

Integrando o Hibernate com o Spring

O **Hibernate** é uma solução open-source para Mapeamento Objeto/Relacional (ORM). ORM é uma técnica de mapeamento que consistem em mapear um modelo de Objetos para um modelo Relacional (usualmente representado por uma base de dados SQL). O Hibernate foi criado em meados de 2001 por **Gavin King** e outros desenvolvedores. Desde então, o Hibernate vem se tornando um popular framework de persistência na comunidade Java. Ele se tornou tão popular de fato, que as próximas versões do EJB e do JDO estarão usando Hibernate como referência. As razões para esta popularidade são devido a ótima documentação, facilidade de uso, exceletes recursos e um esperto gerenciamento de projeto.

Hibernate libera você de codificar manualmente o JDBC. Melhor do que usar SQL e JDBC, você pode usar domínios de objetos (normalmente POJOs) e de forma simples criar arquivos de mapeamento baseados em XML. Estes arquivos indicam quais campos (em um objeto) serão mapeados para suas respectivas colunas (na tabela). O Hibernate possui uma poderosa linguagem de pesquisa chamada Hibernate Query Language (HQL). Esta linguagem permite a você escrever SQL, mas também usar semânticas orientadas a objeto. Uma das melhores partes sobre essa linguagem de pesquisa é que você pode literalmente adivinhá-la.

A interface **Session** do Hibernate é similar a uma conexão com o banco de dados, ela tem de ser aberta e fechada nos tempos apropriados para prevenir erros e leaks de memória. Na minha opinião, a maior vantagem de usar Spring com Hibernate é que você não tem de gerenciar as aberturas e fechamentos das Sessions.

Nota: O suporte as classes do Hibernate no Spring estão localizados nos pacotes `org.springframework.orm.hibernate` e `org.springframework.orm.hibernate-support`.

Dependências

O Hibernate possui algumas bibliotecas de terceiros da qual ele depende. Todas elas estão disponíveis como parte do download do Hibernate. Abaixo estão os JARs incluindo no download do Hibernate 2.1.6. Todas elas são requeridas, exceto as marcadas como opcional. O Spring requer o Hibernate 2.1 ou superior.

- **hibernate2.jar**: core do Hibernate
- **c3p0-0.8.4.5.jar**: Pool de conexão básico para executar testes unitários
- **cglib-full-2.0.2.jar**: Biblioteca geradora de código para criação de proxies para classes persistentes.
- **dom4j-1.4.jar**: biblioteca XML para parse de arquivos de configuração e mapeamento.
- **ehcache-0.9.jar**: Cache puro em Java, cache padrão do Hibernate.
- **jta.jar**: Java Transaction API.
- **odmg-2.0.1.jar**: Base para o mapeamento de produtos objeto-relacional.



- (opcional) **oscache-2.0.1.jar** e (cluster-aware) **swarmcache-1.0rc2.jar**: Implementações de caches alternativos.

Tipo: Enquanto o Spring não suporta o Hibernate 3 (agora em beta), um patch está disponível.

Configuração

Para tornar seus objetos persistentes com Hibernate, primeiro crie um arquivo de mapeamento (Este capítulo assume que você já criou anteriormente um POJO `User` em `src/org/model`, com as propriedades `id`, `firstName` e `lastName`).

