User Stories and Set-up Assignment

Brendan Byers, Ryan Sisco, Iliana J, Aidan Grimshaw, Yufei Zeng
byersbr, siscor, javieri, grimshaa, zengyu

Contents

1 Product Release

1.1 Server Side Implimentation

Currently the server code can be downloaded and run locally. By cloning the repository located at

```
https://github.com/B13rg/Talkatiel_API.git
```

one can run the server. There is also extensive documentation detailing all the steps that need to be taken to run the server. There is also a SQL script that will create a database when run. This can be used to create a database for use on a different engine. Currently it is configured to run on the localhost on port 5002. This can be tested to navigating to

`127.0.0.1:5002/Posts/New`

where you can see the raw Json output of a post. In the repository there is also an sqlite database that is run alongside the python code. This will connect with the python to respond to different sql queries. It is fully functioning, and the only thing left to do is test it, and find a permanent URL.

1.2 Client Side Implimentation

2 User Story

2.1 Add a post

The server side group worked on adding this ability to the api. Currently a user is able to POST a json string to the server. The server takes the json and decodes a post object out of it. Then the post is passed into the database to be retrieved later. Implimentation of this took 2 hours. Currently we still need to test the implimentation with more extensive unit tests.

2.2 Add a Comment

The server side group worked on adding this ability to the api. Currently a user is able to POST a json string to the server. The server takes the json and decodes a comment object out of it. Then the post is passed into the database to be retrieved later. Implimentation of this took 1 hour. It was extremely similar to adding a post. Currently we still need to test the implimentation with more extensive unit tests.

2.3 Like/Dislike

The server side group implimentated a way to like and dislike posts. It is done by throwing a 1 or a 0 to the url of the post. This will look like this:

`https://Talkatiel.com/Posts/<int:postID>/<int:voteType> .`

The server decodes the url and applies the proper vote to the given post. Implimentation took around 3 hours. Currently there needs to be more testing done.

2.4 Report

Server group implimented this through the API. In this case, when the server recieves a report about a post, the postID reporting userID, and report message content will be added to a special "reports" table in the database that will be able to viewed by admins later.The implimentation took around 3 hours. Only thing left to do is add a way for admins to access the reports in an orderly way.

2.5 Delete

The server group worked on the server side implimentation of this. In order to delete a post a url is GET requested with the postID. The id is decoded, and the given post is marked as not visible. When posts are retrieved from the database for users, the deleted posts are ignored. The work on this took 2 hours. It was closely tied to the refresh function. Currently we need to impliment some form of authentication so only requests with the correct key can delete a post.

2.6 Refresh

The server group worked on this. In order to refresh and retrieve new posts on the deivce, one of 3 urls are sent GET requests. The URL's determine the type of sort the post are in when returned. When retrieving posts for the user, the server first checks it's cache. The posts for any given category are only calculated once every two minutes. Each time it fetches it will store what it returns in the cache. This is so we don't overload the database with too many requests. The only exception in when fetching "New" posts. There is no caching done for fetching them because theres no calculating to be done. Currently the API is functional. Implimentation of this portion took 6 hours. The only thing left is testing.

2.7 Sort

The server side implimeted the different types of sorting. The posts are fetched from the database based on post date. When fetching "Best" or "Top", the order of posts must be calculated. For the "Top" category, the best posts of the last 24 hours are at the top. They are ranked based on upvotes minus downvotes. They are only considered if they're marked as visible. This calculation takes place every 2 minutes due to the use of a cache. For fetching "Best", the post order is calculated every 2 minutes thanks to the cache. This cuts down on how many requests we hit the database wth. The calculation factors in post time and number of upvotes and downvotes. This is a potentially expensive alculation given post activity, so the use of the cache was really important for this set. The total time for implimentation took around 7 hours. The only thing left is to test the server side of things.

2.8 View Newly Created Post

Server side worked on this. Because of how sorting by new is implimented, a user is able to see their posts immedietly. This is because there is no caching done on that sorting method. This makes it so the database is queried every time. If the server load was a lot, it would still be acceptable to cache things for even 5 minutes to lighten the load on the server.

