SANTA CLARA UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

Date: November 27, 2018

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

Thomas King
Samantha Lee
Madison Martin

ENTITLED

Bug Reporting System

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE IN COMPUTER ENGINEERING

Thesis Advisor

Thesis Advisor

Department Chair

Department Chair

Bug Reporting System

by

Thomas King
Samantha Lee
Madison Martin

Submitted in partial fulfillment of the requirements
for the degree of
Bachelor of Science in Computer Engineering
School of Engineering
Santa Clara University

Santa Clara, California
November 27, 2018

Bug Reporting System

Thomas King
Samantha Lee
Madison Martin

Department of Computer Engineering
Santa Clara University
November 27, 2018

ABSTRACT

Currently, there aren't adequate bug detecting systems for Santa Clara University's IT department to efficiently track and eliminate bugs within the system. Our project's objective is to provide a solution to aid with the process of finding, reporting, testing, eliminating, and mitigating present and future bugs. Within the system there are four active user types: reporter, tester, developer, and manager. Each user is able to access specific webpages through their account they can log in and log out of. Reporters can report a bug and check on the status of the bugs they report. Testers and developers are able to check the status of all bugs, test bugs and solutions, assign bugs to other testers and developers, and update the status of bugs they're working on. Managers can also check the status of the bugs, assign bugs, and update the statuses of bugs. Reports, history, and updates on the bugs are inputted through the web page forms on our site and can be organized by priority or frequency. Information will be kept in a database using SQLite. The back-end of our site will run using Python, specifically the Flask framework. We will be using a client-server architecture with a server connected to a database to allow multiple clients to access the services at the same time.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Bugs happen. When bugs in code occur in regularly used software, the unexpected results can hinder productivity. In a school setting, if the online websites for students do not work as intended, a student could not register for their desired courses, get the study materials their professors put online, submit an assignment on time, or use other functions.

The SCU community should be able to report bugs in real time and have those issues follow a chain of command so they can be fixed by the proper people in a timely manner. When someone using SCU software encounters a bug, the only way to report it is to talk to the IT Department directly, which is too much effort for most users, so the bugs remain unreported. The only other time bugs are fixed is when they are found by the developers or testers. Providing an easy method for user feedback allows more bugs to be found earlier so they can be fixed before affecting too many people.

1.2 Solution

Our solution is to create a website that would enable people with an SCU log in to report bugs on SCU applications. The solution will consist of a structured flow of web pages allowing the testers, managers, and developers to share feedback and update the status of a given bug while adding notes on the specifics of the bug as well as the severity and the time to completion. There will also be a searchable history feature that will help with future bug mitigation by congregating all previous bugs with the steps taken to reproduce it and the solution which can facilitate a faster repair the next time a similar bug happens. The web pages will communicate with a server which will save information like bug information and status, email addresses for notifications, and account information for authentication, in a database. When a new session is launched the web page will retrieve relevant information from the server.

Chapter 2

Requirements

The functional and non-functional requirements, as well as the design constraints, are listed below in order to define the needs of the system. Critical requirements are those that are necessary, and recommended are additional features that could be added given enough time.

2.1 Functional

2.1.1 Critical

The system will allow a user to:

- Authenticate using their SCU account
- Report bugs into the system
- Check on status of previous bugs
- Assign bugs to qualified developers (if manager)
- View overall reports (if manager)
- Change the status of a bug (if tester or developer)

2.1.2 Recommended

It is a recommended functional requirement to allow:

- Reporters to get an email confirmation after submitting their bug
- Testers and managers to keep track of duplicate bugs
- Users to search through the history of bugs

2.2 Nonfunctional

2.2.1 Critical

It is necessary that the bug reporting system is:

- Visually appealing and not cluttered with too much information on any single page
- Easy to use for each of the roles
- Easy to expand with new features
- Updated frequently with the status changes of the bugs

2.2.2 Recommended

It is recommended that the system is:

- Useful to everyone with a SCU account
- Modular and easy to expand to other platforms
- Up to date with the most recent standards

2.3 Design Constraints

The system must:

- Work on the Linux machines in the Design Center
- Use authentication to differ between users

Chapter 3

Use Cases

This system has four different roles: reporter, tester, manager, and developer. A reporter is a user that reports a bug in the system. A tester is a person who reproduces reported bugs and tests bug fixes. A manager is the individual who assigns a bug to a developer. A developer is the person who fixes the bug and once the fix is completed, sends it back to the tester to make sure their solution is working.

3.1 Diagram

Figure 3.1 below demonstrates the overlap of actions each type of user can perform in the system.

