FACULTY OF ENGINEERING

REII 414
Practical:
E-Learing Platform

by:

Jacques Beukes 26028107

Dewald Krynauw 26013835

Submitted in pursuit of the degree

BACHELOR OF ENGINEERING

In

COMPUTER AND ELECTRONIC ENGINEERING

North-West-University Potchefstroom Campus

Supervisor: Mr. A. Alberts
Potchefstroom
2018

Contents

Abbreviations

List of Figures

List of Tables

1 Introduction

2 Background

3 Front-end

3.1 Login and Register Interface

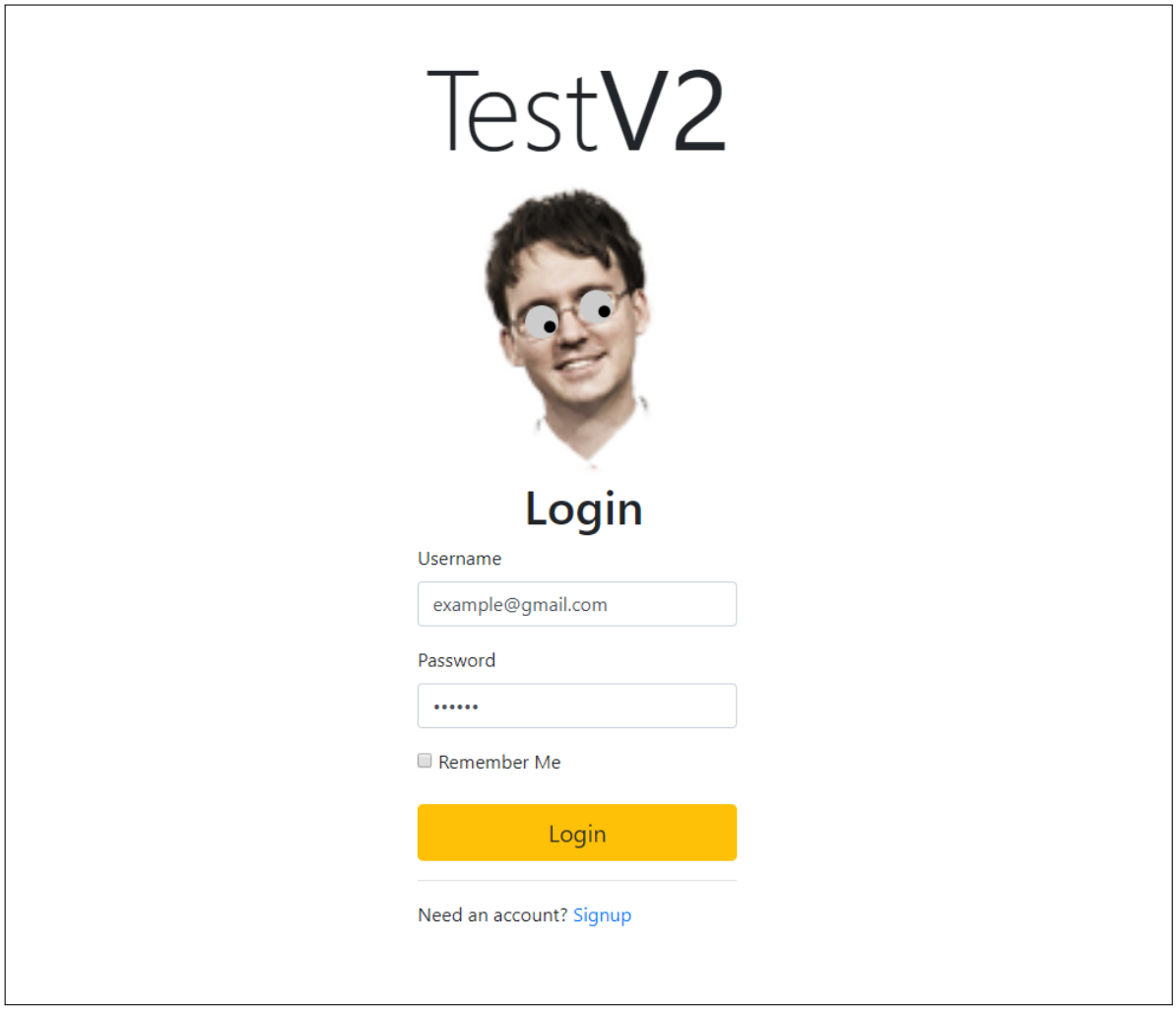

Figure 3.1: Login page

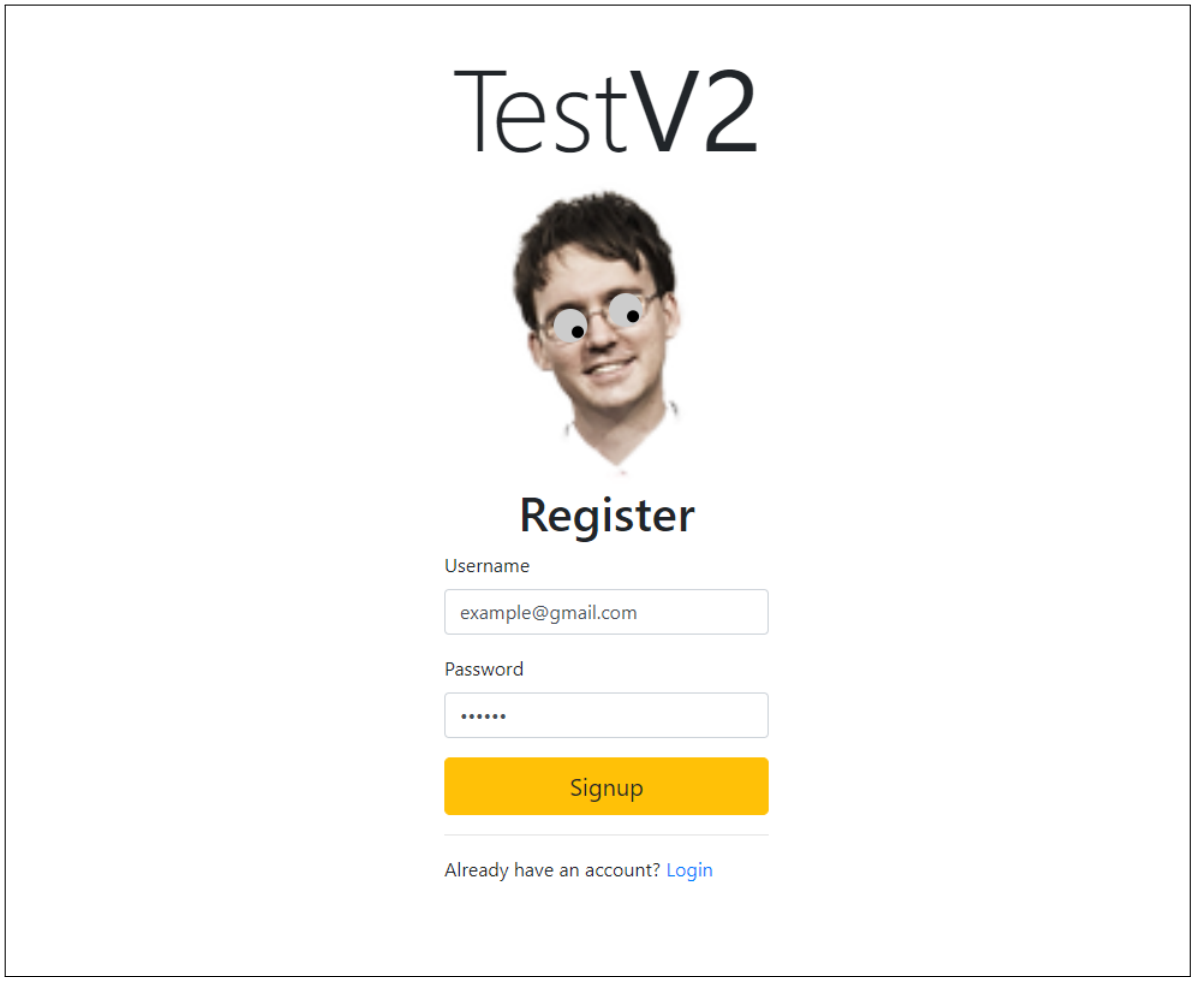

Figure 3.2: Register page

3.2 Student Interface

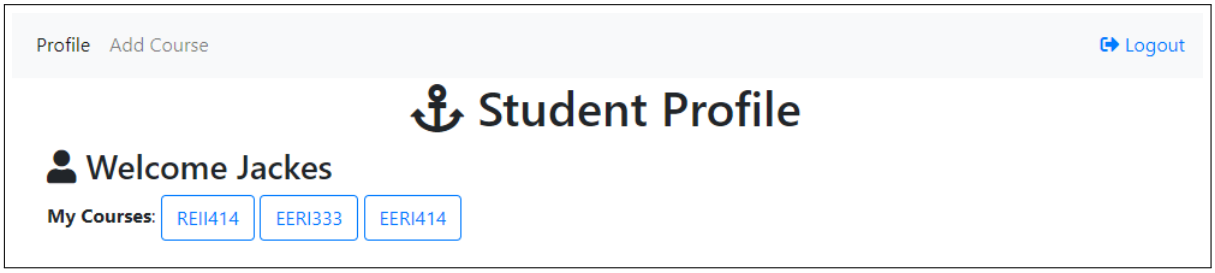

Figure 3.3: Student profile page

3.3 Lecturer Interface

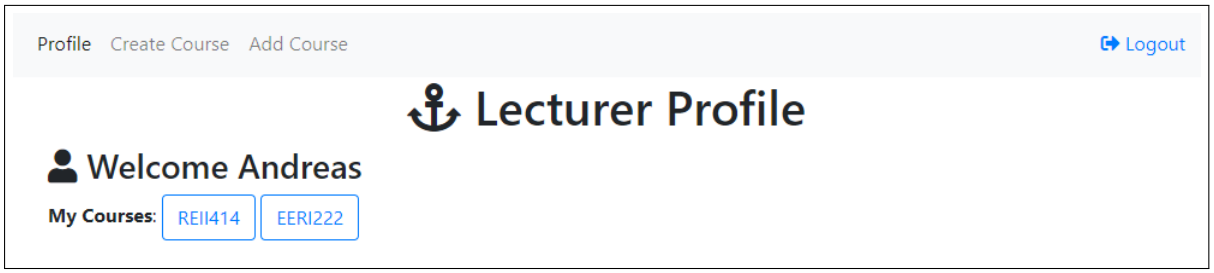

Figure 3.4: Lecturer profile page

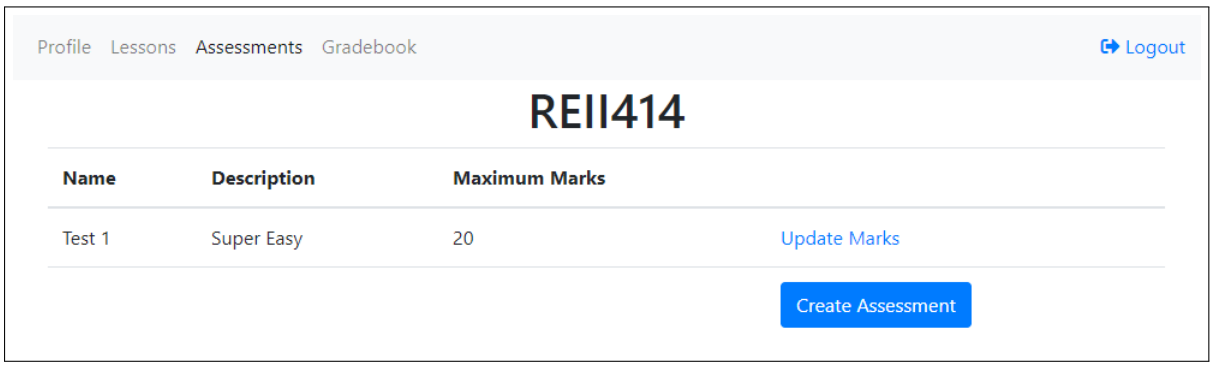

Figure 3.5: Lecturer assessments page

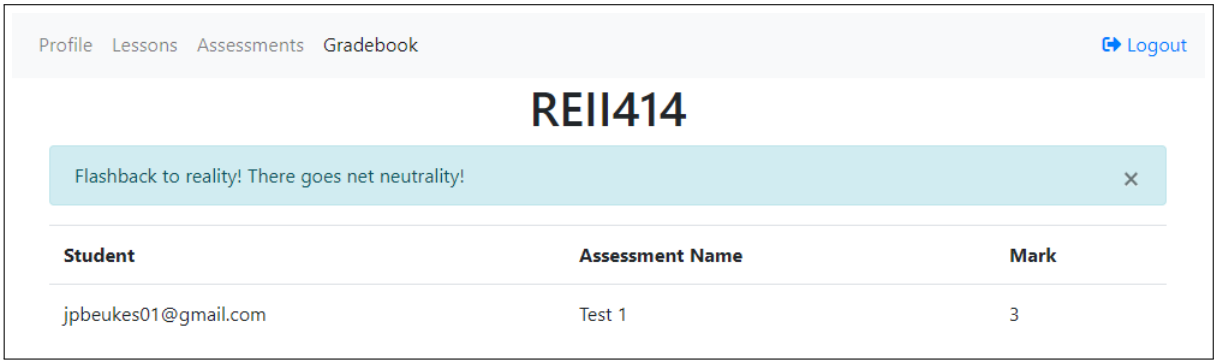

Figure 3.6: Lecturer gradebook page

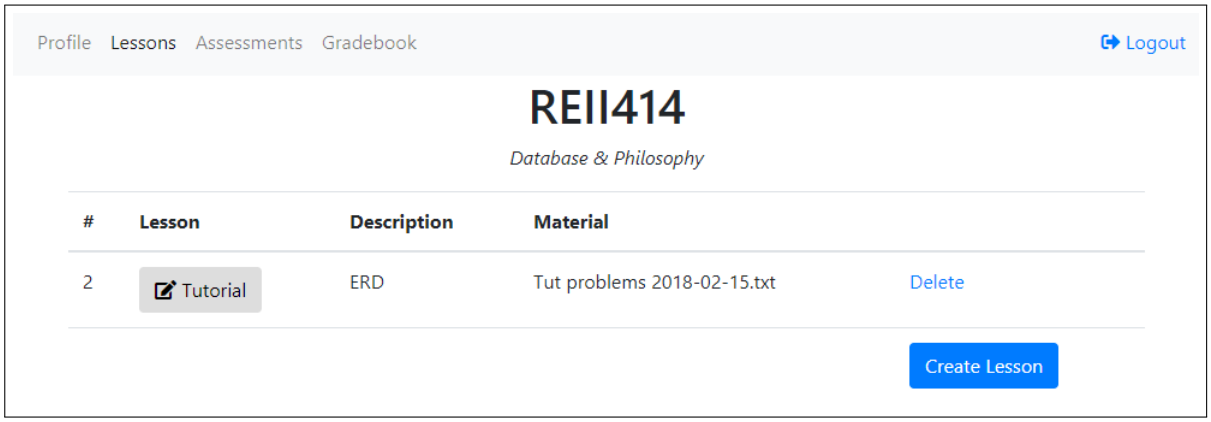

Figure 3.7: Lecturer lessons page

4 Back-end

4.1 Sign up and Login

The 'passport' package in Nodejs allows the site to handle sessions as well as cookies in an organised way. The session stores the current user's personal info inside his/her browser to allow for a personalised site.
