KABIR SINGH BHATIA

Node.js + Docker based Implementation

OVERVIEW

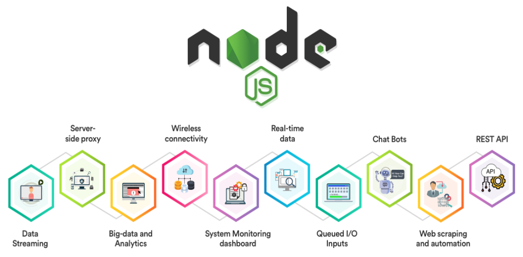
It is a RESTful Task Management API that allows authenticated users to perform CRUD (Create, Read, Update & Delete) operations on their tasks along with documentation/testing interface for developers.

Characteristics & Features

1. Made using Node.js, Express.js, MongoDB.
2. JWT (JSON Web Token) based authentication.
3. Made Documentation/Testing interface SwaggerUI
4. Docker support.
5. Separate volume in the local/host system for the database.
6. Can be used as a separate service in a larger application with some tweaks (Thanks to Docker).

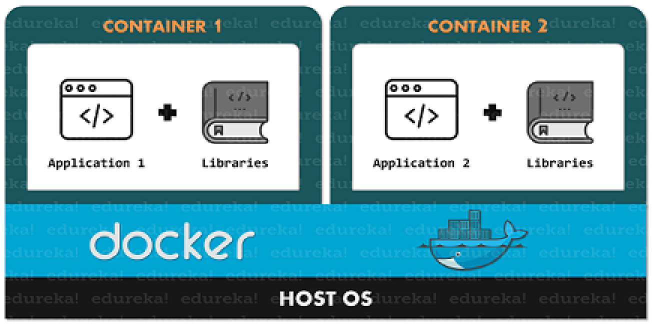
Technologies Used

Node.js



It is a runtime environment and a means for backend development and writing server-side code for web-applications. It can be used for applications such as system monitoring, proxies, REST API and many more

Docker



Docker is an open source platform for building, deploying, and managing containerized applications. It enables developers to package applications into containers that contain application source code with the operating system (OS) libraries and dependencies required to run that code in any environment. They can be used to isolate different application environments and link certain aspects of them while being lightweight at the same time. Docker can be used for integration and deployment of such containers into production environments and scale them very easily.

Requirements

1. Node.js / npm
2. MongoDB

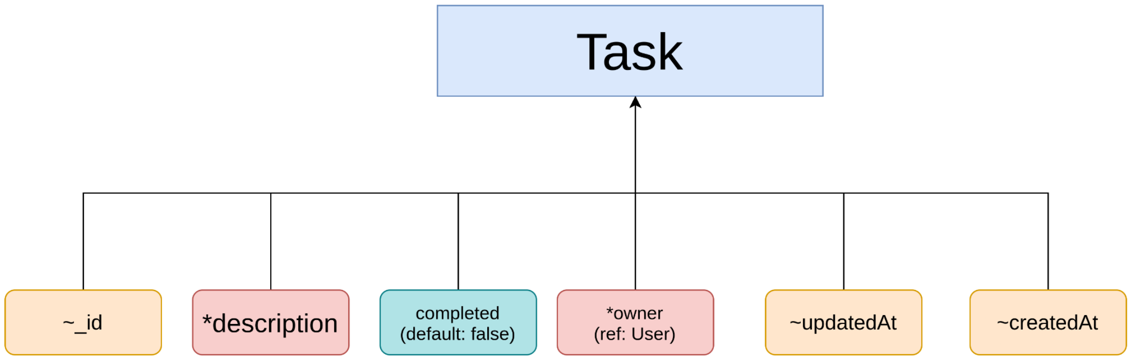
Above requirements are not needed if using Docker which you should.

Working

Models

Note: Fields marked with '~' are generated by the application and are filled automatically. Fields marked with '*' are the required fields. Every other field is optional and has some default value.

