SE1 Student Tracker – C# and Razor Structure Guide

Version: **1.0** | Date: October 27, 2025

# 1. Overview

This document explains how the SE1 Student Tracker app is organized in C# and Razor Pages so that it connects cleanly to the Oracle database and reads/writes data reliably. It covers the folder layout, dependency injection, services, repositories, POCO models, and the Razor Page pattern (cshtml + PageModel). Use this as a contributor’s guide for new teammates.

# 2. Project Layout (Suggested)

Below is the recommended structure. Your repo may already contain many of these files; align to this layout:

SE1StudentTracker/  
├─ SE1StudentTracker.sln  
├─ SE1StudentTracker/  
│ ├─ Program.cs  
│ ├─ appsettings.json  
│ ├─ appsettings.Development.json  
│ ├─ Models/  
│ │ ├─ Role.cs  
│ │ ├─ UserAccount.cs  
│ │ ├─ Course.cs  
│ │ ├─ Section.cs  
│ │ └─ TimeSession[Full].cs  
│ ├─ Services/  
│ │ ├─ IOracleService.cs  
│ │ └─ OracleService.cs  
│ ├─ Repositories/  
│ │ ├─ TimeSessionRepository.cs  
│ │ └─ UserRepository.cs  
│ ├─ Pages/  
│ │ ├─ Index.cshtml  
│ │ ├─ Index.cshtml.cs  
│ │ └─ TimeSessions/  
│ │ ├─ Index.cshtml  
│ │ └─ Index.cshtml.cs  
│ └─ wwwroot/  
│ └─ site.js  
└─ DbSmokeTest/ (optional console health check)  
 ├─ Program.cs  
 └─ DbSmokeTest.csproj

# 3. Configuration: Connection Strings

Connection strings live in configuration, not hard-coded. In development, we keep a local connection string in appsettings.Development.json and load it through DI.

## 3.1 Example: appsettings.Development.json

{  
 "DetailedErrors": true,  
 "Logging": {  
 "LogLevel": {  
 "Default": "Information",  
 "Microsoft.AspNetCore": "Warning"  
 }  
 },  
 "ConnectionStrings": {  
 "StudentTracker": "User Id=STUDENT\_TRACKER;Password=Strong#Password1;Data Source=localhost:1521/XEPDB1;"  
 }  
}

# 4. Program Startup (Program.cs)

Program.cs wires up Razor Pages and registers our Oracle service and repositories using dependency injection. We use a scoped lifetime for database access to keep connections request-bound and predictable.

using SE1StudentTracker.Services;  
using SE1StudentTracker.Repositories;  
  
var builder = WebApplication.CreateBuilder(args);  
  
builder.Services.AddRazorPages();  
  
var connStr = builder.Configuration.GetConnectionString("StudentTracker")  
 ?? throw new InvalidOperationException("Missing ConnectionStrings:StudentTracker");  
  
builder.Services.AddScoped<IOracleService>(sp => new OracleService(connStr));  
  
builder.Services.AddScoped<TimeSessionRepository>();  
builder.Services.AddScoped<UserRepository>();  
  
var app = builder.Build();  
  
if (!app.Environment.IsDevelopment())  
{  
 app.UseExceptionHandler("/Error");  
 app.UseHsts();  
}  
  
app.UseHttpsRedirection();  
app.UseStaticFiles();  
app.UseRouting();  
app.UseAuthorization();  
  
// Optional: DB health endpoint  
app.MapGet("/health/db", async (IOracleService db) =>  
{  
 var ok = await db.QuerySingleAsync<int>("SELECT 1 AS ok FROM dual", r => r.GetInt32(0));  
 return Results.Ok(new { database = "Oracle", ok = ok == 1 });  
});  
  
app.MapRazorPages();  
app.Run();

# 5. Services Layer

The Services layer encapsulates raw ADO.NET calls to Oracle. It exposes three primitives used by repositories:  
• ExecuteAsync – for INSERT/UPDATE/DELETE  
• QuerySingleAsync<T> – for one-row/one-object queries  
• QueryAsync<T> – for multi-row queries

## 5.1 IOracleService.cs

public interface IOracleService  
{  
 Task<int> ExecuteAsync(string sql, IEnumerable<(string name, object? value)>? parameters = null);  
 Task<T?> QuerySingleAsync<T>(string sql, Func<IDataRecord, T> map, IEnumerable<(string name, object? value)>? parameters = null);  
 Task<List<T>> QueryAsync<T>(string sql, Func<IDataRecord, T> map, IEnumerable<(string name, object? value)>? parameters = null);  
}

