

# Team 6 - Milestone #6

Andrew Wang, Keyu Lin, Lexie King, Shrey Bahadur

# Project Overview

Repository: [https://github.com/lhking1122/326\\_Course\\_Recommendation\\_System.git](https://github.com/lhking1122/326_Course_Recommendation_System.git)

Issues: [https://github.com/lhking1122/326\\_Course\\_Recommendation\\_System/issues](https://github.com/lhking1122/326_Course_Recommendation_System/issues)

Milestones:

Description: The user will input the courses they have already taken, and optionally any academic or career interests they have, and the app will recommend future courses they might be interested in. The recommendations will also consider prerequisite/eligibility requirements to ensure only available courses are suggested. A visual CS major course progress map will show completed courses, current courses, and suggested future courses to help students plan their academic path.

Key Features:

- User Input: Students can enter courses taken, interests, and career goals.
- Course Recommendation: The system recommends appropriate courses based on inputs and provides course details (e.g., prerequisites, credits, schedule, etc.).
- Course Schedule Builder: Students can select courses and generate schedules, the system will indicate time conflicts and provide suggestions for adjustments.
- Course description search up: Students can search up interested courses through the course details tab.
- Course Reviews: Students can look at and leave reviews for courses

# Roles

Lexie King:

- Role: Project Manager
- Issues: General UI Design, Past Course Storage, Course Details, Course Recommendation Home Page

Shrey Bahadur:

Role: Note Taker

Issues: Course Review Page, User login data,

Andrew Wang:

Role: Integration Manager

Issues: Ensuring UI, data structures, and the various pages are compatible with each other. Course Progress Chart page.

Keyu Lin:

Role: User Experience Tester

Issues: User Data Structure, Course Recommendation Home Page, User Experience Tester

# Historical Timeline

# Lexie - Assigned tasks & Pull Request

[https://github.com/voidpls/326\\_Course\\_Recommendation\\_System/pull/100](https://github.com/voidpls/326_Course_Recommendation_System/pull/100)

- Project manager
- General UI Design
- Course Recommendation Home Page UI
- Course Details Page UI
  - Course Data Structure
  - Course search within Course Details Page
- User Data Structure
- Past Course Storage
- Course Recommendation Algorithm
- Login Page
  - Login and create account features.

# Lexie - Completed Task Description

- Login-Page
  - Created login page and create account page using Multi-view.
  - Added a SQLite database (users.db) and users table for storing credentials
  - Installed and imported bcrypt to hash passwords securely
  - Created /api/signup endpoint that:
    - Checks if username/password are provided
    - Verifies the username isn't already in use
    - Hashes the password and inserts a new user record
  - Created /api/login endpoint that:
    - Looks up the hashed password by username
    - Uses bcrypt.compare to verify credentials
  - Returns JSON indicating success or a specific error message
  - Installed and configured express-session to track login state via a session cookie
  - Modified the root (/) route to:
    - Serve login.html when there's no valid session
    - Serve index.html when the user is already logged in
  - Wired up the front-end JavaScript to:
    - POST to /api/signup and /api/login with fetch
    - Display server-returned messages (alert(data.message))
    - Redirect to the app only on successful login

# Lexie - Features

Features Implemented:

User can Create Account (with unique-username check)

User can Log In (with bcrypt-protected passwords)

Guest Mode (stores a random ID in localStorage)

