University College London

Department of Computer Science

MSc Summer Project Report

Little Bits of Good

A Web Application for Connecting Software Developers with Charities

Author:
Mikko Poutanen

Supervisor:
Dr. Graham Roberts

September 2013

Abstract

The idea of the project was to explore the concept of creating an online platform with social media integration, where charitable organisations could find volunteer software developers for their IT projects. The idea was the author's own and the application was not developed for a third party. Nevertheless, the goal was to build an attractive, robust, maintainable, and deployable web application. This was largely achieved.

The application back end was created using Python and its Django framework with its numerous open source libraries. The web browser front end was built with HTML5, CSS3, and JavaScript and also took advantage of many publicly available CSS and JavaScript libraries. There is also a separate mobile front end. Many of the afore-mentioned technologies were new to the author and the learning experience was intense but rewarding.

There is scope for future improvements and extensions, but overall most of the goals and aims were achieved and the project can be considered a success.

Acknowledgements

I'd like to start by thanking my supervisor Dr Graham Roberts, whose support, guidance, and ideas were invaluable during all stages of the project. I would also like to thank my friends, flatmates, and family for their understanding, support, and well-timed encouraging words throughout the summer. I can't wait to return the favour.

Contents

List of Figures

Abbreviations

AJAX	**A**synchronous **J**avaScript **A**nd **X**ML
API	**A**pplication **P**rogramming **I**nterface
CBV	**C**lass **B**ased **V**iev
CSRF	**C**ross **S**ite **R**equest **F**orgery
CSS	**C**ascading **S**tyle**S**heets
DRY	**D**on't **R**epeat **Y**ourself
ERD	**E**ntity **R**elationship **D**iagram
FBV	**F**unction **B**ased **V**iev
GB	**G**iga**B**yte
HTML	**H**yper**T**ext **M**arkup **L**anguage
HTTP	**H**yper**T**ext **T**ransfer **P**rotocol
HTTPS	**H**yper**T**ext **T**ransfer **P**rotocol **S**ecure
IDE	**I**ntegrated **D**evelopment **E**nvironment
LBOG	**L**ittle **B**its **o**f **G**ood
MAC	**M**essage **A**uthentication **C**ode
MVC	**M**odel **V**iew **C**ontroller
ORM	**O**bject-**R**elational **M**apper
OWASP	**O**pen **W**eb **A**pplication **S**ecurity **P**roject
PDB	**P**ython **D**e**B**ugger
RAM	**R**andom **A**ccess **M**emory
SSL	**S**ecure **S**ockets **L**ayer
TDD	**T**est **D**riven **D**evelopment
TLS	**T**ransport **L**ayer **S**ecurity
UI	**U**ser **I**nterface

UP	**U**nified **P**rocess
URL	**U**niform **R**esource **L**ocator
VM	**V**irtual **M**achine
XML	e**X**tensible **M**arkup **L**anguage
XSS	**X**(C)ross **S**ite **S**cripting

Chapter 1

Introduction

1.1 Problem Domain

The concept of Little Bits of Good (LBOG) was born on an idle night when the author was browsing the web and looking for possible charity software development projects to contribute to. As the search went on, it soon became clear that whilst platforms for matching developers with charities exist, none of them allowed for developers to work on smaller projects from the comfort of their own home, when it suits them. However, anecdotal evidence suggested that this is how many software development enthusiasts prefer to work on projects outside of their main job. As the idea began to take shape, it became clear that a platform allowing for easy matching of volunteer developers with suitable charity projects could provide a viable addition to existing forms of charity in the field of IT.

Whilst donating one's resources to charity is an age-old concept, it is still mostly being achieved by directly asking for money or by requiring volunteers to be on site and even to take time off work to be able to contribute to projects. For many kinds of activities, for example taking care of elderly citizens, being physically present is the only possible way of donating one's time. However, websites such as

PeoplePerHour[1] and eLance[2] as well as numerous open source software projects have already proven that freelance software developers are able to effectively contribute to projects by telecommuting. This is achieved by taking advantage of modern communication methods like video conferencing and discussion forums as well as platforms such as GitHub[3] that allow for straightforward code distribution, review, and deployment.

"What's the point of doing something good if nobody's watching?"

Nicole Kidman

Perhaps better known for her 5'11" stature and starring roles in numerous blockbuster movies, Nicole Kidman summarises a growing body of evidence stating that in many cases the true motivation behind helping those less fortunate isn't necessarily a purely altruistic one. In the author's opinion this isn't inherently either good or bad. Instead, it motivated the author to explore the problem by trying to come up with new ways to motivate people to contribute to charitable projects. The research behind the concept of this project is covered in more detail in Section 2.1.

1.2 Project Goals

In summary, the goal of the project was to create an online platform for matching volunteer software developers with charitable projects that they can complete on their free time on a pro bono basis. The site should be seamlessly integrated with social media to maximise publicity both to attract as many people and projects as possible to the site and to motivate volunteers to donate their time. The following list provides an overview of a more specific set of goals.

[1] www.peopleperhour.com
[2] www.elance.com
[3] www.github.com

- Create a secure and deployable web application.
- Create a clear user interface that is easy to use for the general public.
- Ensure maintainability and compatibility with Django design patterns to ensure ease of future improvements, either by the author or others.

1.3 Personal Aims

After the "GC06 - Database and Information Systems" module, the author was keen to learn more about creating Web 2.0. applications where users are largely responsible for the site contents. Having already gained exposure to PHP, the summer project seemed like an attractive opportunity to learn a completely new language and framework. The choice was quicky narrowed down to Ruby on Rails and Python's Django framework. From the application specification standpoint either one would have been a viable option. The decision to go with Python was largely a personal preference and heavily influenced by the variety of ways it is used in commercial applications as opposed to Ruby, which is mainly popular within the web developer community. The following list provides an overview of the project goals:

- Select a new language/framework to learn that is suitable for the task, in this case Python/Django[1].
- Learn more HTML and CSS, including the use of open source packages such as Twitter Bootstrap[2] and Chosen[3].
- Learn more about creating interactive user interfaces with JavaScript and the jQuery[4] library.
- Use Linux as the development and deployment environment and learn how to configure an Apache[5] server to host a Django project both locally and on a virtual machine.

1.4 Project Management

The project planning followed the five stages of the Unified Process (UP) as defined by Arlow and Neustadt[6]: requirements, analysis, design, implementation, testing. Given the number of new technologies to learn, the best choice for organising the project workflow was the incremental model as shown in Figure 1.1. The incremental model assumes that the initial requirements are reasonably well defined which was the case with LBOG. In the initial stage a very basic version is created, tested and then deployed. In the second increment more features are added, tested and deployed and so the process continues until final delivery date[7]. This approach allowed the author to learn about every component of a Django application (models, views, templates, form handling, server configuration, static files etc.) in the first iteration with only a handful of features. The second increment was built noticeably faster due to having learned from the mistakes made in the first one. Ultimately the process was refined to a stage where new features could be added in rapid succession.

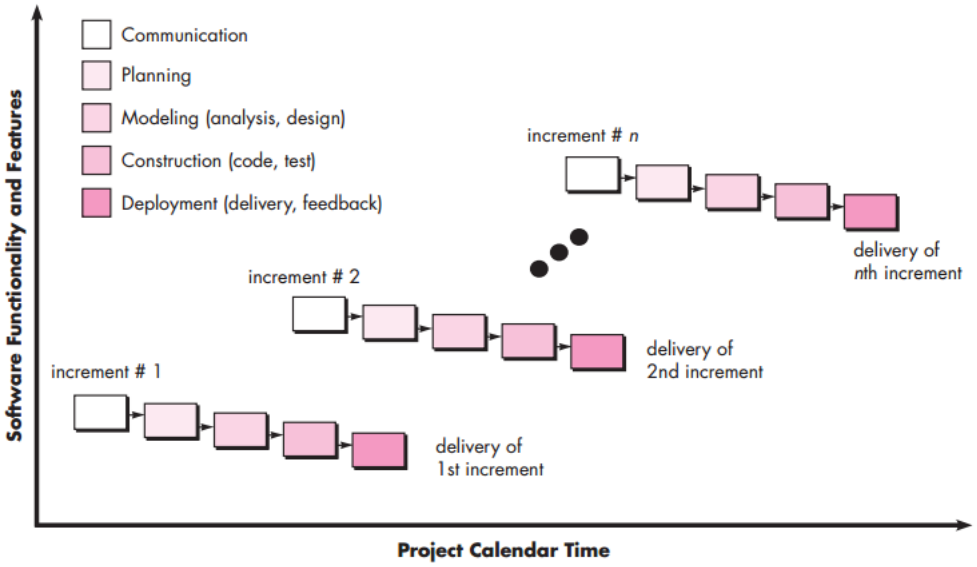

FIGURE 1.1: The Incremental Model[7]

1.5 Report Overview

Chapter 2

Chapter 2 includes a literature review, a competitor analysis, and an overview of the programming languages, libraries, and tools used in this project.

Chapter 3

Chapter 3 gives a detailed description of the requirements and use cases as well as the design of the data model and website templates.

Chapter 4

Chapter 4 starts with a general description of the structure of a Django application. The remainder of the chapter covers the application design as well as handling security issues.

Chapter 5

Chapter 5 covers the testing methodologies used during the project.

Chapter 6

Chapter 6 will provide the reader with the author's own evaluation of the project's successes and failures. The chapter concludes with ideas for future work and some final thoughts.

Chapter 2

Background Information and Related Applications

2.1 The Science behind Charitable Deeds

A significant amount of research has gone into trying to model what drives people's willingness to sacrifice their own resources for the benefit of others and consequently how to entice them to do so. Starting with the question of whether charities should ask for time or for money, Aaker and Liu find clear evidence that the two trigger different parts of the brain and that the emotional bond created by time donations generally results in a more meaningful overall contribution[8]. Intuitively this makes sense as the donor is able to see the results immediately.

But why do people do this? Historically researchers had started with the assumption that any government activity in producing public goods would crowd out private donations dollar for dollar, which is a logical extension of the idea that donors are only interested in the well-being of the recipients of the donations. It seems however intuitive that there are other factors motivating potential donors and this was formalised as a concept in 1989 by James Andreoni [9]. Based on empirical evidence, Andreoni constructs a model he calls "impure altruism" that

includes a "warm glow" effect, which in essence sums up the good feeling a donor gets from donating irrespective of the outcome to the receiving party. In a meta-study from 2007, Andreoni, Harbaugh, and Vesterlund [10] find references to both pure altruism and warm glow affecting people's willingness to support charitable causes. Following these findings the focus shifted to research on how to best entice people to contribute to charity. In their 1989 research paper, Cooter and Broughman explore different policy options to increase charitable activities. Their results highlight the importance of publicity over e.g. tax breaks as a catalyst for good deeds among individuals[11]. In other words, volunteers and donors want others to know what they've done.

Whilst the author of this report does not claim to have done a thorough research review on the subject of charitable donations, there is strong enough evidence that a platform that makes it easy and convenient to publish one's charitable work could well be an effective tool in increasing people's participation in such activities.

2.2 Existing Applications

This section gives an overview of the competitor analysis. The findings are listed below and are grouped together by type of charity.

Give Camp[1], Code for Good Challenge[2], Social Innovation Camp[3], Battle Hack[4], Coders for Charities[5]: All of these organisations work in a similar manner, by organising weekend-long hackathons where programmers can work on charitable projects. This approach has its advantages, but it is also far less accessible than an open internet platform. First, it requires programmers to set aside an entire weekend for a charity event as opposed to working in shorter intervals

[1] http://givecamp.org/

[2] http://techcareers.jpmorganchase.com/techcareers/us/opportunities/codeforgood/ny-feb2013

[3] http://sicamp.org/

[4] http://battlehack.org/

[5] http://coders4charities.org/

when suitable. Second, it typically requires the charities to register their projects well in advance and have them vetted by the event organiser. Both of these can be considered additional hurdles that LBOG avoids.

Social Coding 4 Good[6]: Social Coding 4 Good is somewhat similar to LBOG in that it allows volunteers to apply to work on a small number of preapproved projects from home. The projects on the site are generally of a very large scale and focused on providing generally useful services (e.g. free access to e-books, free subtitling etc.) rather than solving a specific problem for a specific organisation.

IT4Communities[7]**:** IT4Communites places IT professionals to charities to help them with their IT needs. The main differences to LBOG are that typically the volunteers will work on site and that the organisation charges a yearly management fee from the participating charities.

Catchafire[8]**, VolunteerMatch**[9]**:** These two come quite close to what LBOG attempts to achieve in that they provide an online platform for matching volunteers from various fields with projects. They are however not specialised in software and lack the suitable filtering tools such as searching by IT skills. Also, their main focus is on projects where volunteers work on site.

2.3 Programming Languages and Libraries

2.3.1 Server Side Programming

As explained in section 1.3, one of the author's personal aims in the project was to learn a new server-side language and framework. As there were no pre-existing limitations on the hosting platform itself, Python and its Django framework ended up being the most attractive option. Django is a powerful framework that is particularly well-suited for database backed websites managing large amounts of varying

[6]http://www.socialcoding4good.org/
[7]http://www.it4communities.org.uk/
[8]http://www.catchafire.org/
[9]http://www.volunteermatch.org/

content[12], making an excellent fit for this project. Before learning Django, the author had to gain a level of comfort with Python's syntax. After a review of the available books (many of which are available online), Mark Pilgrim's Dive Into Python 3[13] quickly emerged as the favourite learning resource. Rather than complete beginners, the book's target audience are people with experience in programming who need a quick introduction to Python. In addition TheNewBoston's video tutorials were a excellent resource for getting started[14]. After learning Python, the author took time to get familiar with Django. The starting point for this was the Django homepage[1] and its "Writing your first Django app" tutorial. In addition to the documentation on Django's homepage, the author was frequently referring to "Two Scoops of Django" by Daniel Greenfeld and Audrey Roy[15] as well as to "The Django Book" by Adrian Holovaty and Jacob Kaplan-Moss[12]. Finally, comprehesive series of video tutorials by Mike Hibbert[16] and a collective of software development enthusiasts called HackedExistence[17] were also used for addressing specific problems.

Being an open-source project, Django also comes with an impressive set of open source libraries. The Django libraries used in LBOG include the following:

Braces

Braces[18] is a library consisting of various mixins for Django class-based views (See Section 4.2). It was mainly used for access control.

Django_extensions

Django_extensions[19] provides a set of additional tools for Django. In LBOG it was mainly used for its custom Python shell which imports all the application modules automatically and thus saves time when interacting with the database in the development phase.

Cleanup

Django_cleanup[20] works in the background and ensures that whenever a user changes an image attached to a database entry, the old one is deleted (See more on image storage in Section 4.1).

South

South[21] is a powerful tool for handling changes to the database schema. Invariably during every project there is a need to make minor changes to the data model. By default Django does not detect changes to models and is only able to create new database tables for each new model. Any schema changes would require the developer to either drop and create the database or to go to the database console and manually implement these changes using e.g. the ALTER TABLE SQL command. South however automates this process and was an invaluable timesaver during the project. For future reference, South is also able to handle database migrations.

Django-Mobile and Mobileesp

Django-mobile[22] is a Django application that replaces the default middleware with one that detects the browsing device and renders a different template accordingly. This was combined with Mobileesp[23], which allows for more detailed options to ensure that all tablet devices and mobile phones are viewing the mobile interface.

Django-dbbackup

Django-dbbackup[24] provides an easy command line interface for backing up and restoring the database and media files.

2.3.2 Client Side Programming

The decision to build LBOG as a web application dictated more or less the use of HTML and CSS as the markup language for designing the page templates and JavaScript to make them more interactive. The author decided to use HTML5 and CSS3, the latest versions of the languages. HTML5 and CSS3 are not yet official standards and browsers continue to add support for their different features as new versions are released[25]. This may cause compatibility issues with older browsers (primarily Internet Explorer 6-8), but as these are becoming less and less prevalent, it was deemed the best overall solution both for the application itself and

for maximising the learning experience. In addition to numerous online sources for learning HTML and CSS such as StackOverflow[10], W3Schools[11], and NetTuts[12], the author mainly used two books, "Responsive Web Design with HTML5 and CSS3" by Ben Frain[26] and "HTML and CSS - Design and Build Websites" by Jon Duckett[27]. For JavaScript the main sources were online forums and tutorials. In addition numerous libraries were used to help make the front end look better and easier to use. These are listed below:

jQuery

jQuery[4] is a JavaScript library that allows developers to write concise and elegant code. It abstracts away time-consuming tasks such as checking for the browser type during an AJAX call and generally provides an efficient syntax for repeatedly occurring tasks. Overall jQuery was an indispensable tool during the project.

Twitter Bootstrap

Twitter Bootstrap[2] is a collection of CSS and JavaScript files that provides a wide array of UI design features from widget formatting to fully responsive components with JavaScript features. In LBOG it was mainly used for its column system, navigation bar and formatting of form elements.

Chosen

Chosen[3] is a set of easy-to-use form elements. In LBOG it is used in single and multi select dropdown menus to provide a search box.

MapQuest

MapQuest API[28] is an open source geocoding library. In LBOG it is used for retrieving coordinates when creating and changing Projects and User Profiles, and when searching Projects or Developers by location.

[10] www.stackoverflow.com
[11] www.w3schools.com
[12] http://net.tutsplus.com/

2.4 Development Environment and Tools

The project was developed on a Linux Mint 14 powered computer. The decision to learn a new operating system was mainly made to increase the learning effect as well as the ease of later deploying the project on a Linux Virtual Machine (VM). The text editor of choice was Sublime Text 2. It provides superb text editing features such as multiple cursors, shortcuts for moving lines of text at a time, fast file navigation etc. In addition it's multitude of external packages such as Emmet (HTML Code Completion), Djaneiro (Django syntax highlighting and autocompletion) PyLinter (Python syntax checker), SublimeCodeIntel (Code Autocompletion for multiple languages) provide it with many IDE-like capabilities.

