



Module 17: Spring and Hibernate

CS544: Enterprise Architecture



Spring and Hibernate

- In this module we will first do a basic combination featuring:
 - Single config file for both Spring and Hibernate
 - Using Hibernate with Spring DI
- After that we will also apply Spring Declarative Transactions to our combined project
 - Change from BMT to CMT
 - Important configuration change to make this work



Spring and Hibernate:

CONFIGURATION & DI



Combined Configuration

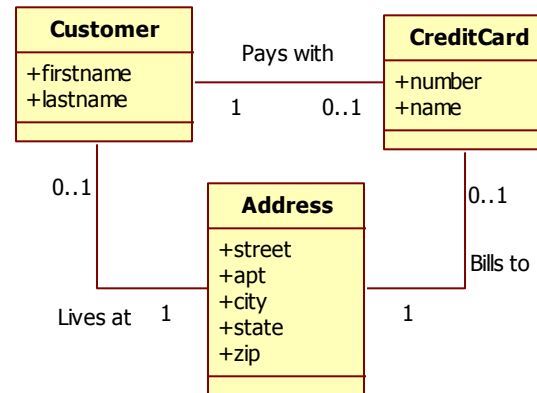
- Hibernate can be configured inside Spring
 - One config file for both Spring and Hibernate
 - SessionFactory becomes a Spring Bean singleton
 - Project can then use full Spring feature set
 - Dependency Injection, AOP, Spring Templates
- No longer need HibernateUtil
 - SessionFactory is created by Spring on startup
 - SessionFactory can easily be retrieved from Spring



Example Application

Application Layers

- View Layer:
 - addCustomer.jsp, customer.jsp, customers.jsp, updaCustomer.jsp, error.jsp, OpenSessionFilter
- Control Layer:
 - ViewAllCustomers, ViewCustomer, AddCustomer, ViewUpdCustomer, UpdCustomer
- Service Layer:
 - Customer Service
- Business Layer:
 - Customer, Address, CreditCard
- Persistence Layer:
 - CustomerDAO, AddressDAO, CreditCardDAO



HibernateUtil class
no longer included



Spring Configuration 1/2

```
<beans ...>
```

```
<bean id="customerService" class="example.service.CustomerService">
```

```
<property name="addressDAO" ref="addressDao" />
```

```
<property name="creditCardDAO" ref="creditCardDao" />
```

```
<property name="customerDAO" ref="customerDao" />
```

```
</bean>
```

```
<bean id="addressDao" class="example.dao.AddressDAO">
```

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

```
</bean>
```

```
<bean id="creditCardDao" class="example.dao.CreditCardDAO">
```

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

```
</bean>
```

```
<bean id="customerDao" class="example.dao.CustomerDAO">
```

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

```
</bean>
```

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
```

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

```
<property name="url" value="jdbc:mysql://localhost/cs544" />
```

```
<property name="username" value="root" />
```

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

```
</bean>
```

```
...
```

DAO and Service classes become Spring Beans allowing us to use Spring Dependency Injection

Database connection is configured in Spring



Spring Configuration 2/2

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="hibernateProperties" ref="hibernateProperties" />
  <property name="annotatedClasses">
    <list>
      <value>example.domain.Address</value>
      <value>example.domain.CreditCard</value>
      <value>example.domain.Customer</value>
    </list>
  </property>
</bean>

<bean id="hibernateProperties"
      class="org.springframework.beans.factory.config.PropertiesFactoryBean">
  <property name="properties">
    <props>
      <prop key="hibernate.hbm2ddl.auto">create</prop>
      <prop key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>
      <prop key="connection.pool.size">1</prop>
      <prop key="hibernate.show_sql">true</prop>
      <prop key="hibernate.current_session_context_class">thread</prop>
    </props>
  </property>
</bean>

</beans>
```

Basic Hibernate Configuration

You can also specify:
<property name="packagesToScan"
value="example.domain" />

Additional Hibernate Property settings



DAO

■ Receive the SessionFactory through DI

```
public class AddressDAO {  
    private SessionFactory sf;  
  
    public void setSessionFactory(SessionFactory sf) {  
        this.sf = sf;  
    }  
  
    public void create(Address addr) {  
        sf.getCurrentSession().persist(addr);  
    }  
  
    public Address get(int id) {  
        return (Address) sf.getCurrentSession().get(Address.class, id);  
    }  
  
    public void update(Address addr) {  
        sf.getCurrentSession().saveOrUpdate(addr);  
    }  
  
    public void delete(Address addr) {  
        sf.getCurrentSession().delete(addr);  
    }  
}
```

