OVERTIME MANAGEMENT SYSTEM DOCUMENTATION

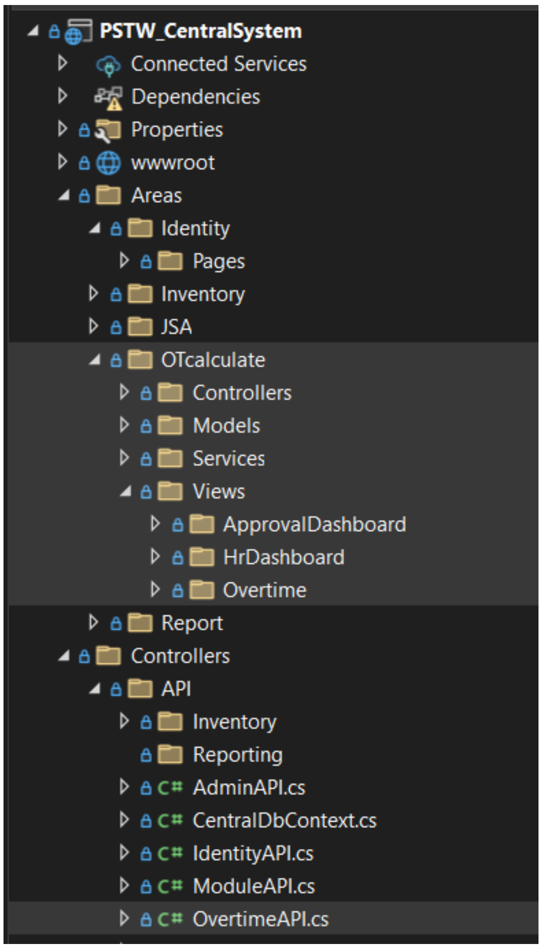
Prepared by:
ANIS NADIA BINTI MOHAMAD ZAKARIA

TABLE OF CONTENTS

FOLDER STRUCTURE

In the **PSTW_CentralSystem** project, the following new folder structure has been implemented under the **Areas** directory to accommodate the new functionality:

- **OTcalculate** Module: This module contains the **Controllers**, **Models**, **Services** and **Views** directories, designed to encapsulate the logic and presentation for the overtime system.
- **API Directory** (Controllers/API): Under the Controllers folder, within the API sub-directory, the **OvertimeAPI.cs** file has been added. This API is crucial for supporting data interaction and reporting functionalities specifically related to the overtime system.



```
PSTW_CentralSystem
    Connected Services
    Dependencies
    Properties
    wwwroot
    Areas
        Identity
            Pages
        Inventory
        JSA
        OTcalculate
            Controllers
            Models
            Services
            Views
                ApprovalDashboard
                HrDashboard
                Overtime
        Report
    Controllers
        API
            Inventory
            Reporting
            AdminAPI.cs
            CentralDbContext.cs
            IdentityAPI.cs
            ModuleAPI.cs
            OvertimeAPI.cs
```

DATABASE TABLES

The **PSTW_CS** database has been extended with the introduction of the following new tables to support the system's enhanced features:

The tables added are **approvalflow**, **flexihours**, **holidays**, **hrusersetting**, **otregisters**, **otstatus**, **rates**, **staffsign**, **states**, **weekends**.