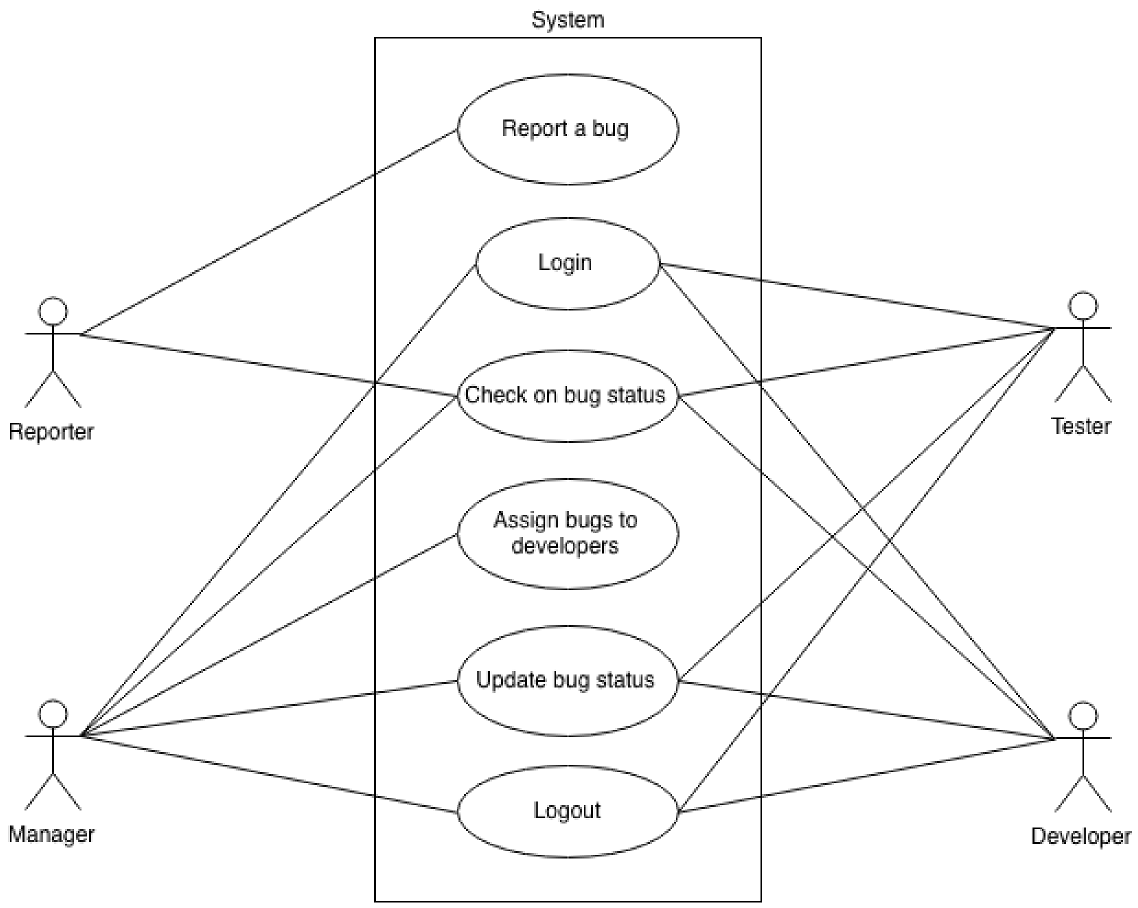


Figure 3.1: Use Cases Diagram

3.2 Cases

- **Log in**

  Goal: Log in using an SCU account to get access to the report page and the bug status page.

  Actors: Tester, Manager, Developer

  Preconditions: User must have an account on the database.

  Postconditions: User is notified if the log in was successful through either an error message or a redirect to their specific user page.

  Exceptions: None

- **Report a bug**

Goal: Fill out the report form and submit it so that the experienced bug will be fixed.

Actors: Reporter

Preconditions: Reporter must have logged in successfully.

Postconditions: Reporter is notified if the bug submission is successful.

Exceptions: None

- **Check on bug status**

  Goal: Find the desired bug and check on its status to see if it is fixed or not.

  Actors: Reporter, Tester, Manager, Developer

  Preconditions: User must have logged in successfully.

  Postconditions: Table displays the history of all bugs that have been reported and their status.

  Exceptions: None

- **Test bugs**

  Goal: Verify the state of the bug

  Actors: Tester, Developer

  Preconditions: Description of bug must be submitted through the reporter form.

  Postconditions: None

  Exceptions: None

- **Assign bugs to developers**

  Goal: Designate a bug to fix to a qualified developer.

  Actors: Manager

  Preconditions: Manager must have logged in successfully and there are bugs that need to be assigned.

  Postconditions: Developer is notified of a new bug to fix.

  Exceptions: None

- **Update bug status**

  Goal: Change the status of the bug depending on what part of the fixing process it is at.

