Шаблон DAO в Java

<https://www.codeflow.site/ru/article/java-dao-pattern>

**Как я понял DAO это прокладка между кодом и базой данных, для возможности работы с последней (CRUD). В связи с появлением JPA, DAO используется только для ограничения функционала по обращению из блока бизнес-логики к реляционной базе данных (разрешая например, только читать информацию, но не записывать). Т.к. все методы по работе с БД уже написаны в JPA и опять их писать в DAO нет смысла.**

Объясните пожалуйста на пальцах что такое DAO, что от него требуется?

Например есть в тебя таблица в БД Users.  
На нее ты пишеш клас entity с названием User. Тебе надо будет поля вставлять, менять, удалять из таблицы Users, в общем исполнять какие-то SQL-запросы.  
И для этих всех SQL-запросов надо создать класс UserDAO и в него поместить все методы, которые исполняют эти SQL-запросы.

А вообще скачай Netbeens, там можна сгенерить всю структуру: entity, DAO, controller, html-страница на основе конкретной таблицы из БД.

**1. Обзор**

Шаблон Data Access Object (DAO) является структурным шаблоном, который позволяет нам **изолировать прикладной/бизнес-уровень от постоянного уровня (обычно это реляционная база данных, но это может быть любой другой постоянный механизм) с использованием абстрактного API**.

Функциональность этого API заключается в том, чтобы скрыть от приложения все сложности, связанные с выполнением операций CRUD в базовом механизме хранения. Это позволяет обоим слоям развиваться отдельно, ничего не зная друг о друге.

В этом уроке мы углубимся в реализацию шаблона и узнаем, как использовать его для абстрагирования вызовов к [Менеджер сущностей JPA](https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html).

**2. Простая реализация**

Чтобы понять, как работает шаблон DAO, давайте создадим базовый пример.

Допустим, мы хотим разработать приложение, которое управляет пользователями. Для того чтобы модель домена приложения не зависела от базы данных, мы создадим **простой класс DAO, который позаботится о том, чтобы эти компоненты были аккуратно отделены друг от друга**.

\*\* 2.1. Домен Класс

\*\*

Поскольку наше приложение будет работать с пользователями, нам нужно определить только один класс для реализации его доменной модели:

**public class User {**

**private String name;**

**private String email;**

**//constructors/standard setters/getters**

**}**

Класс *User*представляет собой простой контейнер для пользовательских данных, поэтому он не реализует никакого другого поведения, заслуживающего внимания.

Конечно, самый важный выбор дизайна, который нам нужно сделать здесь, - это как сохранить приложение, использующее этот класс, изолированным от любого механизма персистентности, который может быть реализован в какой-то момент.

Ну, это именно та проблема, которую пытается решить шаблон DAO.

**2.2. DAO API**

Давайте определим базовый уровень DAO, чтобы мы могли видеть, как он может **полностью отделить модель домена от уровня постоянства.**

Вот API DAO:

public interface Dao<T> {

Optional<T> get(long id);

List<T> getAll();

void save(T t);

void update(T t, String[]params);

void delete(T t);

}

С высоты птичьего полета ясно, что интерфейс *Dao*определяет абстрактный API, который выполняет операции CRUD над объектами типа *T*.

Благодаря высокому уровню абстракции, который обеспечивает интерфейс, легко создать конкретную, детальную реализацию, которая работает с объектами *User*.

**2.3. *UserDao*Класс**

Давайте определим пользовательскую реализацию интерфейса *Dao*:

public class UserDao implements Dao<User> {

private List<User> users = new ArrayList<>();

public UserDao() {

users.add(new User("John", "[email protected]"));

users.add(new User("Susan", "[email protected]"));

}

@Override

public Optional<User> get(long id) {

return Optional.ofNullable(users.get((int) id));

}

@Override

public List<User> getAll() {

return users;

}

@Override

public void save(User user) {

users.add(user);

}

@Override

public void update(User user, String[]params) {

user.setName(Objects.requireNonNull(

params[0], "Name cannot be null"));

user.setEmail(Objects.requireNonNull(

params[1], "Email cannot be null"));

users.add(user);

}

@Override

public void delete(User user) {

users.remove(user);

}

}

Класс *UserDao*реализует все функции, необходимые для извлечения, обновления и удаления объектов *User*.

