Team 4 – BugOverflow

Midterm Paper

Brian Curtis - Project Manager – p.1

Cindy Xie – UI/UX – p.9

Uriel Duran – Developer – p.24

Hong Zhang – Operations and Infrastructure – p.26

Sebrina Neal – Security – p.38

[Year]

**Project Management**

**Summary**

Our project is to create a question and answer website that is similar to the StackOverflow site. Our site, BugOverflow, is focused on insects, rather than coding errors.

**Database platform – MySQL**

MySQL is currently an industry standard. This means that the technology has been proven to work, there is plenty of support for it, and it is compatible with many systems. The main alternative to MySQL would be NoSQL. NoSQL has advantages over MySQL, mainly because its simpler and its table-less, meaning that its lighter. In fact, many large database websites such as Facebook and Google use NoSQL. For the time being as a new site, we have decided to go with what we know, and with what is a standard.

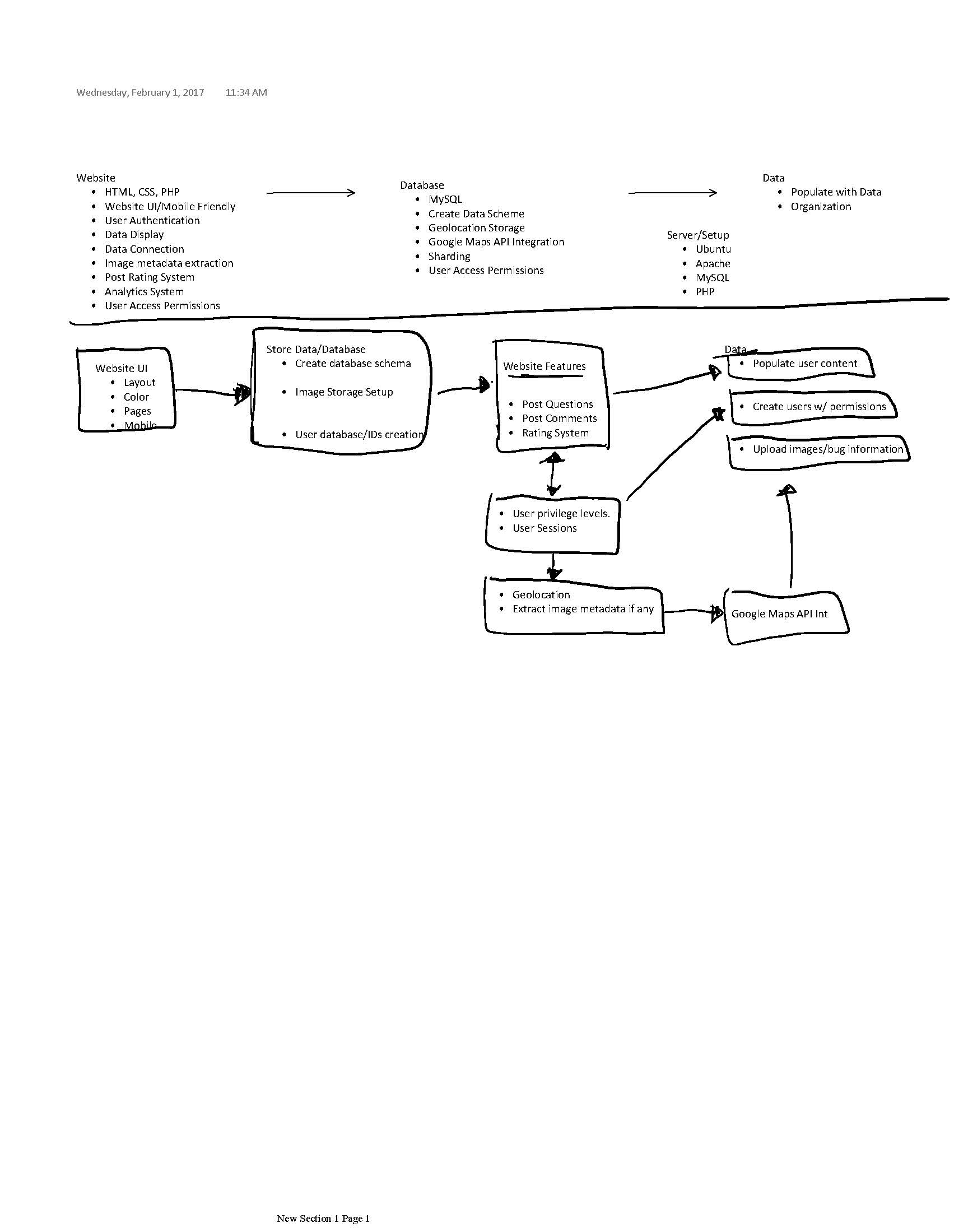
**OS and application stack – Windows 10, MySQL Workbench, Vagrant, Packer, Xamp, Apache**

Our team works primarily on Windows operating systems, so it made sense to develop on there. Our team’s database is developed and ran using MySQL workbench. The reason for using MySQL was explained above. Workbench is a free program that our development and infrastructure team was familiar with using. Vagrant and Packer are used to both copy and deploy exact copies of our virtualized servers so that each team member is working with the same setup, and they can deploy the site themselves. Xamp is used strictly for quick testing of server elements, such as our PHP code. Apache is used so that windows can properly view development PHP code. While we work on Windows machines, our actual servers run on a virtualized version of Ubuntu 14. Ubuntu is a lightweight and powerful version of Linux (Unix based) that can be easily virtualized and copied. Amazon Web Services (AWS) is planned to be used to store photo data, as well as host the servers.

**Programming languages/ frameworks – HTML5, CSS, PHP**

Our main code is PHP. PHP is a powerful language capable of interacting with SQL code, as well as managing HTML and CSS. Authentication, data, and site structure will all be managed with PHP. Along with PHP, most of the actual frontend page is still HTML5, and the design is CSS, since these are basic standards that work well.

V**alue Stream – below**



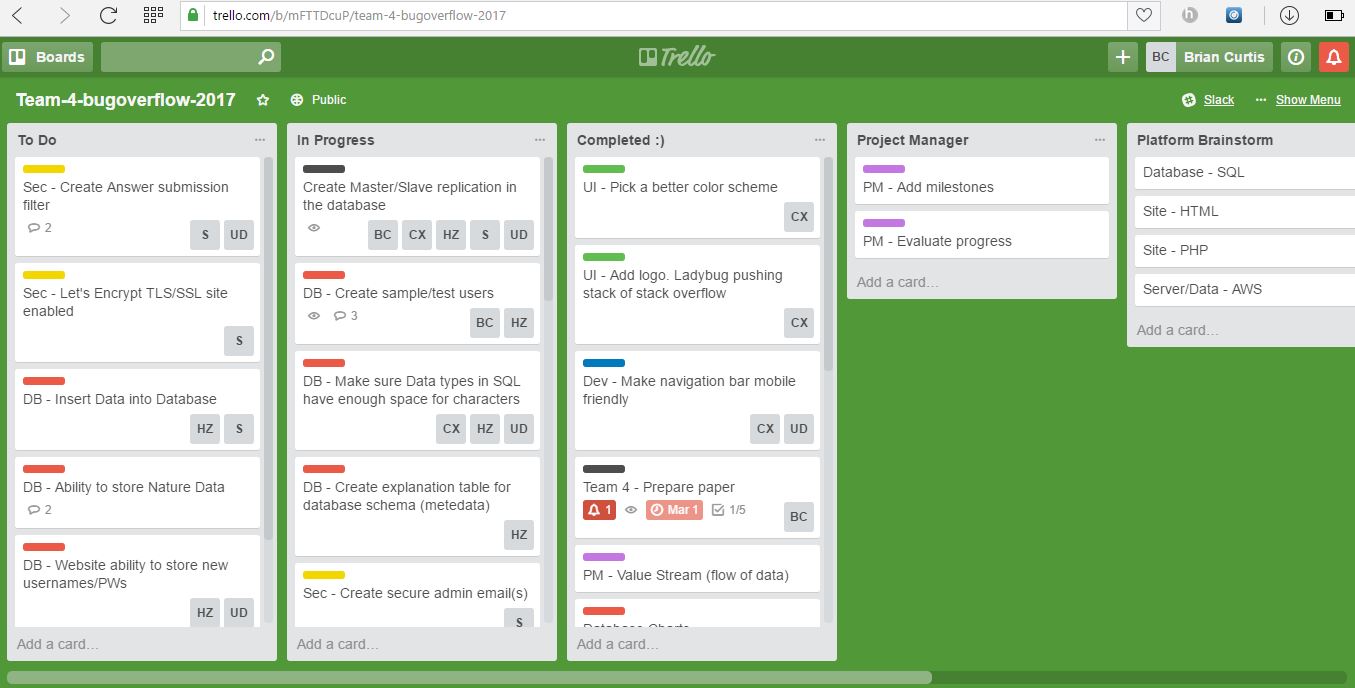
This diagram shows the basics of how we have planned the site to work. From the frontend UI on the left, we move towards the storage and retrieval of data. This data management is crucial to how we display posts from users. The site interacts with the database to grant permissions to varying levels of user privileges, and store user, post, and image information in the right places. Geolocation is a special feature that is mainly important for the user uploading of photos, which interacts with the Google Maps API. All of that comes around to the main storage of data, whether its creating users, creating content, or having the AWS store photos. Overall, this plan is to allow users to go through a line of security (Authentication), then let them upload photos with questions, answer questions, and they are able to retrieve that data.

