Appendix 2: Source Code

I. BACKEND FOR TEACHER ACCOUNT

```
const mongoose = require("mongoose");
const { teacherV2 } =
require("../../models/v2/teacher.schema");
const { classesV2 } =
require("../../models/v2/classes.schema");
const { studentV2 } =
require("../../models/v2/student.schema");
const bcrypt = require("bcrypt");

const teacherController = {
  all: async (req, res) => {
    try {
      let filter = {};

      const entry = await teacherV2.find(filter);

      return res.json({ status: true, data: entry });
    } catch (error) {
      console.log(error);
      return res.json({ status: false, error });
    }
  },
  paginate: async (req, res) => {
    const page = req.query.page || 1;

    try {
      const options = {
        sort: { createdAt: "desc" },
        page,
        limit: req.query.count || 10,
      };

      let query = {};

      if (req.query.search) {
        let regex = new RegExp(req.query.search, "i");
        query = {
          ...query,
          $and: [
            {
              $or: [{ fullname: regex }, { email: regex }],
            },
          ],
        };
      }

      const entry = await teacherV2.paginate(query,
options);

      return res.json({ status: true, data: entry });
    } catch (error) {
      console.log(error);
      return res.json({ status: false, error });
    }
  },
  view: async (req, res) => {
    try {
      const entry = await teacherV2.findOne({ _id:
req.params.id });

      if (!entry) throw "Teacher not found";

      return res.json({ status: true, data: entry });
    } catch (error) {
      console.log(error);

      return res.json({ status: false, error });
    }
  },
  create: async (req, res) => {
    try {
      const data = req.body;

      if (req.user.role != "ADMIN") throw "You are not an
admin";

      if (!data.password) throw "Password is required!";
      if (data.password != data.confirm_password) throw
"Password not match!";
      if (!data.email) throw "Email is required!";
      if (!data.fullname) throw "Username is required!";
      if (!data.firstname) throw "First Name is required!";
      if (!data.lastname) throw "Last Name is required!";
      if (!data.contact) throw "Contact is required!";

      // check if names are alpha only
      const isAlphaOnly = (str) => /^[a-zA-Z
]+$/.test(str);

      if (!isAlphaOnly(data.firstname))
        throw "First name can only contain letters!";
      if (!isAlphaOnly(data.lastname))
        throw "Last name can only contain letters!";

      if (data.middlename != "" &&
!isAlphaOnly(data.middlename))
        throw "Middle name can only contain letters!";

      // check if contact number is valid
      const isValidNumber = (number) =>
/^\d{9}$/.test(number);
      if (!isValidNumber(data.contact))
        throw "Contact number format is invalid!";

      // check if password is alpha numeric and between
8 to 20 characters
      if (data.password.length < 8 ||
data.password.length > 20)
        throw "Password must be between 8 and 20
characters!";

      const isAlphaNumeric = (str) => /^[a-zA-Z0-
9]+$/.test(str);
      if (!isAlphaNumeric(data.password))
        throw "Password can only contain letters and
numbers!";

      const password = await
bcrypt.hash(data.password, 10);

      const validateEmail = await teacherV2.findOne({
email: data.email });
      if (validateEmail) throw "Email is already taken.";

      const entry = await teacherV2.create({
```

```
      email: data.email,
      password,
      fullname: data.fullname,
      firstname: data.firstname,
      middlename: data.middlename || "",
      lastname: data.lastname,
      contact: data.contact,
    });

    if (!entry) throw "Teacher not created";
    return res.json({ status: true, data: entry });
   } catch (error) {
    console.log(error);
    return res.json({ status: false, error });
   }
  },
  update: async (req, res) => {
   try {
    const data = req.body;

    if (req.user.role != "ADMIN") throw "You are not an
admin";

    if (!data.email) throw "Email is required!";
    if (!data.fullname) throw "Username is required!";
    if (!data.firstname) throw "First Name is required!";
    if (!data.lastname) throw "Last Name is required!";
    if (!data.contact) throw "Contact is required!";
    if (!data.status) throw "Status is requried!";

    // check if names are alpha only
    const isAlphaOnly = (str) => /^[a-zA-Z
]+$/.test(str);

    if (!isAlphaOnly(data.firstname))
     throw "First name can only contain letters!";
    if (!isAlphaOnly(data.lastname))
     throw "Last name can only contain letters!";

    if (data.middlename != "" &&
!isAlphaOnly(data.middlename))
     throw "Middle name can only contain letters!";

    // check if contact number is valid
    const isValidNumber = (number) =>
/^\d{9}$/.test(number);
    if (!isValidNumber(data.contact))
     throw "Contact number format is invalid!";

    const entry = await teacherV2.findOneAndUpdate(
     { _id: req.params.id },
     {
      email: data.email,
      fullname: data.fullname,
      firstname: data.firstname,
      middlename: data.middlename || "",
      lastname: data.lastname,
      contact: data.contact,
      status: data.status,
     },
     { new: true, runValidators: true }
    );

    if (!entry) throw "Teacher not found";

    return res.json({ status: true, data: entry });
   } catch (error) {
    console.log(error);
    return res.json({ status: false, error });
   }
  },
  delete: async (req, res) => {
   try {
    if (req.user.role != "ADMIN") throw "You are not an

admin";
    const entry = await teacherV2.deleteOne({ _id:
req.params.id });
    return res.json({ status: true, data: entry });
   } catch (error) {
    console.log(error);
    return res.json({ status: false, error });
   }
  },
  approveStudent: async (req, res) => {
   try {
    const data = req.body;
    const user = req.user;

    if (user.role != "TEACHER") throw "You are not a
teacher";

    if (!data.student) throw "Student ID is required";
    if (!data.class) throw "Class ID is required";

    const student = await studentV2.findOne({ _id:
data.student });
    if (!student) throw "Student not found";

    const classs = await classesV2.findOne({ _id:
data.class });
    if (!classs) throw "Class not found";

    // check the teacher is the owner of the class
    if (classs.teacher != user.id)
     throw "You are not the teacher of this class";

    // check if the classId in the student matches the
classId in the request
    if (student.classId != data.class && student.classId
!= null)
     throw "Student is not in this class";

    // check the student is pending for this class
    if (student.classStatus == "JOINED")
     throw "Student is joined in this class";

    // update the student status to joined
    const updateStudent = await
studentV2.findOneAndUpdate(
     { _id: data.student },
     { classStatus: "JOINED", classId: data.class },
     { new: true }
    );

    if (!updateStudent) throw "Student not found";

    return res.json({ status: true, data: updateStudent
});
   } catch (error) {
    console.log(error);
    return res.json({ status: false, error });
   }
  },
```

```
  dropStudent: async (req, res) => {
    try {
      const data = req.body;
      const user = req.user;

      if (user.role != "TEACHER") throw "You are not a
teacher";


      if (!data.student) throw "Student ID is required";
      if (!data.class) throw "Class ID is required";

      const student = await studentV2.findOne({ _id:
data.student });
      if (!student) throw "Student not found";

      const classs = await classesV2.findOne({ _id:
data.class });
      if (!classs) throw "Class not found";

      // check the teacher is the owner of the class
      if (classs.teacher != user.id)
        throw "You are not the teacher of this class";

      // check if the classId in the student matches the
classId in the request
      if (student.classId != data.class && student.classId
!= null)
        throw "Student is not in this class";

      // remove the student from the class
      const updateStudent = await
studentV2.findOneAndUpdate(
        { _id: data.student },
        { classStatus: null, classId: null },
        { new: true }
      );

      return res.json({ status: true, data: updateStudent
});
    } catch (error) {
      console.log(error);
      return res.json({ status: false, error });
    }
  },
};

module.exports = teacherController;
```