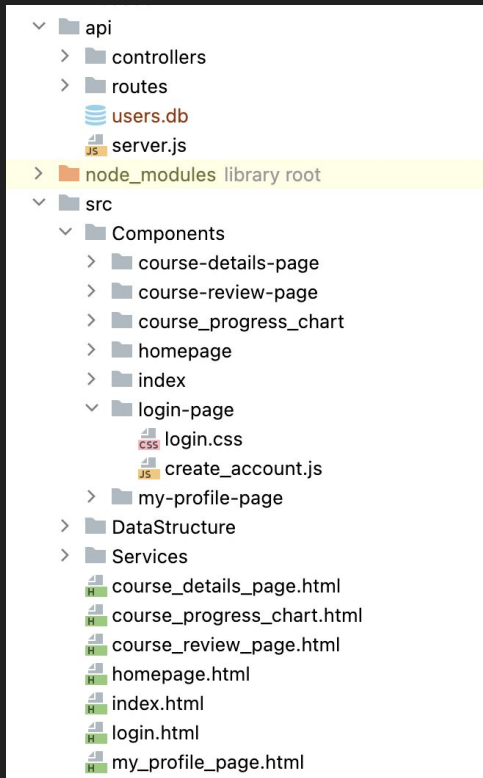
Completion Progress

Front-End: forms, view toggles, fetch calls, alerts, redirects

Back-End: /api/signup, /api/login, session handling, SQLite storage

Git Branch : login-page

# Lexie: Code Structure & Organization



## Separation of Concerns:

- Front-End lives entirely under /src (HTML, CSS, client JS)
- Back-End lives under /server (Express, SQLite, bcrypt, sessions)

## Key Components:

- login.html (UI + event listeners)
- server.js (API routes + middleware)
- routes/ (modular course-related endpoints)



# Lexie Work Summary - commits

Commits on Apr 25, 2025

created login page and create account page via multi-view. Implemented create account and login features, created api for signup and login, added a SQLite database (users.db) and users table for st...

d395c53



lhking2002 committed now · ✓ 1 / 1

# Lexie - Screen Shots of UI Implementation #1

## Login

lhking@umass.edu

\*\*\*\*\*

Login

Continue as Guest

Create Account

localhost:3000 says

Error: Username already taken.

OK

## Create Account

lhking@umass.edu

\*\*\*\*\*

Sign Up

Continue as Guest

Back to Login

# Lexie - Code Snippet and Explanation (Front-End)

```
// signup form
document.getElementById('signup-form')
  .addEventListener('submit', async e => {
    e.preventDefault();
    const username = document.getElementById('create-username').value;
    const password = document.getElementById('create-password').value;
    const res = await fetch('/api/signup', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ username, password })
    });
    const data = await res.json();
    if (data.success) {
      alert('Account created! You can now log in.');
```

```
      show('login');
```

```
    } else {
      alert('Error: ' + data.message);
    }
  });
```

## UI Impact

- Smooth toggle between Login / Create Account views
- Immediate feedback via alert(data.message)
- Redirect to index.html on successful login

## Integration

- Uses fetch to call Express API endpoints
- Relies on consistent URL paths (/api/signup, /api/login)

## Challenges & Solutions

- 404s from mismatched routes → aligned fetch URLs with server paths
- “undefined” errors → standardized { success, message } JSON

# Lexie - Code Snippet and Explanation (Back-End)

```
app.post( path: '/api/signup', handlers: (req : Request<P, ResBody, ReqBody, ReqQuery, LocalsObj> , res : Response<ResBody, LocalsObj>) => {
  const { username, password } = req.body;
  if (!username || !password) {
    return res.status( code: 400).json( body: {
      success: false,
      message: 'Username and password are required.'
    });
  }

  // Check if the username is already taken
  db.get(
    sql: 'SELECT 1 FROM users WHERE username = ?',
    params: [username],
    callback: (err : Error | null , row) : any | undefined => {
      if (err) {
        console.error('DB error on SELECT:', err);
        return res.status( code: 500).json( body: {
          success: false,
          message: 'Internal server error.'
        });
      }

      if (row) {
        // Already exists → bail out
        return res.status( code: 400).json( body: {
          success: false,
          message: 'Username already taken.'
        });
      }
    }
  );
});
```

## Structure

- server.js houses session setup, DB init, API routes
- SQLite (users.db) with users table for credentials
- express-session for login state (session cookie sid)

## Integration

- API routes consumed by front-end fetch calls
- Root route branches: serves login.html vs. index.html based on req.session.user

