Software Requirements Specification

FlatFindr

Version 2.2.1 approved

group8

ESE, University of Bern

17.10.2016

Table of Contents

Revision History

Name	Reason For Changes	Version	Date
rrö	Added 1, 2, 4.1, 4.2	1.0.1	10.10.2016
rae	Created 4.3, 5.1, 5.3, minor text edits	1.0.2	10.10.2016
dkl	Added 3.2	1.0.3	10.10.2016
ela	Minor style edits	1.0.4	11.10.2016
rae	Removed unnecessary parts, updated following ‘1st discussion’.	1.1.0	16.10.2016
vho	Architectural Guidelines added, design unified	1.1.1	17.10.2016
rae	Removed initial comments from template and created SRS_template.doc as reference	1.1.2	17.10.2016
rae	SRS updated according to ‘2nd discussion’.	1.2.0	19.10.2016
ela	SRS updated according to ‘3rd discussion’.	1.3.0	28.10.2016
rae	Added 4.8 and extracted some of the Functional requirements of 4.6 to 4.8.	1.3.1	31.10.2016
ela	SRS updated according to ‘4th discussion’.	2.0.0	05.11.2016
ela	SRS updated according to ‘5th discussion’.	2.1.0	12.11.2016
ela	SRS updated according to ‘6th discussion’.	2.1.1	28.11.2016
rae	SRS updated according to ‘7th discussion’; Restructured 5.1 into 5.1 and 5.2, added 5.2.2.4. Added link to repo in header.	2.2.0	06.12.2016
ela	SRS updated according to ‘8th discussion’. Minor design updates	2.2.1	11.12.2016

1. Introduction

1.1 Purpose

This SRS documentation describes the requirements of the web application FlatFindr and relates to the most current version. In this SRS documentation all the functional as well as non-functional requirements from the customer are listed. The implementations of those requirements are documented by the developers.

The SRS is available to the customer, the developers and the users.

1.2 Document Conventions

Domain specific terms are *italicized* and explained in the glossary.

1.3 Intended Audience and Reading Suggestions

Our SRS document is meant for different stakeholders:

- The customer: The controlling can always check on the progress made and the correct fulfillment of their needs.
- The developer: Are collecting the customer's requirements and describe their implementation.
- The user: The system will be used by an end-user, which may want to know the complete range of function, which the system provides.

1.4 Product Scope

The FlatFindr web application is a system, which provides a platform for users to place and respond on ads for real estate. The system should be easy to use and provide a variety of different functions, as requested by the customer (see 2.2 Product Functions) When we talk about FlatFindr, we are always referencing the system as a whole unit.

1.5 References

On the following link, the software will be developed and also versioned. Only the developers are allowed to modify these files, and on any questions, same developers are to be addressed at https://github.com/scg-unibe-ch/ese2016-team8 by either contacting the developers or creating an issue in case of malfunctioning software.