One of these capabilities however isn't debugging. Due to the different components of a web application and thus a wide array of potential bugs, there is no one-size-fits-all solution for debugging web applications. On the server side, with debug mode enabled, Django provides detailed error messages in the browser window when an error is found. For more subtle bugs such as database queries that don't return the expected values, Django's development server can be used together with Python debugger (PDB). PDB pauses the Python code at the point where it's inserted and allows the user to inspect and make changes to variables in the Python shell before continuing with the processing of the HTTP request. The front end debugging was done with integrated browser tools. To get the CSS formatting right, Firebug and Chrome Developer Tools both allow for changing parameters on the fly without having to go to the text editor and refresh the page to test different layouts. Chrome Developer Tools was also vital for debugging JavaScript, as without a specialised tool the scripts will simply fail silently at the first error and with hundreds of lines of code, locating the error using e.g. alerts becomes a nightmare. Chrome Developer Tools provides detailed feedback on where the script fails and for what reason.

Testing tools and strategies are covered in detail in Chapter 5.

2.5 Production Environment and Other Tools

To deploy the application the author chose to set up a Linux VM with an Apache server processing the HTTP requests. Thanks to UCL's collaboration with Microsoft, this could be done free of charge on Windows Azure[13] which was an added bonus. There was a considerable learning curve in learning how to configure the server correctly for Django, locating and interpreting the error logs, learning when the server needs to be restarted and so on. There was no single source for this information, so it ended up being gathered from numerous different online sources as specific questions arose. One of the advantages of running an Apache server on a Linux VM is that the deployment environment can be tested locally with near-identical settings. For version control the author chose Git due to some previous familiarity. As the project matured, the typical development cycle consisted of creating a branch in the project, writing the code, testing it, and finally merging it with the master branch and pushing it to GitHub[14]. This tested branch could then be downloaded on to the production server with a simple "git pull". For more information on how Git works, please refer to Git homepage[29]. PostgreSQL[30] was chosen as the database engine. The reason was simply to try out a different database management system to MySQL, which the author had used on a previous course. Finally, the UML charts were produced with a tool called Visual Paradigm[31].

[13] http://www.windowsazure.com/
[14] https://github.com/

Chapter 3

Requirements and Analysis

3.1 Problem Statement

The problem statement and project goals have not been subject to many changes during the course of the project and have been covered in detail in chapters 1 and 2. To summarise it again, LBOG was about exploring the concept of creating a web platform where developers could find charitable projects to work on in their free time. The code was to be well structured and thus easily maintainable so that if the project was to go live, it would also be possible for others to maintain it or even contribute in an open source setting.

3.2 Requirements Gathering

In a typical project the software engineers and the client go through a series of interviews, refining the requirements during the process. In this project there was no external client to gather requirements from, so the nature of the exercise was somewhat different and typical risks of for example misinterpreting the client or changing requirements at the last minute[7] were non-existent. Even without a client, motivation for executing requirements gathering thoroughly was however not difficult to find as Everett and McLeod find that a staggering 85% of all

software defects are introduced *before* any code has been written[32]. As the author had complete freedom, the emphasis shifted towards ensuring that the requirements are complete and well prioritised and that the overall scope would be suitable for a three-month project. There needed to be some flexibility on which features to include as nearly all technologies used were previously unknown to the author and it was thus difficult to predict how long each phase would end up taking.

Figures 3.1 and 3.2 show a complete listing of the functional and non-functional requirements. Functional requirements represent things that the system should do and non-functional requirements other types of constraints. The requirements are prioritised using the MoSCoW methow, where the letters stand for Must, Should, Could, and Want to have (note that sometimes the W is interpreted as Won't or Would have, but has the same meaning). Must haves are features that need to be included, and in the context of this project can be interpreted as the author's vision of what the application needs to be considered a minimum viable product. Should haves are important requirements that can be omitted if time were to run out. Could haves are truly optional features and Want to haves are features that can wait for future versions of the software[6]. The bolded and capitalised words represent entities that would eventually become models and bolded, but not capitalised words represent potential model fields.

Req. ID	Functional Requirements	Category	Priority
FReq1	The system shall allow **Administrators** to edit and delete all accounts	Accounts	Must
FReq2	The system shall allow **Users** to create new accounts	Accounts	Must
FReq3	The system shall allow **Users** to edit their **Account** details and deactivate accounts	Accounts	Must
FReq4	The system shall have three different user account types: **Developer**, **Charity**, and **Administrator**	Accounts	Must
FReq5	The system shall allow existing **Administrators** to upgrade other users to **Administrators**	Accounts	Must
FReq6	The system shall log **Users** in with a **username** and **password**	Accounts	Must
FReq7	The system shall allow **Users** to log out	Accounts	Must
FReq8	The system shall enable the **Users** to recover a lost **password**	Accounts	Must
FReq9	The system shall authenticate **Users** before posting **Projects** or asking to help on **Projects**	Accounts	Must
FReq10	The system shall allow **Charities** to search for **Developers** by **keyword**, **distance**, or **Skills**	Accounts	Must
FReq11	A **Developer** shall be **allowed** to opt out of appearing in search results and unsolicited **contact**	Accounts	Must
FReq12	The system shall allow the **Users** to send **e-mails** to each other	Communications	Must
FReq13	The system shall allow the **Users** to contact site **Administrators**	Communications	Must
FReq14	The system shall use a geocoding service to determine the **Users'** and **Projects'** coordinates	Other	Must
FReq15	The system shall allow **Administrators** to create, edit and remove **Projects**	Projects	Must
FReq16	The system shall allow **Charities** to create, edit and delete **Projects**	Projects	Must
FReq17	The system shall allow **Charities** to change **Project** status	Projects	Must
FReq18	The system shall allow **Charities** to decide which **Developers** are signed up for **Projects**	Projects	Must
FReq19	The system shall allow **Developers** to ask to help in **Projects**	Projects	Must
FReq20	The system shall allow **Developers** to search for **Projects** by **keyword**, **distance**, or **Skills**	Projects	Must
FReq21	The system shall allow likes and shares with Facebook	Social Media	Must
FReq22	The system shall allow Administrators to write, edit, and delete **Stories** about the **Projects**	Stories	Must
FReq23	The system shall allow **Users** to change the UI language	User Experience	Should
FReq24	The system shall render a different interface to mobile and desktop devices	User Experience	Should
FReq25	The system shall allow **Users** to communicate via a discussion forum	Communications	Could
FReq26	The system should allow **Developers** to post comments and questions on **Projects** on a discussion forum	Communications	Could
FReq27	The system shall allow custom Facebook activities such as thank you messages	Social Media	Could

FIGURE 3.1: Functional Requirements

Req. ID	Non-Functional Requirements	Category	Priority
NFReq1	The system shall be easy and pleasant to use	User Experience	Must
NFReq2	The system shall be an online service accessible via a web browser	User Experience	Must
NFReq3	The system shall store accounts and other info into a relational database	Data Storage	Must
NFReq4	The system shall be built with HTML5 and Python/Django	Other	Must
NFReq5	The system database shall have capacity for at least 1,000 projects and a similar number of **Users**	Capacity	Must
NFReq6	The system shall be secure against common hacker techniques	Security	Must
NFReq7	The system shall terminate the session after 60 minutes of logging in	Security	Should
NFReq8	The system shall be available a minimum of 360 days a year, 24 hours a day	Capacity	Should
NFReq9	The system shall login a **User** within 5 seconds	Capacity	Should
NFReq10	The system shall be compatible with Chrome, Mozilla Firefox, Opera and Internet Explorer	User Experience	Should
NFReq11	The system shall be able to accommodate 100 simultaneous logins	Capacity	Should
NFReq12	The system shall have data recovery plan in place	Security	Should
NFReq13	The system shall allow advertising on the front page	Other	Could
NFReq14	The system shall track statistics on a Project and website level	Other	Won't

FIGURE 3.2: Non-Functional Requirements

3.3 Use Cases

Use cases are an effective way of modeling the interaction between a system and its users, especially if the system requirements are primarily of the functional type (as opposed to e.g. performance-related) and there are different actors with different user interfaces[6]. Each use case will have a primary actor, in LBOG either a Developer, a Charity, or an Administrator, and the use case flow will describe in detail the user's actions and the system's responses from the user's point of

view[33]. Shortly after the requirements gathering, a list of the key actors and an overview of the use cases was produced. Detailed specifications for each use case were added as the page structure was being sketched. Figure 3.3 gives an overview of the use cases and detailed descriptions can be found in Appendix A.3. These are written using a template provided by Arlow and Neustadt[6].

ID	Use Case	Primary Actors	Secondary Actors
UC1	RegisterDeveloperAccount	Developer	None
UC2	RegisterCharityAccount	Charity	None
UC3	UpgradeUserToAdmin	Admin	None
UC4	LoginToSystem	All	None
UC5	Logout	All	None
UC6	ChangeAccountDetails	All	None
UC7	DeactivateAccount	All	None
UC8	ChangePassword	All	None
UC9	RecoverForgottenPassword	All	None
UC10	ContactSiteAdmin	Developer, Charity	None
UC11	CreateProject	Charity	None
UC12	EditOwnProject	Charity	None
UC13	DeleteProject	Charity	None
UC14	SearchProjects	Developer	Charity
UC15	ViewProjectDetails	Developer	Charity
UC16	ViewCharityDetails	Developer	Charity
UC17	SearchDevelopers	Charity	Developer
UC18	ViewDeveloperDetails	Charity	Developer
UC19	ContactCharity	Developer	Charity
UC20	ContactDeveloper	Charity	Developer
UC21	OfferToHelp	Developer	None
UC22	ViewHelpOfferStatus	Developer, Charity	None
UC23	DeleteHelpOffer	Developer, Charity	None
UC24	AcceptHelpOffer	Charity	None
UC25	RejectHelpOffer	Charity	None
UC26	CreateStory	Admin	None
UC27	ReadStory	All	None
UC28	DeleteStory	Admin	None
UC29	LikePageOnFacebook	All	None
UC30	SharePageOnFacebook	All	None
UC31	LikeProjectOnFacebook	All	None
UC32	ShareProjectOnFacebook	All	None

FIGURE 3.3: Overview of Use Cases

3.4 Template Design

In order to not confuse the reader at this point, it is worth pointing out that what might conventionally be referred to as views in the MVC pattern, are referred to as templates in the Django framework. This is explained in more detail in Section

4.2. The template design process went hand in hand with writing the use cases. Initially they were sketched on paper and then turned into HTML pages. At this stage the key was not to get all the formatting to be perfect but rather to get the page structure right and identify shared parts of the HTML code. LBOG's user interface aims to follow the core design principles of Donald Norman, including the following (Norman's keywords in bold[34]):

- **Feedback** is given to the user after any changes to the data.
- Navigation controls are kept **visible** at all times.
- The user is **constrained** wherever possible by offering multiple choice widgets instead of free text fields and by hiding irrelevant links.
- The **mapping** is kept consistent by offering for example the same menu bar for items related to the user's account on all account-related pages.
- The design largely follows common web design patterns and will look **familiar** to the users at first glance.
- **Error** checks are in place everywhere where the user is entering data or accessing restricted pages.

In addition LBOG incorporates key points for effective website UI design laid out by Rogers, Sharp and Preece. These are easy navigation, aesthetically pleasing looks, and plenty of visible links to other parts of the site[35]. The first point has been covered earlier in this section. As far as the second point goes, the author has used frameworks such as Twitter Bootstrap[2] but did not place too much emphasis on graphic design as it is a time-consuming effort best left to professionals. Throughout the project the goal was to make the site look good enough to not distract from proving the concept itself. The third point is interesting in that researchers have found out that the typical internet user will only glance at a website and click on the first interesting looking link. For this reason the front page of LBOG provides plenty of captivating pictures and additional links to interesting

content to make it as readily available as possible. A screenshot of the front page is provided in Appendix A.1

3.5 Analysis and Data Modelling

The data model was derived from the keywords identified in the requirements and use cases. LBOG's data model was relatively small and straightforward to create, so the author decided to skip the analysis class diagram and conceptual entity relationship diagram (ERD) and move straight to a logical ERD which is included in Appendix A.2. A logical ERD captures entity relationships, foreign keys, primary keys, attributes and so on, but does not yet contain information about the physical schema such as table names and column data types[36]. Each of the tables in the ERD contains an id column as a primary key. The id columns were added later to reflect how Django implements the tables. Physical database schema modelling was unnecessary at this stage as it is taken care of by Django after defining the model classes.

A data model is in third normal form when all attributes depend on the entire primary key and there are no transitive dependencies[36]. LBOG's data model conforms with these principles. It is also for the most part perfectly normalised. Help Offers is the only entity where a separate receiver field is redundant. When a Developer is sending a Help Offer on a Project, the receiver could be defined as the project owner. The author simply felt that having "symmetrical" sender and receiver fields for both Notifications and Help Offers makes the code more readable and results in less complicated and faster SQL queries. Also, the system does not allow for changing the Project's owner, so there is no danger of data inconsistencies.

3.6 Other Notes on Analysis

As explained above, the analysis part focused on creating an accurate representation of the data entities and their attributes as well as sketching prototype views and creating the page structure. No object-oriented analysis was conducted at this point for the simple reason that Django enforces a certain structure on the code and the author was at this point unfamiliar with it. The structure of Django's Model-Template-View pattern and the implementation of the analysis and designs is covered in detail in Chapter 4.

Chapter 4

Design and Implementation

4.1 Three Tier Web Application Structure

LBOG more or less follows the classic ANSI-SPARC three-level application structure, which is depicted in Appendix B.1. The three tier structure separates a web application in to three layers, the presentation tier, logic tier, and the data tier. The core idea is that views are separate from the logic tier so that the same data can be presented and manipulated through different user interfaces with no changes to the application structure. In addition, the user should be able to communicate with the database without knowing how the data is physically stored (e.g. hashing, indexing or table structure). Also, by separating the physical data tier from the logic tier, the database administrator can for example swap the data-storage hard drive to a backup copy without affecting the application itself[36]. In LBOG the database is not stored on a separate physical drive but instead resides on the same virtual machine. Furthermore, due to Django's default image-handling properties, images are not stored in the database itself for example as large binary objects as would be the recommended way[37]. Instead, the path of the image is stored as a normal varchar-field and the image itself is stored in the application's media-folder.

4.2 Django Model-Template-View

Before covering LBOG's design in detail, it is worth taking a closer look at Django's Model-Template-View (MTV) pattern and its differences to the classic Model-View-Controller (MVC). In MVC, a model is where the application's data objects are stored and each data entry represents an instance of a model. A view is what's presented to the user and defines how users interact with the app. A controller is the glue between these two classes. It updates the view when the model data changes and also adds event listeners to the user interface, updating the model when the user interacts with the view[38].

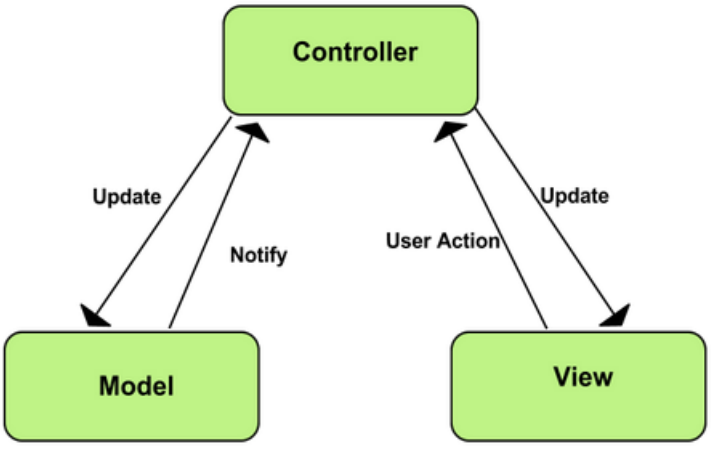

FIGURE 4.1: MVC Pattern[38]

Django's MTV is somewhat different. Models work similarly to their counterparts in MVC, containing the business logic and being responsible for accessing and manipulating data. Templates are similar to views in MVC in that they are responsible for creating the user interface. This is achieved through HTML, CSS, JavaScript and Django's template language. The capacity of the template language is restricted on purpose and mainly offers simple "if" statements and "for" loops for presenting the data passed on to it by the application. There is no Python code within the templates and it is not possible to make database queries directly from them, structurally enforcing separation of concerns[12].

The controller functionality in Django is implemented in a different way. A part of

it lives inside the framework and is implemented through its URL dispatcher. All incoming HTTP requests are redirected to the URLconf file which can be set in the project settings. The URLconf file contains a list of possible URLs which are a mix of hard coded text parameters and regular expressions. The latter are used for passing parameters to views. The incoming HTTP request is then matched to one of the URLs in the URLconf by the "patterns" function. Each URL defined in the URLconf file maps to a different view, which are the closest equivalent to a controller class that the framework has (and in the author's opinion named somewhat confusingly)[39]. From here on in any reference to a "view" should be interpreted as a Django view unless specified otherwise. The views can be implemented as Python functions or classes that inherit from one of Django's base view classes. The following example is taken from the Django documentation and gives a simple illustration of a class-based view (CBV)[40].

```
# example code from Django documentation
from django.views.generic.edit import UpdateView
from myapp.models import Author

class AuthorUpdate(UpdateView):
    model = Author
    fields = ['name']
    template_name_suffix = '_update_form'
```

The UpdateView class automatically contains a form object based on the specified model, in this case "Author", and knows which instance of Author to update based on the id passed in the URL. The URL might look something like this: *www.example.com/users/update-author/10/*, where number 10 corresponds to the id of that database entry (all database tables generated by Django contain an id column by default). The "fields" variables defines which form fields to include in the form to be rendered and the "template_name_suffix" will cause the framefork to automatically look for a template named "author_update_form.html". Using the example URL above, the template will now be able to render a form containing

the name of the "Author" instance with the id 10. In Django, forms are submitted to the same URL and the view will be able to distinguish between a GET and a POST request, where the former renders the form and the latter saves the changes.