## Challenges & Solutions

- Static middleware auto-serving index → reordered routes / disabled index.html
- Missing bcrypt import → added const bcrypt = require("bcrypt")

# Lexie - Challenges and Insights

Implementing signup and login brought up a few tricky problems. Using bcrypt to hash passwords needed extra care in handling its asynchronous calls and making sure it was imported correctly—early mistakes caused errors nobody saw. Mismatches between the front-end fetch URLs and the Express routes (for example /signup vs. /api/signup) led to 404 “Not found” messages that were hard to trace. Express’s static-file setup also got in the way by serving index.html before our custom root route ran. Finally, getting sessions to work meant placing the session middleware in just the right spot and picking cookie settings that reliably kept users logged in.

From these problems we learned some simple but important lessons. Keeping the client’s fetch URLs and the server’s route paths exactly the same avoids a lot of bugs. The order you add middleware and routes in Express really matters—custom routes need to come before static or fallback handlers. Watching request and response logs in both the browser and server helps you spot mistakes quickly. And giving clear error messages, plus setting up password hashing and sessions from the start, makes the app easier to use and more secure.

Overall, our team had great communication for this milestone, Andrew was very helpful by setting up the API structure for us very early on, and was helpful in setting up vercel for us. I was caught in between 2 exams so I did not lead the team that much for this milestone, really appreciate the help and cooperation my teammates has given.

# Lexie - Future Improvements & Next Steps

I'll build a "forgot password" flow with emailed reset links, add a proper logout button, and replace alerts with inline form validation. Also Incorporate my-profile-page with user data and login page.

# Lexie - Links to Issues

[https://github.com/voidpls/326\\_Course\\_Recommendation\\_System/issues/89](https://github.com/voidpls/326_Course_Recommendation_System/issues/89)

[https://github.com/voidpls/326\\_Course\\_Recommendation\\_System/issues/90](https://github.com/voidpls/326_Course_Recommendation_System/issues/90)

# Andrew Work Summary

## For this milestone I did the following:

- Set up the backend express server along with a template for all the possible routes we would use
- Created GET, POST, and DELETE routes for the course progress selection page
  - GET /course-progress/userId: retrieve the saved user inputs and course list
  - POST /course-progress: update or create a user's inputs or course list
  - DELETE /course-progress/userId: delete a user's stored inputs and course list
- Implemented a basic in-memory store on the server to mimic retrieving/storing user info in a database
- Client side:
  - Implemented retrieval of saved user data upon loading the course progress page on the client
  - Implemented submission of user data to the server upon submitting the form

**ALL WORK DONE IN `course_selection` BRANCH**



# Andrew Work Summary

**The issues that I resolved for this milestone are:**

Set up express API boilerplate and route templates

[https://github.com/voidpls/326\\_Course\\_Recommendation\\_System/issues/83](https://github.com/voidpls/326_Course_Recommendation_System/issues/83)

Back end handling for course progress page

[https://github.com/voidpls/326\\_Course\\_Recommendation\\_System/issues/79](https://github.com/voidpls/326_Course_Recommendation_System/issues/79)

# Andrew Work Summary

## PRs Closed:

Create API route and file structure

[https://github.com/voidpls/326\\_Course\\_Recommendation\\_System/pull/87](https://github.com/voidpls/326_Course_Recommendation_System/pull/87)

Course progress GET and POST endpoints

[https://github.com/voidpls/326\\_Course\\_Recommendation\\_System/pull/93](https://github.com/voidpls/326_Course_Recommendation_System/pull/93)

Use in memory store for getting/setting user data

[https://github.com/voidpls/326\\_Course\\_Recommendation\\_System/pull/94](https://github.com/voidpls/326_Course_Recommendation_System/pull/94)

