CitiusTech

Mapping Relations and Collection of Value Types

Session 4

Version 1.2

September 2018

Agenda

- **Collection Mapping**
- Mapping Relations

Relationship Types

- Association
  - Mapping relationships between two objects
  - Example – Account and AccountOwner
- Collection
  - Collection of values representing individual pieces of data
  - Example: Set of accounts or String array of months

Hibernate Collection Mapping

- The Hibernate mapping element used for mapping a collection depends upon the type of interface
  - For example, a `<set>` element is used for mapping properties of type Set
- Collections can contain almost any other Hibernate type, including: basic types, custom types, components and references to other entities
- The contained type is referred to as the collection element type.
- Collection elements are mapped by `<element>` or `<composite-element>`, or in the case of entity references, with `<one-to-many>` or `<many-to-many>`
- Mapping Collection Of Value Types
  - An object of value type has no database entity, it belongs to an entity instance
  - If an entity class has collection of value types an additional table called collection table is needed

Hibernate Collection Types

- **<set>**
  - Unordered/Ordered, requiring value column
- **<map>**
  - Unordered/Ordered, requiring key and value columns
- **<list>**
  - Ordered, require an index column on the referenced object table
- **<array>**
  - Map to Java primitive arrays
  - Ordered, require an index column on the referenced object table
- **<bag>**
  - No direct implementation available in Java
  - Unordered/ordered collection allowing duplicates
  - Requires value column
- **<idbag>**
  - Used for many-to-many relationships
  - Same as Bag, with additional identifier column used for surrogate keys

Collection Mapping as Set

- A `java.util.Set` is mapped with a <set> element
- Initialize the collection with a `java.util.HashSet`
- The order of its elements isn't preserved, and duplicate elements aren't allowed

```
<class name="item" table="Item">
 <id name="id" type="int" column="item_id">
 <generator class="assigned"/>
 </id>
 <property name="name"/>
 <set name="images" table="item_image">
  <key column="item_id"/>
   <element column="filename" not-null="true" type="string"/>
 </set>
 </class>
```

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	FILENAME
1	fooimage1.jpg
1	fooimage2.jpg
2	barimage1.jpg

Collection Mapping as List

- A List is a collection of ordered elements that permits duplicates
- A list mapping requires addition of index column to collection table
- The index column defines the position of the element in the collection

```
<class name="list.ListParent" table="Item">
    <id name="item_id" type="int">
      <generator class="assigned"/>
    </id>
    <property name="name"/>
    <list name="listImages" table="item_image">
      <key column="item_id" />
      <list-index column="Position"/>
      <element type="string" column="filename" not-null="true"/>
    </list>
    </class>
```

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	FILENAME
1	fooimage1.jpg
1	fooimage2.jpg
2	barimage1.jpg

Collection Mapping as Map

- A `java.util.Map` can be mapped with <map>, preserving key and value pairs
- Use a `java.util.HashMap` to initialize a property
- A `java.util.SortedMap` can be mapped with <map> element, and the sort attribute can be set to either a comparator or natural ordering for in-memory sorting
- Initialize the collection with a `java.util.TreeMap` instance

```
<class name="map.MapParent“   table=“item">
      <id name=“item_id" type="int">
    <generator class="assigned"/>
  </id>
  <property name=“name"/>
  <map name="mapImages" table="item_image">
    <key column="item_id“ />
    <map-key column="imagename“ type="string“ />
    <element type="string" column="filename" not-null="true“ />
  </map>
 </class>
```

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	IMAGENAME	FILENAME
1	Image One	fooimage1.jpg
1	Image Two	fooimage2.jpg
1	Image Three	foomage3.jpg

Collection Mapping as Bag

- An unordered collection that permits duplicate elements is called a Bag
- A `java.util.Collection` can be mapped with `<bag>` or `<idbag>` as java doesn't have a Bag interface or an implementation
- Hibernate supports persistent bags (it uses lists internally but ignores the index of the elements)
- Use a `java.util.ArrayList` to initialize a bag collection

```
<class name="bag.BagParent" table="item">
   <id name="item_id" type="int">
    <generator class="assigned"/>
   </id>
   <property name="name"/>
   <bag name="bagImages" table="item_image" >
        <key column="item_id" />
      <element type="string" column="filename" not-null="true"/>
   </bag>
 </class>
```

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	FILENAME
1	fooimage1.jpg
1	fooimage2.jpg
2	barimage1.jpg

