Rent It

Second-Year Project,
Bachelor in Software Development,
IT University of Copenhagen

Group 12
Jakob Melnyk, jmel@itu.dk
Frederik Lysgaard, frly@itu.dk
Ulrik Flænø Damm, ulfd@itu.dk
Niklas Hansen, nikl@itu.dk
Jacob Claudius Grooss, jcgr@itu.dk

April 26th, 2012

Contents

Chapter 1

Preface

This document contains excerpts of our not yet finished report. All the parts within are works in progress and some are incomplete or missing entirely. Some figures and/or images may be missing as well.

Chapter 2

Requirements

2.1 Required features

List movies Be able to get a list of all movies.

Search Be able to search for a movie title.

Browse by genre Be able to get all genres and get all movie in a genre.

View movie details Get details about a movie.

Download movie Be able to download a movie.

Create user, login/logout Be able to create a user, and be able to log in and out of the system.

Upload movie Be able to upload a movie file.

Edit movie information Be able to edit information about a movie.

2.2 Optional features

Release dates Be able to specify a release date for a movie, so that users won't be able to download a movie before.

Improved search Making the search check for spelling errors, find incomplete matches and search in movie metadata.

Movie tags Be able to attach tags to a movie for improved search results.

Movie ratings Making users able to give a rating to movies they have watched.

User banning Making admins able to ban users for various reasons.

Discounts Giving discounts on multiple movie purchases.

Movie series Being able to mark movies as in a series.

Trailers Be able to watch trailers to movies.

In-app movie player Be able to watch rented movies from within the app.

Chapter 3

Design

In this chapter we discuss our decisions on architecture, design patterns, user interface, error handling and more.

3.1 Graphical User Interface

3.1.1 Usability testing

When the time came to start working on the client part of the project, we were hesitant with starting to code right away since all members of the group knows software, where a bad graphical user interface (GUI) had caused it to flunk. Because of this we decided to take a more planned approach on how to design our GUI - this involved making usability tests. The aim of performing usability tests is to observe people using our product to discover errors and areas that can be improved.

Making a usability test

To be able to make a usability test in the first place, we need an interface for the user to test, so we sat down as a team and discused what functionality our client should have. Then we made a couple of paper mockups for users to test in the first usability test. Before we started the usability test, we prepared some usability goals, which, if met, would convince us that the current GUI design should be our final GUI design.

List of usability goals:

- The user should be able to finish all given tasks within a time periode of 45 seconds.
- The user should be able to maneuver the client without need to ask the tester qustions.
- The user should be positive of the design.
- The user should be able to recommend the service to his or her friends.

Usability scenarios

For the usability test itself we created a couple of scenarios. These scenarios were designed so that the user would be forced to navigate through all the clients functionalities.

List of usability scenarios:

- Your have heard of this new movie rental service and you would like to sign up for it.
- You would like to rent "Batman the Begining" from the service.
- You would like to see what movies are most popular at the moment.
- You have gotten a new email and would like to change your profile so it uses your new email.
- As a movie company employee, you would like to upload some movies to the service.
- You have uploaded a movie with the wrong title - change it
- As an admin for the service you've seen some companies upload explicit material to service, delete those companies from the service.
- You would like to see a list of all the users who are using the service.

Results of the first iteration of tests

After the first round of tests we found that the user had many difficulties with navigating through the program, specifically when asked to do the scenarios which involved saving and editing of information.
The problem was that the user wasn't used to being able to edit in the fields without clicking on a editbutton of some kind first and then edit the desired fields, and then get the option to save those changes. The user simply lacked some confirmation on whether their changes was saved or not.

This came as a surprise since we had made mockups of a client which we did not feel had a lot of redundent data and functionalities. This also includes a lack of pop-up windows or extra save/edit buttons, but appafrently the need for confirmation is so ingrained in todays average user that it can't be removed without the user missing it.

We set out to solve the problems spotted by our testers throughout the first iteration of tests. These were solved by adding more save buttons and some pop-up notification boxes asking for the usual "Are you sure you want to edit the information?". When this was done, we began making the alpha version of the client so that we could make a usability test with a digital mock-up.