## 5.2 OracleService.cs (Key Points)

* Uses Oracle.ManagedDataAccess.Core; opens connections on demand.
* BindByName = true ensures parameter names match SQL placeholders.
* Disposes connections and readers via using/await using to prevent leaks.

public class OracleService : IOracleService  
{  
 private readonly string \_connectionString;  
 public OracleService(string connectionString) => \_connectionString = connectionString;  
  
 private OracleConnection CreateConnection() => new OracleConnection(\_connectionString);  
  
 public async Task<int> ExecuteAsync(string sql, IEnumerable<(string name, object? value)>? parameters = null)  
 { /\* ... \*/ }  
  
 public async Task<T?> QuerySingleAsync<T>(string sql, Func<IDataRecord, T> map, IEnumerable<(string name, object? value)>? parameters = null)  
 { /\* ... \*/ }  
  
 public async Task<List<T>> QueryAsync<T>(string sql, Func<IDataRecord, T> map, IEnumerable<(string name, object? value)>? parameters = null)  
 { /\* ... \*/ }  
}

# 6. Models (POCOs)

Models are plain C# classes matching table/view columns. Keep names and types aligned to your schema. These map database rows to strongly-typed objects for the UI.

public class TimeSessionFull  
{  
 public int SessionId { get; set; }  
 public int UserId { get; set; }  
 public string FirstName { get; set; } = "";  
 public string LastName { get; set; } = "";  
 public string Email { get; set; } = "";  
 public int RoleId { get; set; }  
 public string RoleName { get; set; } = "";  
 public string SessionType { get; set; } = "";  
 public string? LocationText { get; set; }  
 public int? SectionId { get; set; }  
 public string? SectionCode { get; set; }  
 public string? CourseCode { get; set; }  
 public string? CourseName { get; set; }  
 public DateTime ClockIn { get; set; }  
 public DateTime? ClockOut { get; set; }  
}

# 7. Repositories

Repositories hold SQL and return typed models. They depend on IOracleService and keep data access out of the UI.  
Example: TimeSessionRepository reading from the v\_timesession\_full view and clocking in/out.

public class TimeSessionRepository  
{  
 private readonly IOracleService \_db;  
 public TimeSessionRepository(IOracleService db) => \_db = db;  
  
 public Task<List<TimeSessionFull>> GetRecentSessionsAsync(int take = 100)  
 {  
 string sql = @"  
 SELECT session\_id, user\_id, first\_name, last\_name, email, role\_id, role\_name,  
 session\_type, location\_text, section\_id, section\_code, course\_code, course\_name,  
 clock\_in, clock\_out  
 FROM v\_timesession\_full  
 ORDER BY clock\_in DESC  
 FETCH FIRST :take ROWS ONLY";  
  
 return \_db.QueryAsync(sql,  
 r => new TimeSessionFull  
 {  
 SessionId = r.GetInt32(r.GetOrdinal("session\_id")),  
 UserId = r.GetInt32(r.GetOrdinal("user\_id")),  
 FirstName = r.GetString(r.GetOrdinal("first\_name")),  
 LastName = r.GetString(r.GetOrdinal("last\_name")),  
 Email = r.GetString(r.GetOrdinal("email")),  
 RoleId = r.GetInt32(r.GetOrdinal("role\_id")),  
 RoleName = r.GetString(r.GetOrdinal("role\_name")),  
 SessionType = r.GetString(r.GetOrdinal("session\_type")),  
 LocationText= r.IsDBNull(r.GetOrdinal("location\_text")) ? null : r.GetString(r.GetOrdinal("location\_text")),  
 SectionId = r.IsDBNull(r.GetOrdinal("section\_id")) ? (int?)null : r.GetInt32(r.GetOrdinal("section\_id")),  
 SectionCode = r.IsDBNull(r.GetOrdinal("section\_code")) ? null : r.GetString(r.GetOrdinal("section\_code")),  
 CourseCode = r.IsDBNull(r.GetOrdinal("course\_code")) ? null : r.GetString(r.GetOrdinal("course\_code")),  
 CourseName = r.IsDBNull(r.GetOrdinal("course\_name")) ? null : r.GetString(r.GetOrdinal("course\_name")),  
 ClockIn = r.GetDateTime(r.GetOrdinal("clock\_in")),  
 ClockOut = r.IsDBNull(r.GetOrdinal("clock\_out")) ? (DateTime?)null : r.GetDateTime(r.GetOrdinal("clock\_out"))  
 },  
 new[] { (":take", (object)take) });  
 }  
  
