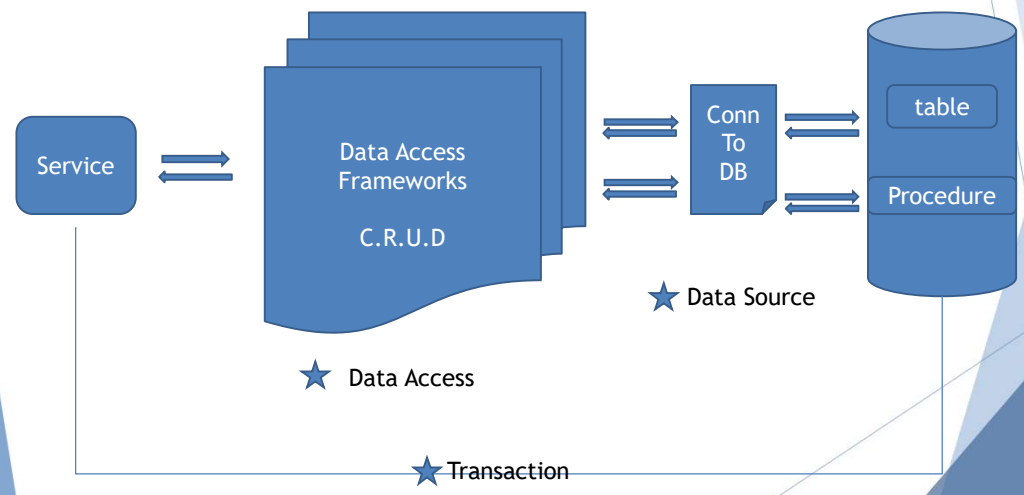
Data Access

Andy

1

Diagram of components



2

Data Access Frameworks

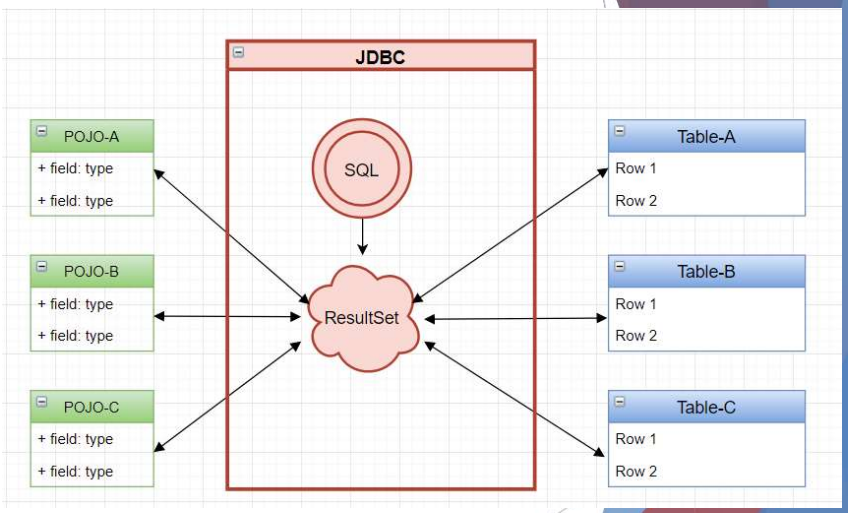
- Content
  - JDBC + Spring JDBC(JdbcTemplate)
  - Hibernate
  - JPA + Spring Data JPA

3

JDBC

4

JDBC



5

JDBC

```
//STEP 1. Import required packages
import java.sql.*;

public class FirstExample {
   // JDBC driver name and database URL
   static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
   static final String DB_URL = "jdbc:mysql://localhost/EMP";

   //  Database credentials
   static final String USER = "username";
   static final String PASS = "password";

   public static void main(String[] args) {
   Connection conn = null;
   Statement stmt = null;
   try{
      //STEP 2: Register JDBC driver
      Class.forName("com.mysql.jdbc.Driver");

      //STEP 3: Open a connection
      System.out.println("Connecting to database...");
      conn = DriverManager.getConnection(DB_URL,USER,PASS);

      //STEP 4: Execute a query
      System.out.println("Creating statement...");
      stmt = conn.createStatement();
      String sql;
      sql = "SELECT id, first, last, age FROM Employees";
      ResultSet rs = stmt.executeQuery(sql);

      //STEP 5: Extract data from result set
      while(rs.next()){
         //Retrieve by column name
         int id  = rs.getInt("id");
         int age = rs.getInt("age");
         String first = rs.getString("first");
         String last = rs.getString("last");
```

```
         //Display values
         System.out.print("ID: " + id);
         System.out.print(", Age: " + age);
         System.out.print(", First: " + first);
         System.out.println(", Last: " + last);
      }
      //STEP 6: Clean-up environment
      rs.close();
      stmt.close();
      conn.close();
   }catch(SQLException se){
      //Handle errors for JDBC
      se.printStackTrace();
   }catch(Exception e){
      //Handle errors for Class.forName
      e.printStackTrace();
   }finally{
      //finally block used to close resources
      try{
         if(stmt!=null)
            stmt.close();
      }catch(SQLException se2){
      }// nothing we can do
      try{
         if(conn!=null)
            conn.close();
      }catch(SQLException se){
         se.printStackTrace();
      }//end finally try
   }//end try
   System.out.println("Goodbye!");
}//end main
}//end FirstExample
```

6

Why Spring JDBC?