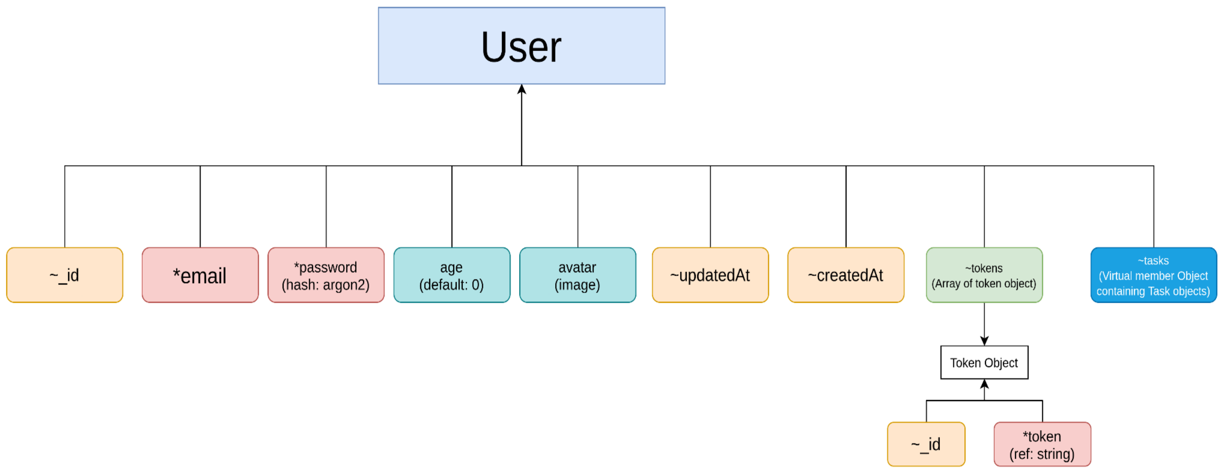
Task



A user can create, read, update and delete a task. Here is a short description of each field.

- _id: Task ID generated by MongoDB (ObjectId) in hexadecimal format.
- Description: It is the description of the task that is needed to be performed in string format.
- Completed: Completion status of the task in boolean.
- Owner: Owner ID of the user who owns this task (MongoDB ObjectId) in hexadecimal format. It is referenced to the user record using this field.
- createdAt: Date and time of creation of the task (type: Date Object)
- updatedAt: Date and time of updation of the task (type: Date Object)

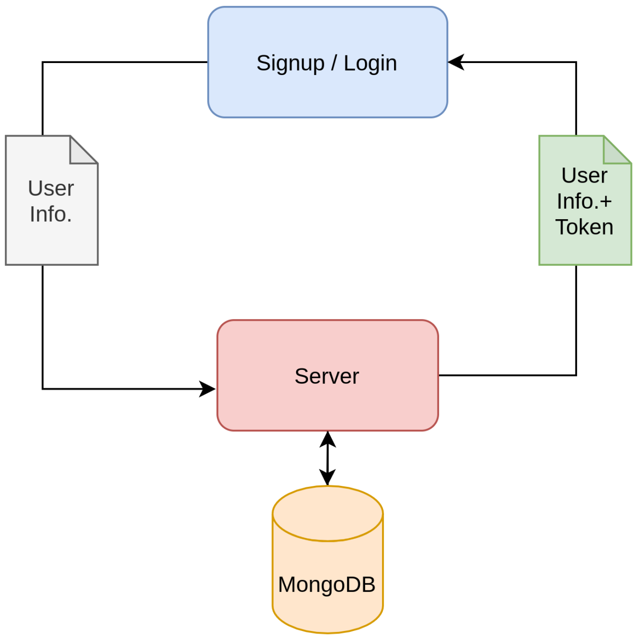
User



- _id: User ID generated by MongoDB (ObjectId) in hexadecimal format.
- email: Email address of the user in string format
- password: Password of the user hashed using argon2 hashing algorithm.
- age: Age of the user in integer format. (default value: 0)
- avatar: User’s avatar image accepted in jpg, jpeg, png format. Stored in dimension scale 250px x 250px.
- createdAt: Date and time of creation of the user (type: Date Object)
- updatedAt: Date and time of updation of the user (type: Date Object)
- tasks: Virtual object (calculated and not actual field) containing all the tasks owned by the user.

