JSTL). You will discover shortly that you can create the view independently of the model. This is great in production because Web page designers, instead of Java developers, can handle the creation and testing of the view.

The controller is the vital link between the model and the view. Spring MVC is a framework for building Web-based applications. In Spring MVC, the controller deals with incoming HTTP Web requests.

A comprehensive collection of controller library components are available for you to subclass. All of the Spring MVC controller library classes implement the org.springframework.web.servlet.mvc.Controller interface. This interface has a single method:

```java
ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) throws Exception
```

A controller examines and processes the incoming request and then returns a ModelAndView object. Spring has an abstract class called org.springframework.web.servlet.mvc.AbstractController that
implements this interface and handles most of the caching and session-management features. Practically all of the specialized Spring controllers subclass this abstract class. The two Spring controller classes you use in the example application are described in Table 4:

<table>
<thead>
<tr>
<th>Controller class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbstractController</td>
<td>Base abstract controller class. Useful when your controller does not need to handle incoming command/parameters or process forms. In the example application, it renders the initial page that lists all the employees in the database.</td>
</tr>
<tr>
<td>AbstractCommandController</td>
<td>An abstract controller class that works with incoming commands. This class parses the incoming HTTP requests and binds a specified Java object instance to the request parameters, enabling easy processing of the parameters in the controller logic.</td>
</tr>
</tbody>
</table>

For more information on other controller base classes that are available in Spring, see Resources.

Creating the MainController

The MainController in this application handles the initial incoming request into the application. This happens when a user enters the home page of the application at the URL http://localhost:8080/dwspring/index.cgi.

This controller displays all the employees in the system and lets the user click a link on an employee number to obtain employee details. The code for MainController is shown in Listing 17:

Listing 17. The MainController for displaying an employee list

```java
package com.ibm.dw.spring2.web;

import java.util.List;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;
import com.ibm.dw.spring2.Employee;
import com.ibm.dw.spring2.EmployeeService;

public class MainController extends AbstractController{
    public ModelAndView handleRequestInternal(HttpServletRequest req,
                                               HttpServletResponse resp) {
        List<Employee> emps = employeeService.findAll();
        return new ModelAndView("home", "employees", emps);
    }

    private EmployeeService employeeService;
}
public void setEmployeeService(EmployeeService employeeService) {
    this.employeeService = employeeService;
}

In Listing 17, note that the MainController does not display any user interface. Instead, it accesses the data tier for the list of employees. To obtain the list, it calls the EmployeeService implementation's findAll() method. This implementation is "injected" into the controller via Spring dependency injection and is an instance of EmployeeDAO. You configure this in the dwspring2-servlet.xml beans descriptor momentarily (in Wiring Spring MVC beans).

To render the UI, the controller creates a ModelAndView object and returns it. The ModelAndView object is created with three arguments, described in Table 5:

<table>
<thead>
<tr>
<th>Position</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>View name</td>
<td>This is a string that names the view. This string is passed to a view resolver component to be resolved into a specific view/presentation component -- a JSP in this case. The view resolver is wired at run time by the Spring engine. You configure the view resolver in the dwspring2-servlet.xml beans descriptor file.</td>
</tr>
<tr>
<td>2</td>
<td>Model name</td>
<td>This is a string that names the data fetched from the domain model. The data itself is passed in the third argument. This data is typically rendered by the view during execution. For JSPs, this name becomes the name of a variable. The value(s) of this variable can be rendered using JSP commands or JSTL tags.</td>
</tr>
<tr>
<td>3</td>
<td>Model object</td>
<td>This is the model data to be rendered by the view. It can be referenced within the view by the name provided in argument two.</td>
</tr>
</tbody>
</table>

To grasp the operation of Spring MVC, it's important to understand the flow of requests. Figure 9 shows the request flow through Spring MVC components:

**Figure 9. Flow of requests through Spring MVC**
The Spring components are the dotted-line rectangles in Figure 9. The incoming request is first handled by Spring's DispatcherServlet. This servlet maps the incoming URL of a request through a wired URL mapper component. This mapper component provides the actual controller that handles the request. Once the controller processes the request, it passes a view name back to Spring MVC. Spring MVC then invokes a wired view-resolver component, passing on the model data to be rendered. The view-resolver component resolves the view name into a view object. This view object is passed the model data and renders it for the user.

Both the URL mapper and the view resolver are both Spring 2 supplied components that you can instruct the Spring engine to wire up in a beans descriptor XML file.

**Wiring Spring MVC beans**

Under Spring MVC, the Web application is executed against a Web application context. This context is configured by a beans descriptor wiring XML file, just like the data tier code.

By default, the name of the configuration file is created using the following rule: Take the servlet name of the Spring DispatcherServlet and append _servlet.xml.

The Spring DispatcherServlet is the class that accepts incoming Web requests through the Web container (Tomcat in this case) and dispatches it to one of the controllers.

In this case, the DispatcherServlet is configured with the name dwspring2 and as result, the configuration file is expected at dwspring2_servlet.xml. This configuration file is shown in Listing 18:

**Listing 18. The dwspring2-servlet.xml beans descriptor**

