# Unit xx: Dependency Injection & Inversion of Control

# 1 Spring Basics

## 1.1 Object Coupling Problem

Let’s consider a simple program whose objective is to read data from various data sources, e.g. a file system, a database, an FTP server, and etc. For simplicity, we will start writing a program that reads the data from a file system for now:

public class DataReaderClient {

private FileReader fileReader = null;

private String fileName = "myfile.txt";

public DataReaderClient() {

fileReader = new FileReader(fileName);

}

private String fetchData() {

return fileReader.read();

}

public static void main(String[] args) {

DataReaderClient dataReader = new DataReaderClient();

System.out.println("Got data: "+dataReader.fetchData());

}

}

public class FileReader {

private StringBuilder builder = null;

private Scanner scanner = null;

public FileReader(String fileName) {

scanner = new Scanner(new File(fileName));

builder= new StringBuilder();

}

public String read() {

while (scanner.hasNext()) {

builder.append(scanner.next());

}

return builder.toString();

}

}

The limitation of the above design is that DataReaderClient depends on FileReader’s contract, meaning if FileReader changes, so does the DataReaderClient. If client has already been distributed and used by, say, 1000 users across the globe, you will have fun refactoring the client!

In addition, the DataReaderClient can only read the data from file system. Imagine, one fine morning, your manager asks you to improvise the program to read data from a Database or Socket instead of File! With the current design, it is not possible to incorporate these changes without refactoring the code.

To improve on the existing design, writing your code against interfaces is a very good practice. As long as we keep the interface contract unchanged, the implementation can be modified any number of times without affecting the client. For our data reader program, we create a Reader interface:

public interface Reader {

String read();

}

The next step is to implement this contract. As we have to read the data from different sources, we create respective concrete implementations such as FileReader, DatabaseReader, and FtpReader.

Once you have the XXXReader ready, the next step is to use it in the client program. However, instead of using the concrete class reference, use the interface reference. For example:

public class DataReaderClient {

private Reader reader = null;

public DataReaderClient(Reader reader) {

this.reader = reader;

}

private String fetchData() {

return reader.read();

}

public static void main(String[] args) {

...

}

}

The DataReaderClient does not know where it is fed the data until runtime. The Reader class will only be resolved at runtime using Polymorphism. All we know is that the client can get any of the concrete implementations of Reader interface. The challenge is to provide the appropriate Reader to the client. One way to do this is to create a concrete implementation of Reader in the client program. For example:

public static void main(String[] args) {

Reader reader = new FileReader();

DataReaderClient client = new DataReaderClient(reader);

...

}

Of course, you could swap FileReader with DatabaseReader or FtpReader without much hassle. However, it is still hard wired, and the client will have to know about which Reader it is going to use. In other words, we still have the concrete Reader coupled to the client. Ideally, we should eliminate this coupling as much as possible. The question is how we can provide an instance of Reader without hardwiring?

**A Dependency Injection (DI) or Inversion of Control (IoC) framework can do this job for us, and currently a very popular implementation of such frameworks is Spring.**

## 1.2 Dependency Injection & Inversion of Control

DI and IoC are important software engineering principles, and the terms are often used interchangeably. Essentially, it means that all the dependencies and relationships are created by the DI/IoC container, and then injected into the main program as properties. This is exactly the reverse of usual program creation and hence is called “Inversion of Control”.

Coming back to our Reader program, the solution is to inject a concrete implementation of Reader into the client on demand. Let’s modify the DataReaderClient as below:

public class DataReaderClient {

private ApplicationContext ctx = null;

private Reader reader = null;

public DataReaderClient() {

ctx = new ClasspathXmlApplicationContext(“reader-beans.xml”);

}

public String getData() {

reader = (Reader) ctx.getBean(“fileReader”);

reader.fetchData();

}

public static void main(String[] args) {

DataReaderClient client = new DataReaderClient();

System.out.println(“Data:”+client.getData());

}

}

There are couple of notable things in the client program:

* A new variable referring to ApplicationContext.
* This is then assigned an instance of ClasspathXmlApplicationContext passing an XML file to the constructor of the class of its instantiation.

