HIBERNATE

Objectos Persistentes com ORM

Jorge Gomes Raimundo
Jorge.Raimundo@iscte.pt

ENQUADRAMENTO

Enquadramento

- Bases de dados relacionais são o coração das empresas modernas
- Principal força do modelo relacional: independência face à manipulação e visão dos dados por parte do nível de aplicação que serve
- O Java, por exemplo, fornece-nos uma visão intuitiva, orientada para objectos, no nível da aplicação.

Enquadramento

- A tecnologia *Hibernate* tenta estabelecer a ponte entre estas duas tecnologias (relacional e orientada para objectos) com a abordagem ORM (*object/relational mapping*, mapeamento objecto/relacional).

Enquadramento

- O armazenamento de dados EIS (*Enterprise Information Systems*) envolve uso extensivo de sistemas de gestão de bases de dados baseados em SQL. O legado existente de bases de dados relacionais deve estar disponível para e através de aplicações *web* orientadas para objectos.

- Contudo, a representação dos dados em tabelas por parte dos sistemas relacionais é fundamentalmente diferente da rede de objectos utilizada em aplicações orientadas para objectos.

- *object/relational paradigm mismatch* (desadaptação(?) dos paradigmas objecto/relacional)

PERSISTÊNCIA E DESADAPTAÇÃO ENTRE PARADIGMAS

Persistência

- A persistência, em Java, é normalmente associada à inserção de dados numa base de dados relacional utilizando SQL.
- **Definição simplista:** Em programação orientada para objectos, a persistência permite a um objecto ter um ciclo de vida que ultrapassa o ciclo de vida do processo que o criou.
- Esta “funcionalidade” não se limita a objectos considerados individualmente. Grafos de objectos interligados podem ser tornados persistentes e, posteriormente, recriados num novo processo.

Object/relational paradigm mismatch

- São fundamentalmente modelos diferentes e não apenas maneiras diferentes de visualizar o mesmo modelo.
- ORM (*object/relational mapping*) é o nome dado às soluções automatizadas para o problema do *mismatch*.

Paradigm Mismatch (aprofundamento)

- ## Problemas de granularidade:

  A granularidade refere-se ao tamanho relativo dos objectos com que trabalhamos. Objectos persistentes podem ter diferentes tipos de granularidade para tabelas e colunas de granularidade inerentemente limitada.

  Por exemplo: para adicionar um novo tipo de dados que armazenasse objectos Java do tipo `Endereço`, criar uma coluna com um novo tipo de dados SQL `ENDEREÇO` deveria garantir interoperabilidade. Contudo, o suporte para *user-defined column types* (UDT) é de certa forma obscuro e carece de portabilidade, daí ser pouco utilizado.

  A nossa solução para este problema passa por várias colunas, ou tipos SQL de terceiros que necessitariam de contrapartidas monetárias.

  As classes no nosso domínio de objectos têm diferentes níveis de granularidade, ao passo que nas bases de dados existem apenas dois níves de granularidade (tabelas e colunas).

  Muitos mecanismos de persistência falham ao reconhecer esta desadaptação e acabam por forçar a representação menos flexível sobre o modelo de objectos. É comum encontrar-se classes do tipo `Utilizador` com propriedades do tipo `código_postal`.

Paradigm Mismatch (aprofundamento)

- Problemas de herança (subtipos)

Em Java implementamos heranças utilizando super- e subclasses. Notamos imediatamente que o SQL não fornece qualquer suporte directo para a herança. Não podemos declarar que a tabela `DETALHES_DE_CONTA_A_CREDITO` é um subtipo da tabela `DETALHES_DE_CONTA` dizendo `DETALHES_DE_CONTA_A_CREDITO EXTENDS DETALHES_DE_CONTA (...).`

Supondo que a classe `Utilizador` está associada à classe `DetalhesDeConta`, um objecto `Utilizador` pode estar ligado a qualquer subclasse de `DetalhesDeConta`.

Da mesma forma gostaríamos de poder escrever *queries* que se referissem à classe `DetalhesDeConta` e obtivéssemos na resposta instâncias das suas subclasses.

Paradigm Mismatch (aprofundamento)

- ## Problemas de identidade

O Java define duas noções de "*identicalidade*":

- Identidade entre objectos (pode-se considerar equivalente à localização na memória, verificada com `a==b`)
- Igualdade, tal como está determinada pela implementação do método `equals()` (também chamada de igualdade por valor)

Por outro lado, o identificador de uma linha na base de dados é expresso pela chave primária que não é naturalmente equivalente nem ao operador `==` nem ao método `equals()`.

É comum que vários objectos não idênticos representem a mesma linha de uma base de dados.

Uma chave primária é, normalmente, gerada automaticamente pelo sistema, sem significado para o utilizador. É introduzida puramente para o benefício do modelo de dados relacional (por exemplo: `USER_ID`). Como deve ser representado esse campo no modelo dos objectos? Deverá ser representado?

Paradigm Mismatch (aprofundamento)

- ## Problemas relacionados com referências

  As linguagens orientadas para objectos representam as associações utilizando referências para objectos e colecções de referências para objectos. No mundo relacional, uma associação é representada por uma coluna com uma chave estrangeira.

  Há diferenças subtis entre as duas representações.

  Referências para objectos são inerentemente direccionais; a associação é de um objecto para o outro. Se uma associação entre objectos deve ser navegável em ambas as direcções, deve-se definir a associação duas vezes, uma em cada uma das classes associadas.

  Por outro lado, associações de chaves estrangeiras não são por natureza direccionais. De facto, a navegação não tem significado no modelo relacional, dado que se podem criar associações arbitrárias de dados com *table joins* e *projections*.

  Não é possível determinar a multiplicidade de uma associação unidireccional olhando simplesmente para as classes Java. As associações Java podem ser de muitos-para-muitos, enquanto que as associações das tabelas são sempre um-para-muitos ou um-para-um. Pode-se aferir imediatamente da multiplicidade olhando para a definição da chave estrangeira.

Paradigm Mismatch (aprofundamento)

- Problemas de navegação de grafos

  Há uma diferença fundamental no modo como se acede a objectos em Java e numa base de dados relacional. Em Java, por exemplo, podemos aceder à informação de uma conta de um utilizador pela linha
  `umUtilizador.detalhesDeFacturação().númeroDeConta();`
  Percorre-se o grafo do objecto. Navega-se de um objecto para outro seguindo associações entre instâncias.

  Infelizmente esta não é uma maneira eficiente de obter dados de uma base de dados SQL. O acesso eficiente a bases de dados SQL passa geralmente por usar *join's* entre as tabelas que interessam.
  Novo problema: é necessário saber que partes do grafo planeamos aceder **antes** de começar a navegação do grafo. Por exemplo:

```
SELECT * from UTILIZADOR u
left outer join DETALHES_FACTURACAO on bd.ID_UTILIZADOR =
u.ID_UTILIZADOR where u.ID_UTILIZADOR = 123
```

Paradigm Mismatch (aprofundamento)

- O custo

  Em aplicações que lidam com bases de dados SQL, cerca de 30% do código escrito destina-se a lidar com SQL/JDBC e a fazer a ponte entre os dois paradigmas.