In reality the views are often more complex than the above example. The view design in LBOG is covered in more detail in Section 4.4.3.

4.3 Django Application Structure

A Django project comprises of the main application folder and additional, separate applications as well as static files (CSS, JavaScript, application images, etc.)and is best illustrated by way of an example. The example on the following page is from a simple project called example_project that contains two subfolders. When the project is first created, Django creates the main application folder called "example_project" (always same as the project name). The main application folder contains by default the application settings and the main URLconf file (urls.py). The wsgi.py file contains server-related settings. Following the main application folder, the project root contains the manage.py file, which is essentially a thin wrapper for the "django-admin.py" command line utility that provides it with some project-specific information. It is used for a variety of things such as starting the development server, accessing the project's data through Python shell, initialising new applications, and others[41]. Lastly, the users folder is a separate application within a project created by the author. The folder contains the autogenerated files after creating the application plus a urls.py file for the application-specific URLs. As can be seen, both folders contain an __init__.py file, meaning that they are structured as Python packages.

```
/example_project
├── example_project/
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
├── manage.py
└── users/
    ├── __init__.py
    ├── models.py
    ├── tests.py
    ├── urls.py
    └── views.py
```

The above example is of course highly simplistic. It does not contain any templates, CSS-files, JavaScript-files, form files, or any custom Python modules. The point is to illustrate how Django allows for modularising different parts of a project. If there were no cross-references to models imported from other applications, the applications could be freely reused in other projects, although in most cases the CSS or template files would need to modified to fit the overall look and feel of the project.

4.4 Application Design

This section provides a detailed look on the design decisions made in LBOG. The first three subsections cover the three main parts of a Django application: models, templates, and views. Finally, subsection 4.4.4 explains the search function. Given Django's modular approach to building a project, the project was developed one application at a time, including all the models, views, and templates as well as related files. The learnings from that process were then applied in the next part of the project. The order was approximately as follows:

1. Users-application, including creating and modifying accounts, password management and login
2. Projects-application, including creating and modifying Projects, matching Developers to Projects, and Notifications
3. Search functions for Projects and Developers
4. Stories application
5. Creating a mobile interface

4.4.1 Models

Django models are the core of each application and define the structure of the application data. Each of the models is a Python class that subclasses the base Django model from "django.db.models.Model". Generally speaking, each of the models maps into a database table and each of their fields represents a column within that table. Models with several common fields could be subclassed from an abstract base model, but this was not needed in LBOG. After creating a logical schema (See Appendix A.2), the conversion to Django models is a very straighforward effort. Django provides some syntactic shortcuts through for example its many-to-many fields, which will automatically generate the table sitting between the two related entities. Once the models have been created, all the programmer needs to do is run the "python manage.py syncdb" command to create the database tables. The models also provide a complete database access API that abstracts away the need for for the programmer to write any raw SQL. Accessing the database through the models' API has the added benefit of Django's automatic SQL injection checks (Security is covered in more detail in Section 4.6).

The author feels that the overall model design is already very accurately represented by the logical schema created in the design process (Appendix A.2) and because of this no separate class diagram was produced. The best way to illustrate

the conversion of the designs to Django models is through a series of examples. The main difference to the logical ERD is the use of Django's User class. The User class provides methods for registration, authentication, managing user groups and others. Attaching additional information to each User object is achieved through a separate UserProfile class, which was written by the author.

The first example shows all models from the Projects application and can be found in Appendix E.1.1. As can be seen from the code, the models map one-to-one to each entity and attribute in the logical ERD. The second and third examples illustrate querying the database through the model API. The example in Appendix E.1.2 shows a series of simple example queries and the example in Appendix E.1.3 illustrates a more complex query that is generated through a list of Django Q-objects (see comments in code example).

Appendix B.3 shows a screenshot of how Django's admin interface looks in LBOG. This is an built in functionality and all that is needed to use it is to enable in the URLs and settings and register each model in a separate admin.py file. After this is done, a user with an Administrator account can freely add, edit, and delete all database entries from all applications within the project. As an example, in LBOG there is no separate form for adding Stories. As the Stories are only to be written by the site's Administrators, they can rely on the admin interface to do this.

4.4.2 Templates and Visual Design

Django template language is designed for abiding by the DRY-principle and supports both "include" and "extend" functionalities. This results in clean code that is easy to maintain and build on[15]. The diagram in Appendix B.2 shows the full page structure as Django templates. The author was unable to find an established UML notation for Django template diagrams and was forced improvise by

applying conventions from other design diagrams. The notations in the diagram are explained below. The names of the yellow template classes are always prefixed with "base" and can be thought of as abstract views as they are never rendered by any of the views. The subclassing arrows denote an "extends" relationship and as can be seen from the diagram, each class subclasses "base.html", which contains e.g. the header and the footer. The blue templates are the ones that are called by the views. The purple templates denoted by the "import" connector are pieces of HTML code that are used in multiple templates by calling the "include" function. The "import" notations do not include refactorings that were introduced to make the code more readable if the included HTML-files were only used on a single template. An example of this is provided in Appendix E.2.2. Some of the templates include method names. These should be interpreted as a sort of pseudocode as there are no method calls in Django templates. The methods denote the data that would need to be passed on to the template from the view and allowed the author to design the views in the same diagram. Appendix E.2.1 shows the "base.html" file. Together with the previous example they illustrate both extending and including templates.

Carefully designing and documenting the template hierarchy is important not only to avoid repeated HTML code but also for reusing CSS and JavaScript files. This was especially important in the later stages of the project when the mobile version of the site was being built. As a result of a well-designed template structure, most of the existing HTML code could be used in its existing form or with minor modifications and the diagram gave an easy overview for making targeted changes to CSS and JavaScript files.

For the visual design the two main libraries used were Twitter Bootstrap[2] and Chosen[3]. Bootstrap offers several built in components and features that make the life of a web designer easier. Starting with the layout, Bootstrap's scaffolding system defines one main div-element of class "container" that has a width of 960px. Inside the container there are div-elements of class "row", which can be split into

12 columns simply by giving the div-elements inside the rows class definitions such as "span3" or "span7". This feature was used extensively throughout the project. Bootstrap also offers different styles for form elements, a collapsible navigation bar, and other features. Chosen offers enhanced form elements and was used for its single and multi-select dropdown menus which offer a search box. For example in geocoding it is much safer to limit the user's country input to a selection from a dropdown menu rather than offering a free text input field, but browsing through nearly 200 countries would be an arduous task without a search box.

JavaScript was used mainly for enhancing the user experience, making AJAX-queries, and connecting with Facebook. For the first two, the JQuery library[4] was used extensively as it provides several syntactic shortcuts for common operations. The Facebook likes and shares are largely based on modified sample code from the Facebook Developers website[42], which includes extensive instructions on how to register your application on the Facebook Developer Console and start building in functionalities such as likes and shares.

4.4.3 URL Routing and Views

As explained in Section 4.2, all incoming HTTP requests are routed through the root URLconf file. This file will typically contain some URLs mapping directly to views like the home page or the in-built login function URLs, but must include references to the URL files of the different applications. This helps keep the applications as separate as possible and contributes to reusability. With a completed template diagram (B.2), designing the URL and view structure was a straightforward process as each of the templates and AJAX calls maps to a separate URL/view pair as is the Django best practice[15]. Appendices E.3.1 and E.3.2 show a part of the main URL file in LBOG as well as the full urls.py file from the "users" application, respectively. The collapsed view in Appendix E.3.3 shows

that each of the views maps 1:1 to one of the users application URLs. The password_change and password_change_done views are a part of the Django framework and could be easily integrated to LBOG by overriding the templates.

Most of the classes were implemented as CBVs, which were introduced in Django 1.3 (LBOG uses version 1.5). They provide greater opportunity for code reuse through subclassing from a wide array of built in base classes as well as Mixins that take advantage of Python's multiple inheritance. CBVs are currently the recommended default option[15] and were the favoured option in LBOG. Getting the view classes to work as desired was one of the greatest challenges during LBOG. The author's own view the code in its current state is readable and concise, but learning exactly which base classes to use and how to customise the views by selectively overriding methods required intense research of literature, online sources, and ultimately Django's source code. An example of a view class that updates the user account information is provided in Appendix E.3.4. The main use for function-based views (FBVs) were the AJAX-calls and the reason is that most of the documentation showed it that way. An example showing the deletion of a Notification can be found in Appendix E.3.5.

4.4.4 Search

An important feature on a site like this is to create algorithms for matching Projects with Developers. In LBOG this is mainly achieved through the search functionalities for both Projects and Developers as well as displaying the latest Projects on the front page. Currently the search functions on the "For Volunteers" and "Looking for Volunteers" pages allow for filtering by three different categories, keywords, location, and skills. The keyword search looks for each keyword in the Project and Developer descriptions (this can be used to search for e.g. specific programming languages). The skill search looks for the skills a Developer has listed as their core areas of expertise or the skills a Project has listed as required skills. The location search will first retrieve the coordinates of the location with

the MapQuest API[28] and then look for Developers or Projects within the distance specified by the user. The search function also allows for filtering within the same country. The search functionality was created as a separate module to enable code reuse. The geolocation JavaScript code can be found in Appendix E.4 and the search module in Appendix E.5.

4.4.5 Facebook Integration

One of the core ideas of the application was integration with social media. Based on anecdotal evidence, the first site that most people think of when talking about it is Facebook[1], which was chosen as the starting point. The current version of the site allows for liking and sharing both the site itself as well as individual projects. The code itself is all in JavaScript and is mostly based on code samples from Facebook Developers[42]. There was also some preparatory work involved as any Facebook integration requires registering a Facebook application on their developer console.

4.5 Data Validation

A good rule of thumb for all web developers is to never trust user input[43]. Data validation is important both to prevent malicious users from interfering with the application's functionality and to make sure users enter the type of data they intended. LBOG's data validation takes place in three stages covered in the following sections.

4.5.1 Client-Side Controls

In Web Application Hacker's Handbook, Stuttard and Pinto explain that ultimately anything that happens on the client side is within the user's control and

[1] www.facebook.com

describe several ways to bypass controls such as hidden form fields, data stored in cookies, url parameters, and JavaScript validation[43]. For this reason LBOG only implements client-side controls to assist benign users to enter correct data instead of protecting against malicious users. This is mainly achieved by using multiple choice UI widgets instead of free text input where applicable as well as JavaScript validation. The latter is used for one purpose only and that is to make sure the user has entered sufficient geographic data before an AJAX request is submitted to MapQuest's geocoding service to determine their location. Example is provided in Appendix E.6.1.

4.5.2 Server Side Form Validation

In Django, forms are created as classes that inherit from one of its inbuilt Form classes. The preferred Django way of rendering these on the screen is to inherit from one of Django's inbuilt FormView classes that expect a form object as a parameter (for example the UpdateView in Section 4.2 is one such view class. All requests are sent to the same URL and these views will then either render the form (GET request) or handle the form (POST request). When a POST request arrives, the view executes its form_valid()-method, which in turn executes all of the form's clean_ methods. For simple forms it is sufficient to simply run the default clean() method, which checks every form field based on its attributes (max length etc.), but in most cases it is necessary to create custom checks. These can be added by simply creating a clean_fieldname() method, where fieldname is replaced by the name of the field, or by overriding the clean()-method. By keeping the form validation logic out of the views, the forms become reusable in other applications within or outside the current project. Appendix E.6.2 provides an example of a simple form (for sending a HelpOffer) with only one field, a message field. However, the clean()-method requires some additional checks to make sure the Help Offer is legitimate. The form validation logic limits the size of all uploaded images are to 1MB, but as Django is only able to evaluate image size after it has been uploaded, there is an additional safeguard in the Apache server settings to limit

the size of any uploaded image to 10MB in case a malicious user tries to upload an unrealistically large image (see Apache settings in Appendix C.5).

4.5.3 Database-Level Controls

The most common attack against web applications today is an injection attack which means a malicious user entering e.g. SQL code into a form on a website[44]. Whilst Django allows for raw SQL code to be written, LBOG only makes database queries through the Django object-relational mapper (ORM). These queries are fully protected against SQL injections[45]. Equally important however is to correctly implement constraints in the database schema to ensure that for example foreign keys are not entered as nulls and that maximum length parameters are observed in case something has been forgotten in the form validation. Also, in larger applications the database might be accessed and manipulated by other applications where the database administrator does not have control over form validation. The model fields in Appendix E.1.1 contain some constraints such "null=False" (NOT NULL in SQL) or "max_length=50" which limits the length of a VARCHAR field. All models in LBOG apply such contraints where necessary.

4.6 Security

Despite the majority of Fortune 1000 companies' IT security resources going towards firewalls, experts estimate that over 70% of attacks attempting to find holes in security systems come through the organisations' web applications[44]. This section covers the measures taken to make LBOG as secure as possible against malicious users.

At this stage it should be pointed out that LBOG does not utilise the HTTPS protocol, but instead relies on regular HTTP. HTTPS uses either the secure sockets layer (SSL) or transport layer secure (TLS) which encrypt any data sent between the client and the server and decorate this data with message authentication codes

(MACs). This prevents an eavesdropping middle-man from either understanding or altering the data[44]. At this stage when LBOG is still more of a proof of concept, an SSL certificate was deemed unnecessary.

4.6.1 Access Control

Access control in web applications can be split into two stages, authentication and authorisation. The former checks whether the user is who they claim to be and latter checks if that user has access to a certain page[44]. This is very important in any web application, but Django's URL system makes it particularly easy to exploit lacking access controls as accessing another user's data is often a simple matter of replacing e.g. a database id or a username in the URL.

As explained in Section 4.4.1, most of the views were implemented using Django's class-based views. Taking advantage of Python's multiple inheritance, each of the view classes that contain sensitive information inherits from two mixins, LoginRequiredMixin and CorrectUserMixin. LoginRequiredMixin was imported from django-braces and CorrectUserMixin is the author's own creation. The AJAX calls are implemented as function-based views, where validation is also done. The flattened view in Appendix E.3.3 illustrates the use of mixins in multiple classes. Appendix E.7.1 shows the implementation of the CorrectUserMixin class and Appendix E.3.4 shows how it can be used in a view (a custom error message is given in the class variables and "self.url_id" is set in the "get_initial" method). Finally, the example function-based view in Appendix E.3.5 shows how access control was implemented in the AJAX calls.

As an additional note, the default password storage methodology in Django is the PBKDF2 algorithm with an SHA256 hash, the methodology currently recommended by the United States National Institute of Standards and Technology[46]. The framework also adds a salt, a random string that increases the complexity

of weak passwords enormously[47]. The author saw no need to deviate from this algorithm.

4.6.2 Data Backups

The data backups are currently done with a tool called Django-dbbackups[24]. It provides an easy to use command line interface for backing up the data and provides several different storage options. The author decided to use Dropbox[2], a widely used cloud storage platform. The following examples show the ease of the backup process once the Dropbox keys and secrets are set up in the settings. Note that to use these commands, the Linux user must have rights to access the database.

```
### Create a backup of the database
/var/www/lbog $ python manage.py dbbackup
### Restore the database
/var/www/lbog $ python manage.py dbrestore
### Create a zipped copy of the media files
/var/www/lbog $ python manage.py backup_media
```

Currently the tool doesn't offer automatic recovery of the media files, but this is simple enough to do by downloading the tar-file from Dropbox and extracting it in to the application's media folder. At present the backup process isn't automated and whilst this would be a desirable improvement for a deployed version, manual backups were deemed sufficient at this stage.

4.6.3 The OWASP Top 10

OWASP, or Open Web Application Security Project, is a not-for-profit charitable organisation that was established in 2001 in the United States. It is an "open

[2]www.dropbox.com

community dedicated to enabling organizations to conceive, develop, acquire, operate, and maintain applications that can be trusted"[48]. As a part of its efforts it periodically publishes a list of the ten most critical web application security risks, the latest of which was published in June 2013[49]. The list below covers (in OWASP's order of importance) LBOG's approach to each of these.

1. Injection

This is covered in detail in Section 4.5.3.

2. Broken Authentication and Session Management System

This is a broad topic, but in general LBOG is protected against most of the attack techniques. For example, Django automatically prevents session fixation by preventing session IDs from appearing in the URL. The framework also and hashes and rotates session IDs which protects against brute force attacks[12],[45]. Also, the settings are configured so that sessions are terminated automatically after 1 hour of logging in. In the current version of LBOG, login data is sent over an unencrypted connection which does make it vulnerable to hijacking. The reason for this is explained at the beginning of Section 4.6.

3. Cross-Site Scripting

XSS means a malicious user injecting HTML or JavaScript into the database which then gets displayed back on the website when rendering database values on to templates. Django template parser automatically escapes all HTML special characters in normal content (between HTML tags on a template). The only time LBOG uses user input to set an HTML tag attribute value (i.e. not between HTML tags) is when it is creating weblinks. These are unescaped values, but protection against XSS is achieved simply by wrapping the value in quotes[45],[12].

4. Insecure Direct Object References

"A direct object occurs when a developer exposes a reference to an internal implementation object such as a file, directory, or database key"[49]. To

maximise security, one could implement a separate server for serving static files. This was not done in LBOG. In the current implementation, the Apache server and Django only allow the user to view the media and static files (JavaScript, CSS, and image files) and so far the author's manual attempts to access other files have failed.