2. Overall Description

2.1 Product Perspective

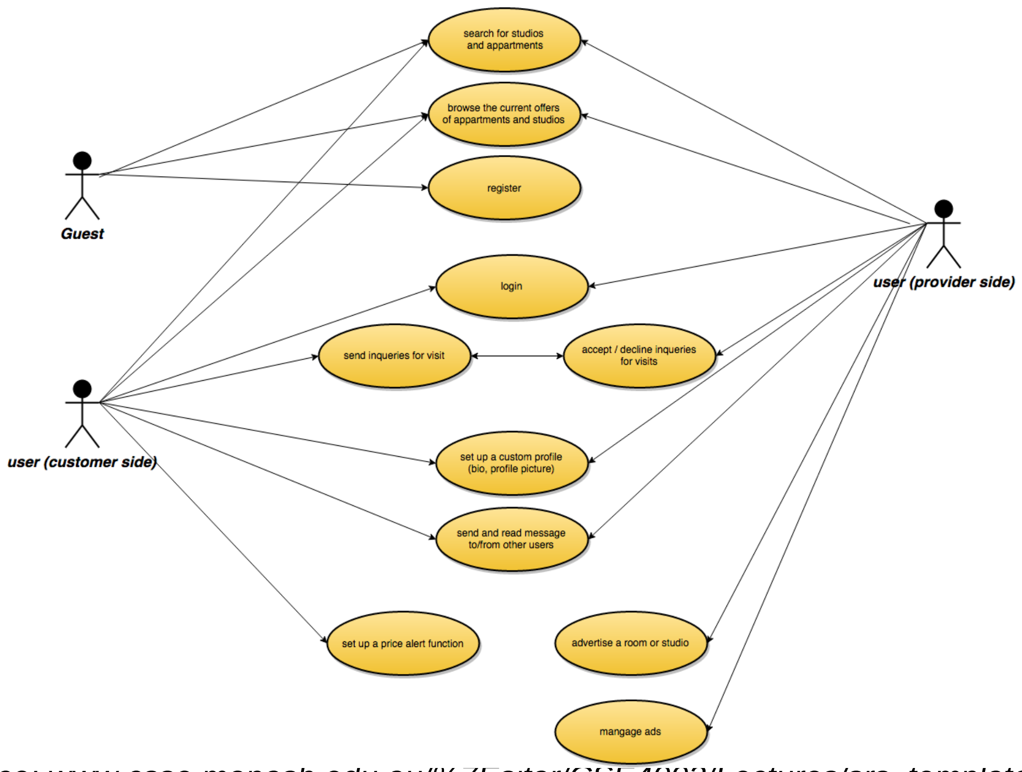
The FlatFindr is an assignment given by the teaching assistants of the Software Engineering Introduction course at the University of Bern. The FlatFindr project is a standalone application and therefore not integrated in other applications.

2.2 Product Functions

The main function of FlatFindr is to provide an online platform for the real estate market. Customers and providers are given the environment needed in order to i.e. rent/ sell apartments, rent rooms in shared apartments, houses, create scheduled visits, organize visits or just get in touch.
There should be some functionality to share more information on ads such as images descriptions etc. Users can appear in different roles, depending on their intention:

- User as customer: login, search for real estate, send messages, send inquiries for visiting hours, set up a price alert function and provide a detailed profile description.
- User as provider: login, place ads, accept and decline visiting inquiries.

For the different roles there should be some additional functionality e.g. list of most promising candidates, bookmark most interesting objects, alerts for interesting objects. Below you'll see a use case diagram, which gives a brief overview of the functions provided by the system. (See chapter 4 for detailed scenarios.)


search for studios and appartments
browse the current offers of appartments and studios
register
Guest
login
user (provider side)
send inqueries for visit
accept / decline inqueries for visits
user (customer side)
set up a custom profile (bio, profile picture)
send and read message to/from other users
set up a price alert function
advertise a room or studio
mangage ads

2.3 User Classes and Characteristics

2.4 Operating Environment

The application should run on a webserver, which supports HTTP protocol and runs a JSP engine.

2.5 Design and Implementation Constraints

Standard Object-Oriented Design principles/ patterns **are to be followed.** Further the more principles as MVC and also Server-Client basics are to be used/ implemented.

Following are the **architectural guidelines**:

2.5.1 Naming

- The names of classes modelling services **should** end with the suffix 'Service'. The suffix Service should only be used for service classes
- The names of classes modelling a controller **should** end with suffix 'Controller'. The suffix Controller should only be used for controller classes.
- The names of classes modelling a form **should** end with the suffix 'Form'. The suffix Form should only be used for form classes.
- The names of classes modelling DAO objects (repositories) **should** end with the suffix 'Dao'. The suffix Dao should only be used for DAO objects.