Course selection: Add DELETE route. Add more verbose error/success responses

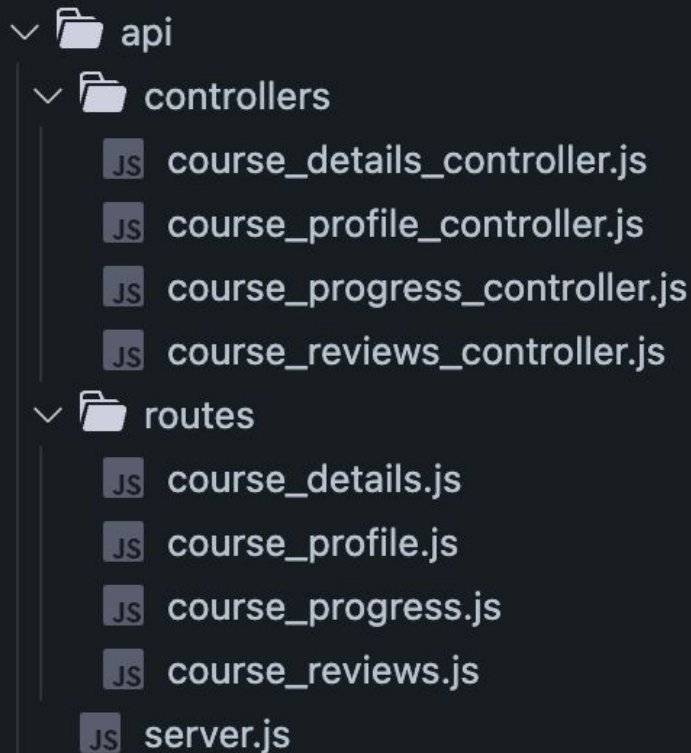
[https://github.com/voidpls/326\\_Course\\_Recommendation\\_System/pull/96](https://github.com/voidpls/326_Course_Recommendation_System/pull/96)

# Andrew Work Features

**Feature:** Express server and API boilerplate and setting up file structure.

Nothing complex, just set up the needed boilerplate for creating API endpoints. Express is also serving the static page itself, which we will be using instead of live server from now on. Referred to course material for organization.

```
app.use(express.static(path.join(__dirname, '../src')))  
app.use(express.json())  
  
app.get('/', (req, res) => {  
  res.sendFile(path.join(__dirname, '../src/index.html'))  
})
```



```
api  
├── controllers  
│   ├── course_details_controller.js  
│   ├── course_profile_controller.js  
│   ├── course_progress_controller.js  
│   └── course_reviews_controller.js  
├── routes  
│   ├── course_details.js  
│   ├── course_profile.js  
│   ├── course_progress.js  
│   └── course_reviews.js  
└── server.js
```

# Andrew Work Features

**Feature:** Course progress GET, POST, DELETE endpoints. Uses in-memory object for persistence (until server closes). User ID = 1 is dummy user with pre-initialized data on server side.

Demonstration (GET => POST => GET):

GET /course-progress/1

```
{"userId":1,"courseProgress":{"selectedIds":["MATH131","MATH132","MATH233_or_STAT315","MATH235","CICS110","CICS160","CICS210","CS240","CS250","CS220","CS_300_elective_1","CS320_or_CS326","CICS305"],"inputValues":{"CS_300_elective_1_input":"383"}},"courses":["MATH131","MATH132","MATH233_or_STAT315","MATH235","CICS110","CICS160","CICS210","CS240","CS250","CS220","CS_300_elective_1:383","CS320_or_CS326","CICS305"],"success":true}
```

POST /course-progress with body: {userId: 1, courseProgress: <updatedData>, courses: <updated data>}

► HTTP/1.1 200 OK (6 headers)

```
1 ▼ {}
2   "success": true
3   }
```

GET /course-progress/1 (retrieves updated data)

```
{"userId":1,"courseProgress":{"selectedIds":["MATH131","CICS110"]},"courses":["MATH131","CICS305"],"success":true}
```

# Andrew Work Features

Continued from last slide

Demonstration (GET => DELETE => GET):