Action	Spring	You
Define connection parameters.		X
Open the connection.	X	
Specify the SQL statement.		X
Declare parameters and provide parameter values		X
Prepare and execute the statement.	X	
Set up the loop to iterate through the results (if any).	X	
Do the work for each iteration.		X
Process any exception.	X	
Handle transactions.	X	
Close the connection, the statement, and the resultset.	X	

Introduce:

JdbcTemplate
NamedParameterJdbcTemplate

7

Spring JdbcTemplate vs NamedParameterJdbcTemplate

NamedParameterJdbcTemplate会为sql里面的parameters
设定一个name，这样就不会把parameter弄混

```
@Repository
public class EmployeeRepository{

    @Autowired
    JdbcTemplate jdbcTemplate;

    @Autowired
    NamedParameterJdbcTemplate namedParameterJdbcTemplate;

    public int getNumberOfEmployees(int minAge,  int maxAge){
        String sql = "SELECT count(*) FROM person WHERE age >= ? and age <= ?";
        int result = jdbcTemplate.queryForObject(sql, new Object[]{minAge, maxAge), Integer.class);
        return result;
    }

    public int getNumberOfEmployeesNamedParameter(int minAge, int maxAge){
        String sql = "SELECT count(*) FROM person WHERE age >= :min_Age and age <= :maxAge";
        SqlParameterSource input = new MapSqlParameterSource();
        ((MapSqlParameterSource) input).addValue( paramName: "min_Age", minAge);
        ((MapSqlParameterSource) input).addValue( paramName: "max_Age", maxAge);
        int result = namedParameterJdbcTemplate.queryForObject(sql, input, Integer.class);
        return result;
    }
}
```

8

Data Source

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

9

Hibernate

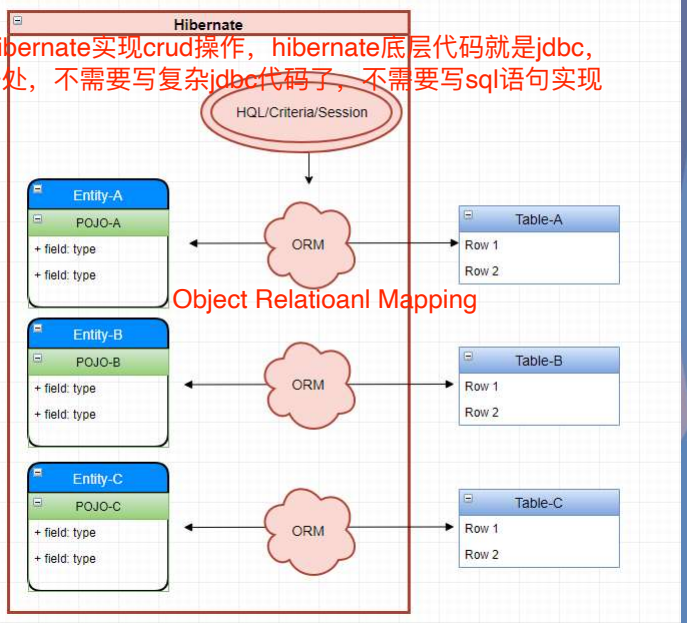
10

1 hibernate框架应用在javaee三层结构中 dao层框架

2 在dao层里面做对数据库crud操作，使用hibernate实现crud操作，hibernate底层代码就是jdbc，hibernate对jdbc进行封装，使用hibernate好处，不需要写复杂jdbc代码了，不需要写sql语句实现

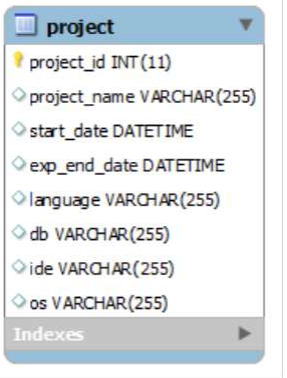
3 hibernate开源的轻量级的框架

Hibernate



11

Entity Mapping




```
@Entity
@Table(name="project")
public class Project{

    @Id
    @GeneratedValue
    private int project_id;

    @Column(name="project_name")
    private String projectName;

    @Column(name="start_date")
    private String startDate;

    @Column(name="exp_end_date")
    private String expirationDate;

    private String language;

    @Column(name="db")
    private String dataBase;

    private String ide;

    @Column(name="os")
    private String operatingSystem;

}
```

