Internet Computing Workshop

Part3:
Spring
Announcement

Hibernate exam will be on Wed 17 Feb 2-5PM

You can bring your notes
You will have access to API
But, you are not allowed to go online or use email or any sort of chat software
Contents

• Why Spring?
  • Description of the problem
• Overall view
• Inspecting a sample of an Spring-Hibernate Application → List of Questions!
• Detailed study of Spring:
  • Dependency Injection
  • Beans and their wiring
  • Aspect oriented Programming
  • Interaction with db
  • Transactions
• Security
Why Spring?

You have learnt Hibernate. It allows ORM + nice functionalities such as Lazy loading, Eager fetching and Cascading.
No matter what technology, we need to access data!
No matter what technology, we need to access data!

1. We tend to re-execute the same types of query in different part of the code
2. We may need to move to a new technology or a newer version of a technology

Spring proposes Data Access Object (DAO)
DAO

DAO allows reading and writing data to database. Application delegates task of access to data to DAO

What is an interface?
Advantages of using DAO for data

One implementation and reuse
• Less code- cheaper, less chance of error, less testing effort

Single point of change
• If migrating to new technology, version, implementation

Questions
• How does the interaction happen?
• Why interfaces?
Why Spring? Simpler Exceptions

In accessing data various things can fail. SQL exception can be thrown if:

■ application is unable to connect to the database.
■ query being performed has errors in its syntax.
■ tables and/or columns referred to in the query do not exist.
■ An attempt is made to insert or update values that violate a database constraint.
Exceptions

Some exceptions if caught are useful, such as rollback
But most are not!
If a connection is failed- no matter what the reason, we can’t do much 😞
Spring promotes the use of unchecked exceptions, which are independent of technology!
Why Spring? Reuse Business Logic

Apart from DAO for data we have “Application DAO”

Business functionality which is reused
Why spring? Decupled applications

Build, Test, Modify one “part” of the application independent of another parts

• Dependency Injection
• Aspect oriented Programming

This although fundamental, we will look at briefly!
A Birdseye view of Architecture via Spring and Hibernate

Note the separation of duties!

- **Presentation tier**
  - JSP, Wicket...

- **Business Services Layer**
  - Independent of persistence layer

- **DAO interface for persistent operations**

- **Persistent domain model**
  - POJO

- **Hibernate ORM**
  - Implementation using ORM

- **DAO interface**

- **ORM**

- **Persistent domain model**

- **POJO**
Description of the example

Consider a simplified e-business system for the bookshop [download the code from www.cs.bham.ac.uk/~xbxb/courses/icw10/part3/shop-spr-hib.jar]

1) create/drop database
2) list-customers
3) print-customer details given the email
4) get a list of Books

OK, all console input! We have Model and Controller- view is missing!... wicket
Description of the example

ehcache.xml
-log4j.properties
spring-hib.xml
-shop/
 |- dao/  DAO for data access
 |- model/  POJO models
 |- usecases/  DAO for Business
 |- Main.java  logic (controller)
Recap

• Why Spring?
  • Description of the problem
• Overall view
• Inspecting a sample of an Spring-Hibernate Application → List of Questions!

**Now you know the general idea of Spring!**

• Detailed study of Spring:
  • Dependency Injection
  • Beans and their wiring
  • Aspect oriented Programming
  • Interaction with db
  • Transactions

• Security
Understanding Dependency injection

1st, let me remember Beans:
Also called components and sometimes objects.
Beans follow three main convention:
1) Serializable
2) Constructor with no parameters
3) Can be accessed via Mutators and Accessors
Hibernate entities!
Understanding Dependency Injection

Also called Inversion of Control (IoC), see http://martinfowler.com/bliki/InversionOfControl.html.

Suppose two classes as follows

```java
public class Investor {
    String name;
    int owns;
    GoodInvestment investment = new GoodInvestment();

    int returnedMoney() {
        return investment.calculate(owns);
    }
}
```

...
public class GoodInvestment {
    public int calculate(int owns) {
        return owns*2;
    }
}

What if we want to use BetterInvestment()? what do we need to change?

