



Universidade do Minho
Departamento de Informática

Engenharia de Aplicações

ORM: Hibernate
Aula 2 - Anotações

António Nestor Ribeiro
anr@di.uminho.pt

Introdução

- A transformação Object-Relational em Hibernate necessita:
 - existência de metadados que regulem a transformação
 - criação de ficheiros XML com a informação relativa a cada conceito persistente e seus relacionamentos (associações e hierarquia)
- Uma forma mais actual de fornecer esta informação consiste na utilização de anotações
- Anotações são elementos standard JDK e dispensa a utilização de ficheiros XML para as regras de transformação
 - mantêm toda a informação importante ao nível do código Java
- Existem várias alternativas de assegurar persistência desta forma, embora as mais comuns sejam:
 - EJB 3
 - Hibernate

Introdução

- O EJB3 também apresenta uma alternativa para a persistência “out of the box”
 - inclui o EJB ORM standard
- Os arquitectos do Hibernate estiveram presentes na definição do standard, logo
 - o Hibernate teve grande influência no EJB 3.0
 - o Hibernate suporta a parte relativa ao ORM do EJB 3.0 (acrescenta contudo extensões proprietárias)
- O developer pode escolher a alternativa a utilizar:
 - persistência EJB 3
 - EJB 3 com utilização de Hibernate
 - apenas Hibernate

Configuração

- Não existe a necessidade de utilizar ficheiros XML, podendo todas as configurações serem feitas ao nível do código
- Declaração das entidades que vão ser alvo de persistência:

```
sessionFactory = new AnnotationConfiguration()
    .addPackage("test.animals") //the fully qualified package name
    .addAnnotatedClass(Flight.class)
    .addAnnotatedClass(Sky.class)
    .addAnnotatedClass(Person.class)
    .addAnnotatedClass(Dog.class)
    .addResource("test/animals/orm.xml")
    .configure().buildSessionFactory();
```

- embora se possa misturar a utilização de ficheiros XML com a escrita de anotações no código
 - não deve existir ambiguidade, para uma mesma classe, de qual é o método que se está a utilizar.

```
<mapping class="pt.uminho.di.aa.Flight"/>
<mapping class="pt.uminho.di.aa.Sky"/>
<mapping class="pt.uminho.di.aa.Person"/>
<mapping class="pt.uminho.di.aa.Dog"/>
```

Mapeamento de Entidades

- **Nota:** utilização de mapeamentos EJB/JPA (comuns a EJB e a Hibernate)
 - cada classe (entity class) é um POJO (Pure Old Java Object) que corresponde a cada uma das entidades que deve ser persistida
 - as anotações podem ser categorizadas em:
 - **anotações lógicas**, descrevendo o modelo de objectos e respectivas associações e relacionamento entre objectos
 - **anotações físicas**, descrevendo o layout físico do suporte relacional escolhido

```
@Entity
public class Flight implements Serializable {
    Long id;

    @Id
    public Long getId() {
        return id;
    }

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

```
@Entity
public class Flight implements Serializable {
    @Id
    Long id;

    public Long getId() {
        return id;
    }

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

Mapeamento de Entidades

- A definição física do nome da tabela associada a uma entidade pode também ser fornecida

```
@Entity  
@Table(name="tbl_sky")  
public class Sky implements Serializable {  
    ...
```

- Definição do nome da coluna da tabela associada a uma propriedade

```
@Entity  
public class Flight implements Serializable {  
    ...  
    @Version  
    @Column(name="OPTLOCK")  
    public Integer getVersion() { ... }  
}
```

```
@Entity  
public class Flight implements Serializable {  
    @Version  
    @Column(name="OPTLOCK")  
    private Integer version;  
    ...  
}
```

Mapeamento de Entidades

- Cada propriedade, não transiente, de uma entidade (de uma classe) é considerada persistente
 - exceptuando se for precedida de `@Transient`

```
private String firstname; //persistent property

@Transient
String getLengthInMeter() { ... } //transient property

String getName() {...} // persistent property

@Basic
int getLength() { ... } // persistent property

@Basic(fetch = FetchType.LAZY)
String getDetailedComment() { ... } // persistent property

@Temporal(TemporalType.TIME)
java.util.Date getDepartureTime() { ... } // persistent property

@Enumerated(EnumType.STRING)
Starred getNote() { ... } //enum persisted as String in database
```