  Apesar de todo este esforço, o resultado final ainda não parece ser o melhor.

  A solução pode passar por vergar o modelo orientado para objectos até se ajustar ao modelo relacional. Esta solução pode ser implementada com sucesso mas não sem perder algumas das vantagens do modelo orientado para objectos.

POSSÍVEIS SOLUÇÕES

Possíveis soluções

- Codificar uma camada de persistência com SQL/JDBC

A abordagem mais comum para implementar persistência em Java é a dos programadores trabalharem directamente com SQL e JDBC. Contudo o trabalho envolvido na codificação manual da persistência para cada classe do domínio é considerável, especialmente quando vários dialectos de SQL são suportados.

Este trabalho acaba por consumir uma grande parte do esforço de desenvolvimento.

Quando os requisitos mudam, uma solução codificada à mão requer mais atenção e maior esforço de manutenção.

Possíveis soluções

- Usar serialização

O Java tem incorporado um mecanismo de persistência: a Serialização fornece-nos a capaciade de escrever um grafo de objectos (o estado da aplicação) para um *byte-stream* que pode ser tornado persistente num ficheiro ou numa base de dados.

Infelizmente um grado *serializado* de objectos interligados apenas pode ser acedido como um todo; é impossível obter qualquer dado do *byte-stream* sem *desserializar* todo o *stream*.

Nem é possível aceder ou actualizar um único objecto ou subgrafo independentemente. Ler e escrever um grafo inteiro em cada transacção não é opção para sistemas desenhados para suportar alta concorrência.

Possíveis soluções

- EJB's (*Enterprise JavaBeans*) *entity beans*

  A popularidade dos EJB's está em rápido declínio. Na prática, os EJB 2.1 *entity beans* foram um desastre. Falhas no desenho da especificação EJB fazem com que a *bean-managed-persistence* (BMP) não consiga ser eficiente.

  Não suporta *queries* nem associações polimórficas, uma das funcionalidades que definem o "verdadeiro" ORM.

  Na prática, os *entity beans* não são portáveis.

  Os *entity beans* não são serializáveis.

  O EJB é um modelo intrusivo; requer um estilo de Java não natural e faz com que a reutilização de código fora de um *container* específico seja extremamente difícil.

Possíveis soluções

- Sistemas de bases de dados orientados para objectos

Dado que trabalhamos com objectos em Java, seria ideal se houvesse um modo de guardar esses objectos numa base de dados sem ter de “dobrar” o modelo de objectos. A meio dos anos 90, bases de dados orientadas para objectos começaram a ganhar atenção.

Não olharemos em detalhe para estas bases de dados, uma vez que não foram largamente adoptadas e não parece provável que o venham a ser num futuro próximo.

A esmagadora maioria dos programadores trabalhará com bases de dados relacionais.

Possíveis soluções

- Outras opções

Naturalmente há outras opções. A persistência através de XML é uma variação do tema da serialização. Esta abordagem visa resolver alguns problemas da serialização em *byte-stream*, ao permitir que ferramentas acedam facilmente à estrutura de dados (mas está sujeita à impedância do *object/hierarchical mismatch*).

Não há qualquer benefício adicional do XML, dado que é apenas mais um formato de ficheiro de texto.

ORM
(*Object/Relational Mapping)*

ORM

- Em breves palavras: ORM é a persistência automatizada de objectos de uma aplicação Java nas tabelas de uma base de dados relacional, utilizando metadados que descrevem o mapeamento entre os objectos e a base de dados.

- Transforma os dados de uma representação para a outra, o que implica certos custos na eficiência/desempenho.

- Uma solução ORM consiste nas quatro peças seguintes:

  1 - Uma API que opera as operações básicas de CRUD (*create*, *read*, *update*, *delete*) em objectos de classes persistentes.

  2 - Uma linguagem ou API para especificar *queries* que se referem a classes e a propriedades de classes.

  3 - Facilidade em especificar metadados mapeados.

  4 - Uma técnica para a implementação ORM interagir com objectos de transacção para executar *dirty checking*, *lazy association fetching*, e outras funções de optimização.

Porquê ORM?

- *Produtividade*: código relacionado com a persistência pode ser o código mais entediante numa num aplicação Java. O *Hibernate* elimina muito desse trabalho aborrecido, permite concentrarmo-nos no problema de negócio e reduz significantemente o tempo de desenvolvimento.

- *Manutenção*: Menos linhas de código faz com que o sistema seja mais compreensível dado que enfatiza a lógica de negócio. Mais importante, um sistema com menos código é mais fácil de refactorizar.

- A persistência codificada à mão origina uma tensão inevitável entre a representação relacional e o modelo de objectos que implementam o domínio. Mudanças num implicam mudanças noutro e frequentemente o desenho de uma representação é comprometido para acomodar a existência do outro.

PEQUENO EXEMPLO

Pequeno exemplo (classe base)

```
public class Mensagem {
    private Long id;
    private String texto;
    private Mensagem proximaMensagem;

    private Mensagem () {}

    public Mensagem(String texto) {
        this.texto = texto;
    }

    public Long getId() {
        return id;
    }

    private void setId(Long id) {
        this.id = id;
    }

    public String getTexto() {
        return texto;
    }

    public void setTexto (String texto) {
        this.texto = texto;
    }

    public Mensagem getProximaMensagem() {
        return proximaMensagem;
    }

    public void setProximaMensagem (Mensagem proximaMensagem) {
        this.proximaMensagem = proximaMensagem;
    }
}
```

Pequeno exemplo (classe base)

```
public class Mensagem {
    private Long id;
    private String texto;
    private Mensagem proximaMensagem;

    private Mensagem () {}

    public Mensagem(String texto) {
        this.texto = texto;
    }

    public Long getId() {
        return id;
    }

    private void setId(Long id) {
        this.id = id;
    }

    public String getTexto() {
        return texto;
    }

    public void setTexto (String texto) {
        this.texto = texto;
    }

    public Mensagem getProximaMensagem() {
        return proximaMensagem;
    }

    public void setProximaMensagem (Mensagem proximaMensagem) {
        this.proximaMensagem = proximaMensagem;
    }
}
```

Três atributos:
- identificador
- texto
- referência para a próxima mensagem

Pequeno exemplo (classe base)

```
public class Mensagem {
    private Long id;
    private String texto;
    private Mensagem proximaMensagem;

    private Mensagem () {}

    public Mensagem(String texto) {
        this.texto = texto;
    }

    public Long getId() {
        return id;
    }

    private void setId(Long id) {
        this.id = id;
    }

    public String getTexto() {
        return texto;
    }

    public void setTexto (String texto) {
        this.texto = texto;
    }

    public Mensagem getProximaMensagem() {
        return proximaMensagem;
    }

    public void setProximaMensagem (Mensagem proximaMensagem) {
        this.proximaMensagem = proximaMensagem;
    }
}
```

Três atributos:
- identificador
- texto
- referência para a próxima mensagem

Todos os atributos de `Mensagem` têm métodos de acesso a propriedades do estilo JavaBean

Pequeno exemplo (onde a magia está)

Para que tudo o que vamos ver a serguir aconteça, é necessário fornecer ao Hibernate a informação sobre como é que a classe `Mensagem` se tornará persistente.

