CONTRACT MONTHLY CLAIM SYSTEM (CMCS) – PROTOTYPE (POE PART)

Keabetswe Masole ST10437711

Table of Contents

1. Introduction (Background & Objectives)

The Contract Monthly Claim System, or CMCS, that's the prototype web app they built. It came from the Portfolio of Evidence, you know, for the PROG6212 module. The whole idea is to make submitting monthly claims simpler for independent contractor lecturers. Verifying and approving them without the usual hassle. In practice, it involves lots of people, like the lecturers, program coordinators, and academic managers. You have to check hours worked, rates, and all that supporting paperwork real carefully.

This POE aims to show some understanding of C# GUI in the .NET Core environment. Applying theory to an actual project. Right now, it's a non-functional prototype. It focuses on design, layout, and the overall flow. Not the back-end or database stuff yet. Still, it gives a solid idea of how it'd work in reality. Makes the UI intuitive and user-friendly, pretty much. The scope for Part 1 of the POE includes. **Planning and documentation**, including design explanations, assumptions, and constraints.

- A **UML class diagram** that models the database structure for claims, users, and supporting documents.
- A **project plan** outlining the tasks, dependencies, and timeline for prototype development.
- A **graphical user interface (GUI) prototype**, developed in ASP.NET Core MVC, which demonstrates the key system workflows: claim submission by lecturers, verification by programme coordinators, approval by academic managers, and the ability to track claim status transparently.

Finishing up this first stage, you know, it sets things up nicely for the rest of the POE. Later on, we'll get into adding those functional features, hooking up the database, and putting in the authentication stuff. I mean, the main thing right now is just making sure the design is clear and documented pretty well. That way, the whole final system ends up being efficient, focused on the users, and it lines up with how the academic claim management process is supposed to go.

2. Design Documentation

The Contract Monthly Claim System prototype, or CMCS as they call it, started up mainly because people wanted something straightforward. It needed to feel intuitive from the get-go. Role-based access worked out great for dealing with lecturer claims. Right now, nothing's actually functional on purpose. It just lays out the visuals and the overall structure. That gives a good sense of how users will interact with it. Before they layer in real features down the line, at least the appearance comes through clean and effective.

Design choices guided the GUI layout in specific directions.
They chose ASP.NET Core MVC to build the application. Picked that one because it separates concerns well, you know. Scales easily without a lot of hassle. Functions nicely for web apps in general. There's a shared layout file called _Layout.cshtml. It handles navigation, branding, and page structure consistently across everything. The top nav bar makes it simple to switch to spots like the dashboard. Or claim submission. Verification. Approval. Progress tracking. Account stuff too, like login or registration.

The dashboard serves as the primary landing page.
Users land there first and see their role right away. Could be Lecturer. Programme Coordinator. Academic Manager, perhaps. From that point, they access their specific tasks. Lecturers get a form for submitting claims. It captures the claim month, hours worked, hourly rate. Includes a spot for supporting docs, even though it's just a placeholder for now. In the verification section, claims appear in a table. Programme coordinators review details and approve as required. Academic managers follow a similar process for final approval. The tracking page lets lecturers check claim status. It adds transparency to the entire workflow, basically.

The data model manages the database structure within the prototype. At present, there's only one Claim model. It covers essentials like Id, LecturerName, ClaimMonth, HoursWorked, HourlyRate, Status. And SupportingDocument. Those form the core elements for a claim.

In the complete version later on, this connects directly to a Claims table in the database. They'll include a Users table for lecturers and admins, incorporating roles. SupportingDocuments will store metadata and file paths for uploads. ClaimStatusHistory will record all changes for auditing purposes. The UML class diagram in the report details these components. It illustrates the relationships between them as well.

Assumptions played a role in shaping this early version.
User authentication and roles remain excluded for the moment. Those come in subsequent phases. Supporting docs function as a simple text field simulation. No real file storage involved. Approvals consist of mock actions that display banner messages. The database receives no updates whatsoever.

Constraints limited the scope for this initial phase.
Only the front-end matters at this stage. No back-end data persistence. No integrations of any kind. This approach allows focus on usability. On the layout. And particularly on role-based flows.

The project plan divides the work over three weeks.
Week one involved requirement gathering. Drafting the UML. Establishing the MVC framework basics. Week two focused on the dashboard and claims views. Added some sample data along the way. Week three refined the UI. Prepared documentation. Readied submission materials, such as the diagram, plan, and report. They implemented version control using GitHub with at least five commits. Each included descriptive messages.