These are the key to using Spring Framework. The instantiation of the ApplicationContext creates the container that consists of the objects defined in the XML file below:

<bean name="fileReader" class="napier.spring.exercise.FileReader"

<constructor-arg value="src/myfile.txt"/>

</bean>

The purpose of this XML file is to create the respective beans and their relationship. This XML file is then provided to the ApplicationContext instance, which creates a container with these beans and their object graphs along with relationships. The Spring container is simply a holder of the bean instances that were created from the XML file.

To wrap up, here are the things that we have done to make our program work without dependencies using Spring:

* We created a concrete Reader implementation, the FileReader as a simple POJO;
* We created the XML file, reader-beans.xml, to configure this bean;
* We then created a container with this bean in our client that loads it by reading the XML file;
* We queried the container to obtain our bean so we can invoke the respective methods.

Currently the client will always be injected with a type of Reader defined in configuration. One last thing you can do to improve your program is to create a service layer. The ReaderService is an interface between the client and the Readers. It abstracts away the implementation details and the Reader interface from the client. The client will only have knowledge of the service; it will know nothing about where the service is going to return the data.

The first step is to write the service:

public class ReaderService {

private Reader reader = null;

public ReaderService (Reader reader) {

this.reader = reader;

}

private String fetchData() {

return reader.read();

}

}

Then, wire the ReaderService with appropriate Reader in our reader-beans.xml file:

<bean name="readerService"

class="napier.spring.exercise.ReaderService">

<constructor-arg ref="fileReader" />

</bean>

<bean name="fileReader" class="napier.spring.exercise.FileReader"

<constructor-arg value="src/myfile.txt"/>

</bean>

When this config file is read by the Spring’s ApplicationContext, the ReaderService and FileReader beans are instantiated. The modified client that uses ReaderService is given below:

public class DataReaderClient {

private ApplicationContext ctx = null;

private ReaderService service = null;

public DataReaderClient() {

ctx = new ClasspathXmlApplicationContext(“reader-beans.xml”);

}

public String getData() {

service = (ReaderService) ctx.getBean(“readerService”);

service.fetchData();

}

public static void main(String[] args) {

DataReaderClient client = new DataReaderClient();

System.out.println(“Data:”+client.getData());

}

}

The notable thing is that the client will only have knowledge of the service – no Readers whatsoever. If you wish to read data from a database, no code changes are required except config changes as shown below:

<bean name="readerService"

class="napier.spring.exercise.ReaderService">

<property name="reader" ref="databaseReader"/>

<!--

<property name="reader" ref="fileReader"/>

<property name="reader" ref="ftpReader"/>

-->

</bean>

<bean name="databaseReader"

class="napier.spring.exercise.DatabaseReader"

<property name="dataSource" ref="mySqlDataSource" />

</bean>

<bean name="ftpReader" class="napier.spring.exercise.FTPReader"

<property name="ftpHost" value="napier.ac.uk" />

<property name="ftpPort" value="21" />

</bean>

<bean name="fileReader" class="napier.spring.exercise.FileReader"

...

</bean>

We have defined all the Reader beans in the above configuration. The readerService is given a reference to the respective Reader without having to make any code change.

## 1.3 Injection Types

Spring allows to inject the properties via constructors or setters. While both types are equally valid and simple, it’s a matter of personal choice in choosing one over the other.

One advantage to using constructor types over setters is that we do not have to write additional setter code. Having said that, it is not ideal to create constructors with lots of properties as arguments.

The snippets below illustrate the constructor injection method. The basic idea is that the class will have a constructor that takes the arguments, and these arguments are wired via the configuration file:

public class FtpReader implements Reader {

private String ftpHost = null;

private int ftpPort = null;

// Constructor with arguments

public FtpReader(String host, String port) {

this.ftpHost = host;

this.ftpPort = port;

}

...

}

<bean name="ftpReader" class="napier.spring.exercise.FTPReader"

<constructor-arg value="napier.ac.uk" />

<constructor-arg value="21" />

</bean>

<bean name="readerService"

class="napier.spring.exercise.ReaderService">

<constructor-arg ref="ftpReader" />

</bean>

For setter type injection, we have to provide setters and getters on the respective variables. For example:

public class ReaderService {

private Reader reader = null;

public ReaderService() { /\* empty constructor \*/}

public void setReader(Reader reader) {

this.reader = reader;

}

public Reader getReader() {

return reader;

}

}

<bean name="readerService"

class="napier.spring.exercise.ReaderService">

<property name="reader" ref="fileReader" />

</bean>

You can mix and match the injection types, too. For example, the revised FileReader class listed below has a constructor as well as a few other properties.