II. BACKEND OF STUDENT ACCOUNT

```
const mongoose = require("mongoose");
const { classesV2 } =
require("../../models/v2/classes.schema");
const { studentV2 } =
require("../../models/v2/student.schema");
const { preTestResultV2 } =
require("../../models/v2/pretest_result.schema");
const { assesmentV2 } =
require("../../models/v2/assesment.schema");
const queryParser = require("../../helpers/query-
parser");

const bcrypt = require("bcrypt");
const {
  postTestAttemptV2,
} = require("../../models/v2/posttest_attempt.schema");

const studentController = {
  all: async (req, res) => {
    try {
      let filter = {};
      let classLists = [];
      if (req.query.class) {
        classLists =
queryParser.parseToArray(req.query.class);

        console.log(classLists);

        filter.classId = { $in: classLists };
      }

      const entry = await studentV2
        .find(filter)
        .populate("classId")
        .populate({
          path: "lessons",
          populate: {
            path: "lessonId",
          },
        });

      return res.json({ status: true, data: entry });
    } catch (error) {
      console.log(error);
      return res.json({ status: false, error });
    }
  },
  paginate: async (req, res) => {
    const page = req.query.page || 1;

    try {
      const options = {
        sort: { createdAt: "desc" },
        page,
        limit: req.query.count || 10,
        populate: [
          {
            path: "classId",
          },
          {
            path: "lessons",
            populate: {
              path: "lessonId",
            },
          },
        ],
      };

      let query = {};
      let classLists = [];

    if (req.query.class) {
      console.log(req.query.class);
      classLists =
queryParser.parseToArray(req.query.class);
      console.log(classLists);
      query.classId = { $in: classLists };
    }

     if (req.query.search) {
      let regex = new RegExp(req.query.search, "i");
      query = {
```

```
        ...query,
        $and: [
          {
            $or: [{ fullname: regex }, { email: regex }],
          },
        ],
      };
    }

    console.log(query);


    const entry = await studentV2.paginate(query,
options);


    return res.json({ status: true, data: entry });
  } catch (error) {
    console.log(error);
    return res.json({ status: false, error });
  }
 },
 view: async (req, res) => {
  try {
    const entry = await studentV2
      .findOne({ _id: req.params.id })
      .populate("classId")
      .populate({
        path: "lessons",
        populate: {
          path: "lessonId",
        },
      });

    if (!entry) throw "Student not found";

    return res.json({ status: true, data: entry });
  } catch (error) {
    console.log(error);
    return res.json({ status: false, error });
  }
 },
 create: async (req, res) => {
  try {
    const data = req.body;

    if (req.user.role != "ADMIN") throw "You are not an
admin";

    if (!data.password) throw "Password is required!";
    if (data.password != data.confirm_password) throw
"Password not match!";
    if (!data.email) throw "Email is required!";
    if (!data.fullname) throw "Username is required!";
    if (!data.firstname) throw "First Name is required!";
    if (!data.lastname) throw "Last Name is required!";
    if (!data.contact) throw "Contact is required!";

    // check if names are alpha only
      const isAlphaOnly = (str) => /^[a-zA-Z
]+$/.test(str);

      if (!isAlphaOnly(data.firstname))
        throw "First name can only contain letters!";
      if (!isAlphaOnly(data.lastname))
        throw "Last name can only contain letters!";

    if (data.middlename != "" &&
!isAlphaOnly(data.middlename))
      throw "Middle name can only contain letters!";

    // check if contact number is valid
    const isValidNumber = (number) =>
/^\d{9}$/.test(number);
    if (!isValidNumber(data.contact))
      throw "Contact number format is invalid!";


    // check if password is alpha numeric and between
8 to 20 characters
    if (data.password.length < 8 ||
data.password.length > 20)
      throw "Password must be between 8 and 20
characters!";

    const isAlphaNumeric = (str) => /^[a-zA-Z0-
9]+$/.test(str);
    if (!isAlphaNumeric(data.password))
      throw "Password can only contain letters and
numbers!";
    const password = await
bcrypt.hash(data.password, 10);

    const validateEmail = await studentV2.findOne({
email: data.email });
    if (validateEmail) throw "Email is already taken.";
    const entry = await studentV2.create({
      email: data.email,
      password,
      fullname: data.fullname,
      firstname: data.firstname,
      middlename: data.middlename || "",
      lastname: data.lastname,
      contact: data.contact,
    });
    return res.json({ status: true, data: entry });
  } catch (error) {
    console.log(error);
    return res.json({ status: false, error });
  }
 },
 update: async (req, res) => {
  try {
    const data = req.body;

    if (req.user.role != "ADMIN") throw "You are not an
admin";

    if (!data.email) throw "Email is required!";
    if (!data.fullname) throw "Username is required!";
    if (!data.firstname) throw "First Name is required!";
    if (!data.lastname) throw "Last Name is required!";
    if (!data.contact) throw "Contact is required!";
    if (!data.status) throw "Status is requried!";

    // check if names are alpha only
    const isAlphaOnly = (str) => /^[a-zA-Z
]+$/.test(str);

    if (!isAlphaOnly(data.firstname))
      throw "First name can only contain letters!";
    if (!isAlphaOnly(data.lastname))
      throw "Last name can only contain letters!";
```

```
    if (data.middlename != "" &&
!isAlphaOnly(data.middlename))
      throw "Middle name can only contain letters!";

    // check if contact number is valid
    const isValidNumber = (number) =>
/^\d{9}$/.test(number);
    if (!isValidNumber(data.contact))
      throw "Contact number format is invalid!";
    const entry = await studentV2.findOneAndUpdate(
      { _id: req.params.id },


  {email: data.email,
        fullname: data.fullname,
        firstname: data.firstname,
        middlename: data.middlename || "",
        lastname: data.lastname,
        contact: data.contact,
        status: data.status,
       },

      { new: true, runValidators: true }
    );

    if (!entry) throw "Student not found";

    return res.json({ status: true, data: entry });
  } catch (error) {
    console.log(error);
    return res.json({ status: false, error });
  }
 },
 delete: async (req, res) => {
  const session = await mongoose.startSession();

  try {
    session.startTransaction();

    if (req.user.role != "ADMIN") throw "You are not an
admin";

    const entry = await studentV2.deleteOne({ _id:
req.params.id });
    const pretests = await
preTestResultV2.deleteMany({
      studentId: req.params.id,
    });
    const posttests = await
postTestAttemptV2.deleteMany({
      studentId: req.params.id,
    });
    const assessments = await
assesmentV2.deleteMany({
      studentId: req.params.id,
    });

    await session.commitTransaction();
    return res.json({ status: true, data: entry });
  } catch (error) {
    console.log(error);
    await session.abortTransaction();
    return res.json({ status: false, error });
  } finally {
    session.endSession();
  }
 },
 joinClass: async (req, res) => {
  try {
    const data = req.body;
    const user = req.user;

    if (user.role != "STUDENT") throw "You are not a
student";

    if (!data.classcode) throw "Class Code is required!";

    // check if class exists
    const classExists = await classesV2.findOne({
      code: data.classcode,
    });


    if (!classExists) throw "Class not found";

    // check if student already has a class
    const studentExists = await studentV2.findOne({
      _id: user.id,
      classId: classExists._id,
    });
    if (studentExists) throw "You are already in this
class";

    // add student to class
    const entry = await studentV2
      .findOneAndUpdate(
        { _id: user.id },
        {
          classId: classExists._id,
          classStatus: "PENDING",
        },
        { new: true }
      )
      .populate("classId");
    if (!entry) throw "Error request not sent";

    return res.json({ status: true, data: entry });
  } catch (error) {
    console.log(error);
    return res.json({ status: false, error });
  }
 },
 leaveClass: async (req, res) => {
  try {
    const user = req.user;

    if (req.user.role != "STUDENT") throw "You are not a
student";

    const entry = await studentV2.findOneAndUpdate(
      { _id: user.id },
      {
        classId: null,
        classStatus: null,
      },
      { new: true }
    );

    if (!entry) throw "Error. class not left";
    return res.json({ status: true, data: entry });
  } catch (error) {
    console.log(error);
    return res.json({ status: false, error });
```

```
    }
  },
  changeLectureType: async (req, res) => {
    try {
      const data = req.body;
      const user = req.user;
      let lecType = null; // true = B, false = A

      if (user.role != "STUDENT") throw "You are not a
student";
      if (!data.concept) throw "Concept is required!";
      if (!data.type) {
        lecType = null;
      } else {
        lecType = data.type.toUpperCase() == "B" ? true :
false;
      }


    const entry = await studentV2
        .findOne({ _id: user.id })
        .populate("classId")
        .populate({
          path: "lessons",
          populate: {
            path: "lessonId",
          },
        });

      console.log(entry.lessons[0]);
      const lesson = entry.lessons.find((lesson) =>
        lesson.concepts.find((concept) =>
concept.conceptId == data.concept)
      );

      if (!lesson) throw "Lesson not found";

      const concept = lesson.concepts.find(
        (concept) => concept.conceptId == data.concept
      );

      if (!concept) throw "Concept not found";

      // lecttype is null toggle the value of the alterntive
lecture type
      if (lecType == null) {
        concept.alternateLecture =
!concept.alternateLecture;
      } else {
        concept.lectType = lecType;
      }

      const updateStudentlecture = await entry.save({
new: true });

      if (!entry) throw "Error lecture type not changed";

      return res.json({ status: true, data:
updateStudentlecture });
    } catch (error) {
      console.log(error);
      return res.json({ status: false, error });
    }
  },
};
module.exports = studentController;
```