This design phase establishes a firm foundation for future development.
The GUI appears sufficiently clear. The data model proves robust. The roadmap directs subsequent steps. In summary, the CMCS prototype structures the claim workflow in a user-friendly manner. Once functionality expands, it will effectively meet technical requirements and usability goals.

3. UML Class Diagram (Database Model)

The database design for the Contract Monthly Claim System (CMCS) is represented using a **UML class diagram**, which models the core entities, their attributes, and relationships. This diagram provides a blueprint for how data will be structured and stored once database functionality is introduced in later stages of the project.

Entities and Attributes

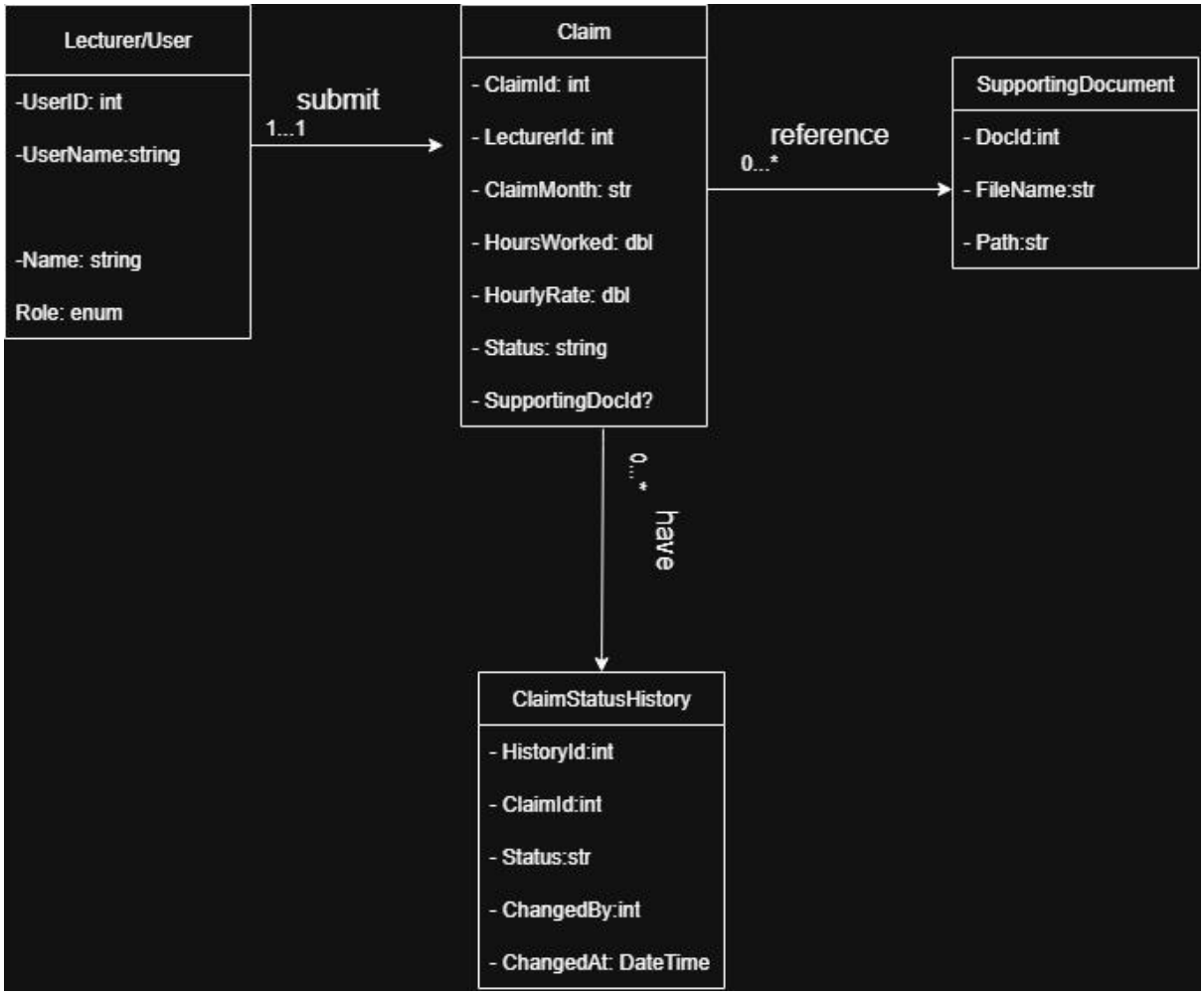
- **User (Lecturer/Staff):** Represents individuals who interact with the system. Key attributes include UserId, Username, Password, FullName, and Role. Roles are defined as Lecturer, Programme Coordinator, or Academic Manager, each with specific permissions.
- **Claim:** Stores claim details submitted by lecturers. Attributes include ClaimId, LecturerId (foreign key to User), ClaimMonth, HoursWorked, HourlyRate, Status, and SupportingDocId.
- **SupportingDocument:** Stores metadata about uploaded files, such as DocId, FileName, and FilePath. Each claim may have one or more associated supporting documents.
- **ClaimStatusHistory:** Records the history of status changes for each claim. Attributes include HistoryId, ClaimId, Status, ChangedBy, and ChangedAt. This ensures an audit trail is maintained.