Mapeamento de Entidades

- Declaração dos atributos de uma coluna:
 - não anotado
 - @Basic, se for de tipo simples
 - @Version, se for necessário garantir gestão de versão dos objectos
 - @Lob, para blobs e clobs
 - @Temporal, para dados que representem data/hora e é necessário gerir o formato e a precisão da informação
 - @org.hibernate.annotations.CollectionOfElements, das extensões adicionais Hibernate

```
@Entity
public class Flight implements Serializable {
...
@Column(updatable = false, name = "flight_name", nullable = false, length=50)
public String getName() { ... }
```

Mapeamento de Entidades

- Composição de Objectos

```
@Entity
public class Person implements Serializable {

    // Persistent component using defaults
    Address homeAddress;

    @Embedded
    @AttributeOverrides( {
        @AttributeOverride(name="iso2", column = @Column(name="bornIso2") ),
        @AttributeOverride(name="name", column = @Column(name="bornCountryName") )
    })
    Country bornIn;
    ...
}
```

Mapeamento de Objectos

```
@Embeddable  
public class Address implements Serializable {  
    String city;  
    Country nationality; //no overriding here  
}
```

```
@Embeddable  
public class Country implements Serializable {  
    private String iso2;  
    @Column(name="countryName") private String name;  
  
    public String getIso2() { return iso2; }  
    public void setIso2(String iso2) { this.iso2 = iso2; }  
  
    public String getName() { return name; }  
    public void setName(String name) { this.name = name; }  
    ...  
}
```

Mapeamento de Objectos

- Definição de chaves primárias compostas
 - anotar a propriedade do contentor como **@Id** e o contido como **@Embeddable**
 - anotar a propriedade do contentor como **@EmbeddedId**
 - anotar o contentor como **@IdClass** e anotar cada propriedade da entidade com **@Id**. Exemplo:

```
@Entity
@IdClass(FootballerPk.class)
public class Footballer {
    //part of the id key
    @Id public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    //part of the id key
    @Id public String getLastname() {
        return lastname;
    }

    public void setLastname(String lastname) {
        this.lastname = lastname;
    }
}
```

Mapeamento de Associações/Relações

- Um para Um (chave primária, chave estrangeira, ou tabela de associação)

```
@Entity
public class Body {
    @Id
    public Long getId() { return id; }

    @OneToOne(cascade = CascadeType.ALL)
    @PrimaryKeyJoinColumn
    public Heart getHeart() {
        return heart;
    }
    ...
}
```

```
@Entity
public class Heart {
    @Id
    public Long getId() { ...}
}
```

Mapeamento de Associações/Relações

- Um para Um, com chave estrangeira

```
@Entity
public class Customer implements Serializable {
    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name="passport_fk")
    public Passport getPassport() {
        ...
    }

@Entity
public class Passport implements Serializable {
    @OneToOne(mappedBy = "passport")
    public Customer getOwner() {
        ...
    }
```

- **mappedBy** refere o nome da propriedade da associação como vista do lado do dono da relação.

Mapeamento de Associações/Relações

- Muitos para um

```
@Entity()
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE} )
    @JoinColumn(name="COMP_ID")
    public Company getCompany() {
        return company;
    }
    ...
}
```

- a classe **Company** tem uma propriedade designada por **id**

Mapeamento de Associações/Relações

- Um para muitos
 - bidireccional
 - unidireccional

```
@Entity
public class Troop {
    @OneToMany(mappedBy="troop")
    public Set<Soldier> getSoldiers() {
        ...
    }

@Entity
public class Soldier {
    @ManyToOne
    @JoinColumn(name="troop_fk")
    public Troop getTroop() {
        ...
    }
```

- uma instância de **Troop** tem muitas instâncias de **Soldier**

Mapeamento de Associações/Relações

- unidireccional (o mais utilizado!)

```
@Entity  
public class Customer implements Serializable {  
    @OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)  
    @JoinColumn(name="CUST_ID")  
    public Set<Ticket> getTickets() {  
        ...  
    }  
}
```

```
@Entity  
public class Ticket implements Serializable {  
    ... //no bidir  
}
```

Mapeamento de Associações/Relações

- Muitos para muitos

```
@Entity
public class Employer implements Serializable {
    @ManyToMany(
        targetEntity=org.hibernate.test.metadata.manytomany.Employee.class,
        cascade={CascadeType.PERSIST, CascadeType.MERGE}
    )
    @JoinTable(
        name="EMPLOYER_EMPLOYEE",
        joinColumns=@JoinColumn(name="EMPER_ID"),
        inverseJoinColumns=@JoinColumn(name="EMPEE_ID")
    )
    public Collection getEmployees() {
        return employees;
    }
    ...
}
```

```
@Entity
public class Employee implements Serializable {
    @ManyToMany(
        cascade = {CascadeType.PERSIST, CascadeType.MERGE},
        mappedBy = "employees",
        targetEntity = Employer.class
    )
    public Collection getEmployers() {
        return employers;
    }
}
```