**Cost of Servers – Azure ~$650; AWS ~$400**

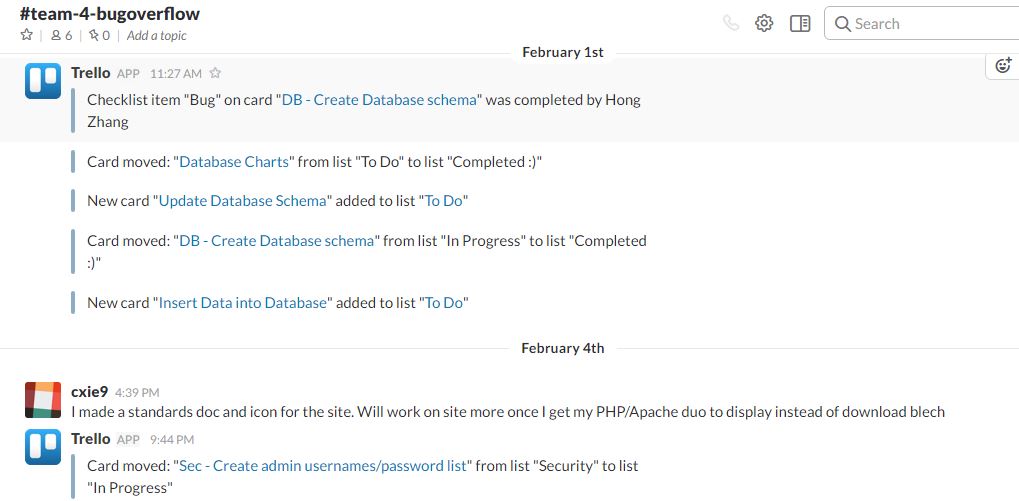
There are two main offsite server hosting options available to us: Microsoft Azure’s services, and Amazon Web Services. Both provide cloud based servers, and customer support. Using Azure’s services would cost an estimated $650 a month to use for 3 servers with business-level tech support within 8 hours. AWS would cost about $400 for 3 servers, 2 of which would be specialized as databases, along with business tech support within 1 hour. More research should be done to discover exactly which service would best suit our needs, but AWS has a better initial offer, as well as being a much larger cloud-based operation.

Project Communication and Tasking

The Bug Overflow team utilizes Trello for most of the project planning. We have a “To Do”, “In progress”, and “Completed” section to the immediate left for immediate visibility. Going right we have individual lists for members if they desire, as well as a list of resources we are using (such as development stacks) and a list of references (such as key security points to remember).



Slack is the official avenue used for team communication. Our slack is used to talk to each other, as well as where we send updates to the Trello board.



Github (<https://github.com/illinoistech-itm/team-4-bugoverflow>) is where our code is kept as a team. We have it semi organized, but more work needs to be done on keeping it tidy and having all of our build instructions placed on it.

**UI/UX**

**Testing and UI/UX**

This project started off with a simple idea of a forum where users could post questions and answers about bugs they found in their area. We started off with a simplistic barebones html site just to show content and layout and to get us off the ground (see Figure 1).

Users we tested shared feedback that the site navigation was clear, but the intent and color scheme was a little lacking, especially the first choice we settled on, a garish green (see Figure 2) that worked well on some screens but awfully on others. This was due to the fact that we just had “green” instead of a specific hex so that was taken into account in our next step. Users also mentioned our lack of mobile friendliness as during testing, most users pulled our site up on their phones or tablets. This made it clear that we needed a more mobile-friendly navigation bar and site overall.

Based on their feedback, we went back to the drawing board and created a wireframe (see Figure 3), using Fluid UI to experiment with complimentary color schemes and settled on #6F94D1, #D1AC6F, #6FD17B. We also developed an icon for site recognition/branding that incorporated the StackOverflow icon that our site is based off of and a starkly contrasted colored bug, the ladybug (see Figure 4).

Upon researching successful e-commerce site and platform, Etsy, we found that the simplicity of their stack would work well with what we intend to accomplish. Etsy is programmed via PHP on Apache probably due to website development versatility. They use MySQL, cache layers, and F5 load balancers to maintain data flow between sellers, buyers, and Etsy (Limpalair, 2016). Our current site is coded with HTML5/CSS and PHP to work seamlessly with our eventual SQL database integration. We also implemented a skeleton CSS framework that we found to work well aesthetically as well as in a mobile-first design by first applying Eric Meyer’s reset.css and then applying the skeleton CSS on top of this.

Our current home page has a few ways to go before looking like our wireframe site, but the basic layout remains intact (see Figure 5). We wanted the site to be simplistic, mobile-first, and intuitive to use to encourage user interaction/postings. Our home page will ultimately list popular “top” questions that are most commonly asked, or highest ratings according to customer rating as well as click-frequency. We chose the Google font Droid Sans due to its easy readability on big and small screens alike.

The sign-up page features common and interest-piquing questions to encourage membership (see Figure 6). While users can anonymously browse questions and answers on BugOverflow, in order to post or answer a question, they must have an account. Buttons for members or non-members not logged in or signed up will take users to the login page that will feature a message saying “Please sign in to post or reply”. The login page will also say “Not a member?” and have a button to take non-members to the sign-up page. The idea behind these choices is two-fold: (1) to encourage membership levels, but not to make the user think about joining as a chore, but as a natural and logical extension of visiting the site (2) during user testing, it was noted that there was a scramble for some testees before clicking on the sign up or login button (dependent upon their membership status) in the navigation bar, so naturally, these links had to be placed in more logical areas to account for different user logic.

Like the sign-up page, the login page is simplistic with only the minimal required fields to log in (see Figure 7). We want users to get back to using BugOverflow as fast as possible. A simple login page serves as a brief interruption to unlock a wider use of resources, not as a space to lure or more likely, annoy, users back with flashy images or graphics.

Our current question and answer page (see Figure 8), does not yet look similar to the wireframing image mentioned before at this point in the project so figure 9 UI choices will be discussed instead. As seen in figure 9, the question and answer page still features the simple fields common to the rest of the site, but with a few more necessary features including image upload and geolocation tracking. During user testing, there was a lot of back and forth concerning the issue of privacy in terms of automatic geolocation tracking. Some users found the geolocation feature to be convenient, saving them the effort of typing in and/or recalling where the bug was found. Others argued that they should be able to type in their location in the case that they moved from the original location the bug was spotted in and did not want a site knowing location without permission. To satisfy both parties, automatic geolocation was turned off and an optional location text field was added in the case that users did not want their location to be known. In the case that no locational information is submitted for the post, location data will be extracted from the photo upload itself. When users select submit, they are agreeing to the release of this information, therefore consenting to a privacy agreement. These errors and flaws we run into from user testing are key to our project’s success. As stated in *The Devops Handbook*, organizations need “a dynamic system of learning that allows us to understand our mistakes and translate that understanding into actions that prevent those mistakes from recurring in the future” (Kim et al., 2016). Testing and making mistakes is vital to preventing future mistakes, ideally before live deployment.

To create valid test data for usage testing, we intend to implement the javascript library of [faker.js](https://github.com/marak/Faker.js/) to generate fake names and accounts. Part of the syntax consists of the following:

**var user = {  
 name: faker.name.findName(),  
 email: faker.internet.email(),  
 bio: faker.lorem.sentence(),  
 image: faker.image.avatar(),**

**faker.helpers.createcard();  
}**

The helpers’ library will create full contact cards including name, email, bios, images, and blog posts to reflect how the site will be engaged once more tables are populated. We currently have about 10 accounts generated from our security team to get testing off the ground for UI/UX choices, but plan to implement at least 40 more. The data we have in our database will be real and valid since we don’t want to corrupt our original data. With users’ ability to add data into our forums, the chance for corruption will already exist so to start user testing we have only factually accurate bug information. As Gene Kim says in his keynote speech (Kim, 2014), the accrual of technical debt leads to fragile artifacts. To avoid the downward spiral of our site, keeping data uncorrupted and organized is key. Eventually our site will need monitoring in the case that inappropriate or inaccurate content is posted. This is the only way to ensure that the most critical, possibly revenue-generating operations of BugOverflow remain less than fragile. Avoiding compounding technical debt by learning from and preventing future problems improves our site and consumers’ experiences and success.