GET /course-progress/1

```
{"userId":1,"courseProgress":{"selectedIds":["MATH131","CICS110"]},"courses":["MATH131","CICS305"],"success":true}
```

DELETE /course-progress/1

▶ HTTP/1.1 200 OK (6 headers)

```
1 ▼ {  
2   "success": true  
3 }
```

GET /course-progress/1 (retrieves non-existent data)

▶ HTTP/1.1 200 OK (6 headers)

```
1 ▼ {  
2   "success": false,  
3   "error": "This user ID does not exist"  
4 }
```

# Andrew Work Features

**Feature:** Client-side retrieval and submission of user data.

Upon initializing the course progress page, the client fetches the stored user data from the server. If this exists, the client can use this data for server-sided persistence (once a DB is implemented).

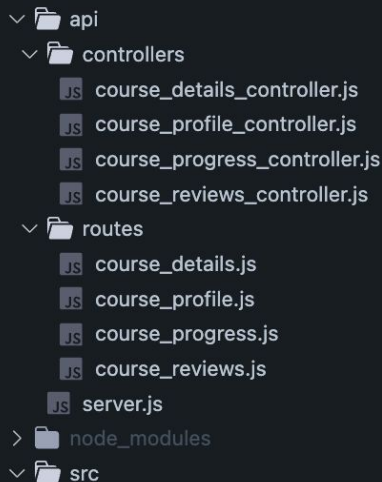
Then, upon submitting the form, a copy of the user inputs and course list is POSTed to the server. Currently, the server just logs the submitted data and updates the in-memory object with it.

What the server logs upon receiving a valid POST request:

```
{
  userId: 1,
  courseProgress: { selectedIds: [ 'MATH131', 'MATH132' ], inputValues: {} },
  courses: [ 'MATH131', 'MATH132' ]
}
```

# Andrew Work Code

**Directory structure:** `course_progress_controller.js` and `course_progress` are my relevant files.



```
├── api
│   ├── controllers
│   │   ├── course_details_controller.js
│   │   ├── course_profile_controller.js
│   │   ├── course_progress_controller.js
│   │   └── course_reviews_controller.js
│   ├── routes
│   │   ├── course_details.js
│   │   ├── course_profile.js
│   │   ├── course_progress.js
│   │   └── course_reviews.js
│   └── server.js
├── node_modules
└── src
```

Separation between frontend and backend is being maintained as all of our backend code is in `/api`, while our frontend/static code is in `/src`.

All files are prefixed with the page that they are made for, for easier maintenance of all of the routes.

# Andrew Work Front End Implementation

```
const rawRes = await fetch('/course-progress', {
  method: "POST",
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    userId: 1,
    courseProgress: formInputs,
    courses: selectedCourses
  })
})

const res = await rawRes.json()
console.log(res)

resultsList.innerHTML = resultsList.innerHTML +
  '<br><br>' +
  'Server response:<br>' +
  JSON.stringify(res)
```

This code is ran after the form is submitted. It sends a POST request to the server with the updated user data (in this case only for dummy user ID = 1).

Then, it receives the server response, and appends it to an element so that you can easily visualize and debug the POST request from the client side.

This code allows the server to receive the updated user data, and update the server-side in-memory store with the new data.

The biggest front-end challenge was converting from a local indexedDB-only model to a hybrid local/backend model. This requires redoing a lot of the logic from my local persistence implementation.



# Andrew Work Back End Implementation

```
exports.getProgress = async (req, res, next) => {  
  let { userId } = req.params;  
  try {  
    userId = parseInt(userId)  
    if (!memoryStore[userId])  
      return res.json({success: false, error: "This user ID does not exist"})  
  
    res.status(200).json({ ...memoryStore[userId], success: true } )  
  } catch (err) {  
    res.status(500).json({success: false, error: err})  
    next(err)  
  }  
}
```

This code is in `course_progress_controller.js`, and is what handles the GET `/course-progress/:userId` route.

