SOFTWARE ARCHITECTURES AND METODOLOGIES (2017)
Ingegneria Informatica
Università degli Studi di Firenze

Software architecture study for a notes software using JPA

Javier Izquierdo Vera
Miguel Medina Ballesteros

September 22, 2017

Contents

List of Figures

1 Introduction

This project is a simulation of a Java EE application using the JPA architecture, but abstracting from the top level user interface layer and focusing in the Domain Model and DAOs. We will perform an application of notes using Java Persistence API, focusing on the domain logic and mapping the classes defined in the design step with the database through JPA annotations.

The application will be similar to Google Keep, although obviously simpler, and without making the presentation layer. We will design the domain logic, maintain persistence through the JPA Entity Manager. To interact with the data will be used DAOs, and we will create controllers as the business logic layer. The following sections describe the features.

2 Requirement Analysis

In the following section we will describe the main features of the software, bearing in mind that we will not implement any web interface since we are focusing on the implementation of JPA architecture. We want to end up with a simple client that allow the user to execute the most basic operations. So, in general, the user's control will be basic and limited to simple operations as create, edit, view and share notes.

2.1 Informal description

The first thing to consider is the sign up. A user need an username and password to login into the application. Additional fields will be requested, like email, etc.

At first, the user must login with his username and password. Then, the application will show him his notes (if any) and the notes shared by other users to him (if any), ordered by date of creation. The user will be able to create a new note, and edit or share the existing ones. The user will also be able to share his notes with others users, by selecting the desired note to share and then writing the desired user's username. The sharing can be with write permissions or only read permissions.

A note can be a text note, audio note, or image note; it is only an example of inheritance implementation in JPA, but we only will implement the content of text note. Audio and Image note will remain as simulations. A text note will be a title, a body text and a optional list of tags. The title and the body text are mandatory fields for all kind of notes.

Finally, the user can logout.

The application is designed to be easy to maintain, extend and modify, in a way that improvements would be easy of implement, as finding notes by text occurrences or more ordering filters.

2.2 Functional Requirements

- Create account.
- Exit application.
- Login.
- Logout.
- Create text note.
- Edit note.
- Add tags to a note.
- Share notes with others users using different permissions levels.
- Show notes.
- View specifics notes.
- Show shared notes.

2.3 Non Functional Requirements

- Changes made in notes must be real time updated, having in mind the possibility of edit permissions. This will be so because many users could use the application at the same time, and we don't want an user to logout to update his changes.
- The title of the note, user and email will be stored in the MySQL database as varchar and will have a limit of 255 characters.
- The application will store the creation and last edit date of the notes to order the notes.
- The note key field in the database won't be the title, will be a hidden identifier.
- The shared notes mustn't be stored twice, obviously.
- Identify Tags by it's text.
- Identify Users by it's username.

2.4 Architectural Requirements

- Users, notes and tags stored in MySQL database.
- Use of Java Persistence API to mapping users, notes and tags java objects to MySQL database.
- Use of Eclipselink solution for JPA.
- Non complex user interface, i.e. a simple command line menu.

2.5 Domain conceptual schema

The next image (Figure 2.1) represents possible interactions in the domain [9].

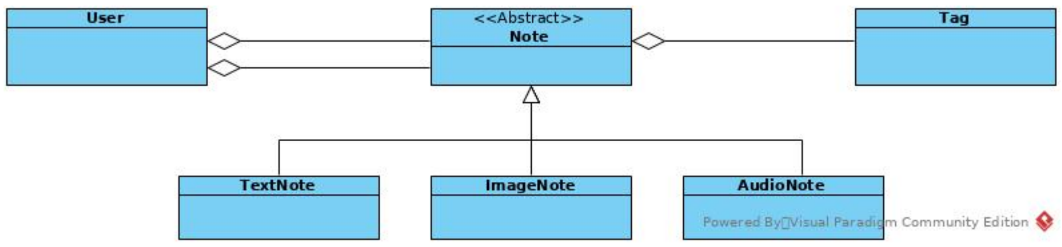

Figure 2.1: Domain conceptual schema

2.6 Use cases

2.6.1 Actors

- Guest: It is who can register and log in to the system.
- User: It is who has notes and he can creates, edits, deletes and shares them.

