A blue and white background

Description automatically generatedA blue and white background

Description automatically generatedA blue and white background

Description automatically generatedA blue and white background

Description automatically generated

Kylah Irvine - ST10435066

PROG6212  
POE Part 1

Table of Contents

[Introduction 4](#_Toc208211741)

[Purpose of the POE 4](#_Toc208211742)

[System Overview 5](#_Toc208211743)

[Target Users 6](#_Toc208211744)

[Design Overview 7](#_Toc208211745)

[Architecture Choice: MVC or WPF 7](#_Toc208211746)

[Reason for Selection 8](#_Toc208211747)

[Main Functional Components 9](#_Toc208211748)

[Design Objectives 10](#_Toc208211749)

[Assumptions and Constraints 11](#_Toc208211750)

[Assumptions About Users, Data & Process 11](#_Toc208211751)

[System Constraints 12](#_Toc208211752)

[Database Design 14](#_Toc208211753)

[Entities and Relationships 14](#_Toc208211754)

[Normalization Choices 15](#_Toc208211755)

[Scalability and Query Considerations 17](#_Toc208211756)

[UML Class Diagram 18](#_Toc208211757)

[Diagram Overview 18](#_Toc208211758)

[Entity Descriptions 19](#_Toc208211759)

[GUI Mock-up and Layout 21](#_Toc208211760)

[User Interface Overview 21](#_Toc208211761)

[Key Screens and Layout 23](#_Toc208211762)

[Role-Based View Breakdown 29](#_Toc208211763)

[Navigation and Usability Features 31](#_Toc208211764)

[Project Plan 33](#_Toc208211765)

[Milestones and Phases in Part 1 33](#_Toc208211766)

[Task Breakdown with Dependencies 34](#_Toc208211767)

[Phase 1: Project Setup & Core Modelling 34](#_Toc208211768)

[Phase 2: Authentication & Session Management 34](#_Toc208211769)

[Phase 3: Claim Submission & Viewing 34](#_Toc208211770)

[Phase 4: Coordinator Review & Status Logging 35](#_Toc208211771)

[Phase 5: Schema Refactoring & Data Integrity 35](#_Toc208211772)

[Phase 6: Stress Testing & Seed Data 36](#_Toc208211773)

[Phase 7: Acknowledge Scope Creep 36](#_Toc208211774)

[Phase 8: Documentation & Handover 36](#_Toc208211775)

[Timeline Summary 38](#_Toc208211776)

[Part 1: Core Functionality & Foundation 38](#_Toc208211777)

[Part 2: Feature Expansion & Usability 39](#_Toc208211778)

[Part 3: Reflection & Presentation 39](#_Toc208211779)

[Version Control Summary 41](#_Toc208211780)

[GitHub Setup 41](#_Toc208211781)

[Commit Strategy 42](#_Toc208211782)

[AI Usage 43](#_Toc208211783)

Introduction

# Purpose of the POE

The purpose of this POE is to demonstrate my ability to design and implement a functional software solution using C# and Entity Framework. The system supports lecturers submitting monthly claims, coordinators reviewing and approving them, and managers overseeing the process. It reflects real-world workflows and emphasizes clarity, maintainability, and user experience.

# System Overview

The Lecturer Claims Management System (LCMS) was designed to fulfil all core requirements outlined in Part 1 of the POE brief, while also introducing enhancements that improve usability and transparency. The system enables lecturers to securely log in, submit monthly claims with detailed breakdowns, and view their claim history in a clean, role-specific dashboard. Coordinators can review pending claims, approve or deny them, and log status changes with optional reasons, all of which are tracked in a dedicated ClaimStatusLog table (in the database) for full auditability. The database schema was normalized for integrity and performance, with a deliberate shift from numeric UserId keys to string-based EmployeeNumber identifiers to better reflect real-world academic systems (EmployeeNumber will act as the unique identifier). The system uses ASP.NET Core MVC for structured routing and Razor views for responsive, user-friendly interfaces. Session tracking ensures personalized access, while role-based redirects maintain security and clarity (the program knows who’s logged in right now and what role they have). Additional features such as denial reason visibility, full name display, and bulk test data (essentially dummy data i.e. not submitted by the app) for stress testing demonstrate a thoughtful, professional approach to real-world system design. Overall, the LCMS not only meets the functional expectations of Part 1, but lays a strong foundation for future expansion in Part 2 (since there’s things in there that they don’t tell you about in part 1).

# Target Users

* Lecturers (submit claims)
* Programme Coordinators (review and approve)
* Academic Managers (oversee claim activity)
* HR Administrators (optional future role for payroll integration)

Design Overview

# Architecture Choice: MVC or WPF

MVC

# Reason for Selection

The system was developed using ASP.NET Core MVC, leveraging Razor views for the user interface and Entity Framework Core for database access. This architecture supports clean separation of concerns, scalable routing, and role-based access control. I also just don’t like working with windows forms that much.