name和class名字一致时可以省略

@Id是指primary key

名字和table里面一样时可以不加annotation

12

Entity Mapping: Composite Primary Key

```java
@Embeddable
public class EmployeeId implements Serializable {

    @Column(name = "company_id")
    private Long companyId;

    @Column(name = "employee_number")
    private Long employeeNumber;

    public EmployeeId() {
    }

    public EmployeeId(Long companyId, Long employeeI
        this.companyId = companyId;
        this.employeeNumber = employeeId;
    }

    public Long getCompanyId() {
        return companyId;
    }

    public Long getEmployeeNumber() {
        return employeeNumber;
    }
```

```java
@Entity(name = "Employee")
@Table(name = "employee")
public class Employee {

    @EmbeddedId
    private EmployeeId id;

    private String name;

    public EmployeeId getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

13

Association: one-to-many and many-to-one

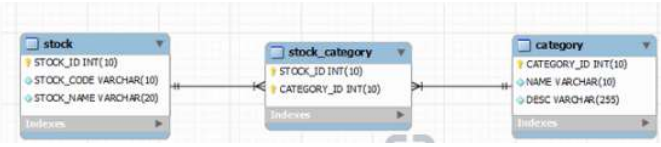
```java
@Entity(name = "Person")
public static class Person {

        @Id
        @GeneratedValue
        private Long id;

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

        //Getters and setters are omitted for brevity

}

@Entity(name = "Phone")
public static class Phone {

        @Id
        @GeneratedValue
        private Long id;

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

        //Getters and setters are omitted for brevity

}
```

```java
@Entity(name = "Person")
public static class Person {

        @Id
        @GeneratedValue
        private Long id;

        //Getters and setters are omitted for brevity

}

@Entity(name = "Phone")
public static 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;

        //Getters and setters are omitted for brevity

}
```

14

Association: Many-to-Many

In Database:



In Hibernate:

```
@Entity
@Table(name="stock")
public class Stock implements Serializable{
    private int stockId;
    private String stockCode;
    private String stockName;
    private Set<Category> categories = new HashSet<Category>();

    //getter & setter

    @ManyToMany(fetch=FetchType.LAZY, cascadeType.ALL)
    @JoinTable(name="stock_category",
                joinColumns={ @JoinColumn(name="STOCK_ID")},
                inverseJoinColumns = { @JoinColumn(name="CATEGORY_ID")} )
    public Set<Category> getCategories(){
        return this.categories;
    }
}
```

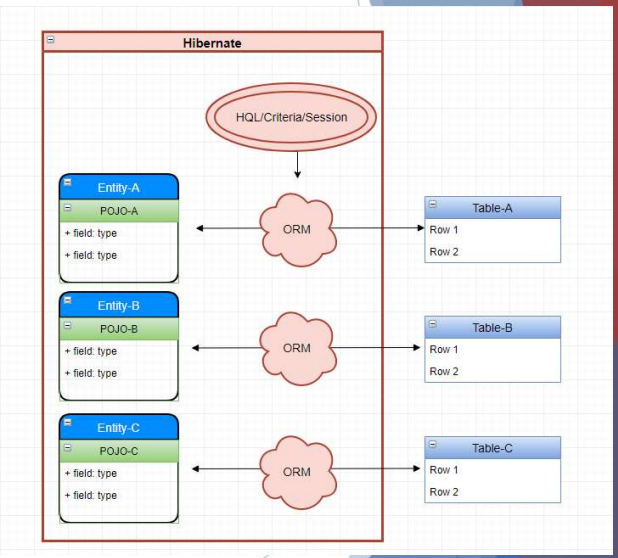
LAZY/EAGER ALL/

```
@Entity
@Table(name = "category")
public class Category implements Serializable {

    private Integer categoryId;
    private String name;
    private String desc;
    private Set<Stock> stocks = new HashSet<Stock>();


    @ManyToMany(fetch = FetchType.LAZY, mappedBy = "categories")
    public Set<Stock> getStocks() {
        return this.stocks;
    }
}
```

15

Hibernate Core Concepts

1. Entity Mapping
2. CRUD → HQL/Criteria
3. Session
4. Transaction
5. SessionFactory
6. Proxy
7. Persistence Context



16

*CRUD: Criteria API

```
Session session = HibernateUtil.getHibernateSession();
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Item> cr = cb.createQuery(Item.class);
Root<Item> root = cr.from(Item.class);
cr.select(root);

Query<Item> query = session.createQuery(cr);
List<Item> results = query.getResultList();
```

select

```
cr.select(root).where(cb.between(root.get("itemPrice"), 100, 200));
```

Chain expression

```
cr.orderBy(
  cb.asc(root.get("itemName")),
  cb.desc(root.get("itemPrice")));
```

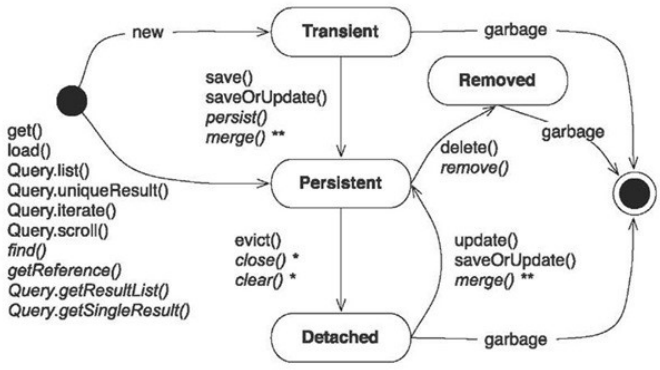
Sort