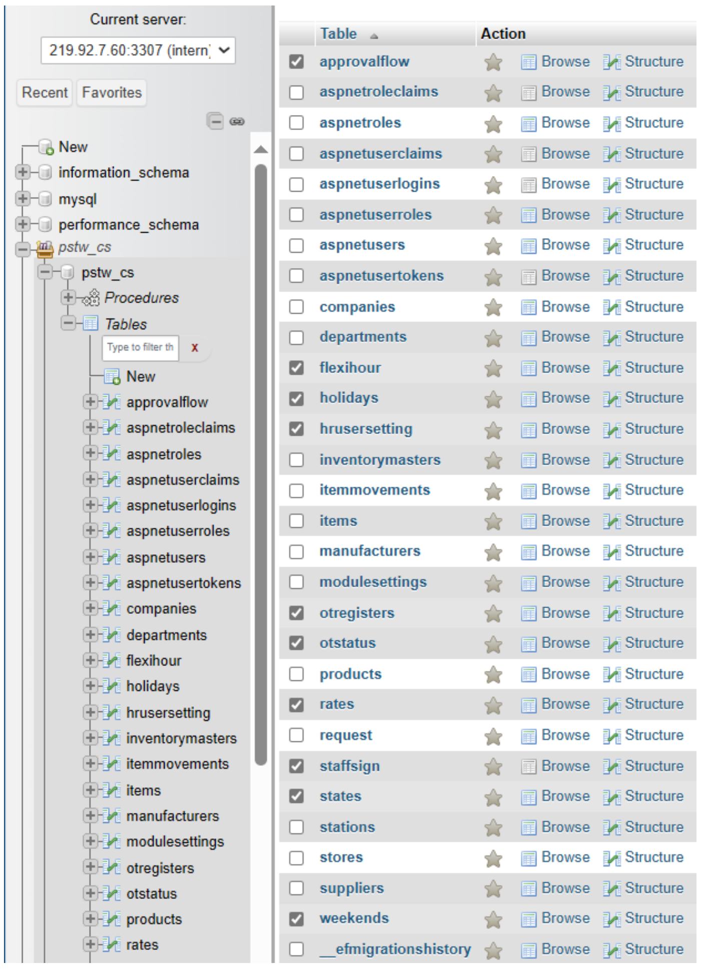


	Table	Action	
✓	approvalflow	Browse	Structure
	aspnetroleclaims	Browse	Structure
	aspnetroles	Browse	Structure
	aspnetuserclaims	Browse	Structure
	aspnetuserlogins	Browse	Structure
	aspnetuserroles	Browse	Structure
	aspnetusers	Browse	Structure
	aspnetusertokens	Browse	Structure
	companies	Browse	Structure
	departments	Browse	Structure
✓	flexihour	Browse	Structure
✓	holidays	Browse	Structure
✓	hrusersetting	Browse	Structure
	inventorymasters	Browse	Structure
	itemmovements	Browse	Structure
	items	Browse	Structure
	manufacturers	Browse	Structure
	modulesettings	Browse	Structure
✓	otregisters	Browse	Structure
✓	otstatus	Browse	Structure
	products	Browse	Structure
✓	rates	Browse	Structure
	request	Browse	Structure
✓	staffsign	Browse	Structure
✓	states	Browse	Structure
	stations	Browse	Structure
	stores	Browse	Structure
	suppliers	Browse	Structure
✓	weekends	Browse	Structure
	__efmigrationshistory	Browse	Structure