III. BACKEND FOR ACCOUNT AUTHENTICATION

```
const { adminV2 } =
require("../../models/v2/admin.schema");
const { teacherV2 } =
require("../../models/v2/teacher.schema");
const { studentV2 } =
require("../../models/v2/student.schema");
const dateDifference =
require("../../helpers/dateDifference")
const sendMail = require("../../helpers/email.config")

const mongoose = require("mongoose");
const bcrypt = require("bcrypt");
const jwt = require("jsonwebtoken");

const timer = 300000 // 5 minutes
const BASE_URL = 'https://aid-gebra-
demo.herokuapp.com'

const authController = {
  login: async (req, res) => {
    try {
      if (req.islogged == true)
        throw "You are already logged in, Logout first to
change accounts";

      if (!req.body.email) throw "Email is required!";
      if (!req.body.password) throw "Password is
required!";
      if (!req.body.role) throw "Role is required!";

      const plainTextPassword = req.body.password;

      // Get The Role
      const role = req.body.role.toUpperCase();
      let entry = null;
      switch (role) {
        case "ADMIN":
          entry = await adminV2.findOne({ email:
req.body.email }).lean();
          break;
        case "TEACHER":
          entry = await teacherV2.findOne({ email:
req.body.email }).lean();
          break;
        case "STUDENT":
          entry = await studentV2.findOne({ email:
req.body.email }).lean();
          break;
        default:
          throw "Role is not valid!";
          break;
      }

      if (!entry) throw "Invalid credentials";
      if (entry.status != "ACTIVE") throw "Your account
is deactivated";
      if (role != "ADMIN" && !entry.isVerified) throw
"Please verify your email"

      if (await bcrypt.compare(plainTextPassword,
entry.password)) {
        const token = jwt.sign(
          {
```

```
        id: entry._id,
        email: entry.email,
        role: role,
        fullname: entry.fullname,
        firstname: entry.firstname,
        middlename: entry.middlename,
        lastname: entry.lastname,
        contact: entry.contact,
        avatar: entry.avatar,
        refreshToken: entry.refreshToken,
        status: entry.status,
       },
       process.env.JWT_SECRET
      );

      res.cookie("token", token, {
       httpOnly: true,
       // secure : true,


       // signed : true
      });
      console.log(token);
      return res.json({
       status: true,
       data: entry,
       token: token,
       role: role,
      });
     } else {
      return res.json({ status: false, error: "Invalid
Credentials" });
     }
    }

catch (error) {
     console.log(error);
     return res.json({ status: false, error });
    }
  },
  register: async (req, res) => {

    const session = await mongoose.startSession();

    try {
     session.startTransaction();
     const data = req.body;

     if (req.islogged == true)
      throw "You are already logged in, Logout first to
create a new account";

     if (!data.password) throw "Password is required!";
     if (data.password != data.confirm_password)
      throw "Password does not match!";
     if (!data.email) throw "Email is required!";
     if (!data.fullname) throw "fullname is required!";
     if (!data.firstname) throw "First Name is required!";
     if (!data.lastname) throw "Last Name is required!";
     if (!data.contact) throw "Contact is required!";

     // check if names are alpha only
     const isAlphaOnly = (str) => /^[a-zA-Z
]+$/.test(str);

     if (!isAlphaOnly(data.firstname))
      throw "First name can only contain letters!";

     if (!isAlphaOnly(data.lastname))
      throw "Last name can only contain letters!";

     if (data.middlename != "" &&
!isAlphaOnly(data.middlename))
      throw "Middle name can only contain letters!";

     // check if contact number is valid
     const isValidNumber = (number) =>
/^\d{9}$/.test(number);
     if (!isValidNumber(data.contact))
      throw "Contact number format is invalid!";

     // check if password is alpha numeric and between
8 to 20 characters
     if (data.password.length < 8 ||
data.password.length > 20)
      throw "Password must be between 8 and 20
characters!";

     const isAlphaNumeric = (str) => /^[a-zA-Z0-
9]+$/.test(str);
     if (!isAlphaNumeric(data.password))
      throw "Password can only contain letters and
numbers!";

     // Hash Password
     const password = await
bcrypt.hash(data.password, 10);

     // Format Data
     const userInfo = {
      email: data.email,
      password,
      fullname: data.fullname,


      firstname: data.firstname,
      middlename: data.middlename || "",
      lastname: data.lastname,
      contact: data.contact,
     };

     // Create User
     let entry = null;
     let entryRole = null;

     switch (req.body.role.toUpperCase()) {
      case "TEACHER":
       entryRole = await teacherV2.find({ email:
req.body.email }, null, {session}).lean();
       if (entryRole.length) throw "Email is already
taken.";

       entry = await teacherV2.create([userInfo],
{session});

       break;
      case "STUDENT":
       entryRole = await studentV2.find({ email:
req.body.email }, null, {session}).lean();
       if (entryRole.length) throw "Email is already
taken.";
       entry = await studentV2.create([userInfo],
{session});

       break;
```

```
      default:
        throw "Role is not valid!";
        break;
    }

    let info = await sendMail({
      to: entry[0].email, // list of receivers
      subject: "AidGebra Account Email Verification", //
Subject line
      html: `
        <center>
          <h1 style="margin-bottom:10px;">Welcome
to AidGebra!</h1>
          <p style="opacity:.8">To confirm your
registration with this email for an AidGebra account,
kindly click the button below:</p> <br/>

          <a
href="${BASE_URL}/verify/email?id=${entry[0]._id}&ro
le=${req.body.role.toUpperCase()}"
style="color:white;text-decoration:none;padding:15px
20px;border-radius:10px;background-color:#00203F">
            VERIFY


          </a>
        </center>
      `, // html body
    })

    if (!info.status) throw info.error

    let token = null;
    if (entry) {
      token = jwt.sign(
        {
          id: entry[0]._id,
          email: entry[0].email,
          role: req.body.role.toUpperCase(),
          fullname: entry[0].fullname,
          firstname: entry[0].firstname,

          middlename: entry[0].middlename,
          lastname: entry[0].lastname,
          contact: entry[0].contact,
          avatar: entry[0].avatar,
          refreshToken: entry.refreshToken,
          status: entry[0].status,
        },
        process.env.JWT_SECRET
      );

      res.cookie("token", token, {
        httpOnly: true,
        // secure : true,
        // signed : true
      });
    }

    await session.commitTransaction();
    return res.json({
      status: true,
      data: entry[0],
      token: token,
      role: req.body.role.toUpperCase(),
    });
  } catch (error) {
    console.log(error);
    await session.abortTransaction();
    return res.json({ status: false, error });
  } finally {
    session.endSession();
  }
},
verifyEmail : async (req, res) => {
  try{
    if(!req.query.role) throw "missing params"
    if(!req.query.id) throw "missing params"

    let entry = ''

    if(req.query.role == "TEACHER"){
      entry = await teacherV2.findOneAndUpdate(
        {_id : req.query.id},
        {isVerified : true},
        {new : true}
      )
    }


  else if(req.query.role == "STUDENT"){
       entry = await studentV2.findOneAndUpdate(
         {_id : req.query.id},
         {isVerified : true},
         {new : true}
       )
    }
    else {
      throw "Invalid"
    }

    return res.json({status: true, data :"Email is
verified"})
  }
  catch(error){
    console.log(error)
    return res.json({status : false, error})
  }
},

sendResetpasswordLink : async (req,res) => {
  try{
    if(!req.query.role) throw "missing params"
    if(!req.query.email) throw "Email is required"

    let info = await sendMail({
      to: req.query.email, // list of receivers
      subject: "Password reset on AidGebra account",
      html: `
        <center>
          <h1 style="margin-bottom:10px;">
                Reset Password
          </h1>
          <p style="opacity:.8">Kindly click the button
below to reset your password on your AidGebra
account.</p> <br/>
          <small style="opacity:.5; margin-top:5px;">if
you did not request for a password reset, please
disregard this email.</small> <br/><br/><br/>

          <a
href="${BASE_URL}/forgotpassword/change?email=$
```

```
{req.query.email}&role=${req.query.role.toUpperCase(
)}" style="color:white;text-
decoration:none;padding:15px 20px;border-
radius:10px;background-color:#00203F">
              Reset password
            </a>
          </center>
        `, // html body
      })

      if (!info.status) throw info.error

      return res.json({status: true, data : "Sent"})
    }
    catch(error){
      console.log(error)
      return res.json({status: false, error})
    }
  },
  forgotpassword : async (req, res) => {
    try{
      if(!req.query.role) throw "missing params"
      if(!req.query.email) throw "missing params"
      if(!req.body.password) throw "Password is
required"
      if(!req.body.repassword) throw "Please confirm
your


password"

      let entry = ''

      // check if password is alpha numeric and between
8 to 20 characters
      if (req.body.password.length < 8 ||
req.body.password.length > 20)
        throw "Password must be between 8 and 20
characters!";

      const isAlphaNumeric = (str) => /^[a-zA-Z0-
9]+$/.test(str);
      if (!isAlphaNumeric(req.body.password))
        throw "Password can only contain letters and
numbers!";

      const password = await
bcrypt.hash(req.body.password, 10);


      if(req.query.role == "TEACHER"){
        entry = await teacherV2.findOneAndUpdate(
          {email : req.query.email},
          {password},
          {new : true}
        )
      }
      else if(req.query.role == "STUDENT"){
        entry = await studentV2.findOneAndUpdate(
          {email : req.query.email},
          {password},
          {new : true}
        )
      }
      else {
        throw "Invalid"
      }

      return res.json({status: true, data : entry})
    }
    catch(error){
      console.log(error)
      return res.json({status: false, error})
    }
  }
};

module.exports = authController;
```