```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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans.xsd">
```
In Listing 18, you can see the wiring of the various beans. First, the mainController bean is an instance of the MainController class, and it is injected with a reference to the employeeService bean. This bean, of course, is an instance of EmployeeDAO, defined at the data tier in dwspring2-service.xml.

Another controller, called empDetailsController, is also injected with the same employeeService instance through the reference. You will see the code for this controller later in this section.

The third bean wired is controllermap. This is an instance of org.springframework.web.servlet.handler.SimpleUrlHandlerMapping. This Spring-supplied bean can map incoming URL requests to different controllers, according to a mappings property. In this case, /home.cgi is mapped to MainController, and /empdet.cgi is mapped to empDetailsController.

The last bean wired is a view resolver. This Spring-supplied bean maps a view name to the actual view resource. In this case, an instance of org.springframework.web.servlet.view.InternalResourceViewResolver is used. This view resolver creates a resource URL from an incoming view name by adding a prefix and a suffix. In Listing 18, the prefix is configured to be /jsp and suffix is .jsp. For example, a view name of home is mapped to /jsp/home.jsp. This means that the URL /jsp/home.jsp renders the view named home.

Configuring the Spring DispatcherServlet

The WEB-INF/web.xml deployment descriptor contains the configuration of the DispatcherServlet. Listing 19 shows the relevant code segment:
Listing 19. Spring DispatcherServlet configuration in web.xml

```xml
<servlet>
  <description>
    Spring MVC Dispatcher Servlet
  </description>
  <display-name>DispatcherServlet</display-name>
  <servlet-name>dwspring2</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>dwspring2</servlet-name>
  <url-pattern>*.cgi</url-pattern>
</servlet-mapping>
```

Specifying `<load-on-startup>` ensures that when the Web application is first started, the servlet is loaded. This also triggers the parsing of the `dwspring2-servlet.xml` configuration file.

The `<servlet-mapping>` tag tells Tomcat to route all Web requests for `*.cgi` resources to the `DispatcherServlet`. The asterisk represents a wildcard match. Therefore, any requests in the form of `http://host:port/dwspring/* .cgi` are forwarded to the `DispatcherServlet`.

Creating a JSP/JSTL based UI with Spring MVC

The `MainController` class passes a `List` of `Employees` to the home view. The home view is resolved by the `InternalResourceViewResolver` to `/jsp/home.jsp`. Listing 20 is the code for home.jsp. This page simply displays the list of employees passed in the `ModelAndView` as `employees`.

Listing 20. home.jsp