1. No diretório `src/org/model`, crie um arquivo `User.hbm.xml` com o conteúdo da listagem 1.0.

Listagem 1.0

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>
  <class name="org.model.User" table="app_user">
    <id name="id" column="id" unsaved-value="0">
      <generator class="increment"/>
    </id>
    <property name="firstName" column="first_name"
      not-null="true"/>
    <property name="lastName" column="last_name"
      not-null="true"/>
  </class>
</hibernate-mapping>
```

No mapeamento acima, o elemento `<id>` usa `"increment"` para indicar o valor máximo mais 1 para a chave primária gerada. O tipo gerador `"increment"` não é recomendado para cluster. Felizmente, o Hibernate possui muitas outras opções.

2. Crie um arquivo `applicationContext-hibernate.xml` no diretório `web/WEB-INF` e adicione uma definição bean para um `DataSource`. Você pode usar um `applicationContext*.xml` existente como um template. Este arquivo deverá possuir `spring-beans.dtd` e o elemento raiz `<beans>` definido antes da definição bean.

O bean `dataSource` neste exemplo usa uma base de dados HSQL, que é um banco de dados puramente Java que roda a partir de um simples arquivo `hsqldb.jar` localizando no diretório `web/WEB-INF/lib`. Mais tarde você irá trocar ele pelo MySQL para ver como é fácil a troca de banco de dados. Veja a listagem 1.1.

Listagem 1.1

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="dataSource"
    class="org.springframework.jdbc.datasource.
      DriverManagerDataSource">
    <property name="driverClassName">
      <value>org.hsqldb.jdbcDriver</value>
    </property>
    <property name="url">
      <value>jdbc:hsqldb:db/app</value>
    </property>
    <property name="username"><value>sa</value></property>
    <property name="password"><value></value></property>
  </bean>
  <!-- Add additional bean definitions here -->
</beans>
```

O **DriverManagerDataSource** é um simples DataSource que configura o Driver JDBC através das propriedades do bean. Você também pode configurar um DataSource JNDI se você preferir use seu container DataSource pré-configurado. Por exemplo, uma estratégia comum é usar o **DriverManagerDataSource** para teste, e o DataSource JNDI (veja a listagem 1.2) para produção.

Listagem 1.2

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

3. Adicione uma definição bean **sessionFactory**, que depende do bean **dataSource** prévio e de um arquivo de mapeamento. A propriedade **dialect** irá mudar baseada no banco de dados e a propriedade **hibernate.hbm2ddl.auto** cria um banco de dados on-the-fly quando a aplicação iniciar. Veja a listagem 1.3.

Listagem 1.3

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate.
      LocalSessionFactoryBean">
  <property name="dataSource"><ref bean="dataSource"/></property>
  <property name="mappingResources">
    <list>
      <value>org/model/User.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">
        net.sf.hibernate.dialect.HSQLDialect
      </prop>
      <prop key="hibernate.hbm2ddl.auto">update</prop>
    </props>
  </property>
</bean>
```

Como uma alternativa ao mapeamento individual para cada arquivo *.hbm.xml*, use as propriedades `mappingJarLocations` ou `mappingDirectoryLocations` para referir-se a um arquivo JAR ou diretório. Você também pode usar um arquivo *hibernate.cfg.xml* para especificar suas configurações e apontar para ele usando uma propriedade `configLocation`.

4. Adicione uma definição bean `transactionManager` que usa a classe `HibernateTransactionManager` do Spring. `SessionFactoryUtils` e `HibernateTemplate` são cientes das threads ligadas as Sessions e irão participar em cada uma das transações automaticamente. É necessário usar uma das duas para que o Hibernate suporte o mecanismo de controle transacional. Veja a listagem 1.4.

Listagem 1.4

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

O `UserHibernateDAO` ou `UserDAOTest` não usa este bean especificamente, mas a definição bean `userManager` a referência.

5. Crie uma classe `UserDAOHibernate.java` em `src/org/dao/hibernate` (você precisa criar este diretório/package). Este arquivo estende `HibernateDaoSupport` e implementa `UserDAO`. Veja a listagem 1.5.

Listagem 1.5

```
package org.dao.hibernate;