Esta informação é normalmente fornecida num *documento de mapeamento XML*.

O documento define, entre outras coisas, como as propriedades de `Mensagem` se mapeiam em colunas da tabela `MENSAGEM`.

Mensagem.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class
            name="cap2.Mensagem"
            table="MENSAGENS">
        <id
                name="id"
                column="MENSAGEM_ID">
            <generator class="increment"/>
        </id>
        <property
                name="texto"
                column="MENSAGEM_TEXTO"/>
        <many-to-one
                name="proximaMensagem"
                cascade="all"
                column="PROXIMA_MENSAGEM_ID"/>
    </class>
</hibernate-mapping>
```

Pequeno exemplo (onde a magia está)

Para que tudo o que vamos ver a serguir aconteça, é necessário fornecer ao Hibernate a informação sobre como é que a classe `Mensagem` se tornará persistente.

Esta informação é normalmente fornecida num *documento de mapeamento XML*.

O documento define, entre outras coisas, como as propriedades de `Mensagem` se mapeiam em colunas da tabela `MENSAGEM`.

Mensagem.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class
            name="cap2.Mensagem"
            table="MENSAGENS">
        <id
                name="id"
                column="MENSAGEM_ID">
            <generator class="increment"/>
        </id>
        <property
                name="texto"
                column="MENSAGEM_TEXTO"/>
        <many-to-one
                name="proximaMensagem"
                cascade="all"
                column="PROXIMA_MENSAGEM_ID"/>
    </class>
</hibernate-mapping>
```

Nota: Hibernate 3.0 e Hibernate 3.1 usam o mesmo DTD.

Pequeno exemplo (onde a magia está)

Para que tudo o que vamos ver a serguir aconteça, é necessário fornecer ao Hibernate a informação sobre como é que a classe `Mensagem` se tornará persistente.

Esta informação é normalmente fornecida num *documento de mapeamento XML*.

O documento define, entre outras coisas, como as propriedades de `Mensagem` se mapeiam em colunas da tabela `MENSAGEM`.

Mensagem.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class
            name="cap2.Mensagem"
            table="MENSAGENS">
        <id
                name="id"
                column="MENSAGEM_ID">
            <generator class="increment"/>
        </id>
        <property
                name="texto"
                column="MENSAGEM_TEXTO"/>
        <many-to-one
                name="proximaMensagem"
                cascade="all"
                column="PROXIMA_MENSAGEM_ID"/>
    </class>
</hibernate-mapping>
```

Nota: Hibernate 3.0 e Hibernate 3.1 usam o mesmo DTD.

A classe `Mensagem` deve ser tornada persistente na tabela `MENSAGENS`

Pequeno exemplo (onde a magia está)

Para que tudo o que vamos ver a serguir aconteça, é necessário fornecer ao Hibernate a informação sobre como é que a classe `Mensagem` se tornará persistente.

Esta informação é normalmente fornecida num *documento de mapeamento XML*.

O documento define, entre outras coisas, como as propriedades de `Mensagem` se mapeiam em colunas da tabela `MENSAGEM`.

Mensagem.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class
            name="cap2.Mensagem"
            table="MENSAGENS">
        <id
                name="id"
                column="MENSAGEM_ID">
            <generator class="increment"/>
        </id>
        <property
                name="texto"
                column="MENSAGEM_TEXTO"/>
        <many-to-one
                name="proximaMensagem"
                cascade="all"
                column="PROXIMA_MENSAGEM_ID"/>
    </class>
</hibernate-mapping>
```

Nota: Hibernate 3.0 e Hibernate 3.1 usam o mesmo DTD.

A classe `Mensagem` deve ser tornada persistente na tabela `MENSAGENS`

A propriedade identificadora mapeia para a coluna `MENSAGEM_ID`

Pequeno exemplo (onde a magia está)

Para que tudo o que vamos ver a serguir aconteça, é necessário fornecer ao Hibernate a informação sobre como é que a classe `Mensagem` se tornará persistente.

Esta informação é normalmente fornecida num *documento de mapeamento XML*.

O documento define, entre outras coisas, como as propriedades de `Mensagem` se mapeiam em colunas da tabela `MENSAGEM`.

Mensagem.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class
            name="cap2.Mensagem"
            table="MENSAGENS">
        <id
                name="id"
                column="MENSAGEM_ID">
            <generator class="increment"/>
        </id>
        <property
                name="texto"
                column="MENSAGEM_TEXTO"/>
        <many-to-one
                name="proximaMensagem"
                cascade="all"
                column="PROXIMA_MENSAGEM_ID"/>
    </class>
</hibernate-mapping>
```

Nota: Hibernate 3.0 e Hibernate 3.1 usam o mesmo DTD.

A classe `Mensagem` deve ser tornada persistente na tabela `MENSAGENS`

A propriedade identificadora mapeia para a coluna `MENSAGEM_ID`

A propriedade `texto` mapeia para a coluna `MENSAGEM_TEXTO`

Pequeno exemplo (onde a magia está)

Para que tudo o que vamos ver a serguir aconteça, é necessário fornecer ao Hibernate a informação sobre como é que a classe `Mensagem` se tornará persistente.

Esta informação é normalmente fornecida num *documento de mapeamento XML*.

O documento define, entre outras coisas, como as propriedades de `Mensagem` se mapeiam em colunas da tabela `MENSAGEM`.

Mensagem.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class
            name="cap2.Mensagem"
            table="MENSAGENS">
        <id
                name="id"
                column="MENSAGEM_ID">
            <generator class="increment"/>
        </id>
        <property
                name="texto"
                column="MENSAGEM_TEXTO"/>
        <many-to-one
                name="proximaMensagem"
                cascade="all"
                column="PROXIMA_MENSAGEM_ID"/>
    </class>
</hibernate-mapping>
```

Nota: Hibernate 3.0 e Hibernate 3.1 usam o mesmo DTD.

A classe `Mensagem` deve ser tornada persistente na tabela `MENSAGENS`

A propriedade identificadora mapeia para a coluna `MENSAGEM_ID`

A propriedade `texto` mapeia para a coluna `MENSAGEM_TEXTO`

A propriedade `proximaMensagem` mapeia para a coluna chamada `PROXIMA_MENSAGEM_ID`

Pequeno exemplo (onde a magia está)

Para que tudo o que vamos ver a serguir aconteça, é necessário fornecer ao Hibernate a informação sobre como é que a classe `Mensagem` se tornará persistente.

Esta informação é normalmente fornecida num *documento de mapeamento XML*.

O documento define, entre outras coisas, como as propriedades de `Mensagem` se mapeiam em colunas da tabela `MENSAGEM`.