2.6.2 Use Case Diagram

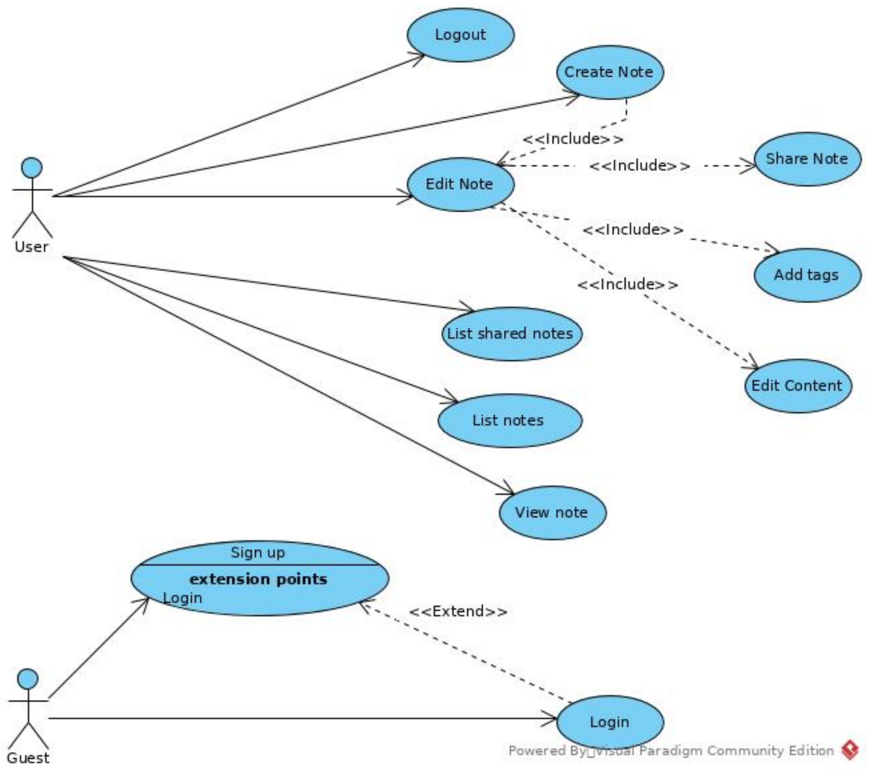

Figure 2.2: Use Case Diagram

3 Implementation

3.1 Domain Logic

In this section we will define the domain logic of the software. It is how data is stored, created and modified, that is, the entities that make up the domain and its relationship with the database.

The following figure shows the interaction scheme of our problem:

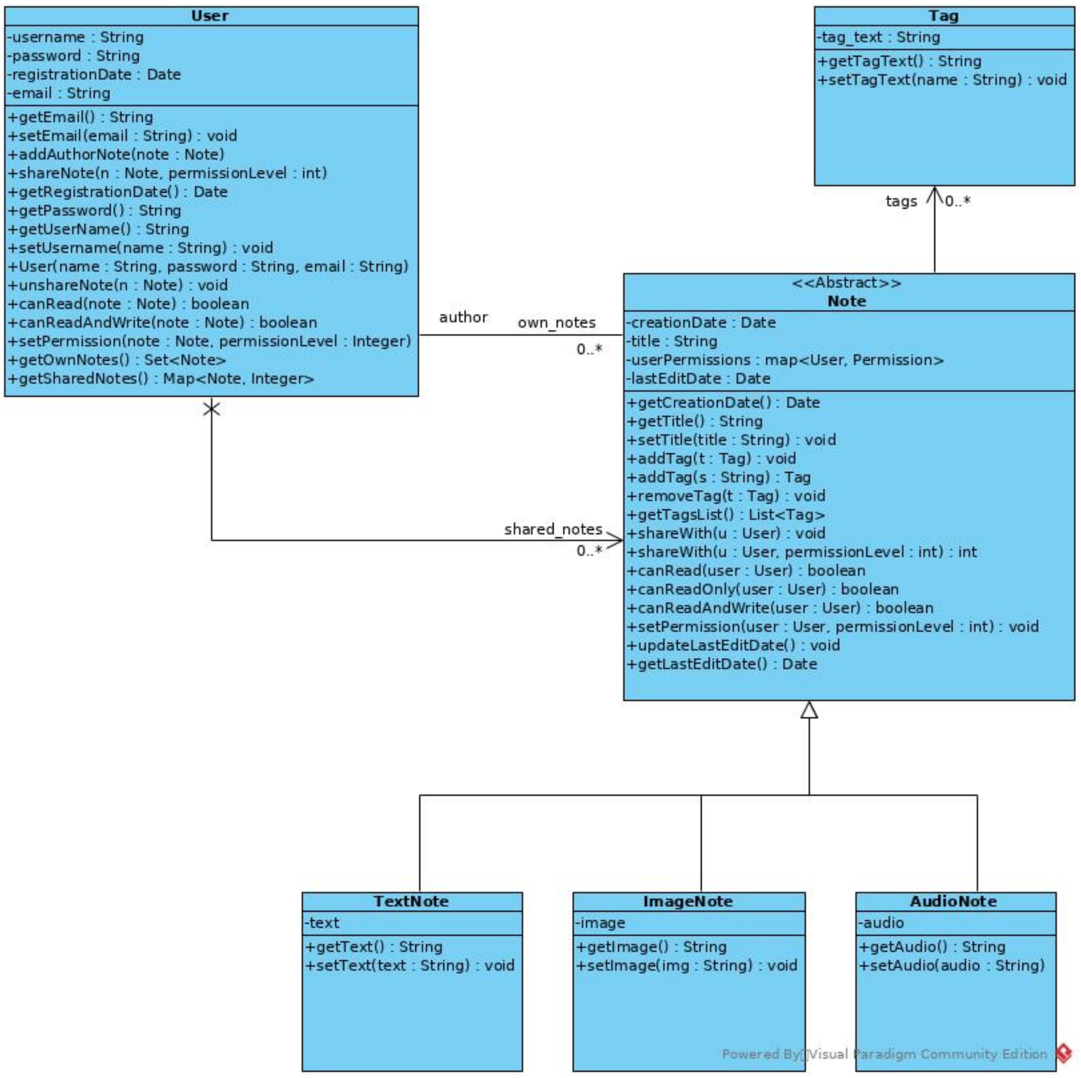
User
-username : String
-password : String
-registrationDate : Date
-email : String
+getEmail() : String
+setEmail(email : String) : void
+addAuthorNote(note : Note)
+shareNote(n : Note, permissionLevel : int)
+getRegistrationDate() : Date
+getPassword() : String
+getUserName() : String
+setUsername(name : String) : void
+User(name : String, password : String, email : String)
+unshareNote(n : Note) : void
+canRead(note : Note) : boolean
+canReadAndWrite(note : Note) : boolean
+setPermission(note : Note, permissionLevel : Integer)
+getOwnNotes() : Set<Note>
+getSharedNotes() : Map<Note, Integer>
Tag
-tag_text : String
+getTagText() : String
+setTagText(name : String) : void
tags
0..*
author
own_notes
0..*
shared_notes
0..*
<<Abstract>>
Note
-creationDate : Date
-title : String
-userPermissions : map<User, Permission>
-lastEditDate : Date
+getCreationDate() : Date
+getTitle() : String
+setTitle(title : String) : void
+addTag(t : Tag) : void
+addTag(s : String) : Tag
+removeTag(t : Tag) : void
+getTagsList() : List<Tag>
+shareWith(u : User) : void
+shareWith(u : User, permissionLevel : int) : int
+canRead(user : User) : boolean
+canReadOnly(user : User) : boolean
+canReadAndWrite(user : User) : boolean
+setPermission(user : User, permissionLevel : int) : void
+updateLastEditDate() : void
+getLastEditDate() : Date
TextNote
-text
+getText() : String
+setText(text : String) : void
ImageNote
-image
+getImage() : String
+setImage(img : String) : void
AudioNote
-audio
+getAudio() : String
+setAudio(audio : String)
Powered By Visual Paradigm Community Edition

