Assignment #3: User Service with Real Databases

Goal

In this assignment, we will update our existing user service to utilize real databases, adding two new features along the way:

- Expiring sessions that are preserved beyond the lifetime of the app server, via Redis
- User data that are preserved beyond the lifetime of the app server, via Mongo

Tasks

You should start with the Assignment 2 version of your user service, and make the following modifications:

- Remove the local in-code storage of users, in favor of storing in MongoDB.
  - You should use the **mongodb** package; if you want to use something else, please ask the instructor.
  - You will connect to a local MongoDB database, just like the lab, *and specify the database name* **assignment3** (don't just re-use the database we used for the lab!)
  - Your collection should be named for the function its used for - **users**
  - **Note:** Remember that Mongo operations are implemented with async/await. Therefore, you should **await** the result of the operation (i.e., await mongoClient.op()), and any function that includes an await – likely, your routes – should be decorated with **async** (i.e., async (req, res) => {}).
- Remove the local in-code storage of sessions, in favor of storing in Redis
  - You should use the **redis** (i.e. node-redis) package, which we used in the lab
  - You will connect to the localhost Redis server, under the default configuration.
  - You should scope your keys to sessions – perhaps something like **sessions:<sessionKey>**
- Implement a change in semantics for the Login API:
  - Sessions should expire after 10 seconds
    - Don't implement expiration yourself – **let Redis do it**! You utilized the EXPIRE function in the Redis lab.
  - A successful Login should remove any previous sessions for that user.
    - This requires you to store a lookup by user, **sessionsIdsByUserId:<userId>**. Redis doesn't have a "search for a value in a key" function (and it's not the right way to do that in redis, anyway). It's easy to just create the structures you need in redis and keep them up to date.
    - You can delete keys in Redis using the **del** command, which is like HTTP DELETE – it deletes the key if exists, but it's perfectly happy if the key already doesn't exist.
    - Note that each session must have a different key, even for the same user (per the unit tests)!

For this assignment, you may only add the redis and mongodb packages to your existing service. You won't need to add any more packages; In fact, you will likely not need the base64url package anymore. You could remove it with **npm uninstall base64url,** but that's not required.

You may not use connect-redis-sessions or any other package you may find online, that would "handles sessions for you" because where's the fun in that?

Testing

Your user service will be tested with a test suite that is similar to the one for Assignment 2, though several tests have been added to it, or changed. You will want to use the Assignment 3 collection, with 85 tests, not the Assignment 2 (75 test) collection.

The unit test grade is now all-or-nothing – you must pass all 85 tests, or you will receive 0/40 points for that section of the rubric.

Note that the API paths should not change – they're still **v1**!

Submission

You must update your primary code file to be named **assignment3.js** (or .mjs, etc)**.**

Copy your assignment 2 content to a new directory. You could copy in Linux or via Windows Explorer, or simply unzip your submission. You should NOT copy the node_modules directory or if you do, you should delete it.

1) **del package*.json** in the new directory
2) **mv assignment2.js assignment3.js** to rename the file.
3) npm init again, ensuring that you enter **assignment3.js** as the entry point.
4) **npm install packageName** for each package you're using – likely express, uuid4, redis, and mongodb.

Your submission should be named **yourAlias-CS261-3.zip**, where yourAlias is the username you use to log into your email and lab PC.

The submission should include:

- **assignment3.mjs (or js)**, the primary Javascript file (note the rename!)
- Your other Javascript files (modules)
- package.json
- package-lock.json

… without any subdirectories *(including node_modules).*

Rubric

40 Points: Unit tests *all* pass (85 total tests)

The Assignment 3 set of unit tests will be run against your server in a configuration that matches what you set up in Assignment 2. If **all 85 tests** pass, then you will receive 40 points **(otherwise 0 points).**

30 points: MongoDB used correctly for user data

If your service is using the correct MongoDB database for user data, without any strange issues, you will generally receive all 30 points. Not using MongoDB at all will receive 0 points. Partial credit may be given in some cases.