  Actors: Tester, Manager, Developer

  Preconditions: Manager must have logged in successfully.

  Postconditions: Status value of the bug reflects the inputted value.

  Exceptions: None

- **Log out**

Goal: After done viewing the desired pages, the user will log out of their account.

Actors: Reporter, Tester, Manager, Developer

Preconditions: User must have logged in successfully.

Postconditions: User is notified that they have logged out of the system.

Exceptions: None

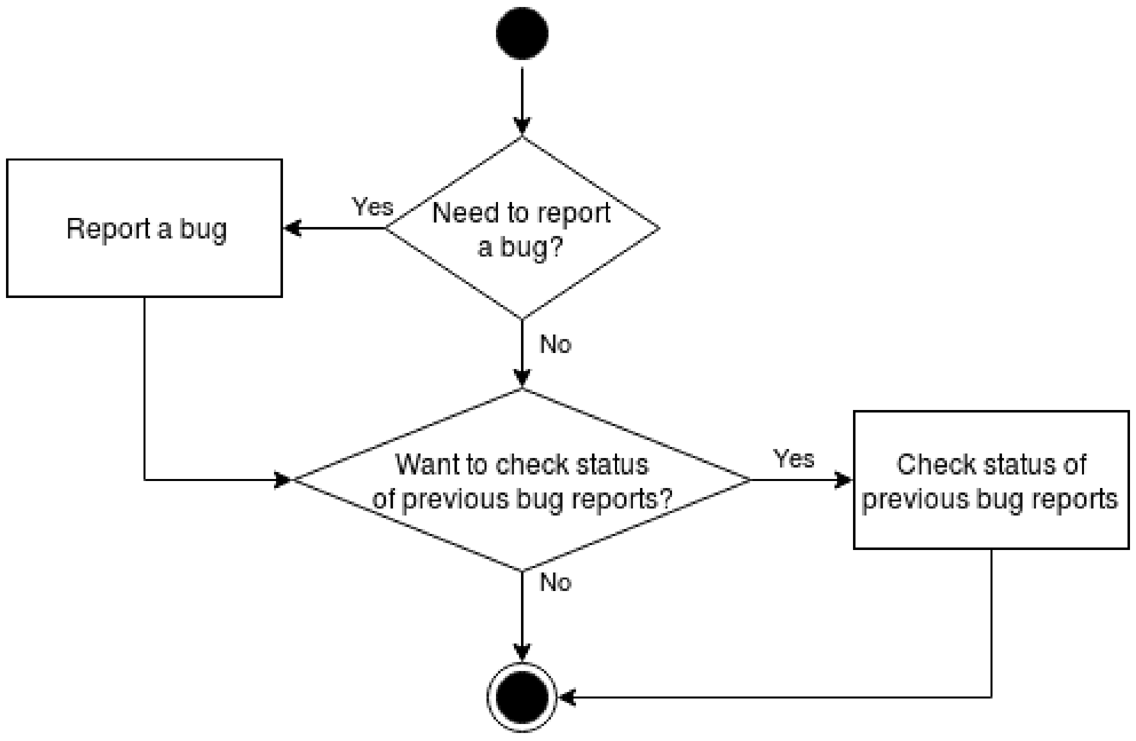
Chapter 4

Activity Diagrams

Figures 4.1-4.4 describe the activity flow for each user type: reporter, tester, developer, and manager.

4.1 Reporter

A reporter can report a bug and check the status of previous bug reports.



Report a bug
Yes
Need to report a bug?
No
Want to check status of previous bug reports?
Yes
Check status of previous bug reports
No

Figure 4.1: Reporter Activity Diagram

4.2 Tester

A tester can mark bugs as duplicates, reproduce bugs, and test bugs to confirm fixes. After logging in, the tester will utilize the system at to test the bug before and after it goes through the actual fixing, where the developer will deal with the issue.

