|  |  |  |  |
| --- | --- | --- | --- |
| API Name | URL Endpoint | Method | Input fields |
| API for creating a topic by admin from admin panel by adding topic URL and TITLE | api/v1/topics | POST | Title and URL |
| API for showing topics to students to submit their preference | api/v1/topics | GET | Title and URL |
| Update a topic once it is published to the Students or added into the Database | api/v1/topics/5de29b84b474a5290b5aa4c2 | UPDATE | Title and URL |
| Delete a topic once it is published to the Students or added into the Database | api/v1/topics/5de29b84b474a5290b5aa4c2 | DELETE |  |
| Create student preference | api/v1/preferences | POST | Student ID, Student Name, Preference 1,  Preference 2,  Preference 3,  Preference 4,  Preference 5 |
| Update student’s preference after it’s submission | api/v1/preferences/5dada2d4cd45eb99939aecc8 | UPDATE | Student ID, Student Name, Preference 1,  Preference 2,  Preference 3,  Preference 4,  Preference 5 |
| Get students preferences to run algorithm and allocate them into groups | api/v1/preferences | GET | Student ID, Student Name, Preference 1,  Preference 2,  Preference 3,  Preference 4,  Preference 5 |
| Creating a user(Student/ admin/ supervisor) | api/v1/users | POST | Email and password |
| Creating groups | api/v1/groups | POST | Name, Topic, Student 1,  Student 2,  Student 3,  Student 4,  Student 5 |
| Get all allocated groups | api/v1/groups | GET |  |
| Update the group with supervisor | api/v1/groups/5de3c9763270323561b9fb6f | UPDATE | Supervisor |
|  |  |  |  |

# Login module:

/\*\*

 \* TODO:

 \* Implement User LOGOUT

 \*/

'use strict';

/\*\*

 \* Module dependencies.

 \*/

const User = require('./../models/User');

const loginRouter = require('express').Router();

/\*\*

 \* User Login

 \*/

const login = async (req, res) => {

  try {

    const { email, password } = req.body;

    const userDetails = await User.findByCredentials(email, password);

    const token = await userDetails.generateAuthToken();

    res.send({ data: { user: userDetails.removeUnwantedFields(), token } });

    // res.send('Logged In');

  } catch (err) {

    res.status(400).send(JSON.stringify(err, ['stack'], 4));

  }

};

loginRouter.post('/login', login);

module.exports = loginRouter;

The login module recalls the user details for the already existing users from the database. Users will be required to type in their email and their password after which the try and catch block will be executed to validate the credentials. If both the email and the passwords match with the existing user details, then the user is allowed to log in successfully.

# Student (create):

const UserSchema = new mongoose.Schema(

  {

    email: {

      type: String,

      required: true,

      trim: true,

      minlength: 4,

      unique: true

    },

    password: {

      type: String,

      required: true,

      minlength: 6

    },

    role: {

      type: Number,

      minlength: 1,

      required: true

    }

  },

  { timestamps: true },

  { usePushEach: true }

);

The above snippet is the code that enables students to create their account and login after creating an account. Then all the data types that will be required are initialized and their conditions such as the minimum length of the data and whether it is required or it can be skipped are determined. For this case, the email is a string data type and only a min of four characters are allowed. The password on the other hand, must be unique for every user, a string data type, and only a minimum of six characters are accepted. The system uses timestamps to be able to know at what time the student logged in the system.

UserSchema.methods = {

  // Generating jwt after creating a user and after login

  async generateAuthToken() {

    try {

      const token = jwt.sign(

        { \_id: this.\_id.toHexString(), email: this.email, role: this.role },

        'secret\_guna',

        { expiresIn: '7d' }

      );

      return token;

    } catch (err) {

      throw err;

    }

  },

  removeUnwantedFields() {

    let { password, \_\_v, ...rest } = this.toObject();

    return rest;

  }

};

For the above code, a jwt is generated after creating an account and login in. After setting up the account with an email and a password, the above code uses a try, catch and throw method of error handling. As indicated earlier, the throw statement allows the program to notify the student using customized error messages. In addition, the try statement tests the code for errors while the catch statement handles it.