Moving forward, ideally, automated testing would be helpful for site improvement be that in terms of UX, database, or general programming deployments. *The Devops Handbook* supplies a model (see figure 10) that works successfully for Google that we will consider adopting when it comes to test builds in BugOverflow. They have their commit stage to automated test stage automatically approved while exploratory tests to production deploy are manually approved (Kim et al., 2016). Though our test times won’t be nearly as fast, it is still a strong and proven successful model for strive for.

**Figures**

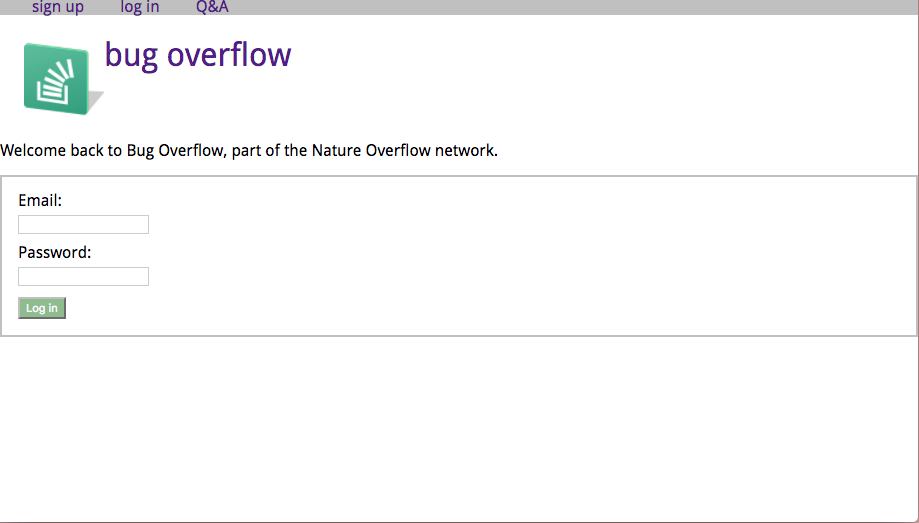
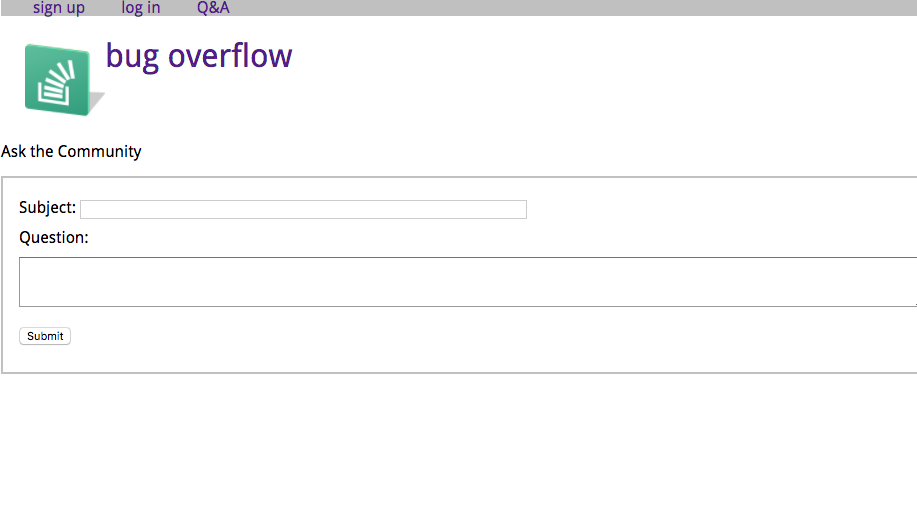
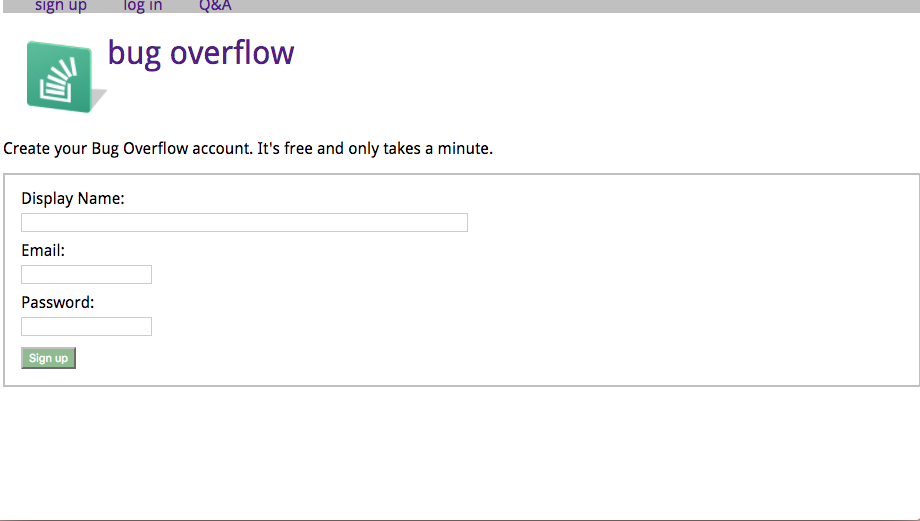
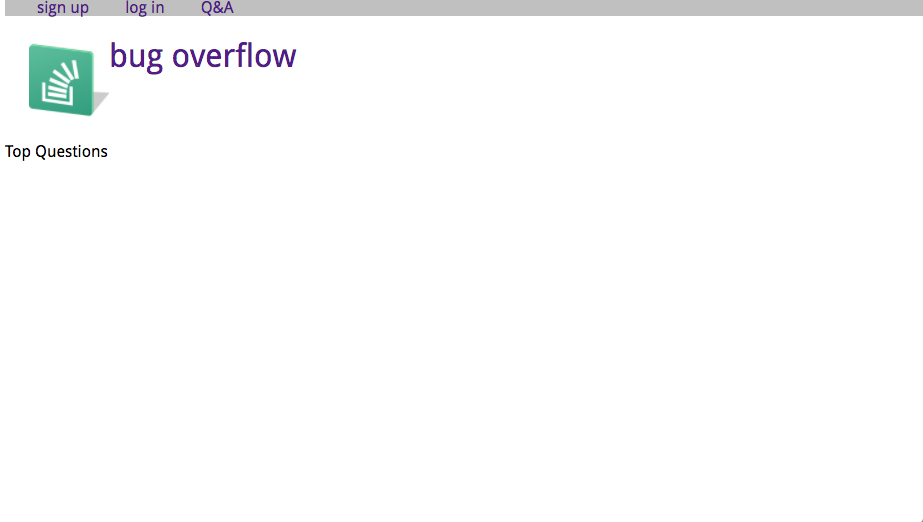