The 'bcrypt' package allows the passwords to be encrypted in the database with the sha256 algorithm. This ensures that the system administrators can enter into the user's sites.
Figures 4.1 and 4.2 can be summarised as, insert new user into database with encrypted password. The user logging in gets flashed error messages; should the details entered be wrong.

```
// =========================================================================
// LOCAL SIGNUP ============================================================
// =========================================================================
// we are using named strategies since we have one for login and one for signup
// by default, if there was no name, it would just be called 'local'

passport.use(
    'local-signup',
    new LocalStrategy({
        // by default, local strategy uses username and password, we will override with email
        usernameField : 'username',
        passwordField : 'password',
        passReqToCallback : true // allows us to pass back the entire request to the callback
    },
    function(req, username, password, done) {
        // find a user whose email is the same as the forms email
        // we are checking to see if the user trying to login already exists
        connection.query("SELECT * FROM users WHERE username = ?",[username], function(err, rows) {
            if (err)
                return done(err);
            if (rows.length) {
                return done(null, false, req.flash('signupMessage', 'That username is already taken.'));
            } else {
                // if there is no user with that username
                // create the user
                var newUserMysql = {
                    username: username,
                    password: bcrypt.hashSync(password, null, null)  // use the generateHash function in our user model
                };

                var insertQuery = "INSERT INTO users ( username, password ) values (?,?)";

                connection.query(insertQuery,[newUserMysql.username, newUserMysql.password],function(err, rows) {
                    newUserMysql.id = rows.insertId;

                    return done(null, newUserMysql);
                });
            }
        });
    })
);
```

Figure 4.1: Code: Local Sign-up

```
// =========================================================================
// LOCAL LOGIN ============================================================
// =========================================================================
