CSC309 A2 README

Project Members

- Alexander Biggs (g1biggse)
- Gabriel Nunes (c2nunesg)
- Daphne Ippolito (g1daphne)

Connecting

Server: redwolf.cdf.toronto.edu
Port: 31285

Organization

The main assignment folder is divided into server, client and sql folders. The server folder is broken down under *Implementation Overview* and *Implementation Details*. The client folder is the teacher's provided client, with an added shell script to automatically update the client to match the latest data from the server. The sql folder contains sql commands that can be run to recreate the database, and includes the schema for the database.

Implementation Overview

The implementation of the server was divided into three major components:

1. The database query calls (*database.js*)
2. The Tumblr API calls (*tumblr.js*)
3. The request handlers (*requestHandlers.js*)

Furthermore, we abstracted various functions for our server into different modules:

1. Specifying which request handlers map to which requests (*index.js*)
2. Mailing the administrators about issues and updates (*mail.js*)
3. Functionality for handling database queries in a specific order (*queue.js*)
4. Updating the database (*updates.js*)
5. Routing requests to the appropriate request handlers (*router.js*)
6. Starting and maintaining the server (*server.js*)

Implementation Details

Database Schema

- **tracked_blogs**: information for each tracked blog
    - *url*: the url of the tracked blogger
    - *username*: the username of the author of the tracked blog
    - *PRIMARY KEY(url)*: the url of each blog should be unique, while multiple blogs may have the same author (username)

- **likes:** Indicates which blog user likes which posts. This table is needed because multiple users can like the same post, but we only want to store one instance of each liked post.
    - *liker*: the username of the blog post person
    - *post_url*: the url of the liked post
    - *PRIMARY KEY(liker, post_url):* both tuples must be part of the key because a liker can like multiple posts, and posts can be liked by multiple users

- **liked_posts**: information for each liked posts
    - *url*: the url of the liked post
    - *date*: the date the post was made
    - *image*: the url of an associated image
    - *text*: text associated with the post
    - *note_count*: the latest note count for the post
    - *num_updates*: the number of updates that have been made to the note count since we started tracking
    - *PRIMARY KEY(url)*: all posts should be unique by their url

- **updates**: An update to the note count of a post. There can be multiple updates for any single post.
    - *url*: the url of the post this update corresponds to
    - *time*: the time the update took place
    - *sequence_index*: the index of this update in the sequence of updates
    - *increment*: the amount the note count has gone up since the last update
    - *PRIMARY KEY(url, sequence_index)*: all updates should be unique

by their post url and the position of the update in the sequence of updates

By including a separate table for likes, we were able to allow for multiple blogs to like the same post without storing redundant post information.

Database Querying

Queries to the database were done used the Node.js Mysql library. In some instances, it was necessary to do queries in a specific order. For example, when creating a new update for a liked post, a "SELECT", then an "UPDATE", then an "INSERT" query, each relying on the successful completion of the previous query, must be executed in the specified order to avoid errors. In order to solve this problem for order-sensitive queries, we use a queue that stores in each node a query and the callback to be executed upon the query's completion. The queue will execute the queries in the order they are added to it.

The connection to the database is not persistent. A new connection is opened and closed each time a query is made. This is to avoid connection timeout problems -- if the connection remains open for longer than a few seconds, an exception is thrown.

Tumblr API Calls

Three API functions were used in the server's implementation:

- */posts*: get info on blog posts or a single blog post
- */likes*: get the posts that are liked by a blog
- */info*: get info on a blog. Allowed us to retrieve the username from a blog URL.

Calls to this API have been wrapped around easy-to-invoke functions. Each function takes a callback to run when the API call has been finished as opposed to returning the values directly at the end of the function's execution, since node runs requests asynchronously.

Request Handlers

We created four request handlers. Three of these correspond each of the three requests specified in the assignment guidelines:

- **trackBlog**: Track a new blog, adding its liked posts to the database and setting up the posts to be updated every hour. Corresponds to *POST*

"/blog"
- **getAllTrends**: Get the trending posts from all blogs. If the limit is not specified, only responds with the first 20 results. Corresponds to *GET "/blog/trends"*.
- **getBlogTrends**: Get the trending posts from a specific blog. If the limit is not specified, only responds with the first 20 results. Corresponds to *GET "/blog/<base_hostname>/trends"*

For the GET requests, if the order is specified as "Trending", the posts are sorted by the increment value of their updates.

The fourth request handler, **updateRequest**, manually updates the server. It can be accessed through the URL *GET "/update"*. In a production version of the server, this call should be removed or disabled, since excessive use of this call can cause violations of Tumblr's API limits.

Updating the Server

Updating the server is handled through a CRON job embedded into the server code that runs hourly and invokes an update method in the *server.js* module. This function in turn invokes functions from the *updates.js* module to add any new liked posts by the tracked blogs to the database and update all the posts that are already in the database.