Figure 1, Basic HTML of BugOverflow site

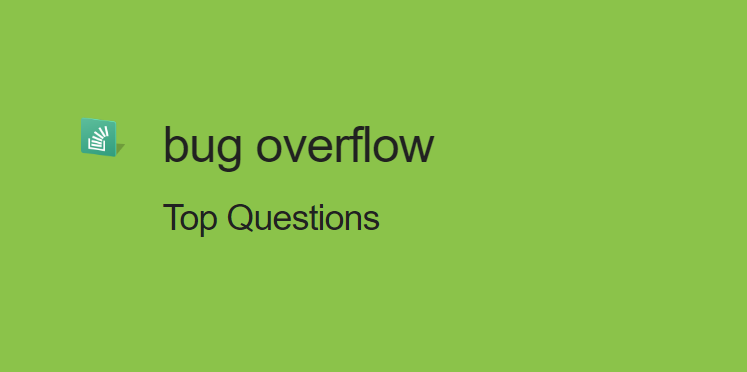


Figure 2, Implementing color/CSS to basic site

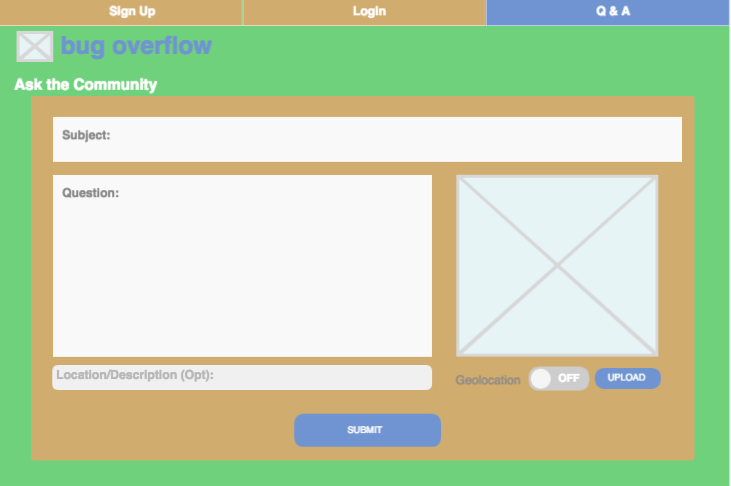
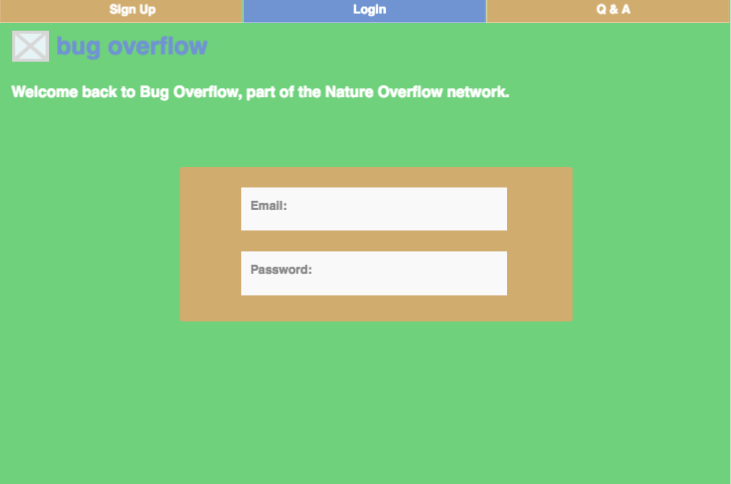
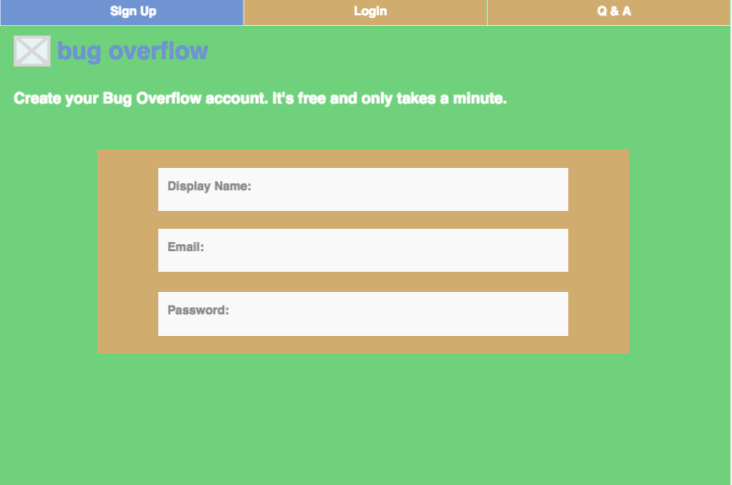


Figure 3, Wireframing and final color/design choice for BugOverflow site

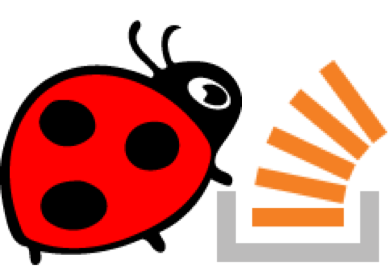


Figure 4, BugOverflow icon integrating memorable bug pushing an overflowing stack taken from StackOverflow icon