5. Security Misconfiguration

The author is not aware of any security misconfigurations (or else they would've been changed). LBOG follows common recommendations (e.g. in [44]) and kept development and production settings separate. These local settings files contain information such as the database access password, the application's secret key, the site's email account information as well as critical data for the Facebook and Dropbox plugins. None of this sensitive data is available on e.g. GitHub.

6. Sensitive Data Exposure

Sensitive Data Exposure refers to unencrypted sensitive data such as login information or credit card information. LBOG does not encrypt data for reasons explained at the beginning of Section 4.6.

7. Missing Function Level Access Control

This refers to users gaining unwanted access to pages that require authentication and belong to a certain user. The implementation in LBOG is covered in detail in Section 4.6.1.

8. Cross Site Request Forgery

"A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application"[49]. Django generates a CSRF token that must be included in any POST request. LBOG follows the recommended security policy that all database-altering requests (including AJAX) should be POST requests (as opposed to GET).

9. Using Components with Known Vulnerabilities

The author is not aware of any such components. The Django framework

itself has been in existence since 2003 and an open source project since 2005[12]. The author feels comfortable with its security features.

10. Unvalidated Redirects and Forwards

An unvalidated redirect might result in the user being redirected to a malicious website. Django settings includes a list of allowed hosts that redirects are allowed to, largely mitigating this risk[50].

Chapter 5

Testing

The importance of software testing across the entire project is difficult to exaggerate. It is best illustrated by a figure based on the research of Basili & Boehm, who note that fixing a bug is 100 times costlier if it is discovered after delivery compared to being discovered in the requirements and technical specifications capture phase. They do however also note that for smaller and non-critical systems the ratio may drop to 5:1 and that with careful design and implementing best practices in the development phase, the cost is reduced considerably for large systems as well[51]. In any case, early testing will reduce overall development time considerably.

5.1 Testing Phases

Copeland defines four levels of testing that can be applied to any software engineering project. Unit testing refers to testing individual parts of the code during the development phase. Integration testing means combining these different units and testing whether they work in harmony. System testing is used to test the functionality of the entire system. Finally, "acceptance testing is defined as that testing, which when completed successfully, will result in the customer accepting the software and giving us their money"[52]. Of these strategies the first three

were implemented as outlined in the remainder of this chapter. Acceptance testing wasn't applicable at this point as the application is more a proof of concept and only had to satisfy the author himself.

5.2 Testing Strategy

In the beginning of introducing test-driven development, Martin outlines four basic testing strategies, Static Testing, White-box Testing, Black-box testing, and Performance Testing. Static testing refers to testing that occurs *before* any code has been written, i.e. whether the documentation and specifications make sense. White-box testing refers to testing with the source code available and designing tests accordingly. In a black-box testing scenario the tester only has access to the executable files. Lastly, performance testing seeks to test the applications capacity to cope with high workloads[53]. In LBOG, the emphasis was on static testing and black-box testing with some elements of white-box testing. Performance testing with unusually high workloads was not conducted, although with 1.75GB of RAM and 20GB of hard drive space, the server should easily be able to cope with anything that can be considered "normal" use. The testing strategies in LBOG are described below.

5.2.1 Static testing

In LBOG Static Testing was done by creating a clear and detailed set of requirements before writing a single line of code. In addition, an overview of use cases was created and more detailed descriptions were added to these as the project progressed. The author feels that the static testing was a success as it was very rare for poor specifications to cause changes in the code later in the project.

5.2.2 Unit Testing / White-box Testing

In true test-driven development (TDD) a programmer observes three rules[53]:

1. "You are not allowed to write any production code until you have first written a failing unit test."
2. "You are not allowed to write more of a unit test than is sufficient to fail—and not compiling is failing."
3. "You are not allowed to write more production code that is sufficient to pass the currently failing unit test."

Django offers an excellent built in toolkit for unit testing[1] that creates a mock database for each testing session and allows to test different parts of the program. Unfortunately, due to the author's lack of previous knowledge about Python and Django as well as limited knowledge about HTML and JavaScript, TDD was not a realistic approach. In all likelyhood the tests would have to have been rewritten as often as the code itself as in many cases the desired outcome (i.e. what to test) of a specific part of code was clear only after significant experimenting and rewriting of code. In future Django projects this would certainly be an area to improve. The Django toolkit was mainly utilised in testing the behaviour of the models with different invalid data inputs. This was necessary as black box testing should never be able to reach this far assuming that form validation has been set correctly. The unit tests are located in the "<application_root>/users/tests.py" file.

5.2.3 Black Box / Integration and System Testing

The main testing strategy for the LBOG project was black box testing using Selenium IDE[2], a tool used for driving the web browser through its user interface.

[1] https://docs.djangoproject.com/en/dev/topics/testing/overview/
[2] http://docs.seleniumhq.org/

The advantage of Selenium is that it tests both the behaviour of the back end and the front end at the same time and thus has the capacity to perform integration tests and system tests simultaneously. In addition it is also capable of testing the application in a production environment via the user interface. The level of integration required was also a drawback of choosing Selenium as the main testing tool as both the front end and back end of the application needed to be created before any testing can be done. For large parts of the application unit testing essentially fell by the wayside for reasons explained in Section 5.2.2. However, given the small scale of the application and the fact that the programming was a single-person effort, this ended up not causing any significant issues.

The tests were designed as cascading tests around the use case definitions. The definition of cascading test cases is that they are not independent of each other[52]. For example in LBOG, the test case for creating a project would precede deleting a project. An essential part of this kind of a testing strategy is to generate representative mock data. As explained in Section 4.4.1, Django allows for database interaction through the Python shell, so creating a module that generates mock data was relatively straightforward. The mock data generator code resides in "<application_root>/filldb.py" and instructions for its use can be found in Appendix C.6.

The following figure includes each of the use cases and all the Selenium tests that were applied. The tests include the main and alternative flows for each use case as well as additional tests whose names are self-explanatory. Each of the tests has passed in both development and deployment environments.

ID	Use Case	Selenium test cases
UC1	RegisterDeveloperAccount	uc01_main_flow, uc01_alt_flow1, uc01_alt_flow2, uc01_already_logged_in
UC2	RegisterCharityAccount	uc02_main_flow, uc02_alt_flow1
UC3	UpgradeUserToAdmin	n/a (tested manually via admin interface)
UC4	LoginToSystem	uc04_main_flow, uc04_alt_flow
UC5	Logout	uc05_main_flow
UC6	ChangeAccountDetails	uc06_main_flow, uc06_alt_flow, uc06_extra_other_users_account
UC7	DeactivateAccount	uc07_main_flow
UC8	ChangePassword	uc08_main_flow
UC9	RecoverForgottenPassword	uc09_main_flow
UC10	ContactSiteAdmin	tested manually (need access to email to confirm)
UC11	CreateProject	uc11_main_flow, uc11_alt_flow, uc11_logged_in_as_developer
UC12	EditOwnProject	uc12_main_flow, uc12_alt_flow, uc12_extra_change_other_users_project
UC13	DeleteProject	uc13_main_flow, uc13_extra_delete_other_users_project
UC14	SearchProjects	uc14_main_flow_distance, uc14_main_flow_keywords, uc14_main_flow_skills
UC15	ViewProjectDetails	uc15_main_flow
UC16	ViewCharityDetails	uc16_main_flow
UC17	SearchDevelopers	uc17_main_flow_distance, uc17_main_flow_keywords, uc17_main_flow_skills
UC18	ViewDeveloperDetails	uc18_main_flow, uc18_no_permission_to_view
UC19	ContactCharity	uc19_main_flow
UC20	ContactDeveloper	uc20_main_flow
UC21	OfferToHelp	uc21_main_flow, uc21_logged_in_as_charity, uc21_already_offered
UC22	ViewHelpOfferStatus	uc22_main_flow, uc22_extra_other_users_account
UC23	DeleteHelpOffer	uc23_main_flow, uc23_wrong_user
UC24	AcceptHelpOffer	uc24_main_flow, uc24_wrong_user
UC25	RejectHelpOffer	uc25_main_flow, uc25_wrong_user
UC26	CreateStory	n/a (tested manually via admin interface)
UC27	ReadStory	n/a (tested manually via admin interface)
UC28	DeleteStory	n/a (tested manually via admin interface)
UC29	LikePageOnFacebook	n/a (tested manually)
UC30	SharePageOnFacebook	n/a (tested manually)
UC31	LikeProjectOnFacebook	n/a (tested manually)
UC32	ShareProjectOnFacebook	n/a (tested manually)

FIGURE 5.1: Use Cases Mapped to Selenium Tests

Chapter 6

Conclusions and Evaluation

6.1 Project Goals

The initial goals for the project were defined quite broadly on purpose as there was significant uncertainty in the beginning about how fast the development would proceed. I am generally happy with the results, although some parts were more successful than others. The first goal was to create a secure and deployable web application. LBOG has been deployed and tested at lbog.cloudapp.net and the security analysis in Section 4.6 indicates that the most common risks have been eliminated or at least significantly mitigated. The second goal was to create a clear and usable user interface. This was also achieved, albeit with a larger time allocation than I would've preferred. Section 6.3 sheds some light on what could have been done differently. The third goal was to comply with Django and general software design patterns so that the project is easy to maintain and understand by others. Overall I believe this part of the development went very well. During the server side development process the there were hardly ever situations where another part of the application had to be restructured to make another part work which in itself is a good sign of separation of concerns.

6.2 Personal Aims Achieved

The personal aims were very much focused on learning more about building dynamic web applications. On the server side the programming language of choice was Python and its Django framework. In the middle of the development phase after a few weeks of learning Python and Django, I had barely gotten a some static pages and basic database manipulation working and was getting a little anxious about the outcome. Shortly thereafter however, for lack of a better expression, things just clicked and I started to realise the true potential of the framework. After that development proceeded at a rapid pace. Overall the learning experience was intense but very rewarding and this particular area can be considered one of the most significant personal achievements during the project. The next two aims were to learn more about formatting websites with HTML and CSS and improving the user experience with JavaScript. As with server-side programming, the learning curve was steep. In the end I gained significant amounts of knowledge on creating attractive user interfaces, but as mentioned in Section 6.1, the process was not without its problems. These are analysed in the next section. The final aim was to learn more about developing and deploying applications in a Linux environment using the command line. This was in hindsight a good decision as a Linux VM is a highly customisable deployment environment and knowing it will certainly be an asset in the future.

6.3 Critical Evaluation

This section aims to summarise the successes and failures of the project. Analysing the process with the benefit of hindsight reveals many areas that I will change in any future Django projects. Firstly, unit testing would be implemented from the beginning. In LBOG the testing strategy was reasonably successful, but if a future project were to be of a significantly larger scale, unit testing would be essential. On the server side, knowing how the Django components interact with each other, I would start developing the front end at a much later stage to have a clear idea of

exactly what information needs to be displayed before creating the templates. On the front end side I would research which CSS library (e.g. Twitter Bootstrap[2]) is the most suitable for the site in question and not customise it much. This would be a considerable timesaver whilst probably producing an adequate end result. I would also make greater use of AJAX to make the pages load quicker.

Perhaps the more interesting area of analysis is evaluating what could've been done differently with better planning. I feel that overall the project planning was a success and the technologies used were the right ones for the project. The one area I feel compelled to highlight as a minor failure is the planning of the user interface. Looking back I probably underestimated the magnitude of the workload required for creating a consistent design for a website with tens of different pages and on the other hand underestimated the difficulty of making things look right both on a mobile device and a desktop screen. This combination resulted in learning some things, as one might express it in my native language Finnish, "through the heel". This translates approximately to learning the hard way, only more painful. The front end in its current state is good enough to be considered a success, but the time spent on creating it kept me from working on other things such as a multilingual site. Looking at the requirements, this is the only "Should have" that was not achieved. One of the areas that could also have been improved is the social media integration, which at the moment only covers Facebook likes and shares. Custom Facebook activities such as sending a special thank you note or challenging a friend to work on a Project, or integration with other social media sites such as Twitter[1] or LinkedIn[2] might have been doable with an extra week or two.

[1] www.twitter.com
[2] www.linkedin.com

6.4 Future Work

Comparing LBOG to some of the competition such as the previously mentioned Catchafire[3], it quickly becomes clear that the site would need to be made a lot more professional to warrant serious consideration from developers and charities. This would include things like user support, making sure the legal side is covered etc. On the programming side the site would firstly need a more attractive front end. I believe that in today's market, anything less than a professional and attractive looking site will not be taken seriously. It could also connect to several different social media sites. At the back end the site could incorporate for example a blog and include some project management functionalities such as project-specific forums and so on. At this stage I believe that the code is structured well enough to serve as a basis for a site with broader functionalities. Realistiscally however LBOG will probably remain a student project, hopefully serving as a starting point for developing great Django applications in the future.

6.5 Final Thoughts

Many expected and unexpected challenges arose throughout the project, but the fact that conquering these remained interesting and fun rather than frustrating is in my opinion a sign of a well-chosen topic for a project. The successes and failures were assessed earlier in this chapter and both served as valuable lessons for the future. As the project meets most of the requirements and left me personally feeling significantly more confident in developing web applications in the future, it can be considered a success.

[3] www.catchafire.org

Bibliography

[1] Django Developers. Django Homepage, 2013. URL https://www.djangoproject.com/. Accessed 16/08/2013.

[2] Twitter Developers. Twitter Bootstrap, 2013. URL http://getbootstrap.com/. Accessed 16/08/2013.

[3] Harvest. Chosen Homepage, 2013. URL http://harvesthq.github.io/chosen/. Accessed 16/08/2013.

[4] jQuery Developers. jQuery Homepage, 2013. URL http://jquery.com/. Accessed 16/08/2013.

[5] Apache Software Foundation. Apache Homepage, 2013. URL http://www.apache.org/. Accessed 16/08/2013.

[6] Jim Arlow and Ila Neustadt. *UML 2 and the Unified Process*. Addison Wesley, 2005.

[7] Roger Pressman. *Software Engineering - A Practitioner's Approach*. McGraw-Hill, 2010.

[8] Jennifer Aaker and Wendy Liu. The Happiness of Giving: The Time-Ask Effect. *Journal of Consumer Research*, pages 543–557, October 2008. URL http://faculty-gsb.stanford.edu/aaker/pages/documents/Happinessofgiving.pdf.

[9] James Andreoni. Giving with Impure Altruism to Charity and Ricardian Equivalence. *Journal of Political Economy*, 97:1447–1458, December 1989. URL http://econ.ucsd.edu/~jandreon/Publications/JPE89.pdf.

[10] James Andreoni, William t. Harbaugh, and Lise Vesterlund. *Altruism in Experiments*. 2007. URL http://www.pitt.edu/~vester/Palgrave.pdf.

[11] Robert Cooter and Brian J. Broughman. Charity, Publicity, and the Donation Registry. *The Economist's Voice*, 97:1447–1458, December 1989. URL http://works.bepress.com/cgi/viewcontent.cgi?article=1108&context=robert_cooter.

[12] Adrian Holovaty and Jacob Kaplan-Moss. *The Django Book*. GNU Free Document License (Originally Apress), 2013. URL http://www.djangobook.com/en/2.0/index.html.

[13] Mark Pilgrim. *Dive Into Python 3*. Apress, 2009. URL http://cloud.github.com/downloads/diveintomark/diveintopython3/dive-into-python3.pdf.

[14] TheNewBoston (Bucky Roberts). TheNewBoston Python Tutorials, 2013. URL http://thenewboston.org/list.php?cat=36. Accessed 18/08/2013.

[15] Daniel Greenfeld and Audrey Roy. *Two Scoops of Django*. Cartwheel Web, 2013.

[16] Mike Hibbert. Mike hibbert Django Tutorials, 2013. URL http://www.youtube.com/playlist?list=PLxxA5z-8B2xk4szCgFmgonNcCboyNneMD. Accessed 18/08/2013.

[17] HackedExistence. Hackedexistence Django Tutorials, 2013. URL http://hackedexistence.com/project-django.html. Accessed 18/08/2013.

[18] GitHub user brack3t. django-braces, 2013. URL https://github.com/brack3t/django-braces. Accessed 17/08/2013.

[19] GitHub user django extensions. django-extensions, 2013. URL https://github.com/django-extensions/django-extensions. Accessed 17/08/2013.

[20] GitHub user un1t. django-cleanup, 2013. URL https://github.com/un1t/django-cleanup. Accessed 17/08/2013.

[21] Torchbox. South, 2013. URL http://south.aeracode.org/. Accessed 17/08/2013.

[22] Gregor Müllegger. Django Mobile, 2013. URL https://github.com/gregmuellegger/django-mobile. Accessed 29/08/2013.

[23] Jury Gerasimov. Mobileesp Python, 2013. URL https://code.google.com/p/mobileesp/. Accessed 29/08/2013.

[24] Michael Shepanski. Django Dbbackup, 2013. URL https://pypi.python.org/pypi/django-dbbackup. Accessed 29/08/2013.

[25] W3Schools. CSS3 Browser Support, 2013. URL http://www.w3schools.com/cssref/css3_browsersupport.asp. Accessed 18/08/2013.

[26] Ben Frain. *Responsive Web Design with HTML5 and CSS3*. Packt Publishing, 2012.

[27] Jon Duckett. *HTML & CSS - Design and Build Websites*. John Wiley & Sons, 2011.

[28] MapQuest. MapQuest API, 2013. URL http://developer.mapquest.com/. Accessed 28/08/2013.

[29] Git Developers. Git Homepage, 2013. URL http://git-scm.com/. Accessed 18/08/2013.