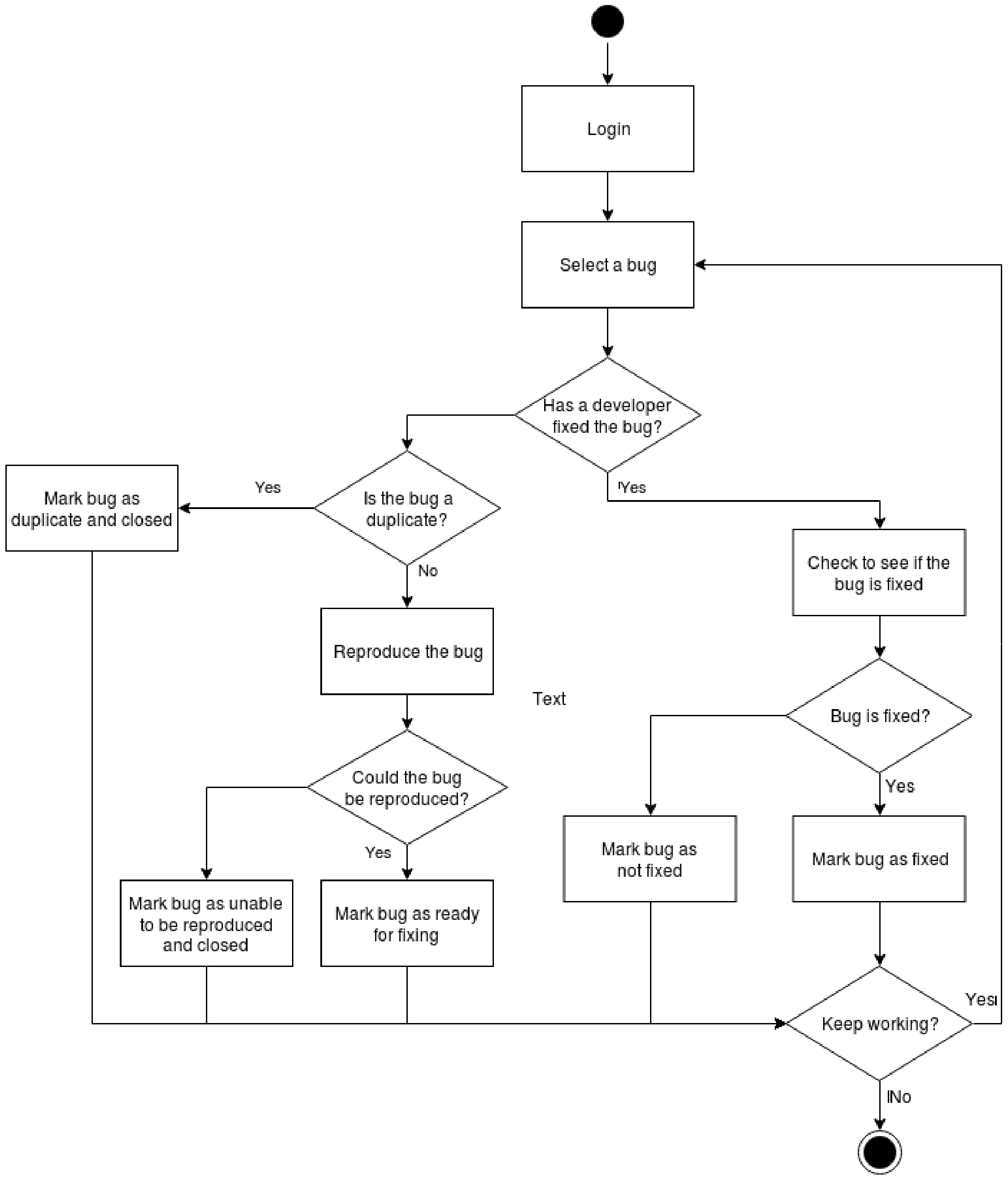

Login
Select a bug
Has a developer fixed the bug?
Yes
Is the bug a duplicate?
Yes
Mark bug as duplicate and closed
No
Reproduce the bug
Text
Could the bug be reproduced?
Yes
Mark bug as unable to be reproduced and closed
Mark bug as ready for fixing
Check to see if the bug is fixed
Bug is fixed?
Yes
Mark bug as not fixed
Mark bug as fixed
Keep working?
Yes
No

Figure 4.2: Tester Activity Diagram

4.3 Developer

A developer can fix bugs and mark them as ready for testing. After logging in, the screen should display the bugs assigned to them to fix, and when they are done with an assigned bug, then the status can be changed and it can be passed on to the tester for post-fix testing.