Collection Mapping as IdBag

```
<class name="idbag.BagParent" table="item">
   <id name="item_id" type="int">
     <generator class="assigned"/>
   </id>
   <property name="name"/>
   <idbag name="bagImages" table="item_image">
   <collection-id type="string" column="item_image_id">
   <generator class="uuid"/>
                </collection-id>
                <key column="item_id"/>
   <element type="string" column="filename" not-null="true"/>
   </idbag>

 </class>
```

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_IMAGE_ID	ITEM_ID	FILENAME
1	1	fooimage1.jpg
2	1	fooimage1.jpg
3	3	barimage1.jpg

Agenda

- Collection Mapping
- **Mapping Relations**

Associations

- Associations describe how two or more entities form a relationship based on a database joining semantics
- Association relations can be classified as:
  - Based on Multiplicity
    - One-to-one
    - Many-to-one
    - One-to-many
    - Many-to-Many
  - Unidirectional associations / Bidirectional associations

Relationships in Java

- Object has an attribute referencing another related object
- For ‘Many’ side, collection API uses:
  - Set
  - Map
  - List
  - Arrays

- For implementing collections in Domain objects:
  - Must always use interfaces
  - Should be instantiated right away, instead of delegating to constructor or setter methods

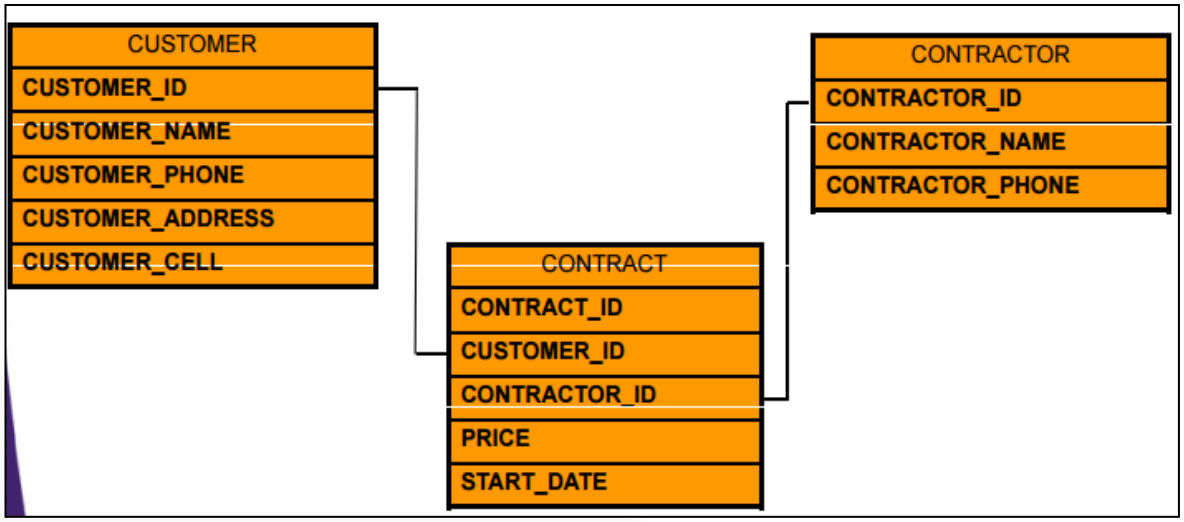
```
List mylist = new ArrayList()
```

Relationships in Database

- **Denormalized Tables**: Easy to query data

ID	CUSTOMER_NAME	CUSTOMER_PHONE	CUSTOMER_ADDRESS	CONTRACTOR_NAME	CONTRACTOR_PHONE	CONTRACT_PRICE	START_DATE
3	CustomerOne	634-222-4572	100 Main Street	Home Depot	800-HOMEDEPOT	275	06-JUL-08 10.23.16.000000 PM
2	CustomerOne	634-222-4572	100 Main Street	Home Depot	800-HOMEDEPOT	4500	19-JUL-08 10.22.57.000000 PM
5	CustomerOne	634-222-4572	100 Main Street	Lowes	888-LOWES-SVC	300	05-SEP-08 10.23.49.000000 PM
1	CustomerOne	634-222-4572	100 Main Street	Home Depot	800-HOMEDEPOT	1000	19-JUN-08 10.22.37.000000 PM
7	CustomerTwo	470-987-9988	275 South Peach Avenue	Home Depot	800-HOMEDEPOT	349	26-AUG-08 10.24.37.000000 PM
6	CustomerTwo	470-987-9988	275 South Peach Avenue	Lowes	888-LOWES-SVC	600	16-JUL-08 10.24.18.000000 PM
8	CustomerTwo	470-987-9988	275 South Peach Avenue	Home Depot	800-HOMEDEPOT	975	07-SEP-08 10.24.46.000000 PM
4	CustomerTwo	470-987-9988	275 South Peach Avenue	Lowes	888-LOWES-SVC	150	14-AUG-08 10.23.33.000000 PM