// use your IDE to organize imports

public class UserDAOHibernate extends HibernateDaoSupport
    implements UserDAO {

    public List getUsers() {
        return getHibernateTemplate().find("from User");
    }

    public User getUser(Long id) {
        User user =
            (User) getHibernateTemplate().get(User.class, id);

        if (user == null) {
            throw
                new ObjectRetrievalFailureException(User.class, id);
        }
        return user;
    }

    public void saveUser(User user) {
        getHibernateTemplate().saveOrUpdate(user);
    }

    public void removeUser(Long id) {
        Object user = getHibernateTemplate().load
            (User.class, id);
        getHibernateTemplate().delete(user);
    }
}
```

6. Adicione uma definição bean para o `userDao` em `applicationContext-hibernate.xml`. Veja a listagem 1.6.

Listagem 1.6

```
<bean id="userDAO"
    class="org.dao.hibernate.UserDAOHibernate">
    <property name="sessionFactory">
        <ref local="sessionFactory"/>
    </property>
</bean>
```

Na classe `UserDAOHibernate`, `HibernateTemplate` faz a maior parte do trabalho. Usar templates para tratar chamadas de persistência é um enredo comum dentro das classes de suporte DAO do Spring. Note também os seguintes itens na classe `UserDAOHibernate.java`.

- O método `getUser()` usa `HibernateTemplate().get()`, que retorna null se ele não achar objetos que combinem. A alternativa é usar `HibernateTemplate().load()`, que lança uma exceção se ele não achar objetos. `HibernateTemplate.get()` é usado no método `removeUser()`, mas você pode facilmente usar `get()` em seu lugar.
- O método `getUser()` lança uma exceção `ObjectRetrievalFailureException` quando ele não achar um usuário.
- Ela não tem exceções checadas. Você provavelmente irá parar de escrever um monte de blocos try/catch com Hibernate.

A variável `logger` já está definida em `HibernateDaoSupport`, permitindo um fácil logging em suas subclasses. Por exemplo, adicione o código na listagem 1.7 no final do método `saveUser()`.

Listagem 1.7

```
if (logger.isDebugEnabled()) {
    logger.debug("User's id set to: " + user.getId());
}
```

Configuração do MySQL

Para trocar do HSQL para MySQL é muito simples. Graças ao Spring isso é simplesmente uma questão de ajuste na configuração.

1. Tenha certeza que o MySQL está instalado e rodando. Certifique-se que o driver JDBC do MySQL está no diretório `web/WEB-INF/lib`.
2. No `applicationContext-hibernate.xml`, mude a propriedade bean `dataSource` conforme o código da listagem 1.8:

Listing 1.8

```
<property name="driverClassName">
    <value>com.mysql.jdbc.Driver</value>
</property>
<property name="url">
    <value>jdbc:mysql://localhost/myusers</value>
</property>
<property name="username"><value>root</value></property>
<property name="password"><value></value></property>
```




O username root e o password vazio são default da instalação. Você deve ajustar isto conforme a sua instalação.

Tipo: Você pode usar um bean `PropertyPlaceholderConfigurer` para setar os valores das propriedades acima (não visto aqui).

3. Mude a propriedade `hibernate.dialect` do bean `sessionFactory` para o MySQL (veja a listagem 1.9):

Listagem 1.9

```
<prop key="hibernate.dialect">
    net.sf.hibernate.dialect.MySQLDialect
</prop>
```

Caching

Uma poderosa característica nos frameworks de persistência é a habilidade para cachear dados e evitar constantes viagens ao banco de dados. O objeto `Session` do Hibernate é um cache em nível de transação de persistência de dados, mas ele não controla caching por classe ou coleção-por-coleção na JVM ou em nível de cluster.

Os exemplos seguintes mostram como configurar o EHCACHE para um caching em nível de JVM.

Nota: EHCACHE é o cache default, assim você não precisa configurar uma entrada `hibernate.cache.provider_class` no *applicationContext-hibernate.xml*.

1. A forma mais simples para ativar o caching para um objeto é adicionar uma tag `<cache>` no arquivo de mapeamento. Para fazer isso com o objeto `user`, adicione o elemento `<cache>` no arquivo *User.hbm.xml* localizando em *src/org/model*. Os valores opcionais são `read-write` e `read-only`. Você pode usar a segunda opção somente se você estiver se referindo a um objeto ou tabela que raramente muda. Veja a listagem 1.10

Listagem 1.10

