HIBERNATE

Aplicaciones Distribuidas

Contenido

- Introducción.
- ORM
- Hibernate.
- HQL.
- Arquitectura.
- Configuración.
- Mapping.
- Ciclo de Vida.
- Caché.
- Conclusiones.

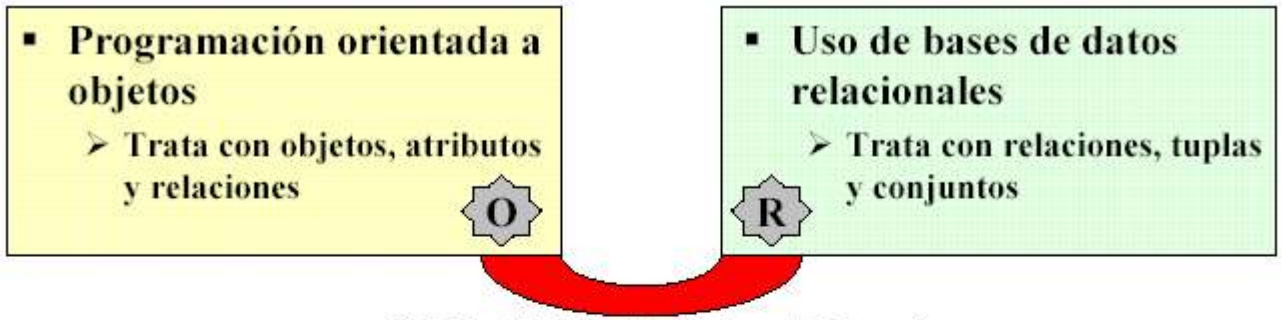
Introducción

- **Motivación**:
  - Dos paradigmas diferentes : programación orientada a objetos y bases de datos relacionales
    - El modelo relacional trata con relaciones, tuplas y conjuntos (matemático por naturaleza)
    - Paradigma orientado a objetos trata con objetos, sus atributos y las relaciones entre objetos
    - GAP Objeto-Relacional

- Ej: Para hacer los objetos persistentes se requiere una conexión JDBC, crear una sentencia SQL y copiar todos los valores de las propiedades sobre un PreparedStatement o en una cadena SQL. ¿Y las asociaciones? ¿Y si el objeto contiene a su vez a otros objetos? ¿Y las claves ajenas?
  - El 35% del código de una aplicación se produce como consecuencia del mapeo de los datos .

- **Granuralidad**: Atributos no primitivos se mapean en una columna.Ejemplo “direccion”.
- **Subtipos**: No hay herencia en las BD SQL.
- **Identidad**: Como realizamos la igualdad, con el operador ==, la operación *equals* o la clave primaria.
- **Asociaciones**: En OO como referencias a objetos y colecciones de objetos, en BD como claves externas. Asociaciones muchos-a-muchos.

- **Solución**: ➔ ORM (Object-Relational Mapping)



ORM: Object-Relational Mapping

- **Problema**: un 35% del código de una aplicación para realizar la correspondencia O ↔ R
- **Solución**: utilizar una ORM, por ejemplo Hibernate

ORM

- Carácterísticas:
  - Un ORM cubre el GAP objeto-relacional
  - Sólo hay que definir la correspondencia entre las clases y las tablas una sola vez (indicando que propiedad se corresponde con que columna, que clase con que tabla, etc…)
  - Utiliza **POJO'**s (Plain Old Java Objects) en la aplicación y los hace persistentes con una sola instrucción: orm.save(myObject)
  - Permite leer y escribir directamente en la BBDD con VO (POJO)

- ORM es el middleware en la capa de persistencia que gestiona la persistencia.
- Esto implica cierta penalización en el rendimiento
- También hay sobrecarga en la gestión de los metadatos del mapeo, pero este coste es menor que el producido cuando se escribe a mano.
  - Necesario hacer uso correcto de las sesiones

- Elementos de un ORM:
  - Un **API** para realizar las operaciones básicas **CRUD** (create, read, update,delete) en objetos persistentes.
  - Un **lenguaje** para especificar **consultas** de objetos y propiedades.
  - Facilidades para definir el **mapeo de los metadatos**.
  - **Optimizaciones**

- Beneficios:
  - **Productividad**: Ahorra mucho trabajo engorroso y repetitivo.
  - **Mantenibilidad**: Al haber menos código la aplicación es más mantenible. Además, evita el acoplamiento en el diseño del modelo de negocio y el de persistencia, cuando este último se hace a mano.
  - **Rendimiento**: ORM realiza muchas optimizaciones, algunas dependientes de la BD en particular.
  - **Independencia del vendedor**: la aplicación es independiente de una BD particular o un dialecto específico SQL.