It checks if the user has data stored in the in-memory store, and if it does, it sends the requested data back. If it doesn't exist, it sends back an error message.

The use of a `success: boolean` field in the response allows the client to easier discern whether or not it should use the server response, along with a verbose error message in all error/does-not-exist cases for client-side debugging.

The biggest challenge for back end implementation was debugging routes, as it is difficult to easily test many REST requests from the browser. I used HTTPie to easily generate REST requests with any route, body, and params that I wanted.

# Andrew Work Challenges and Insights

Because I already had some experience working with REST back ends and front ends, this milestone was not extremely challenging for me. However, I had never used controllers and routers before, only less organized API structures. The introduction of this added some complexity to implementing it for me, while also making me regret not trying it sooner, as it definitely makes organizing various components of the back end much easier.

A challenge of working in a team environment in this milestone was that we couldn't really start implementing a lot of features until we agreed and implemented some core implementation details, such as the API/route structure and the usage of user IDs and dummy user IDs for testing.

# Andrew Work Next Steps

We definitely need to implement a database primarily, so that I can implement backend persistence for the course details page, and so that other pages can use user course progress data.

This is easier said than done, as designing a database structure and structures for our stored data is a very very important step that requires a lot of forethought. We also need to weight the pros and cons of the possible databases we might use.

We also definitely need to implement authentication of some kind, as the majority of the (POST, PUT, DELETE) routes that we implemented are routes that should only be accessed by a specific, authenticated user.

# Shrey Work Summary

For this milestone I had to implement and integrate endpoints for the review part of our project. I created a GET, POST, and DELETE endpoint and then integrated it to the frontend. I also had to fix the way my page was being loaded in.

# Shrey - Commits and PRs and Issues

Commits:

fixed review page load



shreyBaha committed 4 hours ago

Merge pull request [#97](#) from voidpls/shrey/review-api



shreyBaha authored 6 minutes ago · ✓ 1 / 1

added endpoints and integrated



shreyBaha committed 7 minutes ago · ✓ 1 / 1

PR: [https://github.com/voidpls/326\\_Course\\_Recommendation\\_System/pull/97](https://github.com/voidpls/326_Course_Recommendation_System/pull/97)

Issues:

[https://github.com/voidpls/326\\_Course\\_Recommendation\\_System/issues/99](https://github.com/voidpls/326_Course_Recommendation_System/issues/99)

[https://github.com/voidpls/326\\_Course\\_Recommendation\\_System/issues/98](https://github.com/voidpls/326_Course_Recommendation_System/issues/98)

[https://github.com/voidpls/326\\_Course\\_Recommendation\\_System/issues/19](https://github.com/voidpls/326_Course_Recommendation_System/issues/19)

# Feature Implementation

The main feature I implemented was the routes for the review section of the website. The bones of each request is there and will be further fleshed out when we have an SQL database with a set structure we can make requests to. All work this milestone was done on the `shrey/review_api` branch

# Shrey - Code structure

```

  api
  |
  |__ controllers
  |   |__ course_details_controller.js
  |   |__ course_profile_controller.js
  |   |__ course_progress_controller.js
  |   |__ course_reviews_controller.js
  |
  |__ routes
  |   |__ course_details.js
  |   |__ course_profile.js
  |   |__ course_progress.js
  |   |__ course_reviews.js
  |
  |__ server.js
  |
  |__ node_modules
  |
  |__ src
  |   |__ Components
  |       |__ course_progress_chart
  |       |__ course-details-page
  |       |__ course-review-page
  |           |__ course_review_page.css
  |           |__ course_review_page.js
  |
  |__ .

```

This milestone I worked on controllers/course\_reviews\_controller.js, routes/course\_reviews.js, and course-review-page/course\_review\_page.js.

We have separated our backend into the api folder which then has the main server.js entrypoint along with the controllers and routes in their respective folders. Our frontend is in the src folder.