30 points: Redis used correctly for *expiring* sessions

If your service is using Redis for sessions that expire after 10 seconds, without any strange issues, you will generally receive all 30 points. Not using Redis at all will receive 0 points. Partial credit may be given in some cases.

Submission Penalties

In addition to the grading rubric above, you can receive these penalties if your submission is not correct:

- -5 if your submission files are not named as above.
- -5 if your submission includes additional files.

I may apply other submission penalties if needed.

Technical Notes

- If redis isn't running, start it up with **sudo service redis-server start**
- Don't forget to start MongoD as well!
- **Tip:** You should make the expiration duration (i.e., 10 seconds) a const at the top of one of your primary files. You could leave it at something long, like 10000 seconds, to aid in debugging of your Redis implementation. Then, when you need to test expiration (and for submission), you can change the value to 10.

Putter Test Descriptions

This is a summary of the tests that are run by Putter vs. your application for Assignment 3. But remember, the source of authority for the tests is the Postman JSON file itself! Always check that first for the most recent information.

Creating a user with POST on users/

Create Succeeds with a new user

- POST to /api/v1/users
- POST Body:
  { "username": "{{testUser}}","password": "{{testPassword}}","avatar": "{{testAvatar}}" }
- Tests:
  - Response Status must be 200
  - Response Body returns a User ID and correct username, password, avatar

Create Succeeds with a Different User

- POST to /api/v1/users
- POST Body:
  { "username": "{{otherUser}}","password": "{{otherPassword}}","avatar": "{{otherAvatar}}" }

- Tests:
    - Response Status must be 200
    - Response Body returns a new unique UserID and correct username, password, avatar

Create Fails with Duplicate User

- POST to /api/v1/users
- POST Body:
  { "username": "{{testUser}}","password": "{{testPassword}}","avatar": "{{testAvatar}}" }
- Tests:
    - Response Status must be 409

Login with POST to login/

Login Succeeds with First User

- POST to /api/v1/login
- POST Body:
  {"username": "{{testUser}}","password": "{{testPassword}}"}
- Tests:
    - Response Status is 200
    - Response Body returns a Session
    - UserID and SessionID aren't the same
    - UserID and Session are both valid

Login Succeeds with a Second Login with Different Session

- POST to /api/v1/login
- POST Body:
  {"username": "{{testUser}}","password": "{{testPassword}}"}
- Tests:
    - Response Status is 200
    - Session is valid and not the same as the previous session

Login Succeeds with a Second User with Different Session

- POST to /api/v1/login
- POST Body:
  {"username": "{{otherUser}}","password": "{{otherPassword}}"}
- Tests:
    - Response Status is 200
    - Session is valid and not the same as prior sessions

Login Fails with Bad Password

- POST to /api/v1/login
- POST Body:
  {"username": "{{testUser}}","password": "{{testPassword}}{{testPassword}}"}
- Note: request body is formatted so that there's no way that the password will be valid
- Tests:
    - Response Status is 403

Login Fails with Bad Username

- POST to /api/v1/login
- POST Body:
  {"username": "{{testUser}}{{testUser}}","password": "{{testPassword}}"}
- Note: request body is formatted so that there's no way that the username will be valid
- Tests:
  - Response Status is 400

Retrieve a user by ID with GET on users/:id

Retrieving the same user succeeds

- GET to /api/v1/users/{{userID}}
  - The userid is part of the route of the url
- GET Body:
  - It is unusual for a GET request to have a Body, but it is actually valid (and useful here)
  - {"session": "{{session}}"}
- Tests:
  - Response Status is 200
  - Returns correct user ID, username, password, avatar

Searching for a user by name with GET on users

Searching for Self Succeeds

- GET /api/v1/users?username={{testuser}}
  - In this case, the route is /api/v1/users and there is a QueryString parameter named "username" with the value {{testuser}}.
- GET Body: posts the session, similar as above
- Tests:
  - Response status is 200
  - Response includes correct username, userid, password, avatar

Searching for a Different User Succeeds

- GET /api/v1/users?username={{otherUser}}
  - Route is /api/v1/users, querystring parameter for rest, see above
- GET Body: includes session
- Tests:
  - Response Status is 200
  - Response includes correct username, userid, avatar
  - Response DOES NOT include password

