Praktisk utveckling av ett informationssystem

Implementation av ett bibliotekssystem

Författare: Oliver Boudet, olibou-0@student.ltu.se

Inledning

Denna projektuppgift gick ut på att utveckla ett informationssystem baserat på en kravspecifikation. Denna specifikation krävde av systemet:

- En GUI
- Databaskoppling
- Implementera arv & polymorphism
- Hantera undantag

Alla dessa krav uppfylls av det färdigutvecklade systemet, och denna rapport beskriver utvecklingsprocessen och dess tillvägagångssätt i detalj.

Resultat

Planering

Versionshantering

Git är ett relativt standardiserat versionshanteringsverktyg, därför valdes det till detta projekt.

Kanban

Kanban tavlan hittas här.

För att hålla koll på alla krav och dess implementeringar, användes en kanban tavla. I tavlan finns fem kolumner:

- To do: *alla* funktioner som krävs för att uppfylla krav. Specificiteten av kanban korten varierade, men det viktigaste var att inte göra dem för generella.
- Next-up: en extra icke standard kolumn som endast användes för att veta vad som behövde göras härnäst.
- In progress: alla kort som någon arbetar på. I detta fall är det inte superviktigt eftersom projektet utförs av en person, men det är bra att ha ändå.
- Done: alla kort som anses färdiga
- Done but needs iteration: alla kort som en gång ansågs färdig, men någon har hittat problem. I tillbakablick skulle denna kolumn tas bort, och istället skulle ett nytt kort skapas för alla problem som hittas.

Kod arkitektur

För att underlätta utvecklingen, används Spring Boot som grund till hela systemet. Huvudanledningen att Spring Boot användes är att det enkelt tillåter oss att koppla våran databas till applikationen, men det innehåller många fler funktioner som underlättar utvecklingen.

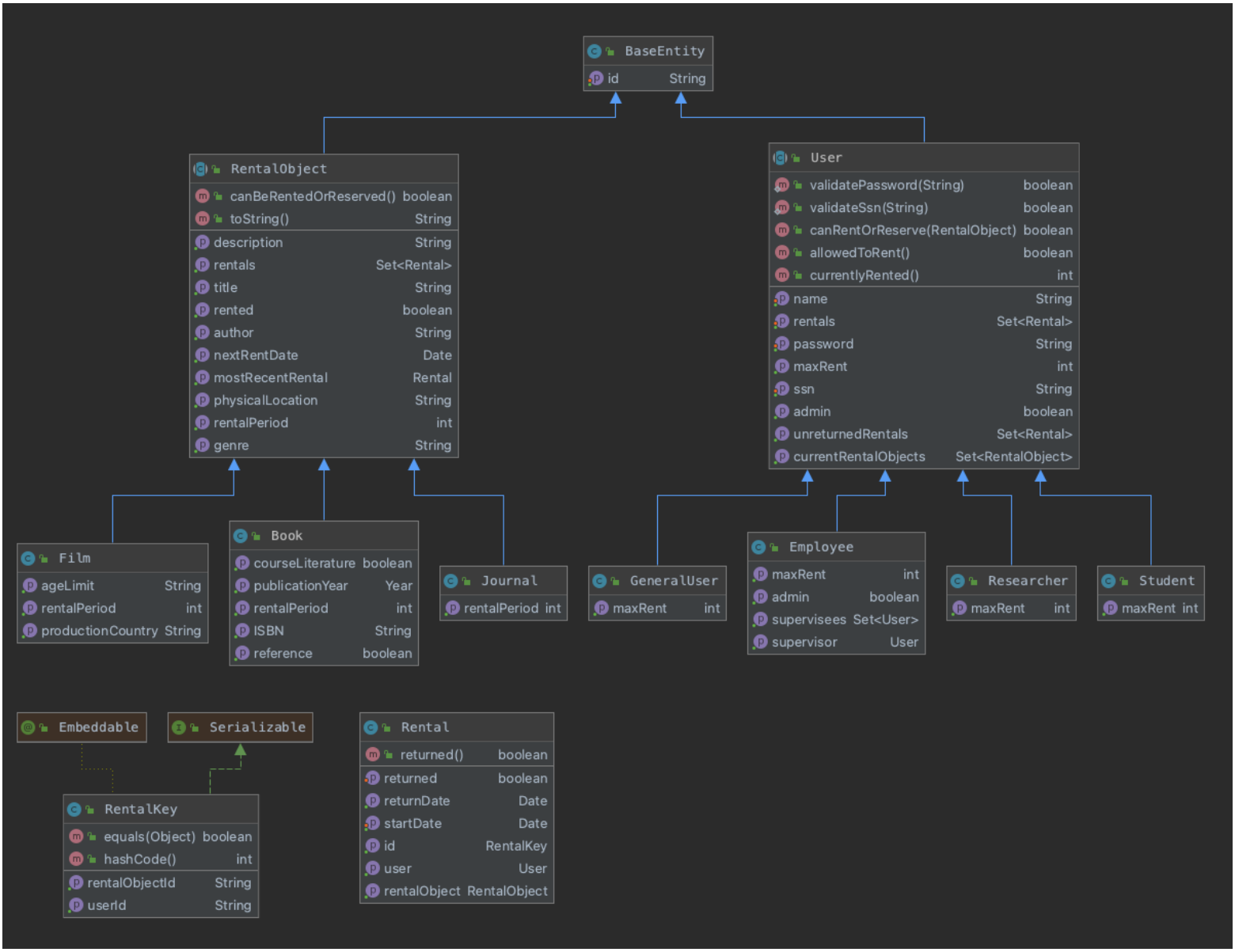
För GUI:n används Swing. Detta valdes på grund av dess simplicitet, men även stabilitet och dess förmåga att uppfylla programmets krav.