[30] PostgreSQL Developers. PostgreSQL Homepage, 2013. URL http://www.postgresql.org/. Accessed 18/08/2013.

[31] Visual Paradigm International. Visual Paradigm Homepage, 2013. URL http://www.visual-paradigm.com/. Accessed 21/08/2013.

[32] Gerald D. Everett and Raymond McLeod. *Software Testing - Testing Across the Entire Software Development Life Cycle*. John Wiley & Sons, 2007.

[33] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2001.

[34] Donald Norman. *The Design of Everyday Things*. Doubleday, 1988.

[35] Yvonne Rogers, Helen Sharp, and Jenny Preece. *Interaction Design - beyond human-computer interaction*. John Wiley & Sons, 2011.

[36] Thomas Connolly and Carolyn Begg. *Database Systems - A Practival Approach to Design, Implementation, and Management*. Addison-Wesley, 2005.

[37] PostgreSQL Developers. PostgreSQL Documentation - 5.6. Storing Binary Data, 2013. URL http://www.postgresql.org/docs/7.3/static/jdbc-binary-data.html. Accessed 21/08/2013.

[38] Chrome Developers. Chrome Developers - MVC, 2013. URL http://developer.chrome.com/apps/app_frameworks.html. Accessed 13/08/2013.

[39] Django Developers. Django URL Dispatcher, 2013. URL https://docs.djangoproject.com/en/dev/topics/http/urls/. Accessed 22/08/2013.

[40] Django Developers. Django generic editing views, 2013. URL https://docs.djangoproject.com/en/dev/ref/class-based-views/generic-editing/. Accessed 21/08/2013.

[41] Django Developers. Django-admin.py and manage.py, 2013. URL https://docs.djangoproject.com/en/dev/ref/django-admin/. Accessed 22/08/2013.

[42] Facebook Developers. Getting Started with OpenGraph, 2013. URL https://developers.facebook.com/docs/opengraph/getting-started/. Accessed 22/08/2013.

[43] Dafydd Stuttard and Marcus Pinto. *Web Application Hacker's Handbook : Finding and Exploiting Security Flaws*. John Wiley & Sons, 2011.

[44] Bryan Sullivan and Vincent Liu. *Web Application Security - A Beginner's Guide*. McGraw-Hill, 2011.

[45] Django Developers. Security in Django, 2013. URL https://docs.djangoproject.com/en/dev/topics/security/. Accessed 13/08/2013.

[46] National Institute of Standards and Technology (of the United States). Recommendation for Password-Based Key Derivation, 2013. URL http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf. Accessed 23/08/2013.

[47] Django Developers. Password Management in Django, 2013. URL https://docs.djangoproject.com/en/dev/topics/auth/passwords/. Accessed 23/08/2013.

[48] OWASP. OWASP - About Us, 2013. URL https://www.owasp.org/index.php/About_OWASP. Accessed 13/08/2013.

[49] OWASP. OWASP Top 10 - 2013, The Ten Most Critical Web Application Security Risks, 2013. URL http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf. Accessed 13/08/2013.

[50] Django Developers. Django Settings - Allowed Hosts, 2013. URL https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-ALLOWED_HOSTS. Accessed 13/08/2013.

[51] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *IEEE Computer Society*, pages 135–137, January 2001. available on IEEE Xplore through UCL Science Library.

[52] Lee Copeland. *A Practitioner's Guide to Software Test Design*. Artech House, 2004.

[53] Robert C. Martin. *The Clean Coder: A Code of Conduct for Professional Programmers*. Prentice Hall, 2011.

[54] Wikipedia. Multitier Structure Image, 2013. URL http://en.wikipedia.org/wiki/Multitier_architecture. Accessed 21/08/2013.

Appendix A

Analysis Diagrams and Figures

A.1 Front Page Screenshot

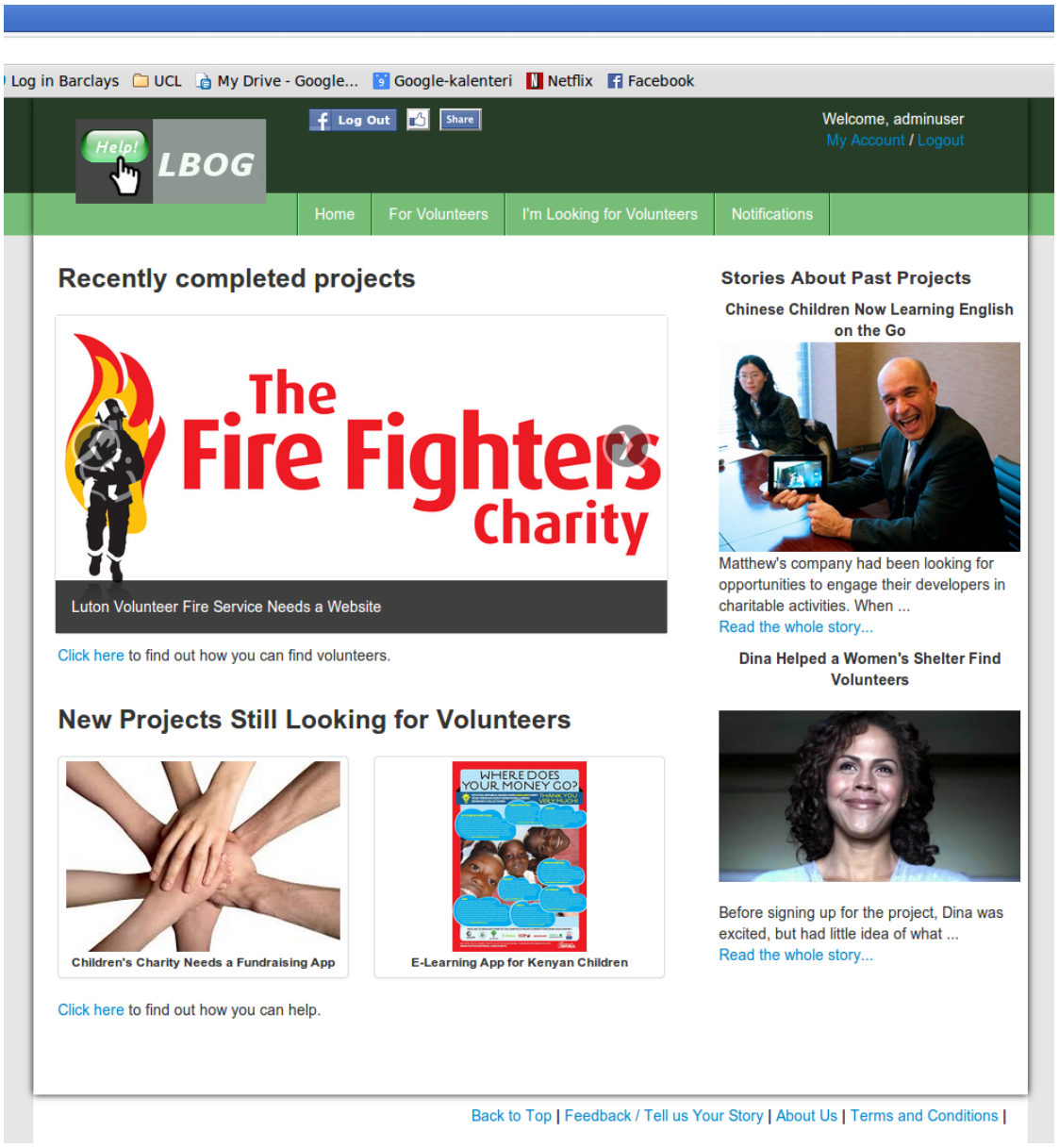

FIGURE A.1: Screenshot of the Front Page

A.2 Logical Entity Relationship Diagram

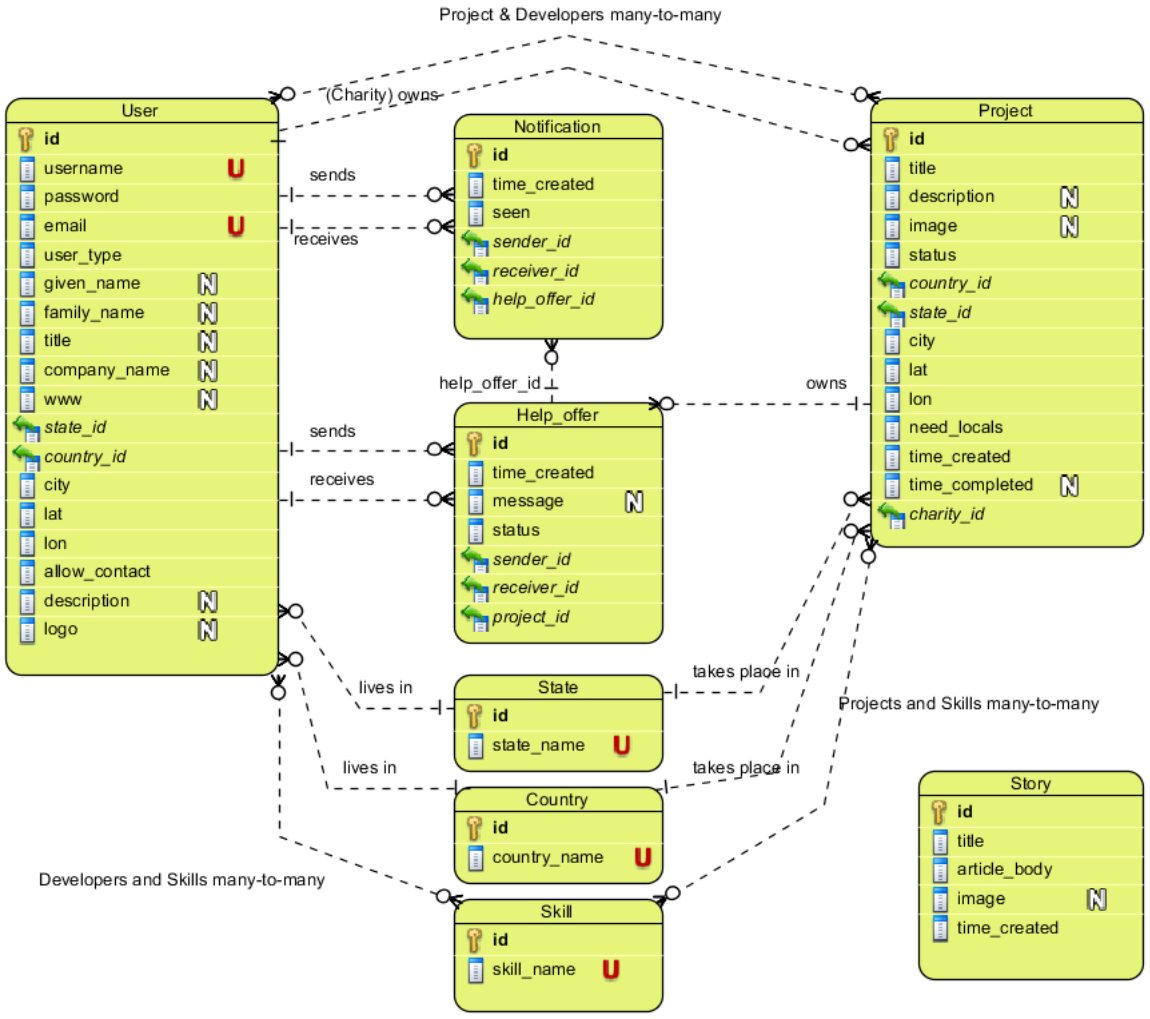

FIGURE A.2: Logical Entity Relationship Diagram

Visual Paradigm[31] Notation:

Capital U: unique

Capital N: nullable

Key Symbol: part of primary key

Green Arrow: foreign key field

Crow's feet notation used[36]

A.3 Detailed Use Cases

Case	RegisterDeveloperAccount
ID	UC1
Brief Description	Creates an Account for a Developer
Primary Actors	Developer
Secondary Actors	None
Preconditions	The user isn't logged in to the system
Main Flow	1. The use case starts when the user clicks on "Join" in the navigation bar. 2. The System displays the the username, password, and email fields and a user type choice. 3. The user selects a Developer account type. 4. The System displays the remaining fields. 5. The User fills in all mandatory and any optional fields and clicks on "Create Account". 6. The System retrieves the User's coordinates and sends a POST request with the form data to the server. 7. The System then displays a confirmation page for 1.5 seconds and redirects to login page.
Post Conditions	The System has added a new Developer account to the database, showed a confirmation message, and redirected the user to the login page.
Alternative Flows	InsufficientUserDetails, T&CNotAccepted

FIGURE A.3: Use Case 1

Case	RegisterCharityAccount
ID	UC2
Brief Description	Creates an Account for a Charity
Primary Actors	Charity
Secondary Actors	None
Preconditions	The user isn't logged in to the system
Main Flow	1. The use case starts when the user clicks on "Join" in the navigation bar. 2. The System displays the the username, password, and email fields and a user type choice. 3. The user selects a Charity account type. 4. The System displays the remaining fields. 5. The user fills in all mandatory and any optional fields and clicks on "Create Account". 6. The System retrieves the user's coordinates and sends a POST request with the form data to the server. 7. The System then displays a confirmation page for 1.5 seconds and redirects to login page.
Post Conditions	The System has added a new Charity account to the database, showed a confirmation message, and redirected the user to the login page.
Alternative Flows	InsufficientUserDetails, T&CNotAccepted

FIGURE A.4: Use Case 2

Case	UpgradeUserToAdmin
ID	UC3
Brief Description	Creates an Account with Admin rights
Primary Actors	Admin
Secondary Actors	None
Preconditions	The user is logged in as an Admin, The Admin-to-be has created a normal account first.
Main Flow	2. The System displays a list of existing users. 3. The Admin then looks for the username to be upgraded and clicks on it. 4. The System displays the account's details. 5. The Admin changes the account's status to "superuser" (Django terminology for Admin) and clicks on "Save". 6. The System saves the account's changed information and displays a confirmation message.
Post Conditions	The System has updated the future Admin's status to "superuser" in the database and showed a confirmation message.
Alternative Flows	InsufficientUserDetails, T&CNotAccepted

FIGURE A.5: Use Case 3

Case	InsufficientProjectDetails
ID	Alt4
Brief Description	Displays error messages when project data is missing
Primary Actors	Charity
Secondary Actors	None
Preconditions	None
Main Flow	1. The alternative flow starts when the user submits a project creation or project change form. 2. The system detects that the user hasn't entered all the necessary information and displays red error messages to help the user fill in the remaining data. 3. The user amends the data and submits the data again. 4. The system returns to normal flow.
Post Conditions	None
Alternative Flows	None

FIGURE A.6: Use Case 4

Case	Logout
ID	UC5
Brief Description	Logs the user out
Primary Actors	All
Secondary Actors	None
Preconditions	The user is logged in to the system
Main Flow	1. The use case starts when the user clicks on "Logout" in the navigation bar. 2. The system redirects the user to the login page instead of the welcome message and "My Account / Logout" shows "Login / Join" in the navigation bar.
Post Conditions	The System has removed the user's details from the session data (logged the user out) and displayed the "Login / Register" links.
Alternative Flows	None

FIGURE A.7: Use Case 5

Case	ChangeAccountDetails
ID	UC6
Brief Description	Updates user account details.
Primary Actors	All
Secondary Actors	None
Preconditions	The user is logged in to the system
Main Flow	1. The use case starts when the user clicks on "My Account" and "Account Details". 2. The System displays a form with the user's account details. 3. The user makes changes to the account details and clicks on "Save Changes". 4. The System retrieves the user's coordinates and sends a POST request with the form data to the server. 5. The System then displays a confirmation page for 1.5 seconds and redirects to My Account.
Post Conditions	The System has updated the user's details in the database and displayed a confirmation message.
Alternative Flows	InsufficientUserDetails

FIGURE A.8: Use Case 6

Case	DeactivateAccount
ID	UC7
Brief Description	Makes the user's account inactive
Primary Actors	All
Secondary Actors	None
Preconditions	The user is logged in to the system
Main Flow	1. The use case starts when the user clicks on "My Account". 2. The System displays a form with the user's account details and a navigation bar on the side. 3. The user clicks on “Deactivate Account”. 4. The System displays a confirmation dialog. 5. The user clicks on ok. 6. The System deactivates the account, logs the user out and displays the login screen.
Post Conditions	The System has changed the user's status to inactive.
Alternative Flows	None

FIGURE A.9: Use Case 7

Case	ChangePassword
ID	UC8
Brief Description	Updates the user's password.
Primary Actors	All
Secondary Actors	None
Preconditions	The user is logged in to the system
Main Flow	1. The use case starts when the user clicks on "My Account" and "Change Password". 2. The System displays a form with the an "Old Password", "New Password", and "Confirm Password" fields. 3. The user enters the required data and clicks on "Save Changes". 4. The System retrieves the user's coordinates and sends a POST request with the form data to the server. 5. The System then displays a confirmation page for 1.5 seconds and redirects to My Account.
Post Conditions	The System has updated the user's password in the database and displayed a confirmation message.
Alternative Flows	IncorrectPassword

FIGURE A.10: Use Case 8

Case	RecoverForgottenPassword
ID	UC9
Brief Description	Changes the user's password via an email link.
Primary Actors	All
Secondary Actors	None
Preconditions	The user isn't logged in to the system
Main Flow	1. The use case starts when the user clicks on "Login" and "Forgotten Password?". 2. The System sends out an email with a password recovery link to the user. 3. The user clicks on the link. 4. The System displays a form with "New Password" and "Confirm Password" fields. 5. The user enters the data and clicks on "Save Password". 6. The System sends a POST request with the form data to the server. 7. The System then displays a confirmation page for 1.5 seconds and redirects to Login page.
Post Conditions	The System has updated the user's password in the database and displayed a confirmation message.
Alternative Flows	None