An if-else statement has been used in the above code to ensure that the code executed as long as the password and the email match with the data that is stored in the database. If the password and the email are collected, the student logs in the system successfully. The throw statement is constantly used in this particular segment of the code to ensure that the system remains user-friendly, thus improving usability[[1]](#footnote-1). When a student or any other user for that matter is notified of what the system is doing along the way, then they feel more comfortable while using the system.

# API for viewing approved topics

The admin is responsible for creating new topics, sending them to students for them to pick their preferred topics and then the admin approves those topics using computer algorithms. After the approval, the students need to view their approved topics.

This API will use a try and catch block as it is meant to give the approved topic to the student. The try statement will examine the code for any errors while the catch statement will handle any errors found.

If the admin had approved a topic for a specific student, the block would be true and thus, the student will be able to view their approved topics. Otherwise, the student will get an error message.

# 

# API for selecting preferences

const createPreference = async (req, res) => {

  try {

    //TODO: Bulk Pref insertions

    let pArray = data.map(pref => {

      const temp = new Preference(pref);

      return temp.save();

    });

    const r = await Promise.all(pArray);

    res.send(r);

    // let preferences = new Preference(req.body);

    // await preferences.save();

    // res.send(preferences);

  } catch (err) {

    logger.error(err.stack);

    res.status(400).send(err);

  }

};

The above snippet has the same code structure as the admin API for creating and viewing users. The code snippet uses module dependencies that are imported from the existing libraries. The second part of the code is where the student creates their preferences after which, a try and catch error handling method is used so that they don't choose a preference that they had chosen before. The catch statement is used to handle any errors that might be found when testing the code using the try block. After the student has created their preferences, they are able to query the system and get their preference details. At this point, the system also employs another try and catch block as a way of handling the errors that might arise.

const getPreferences = async (req, res) => {

  try {

    const {studentId} = req.query;

    const preferences = await Preference.find({studentId});

    res.send(preferences);

  } catch (err) {

    logger.error(err.stack);

    res.status(404).json(err);

  }

};

let preferences = await Preference.findOneAndUpdate(

      {studentId: id},

      req.body,

      {

        new: true

      }

    );

Following the creation and the viewing preferences, the students are now able to update their preference details while using the same API for selecting preferences. A try catch-and-throw block is used to help the student in dealing with any system error that might arise. In this case, the throw statement is used to notify the student when they make the wrong choice. For instance, when they input an invalid preferenceID. An if a statement is also used to execute the code as long as the student has inputted wrong details, the system will always give them the same message of invalid preference else the system will update the preference details.

After successfully updating the preference details, the API for selecting preferences also allows the students to delete these preferences in case they are not happy with their choices.

### 6.2.2.3 API for showing allocated topics and groups

const getGroups = async (req, res) => {

  try {

    // const {studentId} = req.query;

    // console.log(studentId);

    const groups = await Group.find({}).lean();

    res.send(groups);

  } catch (err) {

    logger.error(err.stack);

    res.status(404).json(err);

  }

};

The administrator is tasked with creating the topics, sending them to the students via an email notification, viewing the student's preferred selections, and then allocating these preferences to students by use of an algorithm.

Once a student has been allocated their preferred topic, then they can view them through the API for showing allocations. The API initializes the data types to be used, for instance, the data type to be used both strings and Boolean type. If the student has been allocated a topic, the system will return true and show them their allocated topic; else, the student will not be shown any topic. The timestamps have been used so that they can be able to monitor when the student saw the topic to avoid having students who might say that they had never seen their allocations.

## 6.2.3 Admin

The admin is responsible for overseeing the whole system as well as maintenance[[2]](#footnote-2). For this particular web application, the admin has roles such as adding topics, publishing approved users, viewing student’s topic preferences, and viewing and publishing topic allocations. The following sections explain the various APIs for the admin.

2 Charles C. Snow, Øystein Devik Fjeldstad, and Arthur M. Langer, “Designing the Digital Organization,” *Journal of Organization Design* 6, no. 1 (June 2017): 7, doi:10.1186/s41469-017-0017-y.