```
<class name="org.model.User" table="app_user">
  <cache usage="read-write"/>
  <id name="id" column="id" unsaved-value="0">
```

2. (Opcional) Criar entradas no arquivo de configuração do EHCACHE para esta classe. Crie um arquivo *ehcache.xml* em *web/WEB-INF/classes* e insira nele o código XML da listagem 1.11.

Listagem 1.11

```
<ehcache>
  <!-- Only needed if overFlowToDisk="true" -->
  <diskStore path="java.io.tmpdir"/>
  <!-- Required element -->
  <defaultCache
    maxElementsInMemory="10000"
    ceternal="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="120"
    overflowToDisk="true"/>
  <!-- Cache settings per class -->
  <cache name="org.model.User"
    maxElementsInMemory="1000"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    overflowToDisk="true"/>
</ehcache>
```

3. Para provar que seu objeto **User** está cacheado, ative o debug logging para o EHCACHE em *web/WEB-INF/classes/log4j.xml*. (veja a listagem 1.12):

Listing 1.12

```
<logger name="net.sf.ehcache">
  <level value="DEBUG"/>
</logger>
```

A documentação de referência do Hibernate contém mais informações sobre como usar e configurar Caches de Segundo Nível. Em geral, o caching é algo que você não precisa configurar para na sua aplicação até que você tenha afinado seu banco de dados (ex: criação de índices). Esta implementação de cache tem como propósito somente demonstrar o seu funcionamento.

Objetos com dependências Lazy-Loading

Uma das muitas features do Hibernate é a habilidade para lazy-load de objetos dependentes. Por exemplo, se uma lista de usuários referir-se a uma coleção de objetos de cargos, você provavelmente não irá precisar que os cargos sejam carregados para exibir somente a lista de usuários. Marcando a coleção de cargos com `lazy-load="true"`, eles não serão carregados enquanto você não fizer nada com eles (usualmente em uma UI)

Para usar esta feature com Spring, configure o `OpenSessionInViewFilter` em sua aplicação. Isto irá abrir uma sessão quando uma determinada URL é requisitada pela primeira vez e fechada quando terminado a carga da página. Para ativar esta feature, adicione o código XML da listagem 1.13 no arquivo web.xml:

Listagem 1.13

```
<filter>
  <filter-name>hibernateFilter</filter-name>
  <filter-class>org.springframework.orm.hibernate.
    support.OpenSessionInViewFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>hibernateFilter</filter-name>
  <url-pattern>*.html</url-pattern>
</filter-mapping>
```

Usando esta feature podemos causar o erro na Listagem 1.14 quando executarmos nossos testes unitários no DAO.

Listagem 1.14

```
[junit] net.sf.hibernate.LazyInitializationException: Failed to
  lazily
  initialize a collection - no session or session was closed
```

Para arrumar isso, adicione o código na listagem 1.15 para os métodos `setUp()` e `tearDown()` do seu teste.

Listagem 1.15

```
protected void setUp() throws Exception {
    // the following is necessary for lazy loading
    sf = (SessionFactory) ctx.getBean("sessionFactory");
    // open and bind the session for this test thread.
    Session s = sf.openSession();
        TransactionSynchronizationManager
            .bindResource(sf, new SessionHolder(s));
    // setup code here
}

protected void tearDown() throws Exception {
    // unbind and close the session.
    SessionHolder holder = (SessionHolder)
        TransactionSynchronizationManager.getResource(sf);
    Session s = holder.getSession();
    s.flush();
    TransactionSynchronizationManager.unbindResource(sf);
    SessionFactoryUtils.closeSessionIfNecessary(s, sf);

    // teardown code here
}
```