2.5.2 Package structure

- All controller objects **should** be placed in a single package named <your_package_prefix>.controller. You can choose the name for <your_package_prefix>. The package <your_package_prefix>.controller should only contain controller classes. Utility classes needed by controllers should be placed in a package named <your_package_prefix>.controller.utils.
- All service objects **should** be placed in a single package named <your_package_prefix>.controller.service. The package <your_package_prefix>.controller.service should only contain service classes. Utility classes needed by services should be placed in a package named <your_package_prefix>.controller.service.utils
- All classes modelling forms **should** be placed in a single package named <your_package_prefix>.controller.pojos.forms. The package <your_package_prefix>.controller.pojos.forms should only contain forms.
- All model entities **should** be placed in a package named <your_package_prefix>.model. The package <your_package_prefix>.model should only contain model entities. Utility classes needed by model entities should be placed in a package named `<your_package_prefix>.model.utils`.

- All classes modelling DAO objects **should** be placed in a package named <your_package_prefix>.model.dao. The package <your_package_prefix>.model.dao should only contain DAO classes.

2.5.3 Dependencies

- Controller objects **should not** be referenced or called directly from any part of the system. This includes also jsp files. Currently the file getsipcodes.jsp is an exception to this rule. You do not have to fix this, however, you should not access services this way.
- Forms **should** only be accessed by controller classes.
- DAO objects **should** only be accessed from service classes.
- Model classes **should** only reference classes from the package <your_package_prefix>.model or any of its sub-packages.

2.5.4 Design

- Within a service class attributes that point to Dao objects (repositories) or other services **must** be annotated with *@Autowired* and not initialized explicitly.
- Within a controller attributes that point to services **must** be annotated with *@Autowired* and not initialized explicitly.
- Within a controller all public methods **must** model service requests (be annotated with *@RequestMapping*).
- DAO objects and services **should** only be stored in instance attributes.
- All access to the database **should** be done through DAO objects. One **should not** use SQL queries to extract data from the database. Currently the service GeoDataService breaks this rule, but as specified by the customer, this service doesn't need to be fixed, but however, one should not use SQL queries in any other services.
- Form classes **should** only contain accessors and attributes.
- All requests to a controller objects **should** have logging support. A log entry **should** be added when a controller receives a request and when the controller returns a value to the caller. Long entries **should** be placed in the file controller.log. Log entries should contain relevant information related to the request. Log entries should follow a common structure.

2.5.5 No files saved client-sided

The developers are asked to design and implement the software in such way that nothing of the following occurs:

- A file is being created at the client's file system.
- An existing file is being modified at the client's system.
- A folder is being created at the client's file system.
- An existing folder is being modified at the client's file system.

2.6 User Documentation

Compare the ‘discussion files’ from https://github.com/scg-unibe-ch/ese2016-team8/tree/master/baseProject/Documentation labeled i.e. ‘1st discussion.txt’, ‘2nd discussion.txt.’ and so on in order to look up what the resumé of the meetings, and therefore the user input, was.

2.7 Assumptions and Dependencies

2.7.1 Assumptions

Everything that is an **implementation detail** and *does not* influence the functionality of the services will be assumed by the developers in the manner of *2.5 Design and Constraints*. If neither specified by the user nor asked by us will be, in case of no further specification, designed according to basic principles of Object-oriented programming and best practice.

2.7.2 Dependencies

The target is to keep the ‚FlatFindr’ as slim as possible, therefore best performing. This demands the developers to use the least additional libraries/ frameworks possible and also best practices for the currently used ones.

3. Accessibility

The website can be accessed from both standard and mobile browsers. Hence normal desktops as well as mobile Phones such as I Phone or Android running devices can read and display the website. So the Hardware is not really limited as long as the device is able to run a browser.

4. System Features

4.1 Sign up

4.1.1 Description and Priority

A vital function of our system is the registration process for new users. They should be able to enroll and gain access to all the features our system provides. (Alternatively see section 4.2)

4.1.2 Stimulus / Response Sequences

When finding the FlatFindr platform a user should be able to create a private account. Therefore he has to enter his first and last name, as well as his email address and a custom password.