For debugging and testing purposes, the call *GET "/updates"* triggers a manual update of the server.

Maintaining the Server

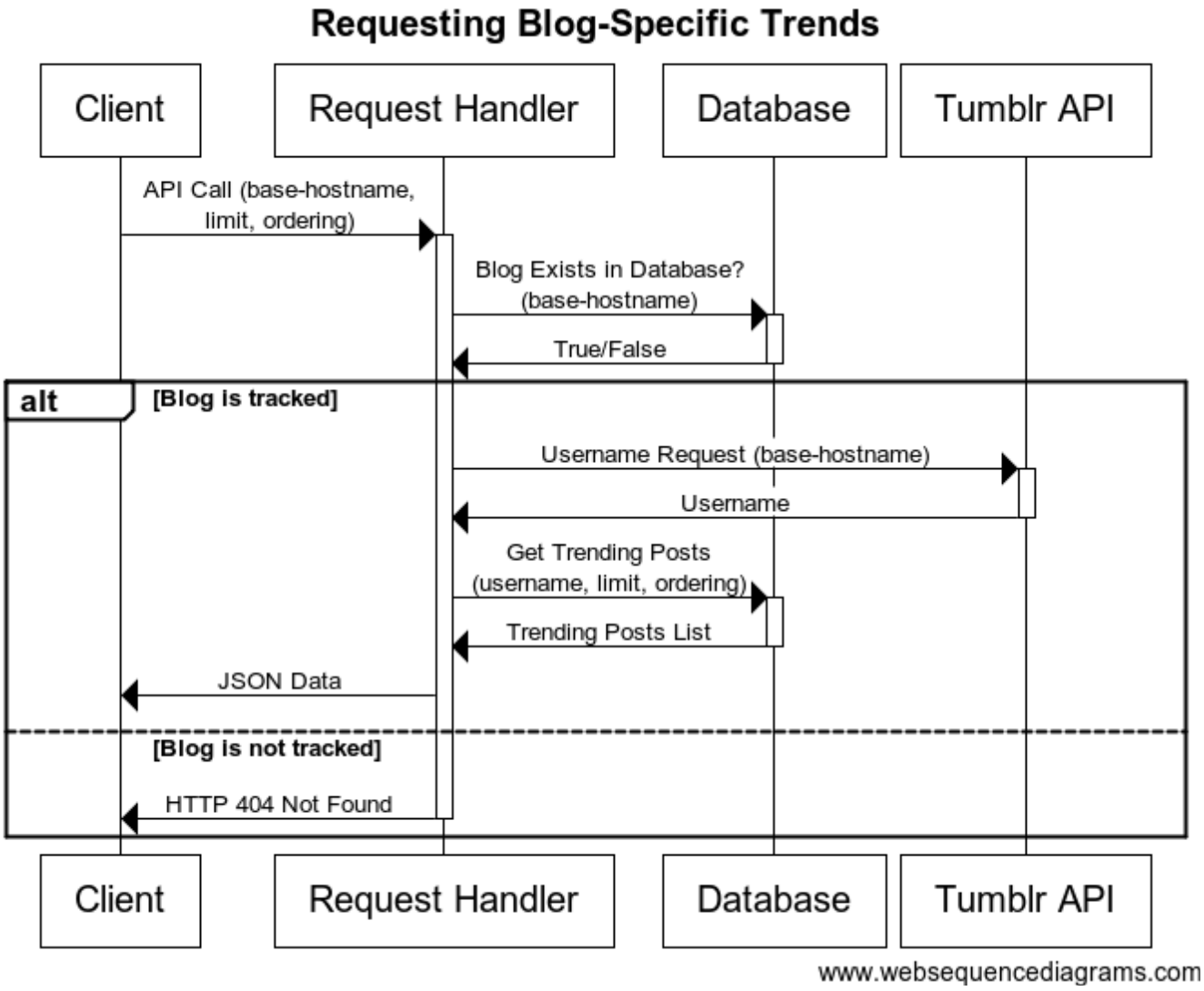
To ensure that the server remains running throughout the evaluation period, the server handles exceptions by emailing all of the group members with an error report including a stack trace of where the error originated from. It is the responsibility of the group members to restart the server when it crashes, because no automatic restarting has been implemented. This is to avoid repeating bugs that could be harmful to the server or database.