IV. BACKEND FOR THE IDENTIFICATION OF KNOWLEDGE LEVEL

```
    // count the total score
    let totalScore = 0;
    questionsList.forEach((question) => {
      if (question.isCorrect) totalScore++;
    });
    resultData.totalScore = totalScore;

    // calculate the mastery
    let correctAnswerTracker = {
      concept1: 0,


      concept2: 0,
      concept3: 0,
      concept4: 0,
      concept5: 0,
    };

    questionsList.forEach((question) => {
      let conceptOrder = null;

      switch (question.order - 1) {
        case 0:
        case 1:
        case 2:
        case 3:
          conceptOrder = 1;
          if (question.isCorrect)
correctAnswerTracker.concept1++;
          break;
        case 4:
        case 5:
        case 6:
        case 7:
          conceptOrder = 2;
          if (question.isCorrect)
correctAnswerTracker.concept2++;
          break;
        case 8:
        case 9:
        case 10:
        case 11:
          conceptOrder = 3;
          if (question.isCorrect)
correctAnswerTracker.concept3++;
          break;
        case 12:
        case 13:
        case 14:
        case 15:
```

```
          conceptOrder = 4;
          if (question.isCorrect)
correctAnswerTracker.concept4++;
          break;
        case 16:
        case 17:
        case 18:
        case 19:
          conceptOrder = 5;
          if (question.isCorrect)
correctAnswerTracker.concept5++;
          break;
        default:
          console.log(question.order);
          throw "Question is not in concept error";
      }
    });
    const conceptList = [];

    concepts.forEach((concept, i) => {
      let con = {
        conceptName: concept.name,
        correctAnswers:
correctAnswerTracker[`concept${i + 1}`],
        mastery:
          correctAnswerTracker[`concept${i + 1}`] >= 3



            ? "MASTERED"
            : "UNMASTERED",
      };
      conceptList.push(con);
    });
    resultData.result = conceptList;

    console.log(
      '"----------RESULTS CREATED----------"'
    );
    console.log(conceptList);
    console.log(
      '"----------RESULTS CREATED----------"'
    );
    // calculate the knowledge level
    let knowledgeLevel = null;

    // turn values of correctAnswerTracker into an
array
    const correctAnswerTrackerArray =
Object.values(correctAnswerTracker);

    // check if all concepts are mastered
    const numberOfMastered =
correctAnswerTrackerArray.filter(
      (concept) => concept >= 3
    ).
length;
    switch (numberOfMastered) {
      case 0:
        knowledgeLevel = "POOR";
        break;
      case 1:
        knowledgeLevel = "FAIR";
        break;
      case 2:
        knowledgeLevel = "AVERAGE";
        break;
      case 3:
        knowledgeLevel = "GOOD";
        break;
      case 4:
        knowledgeLevel = "VERY GOOD";
        break;
      case 5:
        knowledgeLevel = "EXCELLENT";
        break;
      default:
        throw "Knowledge level error";
    }
    resultData.knowledgeLevel = knowledgeLevel;

    // create the pretest result
    const pretestResult = await
preTestResultV2.create(resultData);
    console.log() };
```