<bean name="fileReader" class="napier.spring.exercise.FileReader"

<constructor-arg componentName="TradeFileReader" />

<property name="fileName" value="src/myfile.txt" />

</bean>

# 2 Spring Beans

## 2.1 Bean Life-cycle

For Spring, all objects are beans. The life cycle of a bean is easy to understand, yet different from the life cycle exposed in a standard Java application. In a normal Java process, a bean is usually instantiated using a new operator. The Spring Framework does a few more things in addition behind the scenes:

* The framework factory loads the bean definitions and creates the bean.
* The bean is then populated with the properties as declared in the bean definitions.
* If the property is a reference to another bean, that other bean will be created and populated, and the reference is injected prior to injecting it into this bean.
* If your bean implements any of Spring’s interfaces, such as BeanNameAware or BeanFactoryAware, appropriate methods will be called.
* The framework also invokes any BeanPostProcessors associated with your bean for pre-initialsation.
* The init-method, if specified, is invoked on the bean.
* The post-initialisation will be performed if specified on the bean.

Let’s discuss these points in a concrete application scenario.

Look at the following XML code snippet:

<bean name="fileReader" class="napier.spring.exercise.FileReader"

<property name="fileName" value="src/myfile.txt" />

</bean>

When the Spring factory reads the definition, it creates the class using the new operator. After the bean is created, the property fileName is injected. In this case, a setter called setFileName is invoked and given a value of src/myfile.txt as an argument. The bean is now instantiated and ready to be used.

However, if the fileReader bean has a dependency on another bean, the other bean will be created and instantiated first. For example:

<bean name="fileReader" class="napier.spring.exercise.FileReader"

<property name="location" ref="fileLocation" />

</bean>

<bean name="fileLocation" class=" napier.spring.exercise.Location">

<property name="fileName" value="myfile.txt"/>

<property name="filePath" value="src/"/>

</bean>

The order of creation is important for Spring. After digesting the configuration metadata, Spring creates a plan (it allocates certain priorities to each bean) with the order of beans that needs to be created to satisfy dependencies. Hence, the fileLocation object is created first, before the fileReader. If Spring encounters any exception while creating the fileLocation object, it will fail fast and quit. It does not create any further beans and lets the developer know why it won’t progress further.

## 2.2 Method Hooks

When the bean is created, you can ask Spring to invoke a specific method on your bean to initialise. This method provides a chance for your bean to do housekeeping stuff and to do some initialisation, such as creating data structures, creating thread pools, etc. Similar to the initialisation, the Spring framework also invokes a destroy method to clean up before destroying the bean. For example:

<bean name="fileReader" class="napier.spring.exercise.FileReader"

init-method="init" destroy-method="destroy"

</bean>

public class FileReader implements Reader {

private List<Location> locations = null;

public void init(){

locations = new ArrayList<Locations>();

}

public void destroy(){

locations = null;

}

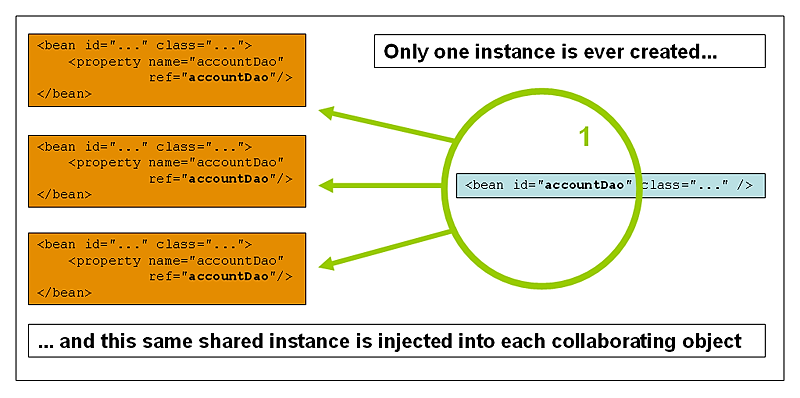
}

## 2.3 Bean Scopes

Spring beans can be defined to be deployed in one of the following scopes:

