

# PROG6212 POE POWERPOINT

ST10440733

LECTURER CLAIM SYSTEM



Updates based on lecturer feedback

## Lecturer Claim implementation of feature

- ▶ Mark : 17 points out of 20
- ▶ Feedback The feature is implemented effectively, meeting basic requirements and functioning adequately.
- ▶ To get full marks: It should exceed basic requirements and enhancing usability.
- ▶ Update: I made the claim form feel personal and smart. It now greets lecturers by name and instantly shows them how much they'll earn as they type in their hours, which builds trust and prevents errors. This focus on a clear, guided experience transforms a basic task into a smooth and professional process, showing a real understanding of what users need.

```
//Auto calculation for total amount
[HttpPost]

public IActionResult CalculateAmount(int hoursWorked)
{
    if (!IsLecturerOrHR()) return Json(new { error = "Not authorized" });

    var lecturer = GetLoggedInLecturer();
    if (lecturer == null) return Json(new { error = "Lecturer not found" });

    var hourlyRate = 250;
    var amount = hoursWorked * hourlyRate;

    return Json(new
    {
        success = true,
        amount = amount.ToString("F2"),
        hourlyRate = hourlyRate,
        validation = hoursWorked <= 180 ? "valid" : "exceeds_limit"
    });
}

<div class="card shadow-sm">
  <div class="card-header bg-dark text-white">
    <h5 class="mb-0">Claim Form - @ViewBag.LecturerName</h5>
  </div>
  <div class="card-body">
    <form asp-action="MakeAClaim" method="post" enctype="multipart/form-data" id="claimForm">
      @Html.AntiForgeryToken()

      <input type="hidden" asp-for="HourlyRate" value="250" />

      <div class="row mb-3">
        <div class="col-md-6">
          <label class="form-label">Your Hourly Rate</label>
          <input type="text" class="form-control" value="R @ViewBag.HourlyRate" readonly>
          <small class="text-muted">Set by HR Department</small>
        </div>
        <div class="col-md-6">
          <label asp-for="HoursWorked" class="form-label">Hours Worked *</label>
          <input asp-for="HoursWorked" class="form-control" id="hoursInput" required />
          <span asp-validation-for="HoursWorked" class="text-danger"></span>
          <small class="text-muted">Maximum 180 hours per month</small>
        </div>
      </div>
    </div>
  </div>
</div>
```

# Claim Status Tracking: Implementation of Tracking System

- ▶ Mark: 9 points out of 10
- ▶ Feedback: The tracking system is implemented effectively, updating claim status reasonably accurately and promptly.
- ▶ To get full marks: Provide precision and reliability and real-time and accurate updates on claim status
- ▶ Updates: I rebuilt the status tracker to work like a reliable delivery notification. Lecturers can now see their claim's entire journey in real-time, complete with accurate timestamps and every piece of feedback in order. They always know exactly where their claim is and what's happening with it.

```
//Lecturer tracks the claims status as it goes through the process
6 references
public async Task<IActionResult> TrackClaimStatus()
{
    if (!IsLecturerOrHR()) return RedirectToAction("Login", "Auth");

    var lecturer = GetLoggedInLecturer();
    if (lecturer == null) return RedirectToAction("Login", "Auth");

    var claims = await _context.Claims
        .Include(c => c.FeedbackMessages)
        .Where(c => c.LecturerId == lecturer.LecturerId)
        .OrderByDescending(c => c.CreatedAt)
        .ToListAsync();

    return View("~/Views/Claim/TrackClaimStatus.cshtml", claims);
}
```

## Programme Coordinators and Managers View: Design of View

► Mark: 17 points out of 20

► Feedback: The design of the view is clear and organised, facilitating easy verification of claims.

► To get full marks: The design of the view is highly intuitive and well-structured, enhancing the verification process significantly.

► Updates: I transformed the verification views into an intuitive workflow that guides coordinators and managers seamlessly through the approval process. By implementing clear action buttons with built-in modals for feedback, the system ensures every decision is documented and transparent. This thoughtful design eliminates confusion and makes verifying claims a smooth, professional experience that significantly enhances both efficiency and accountability.

```
<div class="modal fade" id="forwardModal@({claim.ClaimId})" tabindex="-1">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title">Forward Claim to Manager</h5>
        <button type="button" class="btn-close" data-bs-dismiss="modal"></button>
      </div>
      <form asp-action="ForwardClaim" method="post">
        @Html.AntiForgeryToken()
        <input type="hidden" name="id" value="@claim.ClaimId" />
        <div class="modal-body">
          <div class="form-group">
            <label>Feedback Message (Optional)</label>
            <textarea name="coordinatorMessage" class="form-control"
              placeholder="Add feedback for the manager..."></textarea>
          </div>
        </div>
        <div class="modal-footer">
          <button type="button" class="btn btn-secondary" data-bs-dismiss="modal">Cancel</button>
          <button type="submit" class="btn btn-success">Forward</button>
        </div>
      </form>
    </div>
  </div>
</div>
```