Tight couplings: One object is responsible for the other object.
Solution: Dependency injection
1. Create interfaces to hide the details of implementation behind it
2. Provide different implementations (GoodImplementation(), Better...)
3. Instead of making the class (Investor) responsible for getting the Investment, assign the Investment to the Investor (this is done via wiring: an xml file)
4. Use Spring container to implement the wiring by wiring the components together.
Walk through the code to see

For the source see:

Do you see the clear decoupling- you need to ponder a bit to work it out.

draw a picture!
Spring Container

All your objects live in the Spring container, which manages your objects:

• create the objects,
• Wire them together,
• configure the objects
• manage their complete lifecycle

Container uses Dependency Injection to manage the components.
Spring Container

Spring has multiple implementation of the container which are either a:

**Bean factory**: implementation of `org.springframework.beans.factory.BeanFactory` interface (used in the example)

**Application context**: implementation of `org.springframework.context.ApplicationContext` interface

Application context has richer set of functionality!
BeanFactory

- Creates and dispenses fully configured beans
- Keep the track of beans and all instantiated objects

Most commonly used implementation is via XML:

```java
org.springframework.beans.factory.
xml.XmlBeanFactory

Then

BeanFactory factory = new
XmlBeanFactory
(new
FileSystemResource("foo.xml"));
```
BeanFactory

- Creates and dispenses fully configured beans
  Most commonly used implementation is via XML:
  `org.springframework.beans.factory.xml.XmlBeanFactory`

Then, reading conf. definition from an XML file
BeanFactory factory = new
`XmlBeanFactory` (new
`FileSystemResource` ("foo.xml"));
This only reads the beans definition
BeanFactory loads beans “lazily”, i.e. instantiated when needed:

Investor investor = (Investor)
    factory.getBean("Achim");

Note, apart from FileSystemResource, which we will be using, there are other implementations of Resource for taking input from a URL, InputStream, Classpath, servlet context, … see the API
ApplicationContext

Enhances functionality of BeanFactory with a generic way to load file resources, can publish events to beans that are registered as listeners and some more ...

Three implementations of ApplicationContext:

ClassPathXmlApplicationContext: Loads context definitions from an XML file in the classpath

FileSystemXmlApplicationContext: Loads context def. from an XML file in the file system.

XmlWebApplicationContext: Loads context definitions from an XML file contained within a web application
ApplicationContext

Usage similar to BeanFactory:
ApplicationContext contxt= new
FileSystemXmlApplicationContext
("investment.xml");

or
ApplicationContext contxt= new
ClassPathXmlApplicationContext
("investment.xml");

ClassPath... looks everywhere but
FileSystem... looks for a specific location
Declaring Beans

After creating an interface and providing an implementation, you can declare a bean in configuration file:

```xml
<bean id="goodinv" class="withDependencyInjection.GoodInvestment"/>
```

Create a component of type `GoodInvestment` and call it `goodinv`.

When instantiated uses default constructor!
Declaring Beans

To use other constructors set the value of 
<constructor-arg value="sth" />

Which is better? Constructor or not?

- Constructor injection enforces a strong dependency contract- must declare all!
- Less code (setter/getter use)

On the other hand

- Keeping the track of parameter purposes of the same type can be hard
- In inheritance must use super() for parents param.
property

Is used for:

- setting value of parameters
  
  `<property name="owns" value="50"/>
  
- referencing other beans
  
  `<property name="investment" ref="goodinv"/>


Properties (parameters) with multiple values

Suppose in your implementation you have a property of type `Collection` such as

```java
private Collection<Investment> investments;
```

Then configuration element (XML file) becomes:

```xml
<property name="investments">
  <list>
    <ref bean="goodinvestment" />
    <ref bean="betterinvestment" />
    <ref bean="prudentinvestment" />
  </list>
</property>
```
Wiring of different types of Collections