Mensagem.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class
            name="cap2.Mensagem"
            table="MENSAGENS">
        <id
                name="id"
                column="MENSAGEM_ID">
            <generator class="increment"/>
        </id>
        <property
                name="texto"
                column="MENSAGEM_TEXTO"/>
        <many-to-one
                name="proximaMensagem"
                cascade="all"
                column="PROXIMA_MENSAGEM_ID"/>
    </class>
</hibernate-mapping>
```

Nota: Hibernate 3.0 e Hibernate 3.1 usam o mesmo DTD.

A classe `Mensagem` deve ser tornada persistente na tabela `MENSAGENS`

Primary Key com auto-incremento

A propriedade `texto` mapeia para a coluna `MENSAGEM_TEXTO`

A propriedade `proximaMensagem` mapeia para a coluna chamada `PROXIMA_MENSAGEM`

Pequeno exemplo (onde a magia está)

Para que tudo o que vamos ver a serguir aconteça, é necessário fornecer ao Hibernate a informação sobre como é que a classe `Mensagem` se tornará persistente.

Esta informação é normalmente fornecida num *documento de mapeamento XML*.

O documento define, entre outras coisas, como as propriedades de `Mensagem` se mapeiam em colunas da tabela `MENSAGEM`.

Mensagem.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class
            name="cap2.Mensagem"
            table="MENSAGENS">
        <id
                name="id"
                column="MENSAGEM_ID">
            <generator class="increment"/>
        </id>
        <property
                name="texto"
                column="MENSAGEM_TEXTO"/>
        <many-to-one
                name="proximaMensagem"
                cascade="all"
                column="PROXIMA_MENSAGEM_ID"/>
    </class>
</hibernate-mapping>
```

Nota: Hibernate 3.0 e Hibernate 3.1 usam o mesmo DTD.

A classe `Mensagem` deve ser tornada persistente na tabela `MENSAGENS`

Primary Key com auto-incremento

A propriedade `texto` mapeia para a coluna `MENSAGEM_TEXTO`

A propriedade `proximaMensagem` é uma associação com multiplicidade *muitos-para-um* (*...1)

Pequeno exemplo (onde a magia está)

Para que tudo o que vamos ver a serguir aconteça, é necessário fornecer ao Hibernate a informação sobre como é que a classe `Mensagem` se tornará persistente.

Esta informação é normalmente fornecida num *documento de mapeamento XML*.

O documento define, entre outras coisas, como as propriedades de `Mensagem` se mapeiam em colunas da tabela `MENSAGEM`.

Mensagem.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class
            name="cap2.Mensagem"
            table="MENSAGENS">
        <id
                name="id"
                column="MENSAGEM_ID">
            <generator class="increment"/>
        </id>
        <property
                name="texto"
                column="MENSAGEM_TEXTO"/>
        <many-to-one
                name="proximaMensagem"
                cascade="all"
                column="PROXIMA_MENSAGEM_ID"/>
    </class>
</hibernate-mapping>
```

Nota: O ficheiro XML pode ser gerado a partir de comentários presentes no código.

Pequeno exemplo (onde a magia está)

Para que tudo o que vamos ver a serguir aconteça, é necessário fornecer ao Hibernate a informação sobre como é que a classe `Mensagem` se tornará persistente.

Esta informação é normalmente fornecida num *documento de mapeamento XML*.

O documento define, entre outras coisas, como as propriedades de `Mensagem` se mapeiam em colunas da tabela `MENSAGEM`.

Mensagem.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class
            name="cap2.Mensagem"
            table="MENSAGENS">
        <id
                name="id"
                column="MENSAGEM_ID">
            <generator class="increment"/>
        </id>
        <property
                name="texto"
                column="MENSAGEM_TEXTO"/>
        <many-to-one
                name="proximaMensagem"
                cascade="all"
                column="PROXIMA_MENSAGEM_ID"/>
    </class>
</hibernate-mapping>
```

Convenções:

1 – Os ficheiros de mapeamento XML têm a extensão `.hbm.xml`

2 – Existe um ficheiro de mapeamento por cada classe persistente.

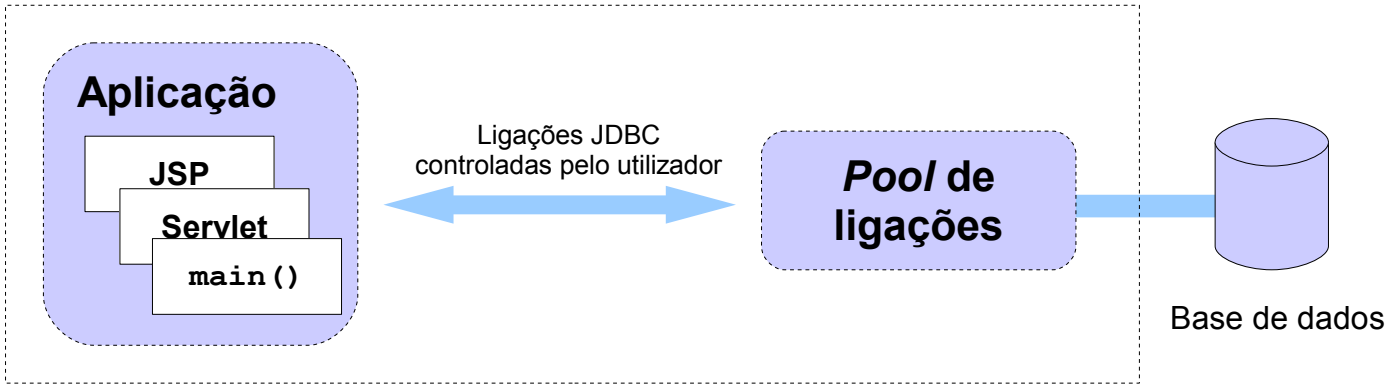
Recomendação:

A documentação *Hibernate* recomenda que o ficheiro de mapeamento de cada classe persistente seja colocado na mesma directoria dessa classe.

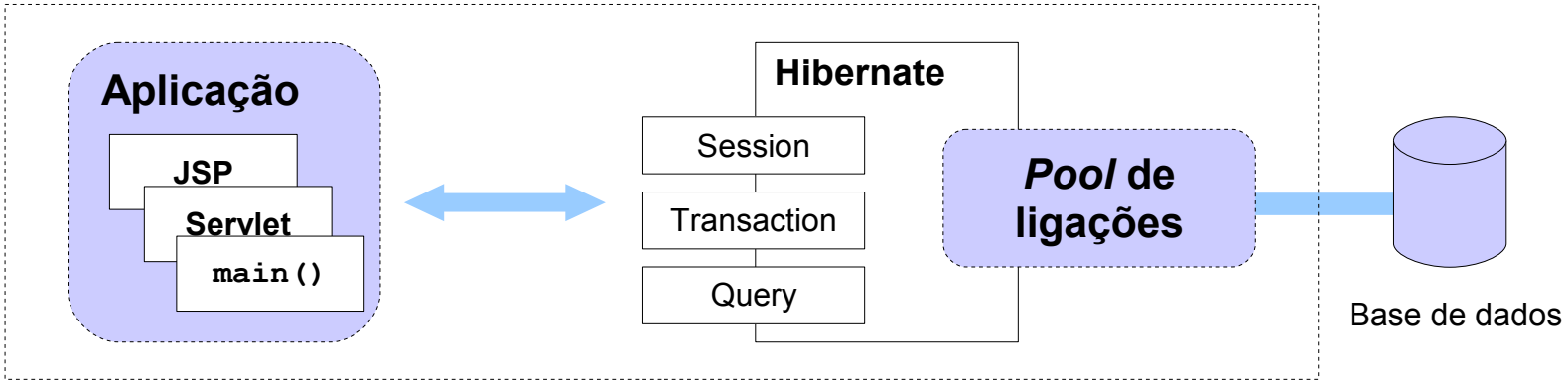
Pequeno exemplo (configuração)