External Libraries
- **Cron:** A Node.js library used to have an update automatically occur after a given period of time. (https://github.com/ncb000gt/node-cron)

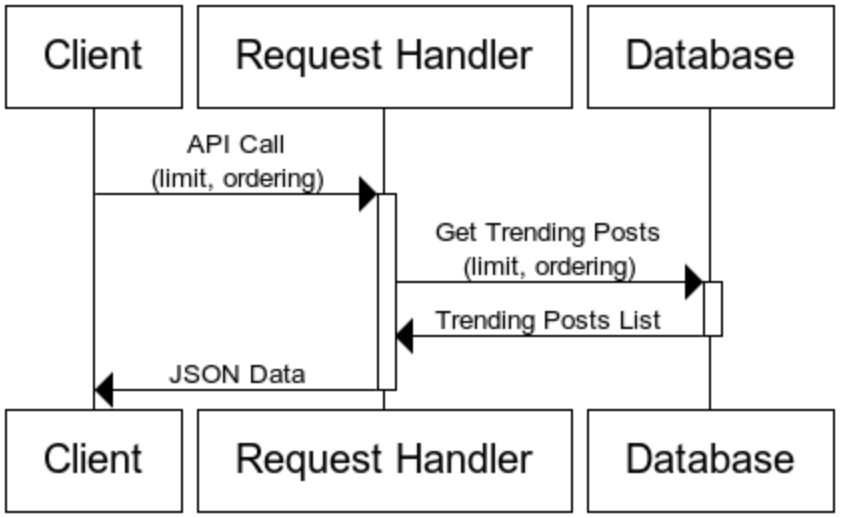
- **Mysql**: A Node.js library used to query the mysql database. (https://github.com/felixge/node-mysql)
- **NodeMailer**: A Node.js library used to send out emails to administrators when the server fails. (https://github.com/andris9/Nodemailer)

Sequence Diagrams

Higher quality images can be found under the "*diagrams*" folder.

