Code Review and Risk Assessment Document

CCAS.4.3 Software Engineering

Project

Ahmed Mahmoud - 202101585
Donia Elanany - 202101818
Nada Adel - 202102397

Contents

1 Introduction

This document presents a detailed analysis of the e-learning platform project's codebase, focusing on key aspects such as security, functionality, readability, maintainability, and performance. It includes both a thorough code review and a comprehensive risk assessment to ensure adherence to best practices, identify potential vulnerabilities, and establish a solid framework for continuous improvement and long-term stability. This process will allow the development team to prioritize key issues, enhance security measures, and proactively mitigate potential problems.

2 Code Review

2.1 Code Review Plan/Checklist

Purpose

The primary purpose of this code review is to rigorously examine the source code, identifying potential defects, security vulnerabilities, and areas for improvement. It aims to ensure that the code is not only functional but also secure, well-structured, and easy to maintain, which contributes to the overall quality and longevity of the e-learning platform.

Scope

This code review will focus on critical components of the website related to user authentication, assignment uploads, and grading processes. The scope includes the following files:

- `server.js`: For API handling, security measures, file storage, user authentication, and database interactions related to assignments and grading.
- `course.html`: Client-side components, including file upload forms, grading interfaces, and user interaction elements.
- `scripts.js`: Client-side JavaScript functions that handle upload requests, form validations, grade submissions, and other related event handling.

Checklist Items

The code review will thoroughly evaluate each file against the following key areas:

1. **Functionality:**
    - Is the user authentication process working as intended? Are user sessions created, managed, and destroyed securely?
    - Does the assignment upload process function correctly? Are files stored appropriately, and are metadata such as file paths, titles, and due dates properly saved?
    - Is the submission grading process working efficiently and accurately? Are grades saved correctly in the database and accessible by the respective users?
    - Are all edge cases handled effectively (e.g., missing fields in form submissions, invalid input formats)?
2. **Readability:**
    - Are variable names descriptive and self-explanatory, reflecting their purpose and intent?
    - Is the code well-organized and logically structured, avoiding excessive nesting or repetition?
    - Are there unnecessary code complexities that could be simplified for better understanding?
3. **Security:**
    - Are sensitive operations, such as password handling and authentication, securely implemented using modern best practices and strong encryption algorithms?
    - Are SQL queries protected against injection attacks by using parameterized queries and escaping user inputs?

- Are file uploads being validated for type and size, mitigating the risks of malicious file injections?
- Are user roles and permissions enforced correctly to prevent unauthorized access or manipulation of resources?

4. **Performance:**
   - Are database queries optimized for performance, using indexing and efficient selection techniques?
   - Are unnecessary database interactions or resource-intensive operations minimized to prevent bottlenecks?
   - Is code for file uploading and handling optimized to manage multiple large files and concurrent requests?
5. **Maintainability:**
   - Is the code organized into logical, reusable modules, facilitating easier modifications and additions?
   - Are functions well-documented, with descriptive comments to clarify purpose and usage?
   - Is there any unnecessary complexity or duplication that could be simplified or eliminated through better modularity?

2.2 Code Review Process

1. **Preparation:** The reviewer gets access to the code files (`server.js`, `course.html`, `scripts.js`) and reviews the project's scope and technical documentation.
2. **Individual Review:** The reviewer manually inspects the code based on the defined checklist, noting any issues, security vulnerabilities, or areas of improvement.
3. **Review Meeting:** The reviewer presents their findings to the team, engaging in discussions, clarifying concerns, and seeking feedback.
4. **Action Plan:** The team collaborates to create a detailed action plan outlining steps to address issues, assigning responsibility and setting timelines for implementation.
5. **Fixing:** The identified issues and required improvements will be implemented, and code updates will be applied.

2.3 Code Review Output

Files Reviewed:

- `server.js` (uploading and grading assignments sections)
- `course.html`
- `scripts.js` (upload/grading related calls)

2.4 Findings

1. **Functionality:**
   - Assignment uploads are functioning correctly, with file paths and titles stored in the database.
   - Due dates are successfully validated to ensure that they are in the future.
   - Maximum grades are validated to ensure positive numbers.
   - Submissions are stored correctly for all users, with clear association to the respective students.
   - Instructors are able to view all submissions for an assignment from all the users.
   - General-purpose error messages, while present, do not provide clear guidance on the error source.

- The assignment grading function works, and the grades are successfully saved to the database.
- Submissions are validated to ensure they are submitted before the assignment due date.
- Submissions are correctly updated when users submit updated assignments.
- Students are able to delete their submissions.
- Instructors are able to delete assignments and learning materials.

2. **Security:**
   - The file upload mechanism is identified as vulnerable to security attacks because of the lack of file type and size validation.
   - User roles are validated correctly before authorizing the users for the operations they are performing.
3. **Code Readability and Maintainability:**
   - Code is generally well-organized, but several functions can be refactored to modularize the logic and reduce duplication.
   - Descriptive comments should be added for more clarity to more complex functions.
4. **Error Handling:**
   - Error messages should be more specific, including more information for better troubleshooting.
   - Client-side error handling would provide better user experience.
5. **Performance:**
   - The upload process performs acceptably currently. However, it is not optimized for multiple concurrent uploads.
   - Database queries are generally efficient but might need optimization when fetching large datasets.
   - Scalability issues are expected when a large number of users try to upload files or grade submissions at the same time.