# Main Functional Components

* Login and session management
* Claim submission form
* Claim viewing dashboard
* Coordinator review interface
* Status logging via ClaimStatusLog

# Design Objectives

The Lecturer Claims Management System was designed with a strong emphasis on clarity, reliability, and low-friction workflows. It was also designed to be as “fool proof” as possible, meaning there’s very little the user can do to break it. From the outset, the goal was to create a system that feels intuitive for users, minimizes technical overhead, and supports real-world academic processes (or at least how I would imagine they’d work?) without unnecessary complexity. Key objectives included:

* **Usability First**: Every interface from claim submission to coordinator review was built to be clean, direct, and easy to navigate. The system avoids clutter and ambiguity (a button says exactly what it does), ensuring that users can complete tasks without confusion or excessive clicking and with minimal training (makes IT’s job easier and lowers barrier to entry).
* **Resilience to Change**: The system was designed to handle schema shifts and evolving requirements, as demonstrated by the “successful” migration from UserId to EmployeeNumber as the primary key. Basically by breaking the entire program by changing the primary key of one table after I’d built half the program and still managing to get it back up and running shows that it can probably be rescued from even the most dire straits. This adaptability ensures long-term maintainability and scalability.
* **Transparency and Accountability**: By introducing a dedicated ClaimStatusLog table, the system tracks every approval and denial with timestamps and situationally dependant reasons (lecturers will probably only care why things got denied, not why they’re approved). This audit trail (because finances are audited and carry big trouble when things go wrong) promotes fairness and gives lecturers clear feedback on their claims.
* **Role-Based Clarity**: Each user role (Lecturer, Coordinator, Manager for part 2) has a tailored experience. Lecturers see only their own claims, coordinators access pending reviews, and managers (if implemented) would oversee broader activity. This separation reduces cognitive load (people don’t have to think about it for more than one screen) and enhances security.
* **Low-Stress Workflow Design**: The system avoids over-engineering (in the UI design at least), and focuses on achievable, functional features. Any task must be preferably 1 click away.
* **Real-World Validation**: Sample data was inserted to stress test the system across various claim statuses. This ensures that the logic holds up under realistic conditions and that edge cases (e.g. denied claims with missing reasons) are properly handled.
* **Professional Presentation**: The inclusion of full names, clean layouts, and structured data reflects a commitment to professionalism and user respect.

Assumptions and Constraints

# Assumptions About Users, Data & Process

**Users**

* Each user has a unique EmployeeNumber, which serves as their primary identifier across the system.
* Users log in using a combination of Username, EmployeeNumber, Email, and Role (which is not manually entered, it is selected before the log in screen) all of which must match data in the database exactly.
* Roles are predefined as Lecturer, Coordinator, or Manager, and determine access level and dashboard visibility.
* Coordinators are responsible for reviewing claims and logging status changes while lecturers can only view and submit their own claims.

**Data**

* Claims are submitted monthly and are linked to a specific ClaimType (e.g. Marking, Tutoring, Moderation) of which more types may be added in future.
* Each claim includes HoursWorked, HourlyRate, and optional Notes, which are stored for audit and payroll purposes.
* Claims are initially marked as Pending and can later be Approved or Denied by a coordinator.
* Denied claims must include a Reason in the ClaimStatusLog to ensure transparency (no spiteful actions or annoyed lecturers confused by denial).
* The ClaimStatusLog table tracks every status change with a timestamp and the coordinator’s EmployeeNumber.

**Process**

* Lecturers submit claims via a form that auto-links their EmployeeNumber from the session.
* Coordinators access a filtered view of pending claims and can approve or deny each one individually.
* The system assumes that users are authenticated before accessing any dashboard or claim-related functionality (if you’re using the site, I assume you’re supposed to be here).

# System Constraints

**Technical Constraints**

* The system is built using ASP.NET Core MVC, which requires a compatible .NET runtime and hosting environment.
* SQL Server is used for data storage, meaning the system depends on a properly configured SQL Server instance and connection string.
* Entity Framework Core is used for ORM, so migrations and schema updates must follow EF conventions.
* The system does not currently support mobile responsiveness or adaptive layouts, it’s optimized for desktop use (I’d assume that they’d do this kind of thing from a computer of sorts anyway).

**Security Constraints**

* User authentication is based on matching four fields: Username, EmployeeNumber, Email, and Role. There is no password-based login or encryption implemented in Part 1.
* Session management is handled via HttpContext.Session, which assumes cookies are enabled and properly configured.

**File Handling Constraints**

* Document uploads (e.g. supporting claim files) are not yet implemented, the DocumentPath field exists but is unused in Part 1.
* If implemented, file types would be restricted to .pdf, .docx, (both could be proof of work done?) and .xlsx (for invoice tables) with a maximum size of 15MB per file (more than enough surely).

**Data Constraints**

* Each user must have a unique EmployeeNumber, which serves as the primary key and foreign key across tables.
* Claims must be linked to a valid user and include required fields (ClaimType, ClaimMonth, HoursWorked, HourlyRate, Status).
* Denied claims must include a non-empty Reason in the ClaimStatusLog, this will be enforced in the coordinator workflow.
* Claims are assumed to be submitted once per month per type (you do lump sum invoices for all your marking that month instead of 5 different marking invoices, for example), duplicate submissions are not currently restricted but may be flagged in future versions.

**Functional Constraints**

* There is no password recovery, user registration, or account management interface, all users must be pre-seeded in the database (possibly a feature not a bug?).
* Managers and HR roles are defined but not yet implemented in the UI or logic (I just saw it mentioned in part 2).
* There is no email notification or automated messaging system for claim status updates (feel like that’s something that real world systems have).
* Some functions mentioned throughout the document have not been functionally implemented even though they are talked about as if they are (possible recency bias)

Database Design

# Entities and Relationships

**Users**

* Primary Key: EmployeeNumber (string)
* Stores login credentials, contact info, and role designation
* Includes Name and Surname for professional display
* Roles include Lecturer, Coordinator, and Manager
* One user can submit many claims
* One coordinator can log many status changes