- Configurar o *Hibernate* não consiste apenas em indicar documentos de mapeamento. É necessário especificar como serão obtidas as ligações às bases de dados, assim como vários outros valores que afectam o *Hibernate* em *runtime*.
- Há várias técnicas de especificar as opções de configuração:
  - Passar uma instância de `java.util.Properties` para o `Configuration.setProperties()`.
  -Definir as propriedades de sistema utilizando `java -Dproperty=value`
  - Colocar um ficheiro chamado `hibernate.properties` no *classpath*.
  - Incluir elementos `<property>` no ficheiro `hibernate.cfg.xml` no classpath.
- As primeira e segunda hipóteses raramente são usadas, excepto para testes rápidos e protótipos, mas a maioria das aplicações necessita de um ficheiro de configuração. A escolha entre a terceira ou a quarta hipótese depende apenas da preferência de sintaxe do programador

Pequeno exemplo (configuração)



Pooling de ligações com JDBC



Pooling de ligações com *Hibernate*

Pequeno exemplo (configuração)

O *Hibernate* define uma arquitectura de *plugins* que permite a integração com qualquer *pool* de ligações. O suporte para C3PO está incorporado, pelo que será esse que utilizaremos.

As definições para essa ligação serão definidas no ficheiro `Hibernate.properties`.

Hibernate.properties

```
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost/hibernate_teste
hibernate.connection.username=root
hibernate.connection.password=teste
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=300
hibernate.c3p0.max_statements=50
hibernate.c3p0.idle_test_period=3000
#hibernate.show_sql=true
```

Pequeno exemplo (configuração)

O *Hibernate* define uma arquitectura de *plugins* que permite a integração com qualquer *pool* de ligações. O suporte para C3PO está incorporado, pelo que será esse que utilizaremos.

As definições para essa ligação serão definidas no ficheiro `Hibernate.properties`.

Hibernate.properties

```
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost/hibernate_teste
hibernate.connection.username=root
hibernate.connection.password=teste
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=300
hibernate.c3p0.max_statements=50
hibernate.c3p0.idle_test_period=3000
#hibernate.show_sql=true
```

Nome da classe que implementa o *driver* JDBC

Pequeno exemplo (configuração)

O *Hibernate* define uma arquitectura de *plugins* que permite a integração com qualquer *pool* de ligações. O suporte para C3PO está incorporado, pelo que será esse que utilizaremos.

As definições para essa ligação serão definidas no ficheiro `Hibernate.properties`.

Hibernate.properties

```
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost/hibernate_teste
hibernate.connection.username=root
hibernate.connection.password=teste
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=300
hibernate.c3p0.max_statements=50
hibernate.c3p0.idle_test_period=3000
#hibernate.show_sql=true
```

Um URL LDBC que especifica o *host* e o nome da base de dados para ligações JDBC

Pequeno exemplo (configuração)

O *Hibernate* define uma arquitectura de *plugins* que permite a integração com qualquer *pool* de ligações. O suporte para C3PO está incorporado, pelo que será esse que utilizaremos.

As definições para essa ligação serão definidas no ficheiro `Hibernate.properties`.

Hibernate.properties

```
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost/hibernate_teste
hibernate.connection.username=root
hibernate.connection.password=teste
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=300
hibernate.c3p0.max_statements=50
hibernate.c3p0.idle_test_period=3000
#hibernate.show_sql=true
```

Nome do utilizador da base de dados.

Pequeno exemplo (configuração)

O *Hibernate* define uma arquitectura de *plugins* que permite a integração com qualquer *pool* de ligações. O suporte para C3PO está incorporado, pelo que será esse que utilizaremos.

As definições para essa ligação serão definidas no ficheiro `Hibernate.properties`.

Hibernate.properties

```
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost/hibernate_teste
hibernate.connection.username=root
hibernate.connection.password=teste
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=300
hibernate.c3p0.max_statements=50
hibernate.c3p0.idle_test_period=3000
#hibernate.show_sql=true
```

Password da base de dados do utilizador especificado.

Pequeno exemplo (configuração)

O *Hibernate* define uma arquitectura de *plugins* que permite a integração com qualquer *pool* de ligações. O suporte para C3PO está incorporado, pelo que será esse que utilizaremos.

As definições para essa ligação serão definidas no ficheiro `Hibernate.properties`.

Hibernate.properties

```
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost/hibernate_teste
hibernate.connection.username=root
hibernate.connection.password=teste
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=300
hibernate.c3p0.max_statements=50
hibernate.c3p0.idle_test_period=3000
#hibernate.show_sql=true
```

Dialecto para a base de dados. Novos dialectos podem ser definidos.

Pequeno exemplo (configuração)

O *Hibernate* define uma arquitectura de *plugins* que permite a integração com qualquer *pool* de ligações. O suporte para C3PO está incorporado, pelo que será esse que utilizaremos.

As definições para essa ligação serão definidas no ficheiro `Hibernate.properties`.

Hibernate.properties

```
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost/hibernate_teste
hibernate.connection.username=root
hibernate.connection.password=teste
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=300
hibernate.c3p0.max_statements=50
hibernate.c3p0.idle_test_period=3000
#hibernate.show_sql=true
```

Número mínimo de ligações JDBC que o C3PO manterá prontas.

Pequeno exemplo (configuração)

O *Hibernate* define uma arquitectura de *plugins* que permite a integração com qualquer *pool* de ligações. O suporte para C3PO está incorporado, pelo que será esse que utilizaremos.

As definições para essa ligação serão definidas no ficheiro `Hibernate.properties`.

Hibernate.properties

```
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost/hibernate_teste
hibernate.connection.username=root
hibernate.connection.password=teste
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=300
hibernate.c3p0.max_statements=50
hibernate.c3p0.idle_test_period=3000
#hibernate.show_sql=true
```

Número máximo de ligações na *pool*. Será lançada uma excepção se este número for esgotado.

Pequeno exemplo (configuração)

O *Hibernate* define uma arquitectura de *plugins* que permite a integração com qualquer *pool* de ligações. O suporte para C3PO está incorporado, pelo que será esse que utilizaremos.

As definições para essa ligação serão definidas no ficheiro `Hibernate.properties`.