2.5 Action Items

1. **Implement File Type and Size Validation:** Implement robust server-side and client-side validation checks to ensure uploaded files conform to defined file types (e.g., `.pdf`, `.doc`, `.docx`, `.ppt`, `.pptx`) and size limitations.
2. **Refactor Upload and Remove Logic:** Abstract common code used for uploading and removing files/data into modular and reusable functions to prevent code duplication and enhance code consistency.
3. **Improve Comments and Error Handling:** Add descriptive comments to complex functions and database interactions, to clarify intent and usage, and to improve overall code understanding. Enhance error handling by using descriptive error messages that inform users of specific issues (e.g., incorrect date, invalid file type, missing parameters), thus improving debugging.
4. **Database Optimization:** Analyze and optimize database queries related to the assignment and submission process, using indexes and optimized query logic to improve performance.
5. **Implement Client-Side Validation:** Add client-side validation on assignment and learning material forms to check for invalid or missing data, helping improve user input quality and feedback.

2.6 Code Changes: Before and After

These code changes demonstrate specific modifications made based on the findings of the code review. They showcase improvements in security, functionality, and maintainability by addressing the identified issues.

`server.js` Changes

Before:

```
const upload = multer({
    storage: storage
});
```

After:

```
const upload = multer({
    storage: storage,
    fileFilter: (req, file, cb) => {
        const allowedMimeTypes = ['application/pdf', 'application/msword',
        'application/vnd.openxmlformats-officedocument.wordprocessingml.document',
        'application/vnd.ms-powerpoint', 'application/vnd.openxmlformats-officedocument.
            presentationml.presentation'];
          if (allowedMimeTypes.includes(file.mimetype)) {
            cb(null, true); // File type is allowed
        } else {
            cb(new Error('Invalid file type. Allowed types are pdf, doc, docx, ppt, pptx
                '), false);
           }
    },
    limits: {
        fileSize: 30 * 1024 * 1024, // Limit file size to 30MB
    }
});
```

Explanation:

- The `multer` middleware is now configured with a `fileFilter` option that validates uploaded file types. Only specific file types are allowed.
- A `limits` option is added to restrict file upload size to help prevent server overloads.

`scripts.js` Changes

Before:

```
async function uploadMaterialAndAssignments() {
  const courseId = new URLSearchParams(window.location.search).get("course_id");
  const materialInput = document.getElementById("material-upload");
  const assignmentInput = document.getElementById("assignment-upload");
  const dueDateInput = document.getElementById("due-date");
  const maximumGradeInput = document.getElementById("maximum-grade"); // Get the maximum
      grade input
  const formData = new FormData();
  if (materialInput.files.length > 0) {
    formData.append("material", materialInput.files[0]);
  }
  if (assignmentInput.files.length > 0) {
    formData.append("assignment", assignmentInput.files[0]);
  }
  formData.append("course_id", courseId);
  if (assignmentInput.files.length > 0) {
    formData.append("due_date", dueDateInput.value);
    formData.append("maximum_grade", maximumGradeInput.value);
  }
  try {
    const response = await fetch("/api/upload-material", {
      method: "POST",
      body: formData,
    });
    if (response.ok) {
```

```
      alert("Files uploaded successfully!");
      window.location.reload();
    } else {
      const errorText = await response.text();
      alert(`Error uploading files: ${errorText}`);
    }
  } catch (error) {
    console.error("Error during file upload:", error);
    alert("Failed to upload files due to network error.");
  }
}
async function submitAssignment(assignmentId, formElement) {
    const studentId = await getStudentId();
      const formData = new FormData(formElement);
      formData.append('assignment_id', assignmentId);
        formData.append('student_id', studentId);
        try {
          const response = await fetch('/api/submit-assignment', {
             method: 'POST',
            body: formData,
        });
           if (response.ok) {
           alert('Assignment submitted successfully!');
           window.location.reload();
          } else {
           const errorText = await response.text()
            alert(`Error submitting assignment: ${errorText}`);
            }
      } catch (error) {
          console.error('Error submitting assignment:', error);
        alert('Failed to submit assignment due to network error.');
     }
}
```