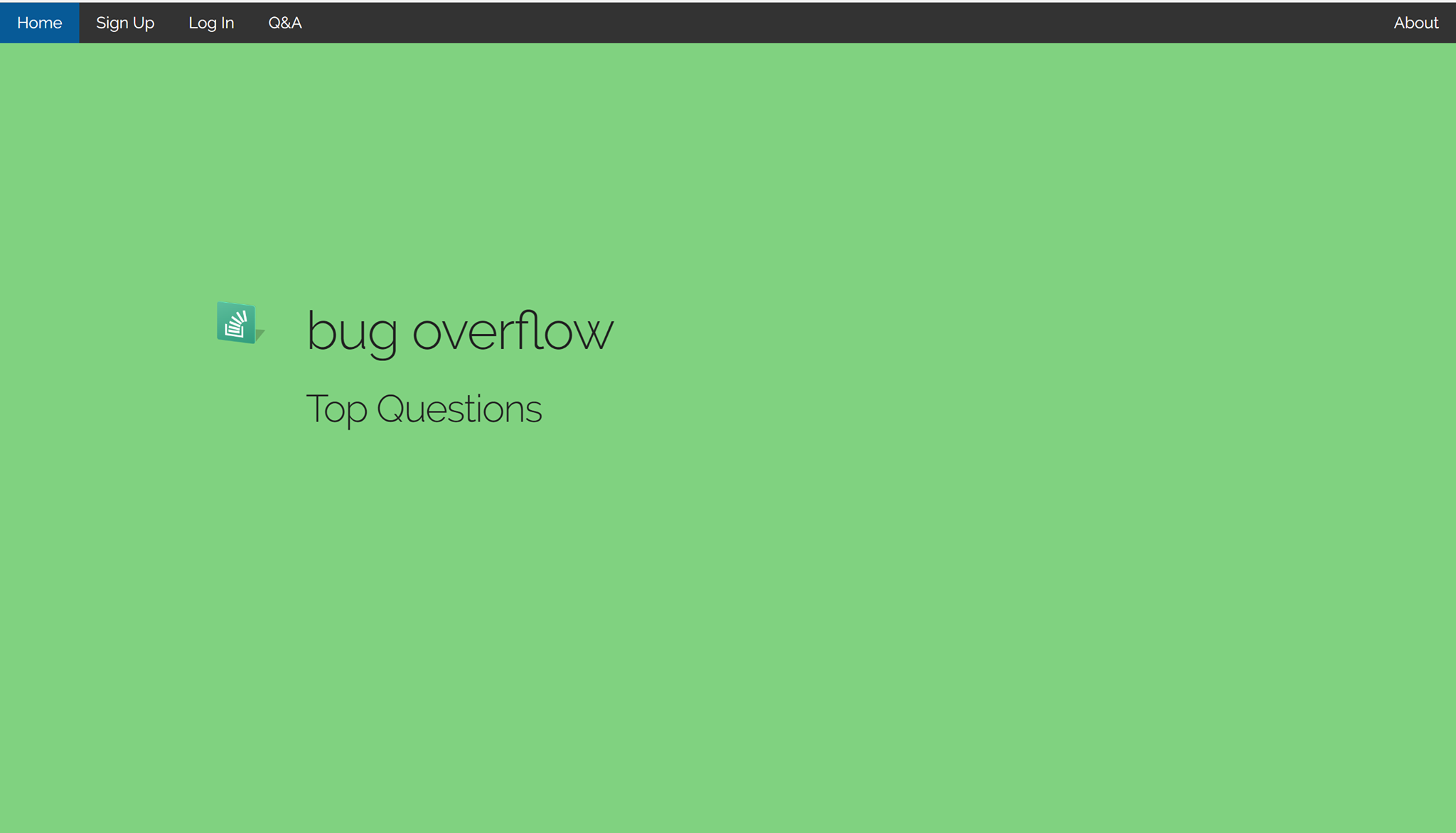


Figure 5, mobile-first design home page during mid-point in project

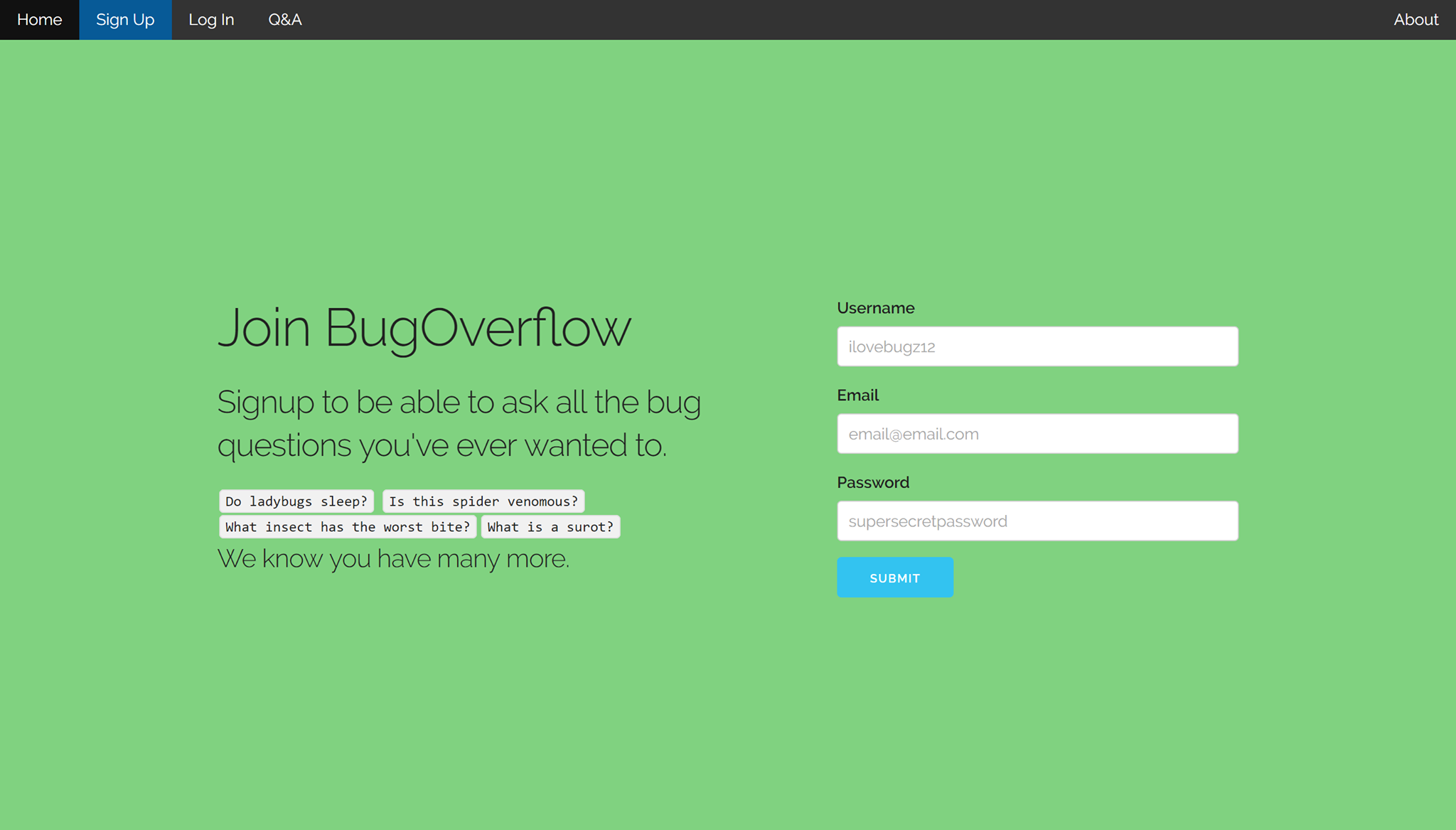


Figure 6, Sign-up page with minimum fields to encourage easy membership access

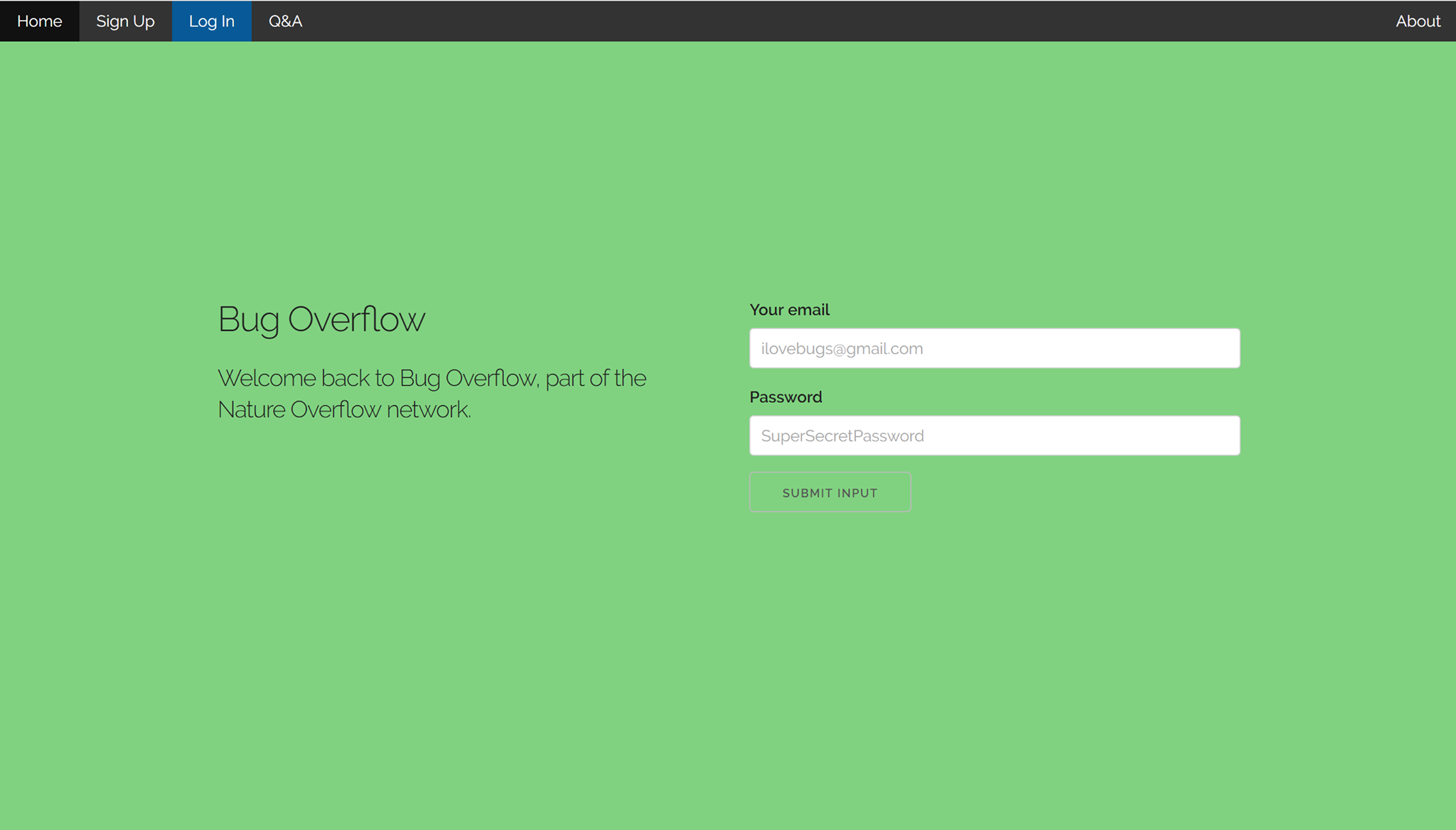


Figure 7, BugOverflow login page

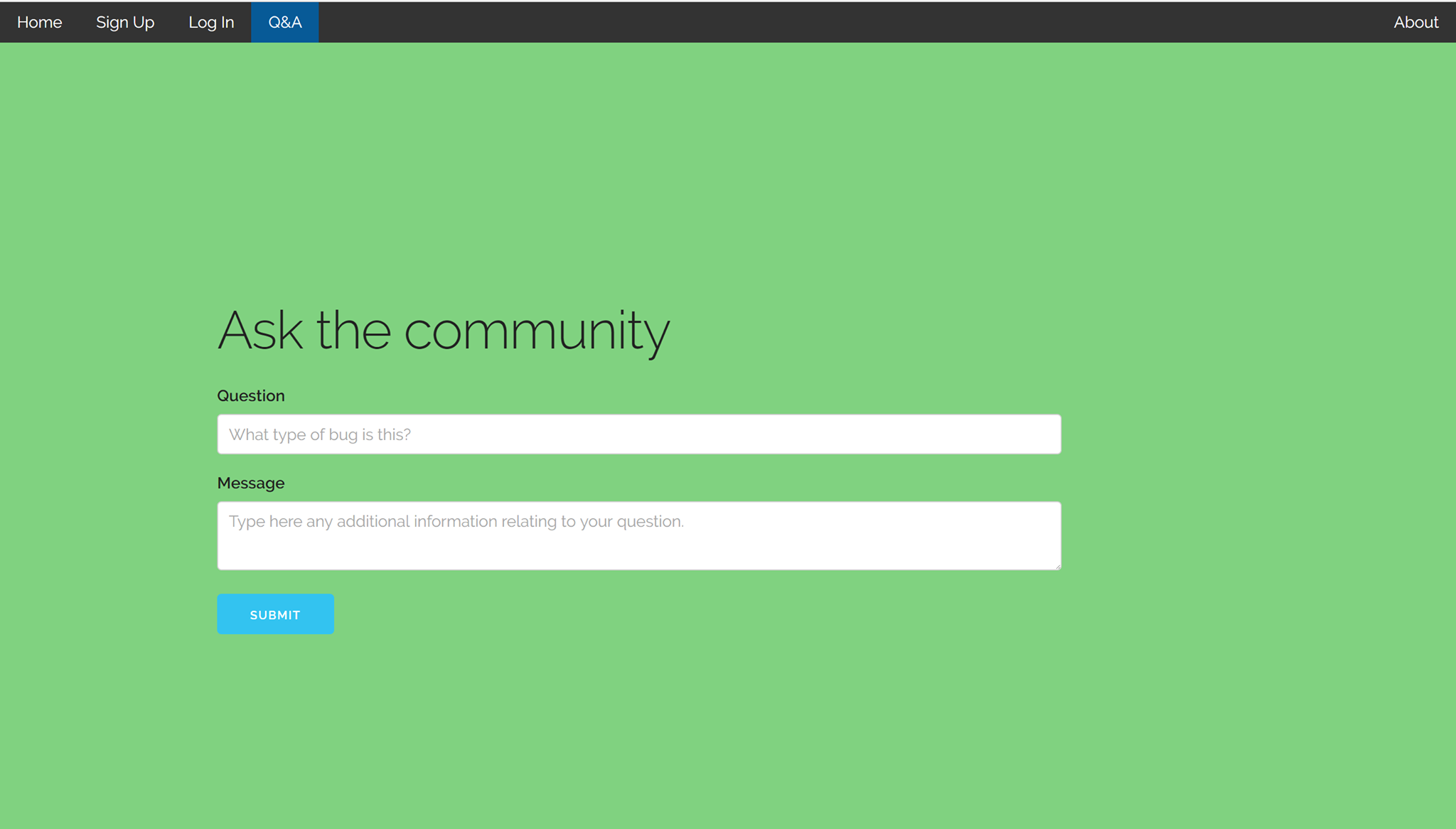


Figure 8, Q & A page still under development

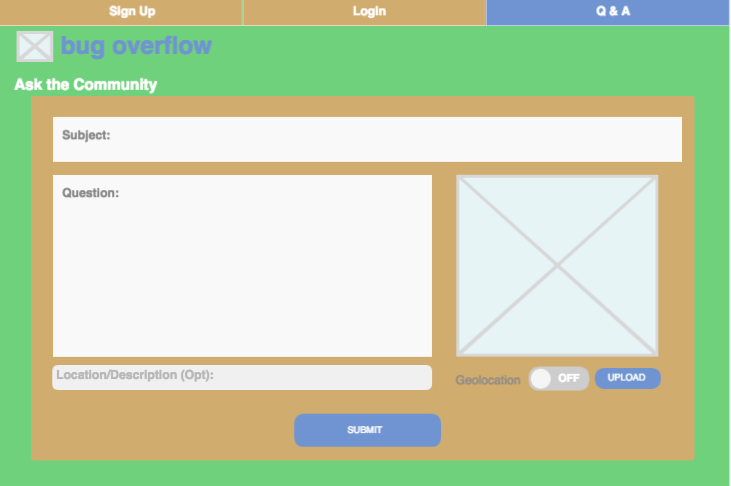


Figure 9, wireframe of Q&A page featuring image upload and geolocation properties

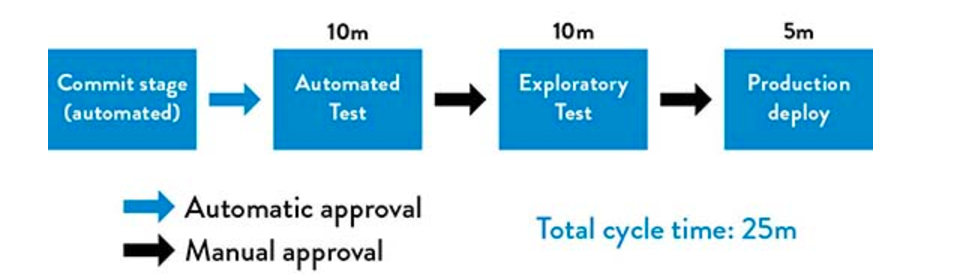


Figure 10, Google’s deployment pipeline (Source: Kim et al., The Devops Handbook, 2016.)

**Application Development**

One of the more important parts of building out this website is the ability for users to log in and be authenticated. Since the website revolves around user submitted questions and user answers it is important that users be signed it to be able to access features. Our user authentication will be carried out using our @hawk.iit.edu emails with the help of OTS. We have not yet discussed how this is done but our team currently assumes that it will be something very similar to authenticating by searching through a database of users and matching username/password combination. Once authenticated, there will be different levels of accounts. These levels and their authorization levels are described below:

* Anonymous (Non-Registered Users):
  + Able to read from the database but not write to it. In other words, cannot post questions.
  + Will have access to the post question page but will be prompted to sign in.
* Users (registered users):
  + Able to write to the database (post questions.)
  + Able to delete self-created content.
  + Able to read database (other questions and answers on website.)
* Administrators
  + Able to delete any user content.
  + Can read and write to database.

These features will be implemented using sessions in PHP and the account-specific options (like deleting any post by the admins) will be done through hidden objects via CSS and HTML manipulation with PHP depending on the user.

When a user is authenticated, either via the OTS libraries or our own user DB, PHP will be used to create a session and store the session ID as a cookie via the $\_SESSION[] function. Based on the user we can flag a session as being an admin account or a normal user. PHP scripts will run once the user logs in to check if the user is an admin or a user. If the user is an admin then there will be PHP functions that insert HTML with UI options only available to administrators. For example, each Question and answer page will now display a button in the UI with the ability to execute an SQL query to delete that specific question. While any other user will only have this option