```
CriteriaQuery<Double> cr = cb.createQuery(Double.class);
Root<Item> root = cr.from(Item.class);
cr.select(cb.avg(root.get("itemPrice")));
Query<Double> query = session.createQuery(cr);
List avgItemPriceList = query.getResultList();
```

aggregate

17

Persistence Context



* Hibernate & JPA, affects all instances in the persistence context
** Merging returns a persistent instance, original doesn't change state

```
Person person = new Person();
person.setId( 1L );
person.setName("John Doe");

session.save( person );
```

```
//persist example - with transaction
Session session2 = sessionFactory.openSession();
Transaction tx2 = session2.beginTransaction();
Employee emp2 = HibernateSaveExample.getTestEmployee();
session2.persist(emp2);
System.out.println("Persist called");
emp2.setName("Kumar"); // will be updated in database too
System.out.println("Employee Name updated");
System.out.println("8. Employee persist called with transaction,
id="+emp2.getId()+", address id="+emp2.getAddress().getId());
tx2.commit();
System.out.println("*****");

// Close resources
sessionFactory.close();
```

Person p = new Person() ——transient
p.setName('a') ——db has nothing
session.save/persist/merge ——db has 'a' persistent
p.setName('b') ——db has 'b'
p.setName('c') ——db has 'c'

18

session.clear/close ——detached
p.setName('d') ——db has 'c'

session.merge()/update ——db has 'c' persistent
p.setName('e') ——db has 'e'

session.remove() ——db has 'e'

1) How do you handle transaction:
   @Transactional
2) update to entities in two steps, but second step failed, what will you do:
   @Tranctional to put them into one transaction if second step failed, rollback entire transaction
3) How do you handle exceptions in rest api/ spring MVC:
   @ExceptionHandler

Transaction

a boilerplate session code

```
SessionFactory sessionFactory = metadata.getSessionFactoryBuilder()
                .build();

Session session = sessionFactory.openSession();
try {
        // calls Connection#setAutoCommit( false ) to
        // signal start of transaction
        session.getTransaction().begin();

        session.createQuery( "UPDATE customer set NAME = 'Sir. '||NAME" )
                        .executeUpdate();

        // calls Connection#commit(), if an error
        // happens we attempt a rollback
        session.getTransaction().commit();
}
catch ( Exception e ) {
        // we may need to rollback depending on
        // where the exception happened
        if ( session.getTransaction().getStatus() == TransactionStatus.ACTIVE
                        || session.getTransaction().getStatus() ==
TransactionStatus.MARKED_ROLLBACK ) {
                session.getTransaction().rollback();
        }
        // handle the underlying error
}
finally {
        session.close();
        sessionFactory.close();
}
```

19

Transaction Management

```
@Transactional(propagation=Propagation.REQUIRED)
@Repository
@Getter
@Setter
public class BookPurchaseDaoImpl implements BookPurchaseDao {

        @Autowired
        private SessionFactory sessionFactory;
        private Session session;

        @Override
        @Transactional(propagation=Propagation.REQUIRED, rollbackFor=Exception.class)
        public void bookPurchase(int bookId, int userId, String userPass) throws Exception {

                if (!authenticate(userId, userPass)) {
                        throw new Exception("Unauthorized Access");
                }
                session = getSessionFactory().getCurrentSession();

                Book book = (Book) session.load(Book.class, bookId);
                BookStock bookStock = (BookStock) session.load(BookStock.class, bookId);
                Account account = (Account) session.load(Account.class, userId);
```

20

Database ACID

- **Atomicity** - a transaction to transfer funds from one account to another involves making a withdrawal operation from the first account and a deposit operation on the second. If the deposit operation failed, you don't want the withdrawal operation to happen either.
- **Consistency** - a database tracking a checking account may only allow unique check numbers to exist for each transaction
- **Isolation** - a teller looking up a balance must be isolated from a concurrent transaction involving a withdrawal from the same account. Only when the withdrawal transaction commits successfully and the teller looks at the balance again will the new balance be reported.
- **Durability** - A system crash or any other failure must not be allowed to lose the results of a transaction or the contents of the database. Durability is often achieved through separate transaction logs that can "re-create" all transactions from some picked point in time (like a backup).

Transaction

Primary key

Locking

Replication

- NoSQL CAP theorem
  - Consistency - data are equivalent to all requests regardless of which server the requests are sent to
  - Availability - System will always responds to a request even with old data
  - Partition Tolerance - even if one partition fails, it won't affect the system and requests

21

Locking

Concern:

A transaction reads data, update data and then commit the data. During commit, it needs to make sure the data is not stale – not modified by other concurrent on-going transaction.

What if it happens?

Rollback current transaction

Types of locking in hibernate

Optimisitic Locking and Pessimistic Locking

22

Optimistic Locking

```java
@Entity(name = "Person")
public static class Person {

        @Id
        @GeneratedValue
        private Long id;

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

        @Version
        private long version;

        //Getters and setters are omitted for brevity

}
```

timestamp

Approach 1: Version

```java
@Entity(name = "Person")
@OptimisticLocking(type = OptimisticLockType.ALL)
@DynamicUpdate
public static class Person {