Hibernate

- Hibernate es un **ORM de libre distribución**.
- Un framework maduro y completo.
- Puede utilizarse en cualquier contexto de ejecución (no necesita un contenedor especial).
- Facilita el mapeo de atributos entre una base de datos relacional y el modelo de objetos de una aplicación, mediante **anotaciones** ó **archivos declarativos** (XML)
- No utiliza técnicas como generación de código a partir de descriptores del modelos de datos o manipulación de bytecodes en tiempo de compilación ni obliga a implementar interfaces específicos.
- Basado en el mecanismo de **reflexión** de Java .

- Carácterísticas:
  - No intrusivo (estilo POJO)
  - Buena documentación, comunidad amplia y activa
  - Transacciones, caché, asociaciones, polimorfismo, herencia, persistencia transitiva, estrategias de fetching.
  - Potente lenguaje de consulta (HQL): subqueries, outer joins, ordering, proyección, paginación.
  - Fácil testeo (basado en pojos).
  - Uso de anotaciones o ficheros XML de mapeo, de donde se obtiene toda la información para realizar las operaciones CRUD.
  - **Uso del estándar JPA o configuración propia no-estándar**

HQL

- **HQL: Hibernate Query Language**
  - Lenguaje para el manejo de consultas a la base de datos.
  - Similar a SQL, orientado a objetos (pojos)
  - Es **solo consulta**.
  - Lenguaje intermedio que será traducido al SQL dependiente de cada base de datos de manera automática y transparente
    - Ejemplo:

    “from Bid” →

    “select B.BID_ID, B.AMOUNT, B.ITEM_ID, B.CREATED from BID B”