```jsp
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
  <link rel="stylesheet" type="text/css" href="css/dwstyles.css"/>
  <title>dW Spring 2 Employee Data from DB2 via JPA</title>
</head>
<body>
<h1>Spring 2 JPA Employee List</h1>
<br/>
 &nbsp;&nbsp;<table>
  <tr>
    <th>Employee number</th>
    <th>Name</th>
  </tr>
  <c:forEach var="emp" items="${employees}">
    <tr>
      <td><a href="empdet.cgi?empID=${emp.empid}">${emp.empno}</td>
      <td>${emp.firstName} ${emp.midInitial} ${emp.lastName}</td>
    </tr>
  </c:forEach>
</table>
</body>
```

Introduction to Spring 2 and JPA

© Copyright IBM Corporation 1994, 2008. All rights reserved.
In Listing 20, you can see the use of JSTL's `<c:forEach>` tag in iterating through each of the employees in the `${employees}` list. The code creates a URL around the `${emp.empno}` information. For example, the URL generated around the Joe Smith row is http://localhost:8080/dwspring/empdet.cgi?empID=1.

When you click on this URL link, the resource mapped to empdat.cgi is activated. From your web.xml configuration, Tomcat knows to send all the *.cgi resource requests to Spring's DispatcherServlet, and Spring's DispatcherServlet knows to use the SimpleUrlHandlerMapping from your configuration in dwspring2-servlet.xml. The SimpleUrlHandlerMapping tells the Spring engine to direct the request to the EmpDetailsController.

A controller that processes commands

The code for EmpDetailsController is shown in Listing 21:

Listing 21. The EmpDetailsController for displaying employee details

```java
package com.ibm.dw.spring2.web;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.validation.BindException;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractCommandController;
import com.ibm.dw.spring2.Employee;
import com.ibm.dw.spring2.EmployeeService;

public class EmpDetailsController extends AbstractCommandController {
    public EmpDetailsController() {
        setCommandClass(EmployeeDetailsCommand.class);
    }

    private EmployeeService employeeService;
    public void setEmployeeService(EmployeeService employeeService) {
        this.employeeService = employeeService;
    }

    protected ModelAndView handle(HttpServletRequest req,
            HttpServletResponse resp, Object cmd, BindException ex) throws Exception {
        EmployeeDetailsCommand ecmd = (EmployeeDetailsCommand) cmd;
        Employee emp = employeeService.findById(ecmd.getEmpID());
        return new ModelAndView("empdetails", "emp", emp);
    }
}
```

The EmpDetailsController is a subclass of AbstractCommandController. AbstractCommandController is a useful controller for subclassing from whenever you are processing a command that you need to perform one-shot based on a button or URL click.
In this case, the user clicks on the employee number URL and the command is used to display the employee's detail information.

AbstractCommandController's main added value is that it parses the incoming request for request arguments and binds them to an instance of an incoming command class that you define. The names of the incoming request parameters are be matched against the attribute(s) of the command class. The command class is called EmployeeDetailsCommand. The code for EmployeeDetailsCommand is shown in Listing 22:

**Listing 22. The EmployeeDetailsCommand command class**

```java
package com.ibm.dw.spring2.web;
public class EmployeeDetailsCommand {
    private long empID;
    public long getEmpID() {
        return empID;
    }
    public void setEmpID(long empID) {
        this.empID = empID;
    }
}
```

In Listing 22, you can see the simple structure of the EmployeeDetailsCommand command class. It has only one attribute, called empID. AbstractCommandController looks for an incoming request parameter called empID and binds it to an instance of the EmployeeDetailsCommand class. It then passes it to the EmployeeDetailsController, where the handle() method is called (see Listing 21).

In general, you can subclass your controller from AbstractCommandController to handle an arbitrary number of incoming request parameters. All you need to do is define a command class with the corresponding attributes.

In Listing 21, the EmployeeDetailsController casts the cmd object passed in back to an EmployeeDetailsCommand instance and extracts the empID. The empID is then used to query the EmployeeService for a specific Employee instance. This is done via the DAO's employeeService.findById() method.

In Listing 21, the resulting Employee record is passed on to the view as an emp variable. The view specified is the empdetails view. Again, the Spring engine looks to the InternalResourceViewResolver to resolve the view. This view resolver appends the appropriate prefix and suffix and returns /jsp/empdetails.jsp as the view handler.

The code for empdetails.jsp is shown in Listing 23:

**Listing 23. empdetails.jsp**
In Listing 23, the incoming \${emp} variable is used to display the employee's detailed information in an HTML table.

Note the link at the bottom of the page that maps back to the employee listings page. The URL is home.cgi. If you click on this link, the mapping processes start again and pass you through to MainController, and then /jsp/home.jsp, and so on.

Adding a Cascading Stylesheet

Both home.jsp and empdet.jsp use the css/dwstyles.css stylesheet to format their HTML. The stylesheet code is shown in Listing 24:

Listing 24. The dwstyles.css stylesheet