# Shrey - Code snippet

These are the routes

```
// // GET /course-reviews/reviews/:courseID
router.get('/reviews/:courseID', ctrl.getReviews);

// // POST /course-reviews/review
router.post('/review', ctrl.createReview);

// // PUT /course-reviews/:courseId
// router.put('/:courseId', ctrl.updateDetails);

// // DELETE /course-reviews/review/:reviewId
router.delete('/review/:reviewId', ctrl.deleteReview);
```

Here is some code from the controller

```
...
exports.getReviews = async (req, res, next) => {
  const { courseId } = req.params;
  try {
    res.json({courseId, reviewData});
  } catch (err) {
    next(err);
  }
};

exports.createReview = async (req, res, next) => {
  try {
    review = req.body;
    reviewData = {review};
    res.status(201).json({ message: 'Review created successfully', data: review });
  } catch (err) {
    next(err);
  }
};

exports.deleteReview = async (req, res, next) => {
  const { reviewId } = req.params;
  try {
    reviewData = {};
    res.json({ message: 'Profile deleted successfully', id: reviewId});
  } catch (err) {
    next(err);
  }
};
}
```

Below is some integration into the frontend

```
async function removeReviews(){
  let rawRes = await fetch('/course-reviews/review/1', { //1 is dummy val
    method: 'DELETE'
  })
  let res = await rawRes.json()
  console.log(res)
}

async function loadReviews(){
  let rawRes = await fetch('/course-reviews/reviews/1', { //1 is dummy val
    method: 'GET',
    headers: {
      'Accept': 'application/json'
    }
  })
  let res = await rawRes.json()
  console.log(res)
}
```



# Front End Implementation

This is some code showing the front end integration where I am calling the DELETE and GET requests from the frontend js file to execute their respective functions.

The biggest challenge was making sure everything was in place to switch from local storage to remote storage

```
async function removeReviews(){
  let rawRes = await fetch('/course-reviews/review/1', { //1 is dummy val
    method: 'DELETE'
  })
  let res = await rawRes.json()
  console.log(res)
}

async function loadReviews(){
  let rawRes = await fetch('/course-reviews/reviews/1', { //1 is dummy val
    method: "GET",
    headers: {
      'Accept': 'application/json'
    }
  })
  let res = await rawRes.json()
  console.log(res)
}
```

# Backend Implementation

To the right are the routes and respective controllers for those routes. These endpoints can be reached using fetch from the frontend to execute the logic within the controller.

The biggest challenge was understanding what was needed on the frontend and how the reflects in what routes are needed and what path variables/body structure is needed

```
// // GET /course-reviews/reviews/:courseID
router.get('/reviews/:courseID', ctrl.getReviews);

// // POST /course-reviews/review
router.post('/review', ctrl.createReview);

// // PUT /course-reviews/:courseId
// router.put('/:courseId', ctrl.updateDetails);

// // DELETE /course-reviews/review/:reviewId
router.delete('/review/:reviewId', ctrl.deleteReview);
```

```
exports.getReviews = async (req, res, next) => {
  const { courseId } = req.params;
  try {
    res.json({courseId, reviewData});
  } catch (err) {
    next(err);
  }
};

exports.createReview = async (req, res, next) => {
  try {
    review = req.body;
    reviewData = {review};
    res.status(201).json({ message: 'Review created successfully', data: review });
  } catch (err) {
    next(err);
  }
};

exports.deleteReview = async (req, res, next) => {
  const { reviewId } = req.params;
  try {
    reviewData = {};
    res.json({ message: 'Profile deleted successfully', id: reviewId});
  } catch (err) {
    next(err);
  }
};
```

# Shrey - Challenges and Key takeaways

The biggest challenge and takeaway was communicating with your team to get a set data structure in preparation for the SQL database while you don't have it. This is important to maintain consistency for when we switch over to a strict relational DB structure. Clear communication helped us progress.

# Shrey - Future improvements

In the future I need to flesh out the controllers more as well as some of the integration as well as set up the db structure. I will also need to add sequelize to the controllers.