### 2.3.1 Singleton

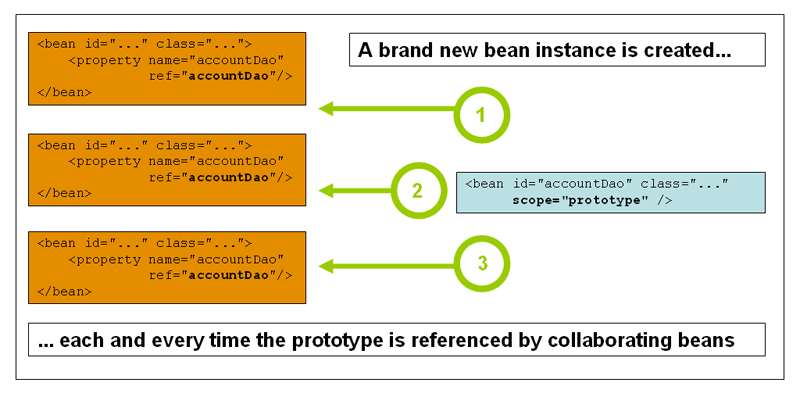
This is the default scope for a Spring bean. A Singleton means that only one shared instance of a bean is managed, and all requests for beans with an id or ids matching that bean definition result in that one specific bean instance being returned by the Spring container. This single instance is stored in a cache of such singleton beans.

****

Spring’s concept of a singleton bean differs from the “Singleton pattern”, which hard-codes the scope of an object such that one and only one instance of a particular class is created ***per Class Loader***. The scope of the Spring singleton is best described as ***per container*** and ***per bean***. This means that if you define one bean for a particular class in a single Spring container, then the Spring container creates one and only one instance of the class defined by that bean definition. To define a bean as a singleton, you would write, for example:

<bean id="fooBean" class="napier.spring.FooBean" scope="singleton" />

### 2.3.2 Prototype



The non-singleton, prototype scope of bean deployment results in the creation of a new bean instance every time a request for that specific bean is made. That is, the bean is injected into another bean or you request it through a getBean() method call on the container. In some respects, the Spring container’s role in regard to a prototype-scoped bean is a replacement for the Java new operator.

When you use singleton-scoped beans with dependencies on prototype beans, be aware that dependencies are resolved at instantiation time. Thus if you dependency-inject a prototype-scoped bean into a singleton-scoped bean, a new prototype bean is instantiated and then dependency-injected into the singleton bean. The prototype instance is the sole instance that is ever supplied to the singleton-scoped bean.

The following example defines a bean as a prototype in XML:

<bean id="fooBean" class="napier.spring.FooBean" scope="prototype" />

### 2.3.3 Request, session, and global session scopes

The request, session, and global session scopes are only available if you use a web-aware Spring ApplicationContext implementation (such as XmlWebApplicationContext). If you use these scopes with regular Spring IoC containers such as the ClassPathXmlApplicationContext, you get an IllegalStateException due to an unknown bean scope.

To support the scoping of beans at the request, session, and global session levels, some minor initial configuration is required before you define your beans. How you accomplish this initial setup depends on your particular Servlet environment.

Consider the following bean definition:

<bean id="login" class="napier.spring.Login" scope="request"/>

The Spring container creates a new instance of the Login class by using the login bean definition for each and every HTTP request. You can change the internal state of the instance that is created as much as you want, because other instances created from the same bean definition will not see these changes in state; they are particular to an individual request. When the request completes processing, the bean that is scoped to the request is discarded.

Consider the following bean definition:

<bean id="cart" class="napier.spring.Cart" scope="session"/>

The Spring container creates a new instance of the Cart class by using the cart bean definition for the lifetime of a single HTTP Session. As with request-scoped beans, you can change the internal state of the instance that is created as much as you want, knowing that other HTTP Session instances that are also using instances created from the same bean definition do not see these changes in state, because they are particular to an individual HTTP Session. When the HTTP Session is eventually discarded, the bean that is scoped to that particular HTTP Session is also discarded.