// we are using named strategies since we have one for login and one for signup
// by default, if there was no name, it would just be called 'local'

passport.use(
    'local-login',
    new LocalStrategy({
        // by default, local strategy uses username and password, we will override with email
        usernameField : 'username',
        passwordField : 'password',
        passReqToCallback : true // allows us to pass back the entire request to the callback
    },
    function(req, username, password, done) { // callback with email and password from our form
        connection.query("SELECT * FROM users WHERE username = ?",[username], function(err, rows){
            if (err)
                return done(err);
            if (!rows.length) {
                return done(null, false, req.flash('loginMessage', 'No user found.')); // req.flash is the way to set flashdata using
                connect-flash
            }

            // if the user is found but the password is wrong
            if (!bcrypt.compareSync(password, rows[0].password))
                return done(null, false, req.flash('loginMessage', 'Oops! Wrong password.')); // create the loginMessage and save it to
                session as flashdata

            // all is well, return successful user
            return done(null, rows[0]);
        });
    })
);
```

Figure 4.2: Code: Local Login

4.2 Courses

Courses can be created and edited by lecturers, whereas students can only view the courses. Figure 4.3 shows how a new course is added to the database. Figure 4.4 shows how the course and all of its content is retrieved as a JSON object from the database and rendered to the user.

```
//post add new course to db
app.post('/courses/new', function(req,res){
    // console.log(req.body);
    let sql = 'INSERT INTO courses (courseName, courseDescription, userID) VALUES (?,?,?)';
    let courseName = req.body.courseName;
    let courseDesc = req.body.courseDescription;
    let query = connection.query(sql,[courseName, courseDesc, req.user.id], (err, results) => {
    if(err) throw err;
        //add demo lesson
        let demo = 'demo';
        let sql = 'INSERT INTO lessons (COURSE_FK, lessonNumber, LESSON_NAME, LESSON_DESCRIPTION, LESSON_MATERIAL) VALUES (?,1,?,?,?) ';
        let query = connection.query(sql,[results.insertId, demo,demo,demo], (err, results) => {
            if(err) throw err;
            res.redirect('/profile');
        });
    });
});
```