V. BACKEND FOR THE LESSON

```
const mongoose = require("mongoose");

const { lessonV2 } =
require("../../models/v2/lesson.schema");
const { preTestV2 } =
require("../../models/v2/pretest.schema");

const { postTestV2 } =
require("../../models/v2/posttest.schema");

const lessonController = {
  all: async (req, res) => {
    try {
      let filter = {};

      const entry = await lessonV2.find(filter).sort({
order: "asc" });

      return res.json({ status: true, data: entry });
    } catch (error) {
      console.log(error);
      return res.json({ status: false, error });
    }
  },
  paginate: async (req, res) => {
    const page = req.query.page || 1;

    try {
      const options = {
        sort: { order: "asc" },
        page,
        limit: req.query.count || 10,
      };
      let query = {};

      if (req.query.search) {
        let regex = new RegExp(req.query.search, "i");
        query = {
          ...query,
          $and: [
            {
              $or: [{ name: regex }],
            },
          ],
        };
```

```
    }
    const entry = await lessonV2.paginate(query,
options);

    return res.json({ status: true, data: entry });
   } catch (error) {
    console.log(error);
    return res.json({ status: false, error });
   }
  },
  view: async (req, res) => {
   try {
    const entry = await lessonV2.findOne({ _id:
req.params.id });

    if (!entry) throw "Lesson not found";

    return res.json({ status: true, data: entry });
   } catch (error) {
    console.log(error);
    return res.json({ status: false, error });
   }
  },
  create: async (req, res) => {
   try {
    const data = req.body;


    // check if allowed
    const allowedRoles = ["ADMIN"];
    if (!allowedRoles.includes(req.user.role)) {
     throw "You are not allowed to do this";
    }

    if (!data.order) throw "Order is required!";
    if (!data.name) throw "Name is required!";

    // check if lesson with same order already exists
    const existing = await lessonV2.findOne({ order:
data.order });
    if (existing) throw "Lesson with same order already
exists!";

    const entry = await lessonV2.create([
     {
      order: data.order,
      name: data.name,
     },
    ]);
    if (!entry) throw "Lesson not created";

    // create pretest of lesson
    const pretest = await preTestV2.create([
     {
      lessonId: entry[0]._id,
     },
    ]);


  if (!pretest) throw "Pretest not created";

    // create posttest of lesson
    const posttest = await postTestV2.create([
     {
      lessonId: entry[0]._id,
     },
    ]);
    if (!posttest) throw "Posttest not created";

    return res.json({ status: true, data: entry });
   } catch (error) {
    console.log(error);
    return res.json({ status: false, error });
   }
  },
  update: async (req, res) => {
   try {
    const data = req.body;

    // check if allowed
    const allowedRoles = ["ADMIN"];
    if (!allowedRoles.includes(req.user.role)) {
     throw "You are not allowed to do this";
    }

    if (!data.name) throw "Name is required!";
    if (!data.order) throw "Order is required!";

    // check if lesson with same order already exists
    const existing = await lessonV2.findOne({ order:
data.order });


   if (existing && existing._id !== data._id)
     throw "Lesson with same order already exists!";

    const entry = await lessonV2.findOneAndUpdate(
     { _id: req.params.id },
     {
      name: data.name,
      order: data.order,
     },
     { new: true }
    );
    if (!entry) throw "Lesson not found";

    return res.json({ status: true, data: entry });
   } catch (error) {
    console.log(error);
    return res.json({ status: false, error });
   }
  },
  delete: async (req, res) => {
   try {
    if (req.user.role != "ADMIN") throw "You are not an
admin";
    const entry = await lessonV2.deleteOne({ _id:
req.params.id });
    return res.json({ status: true, data: entry });
   } catch (error) {
    console.log(error);
    return res.json({ status: false, error });
   }
  },
};
module.exports = lessonController;
```