Implementation via `java.util.Collection`

- `<list>` a list of values, duplicates allowed.
- `<set>` a set of values, no duplicates

Implementation via `java.util.Map`

- `<map>` a collection of name-value pairs where name and value can be of any type

Implementation via `java.util.Properties`

- `<props>` a collection of name-value pairs where the name & value are both `String`

Check the API and manual.
More advance wiring methods

`autowire` allows ref to beans of the same Name (or ID), type or constructor, or autodetect

(autodetect will attempt to wire by Constructor and if not by type)

You can manually initialise, create, destroy, … beans. **We will not be using these feature.**

For details see the manual/API.
Recap

• Why Spring?
  • Description of the problem
• Overall view
• Inspecting a sample of an Spring-Hibernate Application → List of Questions!
• Detailed study of Spring:
  • Dependency Injection
  • Beans and their wiring
  **Now you know about beans and their wiring**
  • Aspect oriented Programming
  • Interaction with db
  • Transactions

• Security
Aspect oriented Programming (AoP)

Contents

• Why AoP?
• Example.
• Aspect terminology
• How does it work?
• Spring’s support for AoP
• Advice: before, after and around…
• Classic support vs. XML configuration
Why AoP?

Spring relies on two sets of techniques
1) Using Beans, wiring them and dependency injection
But some of the properties that we wish to decouple can not be represented as beans gracefully!
Consider logging
It is what is known as a “Crosscutting concern”
Every beans uses it but no beans own it!
Why AoP?

Other examples of crosscutting concerns are: security, transactions, optimisation...

Back to example of Logs:
Logs can be implemented by inheritance and delegation.

What is delegation?

Clear advantage: functionality of log is kept in one place

But: your business logic has nothing to do with logging?!?
Aspect oriented Programming allows you
1) define crosscutting concerns in one place as an object called aspect.
2) declaratively specify when and where the functionality should be applied

Hence
• You modify them in one place (easier and cheaper maintenance)
• Reduce the code at the business logic
A logging example

Let us look at some code:

Suppose we wish to print messages before and after the execution of a given method (target)

Pay attention that Helloworld.java has NO information about the Aspect!
Before we can discuss AoP you need to know the AoP’s lingo 😊

**Advice:** describes

1) The job to be applied

2) The place that it should be applied:
   - Before/after/both invoking a method
   - method throws an exception?
Terminologies

Joinpoint:
point in the execution of the application where an aspect can be plugged in:
• point could be a method being called
• an exception being thrown
• a field being modified (Not supported directly, you must use AspectJ)
Pointcuts specify *where* within a joint points the advice should be woven:

- Class name
- Method name

or

- regular expressions that define a matching class or method name
Terminologies

Introduction:
Allows adding new methods or attributes to existing classes

Target:
the object that is being advised.
AoP results in Target object
1) Having less code
2) Focused on the application logic
Terminologies

object created after applying advice to the target object.

Creation of proxy is transparent: to other objects target object and the proxy created after the weaving are the same!

=> The rest of application remains unchanged.
A few words on how does it work

Underlying idea: weaving
Applying aspects to a target object to create a new, proxied object at
Compile time: Weaving compilers (AspectJ)
Classload time: woven when class is loaded (AspectJ5 and above)
Runtime: weaving during the execution time-no complier needed (Spring)
A few words on how does it work

1. From the aspects proxy objects are created
2. Proxy intercepts the method calls to the target object
3. The proxy performs the business logic for the advice (for example logging)
4. Then the actual method calls to the target happens

Proxy is created lazily when needed by the BeanFactory
Three points to remember

1) If the target object implements an interface, Spring uses reflection API. IF NOT YOU MUST ADD a CGLIB library to PATH. Better to let obj. implement an interface.

2) Avoid `Final` methods! (to allow spring override them)

3) Spring only supports `method` joinpoints (not attributes) - remember we are not using an aspect compiler.