No longer gets the sessionFactory
from HibernateUtil

Spring automatically sets the sf
through this setter method (DI)

Methods are exactly the same



Service

- Uses Spring Dependency Injection

```
public class CustomerService {  
    private CustomerDAO customerDao;  
    private AddressDAO addressDao;  
    private CreditCardDAO ccDao;
```

No longer creates DAO object references

```
    public void setCustomerDAO(CustomerDAO customerDao) {  
        this.customerDao = customerDao;  
    }
```

```
    public void setAddressDAO(AddressDAO addressDao) {  
        this.addressDao = addressDao;  
    }
```

Instead has DAO objects set by Spring through setter methods

```
    public void setCreditCardDAO(CreditCardDAO ccDao) {  
        this.ccDao = ccDao;  
    }
```

```
    ...
```



OpenSessionInView Filter

```
public class OpenSessionInView implements Filter {
    private SessionFactory sf;

    public void init(FilterConfig config) throws ServletException {
        ServletContext context = config.getServletContext();
        WebApplicationContext applicationContext =
            WebApplicationContextUtils.getWebApplicationContext(context);
        sf = applicationContext.getBean("sessionFactory", SessionFactory.class);
    }
    public void destroy() {}

    public void doFilter(ServletRequest req, ServletResponse resp,
        FilterChain chain) throws IOException, ServletException {
        Transaction tx = null;
        try {
            tx = sf.getCurrentSession().beginTransaction();
            chain.doFilter(req, resp);
            tx.commit();
        } catch (RuntimeException ex) {
            try {
                ex.printStackTrace();
                tx.rollback();
            } catch (RuntimeException rbEx) {
                System.out.println("Could not rollback transaction " + rbEx);
                rbEx.printStackTrace();
            }
            throw ex;
        }
    }
}
```

Gets SessionFactory
from Spring instead of
HibernateUtil class



Controller

```
public class ViewCustomer extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        int custId = Integer.parseInt(req.getParameter("custId"));

        // get customerService bean from spring
        ServletContext context = getServletContext();
        WebApplicationContext applicationContext =
            WebApplicationContextUtils.getWebApplicationContext(context);
        CustomerService custServ = applicationContext.getBean(
            "customerService", CustomerService.class);

        // make customer available in request, for view rendering
        Customer cust = custServ.getCust(custId);
        req.setAttribute("cust", cust);

        // forward to view customer page
        req.getRequestDispatcher("customer.jsp").forward(req, resp);
    }
}
```

Get customerService from Spring instead of creating a new object



Web.xml

```
<web-app>
...

<filter>
  <filter-name>OpenSessionInView</filter-name>
  <filter-class>example.filter.OpenSessionInView</filter-class>
</filter>

<filter-mapping>
  <filter-name>OpenSessionInView</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/springconfig.xml</param-value>
</context-param>
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>

...

</web-app>
```

Same as before, our filter

Listener starts the spring context needed by servlets and our custom filter



Spring and Hibernate:

DECLARATIVE TRANSACTIONS (CMT)



Spring and Hibernate

- So far we've only shown basic integration between Spring and Hibernate
 - Our project still has the same transaction issues
- We haven't yet shown how Spring can really enhance our project
 - Spring managed SessionFactory was a nice bonus
 - Spring Transaction Demarcation will allow us to solve our transaction issues