**Claims**

* Primary Key: ClaimId (int, auto-increment) (may make this into some autogenerated string? Not sure about it though, just an idea)
* Foreign Key: EmployeeNumber links to Users
* Stores claim details: type, month, hours, rate, notes, and status
* Each claim is submitted by one lecturer
* Each claim can have multiple status changes over time (maybe the denial was unjust, so the DBA or something can go into the database and change the status manually)
* Includes optional DocumentPath for future file uploads

**ClaimStatusLog**

* Primary Key: LogId (int, auto-increment)
* Foreign Keys:
  + ClaimId links to Claims
  + ChangedBy links to Users.EmployeeNumber (coordinator who made the change)
* Tracks every status update with timestamp and optional reason
* Used to build a full audit trail of claim decisions

# Normalization Choices

**First Normal Form (1NF)**

* All tables store atomic values, no repeating groups or multi-valued fields.
* Each column contains only one type of data (e.g. ClaimType is a single string, not a list).
* Primary keys are clearly defined: EmployeeNumber for Users, ClaimId for Claims, and LogId for ClaimStatusLog.

**Second Normal Form (2NF)**

* All non-key attributes are fully dependent on the primary key:
  + In Claims, fields like HoursWorked, ClaimMonth, and Status depend on ClaimId.
  + In Users, Name, Surname, Email, and Role depend on EmployeeNumber.
* No partial dependencies exist, composite keys aren’t used

**Third Normal Form (3NF)**

* No transitive dependencies:
  + Users doesn’t store derived data like total claims or status counts, those will calculated in queries.
  + ClaimStatusLog separates status history from the main Claims table, avoiding duplication and preserving historical accuracy.

**Relationship Design**

* Foreign keys are used to link Claims to Users via EmployeeNumber, and ClaimStatusLog to both Claims and Users (via ChangedBy).
* This structure supports:
  + One-to-many relationships between users and claims
  + One-to-many relationships between claims and status logs
  + Clear audit trails without bloating the Claims table

**Notable Adjustments Made During Design**

* I replaced UserId (int) with EmployeeNumber (varchar) to better reflect real-world identifiers used in academic institutions (like our student numbers).
* I added Name and Surname to improve UI clarity and professionalism.
* I manually adjusted the ChangedBy column in ClaimStatusLog to match the new EmployeeNumber format ensuring referential integrity.

# Scalability and Query Considerations

Comment on how you've designed it to handle growing numbers of users or claims without performance loss.

UML Class Diagram

# Diagram Overview

Users

ClaimStatusLog

EmployeeNumber (PK)  
Name  
Surname  
Username  
Email  
Role

LogId (PK)  
ClaimId (FK)  
ChangedBy  
NewStatus  
ChangeDate  
Reason

\*

1

1

\*

\*

1

Claims

ClaimId (PK)  
EmployeeNumber (FK)  
ClaimType  
ClaimMonth  
HoursWorked  
HourlyRate  
Notes  
DocumentPath  
Status  
SubmittedAt

# Entity Descriptions

**User**

The User entity represents individuals who interact with the system. Each user is uniquely identified by their EmployeeNumber, which serves as the primary key and foreign key in related tables. Users are assigned a Role that determines their access level (e.g. Lecturer, Coordinator, Manager). Additional fields such as Name, Surname, Username, and Email support login functionality and professional display across the UI.

**Key Attributes:**

* EmployeeNumber (PK): Unique identifier for each user
* Username: Used for login matching
* Email: Used for login matching
* Role: Defines access level and dashboard visibility
* Name, Surname: Used for full name display in views

**Claim**

The Claim entity stores monthly claim submissions made by lecturers. Each claim is linked to a specific user via EmployeeNumber, and includes details such as the type of claim, the month it applies to, hours worked, hourly rate, and optional notes. Claims begin with a Pending status and can later be Approved or Denied by a coordinator. The SubmittedAt field captures the timestamp of submission, and DocumentPath is reserved for future file upload functionality.

**Key Attributes:**

* ClaimId (PK): Unique identifier for each claim
* EmployeeNumber (FK): Links claim to submitting user
* ClaimType: Type of work claimed (e.g. Marking, Tutoring)
* ClaimMonth: Month the work was performed
* HoursWorked, HourlyRate: Used to calculate total claim value
* Notes: Optional comments from the lecturer
* Status: Current claim status (Pending, Approved, Denied)
* SubmittedAt: Timestamp of submission
* DocumentPath: Placeholder for future document uploads

**ClaimStatusLog**

The ClaimStatusLog entity tracks every status change applied to a claim. It serves as an audit trail, recording who made the change, when it occurred, what the new status is, and an optional reason for the decision. This entity supports transparency and accountability, especially for denied claims (no workplace drama). Each log entry is linked to both the claim it modifies and the coordinator who made the change.

**Key Attributes:**

* LogId (PK): Unique identifier for each status change
* ClaimId (FK): Links log entry to the relevant claim
* ChangedBy (FK): Coordinator’s EmployeeNumber who made the change
* NewStatus: The updated status (Approved or Denied)
* ChangeDate: Timestamp of the status update
* Reason: Optional explanation for the decision

GUI Mock-up and Layout

# User Interface Overview

The Lecturer Claims Management System is designed with simplicity and clarity in mind, ensuring that users can easily find what they need and complete tasks without confusion. Upon launching the system and selecting a role, users are greeted with a clean (meaning there’s not much they can do that they’re not supposed to be able to do) login screen that prompts them to enter their Username, EmployeeNumber and Email. This multi-field login ensures accurate identification and role-based access control.