Relationships

- A **User** can submit multiple **Claims**, but each claim is linked to only one user.
- A **Claim** may reference zero or more **SupportingDocuments**, depending on whether evidence is required.
- A **Claim** can multiple entries in the **ClaimStatusHistory**, reflecting its progression through the workflow (e.g., Pending → Verified → Approved).

Diagram

Figure 1: UML Class Diagram for CMCS Database Schema



Explanation

The diagram lays out the logical setup for the database. It helps track and handle claims in a solid way. You know, by breaking everything down into users, claims, documents, and history. That kind of arrangement keeps the data reliable. It allows for full audits. And it scales without a ton of problems.

Take the ClaimStatusHistory entity for example. It logs every action that happens on a claim. That gives you real transparency. Oh and, putting supporting documents in their own separate area avoids all that duplication mess. File management gets a lot easier, basically.

This structure fits right in with the MVC prototype we're building now. The Claim class we have matches the Claims entity in the diagram. We'll add in things like User and SupportingDocument entities later on. You know, during those next development phases. So the prototype ends up connecting to a full working back-end.

4. Project Plan

The project plan for the Contract Monthly Claim System (CMCS) prototype was structured over a three-week period to cover design, development, documentation, and submission requirements. Tasks were broken down into manageable units, with dependencies identified to ensure logical progression.

Week	Task	Dependencies	Duration
Week 1	Gather requirements and identify assumptions	–	1 day
Week 1	Draft UML class diagram (initial design)	Requirements identified	1 day
Week 1	Create MVC project skeleton and layout page	–	2 days
Week 2	Implement Dashboard, Submit, Verify, and Track views	Layout completed	3 days
Week 2	Define Claim model and seed sample data	UML draft completed	2 days
Week 2	Apply CSS styling and ensure responsive layout	Core views in place	2 days
Week 3	Finalise UML diagram and refine design documentation	Prototype complete	2 days
Week 3	Prepare project plan and report (Word document)	Documentation drafted	2 days
Week 3	Push at least 5 commits to GitHub with descriptive messages	Development tasks completed	1 day

CMCSPrototype

Private

Watch 0

master | 1 Branch | 0 Tags | Go to file t | Add file | Code

St10441486 Add project files. 3b0e30e · now 2 Commits

Controllers	Add project files.	now
Models	Add project files.	now
Properties	Add project files.	now
Views	Add project files.	now
wwwroot	Add project files.	now
.gitattributes	Add .gitattributes, .gitignore, and README.md.	1 minute ago
.gitignore	Add .gitattributes, .gitignore, and README.md.	1 minute ago
CMCSPrototype.csproj	Add project files.	now
CMCSPrototype.sln	Add project files.	now
CONTRACT MONTHLY CLAIM SYSTEM (CMCS)...	Add project files.	now
Program.cs	Add project files.	now
README.md	Add .gitattributes, .gitignore, and README.md.	1 minute ago
appsettings.Development.json	Add project files.	now
appsettings.json	Add project files.	now

README

7. Conclusion

The Contract Monthly Claim System prototype. That's for Part 1 of the Portfolio of Evidence. It lays out the planning and design work, you know, and gets the front-end going for how the app should function. We've hit all the deliverables in here. Like, we documented the reasons behind our choices. Put together a UML class diagram for the database structure. Sketched a project plan that actually makes sense. And built a basic GUI prototype with ASP.NET Core MVC.

This setup follows the workflows just about as we planned them. Lecturers can submit claims. Coordinators review them. Managers approve or reject. Oh and there's a tracking area, so lecturers see the status of their submissions all along the way. Right now, it's not fully operational. I mean, approvals are fake. Document uploads are dummies. No real database storage happening. Still, it establishes the foundation. For when we implement the actual features in Part 2.