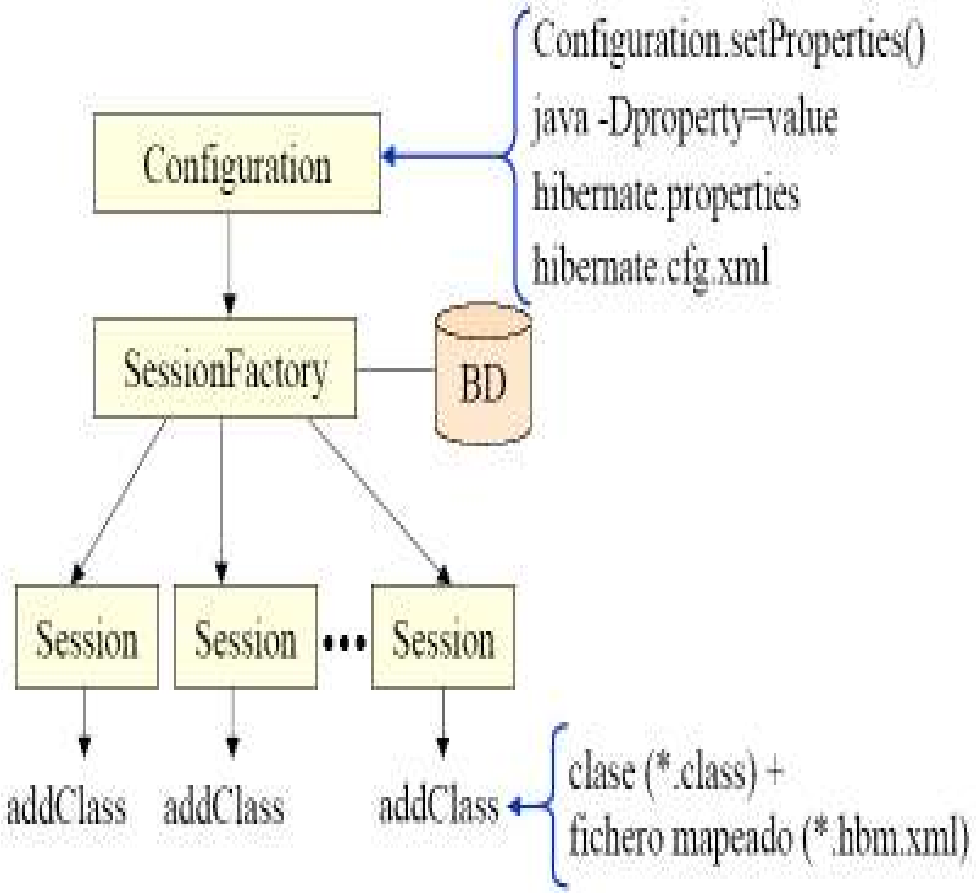
Conceptos Básicos

- Comunicación con el motor de Hibernate mediante **Session**
- Se requiere **una instancia de session para cada tipo de BD**
  - Permite delimitar una o varias operaciones relacionadas dentro de un proceso de negocio, demarcar una transacción y otros servicios (caché de objetos,…)
    - *save(Object object), createQuery(String queryString), beginTransaction(), etc…*
- Una session hace de **cache de objetos** cargados
- Objetos tipo **transient** y tipo **persistent**.
  - **Transient**: objetos que sólo existen en memoria y no en un almacén de datos.
  - **Persistent**: objetos ya almacenados y por tanto persistentes

- Necesidad de crear y cerrar explícitamente las sesiones de Hibernate
- Una sesión siempre va a pertenecer a un mismo ***thread*** de ejecución (el que pertenece a la ejecución de un método de negocio para un usuario concreto)
  - Técnicamente se pueden compartir sesiones entre threads (no aconsejable)
- Es decir (escenario aconsejable), en un entorno multiusuario y por tanto multithread habrá por tanto múltiples sesiones simultáneas, cada una perteneciente a su correspondientes thread y con su contexto de objetos en caché, transacciones, etc.

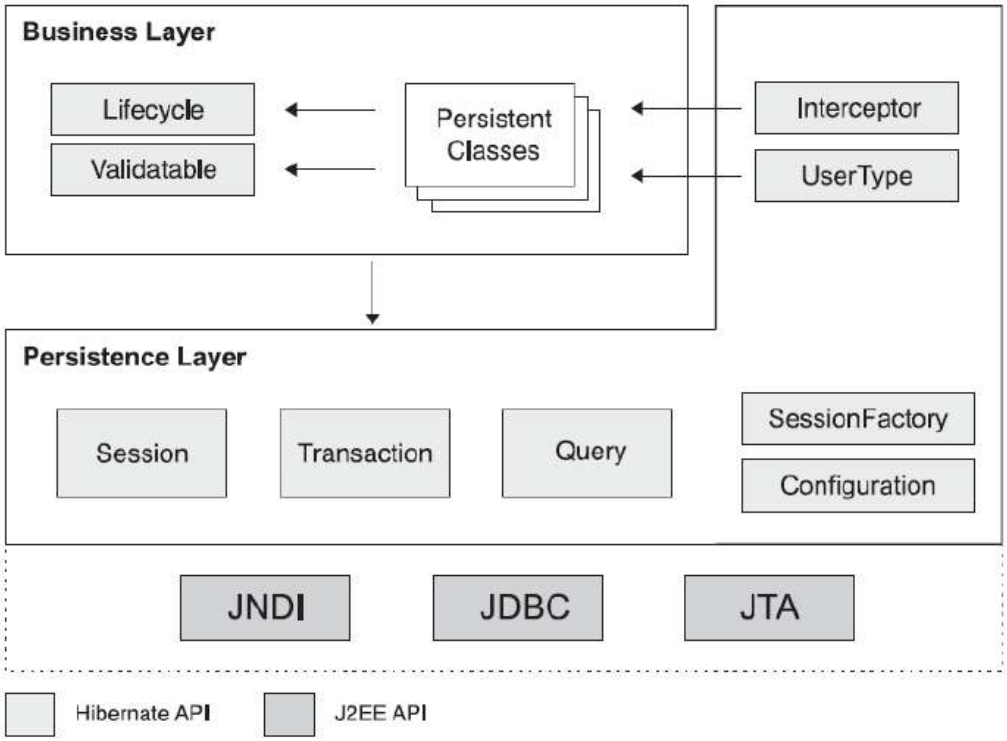
- **SessionFactory** para crear instancias de sesiones y realizar operaciones comunes a los diferentes threads: gestión de una caché compartida entre threads, etc…
  - openSession(), evict(Class persistentClass), …
- ¿Qué sucede si en un entorno de múltiples hilos se accede a un mismo objeto desde dos sesiones diferentes?
  - Una instancia de un objeto persistente no es compartida por dos sesiones (dos instancias dentro de la misma máquina virtual Java para un “mismo” objeto de datos)

- **Configuration** define la configuración (ficheros de mapeo) y arranca Hibernate.