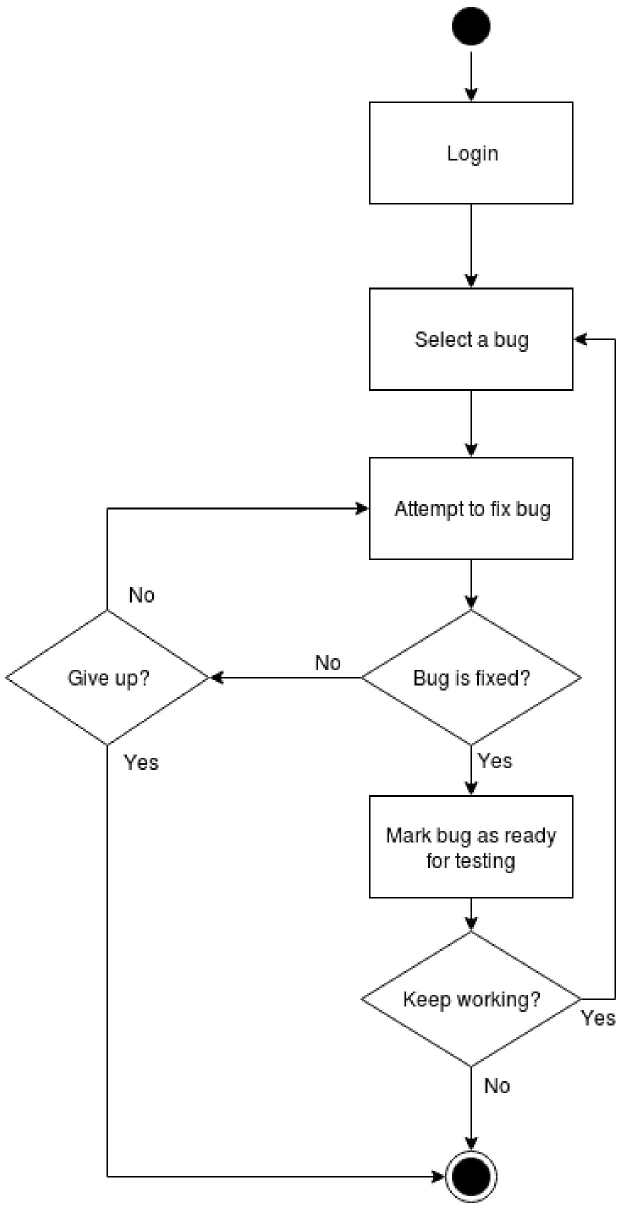
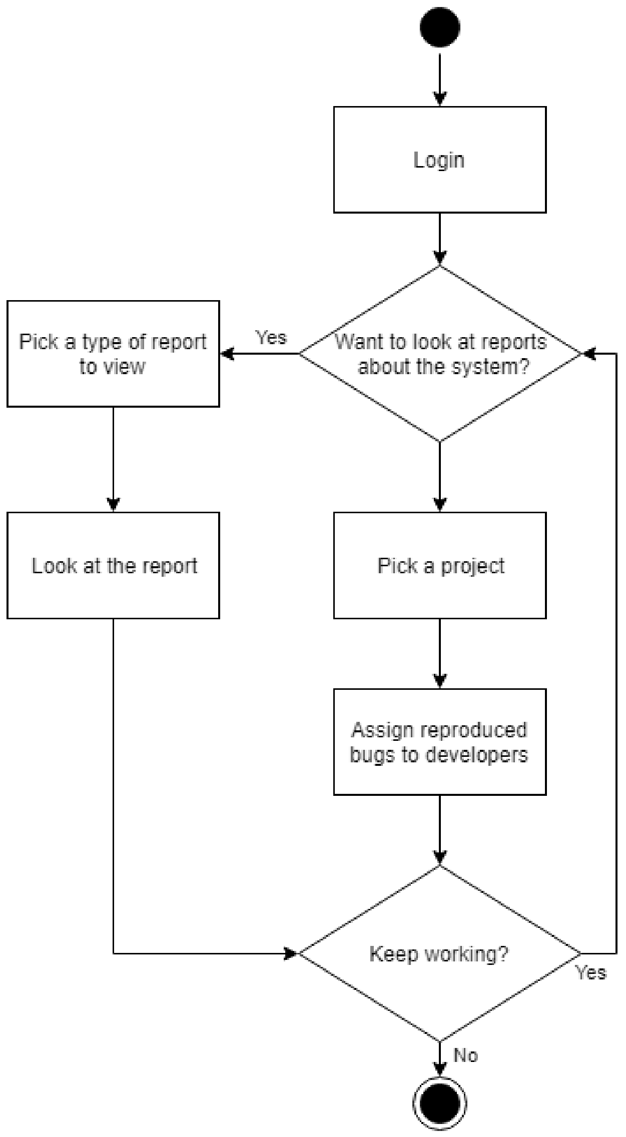


Figure 4.3: Developer Activity Diagram

4.4 Manager

A manager can look at reports about the system and assign reproduced bugs to developers. After logging in, the manager can change the status of the bug and who the bug is assigned to. They also are able to generate a bug history report from the bug history page.



Login
Want to look at reports about the system?
Yes
Pick a type of report to view
Look at the report
Pick a project
Assign reproduced bugs to developers
Keep working?
Yes
No

Figure 4.4: Manager Activity Diagram

Chapter 5

Technologies

This section details the types of technology we used to build the bug reporting web application and how we will use them in our system.

5.1 HTML/CSS/JS

We will use Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript (JS) for the front-end of the website. HTML describes the elements of the webpage using tags. CSS tells the client's web browser how to format those elements. JS is used for client-side scripting. The combination of these technologies lets us create an interactive website and customize it how we like. For example, to make a form, HTML would be used to describe the form and its fields, CSS would handle what it looks like, and JS would send data to the sever when the form is submitted.