We've put a ton of emphasis on usability. And navigation that's clear based on user roles. The UML diagram and project plan guide everything technically. With a schedule that fits. We used version control the whole time too. Lots of commits on GitHub. They show the build-up step by step.

Design and planning are complete at this point. So yeah, the project's set for Part 2. We'll add the back-end components there. Login functionality. Real data saving. Proper file handling. That phase makes it functional for real. Aligning with the overall goal of simplifying claims and approvals for lecturers.

8 Reference list

Draw.io (2024). *Flowchart Maker & Online Diagram Software*. [online] app.diagrams.net. Available at: https://app.diagrams.net.

Freeman, A. (2013). *Pro ASP.NET MVC 5*. Berkeley, Ca Apress.

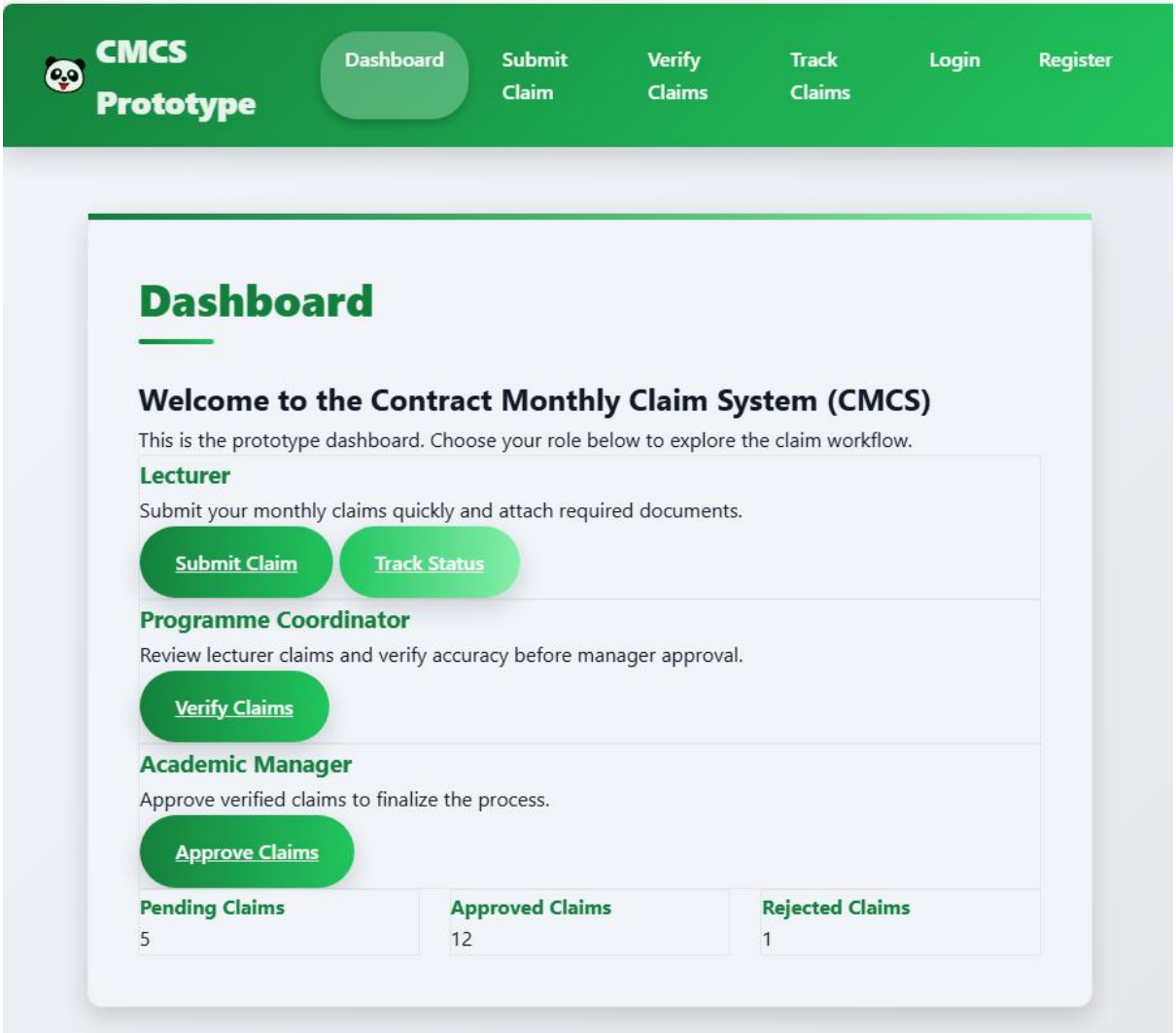
Ramotion (2023). *MVC Web App Architecture | Ramotion Agency*. [online] Web Design, UI/UX, Branding, and App Development Blog. Available at: https://www.ramotion.com/blog/mvc-architecture-in-web-application.

