Hibernate (ORM) - детальный разбор

1. Сущности (Entities) и маппинг

1.1. Базовые аннотации

`@Entity`

Помечает класс как сущность БД. Требует пустого конструктора.

```
@Entity
public class User {
    // поля и методы
}
```

`@Table`

Задает имя таблицы (если отличается от имени класса).

```
@Entity
@Table(name = "app_users")
public class User {
    // ...
}
```

`@Id`

Определяет первичный ключ.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

Стратегии генерации ID:

- `AUTO` (по умолчанию, зависит от БД)
- `IDENTITY` (MySQL auto_increment)
- `SEQUENCE` (Oracle, PostgreSQL)
- `TABLE` (эмуляция через отдельную таблицу)

1.2. Маппинг полей

`@Column`

Настройка колонки:

```
@Column(name = "user_name", nullable = false, length = 50)
private String username;
```

`@Enumerated`

Для enum-полей:

```
@Enumerated(EnumType.STRING) // или ORDINAL (числовое значение)
private UserRole role;
```

`@Temporal`

Для дат (в Java 8+ лучше использовать `java.time.*`):

```
@Temporal(TemporalType.DATE)
private Date birthDate;
```

`@Lob`

Для больших объектов (BLOB/CLOB):

```
@Lob
private byte[] avatar;
```

---

2. Связи между сущностями

2.1. `@OneToMany` и `@ManyToOne`

Пример: Пользователь и Заказы (1-N)

```
@Entity
public class User {
    @Id
    private Long id;

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Order> orders = new ArrayList<>();
}

@Entity
public class Order {
    @Id
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "user_id")
    private User user;
}
```

Ключевые параметры:

- `mappedBy` - указывает поле-владельца связи.
- `cascade` - какие операции каскадируются (ALL, PERSIST, MERGE и др.).
- `orphanRemoval` - автоматическое удаление "осиротевших" объектов.
- `fetch` - стратегия загрузки (LAZY или EAGER).

2.2. `@ManyToMany`

Пример: Студенты и Курсы (N-M)

```
@Entity
public class Student {
    @Id
    private Long id;

    @ManyToMany
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id")
    )
    private Set<Course> courses = new HashSet<>();
}

@Entity
public class Course {
    @Id
    private Long id;

    @ManyToMany(mappedBy = "courses")
    private Set<Student> students = new HashSet<>();
}
```

Особенности:

- Требуется промежуточная таблица (`@JoinTable`).
- Владелец связи определяется `mappedBy`.

---

3. Запросы в Hibernate

3.1. HQL (Hibernate Query Language)

Аналог SQL, но работает с сущностями, а не таблицами.

```
String hql = "FROM User u WHERE u.age > :age";
Query<User> query = session.createQuery(hql, User.class);
query.setParameter("age", 18);
List<User> users = query.getResultList();
```

Примеры:

- JOIN:

```
FROM Order o JOIN o.user u WHERE u.name = :name
```

- Агрегатные функции:

```
SELECT COUNT(u) FROM User u
```

3.2. Criteria API

Типобезопасный построитель запросов.

```
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<User> cq = cb.createQuery(User.class);
Root<User> root = cq.from(User.class);

cq.select(root)
  .where(cb.gt(root.get("age"), 18));

List<User> users = session.createQuery(cq).getResultList();
```

Преимущества:

- Проверка на этапе компиляции.
- Динамическое построение запросов.

3.3. `@NamedQuery`

Предопределенные запросы (кешируются при старте).

```
@Entity
@NamedQuery(
    name = "User.findByAge",
    query = "SELECT u FROM User u WHERE u.age > :age"
)
public class User {
    // ...
}

// Использование
Query<User> query = session.createNamedQuery("User.findByAge", User.class);
query.setParameter("age", 18);
List<User> users = query.getResultList();
```

---

4. Кэширование

4.1. 1st Level Cache (Кэш сессии)

- **Автоматический**, живет в рамках одной сессии (`Session`).
- **Пример:**

```
User user1 = session.get(User.class, 1L); // Запрос к БД
User user2 = session.get(User.class, 1L); // Берется из кэша
```

4.2. 2nd Level Cache (Общий кэш)

- **Настраиваемый**, работает между разными сессиями.
- **Популярные реализации**: Ehcache, Infinispan.
- **Включение:**

```
<!-- pom.xml -->
<dependency>
```

```
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-ehcache</artifactId>
    </dependency>

    # application.properties
    hibernate.cache.use_second_level_cache=true
    hibernate.cache.region.factory_class=org.hibernate.cache.ehcache.EhCacheRegion
    Factory
```

- **Аннотации для кэширования:**

```
    @Entity
    @Cacheable
    @Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
    public class Product {
        // ...
    }
```

Стратегии кэширования:

- `READ_ONLY` - только для неизменяемых данных.
- `READ_WRITE` - с поддержкой изменений.
- `NONSTRICT_READ_WRITE` - без строгой синхронизации.

---

5. Дополнительные возможности

5.1. Слушатели (Listeners)

```
@Entity
@EntityListeners(AuditListener.class)
public class User {
    // ...
}

public class AuditListener {
    @PrePersist
    public void prePersist(Object entity) {
        // Логика перед сохранением
    }
}
```

5.2. Фильтры (`@Filter`)

```
@Entity
@FilterDef(name = "activeUserFilter", parameters = @ParamDef(name = "active", type =
Boolean.class))
@Filter(name = "activeUserFilter", condition = "is_active = :active")
public class User {
    // ...
}

// Использование
session.enableFilter("activeUserFilter").setParameter("active", true);
```

Итог

- **Сущности**: `@Entity`, `@Id`, `@Column`.
- **Связи**: `@OneToMany`, `@ManyToOne`, `@ManyToMany`.
- **Запросы**: HQL, Criteria API, `@NamedQuery`.
- **Кэширование**: 1st Level (сессия), 2nd Level (Ehcache).
- **Дополнительно**: Listeners, Filters.

Hibernate — мощный ORM-инструмент, который значительно упрощает работу с БД в Java-приложениях.