I högsta nivå är mestadels av koden i ett paket, **Application.** Det enda undantaget är instansieringsklassen, där själva applikationen initieras. Därefter har vi flera paket:

- Components: innehåller klasser med kod som ska exekveras innan resten av applikationen startar. Används i dagsläget endast för att populera databasen med test data.
- Entities: innehåller alla databasentiteter.
- Exceptions: innehåller alla Exceptions.
- GUI: innehåller kontroll, olika vyer, och dialoger för GUI.

- Repositories: innehåller definitioner för repositories, där alla transaktioner definieras. Spring Boot skapar automatiskt SQL queries baserat på metodernas namn.
- Services: innehåller hjälpmetoder som utför olika funktioner, som t.ex. skapar ett nytt lån åt en användare. Dessa använder sig av **Autowired** attribut, som Spring Boot injicerar automatiskt när en klass har en **Service** annotation.

Utöver detta, har vi configurationsfiler som t.ex. **application.properties** och **pom.xml**.

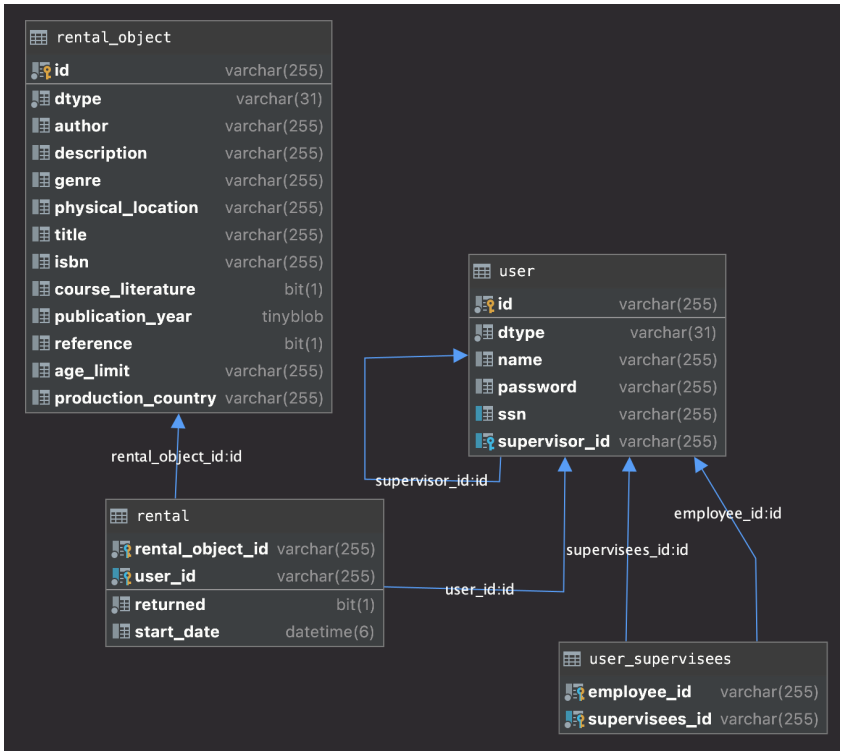


Figur 1: autogenererat klassdiagram (IntelliJ IDEA)

Databas

Koppling mellan databas och applikation

För att underlätta utvecklingen av systemet används som sagt Spring Boot, vilket har väldigt bra integrering med JPA. Därför användes JPA som databaskopplingsbibliotek, och koden skrevs med "code-first" principen. Detta betyder även att databasmodellen inte behövde utvecklas manuellt, och att den istället genererades av ORM:en. Det finns nackdelar till att använda en ORM, som till exempel långsammare sök queries, men fördelarna bedömdes väga mer än nackdelarna. Fördelar inkluderar sparad tid på databasutveckling, enkel användning, och tight integrering med resten av systemet.



Figur 2: autogenererat fysisk databasmodell (DataGrip).

En utökning från kravspecifikationen är **user_supervisees** tabellen, som ska användas för att hålla koll på anställda och dess chefer. Däremot implementerades aldrig denna funktion i systemet, och passar istället bättre in i ett separat system.