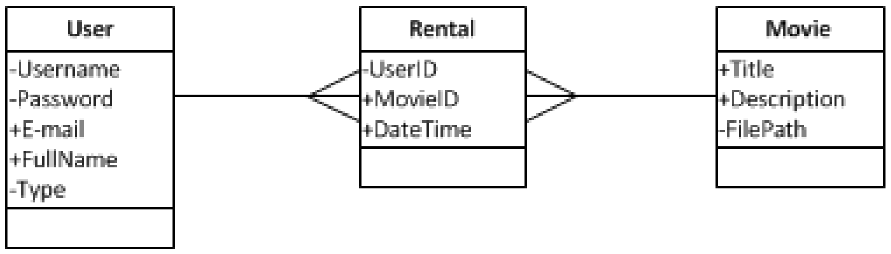
3.2 Database

Since we are using Scrum, our database is constantly evolving. The database explained here is not the final database, and it is not the initial database either. It is a snapshot of how it looks at the time of writing.

We decided quite early on to focus on simplicity and feature completeness, instead of adding a lot of half-done features and/or untested functionality. That also meant a very simple data model, as we only added what was needed to do the job.

Figure 3.1 on page 7 displays our simple initial data model. This captures the basic information about users, such as username, password, email address and the user's full name. We also have a field called type, representing whether a given user is a normal user, a content provider or a system administrator.

Figure 3.1: Initial datamodel



To begin with we only wanted to capture the basic information about the movies - like title, description and the file path. To finish it off, we added a table for capturing movie rentals. This basically was a junction table with references to the user renting the movie, the movie being rented and the time of the rental.

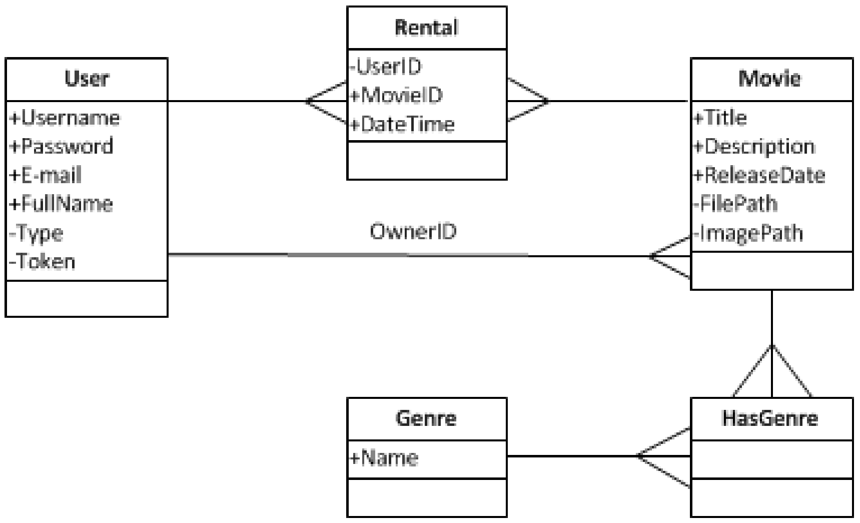


Figure 3.2: Updated datamodel

Figure 3.2 on page 7 displays our current data model. Since we are using scrum and work in sprints, our data model is constantly evolving. We only add things to the code and the database when it is needed. At the time of writing, our data model has been updated to the one showed in figure 3.2. The most notable difference is the genres now being in its own table. Another difference is the addition of a token to the user, which is explained later on. A movie now also has an owner and a release date.

3.2.1 Tables

User

user_id The user's unique ID number.
username The user's unique username - used for login.

password The user's password - hashed and salted.
email The user's email address.
full_name Full name of the user.
type User, Content provider or System Administrator.
token Unique session token generated at login and cleared at logout.