Figure 3.1: Domain Logic Class diagram

- User: It represents a system user. A user has a list of notes. It can also have shared notes (notes shared with this user by other users). And it has attributes like name, password,..

- Tag: A note can be classified with one or more tags. A tag is recognized by its name.
- Note: This entity represents a note created by a user. It has an author (user who creates it) and a list of tags. A note can be:
  - TextNote: A written note.
  - ImageNote: An image saved as a note
  - AudioNote: An audio note.

3.1.1 Persistence

With the aim of mapping the Java Objects from the Domain Logic to a relational database, we started considering two frameworks as JPA implementation: Hibernate and Eclipselink. Finally, for lack of better knowledge of both solutions, we decided to go on with Eclipselink as it's the one that's used in DZone Refcard "Getting Started with JPA" by Mike Keith [7].

We have three persistence classes, in addition to the class of inheritance and several collection of objects. All classes have been annotated as @Entity. The inherintance structure of Note use a strategy of InheritanceType.JOINED, it means JPA will create one table per note node, besides one table by Note. A Note has a set of tags, and this is mapped as a @ManyToMany relationship and will result in a MySQL table for joining Notes table and Tags table. The same occurs with User, that has a Map of Notes and Permission Level annotated as @ElementCollection [8], that will result in another join table but with an extra field representing the Permission Level of the User over the Note. It's important to note that we have been able to map the Map in this way because we are using JPA 2.1. In previous versions, JPA 1.0, this would have been resolved by having a class as SharingEvent and having a set of sharing events [6, 3].

All of this classes extend from BaseEntity, which defines an id attribute for identifying the entities persisted and also overrides equals() and hashCode() methods, which are needed to support Sets representating many-valued associations [2].

3.2 Data Access Object

DAOs are the interface between Domain Logic and Business Logic, that is to say, DAOs separate database access and the rest of the software. Every DAO extends from BaseDao, which has the basics operations needed for persist and remove an entity. A DAO needs an instance of a working Entity Manager for the correct management of the persistence context, and it will be passed as argument in the constructor.