if the userID of the active user matches the userID associated with that post. HTML manipulation through PHP is already being used on the current website. Our navigation bar code exists in a separate. The php file and every other page has PHP code to load in the navbar code. This way if we need to change navigation bar features we can do It from a single source and not copy and paste it to every page the website has.

Other features being developed is the ability to upload a picture of a bug taken by the user. The image will then be processed via the PHP exif\_read\_data($file) function and some more PHP code will look at the outputted array trying to find the geolocation metadata of the image. If no geolocation is found the image will still be uploaded and displayed in the post. However, if geolocation of the image is found it will ask the user if they wish to add the location. Alternatively, the user can manually enter the location. This location data will be stored in the database and then through Google Map’s API we will plot the location of the bug on a map. The aim of this is to be able to have a sort of “Bug Spotting” feature where users can look up types of bugs and they can see on the map the different places they have been found.

Similarly, another important feature is the ability to search. There will be a search bar in the navigation bar where a user can search for keywords. A PHP function will then submit a SQL query to the database searching through the questions or comments and will display back a list of where the keyword was found. We are still finding out the best way to do this, looking to see if there are any existing search algorithms we can use because as we currently have it the display will simply be a list of places where the string being searched for may appear even if the question has not little to no relevance.

**Operations and Infrastructure**

**Database Schema**

We use MySql to implement our BugOverflow database. There are a total of seven entities which are Bugs, BugLocations, Users, Questions, Answers, VoteTypes and PostFeedback. All bugs’ information all store in the “Bugs” tables and the relative location data will keep in the Buglocations’ tables. Users can create questions to report the new bugs which they found or answer other users’ questions. Everyone can vote on these questions and keep these records in the table of PostFeedback. We can use PostFeedback to rank these questions and find which one is more popular.

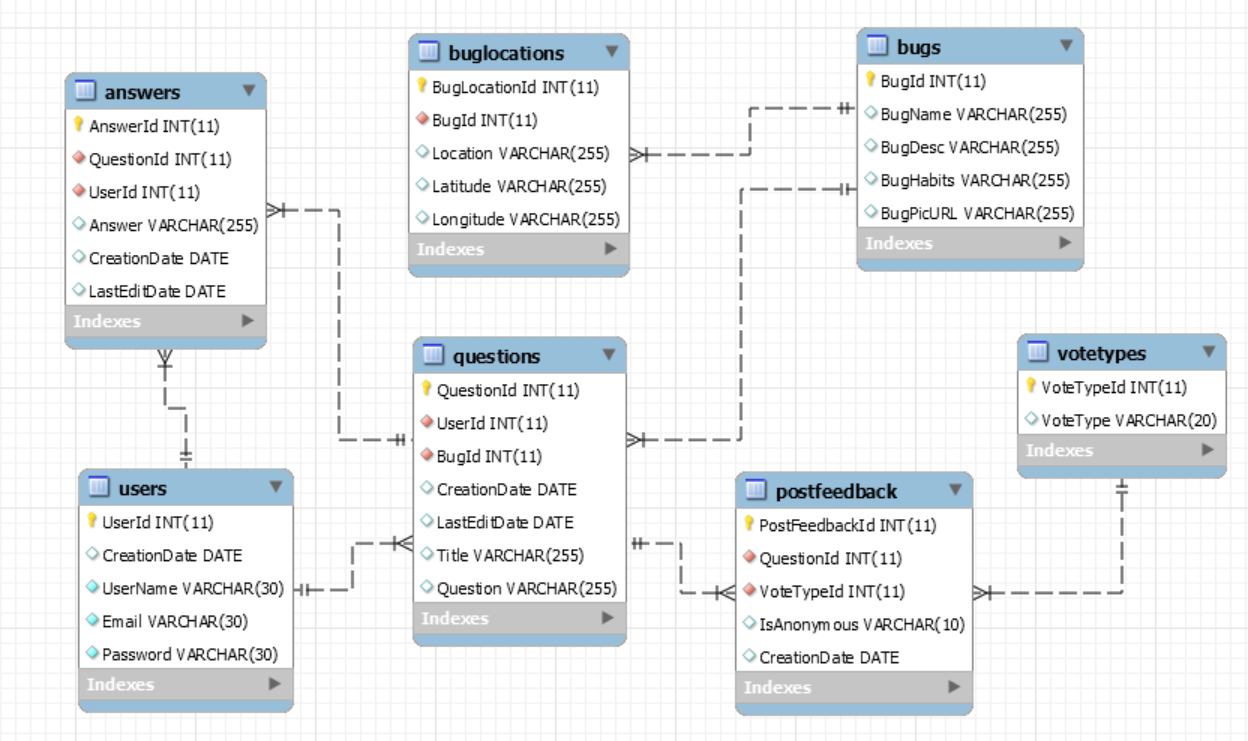


Figure: Current Database Schema of Bugoverflow

The table, “Bugs” will store the bugs’ data which include bugs’ names, descriptions, habits and pictures. Specifically with pictures, we will store them in the AWS Cloud and keep their URL in the table.