# Design and Key Features for HR

# HR Dashboard and Overview

- ▶ Provides HR with a complete overview of the entire system's activity and performance
- ▶ Shows real-time counts of pending invoices, total payments, and user statistics for quick decision making
- ▶ Displays recent approved claims so HR can see the latest activity that needs attention
- ▶ Serves as the main control centre where HR can monitor all aspects of the claim system
- ▶ Helps prioritize work by showing what tasks are most urgent and what's happening in the system

```
0 references
public async Task<IActionResult> Index()
{
    if (!IsHR()) return RedirectToAction("Login", "Auth");

    var pendingInvoices = await _context.Claims
        .CountAsync(c => c.Status == ClaimStatus.Approved);

    var totalInvoices = await _context.Invoices.CountAsync();
    var pendingPayments = await _context.Invoices.CountAsync();
    var totalLecturers = await _context.Lecturers.CountAsync();
    var totalUsers = await _context.Users.CountAsync();

    //Get recent approved claims for the activity section
    var recentClaims = await _context.Claims
        .Include(c => c.Lecturer)
        .Where(c => c.Status == ClaimStatus.Approved)
        .OrderByDescending(c => c.ApprovedAt)
        .Take(5)
        .ToListAsync();

    ViewBag.PendingInvoicesCount = pendingInvoices;
    ViewBag.TotalInvoicesCount = totalInvoices;
    ViewBag.PendingPaymentsCount = pendingPayments;
    ViewBag.TotalLecturersCount = totalLecturers;
    ViewBag.TotalUsersCount = totalUsers;
    ViewBag.RecentClaims = recentClaims;

    return View();
}
```

# Key Feature 1 – Approved Claims and Invoice Generation Workflow

- ▶ Shows all claims that have been approved by managers but don't have invoices yet
- ▶ Displays lecturer information with each claim so HR knows who submitted the work
- ▶ Filters out claims that already have invoices to avoid duplicate processing
- ▶ Orders claims by creation date to ensure oldest claims get processed first
- ▶ Creates a clear workflow where HR can see exactly what needs invoice generation

```
//View approved claims that need invoices
0 references
public async Task<IActionResult> ApprovedClaims()
{
    if (!IsHR()) return RedirectToAction("Login", "Auth");

    var claims = await _context.Claims
        .Include(c => c.Lecturer)
        .Where(c => c.Status == ClaimStatus.Approved && string.IsNullOrEmpty(c.InvoiceNumber))
        .OrderBy(c => c.CreatedAt)
        .ToListAsync();

    return View(claims);
}
```

# Key Feature 2 – User Creation

- ▶ Prevents duplicate usernames by checking the database before creating new accounts
- ▶ Automatically sets up new users with correct creation dates and active status
- ▶ Applies the standard lecturer rate of 250 while other roles get zero rate automatically
- ▶ Validates all information thoroughly before saving to ensure data quality
- ▶ Provides clear feedback messages so HR knows immediately if the user was created successfully
- ▶ Handles any errors gracefully with helpful messages about what went wrong

```
[HttpPost]
[ValidateAntiForgeryToken]
0 references
public async Task<IActionResult> CreateUser(User u
{
    if (!IsHR()) return RedirectToAction("Login", "Auth");

    //Check if username already exists
    if (_context.Users.Any(u => u.Username == user.Username))
    {
        ModelState.AddModelError("Username", "Username already exist");
        return View(user);
    }

    //Set default values
    user.CreatedAt = DateTime.Now;
    user.IsActive = true;

    //Set hourly rate
    user.HourlyRate = user.Role == "Lecturer" ? 250 : 0;

    if (ModelState.IsValid)
    {
        try
        {
            _context.Users.Add(user);
            await _context.SaveChangesAsync();

            //User created successfully message
            TempData["Success"] = "User created successfully!";
            return RedirectToAction("ManageUsers");
        }
        catch (Exception ex)
        {
            ModelState.AddModelError("", $"Error creating user: {ex.Message}");
        }
    }
    return View(user);
}
```

## Key feature 3 – User Management System

- Gives HR complete control over all user accounts in one organized view
- Shows only active users to keep the list clean and focused on current staff members
- Displays both user information and their related lecturer profiles for full context
- Serves as the starting point for creating new users, editing existing ones, or managing access
- Ensures HR can quickly find and manage any user without searching through multiple screens

```
//user management by HR
0 references
public IActionResult ManageUsers()
{
    if (!IsHR()) return RedirectToAction("Login", "Aut

    var users = _context.Users
        .Include(u => u.Lecturer)
        .Where(u => u.IsActive)
        .ToList();
    return View(users);
}
```

```

//Invoice creation
0 references
public async Task<IActionResult> CreateInvoice(int id)
{
    if (!IsHR()) return RedirectToAction("Login", "Auth");

    var claim = await _context.Claims
        .Include(c => c.Lecturer)
        .FirstOrDefaultAsync(c => c.ClaimId == id);

    if (claim == null)
    {
        TempData["Error"] = "Claim not found!";
        return RedirectToAction("ApprovedClaims");
    }

    //Invoice number creation
    var invoiceNumber = $"INV-{DateTime.Now:yyyyMMdd}-{claim.ClaimId}";

    claim.InvoiceNumber = invoiceNumber;

    //Invoice record
    var invoice = new Invoice
    {
        InvoiceNumber = invoiceNumber,
        ClaimId = claim.ClaimId,
        LecturerId = (int)claim.LecturerId,
        Amount = claim.Amount,
        GeneratedDate = DateTime.Now
    };

    _context.Invoices.Add(invoice);
    await _context.SaveChangesAsync();

    TempData["Success"] = $"Invoice {invoiceNumber} created successfully";
    return RedirectToAction("ApprovedClaims");
}