VI. BACKEND FOR ASSESSMENT SESSION

```
const { assesmentV2 } =
require("../../models/v2/assesment.schema");
const { conceptV2 } =
require("../../models/v2/concept.schema");
const {
  conceptQuestionV2,
} =
require("../../models/v2/concept_question.schema");
const { studentV2 } =
require("../../models/v2/student.schema");

const assesmentController = {
  all: async (req, res) => {
    try {
      let filter = {};

      if (req.query.concept) {
        filter.conceptId = req.query.concept;
      }

      if (req.query.student) {
        filter.studentId = req.query.student;
      }

      const entry = await
assesmentV2.find(filter).populate("conceptId");


      return res.json({ status: true, data: entry });
    } catch (error) {
      console.log(error);
      return res.json({ status: false, error });
    }
  },
  paginate: async (req, res) => {
    const page = req.query.page || 1;

    try {
      const options = {
        sort: { createdAt: "desc" },
        page,
        limit: req.query.count || 10,
        populate: ["conceptId"],
      };

      let query = {};

      if (req.query.concept) {
        query.conceptId = req.query.concept;
      }

      if (req.query.student) {
        query.studentId = req.query.student;
      }

      if (req.query.search) {
        let regex = new RegExp(req.query.search, "i");
        query = {
          ...query,
          $and: [

{
            $or: [{ uuid: regex }],
          },
        ],
      };
    }

    const entry = await assesmentV2.paginate(query,
options);

    return res.json({ status: true, data: entry });
  } catch (error) {
    console.log(error);
    return res.json({ status: false, error });
  }
},
view: async (req, res) => {
  try {
    const entry = await assesmentV2
      .findOne({ _id: req.params.id })
      .populate("conceptId");

    if (!entry) throw "Assesment session not found";

    return res.json({ status: true, data: entry });
  } catch (error) {
    console.log(error);
    return res.json({ status: false, error });
  }
},



create: async (req, res) => {
  try {
    const data = req.body;
    const user = req.user;
    // check if allowed
    const allowedRoles = ["STUDENT"];
    if (!allowedRoles.includes(user.role)) {
      throw "Only students can create assesment
sessions";
    }
    if (!data.concept) throw "Concept is required";

    // check if concept exists
    const concept = await conceptV2.findOne({ _id:
data.concept });
    if (!concept) throw "Concept not found";

    // create the session
    const assesmentUuid =
require("crypto").randomUUID();
    const entry = await assesmentV2.create([
      {
        uuid: assesmentUuid,
        conceptId: concept._id,
        studentId: user.id,
        conceptName: concept.name,
        status: "INCOMPLETE",
        results: [],
      },
    ]);
    if (!entry) throw "Assesment not created";
    return res.json({ status: true, data: entry[0] });
  } catch (error) {
    console.log(error);
    return res.json({ status: false, error });
  }
```

```
  },

  update: async (req, res) => {
    try {
      const data = req.body;
      const user = req.user;

      // check if allowed
      const allowedRoles = ["STUDENT"];
      if (!allowedRoles.includes(user.role)) {
        throw "Only students can update assesment
sessions";
      }

      if (!req.params.uuid) throw "UUID is required";
      if (!data.question) throw "Question Id is required";
      if (!data.answer) throw "Answer is required";

      // check if assesment exists
      const assesment = await assesmentV2.findOne({
uuid: req.params.uuid });
      if (!assesment) throw "Assesment not found";

      // check if assesment is already completed
      if (assesment.status == "COMPLETE") throw
"Assesment already completed";

      let updatedData = {};

      // check and add the question to the results
      const userAnswer = data.answer.toUpperCase();


     let questionResult = null;

      // get the question
      const question = await
conceptQuestionV2.findOne({
        _id: data.question,
      });

      if (!question) throw "Question not found";

      // get the answer
      questionResult = {
        order: assesment.results.length + 1,
        question: question.text,
        answer: userAnswer,
        correctAnswer: question.answer,
        mark: userAnswer == question.answer ?
"CORRECT" : "INCORRECT",
      };

      // add to the list of results
      updatedData["$push"] = {
        results: questionResult,
      };

      let entry = await assesmentV2
        .findOneAndUpdate({ uuid: req.params.uuid },
updatedData, { new: true })
        .populate("conceptId");

      // check the results of already has 5 correct
answers
      const results = entry.results;
      const correctAnswers = results.filter(
        (result) => result.mark == "CORRECT"
      );
      if (correctAnswers.length >= 5) {
        entry.status = "COMPLETE";



  console.log(
          "---------------------- COMPLETED UPDATE THE
USER ----------------------"
        );
        // set the concept as completed in student
        const userStudent = await studentV2
          .findOne({ _id: user.id })
          .populate("classId")
          .populate({
            path: "lessons",
            populate: {
              path: "lessonId",
            },
          });

        console.log(userStudent.lessons);
        console.log(entry);

        console.log(userStudent.lessons[0].lessonId._id);
        console.log(entry.conceptId.lessonId);
        // get the lesson in students info
        const existingLesson =
userStudent.lessons.find((lesson) => {
          if (
            lesson.lessonId._id.toString() ==
            entry.conceptId.lessonId.toString()


          ) {
            return lesson;
          }
        });

        console.log(existingLesson);

        // check if the lesson already has concepts
        if (existingLesson.concepts.length > 0) {
          const concept =
existingLesson.concepts.find((concept) => {
            if (
              concept.conceptId._id.toString() ==
entry.conceptId._id.toString()
            ) {
              return concept;
            }
          });

          // if there are concepts but not with this id then
add it
          if (!concept) {
            existingLesson.concepts.push({
              conceptId: entry.conceptId._id,
              conceptName: entry.conceptId.name,
              isCompleted: true,
              alternateLecture: false,
            });
          } else {
            // if there are concepts with this id then update
it
            const concept = existingLesson.concepts.find(
```

```
            (concept) =>
              concept.conceptId.toString() ==
entry.conceptId._id.toString()
            );
            concept.isCompleted = true;
          }
        } else {

    // if no concepts yet, add the concept
          existingLesson.concepts.push({
            conceptId: entry.conceptId._id,
            conceptName: entry.conceptId.name,
            isCompleted: true,
            alternateLecture: false,
          });
        }

        await userStudent.save();
      }

      entry = await entry.save({ new: true });

      if (!entry) throw "Assesment session not found";

      return res.json({
        status: true,
        data: entry,
        correct: userAnswer == question.answer ?
"CORRECT" : "INCORRECT",
      });
    } catch (error) {
      console.log(error);
      return res.json({ status: false, error });
    }
  },


  delete: async (req, res) => {
    try {
      if (req.user.role != "ADMIN") throw "You are not an
admin";

      const entry = await assesmentV2.deleteOne({ _id:
req.params.id });

      return res.json({ status: true, data: entry });
    } catch (error) {
      console.log(error);
      return res.json({ status: false, error });
    }
  },
};

module.exports = assesmentController;
```