### 6.2.3.1 API for adding topics

In the first segment of the above code snippet, the topic is initialized and all the required libraries imported. JavaScript models are also used in reference to the libraries. The second part of the code is where the admin creates a new topic. At this point, the system uses a try and catch block to validate and make sure that only topics that are new can be added to the system. The last segment in the above snippet is the code that the admin uses when checking for topic details to ensure they have been added to the system successfully.

const createTopic = async (req, res) => {

  try {

    let topic = new Topic(req.body);

    await topic.save();

    res.send(topic);

  } catch (err) {

    logger.error(err.stack);

    res.status(400).send(err);

  }

};

const getTopics = async (req, res) => {

  try {

    let options = {active: true};

    if (

      req.headers['referer']

        ? req.headers['referer'].split('/').pop() === 'admin1.html'

        : true

    ) {

      options = {};

    }

    const topics = await Topic.find(options);

    res.send(topics);

  } catch (err) {

    logger.error(err.stack);

    res.status(404).json(err);

  }

};

This part of the API has two segments where the admin is able to both update and delete topics from the system. While updating the topics, the admin a try, throw, and catch block have been used the try block tests the code segment for any errors. The throw block allows the admin to receive customized error messages from the system. The same procedure has been used for the deleted segment of the code where a throw statement has been added to notify the admin when the select an invalid topicID. Additionally, the try and catch statement tests the code and handles it respectively. An if the statement has been used for both segments of the code and allows either segment to be executed as long as certain conditions hold true.

### 6.2.3.3 API for viewing preferences

 \*/

const getPreferences = async (req, res) => {

  try {

    const {studentId} = req.query;

    const preferences = await Preference.find({studentId});

    res.send(preferences);

  } catch (err) {

    logger.error(err.stack);

    res.status(404).json(err);

  }

};

The admin is responsible for creating new topics, sending them to students for them to pick their preferred topics, and then the admin approves those topics using a computer algorithm. After the approval, the admin needs to view these student’s preferred topics.

This API will use a try and catch block as it is meant to give the student preferred topics to the admin. The try statement will examine the code for any errors while the catch statement will handle any errors found.

If the admin has queried the system for preferences for a specific student, the block will be true, and thus the admin will be able to view their preferred topics. Otherwise, the admin will get an error message.

### 6.2.3.5 API for publishing allocations

This API helps the admin to publish the topics that the students have been allocated. The supervisor will then take over from when the publishing of allocation is done. The try statement is necessary to test this code segment in the API for publishing approved users for any possible errors. The catch block ensures that any of these errors that are identified when testing are well handled[[3]](#footnote-3). And finally, the throw statement ensures that the admin is able to receive personalized messages telling them of what the system is doing at any particular instance. Timestamps have been used to allow the admin to know the time when they published the approved users.

3 Elishai Ezra Tsur, “Rapid Development of Entity-Based Data Models for Bioinformatics with Persistence Object-Oriented Design and Structured Interfaces,” *BioData Mining* 10, no. 1 (March 2017): 11, doi:10.1186/s13040-017-0130-z.

## 6.2.4 Supervisor

The supervisor checks groups that the students have been allocated and, at the same time, needs to select one group from the list and submit it, which confirms that he/she is ready to supervise that particular group

1. Renee Garett et al., “A Literature Review: Website Design and User Engagement,” *Online Journal of Communication and Media Technologies* 6, no. 3 (July 2016): 1–14. [↑](#footnote-ref-1)
2. Charles C. Snow, Øystein Devik Fjeldstad, and Arthur M. Langer, “Designing the Digital Organization,” *Journal of Organization Design* 6, no. 1 (June 2017): 7, doi:10.1186/s41469-017-0017-y. [↑](#footnote-ref-2)
3. Elishai Ezra Tsur, “Rapid Development of Entity-Based Data Models for Bioinformatics with Persistence Object-Oriented Design and Structured Interfaces,” *BioData Mining* 10, no. 1 (March 2017): 11, doi:10.1186/s13040-017-0130-z. [↑](#footnote-ref-3)