AspectJ can be used in Spring- not covered here.
Four methods of doing AoP in Spring

1. Classic proxy based method (Java-based implements one of aop interfaces to produce an advice)
   Example: logfile example earlier slides

2. Pure-POJO aspects (using XML config file)
   shop-hbr.spring.jar example

*We will use this in transactions mostly.*

There are two other methods using AspectJ, which we will not study in this course
Implement one of the following interfaces.

Around a method:
org.aopalliance.intercept.MethodInterceptor

Before a method call:
org.springframework.aop.MethodBeforeAdvice

After-returning from a method call:
org.springframework.aop.AfterReturningAdvice

After-throwing an exception:
org.springframework.aop.ThrowsAdvice
Around a method

Around a method is a combination of the other three.

Interface `MethodInterceptor` requires implementation of only one method `invoke()` which has a try/catch and three parts:

1) The code before the target method
2) Then `proceed()` is called, remember `Object obj = methodInvocation.proceed();`
3) The code after the target
Around a method

Then `PerformanceException` is caught.

--

We can get away with using “around” always! For details of the other three, see the Spring manual.
Declaring pure-POJO aspects

Declare Aop information in XML config. files via the following tags:

- `<aop:config>` top-level element containing all other elements
- `<aop:aspect>` defines an aspect.
- `<aop:pointcut>` defines pointcuts

Also

- `<aop:after>`
- `<aop:after-returning>`
- `<aop:after-throwing>`
- `<aop:around>`
Sample code

In shop-spring-bib.xml we have:

```xml
<aop:config>
  <aop:pointcut
    id="shopServiceOperation"
    expression="execution
                  (*shop.usecases..*(..))"/>
  <aop:advisor advice-ref="txAdvice"
                pointcutref="shopServiceOperation"
                />
</aop:config>

We will study these in the transactions section
Recap

• Why Spring?
  • Description of the problem

• Overall view

• Inspecting a sample of an Spring-Hibernate Application → List of Questions!

• Detailed study of Spring:
  • Dependency Injection
  • Beans and their wiring
  • Aspect oriented Programming

  **Now you know all about technologies underlying Spring**
  • Interaction with db
  • Transactions

• Security
Back to start of Lecture: A view of Architecture via Spring and Hibernate

Note the separation of duties!

Presentation tier
JSP, Wicket…

Business Services
Layer- independent
of persistence layer

Next we look at database access.
Data Access Object (DAO)

Service objects accesses data through a DAO interface

Advantages:
1. Better testing (use dummy implementation for DAO)
2. Different DB technologies can be used with minimal changes (Services and DB are decoupled)
Data Access Object (DAO)

View code now
- Interfaces in `shop.dao` directory - inspect
- **DAO implementations** in Hibernate directory
- For a JDBC implementation we require a JDBC directory

Now we will focus on DAO implementation
Data Access Object (DAO)

look at the code for `BookDAOImpl` in hibernate folder.

Apart from the following bullet points, the rest is hibernate code—what is going on?

- `getHibernateTemplate()`
- `extends HibernateDaoSupport`
- `HibernateCallback()`
- `doInHibernate(Session session)`

Let us look at DAO implementation in details
DAO Impl.: **Template & callback**

In Spring data access process has two parts:
1. Templates (JDBC, Hibernate, …)
2. Callbacks //What is a callback?

Look at **HibernateCallback Object**
DAO Impl.: **Template & callback**

Callback: technology dependent- creating statements, binding parameters ....

Templates: implements transaction control, managing of resources, and handling exceptions via `JdbcTemplate` or `HibernateTemplate` ...

Callback part is a snippet of Hibernate code, but how does Template works?
how does Template works?

Templates require following configurations:

1) **Configure data source** (database, password, username, …)

2) **configuring Hibernate** (sessions, mapping files, …)

3) **extending DAO support classes** (remember bullet point: `extends HibernateDaoSupport`)

Next we will look at the above three.
1) Configure data source

This will configure database, username, password, arguments about pool size,…

There are three methods, we use Database Connection Pools (DBCP)

See [jakarta.apache.org/commons/dbcp](http://jakarta.apache.org/commons/dbcp)

Look at Lines 16-32 of the boilerplate code at `shop-spring-hib.xml`

Pay attention to wiring of password by "ref=" to `shop.Main.getPassword()`. For password setting with value use