- **Join Tables**: Built on foreign key model and enables M:M relationships



Setting up Relationships

- Determine your domain model relationships
  - Define each as an association or collection
  - Identify the multiplicity between objects
  - Decide on the directionality
- Define your relationship implementation types and add the appropriate interface
- Create the mapping in the corresponding object mapping files
- If bidirectional, optionally add bidirectional maintenance code

Association Mapping: One-to-One (1/5)

- Can either be unidirectional or bidirectional
- A unidirectional association follows the relational database foreign key semantics, the client-side owning the relationship
- A bidirectional association features a **mappedBy @OneToOne** parent side too
- Consider the relationship between Phone and Phone Details using unidirectional one-to-one mapping:

Association Mapping: One-to-One (2/5)

- Following code snippet shows annotations for mapping one-to-one association:

```
@Entity(name = "Phone")
public class Phone {
        @Id
        @GeneratedValue
        private Long id;

        @Column(name = "number")
        private String number;

        @OneToOne(cascade = CascadeType.ALL)
        @JoinColumn (name = "details_id")
        private PhoneDetails details;

        public Phone() {
        }
        public Phone(String Number) {
        }


        // Setter & Getters
```

It is not necessary to specify the associated target entity explicitly, since it can be inferred from the type of the object being referenced.

@JoinColumn – defines the database column that will be mapped with the attribute.

Association Mapping: One-to-One (3/5)

- Following code snippet shows the **PhoneDetails** class:

```
@Entity(name = "PhoneDetails")
public class PhoneDetails {

private Long id;

private String provider;

private String technology;

public PhoneDetails() {
}

public PhoneDetails(Long id, String provider, String technology) {
this.id = id;
this.provider = provider;
this.technology = technology;
}
//getters and getters
}
```

Association Mapping: One-to-One (4/5)

- Following code snippet shows the persistence of the Phone and PhoneDetails objects into the database:

```
public class TestOneToOneAssociation {

 public static void main(String[] args) {

                Transaction  transaction = null;
                SessionFactory  factory = new
                Configuration().configure().buildSessionFactory();
                Session session = factory.openSession();
                Transaction transaction = session.beginTransaction();
                PhoneDetails details = new PhoneDetails(10L, "Google" , "Android" );
                Phone phone = new Phone("9876348923");
                phone.setDetails(details);
                session.save(phone);
                transaction.commit();
                session.close();
                factory.close();
}
```

Association Mapping: One-to-One (5/5)

- Following figures shows the creation of corresponding table structure in the database for one-to-one association:

```
CREATE TABLE Phone (
                        id BIGINT NOT NULL ,
                        number VARCHAR(255) ,
                        details_id BIGINT ,
                        PRIMARY KEY ( id )
            )
CREATE TABLE PhoneDetails (
                        id BIGINT NOT NULL ,
                        provider VARCHAR(255) ,
                        technology VARCHAR(255) ,
                        PRIMARY KEY ( id )
            )
ALTER TABLE Phone
ADD CONSTRAINT FKnoj7cj83ppfbv
FOREIGN KEY (details_id) REFERENCES PhoneDetails
```

- From a relational database point of view, the underlying schema is identical to the unidirectional **@ManyToOne** association
- Client-side controls the relationship based on the foreign key column

Bidirectional @OneToOne (1/2)

- For Bidirectional relationship, consider:
  - Phone as the parent-side
  - PhoneDetails as the child-side
- Pushing foreign key into the **PhoneDetails** table requires a bidirectional @OneToOne association

Bidirectional @OneToOne (2/2)

- Following code snippet shows the bidirectional one-to-one relationship:

```
@Entity(name = "Phone")
public static class Phone
{
 . . .
 @OneToOne(mappedBy = "phone", cascade = CascadeType.ALL,
 orphanRemoval = true, fetch = FetchType.LAZY)
  private PhoneDetails details;
 . . .
}
 . . .
public void addDetails(PhoneDetails details) {
      details.setPhone( this );
      this.details = details;
   }
. . .
@Entity(name = "PhoneDetails")
public static class PhoneDetails
{
 . . .
   @OneToOne(fetch = FetchType.LAZY)
   @JoinColumn(name = "phone_id")
   private Phone phone;
 . . .
}
```

Association Mapping: Many-to-Many (1/4)