5.2 Bootstrap

Bootstrap is a framework that integrates HTML, CSS, and JS used for web development that we will utilize to create user-friendly web pages.

5.3 Flask

Flask is a web microframework written in Python that we will use for the back-end of the web application, in other words, the software that the server will run. Flask will serve the clients the correct pages and handle their requests. All interaction with the database will be done through requests made to endpoints defined by Flask, which will call Python functions to handle them. It will also handle any other services like authentication.

5.4 SQLite

Our data will by stored in a SQLite database. SQLite is a lightweight relation database that uses stores the data in a file instead of a database server. Interaction with the database in done through standard SQL queries, run by Flask using Python's SQLite library.

5.5 GitHub

GitHub is a web-hosting service that utilizes Git for source control. We will use GitHub to ensure all of our code is stored in a safe place that all team members can access and work collaboratively through the use of branches.

Chapter 6

Architectural Diagram

Figure 6.1 describes the architectural relationship between clients, server, and database.

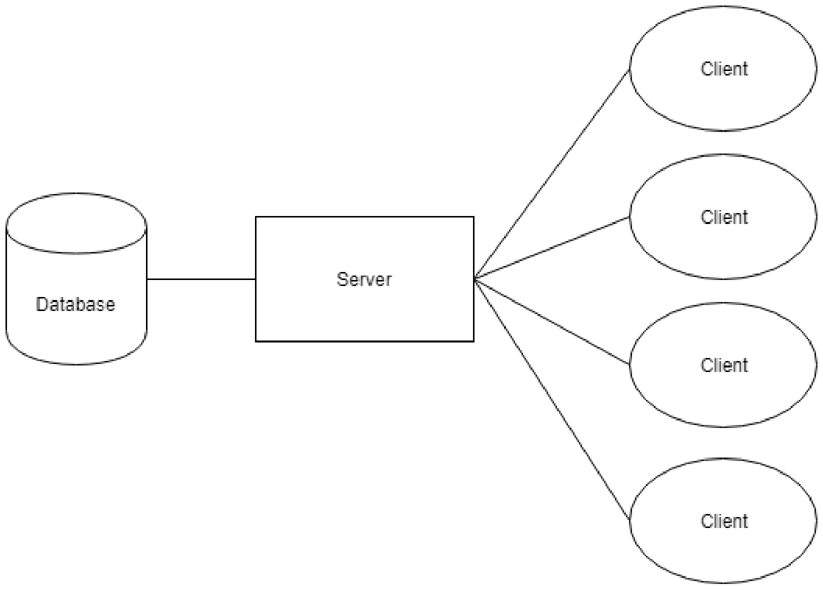


Figure 6.1: Architectural Diagram

Our solution will use a client-server architecture with a database. This architecture is typical for web development and allows for easy processing of data between multiple clients and the server. The clients' web browsers handle part of the processing like handling user input and formatting the web page based on information given to them from the server. The server acts as an interface for the database. Clients send the server requests and the server will perform the appropriate operations on the database. The server also handles services like authentication and provides the HTML, CSS, and JS for the web pages.

Chapter 7

Design Rationale

Below describes why we used the certain technologies that we did and how they pertain to our system.

7.1 User Interface

Our web application is broken up into several web pages. Having different pages makes it easy to restrict access to certain types of accounts and simplifies the interface into understandable parts. Clients have access to a bug report page and a bug history page. These pages are separate so it is easy for new users to use the basic functionality of bug reporting without getting overwhelmed with the history page. Developers, testers, and managers all have pages to manage bugs. Although there is a large amount of overlap between these pages some elements are different so we gave each of the actors their own page. For example, the developer only should be able to change the status of a bug, while a manager should be able to assign a bug to a developer and change the status. These variations of the page are displayed in the rightmost column. Under the hood we can use templates to avoid copying too much code. All of the pages with a table of bugs also has a pop-up modal with detailed info of a specific bug. We chose to make this a pop-up instead of a separate page to minimize navigating between pages and make it easy to quickly look at bug information.

7.2 HTML/CSS/JS

Using these technologies is practically mandatory in web applications since it is what the browser needs to create web pages. There are frameworks that can compile other types of code to these technologies but we chose not to use any of them because our website is fairly simple and we want to keep the amount of technologies we need to learn to a minimum to save on development time.

7.3 Bootstrap

Bootstrap will be used as a library to establish a consistent looking interface for the user. This will benefit the user experience with our web application.