FIGURE A.11: Use Case 9

Case	ContactSiteAdmin
ID	UC10
Brief Description	Sends an email to the site account via a contact form.
Primary Actors	Developer, Charity
Secondary Actors	None
Preconditions	None
Main Flow	1. The use case starts when the user clicks on "Contact Us" in the footer. 2. The System displays a contact form. 3. The user writes a message and clicks on "Send Message". 4. The System sends an email to the LBOG account and displays a thank you message.
Post Conditions	The System has sent an email to the site account.
Alternative Flows	None

FIGURE A.12: Use Case 10

Case	CreateProject
ID	UC11
Brief Description	Creates a new project.
Primary Actors	Charity
Secondary Actors	None
Preconditions	The user is logged in with a Charity Account
Main Flow	1. The use case starts when the user clicks on "I Need Help" on the navigation bar and on "Upload new Project" tab OR on "My Account > "My Projects" > "Create New Project". 2. The System displays a project creation form. 3. The user fills in the project data and clicks on "Upload Project". 4. The System retrieves the project location coordinates and sends a POST request with the form data to the server. 5. The System then displays a confirmation page for 1.5 seconds and redirects to the user's "Projects" page.
Post Conditions	The System has added a new Project to the database, displayed a confirmation message, and redirected the user to his/her "Projects" page.
Alternative Flows	InsufficientProjectDetails

FIGURE A.13: Use Case 11

Case	EditOwnProject
ID	UC12
Brief Description	Updates a Project's details.
Primary Actors	Charity
Secondary Actors	None
Preconditions	The user is logged in with a Charity Account
Main Flow	1. The use case starts when the user clicks on "My Account" > "My Projects" > "Project Link Here" > "Change Project Details". 2. The System displays a form with the Project's details. 3. The user makes changes to the account details and clicks on "Save Changes". 4. The System retrieves the Project location's coordinates and sends a POST request with the form data to the server. 5. The System then displays a confirmation page for 1.5 seconds and redirects the user to his/her "Projects" page.
Post Conditions	The System has updated the Project's details in the database and displayed a confirmation message.
Alternative Flows	InsufficientProjectDetails

FIGURE A.14: Use Case 12

Case	DeleteProject
ID	UC13
Brief Description	A Charity deletes a previously created Project.
Primary Actors	Charity
Secondary Actors	None
Preconditions	The user is logged in with a Charity Account
Main Flow	1. The use case starts when the user clicks on "My Account" > "My Projects" > "Delete Project". 2. The System displays a confirmation dialog. 3. The user clicks on "Yes". 4. The System deletes the Project from the database and redirects the user to his/her Project list.
Post Conditions	The System has removed the Project from the database.
Alternative Flows	None

FIGURE A.15: Use Case 13

Case	SearchProjects
ID	UC14
Brief Description	Shows a list of Projects corresponding to the user's search parameters.
Primary Actors	Developer
Secondary Actors	Charity
Preconditions	None
Main Flow	1. The use case starts when the user clicks on "For Volunteers" on the navigation bar. 2. The System displays a search form in the side bar. 3. The user fills in the search parameters and clicks on "Filter Results". 4. The System retrieves the search location coordinates and sends a POST request with the form data to the server. 5. The System then displays the search results or a message to confirm that none were found.
Post Conditions	The System has displayed a list of Projects corresponding to the search parameters.
Alternative Flows	None

FIGURE A.16: Use Case 14

Case	ViewProjectDetails
ID	UC15
Brief Description	The user views a Project's details.
Primary Actors	Developer
Secondary Actors	Charity
Preconditions	None
Main Flow	1. The use case starts when the user clicks on a project on the front page or in Project Search Results. 2. The System displays the Project's details and a form for sending an offer to help on the project.
Post Conditions	None
Alternative Flows	None

FIGURE A.17: Use Case 15

Case	ViewCharityDetails
ID	UC16
Brief Description	The user views a Charity's details.
Primary Actors	Developer
Secondary Actors	Charity
Preconditions	None
Main Flow	1. The use case starts when the user clicks on a project on the front page or in Project Search Results. 2. The System displays the Project's details and a list of links to all Projects in the sidebar. 3. The user clicks on the name or picture of the Charity. 4. The System displays the Charity's details.
Post Conditions	None
Alternative Flows	None

FIGURE A.18: Use Case 16

Case	SearchDevelopers
ID	UC17
Brief Description	Shows a list of Developers corresponding to the user's search parameters.
Primary Actors	Charity
Secondary Actors	Developer
Preconditions	None
Main Flow	1. The use case starts when the user clicks on "I Want to Help" on the navigation bar. 2. The System displays a search form in the side bar. 3. The user fills in the search parameters and clicks on "Filter Results". 4. The System retrieves the search location coordinates and sends a POST request with the form data to the server. 5. The System then displays the search results or a message to confirm that none were found.
Post Conditions	The System has displayed a list of Developers corresponding to the search parameters.
Alternative Flows	None

FIGURE A.19: Use Case 17

Case	ViewDeveloperDetails
ID	UC18
Brief Description	The user views a Developer's details.
Primary Actors	Charity
Secondary Actors	Developer
Preconditions	None
Main Flow	1. The use case starts when the user clicks on a Developer in Developer search results. 2. The System displays the Developer's details.
Post Conditions	None
Alternative Flows	None

FIGURE A.20: Use Case 18

Case	ContactCharity
ID	UC19
Brief Description	Sends an email to the Charity via a contact form.
Primary Actors	Developer
Secondary Actors	Charity
Preconditions	The user is logged in to the system
Main Flow	1. The use case starts when the user views the Charity's details (see UC16). 2. The System displays the Charity's details including the e-mail address. 3. The user clicks on the email address. 4. The computer displays the default new e-mail dialog. 5. The user writes the e-mail and clicks “send”. 6. The computer sends the message.
Post Conditions	The System has sent an email to the Charity's account.
Alternative Flows	None

FIGURE A.21: Use Case 19

Case	ContactDeveloper
ID	UC20
Brief Description	Sends an email to the Developer via a contact form.
Primary Actors	Charity
Secondary Actors	Developer
Preconditions	The user is logged in to the system
Main Flow	1. The use case starts when the user views the Developer's details (see UC18). 2. The System displays the Developer's details including the e-mail address. 3. The user clicks on the email address. 4. The computer displays the default new e-mail dialog. 5. The user writes the e-mail and clicks “send”. 6. The computer sends the message.
Post Conditions	The System has sent an email to the Developer's account.
Alternative Flows	None

FIGURE A.22: Use Case 20

Case	OfferToHelp
ID	UC21
Brief Description	A Developer sends a Help Offer to a Charity offering to help on a Project.
Primary Actors	Developer
Secondary Actors	None
Preconditions	The user is logged in with a Developer Account
Main Flow	1. The use case starts when a Developer is viewing a Project's details (see UC14), fills in the form and clicks on "Offer to Help". 2. The System creates a Help Offer with the status "pending", sends a Notification to the Project's owner, and displays a confirmation message.
Post Conditions	The System has added a Help Offer and a Notification of the Help Offer to the database
Alternative Flows	None

FIGURE A.23: Use Case 21

Case	ViewHelpOfferStatus
ID	UC22
Brief Description	Displays the details of a Help Offer
Primary Actors	Developer, Charity
Secondary Actors	None
Preconditions	The user is logged in to the system
Main Flow	1. The use case starts when the user views a Help Offer either by clicking on a Notification in the navigation bar dropdown menu or by clicking on "My Account" > "My Help Offers". 2. The System displays the details of the Help Offer and a link to the sender's profile.
Post Conditions	None
Alternative Flows	None

FIGURE A.24: Use Case 22

Case	DeleteHelpOffer
ID	UC23
Brief Description	Deletes a Help Offer from the database
Primary Actors	Developer, Charity
Secondary Actors	None
Preconditions	The user is logged in to the system
Main Flow	1. The use case starts when the user views a Help Offer either by clicking on a Notification in the navigation bar dropdown menu or by clicking on "My Account" > "My Help Offers". 2. The System displays the details of the Help Offer, including a Delete-button.. 3. The user clicks on Delete. 4. The system displays a confirmation dialog. 5. The user clicks “ok”.
Post Conditions	The Help Offer has been deleted from the System
Alternative Flows	None

FIGURE A.25: Use Case 23

Case	AcceptHelpOffer
ID	UC24
Brief Description	A Charity accepts a Developer's Help Offer to contribute to a project.
Primary Actors	Charity
Secondary Actors	None
Preconditions	The user is logged in with a Charity Account
Main Flow	1. The use case starts when the user views a Help Offer either by clicking on a Notification in the navigation bar dropdown menu or by clicking on "My Account" > "My Help Offers". 2. The System displays the details of the Help Offer and a link to the sender's profile. 3. The user clicks on "Accept". 4. The System: - Changes the Help Offer status to "accepted" in the database - Reminds the user that he/she may want to change the status of the Project that the Help Offer was attached attached to - Sends a Notification of the accepted Help Offer back to the sender - Adds the sender to the Developers of the Project
Post Conditions	added the Developer to the Project.
Alternative Flows	None

FIGURE A.26: Use Case 24

Case	RejectHelpOffer
ID	UC25
Brief Description	A Charity rejects a Developer's Help Offer to contribute to a project.
Primary Actors	Charity
Secondary Actors	None
Preconditions	The user is logged in with a Charity Account
Main Flow	1. The use case starts when the user views a Help Offer either by clicking on a Notification in the navigation bar dropdown menu or by clicking on "My Account" > "My Help Offers". 2. The System displays the details of the Help Offer and a link to the sender's profile. 3. The user clicks on "Reject". 4. The System changes the Help Offer status to "rejected" in the database, sends a Notification of this back to the sender.
Post Conditions	The System has changed the Help Offer status to "rejected" and sent a Notification to the Help Offer sender.
Alternative Flows	None

FIGURE A.27: Use Case 25

Case	CreateStory
ID	UC26
Brief Description	Creates a new Story.
Primary Actors	Admin
Secondary Actors	None
Preconditions	The user is logged in as an Admin
Main Flow	1. The use case starts when a user enters the Admin interface. 2. The System displays a list of all database item types. 3. The user clicks on "Stories". 4. The System displays a list of existing Stories and other options. 5. The user clicks on "Create New". 6. The System displays a form. 7. The user enters the data and clicks on "Save". 8. The System saves the Story in the database and redirects the user back to the Story list.
Post Conditions	The System has created a new Story in the database.
Alternative Flows	None

FIGURE A.28: Use Case 26

Case	ReadStory
ID	UC27
Brief Description	Displays the full Story.
Primary Actors	All
Secondary Actors	None
Preconditions	None
Main Flow	1. The use case starts when a user clicks on a story on the front page. 2. The System displays the full story and a sidebar with links to other stories.
Post Conditions	None
Alternative Flows	None

FIGURE A.29: Use Case 27

Case	DeleteStory
ID	UC28
Brief Description	Deletes a Story.
Primary Actors	Admin
Secondary Actors	None
Preconditions	The user is logged in as an Admin
Main Flow	1. The use case starts when a user enters the Admin interface. 2. The System displays a list of all database item types. 3. The user clicks on "Stories". 4. The System displays a list of existing Stories and other options. 5. The user ticks the checkbox next to the Story to be deleted, chooses action "Delete" from a dropdown and clicks on "Go". 6. The System deletes the Story from the database and confirms the deletion.
Post Conditions	The System has deleted a Story in the database.
Alternative Flows	None

FIGURE A.30: Use Case 28

Case	LikePageOnFacebook
ID	UC29
Brief Description	Posts a like on the user's Facebook wall.
Primary Actors	All
Secondary Actors	None
Preconditions	The user is logged in to Facebook
Main Flow	1. The use case starts when a user clicks on the like button at the top of the page. 2. The System connects with Facebook and posts the like on the user's wall.
Post Conditions	The user has liked the site on his/her Facebook wall
Alternative Flows	None

FIGURE A.31: Use Case 29

Case	SharePageOnFacebook
ID	UC30
Brief Description	Shares the site on the user's wall.
Primary Actors	All
Secondary Actors	None
Preconditions	The user is logged in to Facebook
Main Flow	1. The use case starts when a user clicks on the share button at the top of the page. 2. The System connects with Facebook and shares the site on the user's wall.
Post Conditions	The user has shared the site on his/her Facebook wall
Alternative Flows	None

FIGURE A.32: Use Case 30

Case	LikeProjectOnFacebook
ID	UC31
Brief Description	Posts a like on the user's Facebook wall.
Primary Actors	All
Secondary Actors	None
Preconditions	The user is logged in to Facebook
Main Flow	1. The use case starts when a user clicks on the like button at the top of the page. 2. The System connects with Facebook and posts the like on the user's wall.
Post Conditions	The user has liked the project on his/her Facebook wall
Alternative Flows	None

FIGURE A.33: Use Case 31

Case	ShareProjectOnFacebook
ID	UC32
Brief Description	Shares the Project on the user's wall.
Primary Actors	All
Secondary Actors	None
Preconditions	The user is logged in to Facebook
Main Flow	1. The use case starts when a user clicks on the share button at the top of the page. 2. The System connects with Facebook and shares the Project on the user's wall.
Post Conditions	The user has shared the project on his/her Facebook wall
Alternative Flows	None

FIGURE A.34: Use Case 32

Case	InsufficientUserDetails
ID	Alt1
Brief Description	Displays error messages when user data is missing
Primary Actors	All
Secondary Actors	None
Preconditions	None
Main Flow	1. The alternative flow starts when the user submits a profile creation or profile change form. 2. The system detects that the user hasn't entered all the necessary information and displays red error messages to help the user fill in the remaining data. 3. The user amends the data and submits the data again. 4. The system returns to normal flow.
Post Conditions	None
Alternative Flows	None

FIGURE A.35: Alternative Flow: InsufficientUserDetails

Case	T&CNotAccepted
ID	Alt2
Brief Description	Displays an error messages if the user hasn't appected Terms & Conditions
Primary Actors	All
Secondary Actors	None
Preconditions	None
Main Flow	1. The alternative flow starts when the user submits a profile creation form without accepting Terms & Conditions. 2. The system detects that the user hasn't ticked the box and displays a red error message to guide the user to do so. 3. The user ticks the box and submits the data again. 4. The system returns to normal flow.
Post Conditions	None
Alternative Flows	None

FIGURE A.36: Alternative Flow: T&CNotAccepted

Case	IncorrectPassword
ID	Alt3
Brief Description	Displays an error message if the user has entered an incorrect password.
Primary Actors	All
Secondary Actors	None
Preconditions	None
Main Flow	1. The alternative flow starts when the user submits a form requiring the user to enter their password. 2. The system detects that the password in incorrect and displays a red error message to confirm this. 3. The user enters the correct password and submits the data again. 4. The system returns to normal flow.
Post Conditions	None
Alternative Flows	None

FIGURE A.37: Alternative Flow: IncorrectPassword

Case	InsufficientProjectDetails
ID	Alt4
Brief Description	Displays error messages when project data is missing
Primary Actors	Charity
Secondary Actors	None
Preconditions	None
Main Flow	1. The alternative flow starts when the user submits a project creation or project change form. 2. The system detects that the user hasn't entered all the necessary information and displays red error messages to help the user fill in the remaining data. 3. The user amends the data and submits the data again. 4. The system returns to normal flow.
Post Conditions	None
Alternative Flows	None

FIGURE A.38: Alternative Flow: InsufficientProjectDetails

Appendix B

Design and Implementation Diagrams

B.1 Three-Tier Web Application Structure

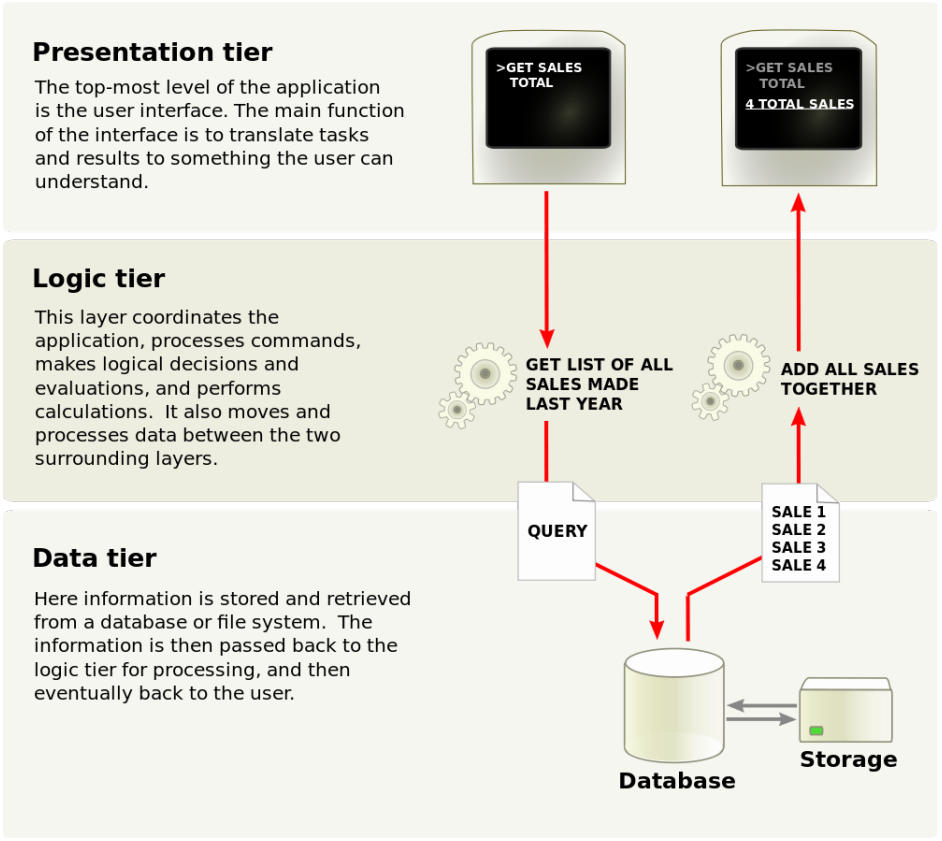

FIGURE B.1: Three-Tier Web Application Structure[54]