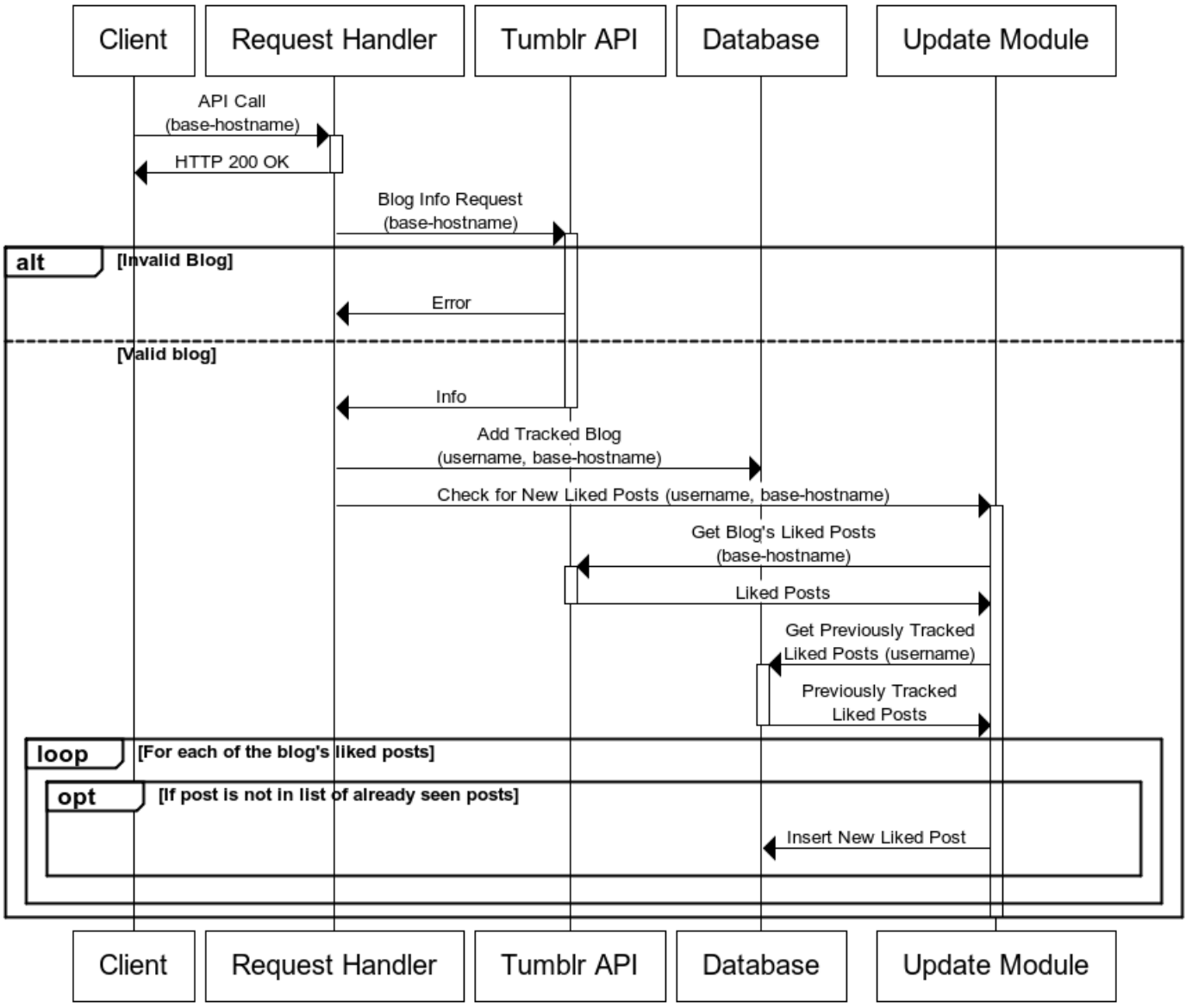


Requesting Global Trends



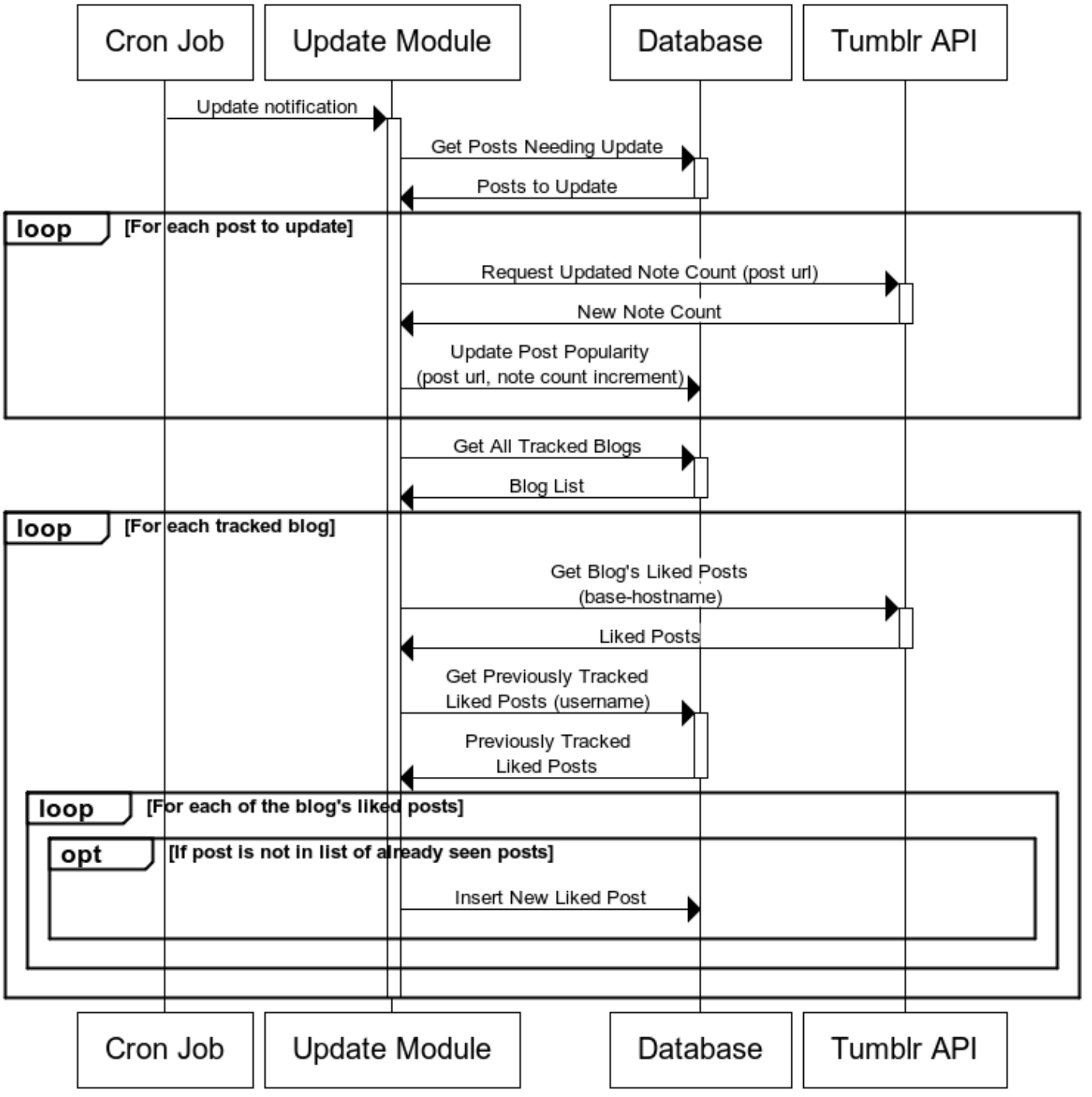
www.websequencediagrams.com

Tracking a New Blog



www.websequencediagrams.com

Updating Database



www.websequencediagrams.com