* Для простоты *users List*действует как база данных в памяти, которая заполнена парой объектов *User*в конструкторе \*\*

Конечно, легко реорганизовать другие методы, чтобы они могли работать, например, с реляционной базой данных.

Хотя оба класса *User*и *UserDao*сосуществуют независимо в одном и том же приложении, нам все еще необходимо выяснить, как последний может использоваться для сохранения уровня персистентности скрытым от логики приложения:

public class UserApplication {

private static Dao userDao;

public static void main(String[]args) {

userDao = new UserDao();

User user1 = getUser(0);

System.out.println(user1);

userDao.update(user1, new String[]{"Jake", "[email protected]"});

User user2 = getUser(1);

userDao.delete(user2);

userDao.save(new User("Julie", "[email protected]"));

userDao.getAll().forEach(user -> System.out.println(user.getName()));

}

private static User getUser(long id) {

Optional<User> user = userDao.get(id);

return user.orElseGet(

() -> new User("non-existing user", "no-email"));

}

}

Пример придуман, но в двух словах показывает мотивы, лежащие в основе шаблона DAO. В этом случае метод *main*просто использует экземпляр *UserDao*для выполнения операций CRUD над несколькими объектами *User*.

* Наиболее значимым аспектом этого процесса является то, как *UserDao*скрывает от приложения все низкоуровневые сведения о том, как объекты сохраняются, обновляются и удаляются \*\* .

**3. Использование шаблона с JPA**

Среди разработчиков существует общая тенденция думать, что выпуск JPA снизил до нуля функциональность шаблона DAO, поскольку шаблон становится просто еще одним уровнем абстракции и сложности, реализованным поверх уровня, предоставляемого менеджером сущностей JPA.

Безусловно, в некоторых сценариях это действительно так. Несмотря на это, иногда мы просто хотим предоставить нашему приложению только несколько доменных методов API-интерфейса менеджера сущностей. В таких случаях шаблон DAO имеет свое место.

**3.1. *JpaUserDao*Класс**

С учетом вышесказанного давайте создадим новую реализацию интерфейса *Dao*, чтобы мы могли увидеть, как он может инкапсулировать функциональность, предоставляемую менеджером сущностей JPA из коробки:

public class JpaUserDao implements Dao<User> {

private EntityManager entityManager;

//standard constructors

@Override

public Optional<User> get(long id) {

return Optional.ofNullable(entityManager.find(User.class, id));

}

@Override

public List<User> getAll() {

Query query = entityManager.createQuery("SELECT e FROM User e");

return query.getResultList();

}

@Override

public void save(User user) {

executeInsideTransaction(entityManager -> entityManager.persist(user));

}

@Override

public void update(User user, String[]params) {

user.setName(Objects.requireNonNull(params[0], "Name cannot be null"));

user.setEmail(Objects.requireNonNull(params[1], "Email cannot be null"));

executeInsideTransaction(entityManager -> entityManager.merge(user));

}

@Override

public void delete(User user) {

executeInsideTransaction(entityManager -> entityManager.remove(user));

}

private void executeInsideTransaction(Consumer<EntityManager> action) {

EntityTransaction tx = entityManager.getTransaction();

try {

tx.begin();

action.accept(entityManager);

tx.commit();

}

catch (RuntimeException e) {

tx.rollback();

throw e;

}

}

}

* Класс *JpaUserDao*способен работать с любой реляционной базой данных, поддерживаемой реализацией JPA. \*\*

Кроме того, если мы внимательно посмотрим на класс, мы поймем, как использовать <https://en.wikipedia.org/wiki/Composition>*over*inheritance[Composition]и [Dependency Инъекция](https://en.wikipedia.org/wiki/Dependency__injection)позволяет нам вызывать только методы менеджера сущностей, требуемые нашим приложением.