Once authenticated, users are redirected to a dashboard tailored to their role. Here are some key features of things the different roles are able to see and do:

* **Lecturers** see a personalized dashboard listing all claims they’ve submitted. Each claim is displayed with key details such as type, month, hours worked, rate, and current status. Color-coded status indicators (e.g. green for approved, red for denied) provide immediate visual feedback. A clearly labelled button allows lecturers to submit a new claim, which opens a structured form with dropdowns and numeric fields (as little freedom to enter their own data as possible keeps the system error free and helps keep data consistent). The form auto-links the claim to the logged-in lecturer using session data, reducing manual input and errors (the system knows who submitted the claim).
* **Coordinators** are directed to a review dashboard showing all pending claims. Each entry includes the lecturer’s name, claim details, and action buttons to approve or deny. Denying a claim will trigger prompt for a reason (was not yet implemented in code), ensuring transparency and accountability. The interface is designed to minimize clicks and streamline decision-making (i.e. work quickly).
* **Managers** (if implemented) would access a broader overview of claim activity, though this role is currently reserved for future expansion (I didn’t read far enough to know what it’s supposed to do).

Throughout the system, navigation is intuitive and consistent. Buttons are clearly labelled, forms are logically grouped, and error messages (will) appear in context to guide users when something goes wrong (e.g. missing denial reason or incorrect login details). The layout avoids clutter, using whitespace (ironic name since I tried to make it dark mode) and headings to separate content and maintain focus.

Accessibility is supported through readable fonts, high-contrast status indicators, and minimal reliance on hidden or dynamic elements. Users can navigate using standard keyboard and mouse input, and the system avoids complex interactions that might confuse non-technical users.

Overall, the interface prioritizes clarity, ease of use, and a smooth workflow — allowing users to focus on their tasks without being distracted by unnecessary complexity.

# Key Screens and Layout

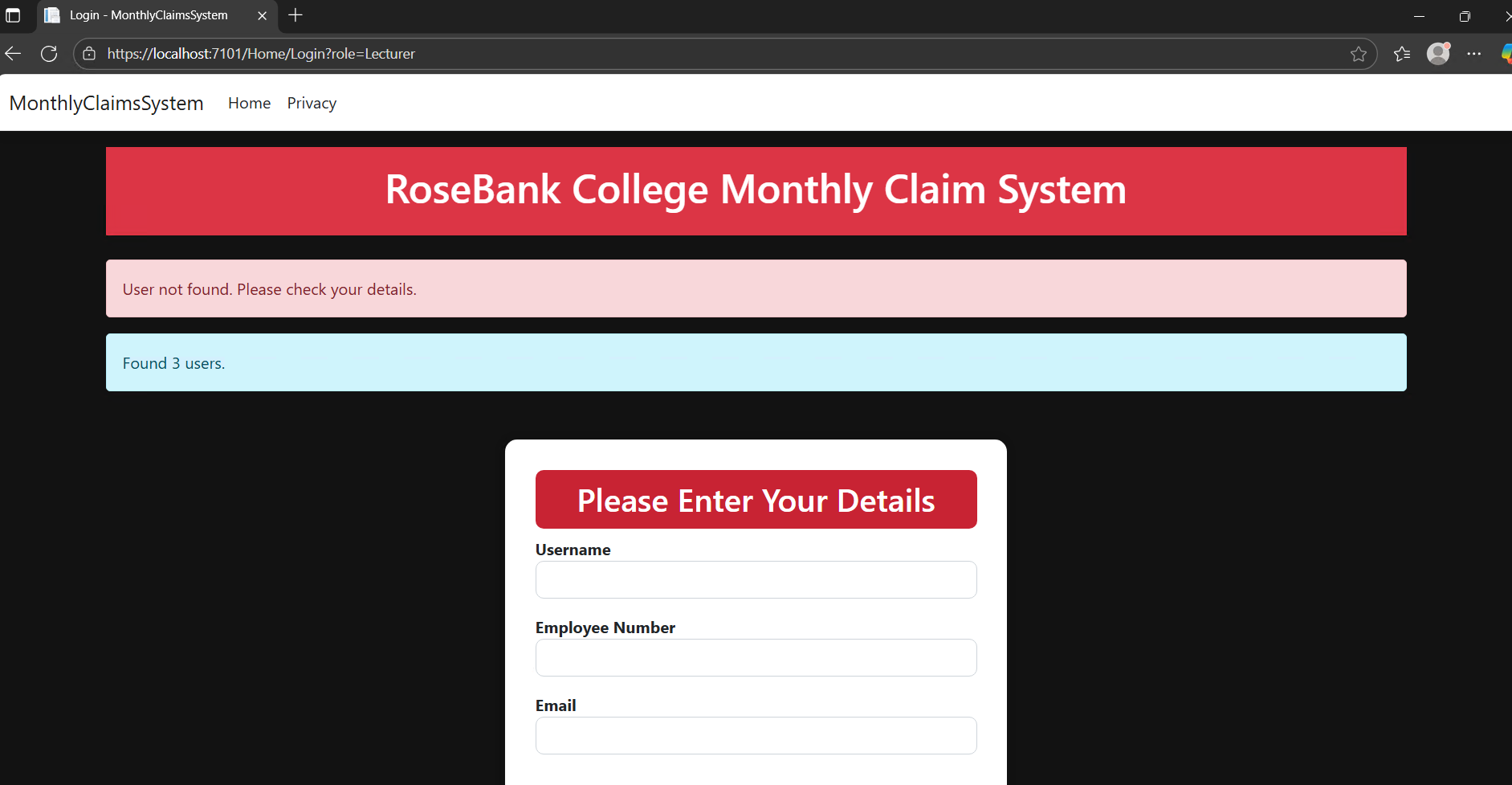
The UI shown may have minor visual inconsistencies but is more than effective for a GUI design. A screenshot of a computer

AI-generated content may be incorrect.