The User table contains data about our users. To provide some security to the passwords, we salt and hash (with the SHA512 algorithm) the passwords before putting it into the database. This means that if the database is hacked, then, unless they know the salt, they will still have a hard time figuring out the password of the users. The value in the password field will be useless to them, so they only find out the usernames and they will need to try and login to our service with all possible passwords in a bruteforce attack, to try and figuring the passwords of our users. The problem comes up if they get access to our codebase, as they will also get access to the salt. This will make them able to bruteforce the passwords locally (and not using the login feature on our service), which will be much faster.

The "type" field is an integer (since SQL Server 2008 doesn't have enum support) between 1 and 3.

type 1 A value of 1 means that a given user is "just" a normal user.
type 2 A value of 2 means that the user is a content provider.
type 3 A value of 3 indicates a system administrator.

The "token" field is a session token. We didn't look much into the different WCF bindings, but we found a binding with streaming support, but with no session support. So we created our own sessions, by generating a session key upon login. This session token is then stored in the "token" field, and is cleared upon logout. This session token has to be provided at every service call (that requires the user to be logged in).

Movie

movie_id The movie's unique ID number.
title Title of the movie.
description A more or less detailed description of the movie.
file_path File path to the video file.
owner_id Reference to the user creating the movie.
release_date The release date when the movie is available for rental.

The Movie table is quite straight forward. A movie has a title and a description to help identify the movie. The file path is a relative path, based from the movie root folder. Currently it is only a file name, but it is called "file_path" if this was to be changed later on. The "owner_id" is a reference to the User table, to a content provider that created that movie.

The release date is quite important. We took a decision to display all movies (also the ones not yet released) to the user, but it is only possible to rent a movie that has been released, and a movie has only been released if the release date has been set and is a time and date before the current time and date.

Genre

genre_id The genre's unique ID number.
name The name of the genre.

We changed this to be its own table, when we discovered that a movie very easily can be of several genres, and instead of putting all these together in one string with a delimiter in between, we decided to move it to its own table. It also makes it easier to re-use genres, which will make it easier when searching for a specific genre, as it is quite easy to misspell a genre. So when the most genres already have been added, and the system suggests genres for a given movie upon creation, genres are re-used, which makes it easier to browse genres.

HasGenre

hasgenre_id Unique ID.
movie_id Reference to a movie in the Movie table.
genre_id Reference to a genre in the Genre table.

Junction table used to associate genres with movies. The movie and genre combination is set as unique, so you cannot associate a given genre with a given movie twice. It wouldn't really matter much if we didn't make the combinations unique, but it would clutter the database and potentially slow it down.

Rental

rental_id The unique ID number of the rental.
user_id Reference to a user in the User table.
movie_id Reference to a movie in the Movie table.
time Time of rental.

This is probably the most important database in a rental system: the tracking of rentals. This has been made quite simple. We capture the user renting the movie, the movie being rented and the time of rental. We currently have a hardcoded rental period, but we plan to make this more dynamic in the future.

3.2.2 Foreign keys

To ensure integrity of fields referencing rows in other tables, we decided to add foreign keys. A foreign key is a check being run when deleting or updating rows in the database. We added those to everywhere we use IDs referencing other tables. These are:

Movie.user_id User reference for the owner of the movie.
Rental.user_id User reference to the user renting a given movie.
Rental.movie_id Movie reference to the movie being rented by the user.
HasGenre.movie_id Movie reference to a movie having a given genre.
HasGenre.genre_id Genre reference to the specific genre the movie has.

When defining foreign keys, it is possible to specify what the database should do if anything gets deleted or updated. We decided early on that we would never delete anything from the database. If we were to delete anything, we would add a "flag" to indicate whether or not a row is active or deleted. We do this for historic reasons, as we don't want to lose information about rentals or other actions, just because a user or a movie is deleted. We also decided to never change the unique ID number of a row (like User.user_id). There is no need to update these anyway, as the user is never presented with this ID and is only a way to quickly identify users.

We decided to specify these delete and update actions anyway. Most of the foreign keys is set to "Cascade", meaning that if I delete a user, all of his rentals will be deleted. If we were to delete a movie, all rentals for that movie would be deleted. It may seem that this conflicts with our previously mentioned decision (that we would never delete anything), but we still feel this is the right choice. If we were to delete anything, it is a special case and in those cases in wouldn't make much sense to keep these empty references. If we were to just set those references to null or don't do anything, we needed to do checks for this in our code, to make sure our code doesn't crash because of incorrect references.