VII. CODE FOR ADAPTABILITY (ASSESSMENT SESSION)

```
//Get a specific student
async getStudent() {
            try {
            const req = await axiosClient.get(
            import.meta.env.VITE_SERVER +

            import.meta.env.VITE_API_STUDENT_SHOW_
V2 + `/${store.state.user.data._id}`
            );
            const res = req.data;
            if (!res.status) throw res.error;
this.student = res.data;
await this.getPretest();
            }

catch (error) {
            console.log(error);
            alert(error);
            }
            },

 async getPretest() {
            try {
             const req = await axiosClient.get(
            import.meta.env.VITE_SERVER +

            import.meta.env.VITE_API_PRETEST_RESULT
S_ALL_V2 +`?student=${this.student._id}`
                                                  );
const res = req.data;
if ( !res.status ) throw res.error;
console.log(res.data)

//Check if any of the pre-test is for this lesson
const pretestLesson = res.data.find ((prettest) =>
prettest.lessonId._id.toString() == this.lessonId.toString()
);

console.log(pretestLesson)

//Check if there is a pre-test for this lesson if not, return
            if ( pretestLesson == null ) {
            this.takenPretest = false;
            return;
                                                  }
            this.pretestResult = pretestLesson

// get the mastery of the concept
const mastery = this.pretestResult.result.filter(
   (r) => r.conceptName == this.concept.name)

this.mastery = mastery[ 0 ].mastery ?? '';
this.knowledgeLevel = this.pretestResult.knowledgeLevel;

// DEFAULT STACK
this.derivedMastery = 'UNMASTERED';

const easyStack = [
'EASY',
'EASY',
'AVERAGE',
'AVERAGE',
'DIFFICULT',
];

const hardStack = [
'EASY',
'AVERAGE',
'DIFFICULT',
'DIFFICULT',
'DIFFICULT',
];

this.difficultyStack = hardStack;
```

```
this.derivedMastery = 'MASTERED';
```