Hibernate.properties

```
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost/hibernate_teste
hibernate.connection.username=root
hibernate.connection.password=teste
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=300
hibernate.c3p0.max_statements=50
hibernate.c3p0.idle_test_period=3000
#hibernate.show_sql=true
```

Período de *timeout* após o qual uma ligação inactiva será eliminada da *pool*. Neste caso 5 minutos(300 segundos).

Pequeno exemplo (configuração)

O *Hibernate* define uma arquitectura de *plugins* que permite a integração com qualquer *pool* de ligações. O suporte para C3PO está incorporado, pelo que será esse que utilizaremos.

As definições para essa ligação serão definidas no ficheiro `Hibernate.properties`.

Hibernate.properties

```
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost/hibernate_teste
hibernate.connection.username=root
hibernate.connection.password=teste
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=300
hibernate.c3p0.max_statements=50
hibernate.c3p0.idle_test_period=3000
#hibernate.show_sql=true
```

Número máximo de *prepared statements* que ficarão na cache. Essencial para um bom desempenho.

Pequeno exemplo (configuração)

O *Hibernate* define uma arquitectura de *plugins* que permite a integração com qualquer *pool* de ligações. O suporte para C3PO está incorporado, pelo que será esse que utilizaremos.

As definições para essa ligação serão definidas no ficheiro `Hibernate.properties`.

Hibernate.properties

```
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost/hibernate_teste
hibernate.connection.username=root
hibernate.connection.password=teste
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=300
hibernate.c3p0.max_statements=50
hibernate.c3p0.idle_test_period=3000
#hibernate.show_sql=true
```

O tempo (segundos) antes de uma ligação ser automaticamente validada.

Pequeno exemplo (configuração)

O *Hibernate* define uma arquitectura de *plugins* que permite a integração com qualquer *pool* de ligações. O suporte para C3PO está incorporado, pelo que será esse que utilizaremos.

As definições para essa ligação serão definidas no ficheiro `Hibernate.properties`.

Hibernate.properties

```
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost/hibernate_teste
hibernate.connection.username=root
hibernate.connection.password=teste
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=300
hibernate.c3p0.max_statements=50
hibernate.c3p0.idle_test_period=3000
#hibernate.show_sql=true
```

Permite ver na consola todo o SQL gerado.

Pequeno exemplo (algum código inicial)

Antes de mais temos que iniciar o *Hibernate*. Na prática é muito fácil: tem que se criar uma `SessionFactory` a partir de uma `Configuration`.

Mensagem.java

```
public static void main(String[] args){

    Configuration config = new Configuration();
    config.addResource("cap2/Mensagem.hbm.xml");
    SessionFactory sessions = config.buildSessionFactory();

    ...
}
```

ou

```
public static void main(String[] args){

    SessionFactory sessions = new Configuration()
        .addClass(Mensagem.class)
        .buildSessionFactory();

    ...
}
```

Pequeno exemplo (algum código inicial)

Antes de mais temos que iniciar o *Hibernate*. Na prática é muito fácil: tem que se criar uma `SessionFactory` a partir de uma `Configuration`.

Mensagem.java

```
public static void main(String[] args){

    Configuration config = new Configuration();
    config.addResource("cap2/Mensagem.hbm.xml");
    SessionFactory sessions = config.buildSessionFactory();

    ...
}
```

ou

```
public static void main(String[] args){

    SessionFactory sessions = new Configuration()
        .addClass(Mensagem.class)
        .buildSessionFactory();

    ...
}
```

Interface `Configuration`:

O objecto do tipo `Configuration` é utilizado para configurar e arrancar o *Hibernate*.

Pequeno exemplo (algum código inicial)

Antes de mais temos que iniciar o *Hibernate*. Na prática é muito fácil: tem que se criar uma `SessionFactory` a partir de uma `Configuration`.

Mensagem.java

```
public static void main(String[] args){

    Configuration config = new Configuration();
    config.addResource("cap2/Mensagem.hbm.xml");
    SessionFactory sessions = config.buildSessionFactory();

    ...
}
```

ou

```
public static void main(String[] args){

    SessionFactory sessions = new Configuration()
        .addClass(Mensagem.class)
        .buildSessionFactory();

    ...
}
```

Interface `Configuration`:

A localização do ficheiro de mapeamento `Mensagem.hbm.xml` é relativa à raíz do *classpath* da aplicação.
Com este método são indicadas quais as classes a serem tornadas persistentes.

Pequeno exemplo (algum código inicial)

Antes de mais temos que iniciar o *Hibernate*. Na prática é muito fácil: tem que se criar uma `SessionFactory` a partir de uma `Configuration`.

Mensagem.java

```
public static void main(String[] args){

    Configuration config = new Configuration();
    config.addResource("cap2/Mensagem.hbm.xml");
    SessionFactory sessions = config.buildSessionFactory();

    ...
}
```

ou

```
public static void main(String[] args){

    SessionFactory sessions = new Configuration()
        .addClass(Mensagem.class)
        .buildSessionFactory();

    ...
}
```

Interface `SessionFactory`:

Um objecto `SessionFactory` é um objecto pesado, pensado para ser partilhado por toda a aplicação.

É responsável pela criação de instâncias do tipo `Session`.

Se a aplicação acede a várias bases de dados é necessário ter uma `SessionFactory` para cada uma delas.

A `SessionFactory` guarda em cache declarações SQL e metadados de mapeamento.

Pequeno exemplo (algum código inicial)

Antes de mais temos que iniciar o *Hibernate*. Na prática é muito fácil: tem que se criar uma `SessionFactory` a partir de uma `Configuration`.

Mensagem.java

```
public static void main(String[] args){

    Configuration config = new Configuration();
    config.addResource("cap2/Mensagem.hbm.xml");
    SessionFactory sessions = config.buildSessionFactory();

    ...
}
```

ou

```
public static void main(String[] args){

    SessionFactory sessions = new Configuration()
        .addClass(Mensagem.class)
        .buildSessionFactory();

    ...
}
```

Utilizando encadeamento de métodos (*method chaining*), e tendo em conta as convenções apresentadas (ficheiros de mapeamento XML com a extensão `.hbm.xml` e existência de um ficheiro de mapeamento por cada classe persistente), podemos ter um código alternativo com a indicação das classes persistentes pelo método `addClass()`.

Pequeno exemplo (algum código inicial)

Vamos agora inserir o nosso primeiro registo na base de dados.

Mensagem.java

```
public static void main(String[] args){

    ...

    Session session = sessions.openSession();
    Transaction transaction = session.beginTransaction();
    Mensagem mensagem = new Mensagem("Hello World");
    session.save(mensagem);
    transaction.commit();
    session.close();
}
```

Pequeno exemplo (algum código inicial)

Vamos agora inserir o nosso primeiro registo na base de dados.