Smith, S. (2024). *Overview of ASP.NET Core MVC*. [online] Microsoft.com. Available at: https://learn.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-9.0.

Tutorialsteacher (2025). *ASP.NET MVC Tutorials*. [online] www.tutorialsteacher.com. Available at: https://www.tutorialsteacher.com/mvc.

Part 2 – Implementation Update

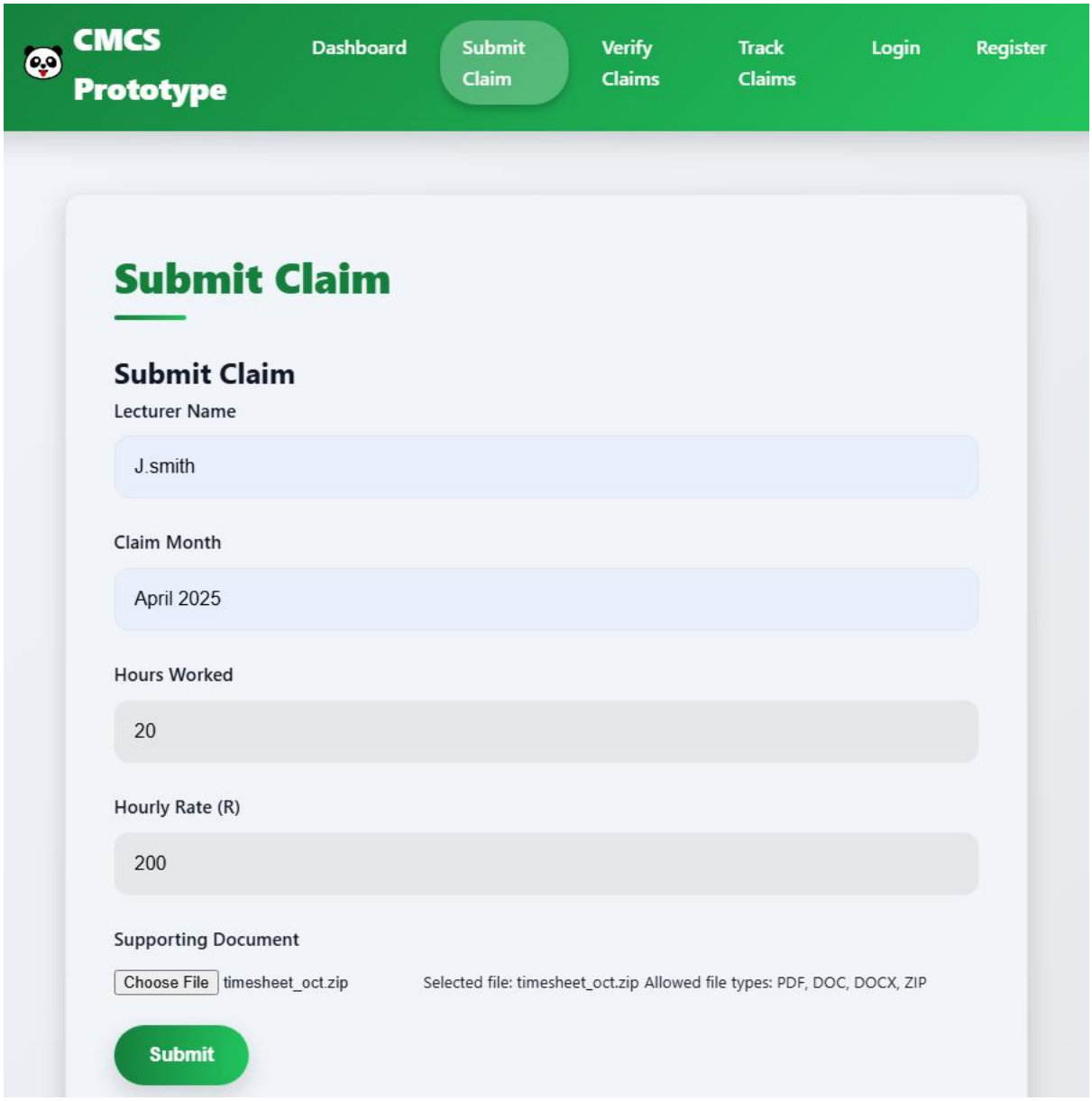
Part 2 of the Contract Monthly Claim System (CMCS) project represents the transition from a conceptual, non-functional prototype to a working web-based application. While Part 1 focused primarily on planning, design documentation, and interface layout, Part 2 extends that foundation by implementing full functionality within the ASP.NET Core MVC framework.