 public Task<int> ClockInAsync(int userId, string sessionType, string? locationText, int? sectionId)  
 {  
 string sql = @"  
 INSERT INTO time\_session (session\_id, user\_id, session\_type, location\_text, section\_id, clock\_in)  
 VALUES (time\_session\_seq.NEXTVAL, :user\_id, :session\_type, :location\_text, :section\_id, SYSTIMESTAMP)";  
 return \_db.ExecuteAsync(sql, new (string, object?)[] {  
 (":user\_id", userId),  
 (":session\_type", sessionType),  
 (":location\_text", (object?)locationText ?? DBNull.Value),  
 (":section\_id", (object?)sectionId ?? DBNull.Value)  
 });  
 }  
  
 public Task<int> ClockOutAsync(int sessionId)  
 {  
 string sql = @"UPDATE time\_session SET clock\_out = SYSTIMESTAMP WHERE session\_id = :session\_id";  
 return \_db.ExecuteAsync(sql, new[] { (":session\_id", (object)sessionId) });  
 }  
}

# 8. Razor Pages Pattern (cshtml + PageModel)

Each Razor Page has two files:  
• \*.cshtml – the markup (view) that renders HTML  
• \*.cshtml.cs – the PageModel (code-behind) that handles requests, obtains data from repositories, and exposes it to the view  
This keeps UI and data access separate.

## 8.1 Example PageModel – Pages/TimeSessions/Index.cshtml.cs

public class IndexModel : PageModel  
{  
 private readonly TimeSessionRepository \_repo;  
 public IndexModel(TimeSessionRepository repo) => \_repo = repo;  
  
 public List<TimeSessionFull> Sessions { get; private set; } = new();  
  
 public async Task OnGetAsync()  
 {  
 Sessions = await \_repo.GetRecentSessionsAsync(100);  
 }  
}

## 8.2 Example View – Pages/TimeSessions/Index.cshtml

@page  
@model SE1StudentTracker.Pages.TimeSessions.IndexModel  
@{  
 ViewData["Title"] = "Recent Sessions";  
}  
<h2>Recent Sessions</h2>  
<table class="table">  
 <thead>  
 <tr>  
 <th>When</th>  
 <th>User</th>  
 <th>Role</th>  
 <th>Type</th>  
 <th>Course/Section</th>  
 <th>Location</th>  
 <th>Clock Out</th>  
 </tr>  
 </thead>  
 <tbody>  
 @foreach (var s in Model.Sessions)  
 {  
 <tr>  
 <td>@s.ClockIn</td>  
 <td>@s.FirstName @s.LastName</td>  
 <td>@s.RoleName</td>  
 <td>@s.SessionType</td>  
 <td>@(string.IsNullOrEmpty(s.CourseCode) ? "-" : $"{s.CourseCode} · {s.SectionCode}")</td>  
 <td>@(s.LocationText ?? "-")</td>  
 <td>@(s.ClockOut?.ToString() ?? "-")</td>  
 </tr>  
 }  
 </tbody>  
</table>

# 9. Health Check Endpoint

A small GET endpoint confirms that the app can open a connection and run a simple query. This is extremely helpful for debugging environment and credential issues.

app.MapGet("/health/db", async (IOracleService db) =>  
{  
 var ok = await db.QuerySingleAsync<int>("SELECT 1 AS ok FROM dual", r => r.GetInt32(0));  
 return Results.Ok(new { database = "Oracle", ok = ok == 1 });  
});

# 10. Error Handling & Logging

* Wrap repository writes in try/catch and bubble up a friendly message to the UI.
* Log exceptions with context (userId, sessionId) but avoid logging secrets.
* Prefer granular repository methods (ClockInAsync/ClockOutAsync) instead of pushing SQL into page models.

# 11. Extending the Pattern: Adding a New Entity

1. Create a POCO model that matches the table/view columns.
2. Add repository methods with parameterized SQL and typed mappers.
3. Register the repository in Program.cs (AddScoped).
4. Create a Razor Page (cshtml + cshtml.cs) that asks the repository for data.
5. Add links/navigation and basic tables/forms in the view.

# 12. Oracle Tips & Gotchas

* Use BindByName = true and ensure parameter names match placeholders (e.g., :user\_id).
* Use SYSTIMESTAMP for server-controlled timestamps; prefer UTC if writing from C#.
* FETCH FIRST n ROWS ONLY requires 12c+. If using older XE, use ROWNUM filtering instead.
* Verify grants: the app user must have SELECT on views and INSERT/UPDATE on target tables.

# 13. Deployment Notes (High-level)

* Store production connection strings in environment variables or user secrets (not in source control).
* Ensure Oracle client networking (listener/tnsnames or Easy Connect) is accessible from your app host.
* Use HTTPS and app-level authz/authn for any pages that modify data.

— End of Guide —