- The @ManyToMany association requires a link table that joins two entities
- The @ManyToMany can be a either unidirectional or bidirectional
- Consider the relationship between Person and Address using unidirectional many-to-many mapping:


```
@Entity(name = "Person")
public class Person {
@Id
@GeneratedValue
private long id;

@ManyToMany(cascade = { CascadeType.PERSIST, CascadeType.MERGE })
private List<Address> addresses = new ArrayList<>();
public Person() {
}
public List<Address> getAddresses( ) {
      return addresses;
  }
}
```

Association Mapping: Many-to-Many (2/4)

Entity

```
@Entity(name = "Address")
public class Address {

@Id
@GeneratedValue
private Long id;

private String street;

@Column(name = "Number")
private String number;

public Address() {

}
public Address(Long id, String street, String number) {
this.id = id;
this.street = street;
this.number = number;
}
// setters & getters
```

Following figures shows the creation of corresponding table structure in the database for many-to-many association:

```
CREATE TABLE Address (
                        id BIGINT NOT NULL ,
                        number VARCHAR(255) ,
                        street VARCHAR(255) ,
                        PRIMARY KEY ( id )
            )
CREATE TABLE Person (
                        id BIGINT NOT NULL ,
                        PRIMARY KEY ( id )
            )

ALTER TABLE Persone_Address
ADD CONSTRAINT FKnoj7cj83ppfbv8e8hkjs08
FOREIGN KEY (address_id) REFERENCES Address

ALTER TABLE Persone_Address
ADD CONSTRAINT FKnoj7kjsddj4h5hk45hll
FOREIGN KEY (Person_id) REFERENCES Person
```

Association Mapping: Many-to-Many (3/4)

- A bidirectional @ManyToMany association has an owning and a mappedBy side
- To preserve synchronicity between both sides, it's good practice to provide helper methods for adding or removing child entities

```
@Entity(name = "Person")
public static class Person {
  . . .
  @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
 private List<Address> addresses = new ArrayList<>();

 . . .
}
. . .
public void addAddress(Address address) {
 addresses.add( address );
 address.getOwners().add( this );
}
```

Association Mapping: Many-to-Many (4/4)

```
@Entity(name = "Address")
public static class Address {
   @Id
   @GeneratedValue
   private Long id;
   private String street;
   @Column(name = "`number`")
   private String number;
   private String postalCode;

   @ManyToMany(mappedBy = "addresses")
   private List<Person> owners = new ArrayList<>();
   . . .
 }
```

Many-to-One Mapping (1/2)

- @ManyToOne is the most common association, having a direct equivalent in the relational database as foreign key
- It establishes a relationship between a child entity and a parent
- Consider the relationship between **Person** and **Phone** using unidirectional many-to-one mapping:


```
@Entity(name = "Person")
public class Person {
@Id
@GeneratedValue
private long id;

public Person() {
}
```

```
@Entity(name = "Phone")
public class Phone {
@Id
@GeneratedValue
private long id;
@Column(name = "number")
private String number;
@ManyToOne
@JoinColumn(name = "person_id",
        foreignkey = @ForeignKey(name =
"PERSON_ID_FK"))
private Person person;
public Phone() {
  }
}
```

Many-to-One Mapping (2/2)

- Following table structure is created for many-to-one mapping relationship in the database:

```
CREATE TABLE Person (
                id BIGINT NOT NULL ,
                PRIMARY KEY ( id )
                )
    CREATE TABLE Phone (
                id BIGINT NOT NULL ,
                number VARCHAR(255) ,
                details_id BIGINT ,
                PRIMARY KEY ( id )
                )
     ALTER TABLE Phone
     ADD CONSTRAINT PERSON_ID_FK
     FOREIGN KEY (person_id) REFERENCES
Person
```

- Each entity has a lifecycle of its own. Once the @ManyToOne association is set,Hibernate will set the associated database foreign key column

```
CREATE TABLE Address (
                                id BIGINT NOT NULL ,
                                number
VARCHAR(255) ,
                                street VARCHAR(255) ,
                                PRIMARY KEY ( id )
                )
CREATE TABLE Person (
                                id BIGINT NOT NULL ,
                                PRIMARY KEY ( id )
                )

ALTER TABLE Persone_Address
ADD CONSTRAINT FKnoj7cj83ppfbv8e8hkjs08
FOREIGN KEY (address_id) REFERENCES Address

ALTER TABLE Persone_Address
ADD CONSTRAINT FKnoj7kjsddj4h5hk45hll
FOREIGN KEY (Person_id) REFERENCES Person
```