Figure 4.3: Code: Create new course

```
// ====================================
// Individual Courses ==================
// ====================================
//open individual course and it's lessons for student
app.get('/course/:courseName', function(req,res){
    //select course and it's corresponding lessons
    let sql = 'SELECT * FROM lessons,courses WHERE lessons.COURSE_FK = courses.COURSE_ID and courses.courseName = ? order by lessonNumber as
        let query = connection.query(sql,[req.params.courseName], (err, results2) => {
            if(err) throw err;
            if(isEmpty(results2)) {
                if(req.user.lecturer > 0) {
                    res.redirect('/profile');//*moet course object paas maar is nie een
                } else {
                    res.redirect('/profile');
                };
            } else {
                if(req.user.lecturer > 0){
                    res.render('courseEdit.ejs',{user: req.user, course: results2});
                } else {
                    res.render('courseView.ejs',{user: req.user, course: results2});
                }
            };
        });
});
```

Figure 4.4: Code: Get Courses

4.3 Lessons

The code in Figure 4.5 shows the post request when the lecturer creates a new lesson. The details of of the lesson is added with the files and are sent to the server and database for later retrieval.

```
// ====================================
// Lessons ============================
// ====================================
app.post('/course/:courseName/addlesson', function(req, res){
    //get course id
    var form = new formidable.IncomingForm();
    form.uploadDir = "/";
    form.parse(req, function (err, fields, files) {
        var oldpath = files.filetoupload.path;
        var newpath = createMaterialPath(req.params.courseName) + '/' + files.filetoupload.name;
        fs.rename(oldpath, newpath, function (err) {
          if (err) throw err;
        //   res.write('File uploaded and moved!');
        //   res.end();
        });

        var lessonNumber = fields.lessonNumber;
        var LESSON_NAME = fields.LESSON_NAME;
        var LESSON_DESCRIPTION = fields.LESSON_DESCRIPTION;
        var LESSON_MATERIAL = fields.LESSON_MATERIAL;
        let sql = 'SELECT COURSE_FK FROM lessons,courses WHERE lessons.COURSE_FK = courses.COURSE_ID and courses.courseName = ?';
        let query = connection.query(sql, [req.params.courseName], (err, results) => {
            if(err) throw err;
            console.log(results[0].COURSE_FK);
            var courseid = results[0].COURSE_FK;

            let sql = 'INSERT INTO lessons (COURSE_FK, lessonNumber, LESSON_NAME, LESSON_DESCRIPTION, LESSON_MATERIAL) VALUES (?,?,?,?,?)
            ';
            let query = connection.query(sql,[courseid, lessonNumber,LESSON_NAME,LESSON_DESCRIPTION,files.filetoupload.name], (err,
            results) => {
                if(err) throw err;
                res.redirect('/course/'+req.params.courseName);
            });
        });
    });
```

