Employee Tracker - A BCX Sample Application

(According to the criteria) **By Rikus Botha**

1. Welcome to Employee Tracker, a simple application that
    a. Keeps track of casually employed employees
    b. Maps tasks assigned to them,
    c. Keeps rates at which they are paid
    d. States what they are owed in total over the entire period

    The application has been written in an N-Layered fashion, and a business logic layer has been omitted for the sake of brevity - however, to implement this simply means creating a project that intercepts calls between the HTTPAPI and DAL projects.

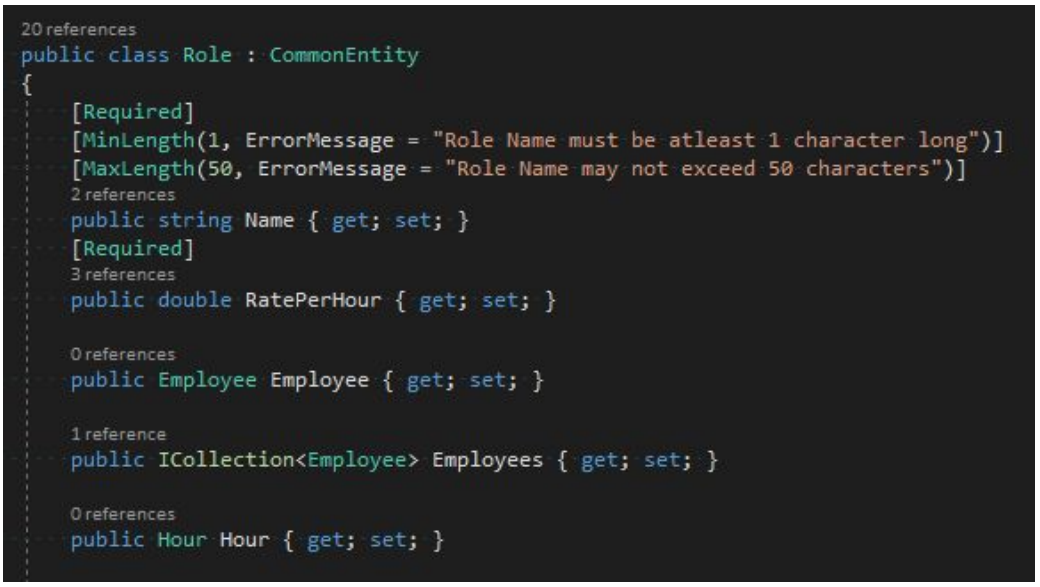
    Structure:
    The solution consists of several projects, all referenced appropriately -
    BCX_CORE contains Entities and Dtos
    BCX_DAL contains the repositories and DBContext configuration
    BCX_HTTPAPI contains REST API controllers that reference BCX_DAL
    BCX_Application contains the ASP.NET application, running views and controllers. Calls are made via HTTP API to BCX_HTTPAPI project from this project.

    Furthermore, the solution uses ASP.NET Core 3.1, MVC, REST API, Javascript (some AJAX), Bootstrap, MS SQL Server, Swashbuckle (SWAGGER) and EF Core as principal technologies.

    To Setup:
    1. Clone/Download the repository to your local environment
    2. Set **BCX.HTTPAPI** project as default startup project
    3. Open up the package management console
    4. Within the above console, set the **BCX.DAL** project as default project
    5. Run the command **update-database** and wait for it to complete
    6. Verify that your localdb instance contains the **BCX_DB** database
    7. Right click on the solution > Setup Multiple Startup Projects
    8. Ensure you set BCX_HTTPAPI and BCX_Application to start and click Ok.

2. Tables in the database provider (MS SQL Server) are as follows:

a. Roles Table // A parent entity that contains role based information



```
20 references
public class Role : CommonEntity
{
    [Required]
    [MinLength(1, ErrorMessage = "Role Name must be atleast 1 character long")]
    [MaxLength(50, ErrorMessage = "Role Name may not exceed 50 characters")]
    2 references
    public string Name { get; set; }
    [Required]
    3 references
    public double RatePerHour { get; set; }

    0 references
    public Employee Employee { get; set; }

    1 reference
    public ICollection<Employee> Employees { get; set; }

    0 references
    public Hour Hour { get; set; }
```

You'll note, all entities inherit from the CommonEntity Class, which supplies common fields such as Audits, CANCELLED and ID fields.