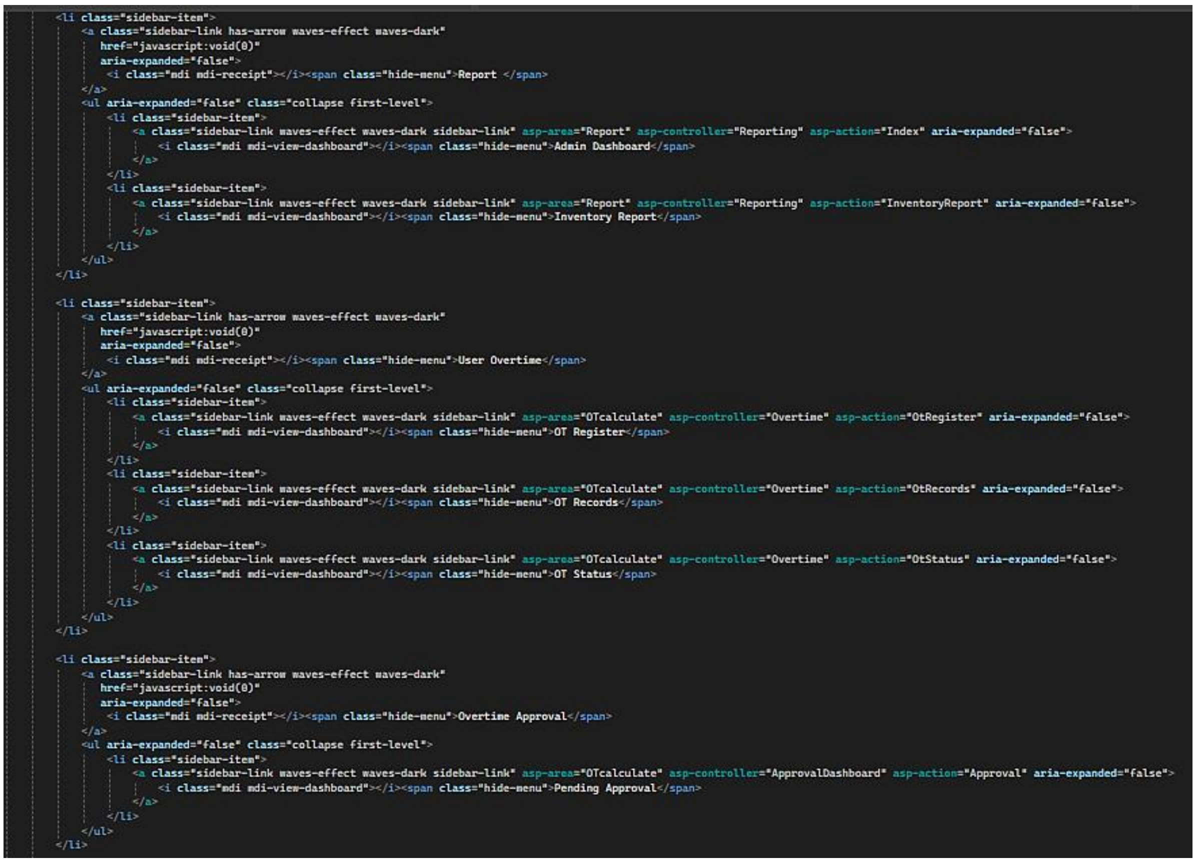
SIDEBAR NAVIGATION (_Layout.cshtml)

The provided code snippet from **_Layout.cshtml** illustrates the integration of the new Overtime Management System's navigation into the application's main sidebar. The following key menu sections have been implemented:

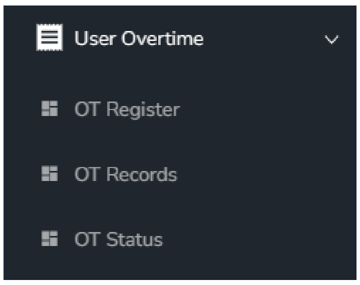
- **User Overtime** Menu Section: This section facilitates user-specific overtime functionalities. It includes a parent menu item, "User Overtime," which expands to reveal sub-links for:
  - "**OT Register**": Allows users to submit new overtime requests.
  - "**OT Records**": Displays a history of a user's submitted overtime.
  - "**OT Status**": Shows the current approval status of overtime requests. These links are directing them to the relevant views within the Otcalculate module developed for this system.

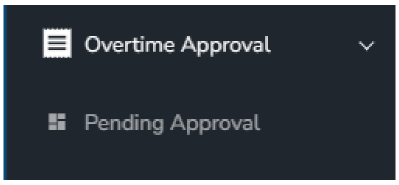
- **Overtime Approval** Menu Section: Designed for approvers, this section enables the management of overtime requests. It features a parent menu item, "Overtime Approval," with a sub-link:
  - "**Pending Approval**": Navigates to a dashboard where pending overtime requests can be reviewed and acted upon.