Searching fails with no session

- GET /api/v1/users?username={{otherUser}}
- GET Body: ""
- Tests:
  - Response Status is 401

Searching fails with bad username (valid session)

- GET /api/v1/users?username=xx{{testUser}}xx
    - Note: the "xx" are there to ensure that the username variable won't match with any existing user.
    - You won't need to add any special handling for the "Xs", but should just treat it as input as usual
- GET Body: includes session
- Tests:
    - Response Status is 404

Searching Fails with Bad Username (Bad Session)

- GET /api/v1/users?username=xx{{testUser}}xx
    - Note: the "xx" are there to ensure that the username variable won't match with any existing user.
    - You won't need to add any special handling for the "Xs", but should just treat it as input as usual
- GET Body: Includes {{session}}{{session}} which will fail to match with anything
- Tests:
    - Response Status is 401

Searching fails with no username (valid session)

- GET /api/v1/users?username=xx{{testUser}}xx
    - Note: the "xx" are there to ensure that the username variable won't match with any existing user.
    - You won't need to add any special handling for the "Xs", but should just treat it as input as usual
- GET Body: Includes sessison
- Tests:
    - Response Status is 400

Updating a User by ID with PUT on users/:id

Updating Succeeds with new data

- PUT /api/v1/users/{{otheruserid}}
    - userID is part of the route
- PUT body
  {"session": "{{session}}",
  "username": "{{newUser}}",
  "password": "{{newPassword}}",
  "avatar": "{{newAvatar}}"}
- Tests:
    - Response Status is 200
    - Response returns new username, password, avatar

Updating fails on different user

- PUT /api/v1/users/{{userId}}
    - userID is part of the route
- PUT body
  {"session": "{{session}}",
  "username": "{{newUser}}",
  "password": "{{newPassword}}",
  "avatar": "{{newAvatar}}"}
- Tests:
    - Response Status is 403

Updating Fails with No Session

- PUT /api/v1/users/{{userid}}
- PUT Body: doesn't include session
- Tests:
    - Response Status is 401

Updating Fails with Bad Session

- PUT /api/v1/users/{{userid}}
- PUT Body: includes wrong session
- Tests:
    - Response code is 401

Updating Fails with Bad Username (valid session)

- PUT /api/v1/users/xx{{userId}}
    - The "x"s are there to make sure this won't find a valid userId by accident
- PUT Body: session, updates
- Tests:
    - Response Status is 404

Updating Fails with Bad Username (bad session)

- PUT /api/v1/users/xx{{userId}}
    - The "x"s are there to make sure this won't find a valid userId by accident
- PUT Body: invalid session, updates
- Tests:
    - Response Status is 401

Changes to First User Persist after failed calls

- GET /api/v1/users/{{userID}}
- GET body: valid session
- Tests:
    - Response Status is 200
    - Correct username, password, avatar are returned

No Changes to Other User after Failed Calls

- GET /api/v1/users/{{otherUser}}

- GET Body: Valid session
- Tests:
    - Response Status is 200
    - Correct Username, UserId, avatar are returned
    - Password is NOT returned

Expiring a Session

Authenticated request fails after more than 10 seconds from login

- GET /api/v1/users?username={{newUser}}
- GET Body: session
- Pre-Request: wait 10.5 seconds!
- Tests:
    - Response Status is 401

Re-Login with the first user succeeds again with a new session

- POST /api/v1/login
- POST Body: valid username, password
- Tests:
    - Response Status is 200
    - Response returns a session that does NOT match original session

Authenticated Request succeeds after fresh login

- GET /api/v1/users?username={{newUser}}
- GET Body: Session
- Tests:
    - Response Status is 200
    - Correct UserID is returned

Prep: Re-Login with the first user to invalidate the previous session

- POST /api/v1/login
- POST Body: valid username & password
- Tests:
    - Response status is 200
    - Response includes a session that doesn't match the previous sessions

Authenticated Request fails with an old valid session

- GET /api/v1/users?username={{username}}
- GET Body: old session
- Tests:
    - Response Status is 401