Figure 4.5: Code: Create Lessons

4.4 Assessments

The lecturer can add assessments to a course and enter the marks of the students for a specific assessment. Figure 4.6 below shows how a new assessment is posted to the database and how marks are inserted for a specific user.

```
app.post('/course/:courseName/addAssessment', function(req, res){
    let sql = 'SELECT COURSE_ID FROM assessments,courses WHERE courses.courseName = ?';
    let query = connection.query(sql, [req.params.courseName], (err, results) => {
        if(err) throw err;
        // console.log(results[0].COURSE_FK);
        var courseid = results[0].COURSE_ID;

            let sql = 'INSERT INTO assessments (COURSE_FK, ASSESSMENT_NAME, ASSESSMENT_DESCRIPTION, ASSESSMENT_MAX_MARK) VALUES (?,?,?,?)';
            var ASSESSMENT_NAME = req.body.ASSESSMENT_NAME;
            var ASSESSMENT_DESCRIPTION =req.body.ASSESSMENT_DESCRIPTION;
            var ASSESSMENT_MAX_MARK = req.body.ASSESSMENT_MAX_MARK;
            let query = connection.query(sql, [courseid, ASSESSMENT_NAME, ASSESSMENT_DESCRIPTION, ASSESSMENT_MAX_MARK], (err, results) => {
                if(err) throw err;
                console.log(results);
                res.redirect('/course/'+req.params.courseName);
            });
    });
});

//inserts marks for student on assessment
app.post('/course/:courseName/assessment/:ASSESSMENT_ID', function(req, res){

    let sql = 'SELECT id FROM users WHERE users.username = ?';
    let query = connection.query(sql, [req.body.username, req.params.ASSESSMENT_ID], (err, results) => {
        if(err) throw err;

        var STUDENT_FK = results[0].id; // moet verander
        var MARK_SCORE = req.body.MARK_SCORE;
        var assID = parseInt(req.params.ASSESSMENT_ID);

        let sql = 'INSERT INTO marks (STUDENT_FK, ASSESSMENT_FK, MARK_SCORE) VALUES (?,?,?)';
            let query = connection.query(sql, [STUDENT_FK, assID, MARK_SCORE], (err, results) => {
                if(err) throw err;
                // console.log(results);
                res.redirect('/course/'+req.params.courseName);
            });
    });
});

app.get('/course/:courseName/assessment/:ASSESSMENT_ID/updateMarks', function(req, res){
    res.render('updateMarks.ejs', {courseName: req.params.courseName, assessmentID:req.params.ASSESSMENT_ID});
});
```

Figure 4.6: Code: Create and Update Marks for Assessments

4.5 Grade-book

Lastly the grade-book is split into two types, one for students and one for lecturers. The student grade book only gets the student's mark for all the assessments in that course. The lecturer's grade book show all the students and their respective mark for all of the assessments.

```
// ====================================
// ============Gradebook================
// ====================================
//vir students view
app.get('/course/:courseName/gradebook', function(req, res){
    if(req.user.lecturer > 0){
        let sql = 'SELECT * FROM users,courses,assessments,marks WHERE assessments.COURSE_FK = courses.COURSE_ID and marks.ASSESSMENT_FK =
        assessments.ASSESSMENT_ID and marks.STUDENT_FK = users.id   and courses.courseName = ? order by ASSESSMENT_NAME asc';
        let query = connection.query(sql, [req.params.courseName], (err, results) => {
        if(err) throw err;
        // console.log(results);
        req.flash('info', 'Flashback to reality! There goes net neutrality!');
        // res.json(results);
        res.render('gradebook.ejs', {message: req.flash('info'), marks: results, courseName:req.params.courseName}); //get marks for user
        });
    } else {
        let sql = 'SELECT * FROM marks,users,assessments,courses WHERE marks.STUDENT_FK=users.id and
        marks.ASSESSMENT_FK=assessments.ASSESSMENT_ID and assessments.COURSE_FK = courses.COURSE_ID and users.id = ? and
        courses.courseName =?';
        let query = connection.query(sql, [req.user.id, req.params.courseName], (err, results) => {
        if(err) throw err;
        // console.log(results);
        req.flash('info', 'Flashback to reality! There goes net neutrality!');
        // res.json(results);
        res.render('gradebook-student.ejs', {message: req.flash('info'), marks: results, courseName:req.params.courseName}); //get marks
        for user
        });
    };

});
```

Figure 4.7: Code: Grade book

5 Database

6 Conclusion

6.1 Strenghts

6.2 Flaws

6.3 Improvements

6.4 Techniques Learned