Mensagem.java

```
public static void main(String[] args){

    ...

    Session session = sessions.openSession();
    Transaction transaction = session.beginTransaction();
    Mensagem mensagem = new Mensagem("Hello World");
    session.save(mensagem);
    transaction.commit();
    session.close();
}
```

Interface `Session`:

O interface `Session` é o principal interface utilizado por aplicações *Hibernate*. Uma instância `Session` é leve e sem custos de criação e destruição.

A maneira mais fácil de pensar numa `Session` é como uma colecção de objectos lidos relativos a uma unidade de trabalho.

É o interface das operações relacionadas com a persistência, tais como guardar e recuperar objectos.

Uma `Session` não é *threadsafe*, e deve ser usada por apenas uma *thread* de cada vez.

Pequeno exemplo (algum código inicial)

Vamos agora inserir o nosso primeiro registo na base de dados.

Mensagem.java

```
public static void main(String[] args){

    ...

    Session session = sessions.openSession();
    Transaction transaction = session.beginTransaction();
    Mensagem mensagem = new Mensagem("Hello World");
    session.save(mensagem);
    transaction.commit();
    session.close();
}
```

Interface `Transaction`:

Uma `Transaction` abstrai o código da aplicação da implementação da transacção subjacente (que pode ser JDBC, JTA ou mesmo CORBA) permitindo à aplicação controlar as fronteiras da transacção através de uma API consistente.

Ajuda a manter as aplicações *Hibernate* portáveis entre diferentes tipos de ambientes de execução.

O interface `Transaction` é opcional. As aplicações *Hibernate* podem escolher não usar este interface e, em vez disso, utilizar o seu próprio código de infraestrutura.

Pequeno exemplo (algum código inicial)

Vamos agora inserir o nosso primeiro registo na base de dados.

Mensagem.java

```
public static void main(String[] args){

    ...

    Session session = sessions.openSession();
    Transaction transaction = session.beginTransaction();
    Mensagem mensagem = new Mensagem("Hello World");
    session.save(mensagem);
    transaction.commit();
    session.close();
}
```

Resulta na execução do seguinte código SQL:

```
insert into MENSAGENS (MENSAGEM_ID, MENSAGEM_TEXTO, PROXIMA_MENSAGEM_ID)
values (1, 'Hello World', null)
```

Pequeno exemplo (algum código inicial)

Vamos agora inserir o nosso primeiro registo na base de dados.

Mensagem.java

```
public static void main(String[] args){

    ...

    Session session = sessions.openSession();
    Transaction transaction = session.beginTransaction();
    Mensagem mensagem = new Mensagem("Hello World");
    session.save(mensagem);
    transaction.commit();
    session.close();
}
```

Atenção!!!

A coluna `MENSAGEM_ID` está a ser inicializada com um valor estranho!

Não definimos a propriedade `id` da `mensagem` em lado nenhum.

De facto a propriedade `id` é especial: é um identificador. Contém um valor único gerado.

O valor é atribuído à instância de `Mensagem` pelo *Hibernate* quando o método `save()` é chamado.

Resulta na execução do seguinte código SQL:

```
insert into MENSAGENS (MENSAGEM_ID, MENSAGEM_TEXTO, PROXIMA_MENSAGEM_ID)
values (1, 'Hello World', null)
```

Pequeno exemplo (algum código inicial)

Vamos agora inserir o nosso primeiro registo na base de dados.

Mensagem.java

```
public static void main(String[] args){

    ...

    Session session = sessions.openSession();
    Transaction transaction = session.beginTransaction();
    Mensagem mensagem = new Mensagem("Olá Mundo");
    session.save(mensagem);
    transaction.commit();
    session.close();
}
```

Atenção!!!

Neste exemplo assumimos que a tabela `MENSAGENS` já existe.

É possível criar as tabelas automaticamente pelo *Hibernate* usando apenas a informação presente no ficheiro de mapeamento.

Resulta na execução do seguinte código SQL:

```
insert into MENSAGENS (MENSAGEM_ID, MENSAGEM_TEXTO, PROXIMA_MENSAGEM_ID)
values (1, 'Hello World', null)
```

Pequeno exemplo (algum código inicial)

O bloco seguinte de código permite-nos ver quais as mensagens presentes na base de dados:

Mensagem.java

```
public static void main(String[] args){

    ...

    Session session = sessions.openSession();
    Transaction transaction = session.beginTransaction();
    List mensagens = session.createQuery(
        "from Mensagem m order by m.texto asc").list();
    System.out.println(mensagens.size() + "mensagem(s) encontrada:");
    for(Iterator iterator = mensagens.iterator();iterator.hasNext();)
    {
        mensagem = (Mensagem) iterator.next();
        System.out.println(mensagem.getTexto());
    }
    transaction.commit();
    session.close();
}
```

Pequeno exemplo (algum código inicial)

O bloco seguinte de código permite-nos ver quais as mensagens presentes na base de dados:

Mensagem.java

```
public static void main(String[] args){

    ...

    Session session = sessions.openSession();
    Transaction transaction = session.beginTransaction();
    List mensagens = session.createQuery(
        "from Mensagem m order by m.texto asc").list();
    System.out.println(mensagens.size() + "mensagem(s) encontrada:");
    for(Iterator iterator = mensagens.iterator();iterator.hasNext();)
    {
        mensagem = (Mensagem) iterator.next();
        System.out.println(mensagem.getTexto());
    }
    transaction.commit();
    session.close();
}
```

Interface `Query`:

O interface `Query` permite realizar consultas à base de dados e controlar o modo como essa consulta é feita.

As consultas são escritas em HQL (*Hibernate Query Language*) ou no dialecto SQL nativo da base de dados.

Esta *query* está escrita em HQL, pelo que se especifica o nome da classe que se mapeia bem como o nome da sua propriedade, e não o nome da tabela e da coluna.

Pequeno exemplo (algum código inicial)

O bloco seguinte de código permite-nos ver quais as mensagens presentes na base de dados:

Mensagem.java

```
public static void main(String[] args){

    ...

    Session session = sessions.openSession();
    Transaction transaction = session.beginTransaction();
    List mensagens = session.createQuery(
        "from Mensagem m order by m.texto asc").list();
    System.out.println(mensagens.size() + "mensagem(s) encontrada:");
    for(Iterator iterator = mensagens.iterator();iterator.hasNext();)
    {
        mensagem = (Mensagem) iterator.next();
        System.out.println(mensagem.getTexto());
    }
    transaction.commit();
    session.close();
}
```

Esta consulta é traduzida internamente no SQL seguinte quando `createQuery()` é chamada:

```
select m.MENSAGEM_ID, m.MENSAGEM_TEXTO, m.PROXIMA_MENSAGEM_ID
from MENSAGENS m
order by m.MENSAGEM_TEXTO asc
```

resultado

```
1 mensagem(s) encontrada:
Olá Mundo
```

Pequeno exemplo (algum código inicial)

Finalmente vamos alterar a nossa primeira mensagem e criar uma nova associada à primeira:

Mensagem.java

```
public static void main(String[] args){

    ...

    Session session = sessions.openSession();
    Transaction transaction = session.beginTransaction();
    mensagem = (Mensagem) session.load(Mensagem.class, new Long(1));
    mensagem.setTexto("Saudações terrestre");
    Mensagem nova_mensagem = new Mensagem("Leva-me ao teu líder
     (por favor)");
    mensagem.setProximaMensagem(nova_mensagem);
    transaction.commit();
    session.close();
}
```

Outro tipo de consulta (*query*).