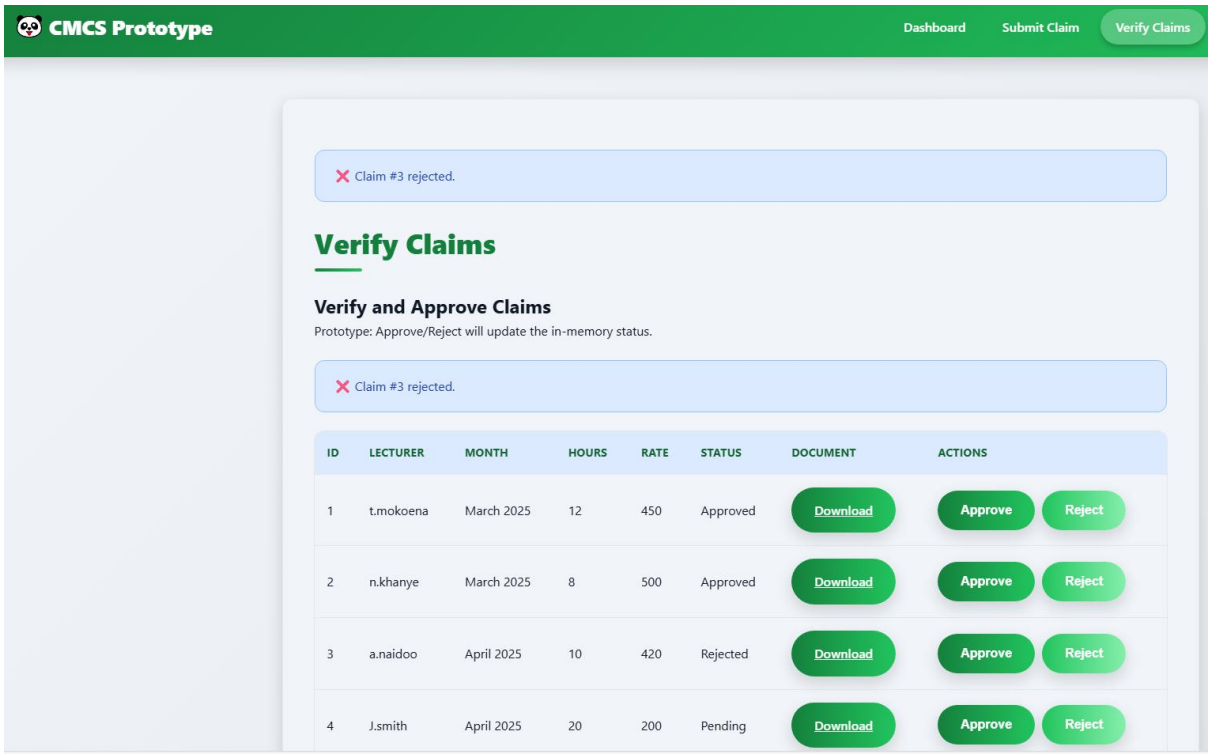
In Part 1, the emphasis was on visual design and system structure. The prototype consisted of static pages that demonstrated user flow but did not process real data. The UML class diagram, project plan, and GUI mock-ups were used to illustrate how lecturers, programme coordinators, and academic managers would interact with the system once functional. No data storage, file management, or logical operations were included at that stage.

Part 2 builds directly on that foundation by introducing interactive and functional components. The **claim submission** process is now fully implemented, allowing lecturers to complete a form with hours worked, hourly rate, and supporting notes, and to submit their claims at the click of a button. Unlike in Part 1, this submission now performs real input validation and automatically assigns a claim ID and default status.