4.1.3 Functional Requirements

The password should not be stored in plain text in the database. Use SHA-256 or higher to ensure at least some encryption.

4.2 Google Sign up

4.2.1 Description and Priority

As an alternative to the original sign up (see 4.1) there should be the possibility to login with an exicsting google account to gain access to all the features our system provides.

4.2.2 Stimulus / Response Sequences

When finding the FlatFindr platform a user should be able to login with his private google account without having to create an account as described in section 4.1.

4.2.3 Functional Requirements

Google OAuth protocol should be used for authorization on a google server.

4.3 Search offers

4.3.1 Description and Priority

Another vital function of our system is the search engine. Users should be able to search all offers and filter depending on their specific needs.

4.3.2 Stimulus / Response Sequences

A user may only want to see advertisements around his current location, or in a specific price range. The system provides an easy search function for those needs.

4.3.3 Functional Requirements

There are different search criteria, which are specified by the user. As a result the system returns all the offers that match the filter. All the matches that are shown to the user should be made visible on a map automatically generated by the system. (so called gedige)

4.4 Create offers

4.4.1 Description and Priority

Users should be given a platform to create offers and further the more chose whether they want to rent, sell the object and also whether to sell directly or by auction.

4.4.2 Stimulus / Response Sequences

Additionally to the offer itself, a user should also have the possibility to add pictures of the object he wants to offer in various formats.

4.4.3 Functional Requirements

The offer itself and all information belonging to it should be saved server-side, and precious information should not be sent in plain text when creating the entry in the DB.

4.5 Rent estate

4.5.1 Description and Priority

A customer, a user who wants to rent something, should be able to do so by clicking on a on-page-button.

4.5.2 Stimulus / Response Sequences

By doing so, the user, and also the owner of the offer, get notified.

4.5.3 Functional Requirements

Display the monthly rent of the object and also create a communication-friendly environment for the user and the owner can come up with a ‘deal’.

4.6 Buy estate

4.6.1 Description and Priority

A customer, a user who wants to buy something, should be able to do so by clicking on a on-page-button.

4.6.2 Stimulus / Response Sequences

By doing so, the user, and also the owner of the offer, get notified.

4.6.3 Functional Requirements

Display the total price of the object and also create a communication-friendly environment for the user and the owner can come up with a 'deal'.

4.7 Bid for estate

4.7.1 Description and Priority

An owner of an object must be able to choose whether he wants to sell his object directly or if he wants the customers to **bid** for it.
The bid will proceed until the time has run out and therefore the most-bidding client will 'win'.

4.7.2 Stimulus / Response Sequences

The owner is not allowed to expand the time-limit. In order to so he needs to cancel, under official law for bidding procedures, his offer and create a new one.
Also users aren't allowed to withdraw their bid, once placed the bid is by lawsuit valid (And also if users don't pay, measures will be taken).
All new bids should be at least CHF 50 higher than the previous bid.

4.7.3 Functional Requirements

-Once the bidding-time is over, the most-bidding user 'wins' the auction.
-The highest-bidding value will be displayed and also the amount of remaining time/ the expiration date will be shown.
-The owner of the estate should be notified, if he wishes so, once a new bid is offered/ being placed.
-When choosing to watch/ bookmark a bidding, users will get notified when certain time limits, in the meaning of remaining time, will be reached.
-Premium users will have an advantage in the case of bidding, which one hasn't been set yet.

4.8 Messages

4.8.1 Description and Priority

Users should be able to send messages between each other. Messages should be stored in an in-box.

4.8.2 Stimulus / Response Sequences

The messages should go to the correct recipient, marked as unread and he should be able to respond.

4.8.3 Functional Requirements

Every user should have an in-box which should only be accessible by the owner. Each message should have a read variable to show unread message.