```css
h1 {
  font-family: arial;
  font-size: 28;
}
```
Creating an Eclipse WTP dynamic Web project

To see the complete application in action, you need to:

1. Compile the code.

2. Build a deployable WAR file. A WAR file is a JAR file created in a standard format for deployment in a J2EE compatible Web tier container (such as Tomcat).

3. Deploy the WAR file to a Tomcat server.

Create a new dynamic Web project in Eclipse by using File->New->Project... In the New Project wizard, select Dynamic Web Project under the Web category and name it spring2Web. See Figure 10:
Next, add the source file and lay it out as shown in Figure 11. If you have the downloaded source distribution (see Download), you can add these files to the src folder by dragging and dropping them from the file manager.

Figure 11. Layout of the src directory
Adding application libraries

Unlike the integration testing environment for the data tier code, the Web application in a WAR file must include all the library JARs that it needs. Figure 12 shows the JAR files that you should drag and drop into the WEB-INF/lib directory:

**Figure 12. Layout of the WebContent directory**
In Figure 12, note that jstl.jar and servlet-api.jar can be found in the lib\j2ee directory of the Spring with-dependencies download. The standard.jar tag library is from the lib\jakarta-taglibs directory.

In addition to the library JAR files, the placement for the other configuration files is also shown in Figure 12. Make sure you have located your files appropriately before continuing.

Exporting the WAR file from the Eclipse project

To create a WAR file that you can deploy to Tomcat, you need to build the project and export it as a WAR file.
You can build the project from Tomcat by right-clicking on the *spring2Web project* in the Navigator and selecting **Build Project**.

To export the project as a WAR file, select **File->Export...**. In the export wizard, select **WAR File**. The **Export** wizard dialog is shown in Figure 13. Name the exported WAR file dwspring.war and click **Next**.

**Figure 13. The export wizard**
Section 8. Tomcat 5.5 as a host for Spring MVC applications

This section shows how to configure Apache Tomcat, an open source Web tier container, to work with Spring and host the sample application.

JSP and servlet support in Tomcat 5.5

There is little question that Tomcat is by far the most popular and mature open source Web tier server available. As a Web tier server, Tomcat can run and execute Web applications that consist of JSPs and servlets. The Tomcat 5.5.x that you will be working with supports the Servlet 2.4 and JSP 2.0 standards (see Resources).

If you haven't already done so, download and install the latest version of Tomcat 5.5.x to try out the example (see Resources). Choose the ZIP file download for the server and unarchive it into a directory of your choice.

This section provides some general operational guidelines for the Tomcat server. See the Tomcat documentation for more details.

Basic Tomcat 5.5 operations

The most frequent operations you perform with the Tomcat server are:

- Starting and stopping the server
- Deploying and undeploying applications to the server

Starting and stopping the Tomcat server

After unarchiving the Tomcat server's binaries, you can start the server by going into the server's bin subdirectory and running the startup.bat script. Another console window pops up running the server.

To shut down the server, go into the bin subdirectory and run the shutdown.bat script.

Deploying applications to Tomcat

To easily deploy applications to Tomcat, you can either use the built-in Manager Web application or copy the WAR file directly into the webapps subdirectory of the Tomcat server.

If you want to use the Manager Web application, you need to enable access by providing a user with the "manager" role. Before you start the Tomcat server, look
under the conf subdirectory for a file called tomcat-users.xml. In this file, look for:

```xml
<User username="tomcat" password="tomcat" roles="tomcat " />
```

Change the line to:

```xml
<User username="tomcat" password="tomcat" roles="tomcat, manager" />
```

The Manager application can then be accessed via: http://localhost:8080/manager/html.

If you deploy by copying the WAR file into the webapps directory, you might need to wait for a little while before the Tomcat server detects the update and deploys the new WAR.

**If redeployment should fail**

As you experiment with the sample application, you might need to redeploy a new version of the code to the Tomcat server.

Depending on the release of the Tomcat you use and when you deploy/redeploy the dwspring.war file, sometimes you might encounter deployment problems. If you ever come across problem with deployment to the server, the fail-safe way to deploy a WAR is:

1. Shut down the Tomcat server.
2. Go into the webapps directory and make sure any dwspring.war file is removed.
3. Remove the dwspring directory.
4. Copy the new dwspring.war file into the webapps directory.
5. Restart the Tomcat server.

---

**Section 9. Preparing Tomcat for Spring 2**

Before you can deploy the dwspring.war file successfully to Tomcat, some server setup is required.

The main procedures you perform in this section are:
1. Adding the Spring 2 classloader to Tomcat
2. Adding the Spring 2 context loader listener to Tomcat
3. Copying DB2 JDBC drivers to Tomcat
4. Configuring JNDI DB2 datasource for Tomcat

Adding the Spring 2 classloader to the Tomcat server

When a Spring JPA application run on Tomcat, bytecode "weaving" during class loading is required for the JPA support to work properly. The standard classloader from Tomcat does not support this. You need to use a Spring-specific classloader to make this happen.

To install this Spring-specific classloader into the Tomcat server, first copying the spring-tomcat-weaver.jar into the Tomcat's server/lib subdirectory. This directory contains libraries that are private to the Tomcat server. You can find the spring-tomcat-weaver.jar library under the dist/weaver directory in the Spring 2.0 download.

Next, you must let Tomcat know that the standard classloader should be replaced for the sample application. You can specify this in the META-INF/context.xml file within the WAR file. The code in bold in Listing 25 configures the classloader:

Listing 25. Configuring the classloader in META-INF/context.xml file

```xml
<Context>
  <Loader loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstrumentableClassLoader"/>
  ...
</Context>
```

Add Spring 2's context loader listener to Tomcat

Spring 2 requires hooking into Tomcat's context loading pipeline. You can configure this by adding the following lines to the WEB-INF/web.xml file in the WAR file:

```xml
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
```

This must go before the `<servlet>` and `<servlet-mapping>` definitions in the web.xml file.
Copying the DB2 JDBC drivers to the Tomcat server

Tomcat is a Web tier container and as such can manage and pool its own database connections. Datasources are managed by Tomcat and are available through a standard Java Naming and Directory Interface (JNDI) lookup mechanism. The employee system runs as a Web application inside Tomcat and should obtain its datasource through Tomcat’s JNDI.

For Tomcat to find the JDBC driver during Web application deployment, you need to copy the JAR files into Tomcat’s system library directory. Before starting your Tomcat server, copy the two JDBC driver JAR files from your DB2 distribution to Tomcat’s common\lib directory. These two files are named db2jcc.jar and db2jcc_license_cu.jar.

Libraries placed into the common\lib directory can be used by both Tomcat server and Web applications.

Configuring DB2 datasource management and JNDI on Tomcat 5

You can configure an application-accessible DB2 datasource as a JNDI resource in the Web application context. You do this by placing the highlighted code in Listing 26 within a META-INF/context.xml file within your WAR:

Listing 26. Configuring JNDI resource in META-INF/context.xml

```xml
<Context>
  ...
  <Resource name="jdbc/dwspring2" auth="Container" type="javax.sql.DataSource"
    maxActive="100" maxIdle="30" maxWait="10000"
    username="bill"
    password="lotus123" driverClassName="com.ibm.db2.jcc.DB2Driver"
    url="jdbc:db2://192.168.23.36:50000/dwspring"/>
</Context>
```

You need to replace the DB2 Express-C server host, username, and password in Listing 26 to reflect your own DB2 Express-C installation.

The configuration in Listing 27 makes the JNDI datasource available through the name java:comp/env/jdbc/dwspring2.

You must also add a resource reference to the deployment descriptor web.xml file. Add this `<resource-ref>` element to the very end of the web.xml file, as shown in Listing 27:

Listing 27. Adding a JNDI resource reference for the datasource