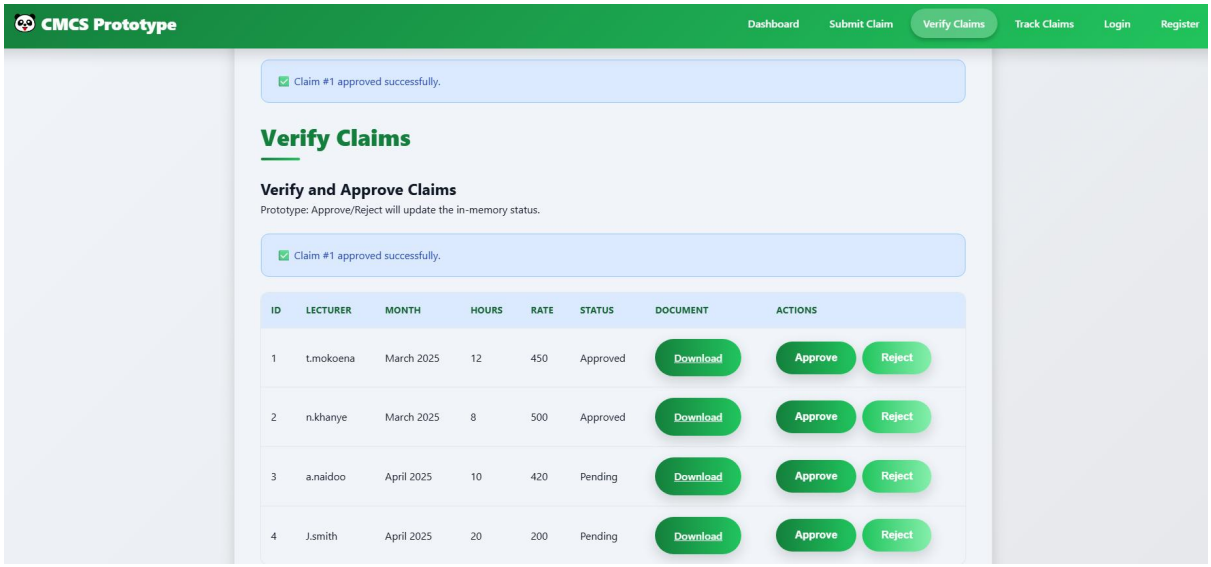
Another major enhancement is the **supporting document upload** feature. Lecturers can upload evidence in accepted formats such as PDF, DOCX, and XLSX. Uploaded files are stored securely in the application's *wwwroot/uploads* directory and linked to their corresponding claims. File handling includes basic validation for size and type, ensuring the system remains secure and stable.



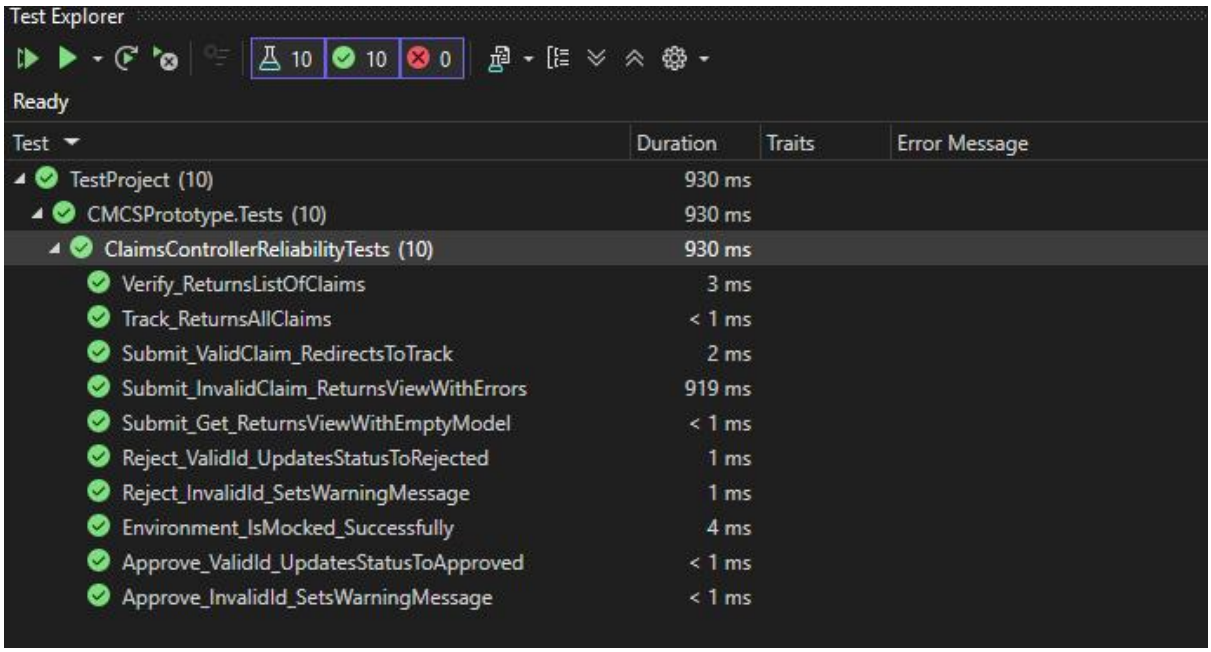
Part 2 also introduces separate **verification and approval interfaces** for Programme Coordinators and Academic Managers. These views now display pending claims and allow users to approve or reject them directly using clearly marked buttons. When an action is performed, the claim's status updates in real time from "Pending" to either "Approved" or "Rejected." This marks a key difference from Part 1, where verification was demonstrated visually only.