Consider the following bean definition:

<bean id="cart" class="napier.spring.Cart" scope="globalSession"/>

The global session scope is similar to the standard HTTP Session scope described above, and applies only in the context of portlet-based web applications. The portlet specification defines the notion of a global Session that is shared among all portlets that make up a single portlet web application. Beans defined at the global session scope are scoped to the lifetime of the global portlet Session. If you write a standard Servlet-based web application and you define one or more beans as having global session scope, the standard HTTP Session scope is used, and no error is raised.

## 2.4 Injecting Java Collections

The example code above involves a Java collection, i.e. List<Location> locations. The Spring framework supports injecting values into four major collection types, including List, Set, Map and Properties.

Consider a Customer object, with four collection properties:

public class Customer

{

private List<Object> lists;

private Set<Object> sets;

private Map<Object, Object> maps;

private Properties pros;

...

}

The collections can be injected as below:

<bean id="customerBean" class="napier.spring.exercise.Customer">

<!-- java.util.List -->

<property name="lists">

<list>

<value>listValue1</value>

<ref bean="personBean" />

<bean class="napier.spring.exercise.Person">

<property name="name" value="Bob" />

<property name="address" value="Edinburgh" />

<property name="age" value="28" />

</bean>

</list>

</property>

<!-- java.util.Set -->

<property name="sets">

<set>

<value>setvalue1</value>

<ref bean="personBean" />

</set>

</property>

<!-- java.util.Map -->

<property name="maps">

<map>

<entry key="Key 1" value="1" />

<entry key="Key 2" value-ref="personBean" />

<entry key="Key 3">

<bean class="napier.spring.exercise.Person">

<property name="name" value="Chris" />

<property name="address" value="Dundee" />

<property name="age" value="22" />

</bean>

</entry>

</map>

</property>

<!-- java.util.Properties -->

<property name="pros">

<props>

<prop key="admin">admin@napier.ac.uk</prop>

<prop key="support">support@napier.ac.uk</prop>

</props>

</property>

</bean>

<bean id="personBean" class="napier.spring.exercise.Person">

<property name="name" value="Alice" />

<property name="address" value="Glasgow" />

<property name="age" value="25" />

</bean>

## 2.5 Autowiring

When creating a bean, we used to set the properties of the bean either using property or constructor-arg attributes. However, Spring has an advanced concept of autowiring the relationships and dependencies. This means that you don’t have to explicitly mention the properties and their values, but setting autowire property to a value allows the framework to wire them with appropriate properties. There are fundamentally three variations of autowiring:

### 2.5.1 Autowiring byName

When autowiring byName is enabled, the framework tries to inject the dependencies by matching the names of the beans. You have to set the value of autowire to byName when defining the bean in the config. The container looks at the properties of the respective bean on which autowiring byName is set. It then tries to match with the beans defined by the same name in the config file. If matches are found, it will inject those beans straight away; otherwise, it will throw exceptions.

For example, see the definition of the TradeReceiver class below:

public class TradeReceiver{

private TradePersistor tradePers = null;

private TradeTransformer tradeTrans = null;

//getters & setters ...

}

The TradeReceiver depends on two other beans: the TradePersistor and TradeTransformer. Our usual way is to define all three beans in the XML config file, then pass the references to TradeReceiver. However, with autowiring, you don’t have to go this far. In order to enable autowiring, we should first tell the framework to do so. We use a property called autowire in order to tell the container what beans it should wire automatically. See the config below for all the three beans:

<bean name="tradeReceiver" class="napier.spring.TradeReceiver"

autowire="byName" />

<bean name="tradePers" class="napier.spring.TradePersistor"/>

<bean name="tradeTrans" class="napier.spring.TradeTransformer" />

### 2.5.2 Autowiring byType

Similar to byName, we need to set the autowire property to byType in order to enable this type of autowiring. In this case, instead of looking for a bean with the same names, the container searches for the same types. Taking the same example of TradeReceiver, setting the autowire="byType" tells the container that it should look for a bean of type TradePersistor and another one with a type of TradeTransformer. If the container finds the appropriate types, it will inject them into the bean. However, if it finds more than one bean with the same type defined in the config, a fatal exception is thrown.