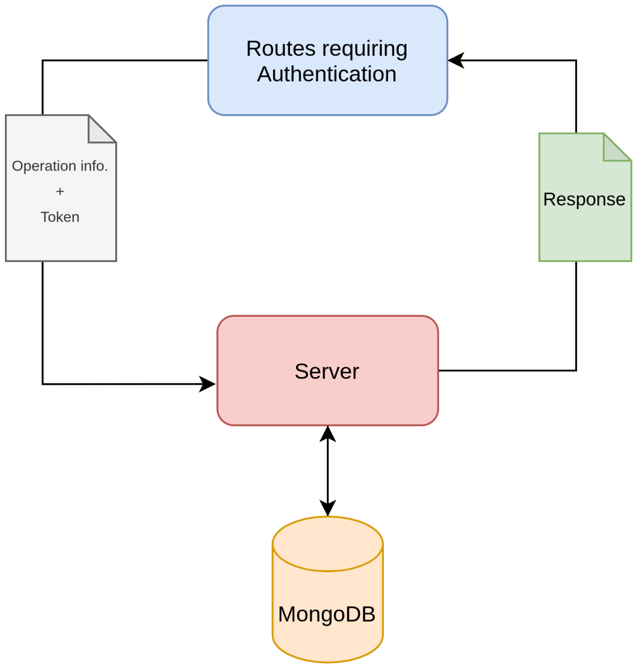
Routing

Authentication

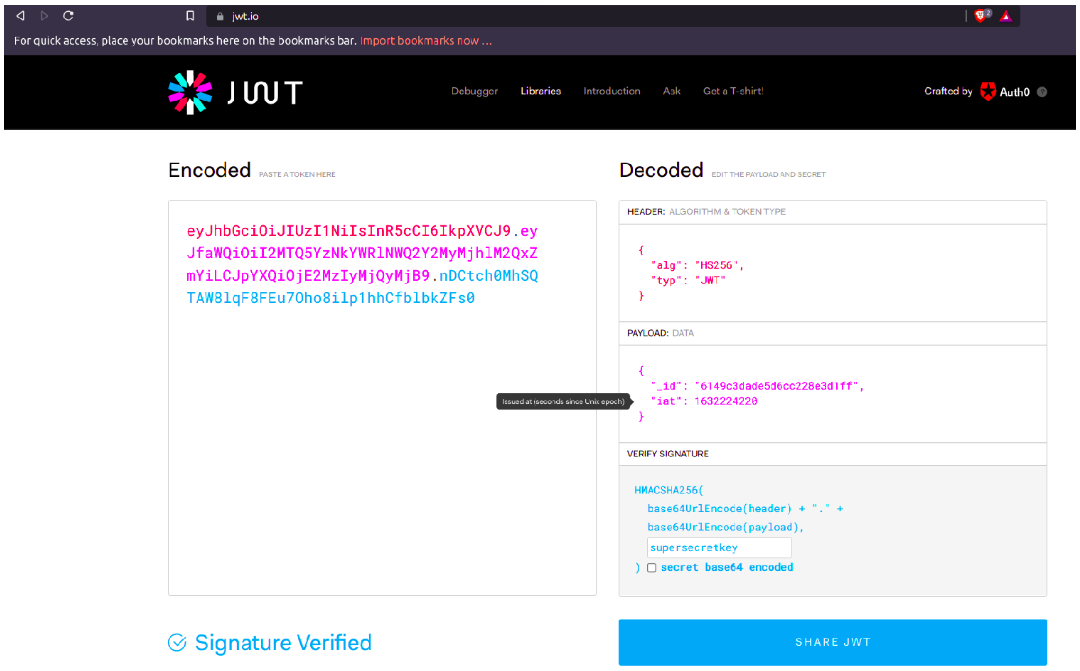


The client needs to send a request to the login/signup routes along with the credentials. After all the verification / validation by the server and MongoDB database, a JWT (JSON Web Token) is for the session is generated and added to the Tokens Array of the user object and the user object (without the Tokens Array) along with that token is sent as a response to the client. This token needs to be saved by the client side to perform operations on routes that require authentication.

Routes requiring Authentication



The JWT obtained by the client during the signup/login process should be saved by the client. This token should be set as the Bearer value of the request header each time the client wants to perform an operation using the routes that need authentication. This token after receiving by the API is decoded and verified by the token's signature.


jwt.io
For quick access, place your bookmarks here on the bookmarks bar. Import bookmarks now ...
JWT
Debugger
Libraries
Introduction
Ask
Get a T-shirt!
Crafted by Auth0

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JfaWQiOiI2MTQ5YzNkYWRlNWQ2Y2MyMjhlM2QxZ
mYiLCJpYXQiOjE2MzIyMjQyMjB9.nDCtch0MhSQ
TAW8lqF8FEu7Oho8ilp1hhCfblbkZFs0
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "_id": "6149c3dade5d6cc228e3d1ff",
  "iat": 1632224220
}
```

Issued at (seconds since Unix epoch)

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  supersecretkey
) ☐ secret base64 encoded
```

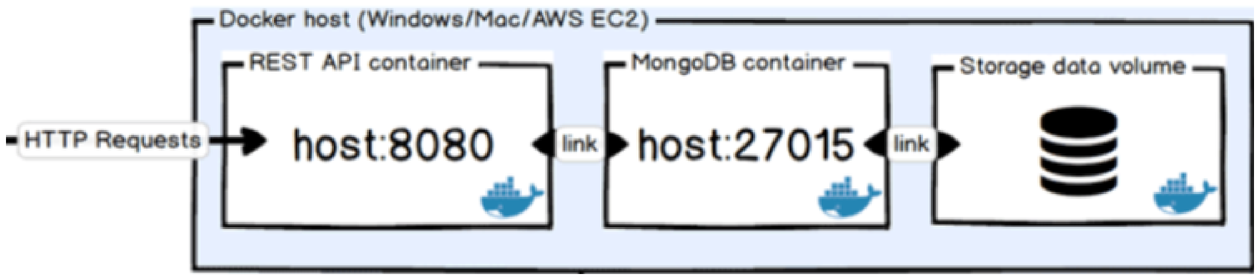
Signature Verified

SHARE JWT

The token has two properties which were set by the API's backend during the creation of the session. The are the user's id (_id) and the time the token was issued at [iat (seconds since Unix epoch)]. It is then checked in the database whether the user with this user id exists and whether this token is present in the user's tokens array (whether the session is valid or not). Only after everything is verified and valid will the user be able to use the routes that require authentication.

Further details of each route and its working is provided in the application's documentation/testing interface provided upon running the application.

Docker support



This application has been dockerized using three images:

1. mongo
2. mongo-express (optional, browser GUI for managing mongodb)
3. The REST API build itself (using node:alpine image)

Three containers are created and configured to communicate with each other using the docker network. The REST API backend container is accessible via port 8080 of the host system and mongo-express is accessible via port 8081. Both of these containers are connected to the mongo container to access the database. For persistence of the databases, a separate volume has been created for mongo which will remain on the host system even if the container goes down for any reason.

Usage of mongo-express image is optional. It allows the developer to manage the database from the web browser itself and can be commented out in the 'docker-compose.yaml' file. You can manage the database by exposing some port of the container and then manage it using a GUI client.