```

## Key Feature 4: Invoice Number Generation System

- Creates unique invoice numbers that include the date and claim ID for easy tracking
- Links invoices directly to their original claims so everything stays connected
- Records the exact generation date and time for audit purposes
- Updates the claim record with the invoice number to prevent duplicate invoices
- Provides immediate confirmation with the actual invoice number for reference
- Ensures both the claim and invoice records get updated together for data consistency

# Key Feature 5: Data Integrity & Error Handling

```
//Delete invoice option added for space
[HttpPost]
0 references
public async Task<ActionResult> DeleteInvoice(int id)
{
    if (!IsHR()) return RedirectToAction("Login", "Auth");

    var invoice = await _context.Invoices.FindAsync(id);
    if (invoice == null)
    {
        TempData["Error"] = "Invoice not found!";
        return RedirectToAction("Invoices");
    }

    try
    {
        _context.Invoices.Remove(invoice);
        await _context.SaveChangesAsync();

        TempData["Success"] = "Invoice deleted successfully!";
    }
    catch (Exception ex)
    {
        TempData["Error"] = "Error deleting invoice: " + ex.Message;
    }
}
```

- ▶ Checks that invoices exist before trying to delete them to prevent errors
- ▶ Uses try-catch blocks to handle any unexpected problems during database operations
- ▶ Provides clear success messages when operations complete successfully
- ▶ Gives helpful error messages that explain what went wrong when operations fail
- ▶ Ensures users are always returned to the appropriate screen after each operation
- ▶ Maintains data consistency by properly handling both successful and failed operations

# Key feature 6: Invoice Management & Payment Tracking

- ▶ Shows all invoices with their related claims and lecturer information in one view
- ▶ Displays invoices with the most recent ones first for easy access to current work
- ▶ Allows HR to mark invoices as paid with a single click when payment is processed
- ▶ Provides immediate confirmation when payment status is updated successfully
- ▶ Handles situations where invoices might not be found with clear error messages
- ▶ Gives HR complete visibility into which invoices are paid and which are still pending

```
//View all invoices
0 references
public async Task<IActionResult> Invoices()
{
    if (!IsHR()) return RedirectToAction("Login", "Auth");

    var invoices = await _context.Invoices
        .Include(i => i.Claim)
        .Include(i => i.Lecturer)
        .OrderByDescending(i => i.GeneratedDate)
        .ToListAsync();

    return View(invoices);
}

//State invoice as paid
0 references
public async Task<IActionResult> MarkAsPaid(int id)
{
    if (!IsHR()) return RedirectToAction("Login", "Auth");

    var invoice = await _context.Invoices.FindAsync(id);
    if (invoice != null)
    {
        invoice.IsPaid = true;
        await _context.SaveChangesAsync();
        TempData["Success"] = "Invoice marked as paid!";
    }
    else
    {
        TempData["Error"] = "Invoice not found!";
    }

    return RedirectToAction("Invoices");
}
```

# Key feature 7: Security & Authorization Framework

- ▶ Checks that users are properly logged in and have HR role before allowing access
- ▶ Prevents HR staff from accidentally deleting their own accounts while managing users
- ▶ Automatically redirects unauthorized users to login page for security
- ▶ Uses session tracking to maintain user identity and role throughout their work session
- ▶ Provides clear error messages when users try to perform actions they're not allowed to do
- ▶ Ensures that only authorized personnel can access sensitive HR functions and data

```
//Check if user is HR
12 references
private bool IsHR()
{
    return _sessionService.IsUserLoggedIn() && _sessionService.GetUserRole() == "HR";
}
```

```
//Delete User method
[HttpPost]
0 references
public async Task<IActionResult> DeleteUser(int id)
{
    if (!IsHR()) return RedirectToAction("Login", "Auth");

    var user = await _context.Users.FindAsync(id);
    if (user == null)
    {
        TempData["Error"] = "User not found!";
        return RedirectToAction("ManageUsers");
    }

    if (user.UserId == _sessionService.GetUserId())
    {
        TempData["Error"] = "You cannot delete your own account!";
        return RedirectToAction("ManageUsers");
    }

    try
    {
        user.IsActive = false;
        await _context.SaveChangesAsync();

        TempData["Success"] = "User deleted successfully!";
    }
    catch (Exception ex)
    {
        TempData["Error"] = "Error deleting user: " + ex.Message;
    }
}
```