The table “BugLocations” will store the bugs locations’ data which include the locations the bugs appeared, the latitude and longitude related to the location. Basiclly, we will use the latitude and longitude as the coordinate to plot the location in the Google Map. Users can offer all this information or some of it. If users only tell us the locations, we will convert it into appropriate latitude and longitude.

The table “Users” will keep users’ information which include the dates when users account was created, usernames, their emails and passwords. The created data will be generated by the system. User name should be unique. If duplicated, the system should ask users to create another username. The passwords should be encryped and we will store the hashed password in the database.

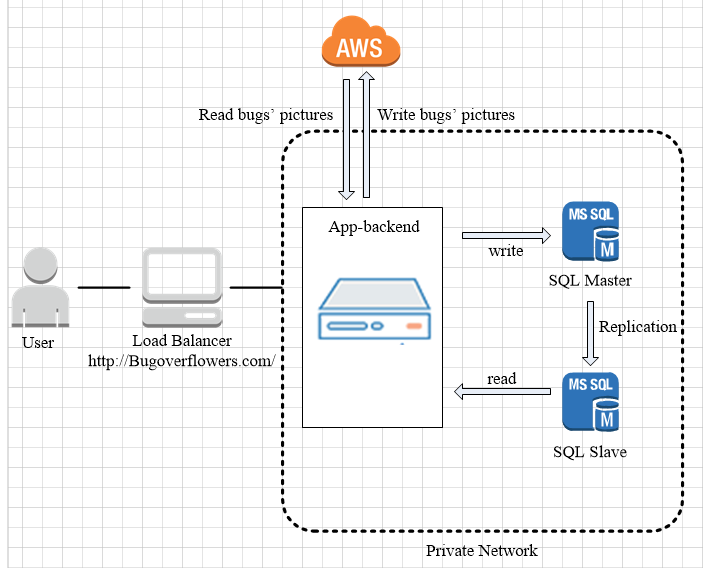
The table “Questions” will have the data of questions which include the questions’ titles, their contents, created dates, and revised dates. Created dates and revised dates are all generated by the system. Revised dates can be updated many times and the database only keep the last update.

The table “Answers” will have the data of answers which include the answers, referred questions’s ides, created dates and revised dates. Like the table “Questions”, created dates and revised dates in the table “Answers” are all generated by the system. Revised dates can be updated many times and the database only keep the last update.

The table “VoteTypes” will have three types of votes. They are “Good”, “Okay”, and “Bad”. Users can choose one of these valuse to vote for questions.

The table “PostFeedback” will store the data of feedbacks which is other users vote for the questions. The attributes in the table inculde related question ids, vote options from the table of “VoteTypes”, anonymous vote, or not, and created date. We will assume these votes are all anonymous and will change it if users log in and vote. Like the table “Questions”, created dates in the table is generated by the system.

**Database function**



We create Master/Slave replication in our database. There are two databases in two servers, SQL Master and SQL Slave. Our application will send the data to SQL Master and read the data from SQL Slave.

**Improvements**

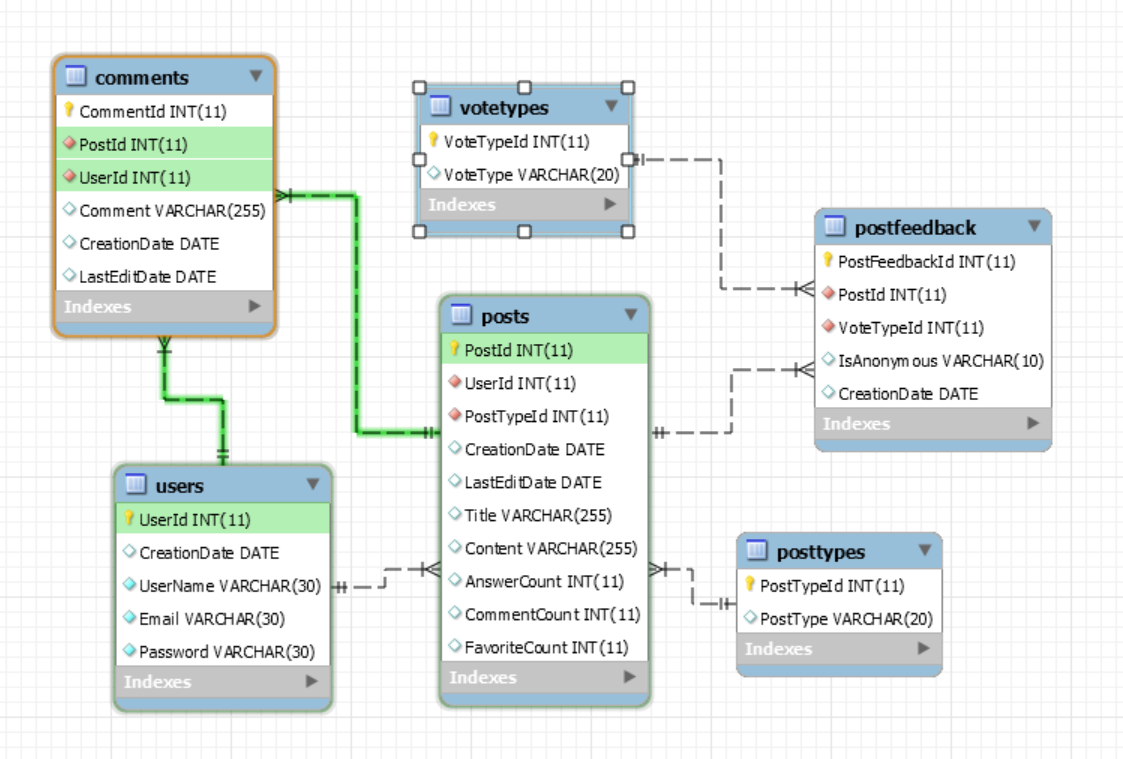


Figure: Database Schema of BugOverflow in old version

When we created the database schema, we choose to use the second way of applying Lean principles to the technology value stream which Gene Kim lists in the book of DevOps Handbook1. This way is from Ops to Dev, the flow feedback. We need to understand and respond to the needs of all customers, whatever internal and external. So, according to lean thinking, we realize that the most important customer is the internal one – the next downstream work center, Dev. Hence, we got the feedback from Dev and revised the database schema. For example, in the old version of database schema, users will create posts which are questions or answers. Users can add comments after each question or answer. Dev went through this schema has too much layers and it was too complex to implement. Then, we changed and use this current one.

In the future, we will create A/B test in our daily work and everyone can test the database function. If anyone finds some place need to improve, he or she can revise the database. It is the third way of applying Lean principles to the technology value stream which Gene Kim lists in the book of DevOps Handbook1. In this way, we can create the continual experimentation and learning. We will break things early and often. Finally, we will make deployments go more smoothly.