All noted tables in this section make use of Navigation properties and foreign keys to ensure data integrity as well as keeping the database in a 3rd normal form.

b. Employees Table // A dependent child table that contains all casual employees' data

```
29 references
public class Employee : CommonEntity
{
    [Required]
    [MinLength(1, ErrorMessage = "First Name must be atleast 2 characters long")]
    [MaxLength(50, ErrorMessage = "First Name may not exceed 50 characters")]
    5 references
    public string FirstName { get; set; }
    [Required]
    [MinLength(1, ErrorMessage = "Last Name must be atleast 2 characters long")]
    [MaxLength(50, ErrorMessage = "Last Name may not exceed 50 characters")]
    5 references
    public string LastName { get; set; }

    0 references
    public string ImagePath { get; set; }
    //Nav Props

    4 references
    public Role Role { get; set; }
    [ForeignKey("RoleId")]
    3 references
    public int RoleId { get; set; }


    2 references
    public ICollection<EmployeeTask> EmployeeTasks { get; set; }

    1 reference
    public ICollection<Hour> Hours { get; set; }

}
```

Some Data Annotations where used where appropriate.
At stages, the Foreign Key annotation was required to ensure EF Core correctly understood the relationships between entities.

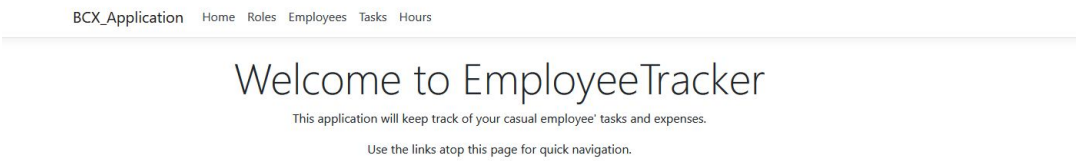
c. EmployeeTasks Table // A bridge entity (N:N) that captures all employees assigned to their respective tasks. This ensures the ability of assigning multiple employees to multiple tasks.

```
//Bridge Entity: N:N
28 references
public class EmployeeTask : CommonEntity
{
    3 references
    public int EmployeeId { get; set; }
    3 references
    public Employee Employee { get; set; }

    3 references
    public int TaskId { get; set; }
    3 references
    public Task Task { get; set; }
}
```