B.2 Template Design Diagram

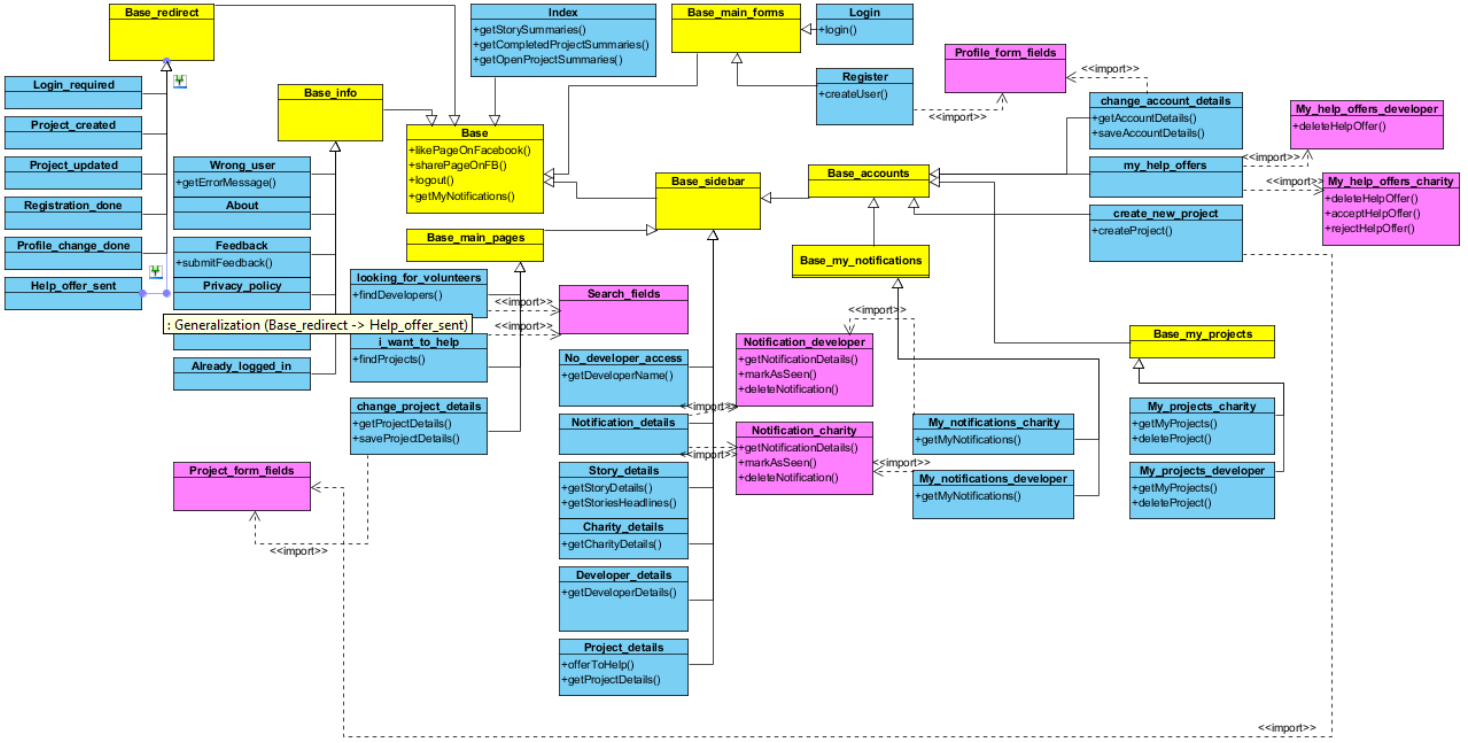

FIGURE B.2: Template Design Diagram

Color coding: Yellow = base templates (to be extended from), **Purple** = included in several templates (denoted by "import"), **Blue** = Template names called from views.

Notation: Subclassing Arrow = extends, **Import Relationship** = includes, **Method Calls** = Pseudocode, indicate which data is needed.

B.3 Admin Interface

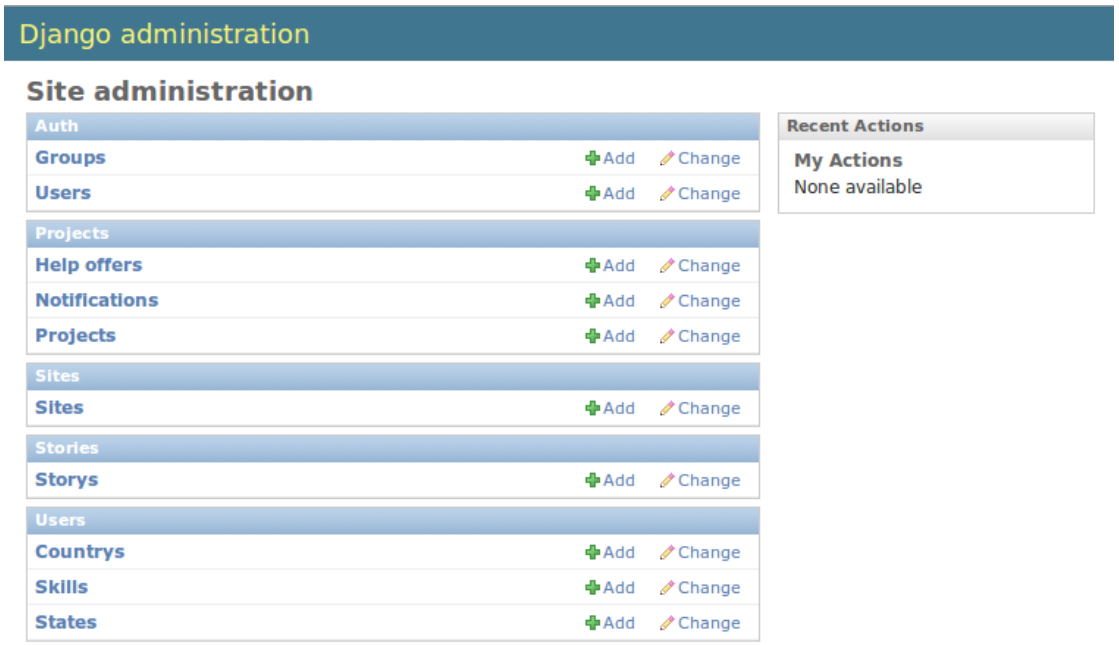

FIGURE B.3: Admin Interface Main Page

Appendix C

System Manual

C.1 Overview

This manual will provide instructions both for users, who simply wish to deploy the application on a Linux VM, and developers. The section starts by giving an overview of the Linux packages that the user needs to install, then covering the database setup, installing the application and finally setting up the server. A decent level of Linux proficiency is required from the reader.

C.2 Required Packages

The following list includes all the packages required to run the application. The name of the package is on the left and the terminal command for installing it on a Linux Ubuntu or Mint is on the right. The reader should install all the Linux packages prior to the Django packages and preferably in the order displayed in the table. The reader should also have Python 2.6 or 2.7 installed which is included by default in most current Linux distributions. Typing "python –version" in Linux terminal will display the currently installed version.

Linux Packages

Git	apt-get install git
Apache Server	apt-get install apache2
Mod_wsgi	apt-get install libapache2-mod-wsgi
Postgresql	apt-get install postgresql
Chkconfig	apt-get install chkconfig
Python Postgresql support	apt-get install python-psycopg
Python Imaging Library	apt-get install python-imaging
Pip (all necessary packages)	apt-get install python-pip python-dev build-essential

Django Packages

Django 1.5	pip install Django==1.5
South	pip install South
Django-extensions	pip install django-extensions
Django-braces	pip install django-braces
Django-cleanup	pip install django-cleanup
Django_mobile	pip install django_mobile
Django-dbbackup	pip install django-dbbackup

C.3 Database Setup

LBOG uses a Postgresql database to store the data. Before the application can be linked with the database, the reader will need to manually create it. The easiest way to do this is via the psql command line tool. To access psql the reader will need to change to an enabled Postgresql user. The default user name is "postgres" and by entering "su postgres" in the terminal, it becomes the active account. Psql is started by typing "psql" in the terminal. After that all the reader needs to do is enter is the SQL command "CREATE TABLE example_db;" and hit enter.

C.4 Installation and Settings Configuration

The application files can be installed by using Git or by copying the files from the enclosed DVD at the back of this report. For running a development version it is not important which directory the application will reside in, but the examples here and in the deployment server configuration section will assume that the application will be installed in the "/var/www/lbog/" directory. To use Git, all that needs to be done is to create the directory "/var/www/lbog/" and run "git clone git://github.com/mtpoutanen/lbog.git". However, at the moment the application resides in a private repository so the reader would need to contact the author (at mtpoutanen@gmail.com) to be added to the developers and get access. The other way of installing the application is simply to copy the "lbog" directory from the enclosed DVD to the /var/www/ directory.

The application includes some private settings such as the e-mail password and Facebook App ID that are not included in any of the publicly available versions. The distributed versions include a file called "example_local_settings.py" that can be found in the /lbog/lbog/ directory. This should be renamed to "local_settings.py" and edited to run the application. The file includes comments that will guide the reader on how to set up the parameters, icluding which ones are optional to run the application.

To create the database tables, the following commands must be entered in the terminal whilst in the application root directory (e.g. /var/www/lbog/):
python manage.py syncdb (if the intention is to use the mock data generator, select "No" when the system prompts for creating a superuser) python manage.py migrate users python manage.py migrate projects python manage.py migrate stories python manage.py migrate django_extensions

Lastly, to activate the static CSS and JavaScript files enter “python manage.py collectstatic”

C.5 Server Setup

Starting the development server is easy, all that is required is entering “python manage.py runserver” in the application root directory. LBOG should now be accessible from the IP address shown on the screen, in most cases 127.0.0.1:8000.

To configure the Apache server, the httpd.conf file will need to be modified to include the following lines:

```
ServerName localhost
WSGIScriptAlias / /var/www/lbgo/apache/django.wsgi
DocumentRoot /var/www/lbog

<Directory /var/www/lbog>
        Order allow,deny
        Allow from all
</Directory>

<Directory /var/www/lbog/apache>
        Order allow,deny
        Allow from all
</Directory>

Alias /static /var/www/lbog/static/
<Directory /var/www/lbog/static>
        Order allow,deny
        Allow from all
</Directory>

Alias /media /var/www/lbog/media/
<Directory /var/www/lbog/media>
        Order allow,deny
        Allow from all
</Directory>

<Directory /var/www/lbog>
        LimitRequestBody 10485760
</Directory>
```

You may also need to append the "hosts" file (typically in /etc/hosts) with the line "127.0.0.1 localhost". After configuring the server it needs to be restarted. At this

point if everything has gone according to plan, the application should be ready to use from the address of the virtual machine. When using an Apache server it is important to give the Apache user (usually "www-data") full permissions to the project directory

C.6 Mock Data Generator

To populate the database with mock data and create test accounts, a mock data generator was created. To access the application database via the Python shell, the reader will need to start by entering "python manage.py shell" in the application root directory. After that, the following three lines are needed:

```
>>> from filldb.py import DB_Filler
>>> dbf = DB_FIller()
>>> dbf.fill_all()
```

Finally, to include the pictures the reader will need to create a directory called "media" in the application root directory and copy the contents of the "media_mock_data" directory in to the "media" directory.

C.7 Facebook and Dropbox Setup (Optional)

To use the Facebook plugins and the Django-dbbackup application, the reader will need to add the Facebook app id and Dropbox app id and app secret to "<project_root>/lbog/local_settings.py". Lacking these will not however prevent the rest of LBOG from working. A Facebook app id can be acquired by setting up a Facebook app, which also requires an active Facebook account. Instructions on to set one up can be found on https://developers.facebook.com/docs/facebook-login/. Note that the like-button will only work on a public server due to Facebook's policies of keeping track of websites using certain functionalities. To set up a Dropbox

app and acquiring the necessary id and secret keys, please refer to detailed instructions here: http://pushingkarma.com/projects/django-dbbackup/%7B/projects/django-dbbackup/dropbox/.

Appendix D

User Manual

D.1 Introduction

Little Bits of Good has been designed as a publicly available website that guides the user through highly visible links, help messages, a simple user interface, and constraints on what can and can't be done and thus the user should intuitively be able to find what he/she wants. For these reasons this section will not provide a comprehensive user manual, but will instead go over a few of the most common use cases.

D.2 Registration

When the user first enters the site, there will be a "Join" link in the top right corner. By clicking this the site will load a registration form. After entering the username, password, and e-mail the user should pick a user type, which are either "Developer" or "Charity / Non-Profit". Each of these will show a slightly different form where mandatory fields are clearly marked. The user will then fill out the rest of the form and click on "Submit". The user may now log in on the site via the "Login" link in the top right corner.

D.3 Creating a Project

If the user created a Charity account and clicks on "My Account" in the top right corner, he/she will see a "Create Project" link in the sidebar. Clicking this will redirect to a form page. The mandatory fields are again clearly marked and the more information is entered, the better the chances of someone finding the project. Once the user has submitted the form, the Project will be saved in the database and can be edited from "My Projects".

D.4 Offering to Help

The first step in getting involved in a project as a Developer is to find a suitable project that is still looking for help. The front page displays some projects, but to find more specific results the user should use the search function on the "For Volunteers" page. Once a project has been found, the user may click on the project title to view the project details. If the user is logged in as a Developer, the page will display a message field and an "Offer to Help" button along with instructions. By filling this form and submitting it, the system generates a help offer and sends a notification of this to the Charity that uploaded the Project.

D.5 Accepting or Refusing a Help Offer

When a Charity user logs in after receiving Help Offers, he/she will see a bright red circle with the number of new notifications in the "Notifications" section in the navigation bar. The user can then go to "My Account" and view each Help Offer in "My Help Offers". There is a green "Accept" button and a red "Reject" button in each of the boxes containing a Help Offer. Should the user accept the offer, the Developer gets added to the Project and is notified of the accepted offer. In case of rejection, a notification is sent to the sender of the offer.

Appendix E

Code Examples

The following includes a selection of code examples that represent different functionalities. They are best read together with Chapter 4, which provides the code some context.

E.1 Models

E.1.1 Models in Projects application

```
class Project(models.Model):
    ''' The main Model class of the application '''

    STATUS_CHOICES = {  ('looking', 'Looking for developers'),
        ('under_way', 'Project under way'),
        ('completed', 'Project completed'),  }
    # model fields
    title           = models.CharField(max_length=50, blank=False, null=False)
    description     = models.TextField(blank=True, max_length=1000)
    # ... continues
```

```
    # Project model fields continue
    image           = models.ImageField(upload_to=get_image_path, blank=True,
                      null=True)
    skills          = models.ManyToManyField(Skill)
    status          = models.CharField(choices=STATUS_CHOICES, max_length=50,
                                       blank=False, null=False, default='looking')
    # Control at form level that only Developes can be added to Projects.
    developers      = models.ManyToManyField(UserProfile,
                            related_name='project_developers',
                            null=True, blank=True)
    charity         = models.ForeignKey(UserProfile,
                                related_name='project_charity',
                                null=False, blank=False)
    need_locals     = models.BooleanField()
    country         = models.ForeignKey(Country, blank=False,
                        null=False)
    state           = models.ForeignKey(State, blank=False,
                        null=False)
    city            = models.CharField(max_length=50, blank=False,
                        null=False)
    lat             = models.FloatField(blank=False, null=False, default=0.0)
    lon             = models.FloatField(blank=False, null=False, default=0.0)
    time_created    = models.DateTimeField(auto_now_add=True,
                        null=False, blank=False)
    time_completed  = models.DateTimeField(null=True, blank=True)

    # The __unicode__ method defines the default description for the model
    def __unicode__(self):
        return self.title
# Project class ends
```

```
class HelpOffer(models.Model):
    '''
    HelpOffers can only be sent by Developers. This is controlled
    at form validation and template rendering level
    '''

    HELP_OFFER_CHOICES = {
        ('pending', 'pending'),
        ('rejected', 'rejected'),
        ('accepted', 'accepted'),
    }

    sender          = models.ForeignKey(UserProfile, null=False, blank=False)
    message         = models.TextField(blank=True)
    project         = models.ForeignKey(Project, null=False, blank=False)
    time_created    = models.DateTimeField(auto_now_add=True,
                        null=False, blank=False)
    status          = models.CharField(max_length=10,
                      choices=HELP_OFFER_CHOICES,
                      null=False, blank=False, default='pending')

    def __unicode__(self):
        return      "Help offer for " + self.project.title +\
        " from " + self.sender.user.username
# HelpOffer class ends
```

```
class Notification(models.Model):
    ''' Notifications are generated automatically when
        a HelpOffer is sent or responded to '''

    help_offer      = models.ForeignKey(HelpOffer)
    sender          = models.ForeignKey(UserProfile,
                    null=False, blank=False, related_name="noti_sender")
    receiver        = models.ForeignKey(UserProfile,
                    null=False, blank=False, related_name="noti_receiver")
    seen            = models.BooleanField(null=False, blank=False,
                    default=False)
    time_created    = models.DateTimeField(auto_now_add=True,
                        null=False, blank=False)

    def __unicode__(self):
        return 'Notification for ' + self.help_offer.project.title
# Notification class ends
```