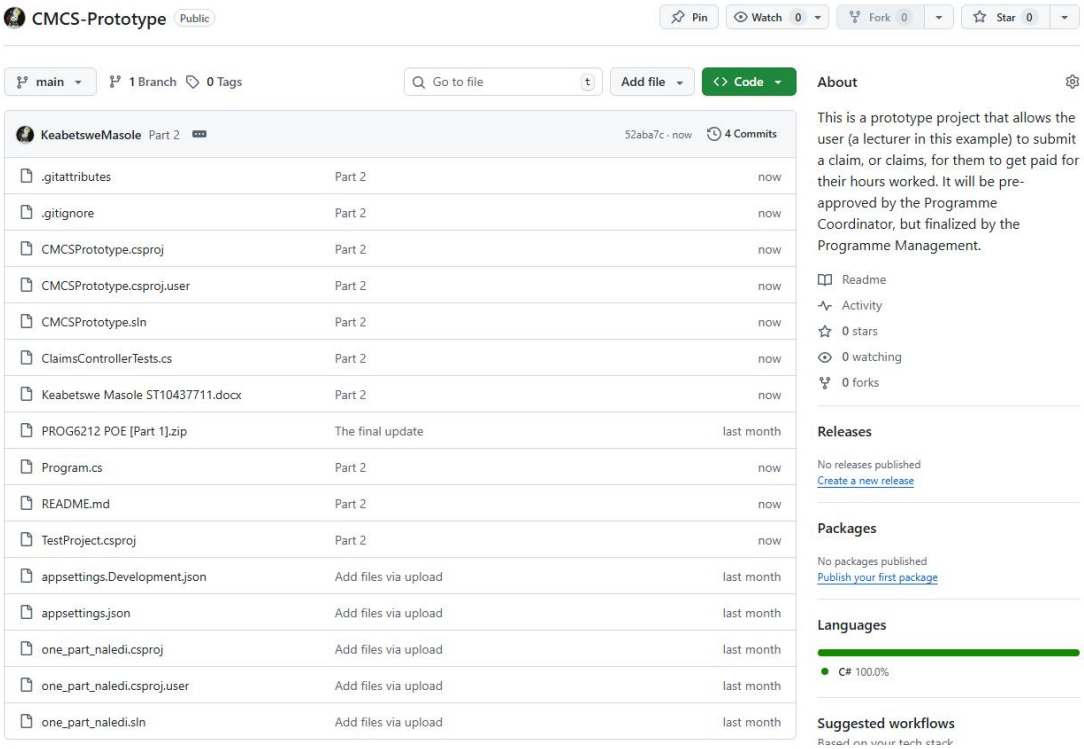
The **claim tracking system** has likewise been implemented. Lecturers can now view the progress of their submitted claims through a transparent, continuously updated interface. Each claim displays its current status, providing visibility and accountability across all workflow stages.



To ensure reliability, Part 2 incorporates **exception handling** and **user feedback mechanisms**. Errors such as missing fields, failed uploads, or invalid actions are caught and displayed to users in a clear and professional manner using temporary messages. Furthermore, **unit testing** was introduced with xUnit and Moq frameworks to verify controller logic, including claim submission, approval, rejection, and tracking.



Version control was also enhanced during this phase. Whereas Part 1 involved initial uploads of static design files, Part 2 required continuous integration with at least five meaningful GitHub commits. Each commit documents key implementation milestones such as form validation, file upload integration, approval logic, and test coverage.



Overall, Part 2 transforms the CMCS from a static prototype into an operational application capable of handling real user interactions. The addition of live functionality, validation, testing, and version control demonstrates technical readiness and practical application of software engineering principles. The system now provides a solid functional baseline for future enhancements such as full database integration, authentication, and deployment in a production environment.