        @Id
        private Long id;

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

        private String country;

        private String city;

        @Column(name = "created_on")
        private Timestamp createdOn;

        //Getters and setters are omitted for brevity
}
```

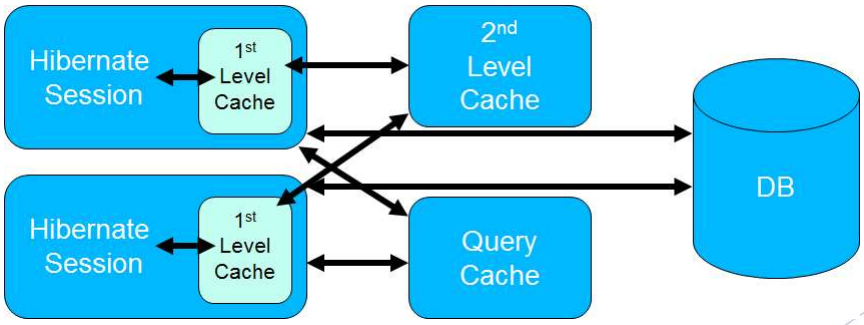
Version
All
Dirty

Approach 2: OptimisticLocking with DynamicUpdate

23

Caching

- Two level cache
  - 1st level cache: stays in session level. Purpose: minimize database visit
  - 2nd level cache: stays in sessionFactory. Purpose: cross session use



24

How do you turn on 2nd level Cache and what is the cache provider:
Set property use_second_level_cache to True.
EHCache is the provider

Configure 2nd level Cache

```
<properties>
    ...
    <property name="hibernate.cache.use_second_level_cache" value="true"/>
    <property name="hibernate.cache.region.factory_class"
      value="org.hibernate.cache.ehcache.EhCacheRegionFactory"/>
    ...
</properties>
```

```
@Entity
@Cacheable
@org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Foo {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "ID")
    private long id;

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

    // getters and setters
}
```

25

Performance Tuning

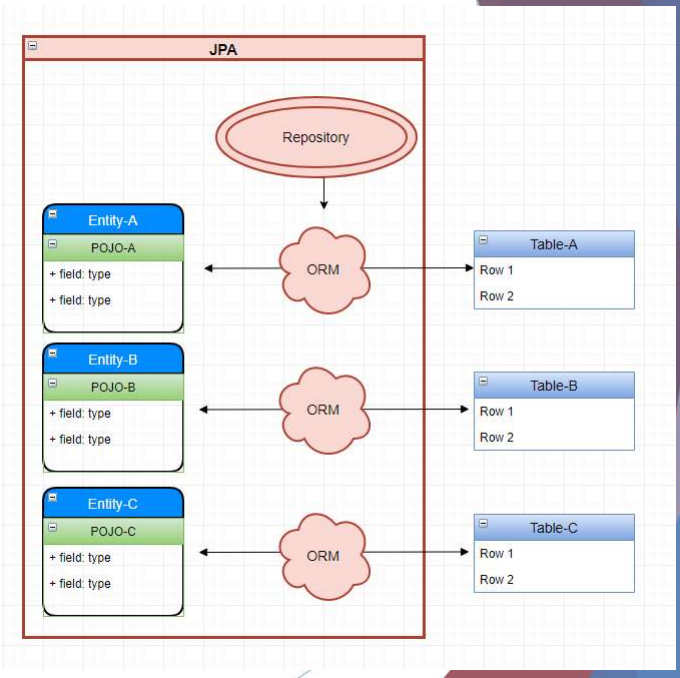
- Use logging to track visiting to database
- Optimize HQL or Native SQL Query
- FetchType = LAZY
- Use cache, but before using 2nd level cache, try to put heavy operation to database
- Bulk updates/deletes

26

JPA & Spring Data JPA

27

JPA



28

Why JPA?

JPA is to eliminate SQL in java

By using Naming Convention in Java Method

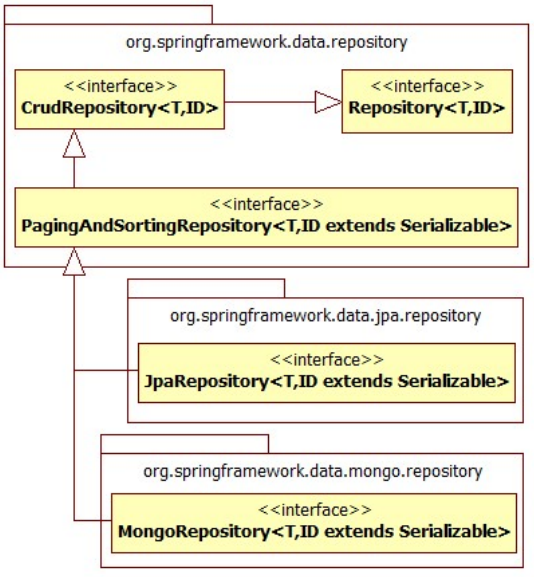
```
public interface CrudRepository<T, ID extends Serializable>
  extends Repository<T, ID> {