As we have said, BaseDao has methods for persist objects and drop them, delegating this work to the Entity Manager. There is also a method to find objects by their id, although the id will be unknown at application level. It is extended by NoteDAO, TagDAO and UserDAO; extending BaseDao with a concrete object type. In addition, TagDAO and UserDAO have a method for find a tag or user by the name string by performing a custom query.

3.3 Business Logic

It uses DAOs to implement the operations that our use case need and abstracts the interface of JPA complexity. The presentation layer would use directly and only this layer to interact with the system. We have four controllers class:

- LoginController: It is able to log in and register new users.
- NoteController: It allows to create, share, and edit notes.
- TagController: Its function is to create and locate tags.
- UserController: It is responsible for performing operations involving users, such as retrieving an user or assigning a shared note to a user.

4 Testing

In every extensible and complex system, it's important to perform regular tests to check that the software is growing properly. But most importantly, a solid testing is essential for the easy maintenance of a complex and extensible system. With this in mind, we have include a hole suite of Unit Tests that assert that the core features of the JPA implementation are working.

4.1 Unit tests

We have decided JUnit as Unit Testing framework for Java [4].

Let's remember that the aim of Unit Testing is to exercise a single behavior of a software module. That module will be our class, and the behavior will be the public methods of the class. This means that our Unit Test's aim is not to test the Persistence API, nor the Entity Manager, nor the database connections. We are focusing on testing our single methods and classes.

To perform this, we could divide the testing explanation in the following sections:

- Model. For the model testing, we have decide to test that the annotations are those that we have decided. If the rest of the JPA Architecture works properly, our Entities, annotated of this way will work properly too. To perform this, we have used the Java reflection mechanism as explains in reference [10].
- BaseEntity Abstract Class. This is an special case. We have also tested here if hashCode() and equals() methods work properly. Inside the test we have definded a class ConcreteEntity that let us test an instance of BaseEntity Abstract Class.
- DAOs. For the DAOs we need to test if DAOs methods are calling to EntityManager methods the way we expect. To check it, we have coded an EntityManagerMock that gives us the information about what methods have been called, so we can make the assertions.
- UserDAO and TagDAO. We have also tested that custom queries calls the Entity-Manager as we expected.
- Mocking. We have mocked EntityManager by implementing its interface for passing the mock to the DAOs as argument in their constructor. The mocked class doesn't persist anything actually, it only sets to true a series of flags that tell if a certain method have been called or not. We have also needed to implement a QueryMock to avoid NullPointerExceptions in the custom query test of the User and Tag DAOs. We could have used any mocking framework as, for example, Mockito [5], but we have considered that EntityManagerMock was enough.

5 Conclusion

Starting from the basis of "An User can take a Note and label it with some Tags", this project simulates the developping steps of a software. In the section architecture requirements, we had "Use JPA to mapping java objects to MySQL database" as restriction, so from then on, we continued specifying how our Java objects will be mapped to a traditional MySQL database by using JPA. We have focused on separating the responsabilities into three main layers: Model, DAOs and Business Logic. The DAOs are the only ones accessing directly to the database through the EntityManager.

Having done all this system, we have made a very simple user interface to test the system at the very top level, depending only of the Business Logic layer. In this way we have implemented an interface via terminal. For instance, others could program a web interface above our system by using other Java EE technologies as CDI + JSF running over an Apache TomEE server.[1]

In this project we mainly see how JavaEE + JPA can provide a clean and extensible solution to Data Persistance, what places JPA as the first option to develop a system that meets Java with databases.

References

[1] Apache tomee jsf+cdi+ejb examples. `http://tomee.apache.org/examples-trunk/jsf-cdi-and-ejb/`.

[2] Elementcollection feature. `https://docs.jboss.org/hibernate/stable/core.old/reference/en/html/persistent-classes-equalshashcode.html`.

[3] Jpa tutorial. `https://www.tutorialspoint.com/es/jpa/`.

[4] Junit framework web. `http://junit.org/junit5/`.

[5] Mockito framework web. `http://site.mockito.org/`.

[6] Elementcollection feature. `http://mdshannan1.blogspot.com.es/2010/09/jpa-2-new-feature-elementcollection.html`, 2010.

[7] Mike Keith. Getting started with jpa. `https://dzone.com/refcardz/getting-started-with-jpa`, 2008.

[8] logicbig. Map with entity keys. `http://www.logicbig.com/tutorials/java-ee-tutorial/jpa/map-with-entity-keys/`.

[9] Microsoft. Uml use case diagrams. `https://msdn.microsoft.com/en-us/library/dd409432.aspx`.

[10] Michael Remijan. Unit testing jpa...stop integration testing! `http://mjremijan.blogspot.com.es/2016/06/unit-testing-jpa-annotationsstop_20.html`, June 20, 2016.