**Database tables:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Table: Bugs** | | | | |
| **Column** | **Attributes** | **Data Type** | **Null or Not Null** | **Explanation** |
| 1 | BugId | INT | Not Null | Primary Key |
| 2 | BugName | VARCHAR (255) | Null | The bug's name could be null first. After identified and offered the answer by other users, we can add its name. |
| 3 | BugDesc | VARCHAR (255) | Null | Description or definition for the bug |
| 4 | BugHabits | VARCHAR (255) | Null | Habits column should be bug's favorite food or living environment. |
| 5 | BugPicURL | VARCHAR (255) | Null | We store the bugs' pictures in the cloud and offer the web links. |
| **Table: BugLocations** | | | | |
| **Column** | **Attributes** | **Data Type** | **Null or Not Null** | **Explanation** |
| 1 | BugLocationId | INT | Not Null | Primary Key |
| 2 | BugId | INT | Not Null | Foreign Key for the table of Bugs |
| 3 | Location | VARCHAR (255) | Null | Users can offer the data of location or latitude and longitude. If users only offer the location, we need using PHP to transfer it into attitude and longitude. |
| 4 | Latitude | VARCHAR (255) | Null |  |
| 5 | Longitude | VARCHAR (255) | Null |  |
| **Table: Users** | | | | |
| **Column** | **Attributes** | **Data Type** | **Null or Not Null** | **Explanation** |
| 1 | UserId | INT | Not Null | Primary Key |
| 2 | CreationDate | Date | Not Null | The system auto generates this data. |
| 3 | UserName | VARCHAR (30) | Not Null | User name should be unique. If duplicated, the system should ask users to create another username. |
| 4 | Email | VARCHAR (30) | Not Null |  |
| 5 | Password | VARCHAR (30) | Not Null | We will store the hashed password in the database |
| **Table: Questions** | | | | |
| **Column** | **Attributes** | **Data Type** | **Null or Not Null** | **Explanation** |
| 1 | QuestionId | INT | Not Null | Primary Key |
| 2 | UserId | INT | Not Null | Foreign Key for the table of Users |
| 3 | BugId | INT | Not Null | Foreign Key for the table of Bugs |
| 4 | CreationDate | Date | Null | The system auto generates this data. |
| 5 | LastEditDate | Date | Null | Question can be edited by users and the system is auto generated by the data. |
| 6 | Title | VARCHAR (255) | Null |  |
| 7 | Question | VARCHAR (255) | Null | This is the content of the question. |
| **Table: Answers** | | | | |
| **Column** | **Attributes** | **Data Type** | **Null or Not Null** | **Explanation** |
| 1 | AnswerId | INT | Not Null | Primary Key |
| 2 | QuestionId | INT | Not Null | Foreign Key for the table of Questions |
| 3 | UserId | INT | Not Null | Foreign Key for the table of Users |
| 4 | Answer | VARCHAR (255) | Null | This is the content of the answer. |
| 5 | CreationDate | Date | Null | The system auto generates this data. |
| 6 | LastEditDate | Date | Null | Question can be edited by users and this data is auto generated by the system. |
| **Table: VoteTypes** | | | | |
| **Column** | **Attributes** | **Data Type** | **Null or Not Null** | **Explanation** |
| 1 | VoteTypeId | INT | Not Null | Primary Key |
| 2 | VoteType | VARCHAR (20) | Null | There are three types of values, good, ok and bad. |
| **Table: PostFeedback** | | | | |
| **Column** | **Attributes** | **Data Type** | **Null or Not Null** | **Explanation** |
| 1 | PostFeedbackId | INT | Not Null | Primary Key |
| 2 | QuestionId | INT | Not Null | Foreign Key for the table of Questions |
| 3 | VoteTypeId | INT | Not Null | Foreign Key for the table of VoteTypes |
| 4 | IsAnonymous | VARCHAR (10) | Not Null | Choose the default value, Yes. |
| 5 | CreationDate | Date | Null | The system auto generates this data. |

**Security**

Security is one of the most important tasks in Information Technology. The tasks of keeping the information of a company, a website, data, and policies secure, is extremely important. Security vulnerabilities are hard to detect, but with the right plans in place, information can be protected from prying eyes. We plan to discover several ways to secure our project, using secure usernames and passwords. If passwords and data are encrypted, they can be hard for non-authorized users to detect. The task of security will also entail testing, setting up firewalls, and helping in other areas. Supplying the data to be entered in the database is one of the areas I will be discussing here.

Our team name is Bug Overflow, and our database will describe the bugs found in and outside of your home. Say for instance you found a bug in your basement that looks weird and you have no idea of what kind of bug it is, where it came from, and why it is there. You can take a picture of the bug, upload it to our site, and it will give you the name and information of that bug. This is a process and we are bound to run into some problems along the way. At that point we must swarm them, mobilizing the person best fit to solve it. The goal to swarming is to contain the problem before it can spread. It is necessary to prevent other problems from progressing downstream. This can affect cost and effort to repair it, and can increase technical debt. Swarming enables learning, and that is why testing is so important also.

In creating the admin usernames and passwords for each team member, the site we used to get secure usernames is Jimpix username generator. The usernames that were generated from the site, were changed around and different characters were added. Some letters were changed to be sure they are unique. Then the passwords were created from Norton’s identity safe site. The same technique was used to change the usernames, to change the passwords to make them unique also. The next step is to set up an admin email account using Google. Source code integrity and code signing should be used meaning all developers should have their own PGP keys created in a managed system, and for this project we will use GitHub.

The data to be stored into the database has been created, and pushed to GitHub. There are thousands of bugs in the world and we’ve come up with a few unique ones, and some common bugs that you may see every day. We want to keep our data secure from SQL injections, and this will require manual and automated testing. Tests that can run continuously in our deployment pipeline. One such tool that can be used is Gauntlt, which puts its security code in Gherkin syntax scripts. Used by a wide variety of developers for unit and functional security testing on every committed change such as static code analysis. We can do this by deploying a static analysis. A Static analysis tool will inspect program code for possible run-time behaviors and seek out coding flaws, errors and backdoors that could be potentially malicious code (known as testing from the inside-out). An example of this is Brakeman, Code Climate.

We could also use Dynamic analysis tool which consists of testing while the program is running, which can monitor items in the systems memory, functional behavior, and response time, (known as testing from the outside-in). It would be wise to also include dependency scanning performed at build time, inside of a deployment environment. This involves inventorying all our dependencies for binaries and executables, making sure they are free from malicious binaries and vulnerabilities. Examples are Maven for Java, and the OWASP (Open Web Application Security Project) Dependency-Check (Kim, pg. 667.9/880).

Applications must be secure by using firewalls, and using tools such as Nmap to ensure that only expected ports are open, and Metasploit to make sure our environment is securely hardened against vulnerabilities such as scanning with SQL injections. It is also important to make sure that our application is not vulnerable to CSS background attachments, and XSS (Cross Site Scripting) which enables attackers to inject client-side malicious scripts into web pages. An attacker can use it to bypass controls. We should enable alerts for items such as XSS, unsuccessful login attempts, to successful logins.

The site will use http session control logins to prevent non-authenticated users from entering the site. By creating a login, and logout account, the user must be authenticated to log into an account. If user does not have the proper credentials, they will not be able to log in. The http code will block any unauthenticated users from entering site. Once the page closes the user is automatically logged out. By looking for database syntax errors and adding unit tests we could ensure that certain types of uncontrolled user input would not be allowed in our database queries (p. 687.8).

By using an algorithm to transform the data to cipher text we can encrypt the content of our database, therefore rendering the text useless to attackers. Unless they can decipher the data, there is nothing they can use. There are multiple techniques and technologies we could use such as Transparent/External encryption (encrypts the entire database). Transparent involves encrypting “data at rest,” which is inactive data that is not being edited or pushed across the network. Also, symmetric, and asymmetric database encryption. They both involve a private key being applied to the data to call it from the database.

Lastly, our database must use TLS/SSL certificate from letsencrypt.org for production site. This site allows us to encrypt for free, and collaborate. The process of setting up our database, ensuring that it is a working and viable source is important. All the aspects and duties lead us to take a good look at how we can secure information not only in our homes, but in our work, our schools, and everywhere.

**Bibliography**

Database encryption. (2016, December 31). Retrieved March 01, 2017, from

https://en.wikipedia.org/wiki/Database\_encryptionDatabase encryption. (2016, December 31). Retrieved March 01, 2017, from <https://en.wikipedia.org/wiki/Database_encryption>

Google. (n.d.). Retrieved March 01, 2017, from

<https://www.google.com/webhp?sourceid=chrome->instant&ion=1&espv=2&ie=UTF-8#q=what is xss attack&\*

Kim, G. (2014, May 19). *DevNation 2014 - Gene Kim - Afternoon Keynote: Why*

*Everyone Needs DevOps Now* [Video file]. Retrieved from <https://youtu.be/vTxAR8EOc8A>.

Kim, G., Humble, J., Debois, P., & Willis, J. (2016). *The Devops Handbook How to*

*Create World-Class Agility, Reliability, and Security In Technology Organizations*. Portland, OR: IT Revolution Press.

Kim, Gene & Humble, Jez & Debois, Patrick & Willis, John. ( © 2016). *The devops*

*handbook: how to create world-class agility, reliability, & security in technology organizations.* [Books24x7 version] Available from <http://common.books24x7.com.ezproxy.gl.iit.edu/toc.aspx?bookid=117823>.

Limpalair, C. (2016, March 23). What does Etsy’s architecture look like today?

Retrieved from <http://highscalability.com/blog/2016/3/23/what-does-etsys-architecture-look-like-today.html>.

Password Manager. (n.d.). Retrieved March 01, 2017, from

<https://identitysafe.norton.com/>

Servlet HttpSession Login and Logout Example - javatpoint. (n.d.). Retrieved March 01,

2017, from <http://www.javatpoint.com/servlet-http-session-login-and-logout->example

Username Generator. (n.d.). Retrieved March 01, 2017, from

20 Feb 2015 at 10:31, John Leyden tweet\_btn(). (n.d.). Evil CSS injection bug warning: Don't let hackers cross paths with your website. Retrieved March 01, 2017, from https://www.theregister.co.uk/2015/02/20/prssi\_web\_vuln/