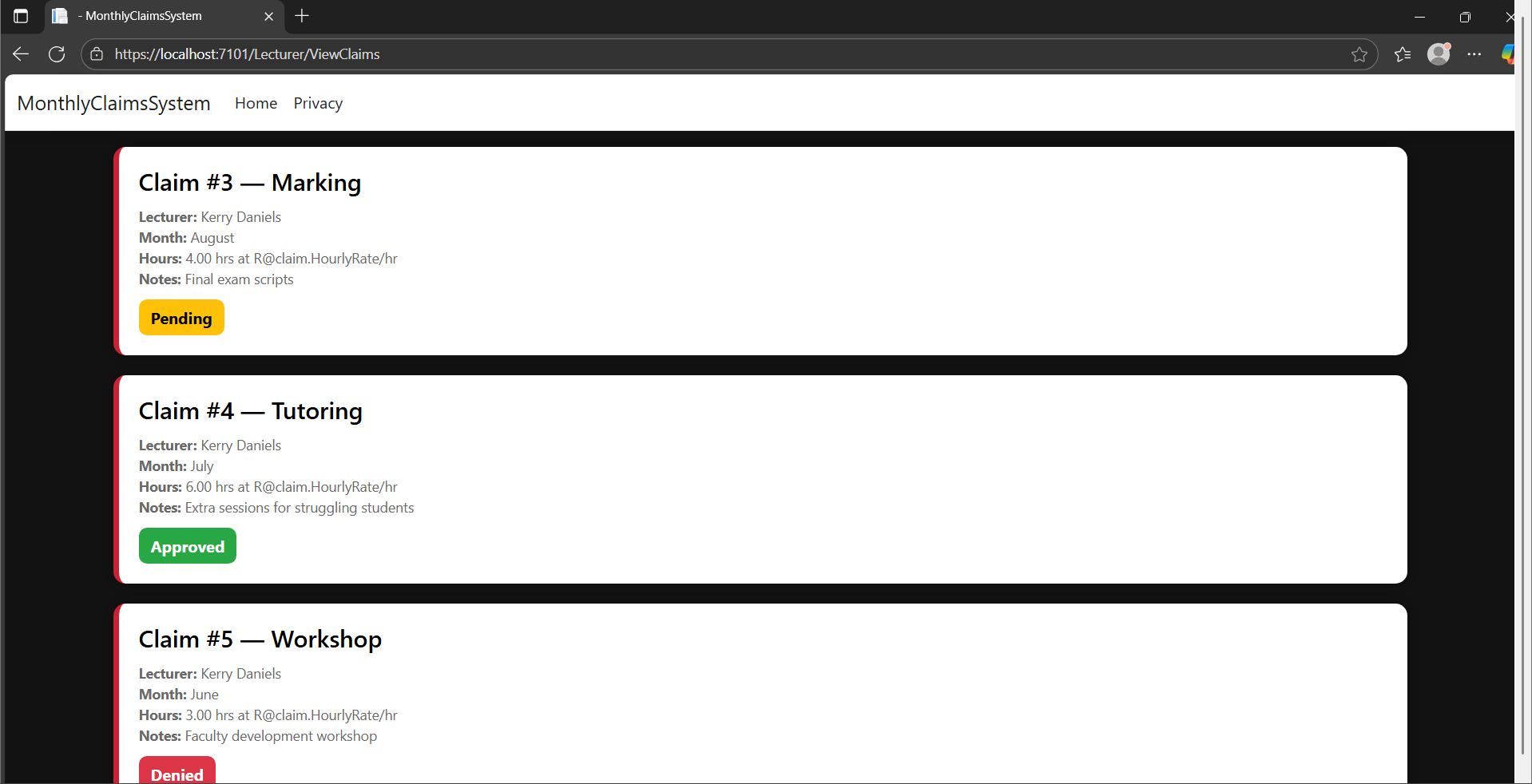
**1. Login Screen**

* **Purpose**: Allows users to authenticate using their Username, EmployeeNumber, Email, and Role (taken from the role selection).
* **Features**:
  + Input validation for all fields
  + Role-based redirection after login
  + A computer screen shot of a computer screen

    AI-generated content may be incorrect.Error message if credentials don’t match



**2. Lecturer Dashboard**

* **Purpose**: Displays all claims submitted by the logged-in lecturer.
* **Features**:
  + List of claims with type, month, hours, rate, and status
  + Full name displayed at the top
  + A screenshot of a computer

    AI-generated content may be incorrect.Status color-coded (e.g. green for approved, red for denied)