7.4 Flask

We chose to use Flask for our back-end framework because it is simple. Flask is a microframework which means it only contains the most necessary features to run a web application. That means the codebase we will need to learn about will be smaller than other frameworks. Our application does not require many extra features so we can afford to avoid larger frameworks. The only features we need that is not provided by Flask is database interaction and authentication, but Flask can be extended by Python libraries that provide those features. As a programming language Python is easy to learn which will help us learn how to use the framework quickly, and Python is available on the lab computers.

7.5 SQLite

SQLite is easy to integrate with Flask because Python has built in support for it, as opposed to other SQL services like OracleDB whose Python libraries are difficult to install on the lab computers. Also, since there is no database server, we don't have to deal with additional setup or rely on Santa Clara University database servers.

7.6 GitHub

GitHub will allow members to create branches, enabling users to work on their own part of code without the risk of overriding important code implemented by another group member by accident. Using Git for version control will allow us to use access previous versions of code, just in case of incorrect implementation causing more errors in the code.

Chapter 8

Description of System Implemented

This chapter describes how our system is implemented.

8.1 Server

Our application uses the Python based Flask framework to run our server. The server hosts all of the web pages and handles data interaction with the SQLite database. It also handles requests made over the web and calls the appropriate Python functions to handle them, assuming the request is properly authenticated.

8.2 Authentication

Managers, testers, and developers need to be logged in to their account to access certain pages. Their passwords are hashed when they are stored in the database so that if the database is compromised their passwords are not.

8.3 Navigation Bar

Every page has the same navigation bar at the top with links to the home page, bug report form, bug history page, FAQs, contact page, and either the log in page or the log out page depending on whether of not the user has logged in yet. If they are logged in there is also a link to their user page with the same name as their role.

8.4 Bug Reporting Form

The bug reporting form (/report) contains a form with fields for the submitter's email, a brief description of the bug, and a more detailed description of the bug, as well as buttons to select which SCU software the bug was experienced on.

8.5 User Pages

Each role with a user account, manager, tester, and developer has a user page. It is the page visited after logging in or accessible in the navigation bar. The page has a list of bugs, managers see all bugs but testers and developers only see bugs assigned to them. The bug table has several features, including sorting by certain columns, pagination, and searching. Clicking on a bug shows a modal containing more detailed information. There are also certain functions that can be performed on bugs by selecting the checkboxes next to the bug that should be changed, selecting a new value in one of the dropdowns to the right of the table and clicking the appropriate button. Managers can change who a bug is assigned to and the status of a bug. Testers and developers can change the severity and status of a bug.

8.6 Bug History

The bug history page (/history) contains a table of all bugs with the same features as the table in the user pages, except there is no checkbox because bugs cannot be changed on this page. Managers can also click the print report button to get a printable version of the table.

8.7 Miscellaneous Pages

The home page (/), contains information about information services. The FAQs page (/faq), contains answers to frequently asked questions. The contact page (/contact) has contact information for SCU's technology help desk.

Chapter 9

Test Plan

Below is a description of the various plans we will use to test the different aspects of the system.

9.1 Validation Testing

For validation testing, we performed two types of tests: acceptance and usability testing. Acceptance testing was done when the finished system was submitted and sent to be reviewed in order to ensure that the finished product was up to the client's standards. Usability was tested by us and making sure that our system looked and functioned similar to SCU's current websites which have gone through outside usability testing.

9.2 Verification Testing

For verification testing, we have been performing three types of tests throughout the development process: unit, integration, and system testing. Doing these tests during the development life cycle allowed us to catch bugs in a timely manner which kept our system working when making improvements and adding new features. Once we executed unit tests of individual pages, we would test multiple pages together and make sure the links on each pages corresponded to the correct, expected behavior. When all the pages were done, we tested each use case end-to-end to make sure the user experience was as seamless as possible.

Chapter 10

Difficulties Encountered

Below is a description of the main problems we encountered while creating our system in the order of occurrence.

10.1 Oracle Database

Santa Clara University uses an Oracle database using MySQL to keep all of the data organized and together and we wanted to mirror their system as much as we could; however, the time it would have taken for us to learn how that system worked and to implement a similar system would have taken more time than we had allotted to get the database functional and would have set us off track from our development time line. So instead of a MySQL database, we chose to use SQLite because it was much easier to use with Flask as well as Python, two important technologies used for our system.

10.2 Repetitive Code

Since we wanted the pages to be uniform and aesthetically similar to sites SCU already uses, many of the pages had similar HTML code for the headers and tables. For that reason, when we made one change to those aspects of our site, we would have to make that change across multiple pages. For clarity and ease, we decided to modularize the headers and the tables for the pages so one change would resonate through all the pages and so the individual page code would be more relevant to what the page's function was, making it more readable.