É lida da base de dados a `mensagem` com o `id == 1`

Pequeno exemplo (algum código inicial)

Finalmente vamos alterar a nossa primeira mensagem e criar uma nova associada à primeira:

Mensagem.java

```
public static void main(String[] args){

    ...

    Session session = sessions.openSession();
    Transaction transaction = session.beginTransaction();
    mensagem = (Mensagem) session.load(Mensagem.class, new Long(1));
    mensagem.setTexto("Saudações terrestre");
    Mensagem nova_mensagem = new Mensagem("Leva-me ao teu líder
     (por favor)");
    mensagem.setProximaMensagem(nova_mensagem);
    transaction.commit();
    session.close();
}
```

Actualização da `mensagem`.
Resultará num comando `update`.

Pequeno exemplo (algum código inicial)

Finalmente vamos alterar a nossa primeira mensagem e criar uma nova associada à primeira:

Mensagem.java

```
public static void main(String[] args){

    ...

    Session session = sessions.openSession();
    Transaction transaction = session.beginTransaction();
    mensagem = (Mensagem) session.load(Mensagem.class, new Long(1));
    mensagem.setTexto("Saudações terrestre");
    Mensagem nova_mensagem = new Mensagem("Leva-me ao teu líder
     (por favor)");
    mensagem.setProximaMensagem(nova_mensagem);
    transaction.commit();
    session.close();
}
```

Criação de uma nova mensagem.
Atenção:
Este objecto ainda não é persistente.

Pequeno exemplo (algum código inicial)

Finalmente vamos alterar a nossa primeira mensagem e criar uma nova associada à primeira:

Mensagem.java

```
public static void main(String[] args){

    ...

    Session session = sessions.openSession();
    Transaction transaction = session.beginTransaction();
    mensagem = (Mensagem) session.load(Mensagem.class, new Long(1));
    mensagem.setTexto("Saudações terrestre");
    Mensagem nova_mensagem = new Mensagem("Leva-me ao teu líder
     (por favor)");
    mensagem.setProximaMensagem(nova_mensagem);
    transaction.commit();
    session.close();
}
```

Actualização do campo `proxima_mensagem` de `mensagem`. Agora sim, a `nova_mensagem` é tornada persistente.

2 em 1: fará mais um `update` (será?) e um `insert`.

Pequeno exemplo (algum código inicial)

Finalmente vamos alterar a nossa primeira mensagem e criar uma nova associada à primeira:

Mensagem.java

```
public static void main(String[] args){

    ...

    Session session = sessions.openSession();
    Transaction transaction = session.beginTransaction();
    mensagem = (Mensagem) session.load(Mensagem.class, new Long(1));
    mensagem.setTexto("Saudações terrestre");
    Mensagem nova_mensagem = new Mensagem("Leva-me ao teu líder
     (por favor)");
    mensagem.setProximaMensagem(nova_mensagem);
    transaction.commit();
    session.close();
}
```

O código SQL gerado é o seguinte:

```
select m.MENSAGEM_ID, m.MENSAGEM_TEXTO, m.PROXIMA_MENSAGEM_ID from MENSAGENS m where m.MENSAGEM_ID = 1

insert into MENSAGENS (MENSAGEM_ID, MENSAGEM_TEXTO, PROXIMA_MENSAGEM_ID)
values (2, 'Leva-me ao teu líder (por favor)', null)

update MENSAGENS set MENSAGENS_TEXT = 'Saudações terrestre', PROXIMA_MENSAGEM_ID = 2 where MENSAGEM_ID = 1
```

Pequeno exemplo (algum código inicial)

Finalmente vamos alterar a nossa primeira mensagem e criar uma nova associada à primeira:

Mensagem.java

```
public static void main(String[] args){

    ...

    Session session = sessions.openSession();
    Transaction transaction = session.beginTransaction();
    mensagem = (Mensagem) session.load(Mensagem.class, new Long(1));
    mensagem.setTexto("Saudações terrestre");
    Mensagem nova_mensagem = new Mensagem("Leva-me ao teu líder
     (por favor)");
    mensagem.setProximaMensagem(nova_mensagem);
    transaction.commit();
    session.close();
}
```

Apenas um `update`. As duas operações de actualização foram unificadas numa só instrução.

O *Hibernate* detecta as modificações ao `texto` e à `próxima_mensagem` e automaticamente actualiza a base de dados.

As instruções SQL geradas não são, portanto, executadas imediatamente, mas são guardadas em cache e eventualmente modificadas.

O código SQL gerado é o seguinte:

```
select m.MENSAGEM_ID, m.MENSAGEM_TEXTO, m.PROXIMA_MENSAGEM_ID from MENSAGENS m where m.MENSAGEM_ID = 1

insert into MENSAGENS (MENSAGEM_ID, MENSAGEM_TEXTO, PROXIMA_MENSAGEM_ID)
values (2, 'Leva-me ao teu líder (por favor)', null)

update MENSAGENS set MENSAGENS_TEXT = 'Saudações terrestre', PROXIMA_MENSAGEM_ID = 2 where MENSAGEM_ID = 1
```

Pequeno exemplo (algum código inicial)

Finalmente vamos alterar a nossa primeira mensagem e criar uma nova associada à primeira:

Mensagem.java

```
public static void main(String[] args){

    ...

    Session session = sessions.openSession();
    Transaction transaction = session.beginTransaction();
    mensagem = (Mensagem) session.load(Mensagem.class, new Long(1));
    mensagem.setTexto("Saudações terrestre");
    Mensagem nova_mensagem = new Mensagem("Leva-me ao teu líder
     (por favor)");
    mensagem.setProximaMensagem(nova_mensagem);
    transaction.commit();
    session.close();
}
```

Da mesma forma que não foi necessário indicar ao *Hibernate* que era necessário actualizar a base de dados, também o processo de tornar a `nova_mensagem` persistente foi automático a partir do momento em que a primeira mensagem a referencia.

Esta funcionalidade é chamada *cascading save*. Poupa-nos o trabalho de chamar o método `save()`, sempre que possa ser alcançada por uma instância já persistente.

O código SQL gerado é o seguinte:

```
select m.MENSAGEM_ID, m.MENSAGEM_TEXTO, m.PROXIMA_MENSAGEM_ID from MENSAGENS m where m.MENSAGEM_ID = 1

insert into MENSAGENS (MENSAGEM_ID, MENSAGEM_TEXTO, PROXIMA_MENSAGEM_ID)
values (2, 'Leva-me ao teu líder (por favor)', null)

update MENSAGENS set MENSAGENS_TEXT = 'Saudações terrestre', PROXIMA_MENSAGEM_ID = 2 where MENSAGEM_ID = 1
```

Finalmente

- Agradecimentos:

  Paulo Zenida ;)