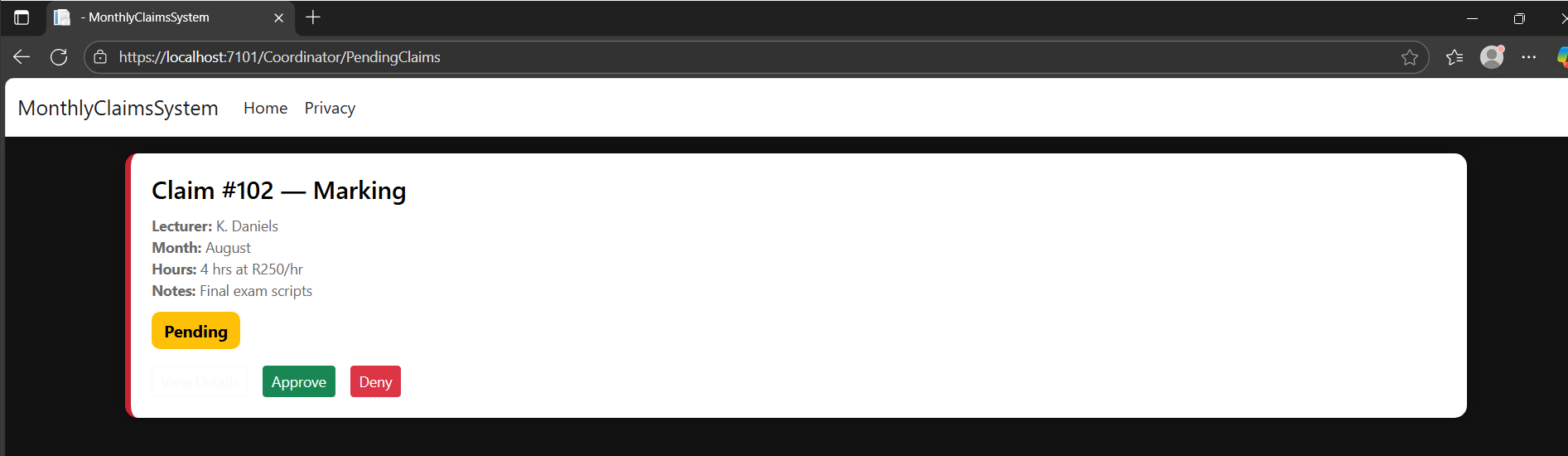
**3. Claim Submission Form**

* **Purpose**: Enables lecturers to submit new monthly claims.
* **Features**:
  + Dropdowns for claim type and month
  + Numeric input for hours and rate
  + Optional notes field
  + A screenshot of a computer

    AI-generated content may be incorrect.Auto-linking to lecturer via session EmployeeNumber

**4. Coordinator Dashboard**

* **Purpose**: Displays all pending claims for review.
* **Features**:
  + List of claims with lecturer info and claim details
  + Buttons to approve or deny each claim
  + A black and white rectangular object

    AI-generated content may be incorrect.A screenshot of a computer

    AI-generated content may be incorrect.Denial requires a reason input

A black and white rectangular object

AI-generated content may be incorrect.

# Role-Based View Breakdown

**Role-Based View Breakdown**

The Lecturer Claims Management System uses role-based access to ensure that each user interacts only with the features relevant to their responsibilities. Upon login, users are redirected to dashboards tailored to their role, with distinct views and permissions that support clarity, security, and workflow efficiency.

**Lecturer**

* **Dashboard View**: Displays all claims submitted by the logged-in lecturer.
* **Permissions**:
  + Submit new monthly claims via a structured form.
  + View status of previously submitted claims (Pending, Approved, Denied).
  + Access full claim details for personal reference.
* **Limitations**:
  + Cannot view or modify other users’ claims.
  + Cannot approve or deny claims.
  + Cannot access coordinator or manager dashboards.

**Coordinator**

* **Dashboard View**: Displays all pending, approved or denied claims submitted by lecturers.
* **Permissions**:
  + Review individual claims with full details.
  + Approve claims with a single click.
  + Deny claims with a required reason input.
  + Log status changes in the ClaimStatusLog table.
* **Limitations**:
  + Cannot submit claims themselves.
  + Cannot view claims already reviewed by other coordinators (unless designed to allow it).
  + Cannot access lecturer or manager dashboards.

**Manager *(Reserved for future implementation)***

* **Intended View**: Overview of all claim activity across roles.
* **Planned Permissions**:
  + View claims by status, type, and user.
  + Generate reports or summaries for HR or payroll purposes.
  + Monitor coordinator activity and claim turnaround times.
* **Current Status**: Role defined in the database but not yet implemented in the UI or logic.

# Navigation and Usability Features

The Lecturer Claims Management System was designed with clarity and ease-of-use at its core. Navigation is streamlined through consistent layouts, clearly labelled buttons, and role-based redirects that guide users to the correct dashboard immediately after login.

**Role-Based Redirection**

* After logging in, users are automatically redirected to their respective dashboard based on their selected role.
* This eliminates confusion and ensures that each user sees only the functionality relevant to them.

**Consistent Layouts**

* All pages follow a uniform structure: headers for context, tables for data, and action buttons placed logically.
* Design is consistent throughout the website to ensure smooth transition between pages.
* Forms are grouped by purpose (one page, one form, one function), with dropdowns, text fields, and numeric inputs clearly separated and labelled.

**Intuitive Controls**

* Buttons like “Submit Claim,” “Approve,” and “Deny” are prominently placed and easy to identify.
* Error messages (will) appear inline and in context, helping users correct mistakes without losing progress.

**Accessibility Considerations**

* Font sizes and contrast levels are chosen for readability.
* Status indicators use colour coding (e.g. green for approved, red for denied) to provide quick visual feedback.
* Navigation does not rely on hidden elements or hover-only interactions, so users only need to be able to move the mouse/select box (if using some kind of software for people why can’t use a mouse) in order to interact with the program.

**Session Awareness**

* Logged-in users retain their session across pages, allowing seamless movement between views without re-authentication.
* Session data is used to auto-link claims to the correct lecturer, reducing manual input and preventing errors.

**Minimal Click Depth**

* Most tasks (e.g. submitting a claim, reviewing a pending item) can be completed within two clicks from the dashboard.
* Users are never more than one step away from their primary action, whether it’s submitting, reviewing, or viewing history.