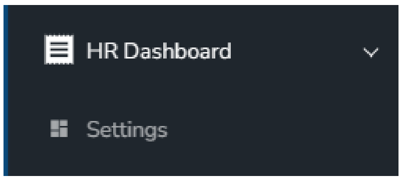
- **HR Dashboard** Menu Section: This dedicated section offers essential administrative capabilities for human resource management. Through its settings menu, HR users can manage employee-related data such as salary values, configure public holiday and weekend policies for various states, and create or modify system approval flows.



```html
<li class="sidebar-item">
    <a class="sidebar-link has-arrow waves-effect waves-dark"
       href="javascript:void(0)"
       aria-expanded="false">
        <i class="mdi mdi-receipt"></i><span class="hide-menu">Report </span>
    </a>
    <ul aria-expanded="false" class="collapse first-level">
        <li class="sidebar-item">
            <a class="sidebar-link waves-effect waves-dark sidebar-link" asp-area="Report" asp-controller="Reporting" asp-action="Index" aria-expanded="false">
                <i class="mdi mdi-view-dashboard"></i><span class="hide-menu">Admin Dashboard</span>
            </a>
        </li>
        <li class="sidebar-item">
            <a class="sidebar-link waves-effect waves-dark sidebar-link" asp-area="Report" asp-controller="Reporting" asp-action="InventoryReport" aria-expanded="false">
                <i class="mdi mdi-view-dashboard"></i><span class="hide-menu">Inventory Report</span>
            </a>
        </li>
    </ul>
</li>

<li class="sidebar-item">
    <a class="sidebar-link has-arrow waves-effect waves-dark"
       href="javascript:void(0)"
       aria-expanded="false">
        <i class="mdi mdi-receipt"></i><span class="hide-menu">User Overtime</span>
    </a>
    <ul aria-expanded="false" class="collapse first-level">
        <li class="sidebar-item">
            <a class="sidebar-link waves-effect waves-dark sidebar-link" asp-area="OTcalculate" asp-controller="Overtime" asp-action="OtRegister" aria-expanded="false">
                <i class="mdi mdi-view-dashboard"></i><span class="hide-menu">OT Register</span>
            </a>
        </li>
        <li class="sidebar-item">
            <a class="sidebar-link waves-effect waves-dark sidebar-link" asp-area="OTcalculate" asp-controller="Overtime" asp-action="OtRecords" aria-expanded="false">
                <i class="mdi mdi-view-dashboard"></i><span class="hide-menu">OT Records</span>
            </a>
        </li>
        <li class="sidebar-item">
            <a class="sidebar-link waves-effect waves-dark sidebar-link" asp-area="OTcalculate" asp-controller="Overtime" asp-action="OtStatus" aria-expanded="false">
                <i class="mdi mdi-view-dashboard"></i><span class="hide-menu">OT Status</span>
            </a>
        </li>
    </ul>
</li>

<li class="sidebar-item">
    <a class="sidebar-link has-arrow waves-effect waves-dark"
       href="javascript:void(0)"
       aria-expanded="false">
        <i class="mdi mdi-receipt"></i><span class="hide-menu">Overtime Approval</span>
    </a>
    <ul aria-expanded="false" class="collapse first-level">
        <li class="sidebar-item">
            <a class="sidebar-link waves-effect waves-dark sidebar-link" asp-area="OTcalculate" asp-controller="ApprovalDashboard" asp-action="Approval" aria-expanded="false">
                <i class="mdi mdi-view-dashboard"></i><span class="hide-menu">Pending Approval</span>
            </a>
        </li>
    </ul>
</li>
```





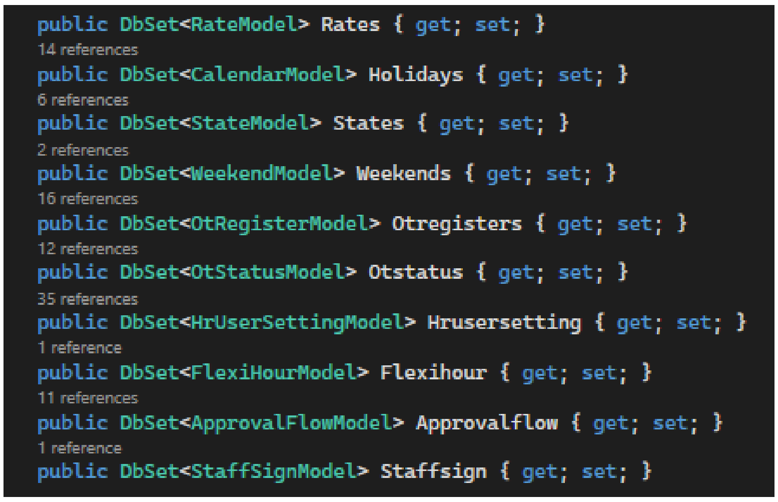


DATABASE CONTEXT CONFIGURATION (CentralSystemContext.cs)

The **CentralSystemContext.cs** file serves as the main gateway for the Entity Framework Core to interact with the application database.