[https://github.com/voidpls/326\\_Course\\_Recommendation\\_System/issues/103](https://github.com/voidpls/326_Course_Recommendation_System/issues/103)

[https://github.com/voidpls/326\\_Course\\_Recommendation\\_System/issues/102](https://github.com/voidpls/326_Course_Recommendation_System/issues/102)

# Keyu Work Summary

Added some details like checking user input.  
Implemented several methods for HTTP

GET /course-profile/:id to retrieve

POST /expenses/:id to create

PUT /expenses/:id to update

DELETE /expenses/:id to remove

# Keyu - Pull requests

[https://github.com/voidpls/326\\_Course\\_Recommendation\\_System/pull/80](https://github.com/voidpls/326_Course_Recommendation_System/pull/80)

[https://github.com/voidpls/326\\_Course\\_Recommendation\\_System/pull/81](https://github.com/voidpls/326_Course_Recommendation_System/pull/81)

[https://github.com/voidpls/326\\_Course\\_Recommendation\\_System/pull/91](https://github.com/voidpls/326_Course_Recommendation_System/pull/91)

# Keyu - Code snippet

```
exports.getProfile = async (req, res, next) => {
  const { userId } = req.params;
  try {
    if (userId === '1') { // Assuming userId is 1 for simplicity
      res.json({ userId, ...profileData }); // Return all profile data
    } else {
      res.status(404).json({ error: 'Profile not found' });
    }
  } catch (err) {
    next(err);
  }
};

exports.createDetails = async (req, res, next) => {
  try {
    profileData = req.body; // Save the profile data
    res.status(201).json({ message: 'Profile created successfully', data: profileData });
  } catch (err) {
    next(err);
  }
};

exports.updateDetails = async (req, res, next) => {
  try {
    profileData = { ...profileData, ...req.body }; // Update the profile data
    res.json({ message: 'Profile updated successfully', data: profileData });
  } catch (err) {
    next(err);
  }
};

exports.deleteDetails = async (req, res, next) => {
  try {
    profileData = {}; // Clear the profile data
```

```
const express = require('express');
const router = express.Router();
const ctrl = require('../controllers/course_profile_controller.js');

// GET /course-profile/:userId
router.get('/:userId', ctrl.getProfile);

// POST /course-profile
router.post('/', ctrl.createDetails);

// PUT /course-profile/:userId
router.put('/:userId', ctrl.updateDetails);

// DELETE /course-profile/:userId
router.delete('/:userId', ctrl.deleteDetails);

module.exports = router;
```

# Keyu - Implementation

Home Course Details Page My Course Progress Course Reviews My Profile

### Profile Summary

**Name:** keyu  
**Email:** 123@umass.edu  
**Phone:** 123456  
**Major:** Computer Science  
**Graduation Year:** 2025  
**Interests:** AI, Machine Learning  
**Preferred Contact:** Email

Logout

### Edit Profile

Name

keyu

Your Email

123@umass.edu

Your Phone

123456

Graduation Year

2025

Interests

AI Machine Learning

Preferred Contact Method

Email

Save Changes

localhost:3000

Profile saved successfully!

OK

http://localhost:3000/course-profile/1

JSON Raw Data Headers

Save Copy Collapse All Expand All Filter JSON

userId:

"1"

name:

"keyu"

email:

"123@umass.edu"

phone:

"123456"

gradYear:

"2025"

contact:

"Email"

interests:

"AI,Machine Learning"



# Keyu - Challenges

After changing to the API there were many problems with data storage and calls, such as data not being saved or read correctly. Due to incomplete understanding of the new knowledge, a lot of time spent in writing the code to test it over and over again, as well as part of the previous code was modified or rewritten from time to time.

# Keyu - Next steps

At present, the main functional modules of the project have been developed according to the plan and reached the expected goals. Next, we will organize internal communication within the team to evaluate the existing functions through collective discussion, focusing on the analysis of the optimization space and potential function expansion needs.