2.9 Secure HTTPS Connection

In order to get a secure https connection, we need to get a security certificate signed by a certificate authority. We havn't chosen where the code will be hosted. Currently we are considering either trying to use the university services or a third party. I believe that we

may be ably to host the API server using virtual environments, which may allow us to run a server on that port. Time spent was 1 hour. We still need to finalize design choices to recieve the certificate.

2.10 Save Website as App on Phone

Frontend team worked on this part. Setting up the temporary server led to many issues. There were many dependencies that needed to be installed, and required a linux operating system to run. After running linux, many other programs needed to be installed before development was possible. Running the server required internet connection on both the laptop as well as the mobile device, and they needed to be on the same network. We soon realized that OSUs network would not allow this, and development had to occur off campus. The first laptop took Aidan nearly 3 days before the server was up and running. The second laptop only took 2 hours. Downloading the dependencies to run the app took nearly 40 minutes. Every new laptop we add will take at least this long to run our app while we develop. It is completed and tested. Once we are satisfied with our mobile app, we will transfer the files to our main server and push an android app to the website for download. The ios app will need to be approved by Apple before it can be added to the marketplace. The UML diagram was not very useful for this. We did not really understand what we needed or how we could accomplish this. We really needed to dive head-first into it in order to find a solution that worked for us. Planning it out ahead of time required us to be correct the first time.

2.11 Responsive Web Page

This was worked on by the Frontend team. There were many different dependencies that needed to be understood in order to make the UI. In addition to pulling in different frameworks, the JavaScript needed to be the only styling in the app. There was no CSS involved, and the only interaction with styling is done through JavaScript. Once the base code was written, the other parts could be added and changed later on. Developing the basic UI took nearly 10 hours in total. This involved building the base that we could build off of later on, and setting the correct styling data to be changed when needed. This is still in progress, but has been implemented. The colors, server data, and size still need to be corrected. The reason that we waited until after we built a base was to have flexibility later on if we felt that we needed a change. We did not want to have to scrap all of our code. The UML diagram was very useful in this case. It helped guide the project and understand what needed to be added for the base. Having an outline allowed the JavaScript to be built for the long run, and it will be able to support the app very well.

3 Tests

3.1 Server Side Implimentation

Currently to test the server side of things one needs to manually query the API themselves. The next step in our plan is to add more unit tests. We plan on making a python script

that will test a given URL and return it's status and how it's doing. It will be able to do through this through a mix of manually crafted GET and POST requests and pinging. This will let us quickly and easily see if anything is going wrong.

4 Design Changes and Rationale

4.1 Server Side Implimentation

For the most part the server group stayed very close to original design. To design the database we worked off the UML design digrams we designed earlier. This allowed us to see what fields we needed to include for each table. We added an extra table not originally in the design doc to handle reports. We found there was no way to keep track of reports server side, so we added a table to handle it. It keeps track of the user who reported it, the post in question, and the text of the users report. By having it's own table it can also be queried seperatly instead of having to sort through who knows what.

4.2 Frontend Implimentation

We have stayed very close to our original design. The changes we have right now are temporary and easy to change. We do not have some things matching such as colors, but we have every correct button and card in position. Styling the rest will be very easy. Our buttons do not currently send data to our actual server. This was done intentionally in order to build the backend and frontend at the same time. We are currently fixing the color scheme as we get closer to connecting our mobile app with the database.

5 Meeting Report

5.1 Team Member Contributions

- Server Side - API - Brendan Byers
- Server Side - Database - Iliana Javier
- Server Side - User Story - Brendan Byers and Iliana Javier
- Server Side - Design Changes - Brendan Byers and Iliana Javier
- Frontend - Mobile App - Ryan Sisco and Aidan Grimshaw and Yufei Zeng
- Frontend - Basic UI JavaScript - Aidan Grimshaw