10.3 SCU Google Authentication

In the same spirit of uniformity with current SCU websites, we wanted to be able to use SCU emails so everyone related to the university could log in and report a potential bug. When trying to gain access to the API that would allow for this utilization, the process to acquire the credentials needed was more time consuming than we had originally expected and we couldn't finish that task in the last couple of weeks we had to fully complete our system. We chose to create our own log in system that only handles managers, testers, and developers so they can gain access to the specialized pages relevant to their duties.

Chapter 11

Suggested Changes

This section details the suggested changes to be made to the system in future development.

11.1 Account Creation

Currently there is no interface to create a new account. There should be some type of admin role that could create accounts for users with a web page without relying on entering data into the database manually.

11.2 Password Changes

Users should be able to change their password on the website. This would also allow new accounts to have temporary passwords so that users can be given an account and they could change their password to a personal one that nobody else knows.

11.3 Bug Duplicates

Currently duplicate bugs are closed, but this results in the person who submitted the duplicate bug not knowing its status. Duplicate bugs should be able to be merged so that one entry is needed for the bugs but all users who encountered it can track its progress.

Chapter 12

Lessons Learned

This section discusses lessons we have learned throughout the development process from most specific to most general.

12.1 Flask

When starting this project, we wanted to use Flask to develop this web application, but since we had not been exposed to this technology throughout our education so far we had to learn the basics and how Flask could work with our project. This was done on our own from online resources by researching various frameworks and Through this research, Flask made more sense because we already had more knowledge about Python (of which Flask works well with) over other framework-language pairs.

12.2 Modularization

As detailed before, many of our web pages had repetitive code, so in order to remedy that we modularized the HTML pages. This was also a new concept to us which we learned about through our research regarding Flask as it is an included feature.

12.3 Git

We learned how to use Git to handle version control as well as how to make commits, pushes, and merges to Github from a local workspace with relative ease. This knowledge allowed all of us to work seamlessly on our system at the same time while not together which helped us produce a working product in a much shorter amount of time. This also made it easier to test out new or extra features without manipulating the working product.

12.4 Full Stack Development

We have had a growing appreciation for working in groups. Luckily, we had people who knew more about front end or back end development who could take control of certain aspects of the project as well as teach the others a little about what they were doing. We've learned that being a full stack engineer and having to complete a project like this on your own or having to know about all aspects of software development is a very difficult task.

Appendix A

Installation Guide

This appendix describes how to install and setup the web server

1. Download the application files and put them into a folder

2. Run the following commands in a terminal, in that folder.

```
setup python3
python3 -m venv venv
. venv/bin/activate
pip install flask flask-login
export FLASK_APP=app
flask init-db
```

Appendix B

User Manual

This appendix describes how to run and access the web server

B.1 Running the server

Run the following commands in a terminal in the folder where the server is installed

```
. venv/bin/activate
export FLASK_APP=app
flask run --host 0.0.0.0
```

B.2 Accessing the website

When the server is running on an SCU DC linux lab computer it can be accessed from any other lab computer using that computers hostname and the port 5000. Access the address linuxXXXXX.dc.engr.scu.edu:5000 with a browser to visit the website, replacing the X's with the lab computer's number. If you don't know the lab computer's number, type hostname into the terminal. To test authentication, the following accounts are available.

- Manager
  - Username: mmanager@scu.edu
  - Password: mp@ssword
- Tester
  - Username: ttesterson@scu.edu
  - Password: tp@ssword
- Developer 1

  - Username: dguy@scu.edu
  - Password: dp@ssword

- Developer 2
  - Username: dgal@scu.edu
  - Password: dp@ssword

B.3 Reporting a bug

To report a bug navigate to /report or click the bug report form button in the navigation bar. Fill out the form with the relevant information and click submit to submit your bug report.

B.4 Logging in

To log in click the "login" button in the navigation bar or navigate to /login. Enter your username and password and then click "login." You will redirected to your user page when successful.

B.5 Bug History Page

Users can see the history of all bugs by going to the bug history page, available on the navigation bar. There they can see a list of all bugs.

B.6 Interacting with the bug table

Any table containing a list of bugs has similar features. Clicking on the column headers sorts by that column, and clicking it more toggles between ascending and descending. If there are many bugs you can change pages in the bottom right. The dropdown menu to the top left of the table can be used to choose the number of bugs per page. Typing in the search bar in the top right filters the list of bugs. Clicking on a bug entry will pop up a modal with detailed bug information.

B.7 Changing a bug's severity, status, or assigned member

Any logged in user can change a bug's status on their user page by clicking the checkbox next to the bug and then selecting the desired status on the right, and clicking the change status button. Managers can also assign bugs to users in a similar way, and testers and developers can change a bug's severity. You can also change multiple bugs at the same time by checking multiple checkboxes.

B.8 Printable reports for managers

Managers can go to the bug history page and click the print report button on the top right to get a printable version of the bug table.