API Project Design Document

INTRODUCTION

The Product team has asked Engineering to build a new API designed for user management. This API has the following requirements:

1. Clients can save a new user when they provide an email, first name, and password.
    a. New users are set to “active” when they are created in the database.
2. Clients can retrieve active users by email and password.
3. Clients can deactivate users when they provide the ID.
    a. The API server must confirm that the user with the provided ID exists.
    b. The API server must deactivate the user with the provided ID before returning a response.

DESIGN OVERVIEW

Based on the product requirements, this API has three components:

- **Node.js.** This API must be written in JavaScript running server-side in Node.js.
- **Server Layer.** This API must be able to handle requests according to requirements 1-5 above.
- **Password Encryption Library.** Since one requirement of this project is to allow clients to retrieve Users by email and password, we must store the User’s password data. Seeing as it is unwise to store plain text passwords, an encryption library must be employed to hash User passwords before persisting to the database.
- **Database Layer.** This API must interact with database storage that persists and retrieves User data (at least while the server is running).

COMPONENTS AND SERVICES

Node.js

This API will employ Node.js 10.13.0, which is the latest LTS version as of the time of this writing. Code should employ ES6 syntax at minimum. ES2016-2018 features may be employed if their use increases efficiency or readability and does not conflict with other packages.

Server Layer

The server layer will be built using the latest version of Express.js (4.16.4 as of the time of this writing). The latest patch version of Express has been updated to remove deprecation warnings on Node.js 10. The latest Express changelog does not give any information about Express' compatibility with Node 11.

Password Encryption Library

For purposes of this API, node.bcrypt.js serves well as an encryption library. This library in particular offers both synchronous and asynchronous versions, with the async version using a thread pool so it does not block the main Node event loop. The documentation for this library explicitly recommends using version 3 and above with Node 10, so this project will use version 3.0.2 (the latest version as of the time of this writing).

Database Layer

The database layer of this project will actually be a pseudo-database. This project will use an in-memory implementation of MongoDB that will allow for testing and development, though no data will be persisted to disk. The latest version of NPM package mongodb-memory-server will be used for this purpose (version 2.7.0 as of the time of this writing).

DATA MODEL

Currently only one MongoDB "Users" collection is required to implement the API. Each User document must have at least the following properties, with the specified type:

```
{
  "first_name": "string",
  "email": "string",
  "password": "string",
  "active": boolean
}
```

The "password" field will be encrypted by the API server. As such, the value stored in the database will be a hashed value, and not the true value of the password.

MongoDB will assign an "_id" field to each document when it is created. Other fields may be included, but only the four fields above are necessary for a client to create a new User.

DATA FLOW

The data flow for this API will ensure that product requirements #1-3 are met by allowing Clients to make specifically structured POST and DELETE requests.

Requirement #1: Client Creates User

The client may create a User by making a POST request to the /users endpoint. The POST request should include a “Content-Type: application/json” header, and the body should be a singular JSON object containing the new user’s data. Per the product requirements, this object should include at least an email, first name, and password.

Emails must be unique among users. Any attempt to create a user with an email already belonging to another user will be unsuccessful. This constraint will be enforced by a unique index on “email.”

If “id” or “active” keys are included in the JSON body, they will be ignored.
The API will respond with an error if any of the following conditions occur:

- The request body does not include at least “email,” “first_name,” and “password” keys.
- The request body includes keys that are not stored on a user document.

Requirement #2: Client Retrieves User by Email and Password

The client may retrieve a User by email and password *only* by making a POST request to the /users endpoint. Like the create request, This POST request must include a “Content-Type: application/json” header.

Unlike the create request, this POST request must include a very specifically formed object as the body content. The request body *must* include a “user_query” key, the value of which is an object that includes the “email” and “password” keys, and nothing more:

```
{
  “user_query”: {
    “email”: “bob@bobsburgers.com”,
    “password”: “sooperSecret”
  }
}
```

The API will respond with an error if any of the following conditions occur:

- The request body includes anything *more* than the “user_query” key.
- The “user_query” object does not include *exactly* the “email” and “password” keys.

