Mapping simple key-value pairs

The basic mappings are pretty simple. JPA and Hibernate allow you to map a *Collection* of elements or an entity association to a *java.util.Map*. The key and value of the *Map* can be a basic type, an embeddable class or an entity.

You might have recognized that *Collections* are not supported. If you want to map your association to a *Map<YourKeyClass, List<YourValueClass>>*, you need to use a small workaround that I show you at the end of this post.

Represent a to-Many Association as a Map<String, EntityClass>

If you want to represent the association with a *Map* instead of the *List*, you just have to do 2 things:

1. change the type of the attribute from *List* to *Map* and
2. define the database column you want to use as a map key.

If you don't want to one of the attributes of the associated entity as a map key, you can also use:

1. a basic type with a *@MapKeyColumn* annotation
2. an enum with a *@MapKeyEnumerated* annotation
3. a *java.util.Date* or a *java.util.Calendar* with a *@MapKeyTemporal* annotation

Hibernate will persist the map key in an additional database column.

```
@Entity
public class Author {

    @ManyToMany
    @JoinTable(
        name="AuthorBookGroup",
        joinColumns={@JoinColumn(name="fk_author",
                referencedColumnName="id")},
        inverseJoinColumns={
            @JoinColumn(name="fk_group",
                referencedColumnName="id")})
    @MapKey(name = "title")
    private Map<String, BookGroup> bookGroups =
        new HashMap<String, BookGroup>();
```

Working with *Map*s instead of *List*s

That's all you need to do to define the mapping. You can now use the map in your domain model to add, retrieve or remove elements from the association.

How to map an association as a java.util.Map

```
Author a = new Author();
a.setFirstName("Thorben");
a.setLastName("Janssen");
em.persist(a);

Book b = new Book();
b.setTitle("Hibernate Tips");
b.setFormat(Format.PAPERBACK);
b.getAuthors().add(a);
em.persist(b);

a.getBooks().put(b.getTitle(), b);
```

You also have 2 options to use the association in JPQL query. You can simply join the 2 entities and use them as a regular many-to-many relationship.

```
TypedQuery<Author> q = em.createQuery(
"SELECT a FROM Author a JOIN a.books b “
+ “WHERE b.title LIKE :title", Author.class);
```

Or you can use the *KEY* function to reference the map key in your query.

```
TypedQuery<Author> q = em.createQuery(
"SELECT a FROM Author a JOIN a.books b “
+ “WHERE KEY(b) LIKE :title ", Author.class);
```

Represent a Collection of Embeddables as a Map<EnumClass, EntityClass>

The mapping for embeddable classes is pretty similar. The main difference is that you need to use the *@ElementCollection* annotation instead of a *@ManyToMany* annotation.

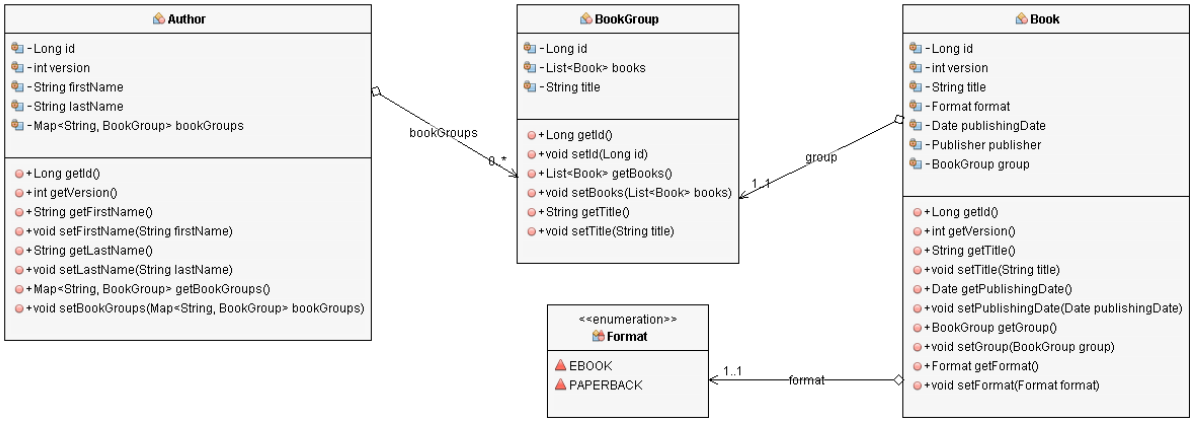
```
@Entity
public class Author {

	@ElementCollection
	@MapKeyColumn(name = "address_type")
	@MapKeyEnumerated(EnumType.STRING)
	private Map<AddressType, Address>address =
		new HashMap<AddressType, Address>();

	...
}
```

Mapping multiple values to the same key

As you've seen in the previous examples, JPA and Hibernate don't support any *Collection* type as a map value. That's unfortunate because, in my experience, that is the most common use case.

But there is a small and easy workaround for these situations. You just need to introduce a wrapper entity that wraps the *List* or *Map* of entities.

How to map an association as a java.util.Map



The mapping definition of that entity is very simple. You just need a primary key and a one-to-many association to the *Book* entity. I also introduced a *title* for the *BookGroup* which I want to use as the map key on the *Author* entity.

```
@Entity
public class BookGroup {

	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	@Column(name = "id", updatable = false,
			nullable = false)
	private Long id;

	@OneToMany(mappedBy = "group")
	private List<Book> books = new ArrayList<Book>();

	private String title;

	...
}
```

You can then model the association between the BookGroup and the Author entity as a Map.

```
@Entity
public class Author {

	@ManyToMany
	@JoinTable(
		name="AuthorBookGroup",
		joinColumns={@JoinColumn(name="fk_author",
				referencedColumnName="id")},
		inverseJoinColumns={
			@JoinColumn(name="fk_group",
				referencedColumnName="id")})
	@MapKey(name = "title")
	private Map<String, BookGroup> bookGroups =
		new HashMap<String, BookGroup>();

	...
}
```

That's all you need to do to model an association as a *Map* with multiple values per map key. You can now use it in the same way as I showed you in the first example.

Please be aware, that this mapping creates overhead and redundancy. The *BookGroup* entity gets mapped to a database table which we didn't need in the previous examples. And I now use the title of the *BookGroup* as the key of the *Map<String, BookGroup> bookGroups*. The *BookGroup* title will be the same as the *title* of all *Books* in the group.