Project Plan

# Milestones and Phases in Part 1

|  |  |
| --- | --- |
| Phase | Milestone |
| Phase 1 Project Setup & Core Modeling | 1.1 Project scaffolding  1.2 Data models & DbContext |
| Phase 2 Authentication & Session Management | 2.1 Login UX & logic  2.2 Role-based redirects |
| Phase 3 Claim Submission & Viewing | 3.1 Lecturer claim form  3.2 Lecturer claim list |
| Phase 4 Coordinator Review & Status Logging | 4.1 Approve/Deny flows  4.2 ClaimStatusLog audit trail |
| Phase 5 Schema Refactoring & Data Integrity | 5.1 Add Name/Surname  5.2 Migrate to EmployeeNumber |
| Phase 6 Stress Testing & Seed Data | 6.1 Bulk claim inserts  6.2 Seed status logs |
| Phase 7: Acknowledge Scope Creep | 7.1 Recognizing Completion Beyond Requirements  7.2 Reflecting on Progress and Transitioning to Documentation |
| Phase 8 Documentation & Handover | 8.1 Doc structure  8.2 Screenshots & reflections |

# Task Breakdown with Dependencies

## Phase 1: Project Setup & Core Modelling

**Milestone 1.1 – Project Scaffolding**

* Create new ASP.NET Core MVC solution
* Initialize GitHub repo (add .gitignore, README.md)
* Make initial commit
* Install EF Core packages (Microsoft.EntityFrameworkCore, Tools, SqlServer)

**Milestone 1.2 – Data Models & DbContext**

* Define User and Claim model classes
* Create ClaimDbContext with DbSet<User> and DbSet<Claim>
* Add initial EF migration (Add-Migration InitialCreate)
* Apply migration to database (Update-Database) and verify tables

## Phase 2: Authentication & Session Management

**Milestone 2.1 – Login UX & Logic**

* Scaffold AccountController with Login GET/POST actions
* Build Login.cshtml view (fields: Username, EmployeeNumber, Email, Role)
* Implement session storage (HttpContext.Session.SetString("Username", …))
* Add “user not found” error feedback

**Milestone 2.2 – Role-Based Redirects**

* In Login POST, redirect based on role (Lecturer goes to Lecturer/Dashboard, Coordinator goes to Coordinator/Dashboard)
* Test login for each role and edge cases

## Phase 3: Claim Submission & Viewing

**Milestone 3.1 – Lecturer Claim Form**

* Scaffold ClaimsController and Create view for new claim
* Link claim to lecturer via EmployeeNumber
* Validate required fields and save to database

**Milestone 3.2 – Lecturer Claim List**

* Implement Index action in ClaimsController to fetch claims for logged-in lecturer
* Include full name (@claim.User.Name @claim.User.Surname) via .Include(c => c.User)
* Build Index.cshtml to display claim details and status

## Phase 4: Coordinator Review & Status Logging

**Milestone 4.1 – Approve/Deny Workflows**

* Scaffold CoordinatorController with PendingClaims view
* Implement ApproveClaim(int id) action (no reason required)
* Implement DenyClaim(int id, string reason) action (enforce non-empty reason)

**Milestone 4.2 – Audit Trail with ClaimStatusLog**

* Add ClaimStatusLog model and DbSet
* Create migration and update database
* Wire up EF Core relationship in OnModelCreating
* Display status history in lecturer’s claim view

## Phase 5: Schema Refactoring & Data Integrity

**Milestone 5.1 – Add Names for Users**

* Alter Users table: add Name, Surname columns
* Run SQL updates for existing users (kdaniels, tmokoena, kirvine)
* Update login and views to use full name

**Milestone 5.2 – Switch to EmployeeNumber Key**

* Alter Claims table: add EmployeeNumber, populate from old UserId
* Drop UserId column; rename EmployeeNumber to match model
* Break the entire project
* Update Claim/User models and OnModelCreating FK config
* Delete broken migrations, re-scaffold clean InitialCreate
* Rebuild database and verify data integrity

## Phase 6: Stress Testing & Seed Data

**Milestone 6.1 – Seed Sample Claims**

* Write SQL inserts for 10+ claims across statuses (Pending, Approved, Denied)
* Seed ClaimStatusLog for approvals/denials

**Milestone 6.2 – Validate Under Load**

* Run UI flows against bulk data (list, review, history)
* Fix edge cases (null reasons, missing navigation props)

## Phase 7: Acknowledge Scope Creep

**Milestone 7.1 – Recognizing Completion Beyond Requirements**

* Reviewed POE Part 1 requirements and realized the system already exceeded expectations
* Confirmed that login, claim submission, role-based access, and status tracking were fully functional
* Identified additional features (e.g. full name display, denial reasons, audit logging) as enhancements, not requirements
* Made the decision to stop writing new code and shift focus to documentation

**Milestone 7.2 – Reflecting on Progress and Transitioning to Documentation**

* Took stock of the development journey: schema changes, EF relationship fixes, UI polish
* Documented challenges (e.g. broken migrations, login logic) and “easy wins” (e.g. GUI layout, session handling)
* Validated that all Part 1 requirements were met and/or exceeded
* Officially transitioned into documentation mode to prepare the final submission

## Phase 8: Documentation & Handover

**Milestone 8.1 – Draft Documentation Structure**

* Set up DOCX template: Cover Page, TOC, Sections 1–11
* Outline key sections (Intro, System Overview, DB Design, etc.)