One-to-Many Mapping

- The @OneToMany association links a parent entity with one or more child entities
- If the @OneToMany doesn’t have a mirroring @ManyToOne association on the child side, the @OneToMany association is unidirectional
- If there is a @ManyToOne association on the child side, the @OneToMany association is bidirectional
- The application developer can navigate this relationship from both ends

Unidirectional @OneToMany (1/2)

- When using a unidirectional `@OneToMany` association, Hibernate resorts to using a link table between the two joining entities
- Following class diagram depicts the one-to-many relationship:


```
@Entity(name = "Person")
public class Person {
@Id
@GeneratedValue
private long id;

@OneToMany(cascade = CascadeType.ALL ,
                orphanRemoval = true)
private List<Address> addresses = new ArrayList<>();

public Person() {
}
public List<Address> getAddresses( ) {
      return addresses;
 }
}
```

```
@Entity(name = "Phone")
public class Phone {
@Id
@GeneratedValue
private long id;
@Column(name = "number")
private String number;

public Phone() {
}
```

Unidirectional @OneToMany (2/2)

- Following diagram shows the corresponding table structure for the unidirectional one-to-many relationship:

```
CREATE TABLE Person (
                id BIGINT NOT NULL ,
                PRIMARY KEY ( id )
                )
CREATE TABLE Person_Phone (
                Person_id BIGINT NOT NULL ,
            phone_id BIGINT NOT NULL
                )
CREATE TABLE Phone (
                id BIGINT NOT NULL ,
                number VARCHAR(255) ,
                PRIMARY KEY ( id )
                )
ALTER TABLE Person_Phone
ADD CONSTRAINT UK_9uhc5itwc9h5gcg9485ofd
FOREIGN KEY (phones_id)

ALTER TABLE Person_Phone
ADD CONSTRAINT FKr45jlkj45j645lk7j45lkk
FOREIGN KEY (phones_id) REFERENCES Phone

ALTER TABLE Person_Phone
ADD CONSTRAINT FK343lkjh4564h3l23hl56h
FOREIGN KEY (Person_id) REFERENCES Person
```

- Following code snippet shows the cascading `@OneToMany` association

```
Person  person = new Person();
Phone  phone1 = new Phone("123-345-654");
Phone  phone2 = new Phone("534-564-567");
person.getPhone().add(phone1);
person.getPhone().add(phone2);

entityManager.persist( person);
entityManager.flush();

person.getPhone().remove( phone1);
```

Bidirectional @OneToMany (1/3)

- The bidirectional @OneToMany association also requires a @ManyToOne association on the child side
- Although the Domain Model exposes two sides to navigate this association, behind the scenes, the relational database has only one foreign key for this relationship
- Every bidirectional association must have one owning side only (the child side), the other one being referred to as the inverse (or the mappedBy) side

Bidirectional @OneToMany (2/3)

- Following example shows `@OneToMany` association mappedBy the `@ManyToOne` side:

```
@Entity(name = "Person")
public class Person {
@Id
@GeneratedValue
private long id;

@OneToMany(mappedBy = "person" ,cascade =
CascadeType.ALL , orphanRemoval = true)
private List<Phone> phones = new ArrayList<>();

public Person() {
}

public Person(Long id) {
this.id = id;
}

public List<Phone> getPhones() {
return phones;
}

public void addPhone(Phone phone) {
phones.add( phone );
phone.setPerson( this );
}
}
```

```
@Entity(name = "Phone")
public class Phone {
@Id
@GeneratedValue
private long id;

@NaturalId
@Column(name = "number" , unique
= true)
private String number;

@ManyToOne
private Person person;

public Phone() {
}
```

Bidirectional @OneToMany (3/3)

- Following diagram shows the corresponding table structure for the bidirectional `@OneToMany` relationship:

```
CREATE TABLE Person (
            id BIGINT NOT NULL ,
            PRIMARY KEY ( id )
            )

 CREATE TABLE Phone (
            id BIGINT NOT NULL ,
            number VARCHAR(255) ,
            person_id  BIGINT ,
            PRIMARY KEY ( id )
            )

ALTER TABLE Phone
ADD CONSTRAINT UK_9uhc5itwc9h5gcg9485ofd
UNIQUE (number)

ALTER TABLE Phone
ADD CONSTRAINT FKr45jlkj45j645lk7j45lkk
FOREIGN KEY (person_id) REFERENCES Person
```

Thank You

CitiusTech Markets

CitiusTech Services

CitiusTech Platforms

Accelerating Innovation

CitiusTech Contacts

Email univerct@citiustech.com

www.citiustech.com