```xml
<property name="password" value=""/>
```
2) configuring Hibernate part

Remember in Hibernate save, update, delete,… was done through Session which was obtained through a SessionFactory:

Lines 54-56:

```xml
<bean id="hibernateTemplate"
  class="org.springframework.orm.hibernate3.HibernateTemplate">
  <property name="sessionFactory"
    ref="sessionFactory"/>
</bean>
```

64
Hibernate Templates: `sessionFactory`

wires to `LocalSessionFactoryBean`

\[\text{org.springframework.orm.hibernate3.LocalSessionFactoryBean}\]

Declare

1) `dataSource` (see earlier slider)
2) List of Hibernate mappingResources
3) Dialect of hibernate

See the boilerplated code at line 35-52
3) extending DAO support classes

It is possible to manually wire the beans, but it is easier that “Dao Impl.” classes extend HibernateDaoSupport

(\texttt{org.springframework.orm.hibernate3}.support.HibernateDaoSupport)
3) extending DAO support classes

Call `getHibernateTemplate()` method and then invoke Hibernate methods such as

- `execute()`
- `save()`
- `saveOrUpdate()`
- ...

(see code for `BookDAOImpl.java`)
A few words on Exceptions

Remember we argued that: although some exceptions if caught are useful, such as rollback … But most are not!

If a connection is failed- no matter what the reason, we can’t do much 😞

Spring has a large set of exceptions for data access which are independent from the technology (Hibernate, JDBC,…)

In addition you can leave the exceptions unchecked! Spring will handle them for you. But you must use Templates.
Samples of exceptions

CannotAcquireLockException
CannotSerializeTransactionException
CleanupFailureDataAccessException
ConcurrencyFailureException
DataAccessException
DataAccessResourceFailureException
DataIntegrityViolationException
DataRetrievalFailureException
DeadlockLoserDataAccessException
EmptyResultDataAccessException

...some more

...Very self explanatory!
Transactions

are handled through

    HibernateTransactionManager bean:

<bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory"
        ref="sessionFactory"/>
</bean>

Wiring to sessionFactory require
Programming transactions

It is possible to directly program transactions using `TransactionCallback`, similar to access to db.

However, we make use of AoP for implementing transactions *declaratively* in configuration files.
Programming transactions

Introduce one or more advice bean

\[ <tx:advice> \]

1. Wiring to HibernateTransactionManager bean

2. Setting parameters within \[ <tx:attributes> \] identifying methods (\[ tx:method \]) that the aspect applies to and \textit{transition attributes}

Note: default behaviour is specified in \[ <aop:config> \]

See lines 98-106
Transition Attributes

Within `tx:method` the following 6 arguments can be set:

1. **propagation**
   - REQUIRED If an existing transaction is in progress, run the method within it, otherwise, create a new transaction
   - REQUIRES_NEW the method must run within its own transaction. A new transaction is started and if an existing transaction is in progress.

For further choices see the API
Transition Attributes

2. isolation: you can specify if the transaction is SERIALIZABLE, REPEATABLE_READ, READ_COMMMITTED, as we discussed in transaction lecture.

3. read-only: no synchronisation until the end of the transaction (data store can optimise)

4. timeout (set a timeout for transaction)

5. rollback-for: lists checked exceptions for which a transaction should be rolled back and not committed

6. no-rollback-for: similar to 5…
Next lecture

Security in Spring
Announcements

NakedObject framework seminar
Assessment for Spring