The following DbSet properties have been explicitly added to this context to support new features and modules, particularly those related to the Overtime Management System.

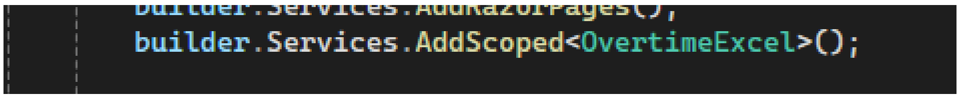
This Db Set is important because it allows the application to easily query and store data for each of these models. It acts as a bridge, connecting our application data models directly to the **PSTW_CS** database tables.



```
public DbSet<RateModel> Rates { get; set; }
14 references
public DbSet<CalendarModel> Holidays { get; set; }
6 references
public DbSet<StateModel> States { get; set; }
2 references
public DbSet<WeekendModel> Weekends { get; set; }
16 references
public DbSet<OtRegisterModel> Otregisters { get; set; }
12 references
public DbSet<OtStatusModel> Otstatus { get; set; }
35 references
public DbSet<HrUserSettingModel> Hrusersetting { get; set; }
1 reference
public DbSet<FlexiHourModel> Flexihour { get; set; }
11 references
public DbSet<ApprovalFlowModel> Approvalflow { get; set; }
1 reference
public DbSet<StaffSignModel> Staffsign { get; set; }
```

REGISTRATION (Program.cs)

To support the Overtime Management System functionality, the following service registration has been added in **Program.cs**:



```
builder.Services.AddRazorPages();
builder.Services.AddScoped<OvertimeExcel>();
```

This means that every time a user interacts with the system (such as clicking a button involving overtime data), a dedicated **OvertimeExcel** object is created just for that interaction. This object then handles all the necessary Excel-related tasks (such as generating reports or exporting data) for that single request. It's like having a dedicated assistant (**OvertimeExcel**) serving one customer at a time, ensuring that all of their Excel needs for that time are met efficiently.

CONTROLLERS

- **ApprovalDashboardController.cs**

  The controller has an "Action Method" that handles specific tasks when a user clicks a link or submits a form.

  - **Approval()** Function:

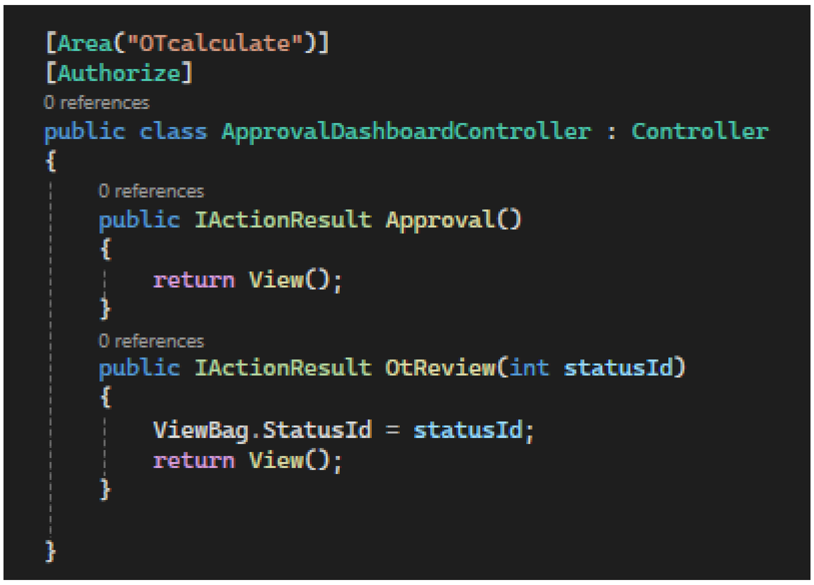
    Purpose: This function is to show the main approval dashboard page.