After:

```
async function uploadMaterialAndAssignments() {
    const courseId = new URLSearchParams(window.location.search).get('course_id');
    const materialInput = document.getElementById('material-upload');
    const assignmentInput = document.getElementById('assignment-upload');
    const dueDateInput = document.getElementById('due-date');
    const maximumGradeInput = document.getElementById('maximum-grade'); // Get the
        maximum grade input
    const formData = new FormData();

  if (materialInput && materialInput.files && materialInput.files.length > 0) {
        if(!isValidFileType(materialInput.files[0])) {
            alert("Please upload a valid file type (pdf, doc, docx, ppt, pptx)")
            return
        }
        formData.append('material', materialInput.files[0]);
  }
  if (assignmentInput && assignmentInput.files && assignmentInput.files.length > 0) {
      if(!isValidFileType(assignmentInput.files[0])) {
         alert("Please upload a valid file type (pdf, doc, docx, ppt, pptx)")
          return
        }
         if(!dueDateInput.value) {
           alert("Please enter a due date");
           return
        }
        if(!maximumGradeInput.value) {
           alert("Please enter maximum grade");
           return
         }
       formData.append('assignment', assignmentInput.files[0]);
  }

    formData.append('course_id', courseId);
        if (assignmentInput && assignmentInput.files && assignmentInput.files.length >
            0) {
           formData.append('due_date', dueDateInput.value)
           formData.append('maximum_grade', maximumGradeInput.value);
     }
```

```
    try {
        const response = await fetch('/api/upload-material', {
            method: 'POST',
            body: formData,
        });
        if (response.ok) {
            alert('Files uploaded successfully!');
            window.location.reload()
        } else {
            const errorText = await response.text()
            alert(`Error uploading files: ${errorText}`);
        }
    } catch (error) {
        console.error('Error during file upload:', error);
        alert('Failed to upload files due to network error.');
    }
}
function isValidFileType(file) {
    const allowedTypes = ['application/pdf', 'application/msword',
        'application/vnd.openxmlformats-officedocument.wordprocessingml.document',
        'application/vnd.ms-powerpoint', 'application/vnd.openxmlformats-officedocument.
            presentationml.presentation'];
    return allowedTypes.includes(file.type);
}
    async function submitAssignment(assignmentId, formElement) {
      const studentId = await getStudentId();
        const submissionInput = formElement.querySelector('input[name="submission"]');
          if (submissionInput && submissionInput.files && submissionInput.files.length >
              0) {
            if(!isValidFileType(submissionInput.files[0])) {
              alert("Please upload a valid file type (pdf, doc, docx)")
              return
            }
        } else {
          alert("Please choose a file");
            return
        }
      const formData = new FormData(formElement);
        formData.append('assignment_id', assignmentId);
        formData.append('student_id', studentId);
        try {
          const response = await fetch('/api/submit-assignment', {
            method: 'POST',
            body: formData,
        });
            if (response.ok) {
            alert('Assignment submitted successfully!');
              window.location.reload();
        } else {
              const errorText = await response.text()
            alert(`Error submitting assignment: ${errorText}`);
        }
    } catch (error) {
            console.error('Error submitting assignment:', error);
          alert('Failed to submit assignment due to network error.');
        }
}
```

Explanation:

- The `uploadMaterialAndAssignments` function checks file types on the client-side using the `isValidFileType` function.
- The `isValidFileType` function validates if the selected file `mimetype` matches any of the allowed types.
- The `submitAssignment` is also updated to check for the correct file types before sending the request.

`course.html` Changes

- No changes were made to `course.html` file structure.

3 Risk Plan

Risk ID	Risk Description	Likelihood	Effect	Mitigation Plan
R-001	**Security Breach Due to Malicious File Uploads**	High	Data Loss, System Unusable	Validate file types and sizes on both the client and server sides. Store uploaded files outside of the web server's document root. Implement virus scanning of uploaded files.
R-002	**SQL Injection Vulnerability**	Medium	Unauthorized Access to Data	Utilize prepared statements with parameterized queries for all database operations. Validate and sanitize user inputs before processing.
R-003	**Unauthorized Grading/Submission Manipulation**	Medium	Grades and Submission Compromise	Verify user roles before enabling access to grading and submitting actions. Validate IDs when updating grades or submitting assignments.
R-004	**Authentication Bypass**	Medium	Unauthorized Access to Resources	Implement robust session and token validation mechanisms and ensure they are regularly checked.
R-005	**Session Hijacking**	Medium	Unauthorized Access	Use secure cookies with HttpOnly and Secure flags. Implement short session timeouts and session expiration.
R-006	**Improper Error Handling**	Medium	Security Vulnerabilities, Poor User Experience	Standardize error responses with informative and user-friendly messages, avoid exposing sensitive system information in error messages.
R-007	**Insufficient Rate-Limiting**	Medium	Denial-of-Service (DoS) Attacks, Server Overload	Implement rate limiting and request throttling at the API level. Queue and manage requests efficiently.
R-008	**Data Loss Due to Server Failures**	Low	Loss of Critical Information, System Downtime	Implement regular database backups with offsite storage. Develop and maintain a disaster recovery plan.
R-009	**Incorrect Due Dates for Assignments**	Medium	Missed Deadlines, System Errors	Validate and enforce due dates on both the server-side and client-side to ensure consistency and prevent submission errors.
R-010	**Password Security Vulnerability**	Low	User Account Compromise	Ensure that password complexity and character requirements are enforced (e.g., minimum length, use of uppercase / lowercase letters, numbers, and special characters). Implement a mechanism to block users after repeated failed login attempts.