4.9 Messages related to bidding

4.9.1 Description and Priority

There should be several functionalities related to messaging concerning ‘auctionable’ ads.

4.9.2 Stimulus / Response Sequences

Every functionality defined below (as of now 4.8.3) should finish in a message sent to the specified user, may it be a user as a client or as an advertiser.

4.9.3 Functional Requirements

- When being the most-bidding user when the time runs out, the user should get notified he ‘won’ the auction.
- When being the most-bidding user, and one gets over-bidden by another user, one should get notified in terms of a message.

4.10 Notifications and supporting features

4.10.1 Description and Priority

There should be some features that help the user while browsing on through the page. The goal is that every user should be comfortable at any time while being on the page and never get lost or lose any unnecessary time.

4.10.2 Stimulus / Response Sequences

Users should be informed about new objects on the page. Browsing through a large database takes a lot of time and is an exhausting occupation specially when already most of the data is already known. When logging in the user should be able to recognize new ads.

4.10.3 Functional Requirements

When logging in there should be some function to show new (since last login) inserted ads on the database.

5. Nonfunctional Requirements

5.1 Non-Premium membership

5.1.1 Description

People should be able to have a membership on the page. This should be completely free and available for everybody. All functional requirements require at least a membership to access to the full functionality of our page.

Non-Premium user should be reminded of their limited membership by showing them a little (negative) image instead of a fabulous star (see 5.2 Premium membership).

5.2 Premium membership

5.2.1 Description

Users should be able to pay a certain amount of money every month in order to profit from a better service, also called premium.

When a user is registered as a premium user this is made recognizable with
a little star on a visually remarkable place.

Premium Users will have following advantages (see 5.2.3).

5.2.2 How to obtain premium

A user should push on a button to show his interest in upgrading to premium. After agreeing on the terms and conditions the user should be able to instantly pay the membership. Once this is done his account should instantly upgrade to a premium account including all the benefits.

5.2.3 Benefits

5.2.3.1 More filter criteria

5.2.3.2 Early results & alerts

5.2.3.3 Ads are placed at the top

5.2.3.4 Premium Ads are visible marked, that they have been offered by a premium user.

5.3 Bookmark

5.3.1 Description

Users should be able to bookmark an offer in order to save it in a personalized list and have easier access later on in order to access it.

5.4 Performance Requirements

5.5 Safety Requirements

Additionally to the points mentioned in 2.5, developers are also asked to keep their code clean in terms of:

- No clear names/ addresses/ phone numbers/ e-mails of the developing team/ and or relatives, and other people working in the same company, in the code in any matter.
- No absolute paths which show more information than what's readable from the GitHub-repository.
- Information about clients are to be treated as **confidential** and therefore should not be visible to any extern in any matter unless the customer wishes so (which is not the case of now).
- In terms of confidentiality no profiles should be publicly linked to thehighest bidder nor any other bidder in any auction.

5.6 Security Requirements

Every data-entity will, once sent to the server, be saved server-side (in a database). There should be no data consistently client-sided. Everything important to any functionality of 'FlatFindr' should be once created, be saved server-side and therefore considered 'safe'.

5.7 Software Quality Attributes

Additionally to the points of 2.5 the software must be reusable, clean, documented and also generic. The software must meet these high standards in order to fully satisfy our customer in every aspect of the service.

6. Appendix A: Glossary

Word	**Description**
Offers	An offer is an entry on FlatFindr which atm. Can either be: i) a room ii) a studio iii) a house

	iv) shared apartment / commune All of the above can be rented, sold or sell-by-bid.
Sell-by-bid	Instead of vending something directly, the owner of an estate may wish to let users bid for his estate, compare the 'System feature: 4.6, Bid for an estate'.
Bid/ bidding	By the word 'bid/ bidding' an action is meant which is specified during a predefined amount of time where a certain amount of money, also called bid, is promised to the owner of the ad, if the bidder wins the auction.

7. Appendix B: Analysis Models