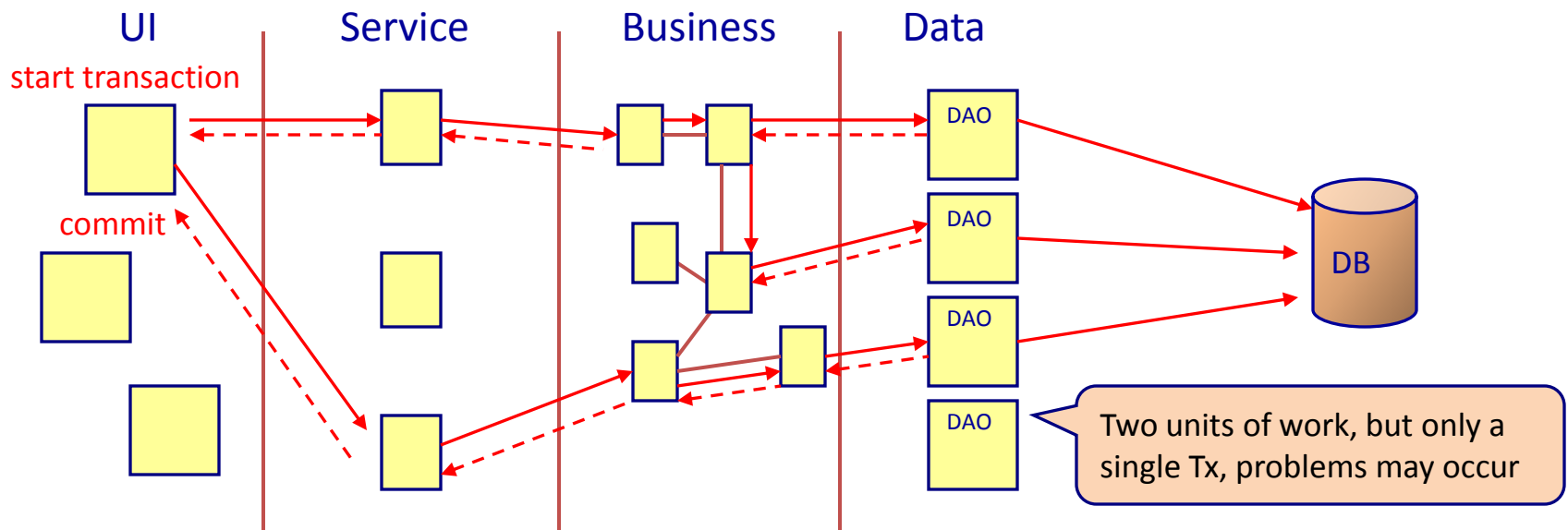
Spring Transaction Support

- Spring is not a transaction manager
 - We still need a transaction manager
 - JDBC transaction manager
 - Hibernate transaction manager
 - XA transaction manger (JTA)
- Spring provides an abstraction for transaction management
 - You declare how the transactions should be managed
 - Spring works with the underlying transaction manger



UI Transaction

- Because of the OpenSessionInView filter our transaction was demarcated in the UI layer
 - Making it possible for a single transaction to span more than one unit of work





Spring Transactions

- Spring allows us to declaratively specify any transactional requirements for a method
 - Letting us specify that **Service level methods** should always **execute in their own Transaction**
 - Thus ensuring that transactions only span a single unit of work, fixing our problem
 - And allowing us to specify that **DAO methods** always already **expect an existing transaction**
 - Ensuring that they are never called without a transaction



Web.xml

```
<web-app>
...

<filter>
  <filter-name>SpringOpenSessionInViewFilter</filter-name>
  <filter-class>
    org.springframework.orm.hibernate3.support.OpenSessionInViewFilter
  </filter-class>
</filter>

<filter-mapping>
  <filter-name>SpringOpenSessionInViewFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/springconfig.xml</param-value>
</context-param>
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>

...

</web-app>
```

Use Spring's open session in view filter instead of our own

Listener starts the spring context needed by the filter



Service 1/2

```
public class CustomerService {  
    private CustomerDAO customerDao;  
    private AddressDAO addressDao;  
    private CreditCardDAO ccDao;  
  
    public CustomerService() {}  
    public void setCustomerDAO(CustomerDAO customerDao) { this.customerDao = customerDao; }  
    public void setAddressDAO(AddressDAO addressDao) { this.addressDao = addressDao; }  
    public void setCreditCardDAO(CreditCardDAO ccDao) { this.ccDao = ccDao; }  
  
    @Transactional(propagation=Propagation.REQUIRES_NEW)  
    public void addNewCustomer(Customer cust, Address shipAddr, CreditCard cc,  
        Address billAddr) {  
  
        cc.setAddress(billAddr);  
        cust.setShipAddress(shipAddr);  
        cust.setCreditCard(cc);  
  
        addressDao.create(shipAddr);  
        addressDao.create(billAddr);  
        ccDao.create(cc);  
        customerDao.create(cust);  
    }  
  
    ...  
}
```