### 2.5.3 Autowiring by Constructor

If a bean has a constructor that takes an argument of another bean type, the container looks for that reference and injects it. For example, we define a TradePersistor class with a single constructor that takes a DataSource object:

public class TradePersistor {

public TradePersistor (DataSource datasource){ ... }

}

If we enable autowiring by constructor, the container looks for an object of type DataSource and injects into the TradePersistor bean:

<bean name="tradePersistor" class="napier.spring.TradePersitor"

autowire="constructor" />

### 2.5.4 Mixing Autowiring with Explicit Wiring

You can get the best of both worlds using auto and explicit wiring. Any ambiguities encountered while autowiring can be dealt with using explicit wiring. For example, the TradePersitor can be injected explicitly while the TradeTransformer can be wired automatically using byName variation:

<bean name="tradeReceiver" class="napier.spring.TradeReceiver"

autowire="byName">

<property name="tradePers" ref="tradePers" />

</bean>

<bean name="tradePers" class="napier.spring.TradePersistor" />

<bean name="tradeTrans" class="napier.spring.TradeTransformer" />

# 3 Spring Data Access

## 3.1 Spring Hibernate

As we already know, the Hibernate’s SessionFactory is the key object that must be created before start working with Hibernate. Spring can create this object using its own implementation called LocalSessionFactoryBean. The LocalSessionFactoryBean is declared in the XML file so that the framework can create and inject the bean it into your data access classes. The configuration allows us to set Hibernate properties such as mapping files, dialects, and others.

The LocalSessionFactoryBean has to be wired in with a dataSource reference and hibernateProperties. See the example configuration below:

<bean id="sessionFactory"

class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">

<property name="dataSource" ref="mySqlDataSource" />

<property name="hibernateProperties">

<props>

<prop key="hibernate.show\_sql">false</prop>

<prop key="hibernate.current\_session\_context\_class">

thread

</prop>

<prop key="hibernate.dialect">

org.hibernate.dialect.MySQL5InnoDBDialect

</prop>

<prop key="hibernate.hbm2ddl.auto">true</prop>

</props>

</property>

*<property name="mappingResources">*

*<list>*

*<value>Trade.hbm.xml</value>*

*</list>*

*</property>*

*<property name="annotatedClasses">*

*<list>*

*<value>napier.spring.exercise.Trade</value>*

*</list>*

*</property>*

</bean>

<bean id="mySqlDataSource"

class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close">

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/XXX" />

<property name="username" value="XXX" />

<property name="password" value="XXX" />

</bean>

Now that your SessionFactory has been declared, the next job is to inject this into your DAO. In order to do this, create a TradeDAO, as shown below:

public class TradeDAO {

private SessionFactory sessionFactory = null;

private Session session = null;

public void setSessionFactory(SessionFactory sessionFactory) {

this.sessionFactory = sessionFactory;

}

public SessionFactory getSessionFactory() {

return sessionFactory;

}

private void beginTx() {

session = getSessionFactory().getCurrentSession();

session.beginTransaction();

}

private void commitTx(){

session.getTransaction().commit();

}

public void persist(Trade t){

beginTx();

session.save(t);

commitTx();

System.out.println("Trade successfully persisted");

}

public Trade getTrade(int tradeId) {

String query = "from Trade where id=:tradeId";

beginTx();

Query q = session.createQuery(query)

.setInteger("tradeId", 1);

Trade t = (Trade) q.uniqueResult();

commitTx();

return t;

}

public List<Trade> getAllTrades() {

beginTx();

List<Trade> trades =

session.createQuery("from Trade").list();

commitTx();

return trades;

}

public void delete(Trade t){

beginTx();

session.delete(t);

commitTx();

}

public int deleteTradesByStatus(String status) {

beginTx();

String query = "delete from Trade where status = :status";

Query q = session.createQuery(query)

.setString("status",status)

int tradesDeleted = q.executeUpdate();

commitTx();

return tradesDeleted;

}

...