Arquitectura

- El API de Hibernate es una arquitectura de dos capas (Capa de persistencia y Capa de Negocio).



- Interfaces usadas en la programacion:
  - **Session**
  - **Transaction**: Abstracción de una transacción concreta (JDBC,JTA, CORBA). Su uso es opcional
  - **Query**: Realiza peticiones a la base de datos y controla cómo se ejecuta dicha petición. HQL o en SQL nativo. Enlaza los parámetros de la petición, limita el numero de resultados devueltos y ejecuta la consulta.
  - **Criteria**: Para crear y ejecutar consultas OO (parecido a Query).
- Interfaces callback: **Interceptor, Lifecycle, y Validatable.**
- Interfaces que permiten extender las funcionalidades de mapeo: **UserType, CompositeUserType, e IdentifierGenerator**

Configuración

- Configurar y usar Hibernate (*enfoque de ficheros declarativos*)
  - La configuración de Hibernate se realiza por medio del fichero **hibernate.cfg.xml**. En él se especifican propiedades (como el dialecto) y los ficheros de mapeo.
  - Situar el *.jar del driver JDBC elegido y el fichero hibernate*.jar en el classpath
  - Añadir las dependencias (librerias requeridas) de Hibernate (directorio *lib*) en el classpath.

- Elegir y configurar una conexión
- Crear una instancia de *Configuration* en nuestra aplicación y cargar los ficheros de mapeado XML utilizando *addResource()* o *addClass().*
- Obtener una *SessionFactory* a partir de *Configuration* llamando a *BuildSessionFactory*().
- Crear una *Session* a partir del *SessionFactory*

Ejemplo

```
SessionFactory sessions = new
Configuration().configure().buildSessionFactory();
User user = new User(); // V.O. ó POJO
...
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
session.save(user);
tx.commit();
session.close();
```

- POJO / JavaBeans:
  - No es necesario que las clases implementen la interfaz Serializable.
  - Se necesita un constructor vacío.
  - Hibernate usa los métodos *getter* / *setter*, aunque no necesita que los *getter* sean públicos.

Mapping

- La definición del mapeo de los metadatos se hace en un fichero XML. Se pueden definir todos los mapeos en una clase, pero es más recomendable un archivo de mapeo para cada clase.
- El convenio es nombrar a los ficheros con el nombre de la clase, añadiendo el sufijo .hbm.xml.
- Cuando los valores de los atributos se omiten, Hibernate usa la reflexión en la clase mapeada para determinar los valores por defecto.
- Una definición típica de una propiedad define el nombre, el nombre de la columna y el tipo de Hibernate.
  - Si el nombre de la columna es el mismo que el de la propiedad se puede omitir (uso reflexión).

- También se pueden añadir atributos como not-null para establecer si puede ser atributo nulo, o una formula para indicar como calcular el valor de un atributo derivado. Ej.:

```
<property name="totalIncludingTax"
formula="TOTAL + TAX_RATE * TOTAL"
type="big_decimal"/>
```

- Se puede asociar la clave primaria de una tabla a un atributo de la clase:

```
<id name="id" column="CATEGORY_ID" type="long">
…
```

- O se puede dejar que Hibernate gestione la identificación internamente:

```
<id column="CATEGORY_ID">
…
```