Due to SQL Server complaining about multiple cascade paths, we decided to not use cascade on Movie.owner_id. We decided that if we delete a content provider, its movies will not be deleted, and will still be available for rental. The problem with "Cascade" in this situation, is that if you delete a content provider, and the movie is then deleted, then the rentals should of course be deleted. But should the content provider's rentals or the movie's rentals be deleted first? That is the downside to our current data model: All user types (users, content providers and system administrators) are gathered in one database table. For the database there isn't any difference between a user and a content provider. But in our code, it is not possible to rent a movie if you are a content provider or a system administrator. On the other hand, it is not possible to create a movie if you are not a content provider. So the problematic "Cascade paths" will actually never occur. But the database doesn't know that.

3.2.3 Entity Framework

We used Entity Framework version 4.1 as an ORM (Object-Relational Mapping), which is Microsoft's ORM to compete with NHibernate and the like. With Entity Framework it is possible to query data with LINQ (the so-called LINQ-to-Entities), but where Entity Framework really shines, is that it focuses on code over configuration. There isn't any need to do a lot of configuration to get it working.

There is three or four different ways of getting started with Entity Framework:

Model-First Drag and drop. You get a visual editor where you can create new boxes (which will end up as tables and entities) and put lines between them, to model how the different entities interact with each other. An XML-file is created automatically, and the database and entities are created from this.

Database-First The other way round compared to Model-First. You start with creating the database, and then Entity Framework creates the XML file from the database. Then, just like Model-First, entities are created.

Code-First With code-first you start out from the code. You create your own entities, which are a lot simpler than the generated entities in the two previous modes. These are called POCO-classes, which means "Plain Old CLR Objects", referring to the simplicity of these classes. When the POCO entities are created, you get Entity Framework to create the database for you.

Code-First The final way of doing it is also called Code-First, but you start out with the database. Code-First doesn't refer to what component you start out with, but it means that you are "code-centric". Just like Database-First you start out with creating the database, and then use a simple tool for Visual Studio to generate the POCO entities from this database. This is also nicknamed "Code-Second".

We chose to use Code-Second, as we really wanted to get the simple POCO entities. The reason being that we wanted to send these entities back and forth over the service to transport data, but without any trace of Entity Framework. The reason we chose Code-Second over Code-First, is that we have more control over the database with Code-Second. Entity Framework doesn't add any indices or unique constraints by itself, so to get these, we had to create the database ourselves.

This has proved to be a bit more difficult than first expected, as noone in our team had experience with SQL Server 2008, which is different from MySQL in certain areas. If we used Model-First, anyone in the team could have changed the data model, where as with Code-Second either the entire group had to read up on SQL Server and T-SQL or one team member should be responsible for the database part all by himself. We chose to pick one person as the database responsible, to make sure the rest of the group could continue working on the service in the meantime. This meant that only that person could alter the data model, but we did get complete control over our data model and database, which we felt was a big advantage.

Chapter 4

Testing

In this chapter we discuss our testing strategy, our results using said strategy and what we feel we could have done to improve our testing overall.

4.1 Strategy

Our testing strategy consists of different kinds of tests and a tight integration with how our user stories work in a Scrum development strategy. We have three different kinds of tests: Scenario tests, Service tests and Graphical User Interface (GUI) tests.

Almost all functionality in our RentIt system is implemented by user stories. A user story is not accepted until tests of the functionality pass and the tests have been reviewed by the team member responsible for QA. The team member responsible for QA is Jakob Melnyk. If he is the one who created the test, a different teammember takes care of the QA for that test.

To us, this ensures an acceptable level of peer review during development of the system. Between the feature freeze and code freeze dates, everyone reviews both the code and the tests, so that we get a more thorough peer review.

4.1.1 Test types

We have different testing categories for testing different levels of the system. The different levels we have defined are scenario, service and end-user levels. This section describes the different levels we use for testing. We originally had method/unit testing as well, but we decided the way we have structured our systems do not make it easy to do.