```xml
  ...
```

Introduction to Spring 2 and JPA
© Copyright IBM Corporation 1994, 2008. All rights reserved.
The configuration in Listing 27 makes the container JDBC managed datasource available for use within the Web application.

Section 10. Configuring the Spring 2 application for Tomcat Deployment

To configure your data tier code for Tomcat deployment, you still need to attend to a couple of details:

- Inform the Spring 2 engine the location of the beans descriptor configuration file
- Modify Spring 2 wiring of the datasource configuration to use Tomcat’s datasource management and pooling through JNDI

Informing Spring 2 engine of the location of configuration files

For the Spring 2 engine to wire up the required data tier beans, it must first be able to locate and process the beans descriptor configuration file(s) for the data tier.

In this case, the file is called dwspring-service.xml. The location of this file needs to be specified in the context parameters supplied to the Tomcat server during deployment.

This context parameter needs to be the first element in the WEB-INF/web.xml deployment descriptor, which is bold in Listing 28:

Listing 28. Context Parameter in web.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4"
 xmlns="http://java.sun.com/xml/ns/j2ee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
 http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
 <display-name>
  spring2web</display-name>
  <context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>/WEB-INF/dwspring2-service.xml</param-value>
  </context-param>
</web-app>
```
The dwspring2-service.xml is a modified version of the configuration file you used earlier during testing.

This configuration file is modified to use the Tomcat server's JDBC connection management, through JNDI, rather than its own.

Modifying Spring 2 wiring to look up Tomcat JNDI datasource

The changes required in the dwspring2-service.xml to use Tomcat 5 for JNDI datasource are highlighted in bold in Listing 29:

Listing 29. Modifying dwspring2-service.xml for Tomcat JNDI lookup

```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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="employeeService" class="com.ibm.dw.spring2.EmployeeDAO">
        <property name="entityManagerFactory" ref="entityManagerFactory"/>
    </bean>

    <bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <property name="jpaVendorAdapter">
            <bean class="org.springframework.orm.jpa.vendor.TopLinkJpaVendorAdapter">
                <property name="showSql" value="true"/>
                <property name="generateDdl" value="false"/>
                <property name="databasePlatform" value="oracle.toplink.essentials.platform.database.DB2Platform"/>
            </bean>
        </property>
        <property name="loadTimeWeaver">
            <bean class="org.springframework.instrument.classloading.SimpleLoadTimeWeaver"/>
        </property>
    </bean>

    <bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="java:comp/env/jdbc/dwspring2"/>
    </bean>

    <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory"/>
        <property name="dataSource" ref="dataSource"/>
    </bean>

</beans>
```

The bean used for creating datasource in Listing 29 is now org.springframework.jndi.JndiObjectFactoryBean. This bean can be used to perform a JNDI lookup for a specific container-managed resource. In this case, the name of the DB2 Express-C datasource managed by Tomcat is configured for use.
to be `java:comp/env/jdbc/dwspring2`.

In Listing 29, note that `jpaVendorAdapter`'s `GenerateDdl` property is set to `false`. This is necessary because you don't want Spring 2 to delete and recreate all the database tables every time you start up the application. This property should only be set to `true` during integration testing.

Section 11. Trying out the Spring 2 Web application

In this section, you add some data programmatically to the employee database and try out the employee information application.

Adding data into DB2 Express-C within a transaction

Before you can try out the application, you need some employee data in the database. Because the integration tests remove all tables each time they run (by rolling back the transaction), the database doesn't include any ready-to-use data.

Of course, you can use DB2 Express-C tools to enter the data manually. You'll learn here how you can add some data programmatically.