  <S extends T> S save(S entity);          1

  Optional<T> findById(ID primaryKey);     2

  Iterable<T> findAll();                   3

  long count();                            4

  void delete(T entity);                   5

  boolean existsById(ID primaryKey);       6

  // … more functionality omitted.
}
```

29

Repository

- CrudRepository
- PagingAndSortingRepository
- JpaRepository

★ *JpaRepository returns a list*
CrudRepository returns Iterable



30

4-step Declaration

```java
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@EnableJpaRepositories
class Config {}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jpa="http://www.springframework.org/schema/data/jpa"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/data/jpa
     http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

   <jpa:repositories base-package="com.acme.repositories"/>

</beans>
```

JpaRepository

```java
interface PersonRepository extends Repository<Person, Long> {
  List<Person> findByLastname(String lastname);
}
```

```java
class SomeClient {

  private final PersonRepository repository;

  SomeClient(PersonRepository repository) {
    this.repository = repository;
  }

  void doSomething() {
    List<Person> persons = repository.findByLastname("Matthews");
  }
}
```

1. Enable JPA Scan → 2. Extends Repository → 3. Define CRUD Query → 4. Inject into Service

31

JPA Naming Convention: Select

SQL Keyword	Example	SQL Script
And	findByLastnameAndFirstname(a,b)	Where lastname = ? and firstname = ?
Or	findByLastnameOrFirstname(a,b)	Where lastname = ? or firstname = ?
between	findByStartDateBetween(a,b)	Where startdate between ? and ?
IsNull	findByAgeIsNull()	Where age is null
IsNotNull	findByAgeIsNotNull()	Where age is not null
NotNull		
OrderBy	findByAgeOrderByLastNameDesc(a)	Where age = ? order by lastname desc
In	findByAgeIn(a[])	Where age in ?
NotIn	findByAgeNotIn(a[])	Where age not in ?
True	findByActiveTrue()	Where active = true
False	findByActiveFalse()	Where active = false
IgnoreCase	findByFirstnameIgnoreCase(a)	Where UPPER(firstname) = UPPER(?)

32

JPA Naming Convention: Insert/Update/Delete

Modifier and Type	Method and Description
long	count() Returns the number of entities available.
void	delete(T entity) Deletes a given entity.
void	deleteAll() Deletes all entities managed by the repository.
void	deleteAll(Iterable<? extends T> entities) Deletes the given entities.
void	deleteById(ID id) Deletes the entity with the given id.
boolean	existsById(ID id) Returns whether an entity with the given id exists.
Iterable<T>	findAll() Returns all instances of the type.
Iterable<T>	findAllById(Iterable<ID> ids) Returns all instances of the type with the given IDs.
Optional<T>	findById(ID id) Retrieves an entity by its id.
<S extends T> S	save(S entity) Saves a given entity.
<S extends T> Iterable<S>	saveAll(Iterable<S> entities) Saves all given entities.

33

Pagination

```
@Repository
public interface EmployeeRepository
                extends PagingAndSortingRepository<Employee, Long> {
    Page<Employee> findAll(Pageable pageable);

    Page<Employee> findByFirstName(String firstName, Pageable pageable);

    Slice<Employee> findByFirstNameAndLastName(String firstName, String lastName, Pageable pageable);
}


@Service
public class EmployeeService{

    @Autowired
    EmployeeRepository employeeRepository;

    public List<Employee> getEmployeeByPageAndSize(int page, int size){
        Pageable pageable = PageRequest.of(page, size);

        Page<Employee> employees = employeeRepository.findAll(pageable);
        return employees.getContent();
    }

}
```

34

Your Own Query

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

/**
 * Specifies methods used to obtain and modify person related information
 * which is stored in the database.
 * @author Petri Kainulainen
 */
public interface PersonRepository extends JpaRepository<Person, Long> {

    /**
     * Finds a person by using the last name as a search criteria.
     * @param lastName
     * @return  A list of persons whose last name is an exact match with the given last name.
     *          If no persons is found, this method returns an empty list.
     */
    @Query("SELECT p FROM Person p WHERE LOWER(p.lastName) = LOWER(:lastName)")
    public List<Person> find(@Param("lastName") String lastName);
}
```

35

Extra Notes: not required

36

Data Access - CRUD

- JdbcTemplate
- NamedParameterJdbcTemplate
- SimpleJdbcInsert & SimpleJdbcCall
- MappingSqlQuery, SqlUpdate,
- StoredProcedure

37

JdbcTemplate: select: query()

```
public List<Actor> findAllActors() {
    return this.jdbcTemplate.query( "select first_name, last_name from t_actor", new ActorMapper());
}

private static final class ActorMapper implements RowMapper<Actor> {

    public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
        Actor actor = new Actor();
        actor.setFirstName(rs.getString("first_name"));
        actor.setLastName(rs.getString("last_name"));
        return actor;
    }
}
```