Scenario-level Mainly tests designed to cover a specific user story.

Service-level Mainly consist of testing the connection to the service and the different service contracts.

End-User-level Test functionality of the GUI for the end-user.

Scenario-level tests

We have used scenario tests to test features in our project. Features such as editing a user profile or renting a movie are tested on the classes containing the logic for editing the database, creating filestreams and filtering information. This is done to seperate the logic from the Service class itself, such that the service can be exchanged without affecting the logic much (if at all). This also enables us to test the logic seperately from the service itself.

Most of our tests are scenario tests. We have chosen to put our focus on scenario tests, because scenarios are integral to the way we have designed our service. Most features are implemented by making one or two methods to cover the necessary implementation of the feature [1].

Service-level tests

Our service levels tests do not cover as much of the API as our scenario tests and are not as thorough in testing the functionality. They are mostly intended to test for connection problems, bindings[2] issues, data contracts and error handling.

Graphical interface tests

We use a framework to automate tests for workflows in our GUI. Our strategy is to design them in such a way that they do not depend (much, if at all) on results from the ViewModel[3], but instead test if it is possible to achieve a set of different workflows in the GUI.

4.1.2 Regression tests

Every time we run in to an error or a bug in the system, our strategy is to create a test that covers the scenario the error presented itself in. The test is intended to fail the first time it is run (to verify the bug exists), then when the issue has been fixed, the test passes. These tests are only meant to cover the specific bug they were designed to, but because they are already designed (to check if the bug has been fixed), they are kept as regression tests to ensure the bug or error is not reintroduced later.

4.1.3 Code coverage

It is rarely an effective strategy to cover every combination of paths through the code, because it is very time-consuming (both in creating tests and actually running those tests). Instead it is important to test enough to cover a lot of your code and functionaliy without crossing the line of where it is redundant to do so[1]. With that in mind, we have decided to use twofold approach to our tests. We write tests to cover different inputs and scenarios for our features, then assess our code coverage. If the coverage has not reached our goal, we think up new tests to increase the coverage procentages.

Our requirements for code coverage are as follows:

- Minimum overall coverage is 50% across solution.

[1]Design and architecture is more thoroughly covered in chapter 3 Design.
[2]Covered briefly in chapter 3 Design.
[3]Discussed in chapter 3 Design.

- Goal is to have 85% overall coverage of client and service.
- Minimum coverage of critical sections is 90%.

Minimum overall While a 50% code coverage may seem like a low number, we use this number because the critical parts of our system do not take up a big percentage of the code. This is due to the GUI elements, the Model-View-ViewModel architecture of the client and some elements in the service. In addition, the 50% requirement is only an absolute minimum and not necessarily a number we will be close to in the end.

Overall goal The overall coverage percentage goal is quite different from our minimum requirement. This is because, in an optimal scenario, we will be able to test the MVVM architecture and the GUI parts (and run code coverage) on the tests we write for those parts. The best code coverage percentage is of course 100%. However, seeing as our system is not ”mission critical”[4] as we have not (yet) implemented actual payment options, we feel that 85% is a good number based on previous experiences.

Critical sections Even though we do not have any ”mission critical”, we have features in our system that are critical to both the concept of the system (renting movies) and to make the service work. When it comes to testing these features, we aim to have at least a 90% code coverage. Optimally we would aim to have 100% code coverage, but as mentioned in The Way of Testivitus[1] the benefit of each test created has diminishing returns on the amount of certainty that no errors and/or bugs exist.

The sections we feel are critical are all on the service side of the system.

- Logging in/out and signing up.
- Renting and download of movies.
- Upload and editing of movie information

Why we feel these parts of the code are critical (and why we feel the rest is not as critical) is explained in greater detail in our Design chapter.

4.2 Test results

4.2.1 Code coverage

4.3 Reflection on test strategy

[4]Mission critical in this case refers to loss of life and/or loss of money.

Bibliography

[1] Blog: http://www.artima.com/weblogs/viewpost.jsp?thread=203994 (22nd of April, 2012)