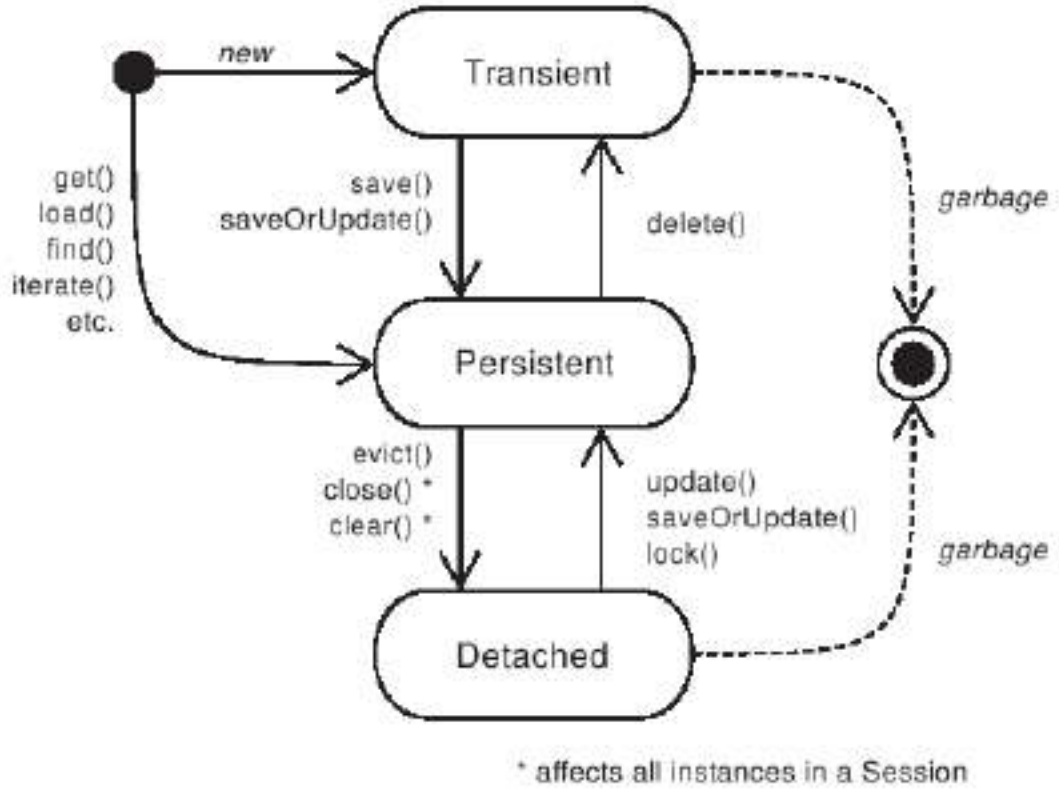
- Hibernate soporta modelos de granulo fino, esto significa que varias clases pueden mapearse a una tabla, una fila representa varios objetos.
  - Por ejemplo: Una tabla Usuario con la información de la dirección personal, dirección fiscal y email se mapea a las clases Address y Email.

```
<class name=”User” table=”USER”>

<component name="homeAddress" class="Address">

    <property name="street" type="string" column="HOME_STREET“ not-null="true"/>

    <property name="city" type="string" column="HOME_CITY" not-null="true"/>

    <property name="zipcode" type="short" column="HOME_ZIPC” not-null="true"/>

</component>
```

- La gestión de las asociaciones entre clases y relaciones entre tablas es una las tareas cruciales en un ORM.
- Multiciplidad: **many-to-one, one-to-many, one-to-one, many-to-many**.
- Con el atributo inverse le indicamos que una asociación es bidireccional.
- Con el atributo cascade=”save-update” le decimos a Hibernate que las nuevas instancias del extremo de la asociación deben persistirse, sin hacer un session.save().
- Con el atributo cascade=”all-delete-orphan” hacemos relaciones padre/hijo, es decir, el padre es responsable del ciclo de vida del hijo.

Ciclo de Vida

- Los objetos para Hibernate tienen los estados:
  - **Transient**
  - **Persistent**
  - **Detached**.



- Los objetos instanciados con new no son persistentes inmediatamente, sino que están en estado **transitorio**.
  - no están en ninguna fila de la tabla y serán perdidos si el objeto se desreferencia.
  - Estos objetos son no transaccionales.
- Una instancia **persistente** es cualquier instancia con identidad de la BD.
  - Están asociadas a un objeto Session y son transaccionales. Su estado es actualizado en la BD al final de la transacción.
  - Por defecto se escriben todos los campos de una tabla en una actualización, pero se puede indicar que solo se escriban las columnas modificadas poniendo 'dynamic-update=true' en el mapeo de <class...
- Un objeto está **despegado** si se cierra la sesión.
  - Estas instancias pueden ser reusadas en una nueva transacción, al reasociarlos con un nuevo gestor de persistencia (la Session).