A great feature of the integration tests based on `AbstractJpaTests` is the fact that all database changes are rolled back upon completion of a test, allowing the next test to run quickly. If each test were to run to completion in a transaction, the data changes would be made permanent in the RDBMS, and you'd need to delete them before starting the next test. So in theory, this class is of little use if you want to add data to the database.

Fortunately, any data modified in an `AbstractJpaTests`-based integration test can be committed to the RDBMS simply by calling the `setComplete()` method within the test. This method commits the transaction instead of rolling it back and makes the changes permanent.

The `FillTableWithEmployeeInfo` class, part of the `Spring2Tutorial` project, is shown in Listing 30. This class takes advantage of this ability to persist information of six employees to DB2 Express-C.

**Listing 30. Adding data to the database via FillTableWithEmployeeInfo**

```java
package com.ibm.dw.spring2;
import java.util.Date;
import java.util.List;
import org.springframework.test.jpa.AbstractJpaTests;
public class FillTableWithEmployeeInfo extends AbstractJpaTests {
```
private EmployeeService employeeService;

public void setEmployeeService(EmployeeService employeeService) {
    this.employeeService = employeeService;
}

protected String[] getConfigLocations() {
    return new String[] {
        "classpath:/com/ibm/dw/spring2/dwspring2-service.xml"};
}

public void testTableFiller() {

    // Employee information
    Employee emp1 = new Employee("0001", "Joe", "R","Smith",
        "4853", "Engineer", 3, 'M',
        20000.00, 0.00, 0.00,
        new Address(10, "Walker Street")
    , new Date(), new Date());
    Employee emp2 = new Employee("0002", "John","T","Lockheed",
        "4333", "Sales", 2, 'M',
        40000.00, 0.00, 5000.00,
        new Address(20, "Walker Street")
    , new Date(), new Date());
    Employee emp3 = new Employee("0003", "Mary","M","Johnson",
        "4383", "Admin", 3, 'F',
        60000.00, 0.00, 390.00,
        new Address(123, "Booth Ave")
    , new Date(), new Date());
    Employee emp4 = new Employee("0004", "Mike","S","Lee",
        "4322", "Sales", 3, 'M',
        30000.00, 0.00, 20000.00,
        new Address(7, "Wilard Drive")
    , new Date(), new Date());
    Employee emp5 = new Employee("0005", "Joan","K","Winfry",
        "4113", "Marketing", 2, 'F',
        40000.00, 0.00, 0.00,
        new Address(1293, "Davis Blvd")
    , new Date(), new Date());
    Employee emp6 = new Employee("0006", "Steve","L","Bingham",
        "4632", "Marketing", 3, 'M',
        50000.00, 0.00, 0.00,
        new Address(5, "Booth Ave")
    , new Date(), new Date());

    employeeService.save(emp1);
    employeeService.save(emp2);
    employeeService.save(emp3);
    employeeService.save(emp4);
    employeeService.save(emp5);
    employeeService.save(emp6);
    setComplete();
}

In Eclipse’s Navigator view, right-click on FillTableWithEmployeeInfo.java and select Run As... JUnit Test to populate the database with data.

Using the employee information application

To access the home page of the application after deployment to Tomcat, enter the
following URL into your browser: http://localhost:8080/dwspring/home.cgi. This assumes that you're running the Tomcat server on your local machine. If you are using another server on the network, just replace the host name in the URL.

The first page of the application, courtesy of MainController and home.jsp, is displayed as in Figure 14:

**Figure 14. Employee selection page**

![Spring 2 JPA Employee List](image)

Each of the employee numbers is a URL link that you can click display an employee-details page containing information for the selected employee as in Figure 15:

**Figure 15. Employee-details page**

![Employee Details](image)
The page in Figure 15 is created by the EmpDetailsController and rendered through the empdet.jsp view.

Section 12. Summary

Spring 2 is a versatile framework. You can create Web-based, server-side applications using software components that are dynamically wired together at runtime through XML configuration files.

You can implement business objects using POJOs. POJOs are easy to create and test outside of any containers and are reusable in other applications or in another part of the same application.