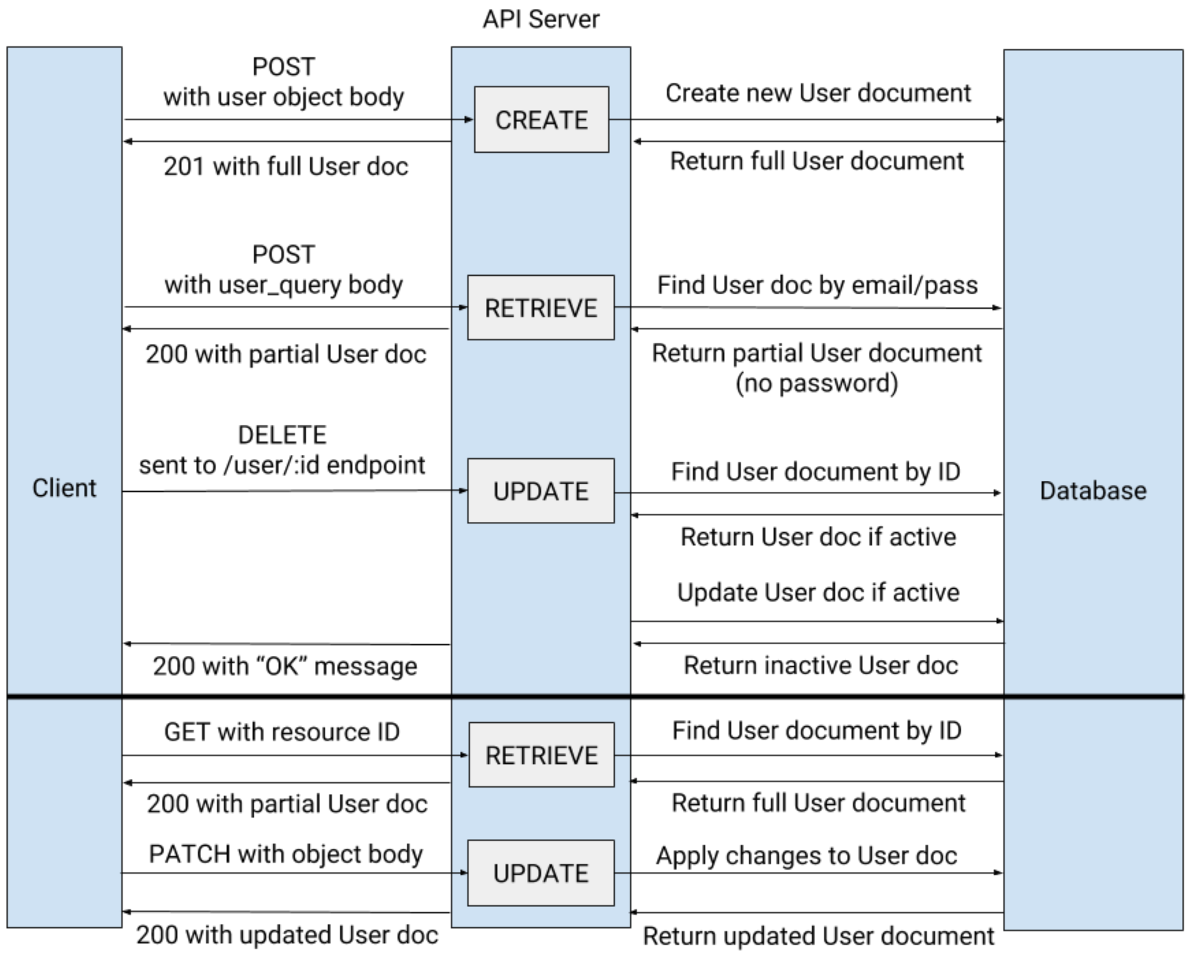
If the request body does not include the “user_query” key at all, then it will be treated as a create User request. See the preceding section for how these requests will be handled.

Requirement #3: Client Deactivates Specific User

The client may issue a request to deactivate a specific user, in which case they must provide the ID of the user to delete. This can be done by the client by sending a DELETE request to the "/users/:id" endpoint.

The API server will first look up the User by the provided ID. If the user exists, and if the user is active, the API server will update the User document to set "active" to false, then return a 200 response to the client with an empty body. If the user does not exist or is already set to be inactive (or both), the API server will respond with a 404.

Data Flow Chart


API Server
Client
Database
POST with user object body
CREATE
Create new User document
201 with full User doc
Return full User document
POST with user_query body
RETRIEVE
Find User doc by email/pass
200 with partial User doc
Return partial User document (no password)
DELETE sent to /user/:id endpoint
UPDATE
Find User document by ID
Return User doc if active
Update User doc if active
200 with "OK" message
Return inactive User doc
GET with resource ID
RETRIEVE
Find User document by ID
200 with partial User doc
Return full User document
PATCH with object body
UPDATE
Apply changes to User doc
200 with updated User doc
Return updated User document

TASKS

These tasks are somewhat fluid. The list can grow longer if each task needs to be broken out further, but generally these correspond to each step in the implementation process.

1. ~~Set up project package.json file with required packages. Ensure installation and configuration of Node v.10.13.0.~~ **COMPLETED.**
2. ~~Bootstrap Express project. Define routes for CRUD actions.~~ **COMPLETED.**
3. ~~Hook up project to Mongo memory server. Create validations/rules for Mongo User collection that will be applied by the Mongo memory server whenever the API server is spun up.~~ **COMPLETED.**
4. ~~Write handlers for User Create actions.~~ **COMPLETED.**
5. ~~Implement password hashing. Ensure correct encryption/decryption of User passwords.~~ **COMPLETED.**
6. ~~Write handlers for User Retrieve actions.~~ **COMPLETED.**
7. ~~Write handlers for User Deactivation actions.~~ **COMPLETED.**
8. ~~Write additional handlers to support standard CRUD actions (i.e., GET /user/:id).~~ **COMPLETED.**
9. ~~Ensure appropriate test coverage.~~ **COMPLETED.**
10. ~~README.md with all relevant information.~~ **COMPLETED.**