38

Spring JDBC

JdbcTemplate: select: queryForObject()

```
int rowCount = this.jdbcTemplate.queryForObject("select count(*) from t_actor", Integer.class);
```

```
Actor actor = this.jdbcTemplate.queryForObject(
        "select first_name, last_name from t_actor where id = ?",
        new Object[]{1212L},
        new RowMapper<Actor>() {
            public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
                Actor actor = new Actor();
                actor.setFirstName(rs.getString("first_name"));
                actor.setLastName(rs.getString("last_name"));
                return actor;
            }
        });
```

39

Spring JDBC

JdbcTemplate:select:queryForList()

```
public List<Customer> findAll(){

    String sql = "SELECT * FROM CUSTOMER";

    List<Customer> customers = new ArrayList<Customer>();

    List<Map> rows = getJdbcTemplate().queryForList(sql);
    for (Map row : rows) {
        Customer customer = new Customer();
        customer.setCustId((Long)(row.get("CUST_ID")));
        customer.setName((String)row.get("NAME"));
        customer.setAge((Integer)row.get("AGE"));
        customers.add(customer);
    }

    return customers;
}
```

40

Spring JDBC

JdbcTemplate: Update/Insert/Delete

- Insert/Update/Delete ("update")

```
this.jdbcTemplate.update(
        "insert into t_actor (first_name, last_name) values (?, ?)",
        "Leonor", "Watling");
```

```
this.jdbcTemplate.update(
        "update t_actor set last_name = ? where id = ?",
        "Banjo", 5276L);
```

```
this.jdbcTemplate.update(
        "delete from actor where id = ?",
        Long.valueOf(actorId));
```

41

Spring JDBC

NamedParameterJdbcTemplate

```
@Service
public class EmployeeService{

    @Autowired
    NamedParameterJdbcTemplate namedParameterJdbcTemplate;

    public Employee getEmployee(String fname, String lname, Date sDate, Date eDate){

        String sql = "select count(*) from Employee
                      where first_name = :firstName and last_name = :lastName
                      and start_date > :startDate
                      and end_date > :endDate "

        Employee e = new Employee();
        e.setFirstName(firstName);
        e.setLastName(lastName);
        e.setStartDate(startDate);
        e.setEndDate(endDate);

        SqlParameterSource input = new BeanPropertySqlParameterSource(e);

        Employee result = this.namedParameterJdbcTemplate.queryForObject(sql, input, Integer.class)

        return result;
    }
}
```

42

Spring JDBC

* SimpleJdbcInsert

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.insertActor = new SimpleJdbcInsert(dataSource).withTableName("t_actor");
    }

    public void add(Actor actor) {
        Map<String, Object> parameters = new HashMap<String, Object>(3);
        parameters.put("id", actor.getId());
        parameters.put("first_name", actor.getFirstName());
        parameters.put("last_name", actor.getLastName());
        insertActor.execute(parameters);
    }

    // ... additional methods
}
```

43

Spring JDBC

* Stored Procedure: SimpleJdbcCall

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcCall procReadActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.procReadActor = new SimpleJdbcCall(dataSource)
                .withProcedureName("read_actor");
    }

    public Actor readActor(Long id) {
        SqlParameterSource in = new MapSqlParameterSource()
                .addValue("in_id", id);
        Map out = procReadActor.execute(in);
        Actor actor = new Actor();
        actor.setId(id);
        actor.setFirstName((String) out.get("out_first_name"));
        actor.setLastName((String) out.get("out_last_name"));
        actor.setBirthDate((Date) out.get("out_birth_date"));
        return actor;
    }

    // ... additional methods
}
```

44

Spring JDBC

* Stored Procedure: StoredProcedure

```
public void moveToHistoryTable(Person person) {
    StoredProcedure procedure = new GenericStoredProcedure();
    procedure.setDataSource(dataSource);
    procedure.setSql("MOVE_TO_HISTORY");
    procedure.setFunction(false);

    SqlParameter[] parameters = {
            new SqlParameter(Types.BIGINT),
            new SqlOutParameter("status_out", Types.BOOLEAN)
    };

    procedure.setParameters(parameters);
    procedure.compile();

    Map<String, Object> result = procedure.execute(person.getId());
}
```

45

Hibernate

Entity Mapping: Inheritance

```
@MappedSuperclass
public static class Account {

        @Id
        private Long id;

        private String owner;

        private BigDecimal balance;

        private BigDecimal interestRate;

        //Getters and setters are omitted for brevity

}

@Entity(name = "DebitAccount")
public static class DebitAccount extends Account {

        private BigDecimal overdraftFee;

        //Getters and setters are omitted for brevity

}

@Entity(name = "CreditAccount")
public static class CreditAccount extends Account {

        private BigDecimal creditLimit;

        //Getters and setters are omitted for brevity

}
```

4 types

Single table

```
CREATE TABLE Account (
    DTYPE VARCHAR(31) NOT NULL ,
    id BIGINT NOT NULL ,
    balance NUMERIC(19, 2) ,
    interestRate NUMERIC(19, 2) ,
    owner VARCHAR(255) ,
    overdraftFee NUMERIC(19, 2) ,
    creditLimit NUMERIC(19, 2) ,
    PRIMARY KEY ( id )
)
```