Spring 2's JPA integration lets you add database persistence metadata to POJOs through JPA annotations within the POJOs' source code.
With Spring DAO support, you can create services that operate on POJOs and use JPA for persistence -- without needing to write tedious entity-manager and transaction-management code explicitly.

Spring's JPA testing support lets you create integration tests on your JPA-enabled POJOs and DAOs. You can execute these tests against an actual RDBMS running either in-memory or on an external server.

You can use Spring MVC to add a Web-based user interface to a tested data-tier stack. Spring MVC can work with a variety of view technologies, including JSP and JSTL, which you used in this tutorial. By keeping the model (domain model) code cleanly separated from the user interface code, Spring simplifies the maintenance of your Web applications.

You can package Web-based Spring applications into WAR files for production deployment on a Tomcat 5 server.
## Downloads

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
<th>Download method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code</td>
<td>j-spring2code.zip</td>
<td>20KB</td>
<td>HTTP</td>
</tr>
</tbody>
</table>

*Information about download methods*
Resources

Learn

- "DB2 Express-C, the developer-friendly alternative" (Grant Hutchison, developerWorks, February 2006): Get started quickly using DB2 Express-C for all of your applications.

- Core J2EE Patterns - Data Access Object: The DAO design pattern is well described in the Core J2EE Pattern Catalog.

- Model-View-Controller: The Sun Java BluePrints reference describes the MVC design pattern, used frequently when connecting a Web tier user interface to data tier components.

- HSQLDB: The 100% Java HSQLDB database is used often in Spring development for integration testing of DAO-based service implementations. It's fast and easy to use in-memory mode of operation lends itself to test applications.

- Spring framework documentation: You can find documentation on the Spring's DAO support classes and MVC base classes in the JavaDoc at the official Spring site.

- JSR 220: Enterprise JavaBeans 3.0 specification: Details of JPA, as well as the rich query language that extends EJB QL, can be found in JSR 220.

- "Design enterprise applications with the EJB 3.0 Java Persistence API" (Borys Burnayev, developerWorks, March 2006): Study a sample application that validates a Java EE design approach leveraging JPA and illustrates key design decisions.

- "Kick-start your Java apps: Free software, fast development" (Sing Li, developerWorks, February 2006) and "Kick-start your Java apps, Part 2 (Sing Li, developerWorks, April 2006): Use IBM-backed open source and free software, including DB2 Express-C, with these tutorials to kick-start your Java Web-based application development.

- Introduction to JavaServer Pages technology (Noel J. Bergman, developerWorks, August 2001) and A JSTL primer, Part 1: The expression language (Mark Kolb, developerWorks, February 2003): Learn more about JSP and JSTL programming.

- Apache Tomcat 5.5, used in this tutorial, supports Servlets 2.4 (specified by JSR-154) and JavaServer Pages 2.0 (specified in JSR-152).

- The Java technology zone: Hundreds of articles about every aspect of Java programming.

Get products and technologies

- The Open source | Eclipse zone: developerWorks offers a wealth of information on this robust platform of frameworks and tools.

- Spring framework: Download the Spring 2 framework.
• **DB2 Express-C**: A freely available version of DB2 9, which offers the same core data server features as DB2 Express Edition and provides a solid base to build and deploy applications developed using C/C++, Java, .NET, PHP, and other programming languages.

• **Apache Tomcat**: The latest version of the Tomcat server is available from the official Tomcat site.

• **JPA reference implementation**: The JPA reference implementation is available as an independent download from the open source GlassFish Java EE 5 server project.

### About the author

Sing Li
Sing is a consultant and an active author with over two decades of industry experience. He has contributed to *Beginning JavaServer Pages*, *Professional Apache Tomcat 5*, *Pro JSP - Third Edition*, *Early Adopter JXTA*, *Professional Jini*, *Beginning J2ME: From Novice to Professional, Third Edition*, and numerous other books. Sing also writes for technical magazines and participates in open source communities. He is an evangelist of the open source, VOIP, and P2P movements. You can reach Sing at westmakaha@yahoo.com.