Programación

- Las sentencias SQL (insert, update, ...) se realizan cuando se ejecuta la operación commit. En este punto, Hibernate consigue una conexión para ejecutar la sentencia.
  - Conseguir un objeto persistente con la operación get().
  - El objeto conseguido así se actualizará automáticamente al final de la transición.
    - User user = (User) session.get(User.class, new Long(1234));
- Un objeto se hace transitorio con la operación delete().

- Formas de recuperar un objeto (I):
  - **Navegar** por el grafo de objetos, desde un objeto ya cargado.
  - **Cargarlo por el identificador**:
    - Con la operación *get(Class, Id)*, o con *load(Class, Id)*, que a diferencia del primero que devuelve null, lanza una excepcion si no lo encuentra
  - Usar el Hibernate Query Language (**HQL**).

- Formas de recuperar un objeto (II):
  - Utilizar ***Criteria*** para hacer consultas. Permite especificar restricciones dinámicamente sin trabajar directamente con cadenas. Parseado en tiempo de compilación.

    Criteria criteria = session.createCriteria(User.class);

    criteria.add(Expression.like("firstname", "Max"));

    List result = criteria.list();

  - Usando consultas **SQL nativas**.

    Query q = session.createQuery("from User u where u.firstname = :fname");

    q.setString("fname", "Max");

    List result = q.list();

- Estrategias de Fetching
  - Hibernate permite especificar la estrategia de búsqueda y recuperación de datos con atributos en los datos de mapeo:
    - **Inmediata**: el objeto asociado es buscado inmediatamente utilizando una lectura de la BD secuencial.
    - **Perezosa**: se busca cuando el objeto es accedido por primera vez. Nueva petición a la BD.
    - **Impaciente**: el objeto/colección es traído junto el objeto poseedor usando un SQL *outer join*.
    - **Por lotes**: para mejorar el rendimiento de la búsqueda perezosa, al traer un lote de objetos/colecciones.

Cache

- Una caché mantiene una representación del estado de la BD cerca de la aplicación, en memoria o en disco.
- Tipos:
  - Ámbito de **transacción**: asociada a la unidad de trabajo actual, ya sea la transacción de la BD o de la aplicación.
  - Ámbito de **proceso**: compartida entre muchas unidades de trabajo
  - Ámbito de **cluster**: compartida entre múltiples procesos (de la misma máquina o en un cluster). Requiere algún tipo de comunicación remota de procesos para mantener la consistencia.

- La **caché de la sesión** es una cache de ámbito transacción. Es obligatoria y no puede desactivarse.
- Es una caché de post-escritura. Es decir los cambios son persistidos cuando:
  - Se confirma la transacción
  - Algunas veces antes de realizar una consulta
  - Llamando explícitamente a Session.flush().
- La caché de sesión asegura que cuando recuperas el mismo objeto persistente dos veces, se devuelve la misma instancia (java).
- Existe una cache de segundo-nivel compartida por todas las sesiones. Las instancias son almacenadas en esta caché en una forma “desensamblada” (parecido a la serialización).

Conclusiones

- **Utilizar** un framework de **ORM** simplifica enormemente la programación de lógica de persistencia
  - Lógica de negocios basada en un modelo de dominio completamente orientado a objetos.
  - Ahorro de código, más simple y fácil de mantener.
  - Proporciona grandes beneficios: independencia de la base de datos, bajo acoplamiento entre negocio y persistencia, y un desarrollo rápido.
- Hibernate ofrece además un lenguaje propio (**HQL)** que lo hace multimotor de base de datos.

- **Hibernate** es una buena herramienta para el mapeo de clases en una base de datos relacional, pero le falta funcionalidad y capacidad en el manejo de transacciones y conexiones
- **Hibernate** es menos invasivo que otros marcos de trabajo de mapeo O/R.
  - Utiliza **reflexión** y la generación de bytecodes en tiempo de ejecución.
  - La generación del código SQL se realiza en el arranque.

Referencias

- http://www.hibernate.org/
- Tutorial de Hibernate:
  http://www.hibernate.org/hib_docs/reference/en/html/
- Trabajo “Hibernate” de Angel Luis Calvo Ortega
- Hibernate in Action Christian Bauer, Gavin King Manning Publications. 2005