Table per class

```
CREATE TABLE Account (
    id BIGINT NOT NULL ,
    balance NUMERIC(19, 2) ,
    interestRate NUMERIC(19, 2) ,
    owner VARCHAR(255) ,
    PRIMARY KEY ( id )
)

CREATE TABLE CreditAccount (
    id BIGINT NOT NULL ,
    balance NUMERIC(19, 2) ,
    interestRate NUMERIC(19, 2) ,
    owner VARCHAR(255) ,
    creditLimit NUMERIC(19, 2) ,
    PRIMARY KEY ( id )
)

CREATE TABLE DebitAccount (
    id BIGINT NOT NULL ,
    balance NUMERIC(19, 2) ,
    interestRate NUMERIC(19, 2) ,
    owner VARCHAR(255) ,
    overdraftFee NUMERIC(19, 2) ,
    PRIMARY KEY ( id )
)
```

Mapped SuperClass

```
CREATE TABLE DebitAccount (
    id BIGINT NOT NULL ,
    balance NUMERIC(19, 2) ,
    interestRate NUMERIC(19, 2) ,
    owner VARCHAR(255) ,
    overdraftFee NUMERIC(19, 2) ,
    PRIMARY KEY ( id )
)

CREATE TABLE CreditAccount (
    id BIGINT NOT NULL ,
    balance NUMERIC(19, 2) ,
    interestRate NUMERIC(19, 2) ,
    owner VARCHAR(255) ,
    creditLimit NUMERIC(19, 2) ,
    PRIMARY KEY ( id )
)
```

Joined table

```
CREATE TABLE Account (
    id BIGINT NOT NULL ,
    balance NUMERIC(19, 2) ,
    interestRate NUMERIC(19, 2) ,
    owner VARCHAR(255) ,
    PRIMARY KEY ( id )
)

CREATE TABLE CreditAccount (
    creditLimit NUMERIC(19, 2) ,
    id BIGINT NOT NULL ,
    PRIMARY KEY ( id )
)

CREATE TABLE DebitAccount (
    overdraftFee NUMERIC(19, 2) ,
    id BIGINT NOT NULL ,
    PRIMARY KEY ( id )
)

ALTER TABLE CreditAccount
ADD CONSTRAINT FKihw8h3j1k0w31cnyu7jcl7n7n
FOREIGN KEY (id) REFERENCES Account

ALTER TABLE DebitAccount
ADD CONSTRAINT FKia914478noepymc468kiaivqm
FOREIGN KEY (id) REFERENCES Account
```

46

Hibernate

*Flush Mode

static FlushMode	**ALWAYS** The Session is flushed before every query.
static FlushMode	**AUTO** The Session is sometimes flushed before query execution in order to ensure that queries never return stale state.
static FlushMode	**COMMIT** The Session is flushed when Transaction.commit() is called.
static FlushMode	**MANUAL** The Session is only ever flushed when Session.flush() is explicitly called by the application.

47

Hibernate

*Example

```
Person person = new Person("John Doe");
entityManager.persist(person);

Session session = entityManager.unwrap( Session.class);
session.setHibernateFlushMode( FlushMode.MANUAL );

assertTrue(((Number) entityManager
    .createQuery("select count(id) from Person")
    .getSingleResult()).intValue() == 0);

assertTrue(((Number) session
    .createNativeQuery("select count(*) from Person")
    .uniqueResult()).intValue() == 0);
```

How to set Flushing Mode

```
entityManager = entityManagerFactory().createEntityManager();
txn = entityManager.getTransaction();
txn.begin();

Person person = new Person( "John Doe" );
entityManager.persist( person );
log.info( "Entity is in persisted state" );

txn.commit();
```

```
--INFO: Entity is in persisted state
INSERT INTO Person (name, id) VALUES ('John Doe', 1)
```

```
Person person = new Person( "John Doe" );
entityManager.persist( person );
entityManager.createQuery( "select p from Advertisement p" ).getResultList();
entityManager.createQuery( "select p from Person p" ).getResultList();
```

Apply: Auto/Commit/Always/Manual

48

JPA

JPA named Query

```
@Entity
@NamedQueries({
    @NamedQuery(
            name="Country.findAll",
            query = "SELECT c  from country c"
    ),
    @NamedQuery(
            name = "Country.findByName",
            query = "SELECT c FROM country c WHERE c.name = :name"
    )
})
public class Country{
    //fields
}

@Repository
public class CountryRepository{

    @Autowired
    EntityManager em;

    public List<Country> getCountries() {
        TypedQuery<Country> query = em.createNamedQuery( s: "Country.findAll", Country.class);
        List<Country> results = query.getResultList();
        return results;
    }
}
```

49

Transaction

Propagation

REQUIRED: Must run in a transaction, create new if no transaction exist

REQUIRED_NEW: Always create a new transaction

SUPPORTS: Run in current transaction or no transaction is needed

NOT_SUPPORTED: do not run in a transaction

MANDATORY: Must run in a transaction or an exception will be thrown

Isolation

Default: Follow underlying database

READ_UNCOMMITTED: Can read uncommitted data

READ_COMMITTED: Only read committed data

50