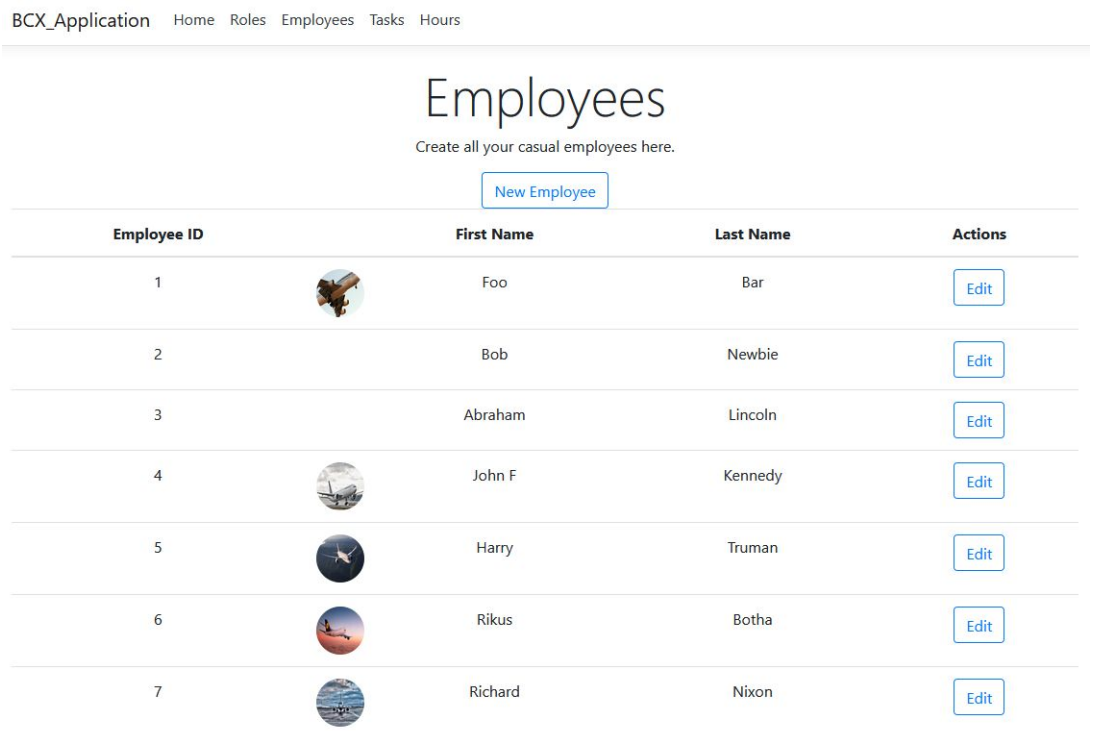
d. Hours Table // Another bridge entity that contains the collective hours worked on any given day, per employee, per task selected. The table also keeps track of hours worked per task, based on the role assigned to the employee at the date of creation.

```
24 references
public class Hour : CommonEntity
{


    [Required]
    2 references
    public Employee Employee { get; set; }
    3 references
    public int EmployeeId { get; set; }

    [Required]
    [DisplayName("Hours Worked")]
    1 reference
    public double HoursWorked { get; set; }


    [DisplayName("Role Rate")]
    2 references
    public double RoleRateAtTime { get; set; }

    3 references
    public double Total { get; set; }

    [DisplayName("Date of Work")]
    [Required]
    2 references
    public DateTime DateWorked { get; set; }

    0 references
    public int TaskId { get; set; }



}
```

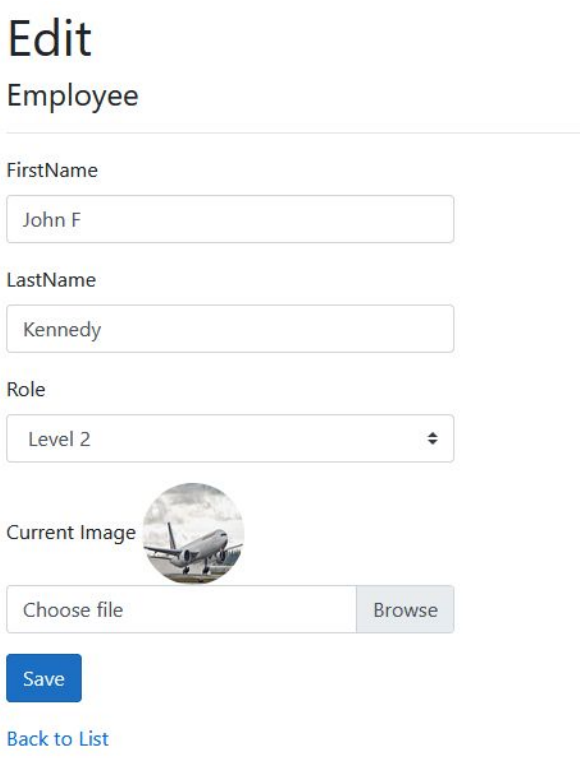
3. The application is built with simplicity in mind. A straight forward - navigation bar is provided at the top of every page (baked into the _Layout file). All views are built using the classic MVC style with ASP.NET backing its many features.