E.1.2 Simple example queries

```
# retrieve project with id of 3
project = Project.objects.get(pk=3)
# delete project with id of 3
Project.objects.get(pk=3).delete()
# get all projects that have the status 'pending'
pending_projects = Project.objects.filter(status='pending')
```

E.1.3 Example query with multiple search conditions

```
# create a list of conditions with Django's Q objects
argument_list = []
# field "description" contains "lorem ipsum" (case insensitive)
argument_list.append( Q(**{'description__icontains': 'lorem ipsum'} ))
# field "status" equals "looking"
argument_list.append( Q(**{'status': 'looking'} ))
# the field "country_name" of the country object equals "United Kingdom"
argument_list.append( Q(**{'country__country_name': 'United Kingdom'} ))
# this query returns the projects that meet all of the above conditions
# duplicates are removed by the .distinct() method.
Project.objects.filter(reduce(and_, argument_list)).distinct()
```

E.2 Templates

E.2.1 Base.html

```
{% load staticfiles %}
<html>
<head>
    {% include 'base_head.html' %}
    {% block extrahead_level1 %}  {% endblock extrahead_level1 %}

    {% block extrahead_level2 %}  {% endblock extrahead_level2 %}

    {% block extrahead_level3 %}  {% endblock extrahead_level3 %}
</head>
{# ...continues #}
{# (curly brackets and hashes denote Django template comments) #}
```

```
<body>
    {% include 'navbar.html' %}
    <div id="fb-app-id" rel="{{ FB_APP_ID }}"></div>
    <div id="site-address" rel='http://{{ SITE_ROOT }}'></div>
    <div id="image-address"
rel='http://{{ SITE_ROOT }}/static/global/images/lbog_logo_large.png'></div>

    {% include 'fbscript.html' %}
    <div class="container" id="box-hide-wrapper">
        <div class="container" id="box-hide"></div>
    </div>
    <!-- These extra divs are needed to create the shadow effect -->
    <div id="my-wrapper">
        <div class="container" id="body-wrapper">
            <div id="fb-root"></div>
            {% block content %}
{# all contents in subclassed templates must be in a block called content #}
            {% endblock content %}
        </div>
    </div>
    {% include 'footer.html' %}
<script src="{% static "global/js/form_listeners.js" %}"></script>

{% block extrascripts %}
{% endblock %}

</body>
</html>
```

E.2.2 Index.html

```
{% extends "base.html" %}
{% load staticfiles %}

{% block extrahead_level1 %}
{# include a CSS-file only used on the front page #}
        <link href="{% static "global/css/front_page.css" %}"
    rel="stylesheet" />
{% endblock extrahead_level1 %}

{% block content %}
        <div class="row">
                <div class="span8" id="front-page-main">
                {% include 'completed_projects.html' %}
                {% include 'new_projects.html' %}
                </div>

                <div class="span4" id="front_page_stories">
                        {% include 'stories.html' %}
                </div>
        </div>
{% endblock content %}
```

E.3 Views

E.3.1 Main URLconf File

```
# from lbog/urls.py, a clipped view of the patterns method call
# A part of the URL patterns function.
urlpatterns = patterns('',
    # ... url patterns clipped from the report
    # The following line includes all URLs from the "users" application
    url(r'^accounts/', include('users.urls')),
    # The following line includes all URLs from the "projects" application
    url(r'^projects/', include('projects.urls')),
    # The URL for the "about us" page
    url(r'^about/$', 'lbog.views.about', name='about'),
    # The URL for the "feedback" page
    url(r'^feedback/$', 'lbog.views.feedback', name='feedback'),
    # ... more url patterns clipped
)
```

E.3.2 Users Application URLs

```
# this contains the entire users/urls.py file.
# the class-based views are called with the
# ViewClass.as_view() method, whereas function-based
# views are called with the full function path.
from users.views import LoginView, RegSuccessView, RegistrationView, \
                ChangeView, SuccessView, CharityView, DeveloperView
from django.conf.urls import patterns, url
# ...continues on next page
```

```
# ...users/urls.py continues


urlpatterns = patterns('',
    url(r'^register/$', view=RegistrationView.as_view(), name='register'),
    url(r'^registration-successful/$',view=RegSuccessView.as_view(),
        name='registration-successful'),
    url(r'^login/$', view=LoginView.as_view(), name='login'),
    url(r'^my-account/(?P<pk>\d+)/$', view=ChangeView.as_view(),
        name='my-account'),
    url(r'^profile-changed/$', view=SuccessView.as_view(),
        name='profile-changed'),
    url(r'^password-change-done/$',
        'django.contrib.auth.views.password_change_done',
        name='password-change-done'),
    url(r'^password-change/$', 'django.contrib.auth.views.password_change',
        name='password-change'),
    url(r'^logout/$', 'django.contrib.auth.views.logout',
        {'next_page': '/accounts/login/'}, name='logout'),
    url(r'^charity-details/(?P<pk>\d+)/$', view=CharityView.as_view(),
        name='charity-details'),
    url(r'^developer-details/(?P<pk>\d+)/$', view=DeveloperView.as_view(),
        name='developer-details'),
    url(r'^deactivate-account/$', 'users.views.deactivate_account',
        name='deactivate-account'),
)
```

E.3.3 Users Application Views (Collapsed)

```
from braces.views import LoginRequiredMixin
from django.contrib.auth.forms import AuthenticationForm, PasswordChangeForm
from django.contrib.auth import login, get_user_model
from django.contrib.auth.models import User
from django.core.urlresolvers import reverse_lazy
from django.views.generic.edit import FormView, UpdateView
from django.views.generic import TemplateView
from users.models import UserProfile
from users.forms import MyCreationForm, MyChangeForm
from projects.models import Project, HelpOffer
from projects.views import CorrectUserMixin
from django.http import HttpResponse
from django.utils import simplejson

# A utility function, not a view
def update_profile(user, form):

# All class-based views
class LoginView(FormView):
class RegistrationView(FormView):
class PasswordChangeView(LoginRequiredMixin, CorrectUserMixin, FormView):
class ChangeView(LoginRequiredMixin, CorrectUserMixin, UpdateView):
class SuccessView(LoginRequiredMixin, TemplateView):
class RegSuccessView(TemplateView):
class CharityView(TemplateView):
class DeveloperView(TemplateView):

# function-based view
def deactivate_account(request):
```

E.3.4 Example Class-Based View

```
class ChangeView(LoginRequiredMixin, CorrectUserMixin, UpdateView):
    '''
    A view class for updating a user's account information.
    '''

    # Class-level variables
    model           = UserProfile
    # used by the LoginRequiredMixin
    login_url       = reverse_lazy('login')
    template_name   = 'change_account_details.html'
    form_class      = MyChangeForm
    success_url     = reverse_lazy('profile-changed')
    # Initialise error message for CorrectUserMixin, if a
    # user tries to access someone else's account details.
    error_message   = 'Oops, something went wrong. \
            The browser was trying to access someone else\'s profile.'

    def get_form(self, form_class):
        '''
        Override get_form to attaches the HTTP request
        to the form to access user information in form validation
        '''
        self.form_class         = MyChangeForm
        form = super(ChangeView, self).get_form(form_class)
        form.view_request       = self.request
        return form

# Continued on next page...
```

```
# ...continued from previous page
    def form_valid(self, form):
        '''
        This method executes when a POST request is received and
        saves the model data to the database if there are no errors
        '''
        user = self.request.user
        update_profile(user, form)
        return super(ChangeView, self).form_valid(form)


    def get_initial(self):
        '''
        Add the skills data to the form object,
        does not work by default.
        '''
        super(ChangeView, self).get_initial()
        # set url_id for CorrectUserMixin
        self.url_id = self.kwargs['pk']
        my_id       = self.kwargs['pk']
        my_user     = User.objects.get(pk=my_id)
        profile     = my_user.get_profile()
        skill_list  = profile.skills
        initial = {
            'skills':     list(skill_list.all()),
        }
        return initial
# Class ChangeView ends.
```

E.3.5 Example Function-Based View

```
def delete_notification(request, pk):
    ''' Deletes a notification '''

    # Get the notification to be deleted.
    notification = Notification.objects.get(id=pk)

    # Check if the URL is being called by the owner of that Notification
    if request.user.id != notification.receiver.id:
        # if not, return an error message
        result =    {  'error_message': 'Something went wrong, this'\
        +'notification belongs to user '+notification.receiver.user.username,
                    'div_id': '',  }
        return HttpResponse(simplejson.dumps(result),
                            mimetype='application/json')
    else:
        # if yes, delete the notification
        # and return a response with no error message.
        if request.method == 'POST':
            notification.delete()
            div_id = "#notification-" + str(pk)
            result = {  'error_message': 'no_errors',
                    'div_id': div_id,  }
            return HttpResponse(simplejson.dumps(result),
                                mimetype='application/json')
```

E.4 Geolocation JavaScript Function

```
$('#submit-button').click(function() {
    // Get the data
    var country         = $("#id_country option:selected").text();
    var state           = $("#id_state option:selected").text();
    var city            = $("#id_city").val();

    // See if the data was entered correctly. This function is implemented
    // in different ways for different purposes
    var errors          = getErrors();
    // Check if the user is searching within one country (returns false
    // when creating or altering profiles or projects)
    var countrySearch   = getCountrySearch();

    if (errors != "") {
        error_msg = "Please correct the following fields:\n\n" + errors;
        alert(error_msg);
    } else {
        // if there are no errors, concatenate the search parameters
        search_text = country;
        if (state != 'n/a (Outside of US or Canada)') {
         search_text = search_text + "," + state;
        }
        if (city != '') {
           search_text = search_text + "," + city;
        }
        // only applies to searches, see above
        if (countrySearch) {
            callback();
        } else {
        // ... continued on next page
```

```
// Geolocation continues where the code has passed error checks
            // standard case, the search text is formatted in to MapQuest API
            // friendly format and used as a parameter in an ajax call.
            var clean_text = search_text.split(" ").join("+");
            $.ajax({
                url: 'http://open.mapquestapi.com/nominatim/v1/search.php?q='
                     + clean_text + '&format=json',
                type: "GET",
                dataType: "json",
                success: function (data) {
                    // show error message, if no coordinates were retrieved
                    if ($.isEmptyObject(data)) {
                        alert('Could not retrieve coordinates for your city.\n'
                            +'Please check the spelling of your city \n'
                            +'(English names will work the best).');
                    } else {
                    // Coordinates retrieved successfully. Callback function
                    // has different implementations for different purposes.
                    var lat = data[0].lat;
                    var lon = data[0].lon;
                        $('#id_lat').val(lat);
                        $('#id_lon').val(lon);
                        callback();
                    }
                }
            });
        }
    }
});
```

E.5 Search

```
from users.models import UserProfile, Skill
from projects.models import Project
from django.db.models import Q
from operator import or_, and_
import math

class Search(object):
    '''A utility class that searches for Developers or Projects
    based on different parameters'''

    SEARCH_DEVELOPERS   = 'developers'
    SEARCH_PROJECTS     = 'projects'

    def __init__(self, request):
        '''
        All variables and some search parameters are identical
        for Project and Developer searches
        '''
        # initialise the variables
        self.country     = request.GET['country']
        self.qlat_str    = request.GET['lat']
        self.qlon_str    = request.GET['lon']
        self.radius      = request.GET['radius']
        self.skills      = request.GET.getlist('skills[]')
        self.keywords    = request.GET['keywords'].split(' ')
        # dummy values in case the user is not searching by location.
        self.qlat        = -1.0
        self.qlon        = -1.0
# __init__ continues...
```

```
# ... continued
        # set coordinates and radius
        if self.qlat_str != '':
            self.qlat = float(self.qlat_str)
        if self.qlon_str != '':
            self.qlon = float(self.qlon_str)
        if self.radius != 'same_country' and self.radius != 'no_radius':
            self.radius = int(self.radius)
        # initialise the argument list and append it with the
        # keyword and skills filters
        self.argument_list = []
        if self.keywords != ['']:
            for kw in self.keywords:
                self.argument_list.append(Q(**{'description__icontains': kw}))
        if self.skills:
            skill_ids = Skill.objects.filter(skill_name__in=self.skills).\
                        values_list('id', flat=True)
            self.argument_list.append( Q(**{'skills__id__in': skill_ids} ))


    def get_search_context(self, search_type):
        '''returns a context dictionary with the projects or developers'''
        context = {}
        # The user is filtering by country name only
        if self.qlat == -1.0 and self.radius == 'same_country':
            self.argument_list.append(
                        Q(**{'country__country_name': self.country} ))
        # The user is not filtering by geography
        elif self.qlat == -1.0:
            pass
# get_search_context continues...
```

```
# ... continued
        else:
        # The user searches by distance
            if search_type == self.SEARCH_PROJECTS:
# this part of the function constructs a list of ids of all projects that are
# within the radius. The get_distance function was taken from here:
# http://www.johndcook.com/python_longitude_latitude.html
# finally, the arguments are appended with the pk__in filter, which looks for
# the primary keys (id) in the adjacent projects list.
# The elif statement does the same for developers.
                distance_proj_ids = []
                all_projects = Project.objects.all()
                for proj in all_projects:
                    if self.get_distance(proj.lat, proj.lon, self.qlat,
                        self.qlon) <= self.radius:
                        distance_proj_ids.append(proj.id)
                self.argument_list.append( Q(**{'pk__in': distance_proj_ids} ))
            elif search_type == self.SEARCH_DEVELOPERS:
                distance_dev_ids = []
                all_developers = UserProfile.objects.all()
                for developer in all_developers:
                    if self.get_distance(developer.lat, developer.lon,
                        self.qlat, self.qlon) <= self.radius:
                        distance_dev_ids.append(developer.id)
                self.argument_list.append( Q(**{'pk__in': distance_dev_ids} ))
# get_search_context continues...
```

```
# ... continued
        # if there are still no arguments (i.e. user didn't enter any), pass
        # a boolean variable to the context
        if not self.argument_list:
            context['no_args'] = True
        # else, perform search
        else:
            if search_type == 'projects':
                # only find projects that have the status 'looking'
                self.argument_list.append( Q(**{'status': 'looking'} ))
                # pass a "projects" object to context (found projects)
                context['projects'] = Project.objects.filter(reduce(and_,
                                                self.argument_list)).distinct()
                if not context['projects']:
                    context['nothing_found'] = True
            elif search_type == 'developers':
                # similarly to above, only return Developers that have
                # explicitly allowed to be found in searches
                self.argument_list.append( Q(**{'allow_contact': True} ))
                self.argument_list.append( Q(**{'user_type': 'Developer'} ))
                context['developers'] = UserProfile.objects.filter(reduce(and_,
                                        self.argument_list)).distinct()
                if not context['developers']:
                    context['nothing_found'] = True
        # finally, return the Projects/Developers or that nothing was found
        return context
```

E.6 Data Validation

E.6.1 Client Side Controls

```
function getErrors() {
    var tempErrors      = "";
    var country         = $("#id_country option:selected").text();
    var state           = $("#id_state option:selected").text();
    var city            = $("#id_city").val();
    var radius          = $("#id_radius option:selected").text();
    var countryEmpty    = (country == "Country...");
    var stateEmpty      = (state == "State...");
    var cityEmpty       = (city == "");
    var radiusEmpty     = (radius == "Radius...");
    if (countryEmpty && stateEmpty && cityEmpty && radiusEmpty) {
        // do nothing, as the user is not filtering by geography
    } else if (country != 'Country...' && radius == 'Selected Country') {
        // do nothing, the user is searching within one country.
    } else { // check if all the data is in place
        if (country == "Country...") {
            tempErrors += "- Please select a country...\n";
        }
        if (state == "State..." && radius   != 'same_country') {
            tempErrors += "- Please select a state...\n";
        }
        if (city == "" && radius!= 'same_country') {
            tempErrors += "- Please enter a city...\n";
        }
        if (radius == "Radius...") {
            tempErrors += "- Please select a radius\n";
        }}
    return tempErrors;
}
```

E.6.2 Form Level Controls

```
class HelpOfferForm(forms.ModelForm):
    '''Sends a help offer.'''
    message = forms.CharField(max_length=500, widget=forms.Textarea,
                              required=False)

    class Meta:
        model   = HelpOffer
        fields  = ('message',)

    # These checks should be redundant as the form is not rendered
    # if the conditions aren't met
    def clean(self):
        # Check if the user is a Developer
        profile = self.view_request.user.get_profile()
        if profile.user_type    != 'Developer':
            raise forms.ValidationError('You must log in as a Developer\
                                         to offer your help on projects')
        project_id = self.view_request_pk
        project    = Project.objects.get(id=project_id)

        # Check if the user has already offered to help
        has_already_offered =   HelpOffer.objects.filter(\
                                    sender=profile, project=project)
        if has_already_offered:
            raise forms.ValidationError('You have already\
                                        offered to help on this project')
        return self.cleaned_data
```

E.7 Access Control

E.7.1 CorrectUserMixin Implementation

```
class CorrectUserMixin(object):
    #default error message
    error_message = 'You are logged in as the wrong user. \
                                No detailed error message was provided'
    url_id       = 0 # default value
    def render_to_response(self, context, **response_kwargs):
        """Returns either the desired template or wrong_user.html."""
        logged_in_id = self.request.user.id
        # if no url_id was set in the view, raise an error
        if self.url_id == 0:
            raise ImproperlyConfigured("url_id missing")
        # if the object belongs to the logged in user, return view template
        if logged_in_id == int(self.url_id):
            return self.response_class(  request = self.request,
                template = self.get_template_names(),
                context = context,
                **response_kwargs  )
        # otherwise, return wrong_user.html with a custom error message
        else:
            context['error_message'] = self.error_message
            return self.response_class(  request = self.request,
                template = 'wrong_user.html',
                context = context,
                **response_kwargs  )
```