**Milestone 8.2 – Populate Content & Finalize**

* Capture screenshots of login, dashboards, forms, status history
* Insert ERD, UML class diagram, SQL snippets where necessary
* Write reflections (challenges, solutions, next steps)
* Proofread, format, and commit “Part 1 Documentation Complete”

# Timeline Summary

Dropping the formal report tone for a minute, I must clarify something about myself. I tend not to stick to timelines like this, I prefer to do bursts of work with periods of “nothing” in between. I will create a timeline for the project based on the work I have done and have yet to do but realistically I won’t be following it. It is near impossible to convert my way of working into a timeline and it is definitely impossible to predict what my schedule will look like in the coming parts. Consider the timeline below an example of a normal developers mindset and abilities and an exercise in my ability to understand and create project timelines. Hence, the work distribution for part 1 is not accurate to how it actually went down but an accurate timeline would not meet requirements.

## Part 1: Core Functionality & Foundation

**Duration**: 4 August – 9 September

**Goal**: Build a working system with login, claim submission, coordinator review, and database structure

**Week 1 (Aug 4–10): Project Kickoff**

* Set up ASP.NET Core MVC project
* Create GitHub repo and commit scaffolding
* Define initial models (User, Claim) and link SQL Server

**Week 2–3 (Aug 11–24): Authentication & Claim Submission**

* Build login form and session logic
* Implement lecturer dashboard and claim form
* Seed test users and validate login flow

**Week 4 (Aug 25–31): Coordinator Review & Status Logging**

* Create coordinator dashboard
* Add approval/denial logic and ClaimStatusLog
* Test full claim lifecycle

**Week 5 (Sept 1–7): Schema Refactor & Final Polish**

* Switch to EmployeeNumber as PK
* Rebuild EF relationships and fix migrations
* Add full name display and finalize UI

**Week 6 (Sept 8–9): Scope Reflection & Documentation**

* Acknowledge scope creep and stop coding
* Begin documentation and screenshot capture
* Submit Part 1

## Part 2: Feature Expansion & Usability

**Duration**: 10 September – 24 October

**Goal**: Add advanced features, polish UI, and improve user experience

**Week 1–2 (Sept 10–23): Planning & Prioritization**

* Review Part 2 brief
* Choose features: file uploads, search/filtering, summary views
* Sketch UI changes and database extensions

**Week 3–4 (Sept 24–Oct 7): Feature Development**

* Implement document upload for claims
* Add dashboard filtering (by month, type, status)
* Create summary views (e.g. total hours, earnings)

**Week 5–6 (Oct 8–17): Manager Role & Reporting**

* Activate Manager dashboard
* Add basic reporting tools (claim counts, approval rates)
* Test across all roles

**Week 7 (Oct 18–23): Final Polish & Documentation**

* Refine UI layout and accessibility
* Capture screenshots of new features
* Update documentation and submit Part 2

## Part 3: Reflection & Presentation

**Duration**: 25 October – 21 November

**Goal**: Reflect on the full journey and present your work professionally

**Week 1–2 (Oct 25–Nov 7): Reflection Writing**

* Document challenges, wins, and lessons learned
* Highlight key decisions and growth
* Draft reflection section for submission

**Week 3 (Nov 8–14): Presentation Prep**

* Build slide deck or summary document
* Include screenshots, diagrams, and key takeaways
* Practice explaining your system clearly

**Week 4 (Nov 15–21): Final Submission**

* Submit all documentation, code, and presentation materials
* Celebrate — full POE complete!

Version Control Summary

# GitHub Setup

Github is public and accessible for markers.

Link: https://github.com/KylahRC/PROG2B\_POE\_ST10435066.git

# Commit Strategy

I will usually commit my work after I have made some amount of meaningful progress. I don’t commit too frequently as I often just forget to do so. I try to explain all I’ve done in my commit messages to make up for this.

AI Usage

Throughout the development of this POE, I used Microsoft Copilot in a supportive and iterative role, particularly during the planning and documentation phases. Rather than accepting AI-generated content outright, I engaged in a (often lengthy) back-and-forth process where Copilot offered suggestions, and I refined, questioned, or reshaped them to align with my own understanding and goals. This collaborative dynamic reflects the “human-in-the-loop” principle outlined in the IIE033 policy, where students maintain full oversight and control over AI contributions (IIE033 section 6.1, point 7).

In technical areas such as database design and Entity Framework configuration, Copilot provided examples and explanations that I evaluated and adapted. While code itself must follow strict syntax to function, I ensured that every implementation was contextually appropriate, tested, and aligned with my own schema decisions. Importantly, Part 1 of the POE did not require any code submission, only GUI design and documentation, meaning that all code-related work was done voluntarily to deepen my understanding of how the program should feel and thus allow me to design it alongside the functionality as well as prepare for future phases. This reinforces that Copilot’s involvement supported learning as I do not feel we know/have been taught enough to be able to write this kind of program all on our own, not simply assessment completion.

In documentation, Copilot helped me structure sections, clarify technical concepts, and reflect on my development process. I do not consciously think about the reasons why I do the things I do when developing a system, so I may ask an AI such as Copilot to help point out the subconscious decisions and preferences that I displayed during my back-and-forth with it. It has a better memory than I do. I also do not simply take what it says as gospel, I check the suggestions thoroughly to see if they sound reasonably like something I would do.

I did not label every sentence that Copilot was involved with in any way, shape or form individually, but I have disclosed the nature of AI involvement and maintained full authorship over the final content under the above heading. This aligns with the policy’s emphasis on transparency, ethical responsibility, and pedagogical integrity, ensuring that AI enhances learning without compromising originality (IIE033 section 6.2, point 7 and section 9, point 7).