Проще говоря, у нас есть специализированный API для конкретного домена, а не весь API менеджера сущностей.

**3.2. Рефакторинг класса *User***

В этом случае мы будем использовать Hibernate в качестве реализации по умолчанию для JPA, поэтому мы соответствующим образом проведем рефакторинг класса *User*:

@Entity

@Table(name = "users")

public class User {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private long id;

private String name;

private String email;

//standard constructors/setters/getters

}

**3.3. Начальная загрузка JPA Entity Manager программно**

Предполагая, что у нас уже есть работающий экземпляр MySQL, работающий локально или удаленно, и таблица базы данных *«пользователи»*, заполненная некоторыми записями пользователей, нам нужно получить диспетчер сущностей JPA, чтобы мы могли использовать класс *JpaUserDao*для выполнения операций CRUD в базе данных.

В большинстве случаев мы выполняем это с помощью типичного файла «persistence.xml», который является стандартным подходом

В этом случае мы воспользуемся подходом *xml-less*и получим менеджер сущностей с простой Java через удобный [*https://docs.jboss.org/hibernate/orm/5.0/javadocs/org/hibernate/*](https://docs.jboss.org/hibernate/orm/5.0/javadocs/org/hibernate/)*. jpa/boot/internal/EntityManagerFactoryBuilderImpl.html[EntityManagerFactoryBuilderImpl]*класс.

Для получения подробного объяснения о том, как запустить реализацию JPA с помощью Java, перейдите по ссылке:/java-bootstrap-jpa[эта статья].

**3.4. *UserApplication*Class**

Наконец, давайте проведем рефакторинг исходного класса *UserApplication*, чтобы он мог работать с экземпляром *JpaUserDao*и выполнять операции CRUD с сущностями *User*:

public class UserApplication {

private static Dao<User> jpaUserDao;

//standard constructors

public static void main(String[]args) {

User user1 = getUser(1);

System.out.println(user1);

updateUser(user1, new String[]{"Jake", "[email protected]"});

saveUser(new User("Monica", "[email protected]"));

deleteUser(getUser(2));

getAllUsers().forEach(user -> System.out.println(user.getName()));

}

public static User getUser(long id) {

Optional<User> user = jpaUserDao.get(id);

return user.orElseGet(

() -> new User("non-existing user", "no-email"));

}

public static List<User> getAllUsers() {

return jpaUserDao.getAll();

}

public static void updateUser(User user, String[]params) {

jpaUserDao.update(user, params);

}

public static void saveUser(User user) {

jpaUserDao.save(user);

}

public static void deleteUser(User user) {

jpaUserDao.delete(user);

}

}

Даже когда пример довольно ограничен, он остается полезным для демонстрации того, как интегрировать функциональность шаблона DAO с той, которую предоставляет менеджер сущностей.

В большинстве приложений есть структура DI, которая отвечает за внедрение экземпляра *JpaUserDao*в класс *UserApplication*. Для простоты мы опустили детали этого процесса.

Наиболее важный момент, на который следует обратить внимание, заключается в том, как класс *JpaUserDao*помогает полностью исключить независимость класса *UserApplication*от того, как уровень персистентности выполняет операции CRUD \*\* .

Кроме того, мы могли бы поменять MySQL на любую другую СУБД (и даже на плоскую базу данных) в будущем, и все же наше приложение продолжало бы работать как положено, благодаря уровню абстракции, обеспечиваемому интерфейсом *Dao*и менеджером сущностей ,

**4. Заключение**

В этой статье мы подробно рассмотрели ключевые концепции шаблона DAO, как реализовать его в Java и как использовать его поверх менеджера сущностей JPA.