//This loop switch is the code that will give the number of questions per difficulty based on the student's knowledge level on each concept. It will serve as the categorical decision tree.

```
switch (this.knowledgeLevel) {
```

//If case is Poor, it means that the student does not mastered any concept therefore he/she will get the easyStack for all concept in a specific lesson.

```
case 'POOR':

this.difficultyStack = easyStack

this.derivedMastery = 'UNMASTERED';
}
break;
```

//If case is 'Fair', it means that the student mastered only 1 concept therefore he/she will get the easyStack for concepts greater than 1 which is concept 2,3,4, and 5 while the student will receive the hardStack for concept 1 in a specific lesson.

```
case 'FAIR':

if(this.concept.order > 1) {

this.difficultyStack = easyStack

this.derivedMastery = 'UNMASTERED';
}
break;
```

//If case is 'Average', it means that the student mastered only 2 concepts therefore he/she will get the easyStack for concepts greater than 2 which is concept 3,4, and 5 while the student will receive the hardStack for concept 1 and 2 in a specific lesson.

```
case 'AVERAGE':

if(this.concept.order > 2) {

this.difficultyStack = easyStack

this.derivedMastery = 'UNMASTERED';
}

break;
```

//If case is 'Good', it means that the student mastered only 3 concepts therefore he/she will get the easyStack for concepts greater than 3 which is concept 4 and 5 while the student will receive the hardStack for concept 1,2, and 3 in a specific lesson.

```
case 'GOOD':
if(this.concept.order > 3) {
this.difficultyStack = easyStack
this.derivedMastery = 'UNMASTERED';
}

break;
```

//If case is 'Very Good', it means that the student mastered only 4 concepts therefore he/she will get the easyStack for concepts greater than 4 which is concept 5 while the student will receive the hardStack for concept 1 to 4 in a specific lesson.

```
    case 'VERY GOOD':
    if ( this.concept.order > 4 ) {
    this.difficultyStack = easyStack
    this.derivedMastery = 'UNMASTERED';
    }

    break;
```

//If case is 'Excellent', it means that the student mastered all concepts therefore, the student will get the hardStack for all concepts in a specific lesson

```
    case 'EXCELLENT':
    this.difficultyStack = hardStack
    this.derivedMastery = 'MASTERED';
    break;
    default:
    break;
    }
     await this.startAssesment();
    async getQuestion () {
    try{


    // convert prevQuestions to comma separated string
    const prevQuestions = this.prevQuestions.map( q =>
    q ).join( ',' );
```

//The system will get an easy, average, or difficult question based on the mastery level of the student in a specific concept.

```
    const difficulty = this.difficultyStack[0]
    const entry = await axiosClient.get(
    import.meta.env.VITE_SERVER +
    import.meta.env.VITE_API_CONCEPT_QUESTIONS
    _ALL_V2 +`?concept=${ this.conceptId }&difficulty=${
    difficulty }&exclude=${prevQuestions}`
                                                    );
    const res = entry.data;
    if ( !res.status ) throw res.error;
    console.log( res.data )
```

//The system will check if the result is correct or wrong

```
    if ( res.data.length === 0 ) {
    this.noQuestionsFound = true;
    return;
    }
```

//Code below: If the student answered the question correctly, the score will increase by one and the system will fetch the next question based on the difficultyStack for that specific concept. If the student got the question wrong, the system will get the questions with the same difficulty level as the question that was answered wrong.

The system will fetch a question with the same difficulty level and that are not yet answered by the student. The system will only show the previously answered questions when all questions were already answered.

```
const randomIndex = Math.floor( Math.random() *
res.data.length );
this.currentQuestion = res.data[ randomIndex ];
this.answer.question = this.currentQuestion._id;
this.answer.answer = '';

}
catch ( error ){
console.log( error );
alert( error );
        }
},

async submitAnswer () {
try{
const req = await axiosClient.put(
import.meta.env.VITE_SERVER +
import.meta.env.VITE_API_ASSESSMENT_UPDATE_
V2 + '/' + this.assesmentSession.uuid,
{
question: this.answer.question,
answer: this.answer.answer,
        }
);

const res = req.data;
if ( !res.status ) throw res.error;
if ( res.correct == 'CORRECT' ){
this.difficultyStack.shift();
this.score++;
this.isCorrect = true;


}
if ( res.correct == 'INCORRECT' ){
this.isCorrect = false;
        }
this.questionCount++;
.isAfterQuestion = true;
                                }
catch ( error ){
console.log( error );
alert( error );
}
},
async nextQuestion () {
this.isAfterQuestion = false;
this.prevQuestions = [...this.prevQuestions,
this.currentQuestion._id]
this.currentQuestion = null;

await this.getQuestion();
if ( this.difficultyStack.length == 0 ){
this.isFinished = true;
                                }
                        }
  },
};
```

Appendix 3: System Flowcharts

I. Students Access Flowchart