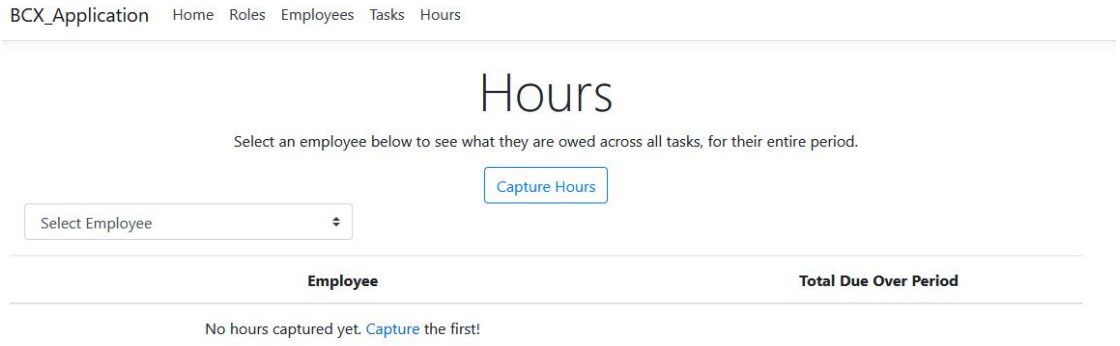
Let’s take a look at the Employees page (simply click on Employees)



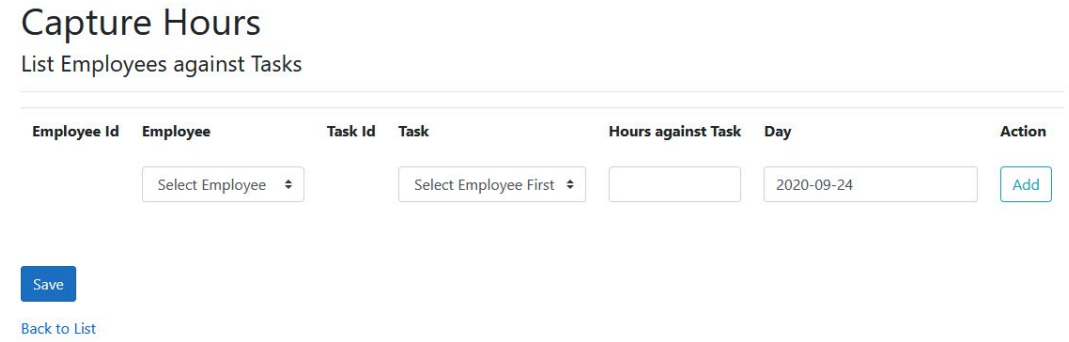
The employees view lists all employees with some supplementary information. Editing an employee allows us to set their role, upload profile picture and change miscellaneous data:



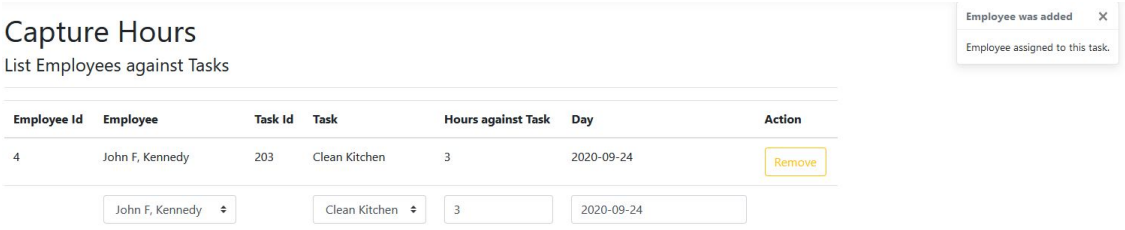
Assuming we have some tasks setup, let's assign this employee to that task. Navigate to Hours:



Click Capture Hours:

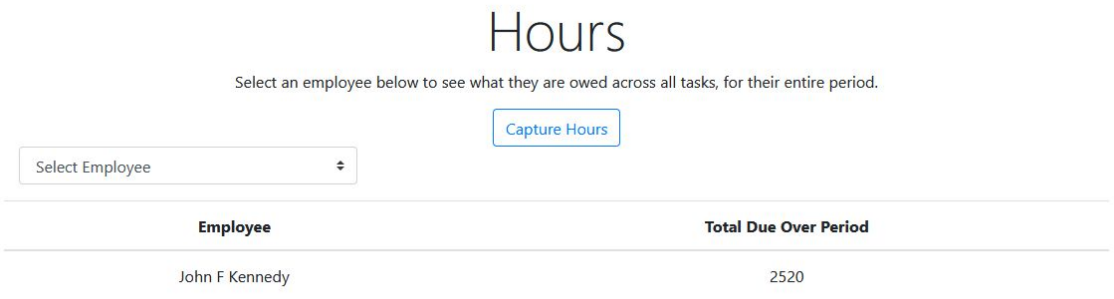


Select the target employee - note how the tasks dropdown gets populated with tasks that the selected employee has been assigned to:



Select the appropriate task, specify hours worked and verify the date. You're ready to click Add and save now.