No default set on the class level

Setter methods have no transactional requirements

All other service level methods are annotated to require a new transaction



Service 2/2

...

```
@Transactional(propagation=Propagation.REQUIRES_NEW, readOnly=true)
public List<Customer> getAll() {
    return customerDao.getAll();
}
```

Methods that only retrieve
can also set to readOnly

```
@Transactional(propagation=Propagation.REQUIRES_NEW, readOnly=true)
public Customer getCust(int custId) {
    return customerDao.get(custId);
}
```

```
@Transactional(propagation=Propagation.REQUIRES_NEW)
public void updCustomer(Customer cust, Address shipAddr, CreditCard cc,
    Address billAddr) {
    cc.setAddress(billAddr);
    cust.setShipAddress(shipAddr);
    cust.setCreditCard(cc);

    addressDao.update(billAddr);
    addressDao.update(shipAddr);
    ccDao.update(cc);
    customerDao.update(cust);
}
}
```



DAO

```
@Transactional(propagation=Propagation.MANDATORY)
public class AddressDAO {
    private SessionFactory sf;

    @Transactional(propagation=Propagation.SUPPORTS)
    public void setSessionFactory(SessionFactory sf) {
        this.sf = sf;
    }

    public void create(Address addr) {
        sf.getCurrentSession().persist(addr);
    }

    public Address get(int id) {
        return (Address) sf.getCurrentSession().get(Address.class, id);
    }

    public void update(Address addr) {
        sf.getCurrentSession().saveOrUpdate(addr);
    }

    public void delete(Address addr) {
        sf.getCurrentSession().delete(addr);
    }
}
```

A transaction has to exist (Mandatory) for every method called on this class

Except for setSessionFactory() it can run with or without



Springconfig.xml

...

```
<bean id="hibernateProperties"
      class="org.springframework.beans.factory.config.PropertiesFactoryBean">
  <property name="properties">
    <props>
      <prop key="hibernate.hbm2ddl.auto">create</prop>
      <prop key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>
      <prop key="connection.pool.size">1</prop>
      <prop key="hibernate.show_sql">true</prop>

      <!-- Let spring be the session context class
      <prop key="hibernate.current_session_context_class">thread</prop>
      -->
    </props>
  </property>
</bean>
```

Important: remove thread local, to let Spring manage the TX and the session

```
<tx:annotation-driven transaction-manager="txManager" />
<bean id="txManager" class="org.springframework.orm.hibernate4.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory"/>
</bean>

</beans>
```

Add configuration for annotation based transaction demarcation

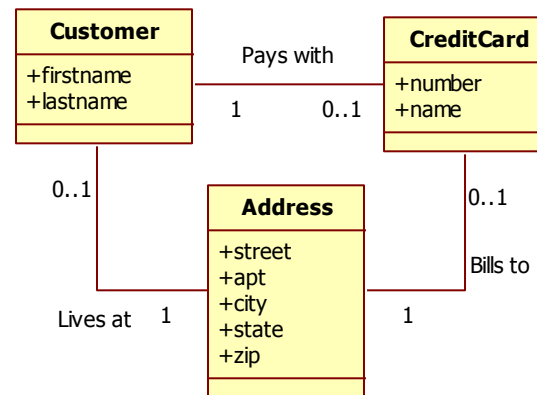


Example Application

Application Layers

- View Layer:
 - addCustomer.jsp, customer.jsp, customers.jsp, updaCustomer.jsp, error.jsp,
- Control Layer:
 - ViewAllCustomers, ViewCustomer, AddCustomer, ViewUpdCustomer, UpdCustomer
- Service Layer:
 - Customer Service
- Business Layer:
 - Customer, Address, CreditCard
- Persistence Layer:
 - CustomerDAO, AddressDAO, CreditCardDAO

OpenSessionFilter class
no longer needed



Our application now just contains code
related to our application, our previous
technology helpers are provided by Spring



Active Learning

- Why don't we need HibernateUtil when using Spring?
- What are the advantages of Spring's OpenSessionInView filter?



Module Summary

- In this module we saw how Spring can add to our Hibernate project
 - Spring and Hibernate are easy to configure together
 - Doing so also removes our need for HibernateUtil
- Spring Declarative Transaction Demarcation is powerful and easy to use feature
 - Simply specify the transactional requirements
 - Cleanly provides needed transactional boundaries
 - Switch to Spring's `OpenSessionInViewFilter`
 - Does create minor Spring Dependency