Transitividade no acesso a informação

- Diferentes tipos de transitividade
 - CascadeType.PERSIST, se persist()
 - CascadeType.MERGE, se merge()
 - CascadeType.REMOVE, se delete()
 - CascadeType.REFRESH, se refresh()
 - CascadeType.ALL, todos os anteriores

Mapeamento de Herança

- Em EJB3 temos três tipos de descrição de relações de herança:
 - Table per Class
 - corresponde ao **<union-class>** do Hibernate
 - Single Table per Class Hierarchy,
 - corresponde ao **<subclass>** do Hibernate
 - Joined Subclass,
 - corresponde ao **<joined-subclass>** do Hibernate
- A definição de qual é o modelo de representação de hierarquia a considerar é feito
 - na entidade raiz da hierarquia
 - com a utilização da anotação **@Inheritance**

Mapeamento de Herança

- Table per Class

```
@Entity  
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)  
public class Flight implements Serializable {
```

- suporta associações de um para muitos, desde que estas sejam bidireccionais
- é mandatório que não se utilize um esquema de geração de chaves, visto que estas tem de ser partilhadas com outras tabelas
- estratégia complexa, principalmente quando a aplicação pretende tirar partido de polimorfismo. O espaço de procura não está contíguo e tem de se percorrer todas as tabelas.

Mapeamento de Herança

- Single Table per Class Hierarchy
 - como todas as tabelas de uma hierarquia estão mapeadas na mesma tabela, é necessário um discriminador para distinguir as diferentes instâncias

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name="planetype",
    discriminatorType=DiscriminatorType.STRING
)
@DiscriminatorValue("Plane")
public class Plane { ... }

@Entity
@DiscriminatorValue("A320")
public class A320 extends Plane { ... }
```

Mapeamento de Herança

- Joined Subclasses

```
@Entity  
@Inheritance(strategy=InheritanceType.JOINED)  
public class Boat implements Serializable { ... }
```

```
@Entity  
public class Ferry extends Boat { ... }
```

```
@Entity  
@PrimaryKeyJoinColumn(name="BOAT_ID")  
public class AmericaCupClass extends Boat { ... }
```

- em que **Boat.id = AmericaCupClass.BOOT_ID**

Mapeamento de Herança

- Classes como elemento de agregação de propriedades
 - existem classes, por vezes abstractas, que são o local de colocação de atributos para serem herdados pelas subclasses
 - não existe a necessidade de assegurar persistência para essas classes

```
@MappedSuperclass
public class BaseEntity {
    @Basic
    @Temporal(TemporalType.TIMESTAMP)
    public Date getLastUpdate() { ... }
    public String getLastUpdater() { ... }
    ...
}

@Entity class Order extends BaseEntity {
    @Id public Integer getId() { ... }
    ...
}
```

Geração do esquema

- É possível indicar e configurar o Hibernate de modo a usar um esquema relacional existente.
 - Útil para quando queremos migrar uma aplicação existente para Hibernate, ou,
 - Quando necessitamos de maior controlo sobre a base de dados.
- É possível no entanto, solicitar ao Hibernate que gere/actualize o esquema de acordo com as nossas anotações.

```
<property name="hibernate.hbm2ddl.auto">create</property>
```

```
<property name="hibernate.hbm2ddl.auto">update</property>
```

Persistência de informação

- A interacção com a persistência é feita através de uma sessão Hibernate.
- O Hibernate disponibiliza métodos para obter e guardar informação directamente.

```
Session s = new Session(...);  
Dog aDog = new Dog(...);  
s.save(aDog);
```

```
Flight myFlight = (Flight) s.get(Flight.class, id);
```

- O Hibernate disponibiliza ainda o HQL – Hibernate Query Language, para executar queries mais expressivas.

```
s.createQuery("from Book where title='MyBook' ");
```

Hibernate vs EJB3

- Hibernate integra com todas as ferramentas, embora possua uma configuração algo complexa
- EJB trabalha essencialmente com JEE
- Compatibilidade e interoperabilidade entre ambas as soluções. Afinal o Hibernate Core tem as especificações EJB3
- Ambos requerem que as classes tenham getters, setters e construtores vazios.
- Hibernate suporta mais colecções que o EJB 3. Hibernate suporta todas as interfaces JDK.
- Ambos suportam as estratégias de mapeamento de herança.

Exercício

1. Converter o exemplo de persistência da aula anterior para passar a utilizar anotações
2. Comparar as duas alternativas