Your casual employee's hours should now be captured. To view the total sum owed per employee, navigate back to Hours, and select the employee in question:



You may opt to change the employee role, or even the rate of the currently linked role - and hours (and therefore totals) already captured will not be affected by this change, however - it will take effect in future entries.

4. To facilitate data manipulation, Entity Framework Core was used. EF Core 3.1.8 still contains some limitations in areas, but was sufficient for the scale of this project. EF Core was implemented in the BCX_DAL project.

    a. Hence the use of the Repository Pattern, implementations of CRUD operations on repository level are quite simple. Let's consider the Roles Repository:

```
2 references
public async Task<Role> GetRole(int Id)
{
    //Query the Roles Context for a specific record by ID
    return await _context.Roles.FirstOrDefaultAsync(c => c.Id == Id);
}

2 references
public async Task<List<Role>> GetListAsync(/*Consider Paging input model here*/)
{
    //Query the Roles Context for all available Roles
    var test = await _context.Roles.ToListAsync();
    return test;
}

2 references
public async Task<Role> InsertRole(Role data, CancellationToken cancellationToken)
{
    //Set the Created property and add Role to Roles context; Save Changes
    data.CreatedTS = DateTime.Now;
    var result =  await _context.Roles.AddAsync(data, cancellationToken);
    await _context.SaveChangesAsync();
    return null;
}

2 references
public async Task<Role> UpdateRole(int Id, Role data)
{
    //Set the Updated property and update the tracked, modified Role entity; Save Changes
    try
    {
        data.UpdatedTS = DateTime.Now;
        _context.Attach(data).State = EntityState.Modified;
        var result = _context.Roles.Update(data);
        await _context.SaveChangesAsync();
        return null;
    }
    catch (Exception Ex)
    {

        throw;
    }

}
```

b. Next, let's consider another table of similar complexity (Tasks),

```
2 references
public async Task<BCX_CORE.Objects.Tasks.Task> GetTask(int Id)
{
    //Query the context for a specific Task by Id
    return await _context.Tasks.FirstOrDefaultAsync(c => c.Id == Id);
}

2 references
public async Task<List<BCX_CORE.Objects.Tasks.Task>> GetListAsync(/*Consider Paging input model here*/)
{
    //Query the context for all tasks
    var test = await _context.Tasks.ToListAsync();
    return test;
}

2 references
public async Task<BCX_CORE.Objects.Tasks.Task> InsertTask(BCX_CORE.Objects.Tasks.Task data, CancellationToken cancellationToken)
{
    //Update the Created property; Add the new entity and pass the cancellationToken received from the HTTPAPI layer. Save changes
    data.CreatedTS = DateTime.Now;
    var result =  await _context.Tasks.AddAsync(data, cancellationToken);
    await _context.SaveChangesAsync();
    return null;
}

2 references
public async Task<BCX_CORE.Objects.Tasks.Task> UpdateTask(int Id, BCX_CORE.Objects.Tasks.Task data)
{
    //Update the Updated property, set the state of the tracked entity to Modified; Update the entity and save those changes
    try
    {
        data.UpdatedTS = DateTime.Now;
        _context.Attach(data).State = EntityState.Modified;
        var result = _context.Tasks.Update(data);
        await _context.SaveChangesAsync();
        return null;
    }
    catch (Exception Ex)
    {

        throw;
    }

}
```

