Part C: Test Report

Backend testing:

Queries were test entered manually to the mysql server (db4) values were simulated. Limit testing (255 char Topic, 2550 char data values) were done during load testing.

Endpoints were deliberately built to interface with posting.html from Assignment 3 to allow for early testing during build.

/ was tested to return posting.html, which allowed further testing using the unmodified A3 version.

/init was tested and configured to attempt to read postdb, then for the posts and reply. If any aren't found they are initialized and messages are output to the console.

/postmessage was tested with the post dialog jquery widget, data was compared with new entries in the database postdb until the /alldata endpoint was completed.

/postresponse was tested with the Add a response button embedded in the Posts, the ability to add a response is only enabled once a post is read. Data was compared with new entries in the database postdb until the /alldata endpoint was completed.

/alldata was tested as part of the page load, if any entries in the posts table exist they display on the first load, otherwise only the Create new post button is on the page. This endpoint can be called at any time from a client to receive a json object {postMessages, replyMessages}. postMessages = {id, topic, data, timestamp} from each post in the posts table, replyMessages = {id, postId, data, timestamp} from each response in the reply table.

Error handling from server side, if topic and/or data are empty for /postmessage an alert is spawned noting the deficiency. If postId and/or data are empty for /postresponse an alert is spawned noting that deficiency. postId should never be able to be empty as it is bound to the response form. Additional defensive programming using event.preventDefault(); in the frontend should prevent missing values from being received.

Frontend Testing:

Post and reply creation was confirmed using an unmodified posting.html from Assignment 3, the endpoints were designed to be compatible with the previous iteration. As each endpoint was configured tests were submitted and verified manually through the database interface on db4. Once /alldata was completed verification was through posting.html.

event.preventDefault(); forces filling in the form before submitting for both new posts and replies.

I noted that the timestamps saved to the posts and reply tables didn't include milliseconds. When testing for updates from additional clients lastUpdateTimestamp = Date.now(); console.log(lastUpdateTimestamp); output from the submitting client. The receiving client reads the lastUpdateTimestamp from the server.

Timestamp taken from:	Raw timestamp
Response submission	1741054895488
Server	1741054895487
Update trigger	1741054895595
Difference from Response to update	107 ms

Functional Test Cases:

Refresh from post on another client – Expected/Required < 300ms. Documented: 107ms.

Empty Topic for new post before enabling event.preventDefault() – Expected: 400 status, “Missing topic”, Alert spawned. Observed: Alert with {error: “Missing topic”}

After: Pop up “Please fill out this field.” bubble next to field with missing data.

Empty Data for new post or reply before enabling event.preventDefault() – Expected: 400 status, “Missing data”, Alert spawned. Observed: Alert with {error: “Missing data”}

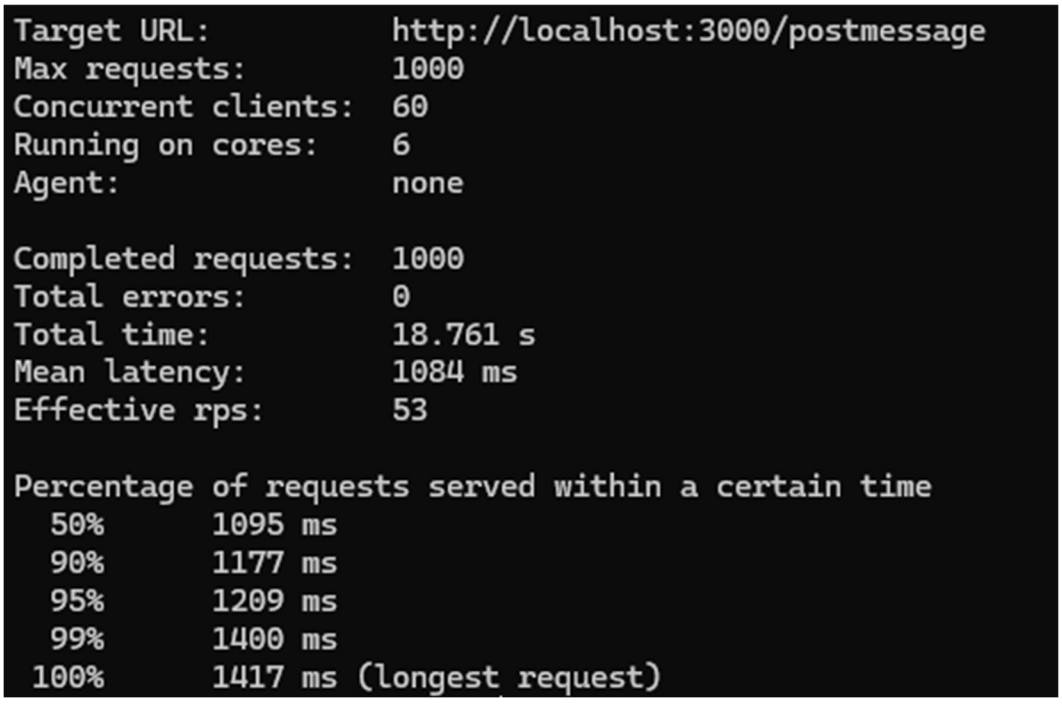
After: Pop up “Please fill out this field.” bubble next to field with missing data.

No other items are optional for submissions.

Loadtesting:

Command=> npx loadtest -c 10 -n 1000 -m POST -T "application/json" -P '{"topic":"test","data":"loadtest"}' http://localhost:3000/postmessage

Result:



No errors, but with 60 concurrent clients and 1000 total requests the server was able to manage 53 rps. Testing from 1 – 100 concurrent clients maintained approximately 50 rps. With –cores 1 and -c 1, 1000 requests dropped the mean latency to 20.2ms, while the overall time maintained 19.7s +/- 1s.