}

<bean id="tradeDAO" class="napier.spring.exercise.TradeDAO">

<property name="sessionFactory" ref="sessionFactory"/>

</bean>

That’s it – our primary job is done. The final step is to see the TradePersistor class in action to persist the trades to the database. This class delegates the persistence mechanism to TradeDAO:

public class TradePersistor {

private ApplicationContext ctx = null;

private TradeDAO tradeDAO = null;

public TradePersistorUsingDAO() {

ctx = new ClassPathXmlApplicationContext(

"hibernate-beans.xml");

tradeDAO = ctx.getBean("tradeDAO", TradeDAO.class);

}

private void persist(Trade t) {

tradeDAO.persist(t);

System.out.println("Trade persisted:" + t);

}

public static void main(String[] args) {

TradePersistor persistor = new TradePersistor();

persistor.persist(DomainUtil.createDummyTrade());

}

}

You may have noticed that we did not use the Spring framework when making the Hibernate calls. Instead, we used the framework only to inject the respective resources. As explained earlier, Spring’s job more or less is done once the task of instantiating and injecting the appropriate objects, including the SessionFactory, is completed.

Note:  
Refer to Hibernate Exercise 3, Task 5. Discuss the benefit of injecting the SessionFactory object using the Spring framework. Remember, the default bean scope in Spring is Singleton.

## 3.2 Spring JPA

The Spring framework supports the JPA API very similar to support for Hibernate. It allows developers to use plain JPA API in their applications via an injected EntityManager class.

Consider the following TradeDAO class:

public class TradeDAO {

private EntityManagerFactory entityManagerFactory = null;

private EntityManager manager = null;

@PersistenceUnit

public void setEntityManagerFactory(

EntityManagerFactory entityManagerFactory) {

this.entityManagerFactory = entityManagerFactory;

}

public EntityManagerFactory getEntityManagerFactory() {

return entityManagerFactory;

}

...

}

The DAO’s setEntityManagerFactory method is annotated with @PersistenceUnit. Spring understands this annotation and accordingly will inject EntityManagerFactory into the DAO. In fact, the @PersistenceUnit annotation belongs to the JPA specification, which makes our DAO fully JPA-compliant. Spring knows that it has to inject an EntityManagerFactory here, because of the existence of a post processor bean, namely PersistenceAnnotationBeanPostProcessor. This bean is wired into the configuration XML like this:

<bean class="org.springframework.orm.jpa.support

.PersistenceAnnotationBeanPostProcessor" />

<bean id="tradeDao" class="napier.spring.exercise.TradeDAO"/>

<bean id="entityManagerFactory"

class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">

<property name="persistenceUnitName" value="trade-mysql-pu" />

</bean>

Now that you have all the ammunition you need, all you have to do is to execute the functionality on the DAO via a test class. For example:

public class TradePersistorJPA {

public TradePersistorJPA() {

ctx = new ClassPathXmlApplicationContext(

"spring-jpa-config.xml");

dao = ctx.getBean("tradeDao",TradeDAO.class);

}

public void persist(Trade t){

dao.persist(t);

}

...

}

## 3.3 Spring JDO

Similar to EntityManagerFactory in JPA, there’s a factory class in JDO as well: PersistenceManagerFactory. Just as EntityManager was the core of JPA, PersistenceManager is used similarly within the JDO world. Use the factory object to fetch your PersistenceManager and accordingly invoke the relevant data access operations. Spring will help to inject the relevant factories into your application, so you have a handle of PersistenceManager instance.

Again, the DAO is something like this:

public class TradeDAO {

private PersistenceManagerFactory persistenceManagerFactory = null;

public void setPersistenceManagerFactory(PersistenceManagerFactory

persistenceManagerFactory) {

this.persistenceManagerFactory = persistenceManagerFactory;

}

public PersistenceManagerFactory getPersistenceManagerFactory() {

return persistenceManagerFactory;

}

...

}

Unlike Hibernate and JPA, Spring does not provide any wrapper classes for JDO, so we are going to inject the PersistenceManagerFactory implementation from Datanucleus:

<bean id="persistenceManagerFactory"

class="org.datanucleus.api.jdo.JDOPersistenceManagerFactory">

<property name="connectionFactory" ref="mySqlDataSource" />

</bean>

<bean id="mySqlDataSource"

class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close">

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/XXX" />

<property name="username" value="XXX" />

<property name="password" value="XXX" />

</bean>

<bean id="tradeDao" class="napier.spring.exercise.TradeDAO">

<property name="persistenceManagerFactory"

ref="persistenceManagerFactory" />

</bean>