c. Finally, let's have a look at a table with complex operations (Hours)

```
2 references
public async Task<Hour> GetHour(int employeeId, DateTime fromDate, DateTime toDate)
{
    //Get list of Total hours per employee.
    //Build IQueryable with joined query based on specific ID, with filters for From and To Dates.
    //Select results into a custom Tuple Class for brevity
    var query = await  (from employee in _context.Employees
        join hour in _context.Hours on employee.Id equals hour.EmployeeId
        where hour.DateWorked < toDate && hour.DateWorked >= fromDate && employee.Id == employeeId
        select new TupleClass() { weirdClass = new Tuple<int, string, string, double>
        (employee.Id, employee.FirstName, employee.LastName, hour.Total) }).ToListAsync();

    //Determine the summed total for all rates worked
    var summed = query.Sum(p => p.weirdClass.Item4);

    //Build simple object to return to view
    Hour final = new Hour() { Employee = new Employee()
    {
        Id = query[0].weirdClass.Item1,
        FirstName = query[0].weirdClass.Item2,
        LastName = query[0].weirdClass.Item3
    },
        Total = summed
    };

    return final;
}
```

```
2 references
public async Task<Hour> InsertHour(Hour data)
{
    //Get employee rate
    //Since hour was not eagerly loaded, find related employee's role rate.
    var employee = await _context.Employees.Include(c => c.Role).FirstOrDefaultAsync(c => c.Id == data.EmployeeId);

    //Perform calculations
    data.RoleRateAtTime = employee.Role.RatePerHour;
    data.Total = data.RoleRateAtTime * data.HoursWorked;

    //Add the entity to the context and save changes
    var result =  await _context.Hours.AddAsync(data);
    await _context.SaveChangesAsync();
    return null;
}

2 references
public async Task<Hour> UpdateHour(int Id, Hour data)
{
    //Update the updated property; Set tracked entity to Modified, update context with the entity and save changes
    try
    {
        data.UpdatedTS = DateTime.Now;
        _context.Attach(data).State = EntityState.Modified;
        var result = _context.Hours.Update(data);
        await _context.SaveChangesAsync();
        return null;
    }
    catch (Exception Ex)
    {

        throw;
    }
}
```

d. Custom Tuple Class used for translation

```
1 reference
public class TupleClass
{
    5 references
    public Tuple<int, string, string, double> weirdClass { get; set; }
}
```

e. Finally, let's take a look at the DBContext setup:

```
modelBuilder.Entity<Employee>()
    .HasOne(c => c.Role)
    .WithMany(o => o.Employees)
    .HasForeignKey(c => c.RoleId)
    .OnDelete(DeleteBehavior.NoAction);

//Many to Many relationship
modelBuilder.Entity<EmployeeTask>()
    .HasOne(c => c.Employee)
    .WithMany(c => c.EmployeeTasks)
    .HasForeignKey(c => c.EmployeeId);

modelBuilder.Entity<EmployeeTask>()
   .HasOne(c => c.Task)
   .WithMany(c => c.EmployeeTasks)
   .HasForeignKey(c => c.TaskId);

modelBuilder.Entity<Hour>()
    .HasOne(c => c.Employee)
    .WithMany(c => c.Hours)
    .HasForeignKey(c => c.EmployeeId);


modelBuilder.Seed();
```

FluentAPI configuration was used here since it allows for quick configuration as opposed to using data annotations.

Although, the .HasOne() and .WithMany() configuration may not always make sense for the table configuration, research on the current version of EF Core requires this configuration to allow 1:N entries without throwing an exception when you're adding an already referenced value to the same table again. (i.e. adding another employee to a task)

You'll also notice I've included a small Seed method at the end. It simply seeds some data in the local db once update-database is run.

5. Application Testing
    a. Again, for the sake of brevity, an xUnit Test project was not hooked into the solution because the same reasoning caused a good and proper business logic layer to be omitted too. Therefore, the application's functionality was manually tested with the aid of Swagger and Postman to test the BCX_HTTPAPI layer, and some elbow grease (manually stepping through all processes) to test the front end while observing changes in the local db.

    b. I found that EF Core is an efficient tool that still requires mastering to make queries simpler. Also found that manually testing the solution costs more time

due to unit testing and regression testing after changes.

c. Suggestions include that a business logic layer should have been in scope from the beginning, and proper unit tests should have been factored in.

Thank you for the opportunity to submit this test application.

Please see the source code provided in the private repo in GitHub, should you have any questions.