Transaktioner

En vanlig användare (student, researcher, general user) vill:

- Söka efter böcker baserat på: titel, författare, isbn, ämnesord.
- Låna objekt
- Återlämna objekt
- Logga in
- Reservera objekt

En anställd (employee) vill:

- Lägga till objekt
- Ta bort objekt
- Uppdatera objekt
- Visa lista på objekt som ej återlämnats i tid

Referensintegritet

- När en **user** tas bort ska alla dess **rental** tas bort (gdpr?)
- När en **rentalObject** tas bort tas alla dess **rental** bort.
- När en **rental** tas bort är dess **rentalObject** kvar.
- När en **rental** tas bort är **user** kvar
- När en **user** tas bort är dess **supervisor** kvar
- När en **supervisor** tas bort är dess **user** kvar

Fetch types

Fetch types definierar hur databashanteraren ska hämta attribut som innehåller listor av andra databastabeller. De fetch types som anges nedan är inte del av koden, men det är dem som ska användas. Anledningen att dessa inte används är att lazy fetching inte fungerade som det skulle.

Rental.rentalObject: **eager**. Kopplingstabell, finns ingen poäng att inte hämta direkt.
Rental.user: **eager**. Kopplingstabell, finns ingen poäng att inte hämta direkt.
RentalObject.rentals: **lazy**. En användare kan vilja se t.ex endast titel och genre utan att behöva rentals.
Employee.supervisees: **lazy**. Vill inte alltid ha alla supervisees.
Employee.supervisor: **lazy**. Vill inte alltid ha supervisor.
User.rentals: **lazy**. Vill inte alltid ha alla rentals

Dessa fetch types är de som skulle vara ideella, men tyvärr uppstod problem när en försökte använda lazy fetching.

Funktioner

Återlämning

Återlämning ska enligt uppgiftsbeskrivning ske genom avläsning av en unik kod på boken. För att inte behöva implementera någon sorts streckkodsavläsning, använder vi istället en ID inmatning. Där matar användaren in bokens unika id. Detta betyder också att användaren inte ens behöver vara inloggad, eftersom vi kan hitta dess motsvarande **Rental** objekt. Det är inte helt ideellt för användbarhet, med tanke på att id:t är ett långt uuid.

Ett sätt att lösa detta:

List box med alla böcker som en användare har lånat, objekt titel och id. Där kan användaren välja vilken bok den vill lämna tillbaks utan att skriva in hela UUID. Detta implementerades till slut, men det gamla sättet finns fortfarande kvar, då det kan vara användbart ifall vi har tillgång till en streckkodsläsare.

Diskussion

Avgränsningar

Inga större anmärkningsvärda avgränsningar har gjorts, däremot kan förbättringar av koden göras. Det största som skulle behöva refactoring är **LibraryApplicationGUI** klassen. Istället för att trycka in nästan alla programmets funktioner där, skulle ett Command pattern kunna användas, där varje funktion har en egen klass med en *run* funktion osv. Det finns även repeterande kod på vissa ställen, som skulle behöva skrivas om till en metod, även om båda metoderna gör lite olika saker.

En sak som möjligtvis kan ses som en avgränsning är att databasen i sig inte har några business rules implementerade, däremot finns alla dessa begränsningar i Java koden direkt. I och med att vi använder en ORM är det inte meningen att vi ska interagera direkt med databashanteraren, om inte för debugging anledningar. Detta kan göra det svårare att återanvända samma databas till andra system, men inte helt omöjligt. En ny databasmodell skulle behöva implementeras, och all data föras över.

I och med att detta endast är en prototyp, samlas inga mailadresser in. Detta betyder också att inga mail kommer skickas ut till användare som inte lämnat tillbaks objekt i tid. Detta kan enkelt implementeras i framtiden med ett cronjob.

Utökningar

Vissa utökningar framtogs under utvecklingsprocessens gång. En av dessa utökningar var en modul som ska ladda in OpenLibrary CSV data, och göra den kompatibel med databasen och resten av systemet. Detta implementerades dock aldrig.

Utvärdering

Det största problemet jag stötte på var den initiala uppsättningen av Spring Boot, JPA, och Swing. Även om jag hade tidigare erfarenhet av att använda ORMs, hade jag endast använt mig av ett Active Record pattern. Det är inte omöjligt att använda sig av ett sådant i JPA, men det gör det mer komplicerat för lite sparad tid. Istället användes ett repository pattern. Jag hade tidigare använt mig av Windows Forms för att skapa GUI appar, och gav mig in på Swing med liknande förväntningar. Efter att ha gjort färdig biblioteksystemet kan jag komma till slutsatsen att de inte är särskilt olika, även om de har vissa skillnader. Den största mest anmärkningsvärda är hur allt fungerar som en grid som standard.

För att dra en slutsats, har jag lärt mig mycket när det kommer till design av databas för praktisk användning, enklare skapande av ett GUI, och implementation av *services*, *controllers,* och *components* i Spring Boot.