    How it works: When someone clicks on "Overtime Approval" in the menu, this function runs. It simply tells the system to display the **Approval.cshtml** web page, which contains a list of overtime requests awaiting approval.

  - **OtReview(int statusId)** Function:

    Purpose: This function is to show a detailed review page for a particular overtime request, based on its status.

    How it works: When the approver opens this page, it may need a statusId with it. This number helps the page know what type of request to show. For example, if the statusId is 1, it may show all overtime data that has a statusId of 1. The function then displays the OtReview.cshtml web page, showing the detailed information.



```
[Area("OTcalculate")]
[Authorize]
0 references
public class ApprovalDashboardController : Controller
{
    0 references
    public IActionResult Approval()
    {
        return View();
    }
    0 references
    public IActionResult OtReview(int statusId)
    {
        ViewBag.StatusId = statusId;
        return View();
    }

}
```

- **HrDashboardController.cs**

  Controllers have "Action Methods" handle specific tasks when a user navigates to a particular page or performs an action.

  - **Rate()** Function (Salary):

    Purpose: This function displays the page where HR can view and manage salary of staff.

    How it works: It simply tells the system to show the **Rate.cshtml** web page.

  - **Settings()** Function:

    Purpose: This function displays a general **settings page for the HR Dashboard**, where various system configurations can be managed. This might be a central hub for multiple administrative settings.

    How it works: It simply tells the system to show the **Settings.cshtml** web page.

  - **Calendar()** Function:

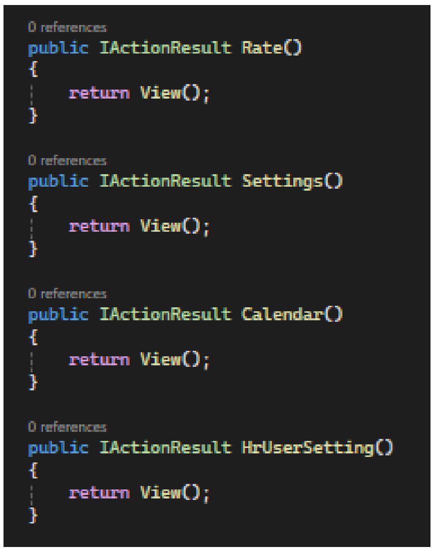
    Purpose: This function is for showing the page where HR can **manage public holidays and other calendar-related settings** that affect overtime calculations

    How it works: It simply tells the system to show the **Calendar.cshtml** web page.

  - **HrUserSetting()** Function:

    Purpose: This function displays a dedicated page for **configuring specific employee data settings** in the system, such as managing employee regions and flexible hours, and approval flows.

    How it works: It simply tells the system to show the **HrUserSetting.cshtml** web page.



```
0 references
public IActionResult Rate()
{
    return View();
}

0 references
public IActionResult Settings()
{
    return View();
}

0 references
public IActionResult Calendar()
{
    return View();
}

0 references
public IActionResult HrUserSetting()
{
    return View();
}
```

- **OvertimeController.cs**

  The controller has "Action Methods" that handle specific tasks when the user navigates to a specific page.

  - **OtRegister()** Function:

    Purpose: This function displays a page where the user can register a new overtime.

    How it works: When an employee clicks on "**OT Register**" in the menu, this function runs. It simply tells the system to show the **OtRegister.cshtml** web page, which contains a form for overtime registration.

  - **OtRecords()** Function:

    Purpose: This function displays a page where the user can see a list of all their past registered overtime.

    How it works: When an employee clicks on "**OT Records**", this function runs. It simply tells the system to show the **OtRecords.cshtml** web page, presenting their registered overtime history.

  - **EditOvertime()** Function:

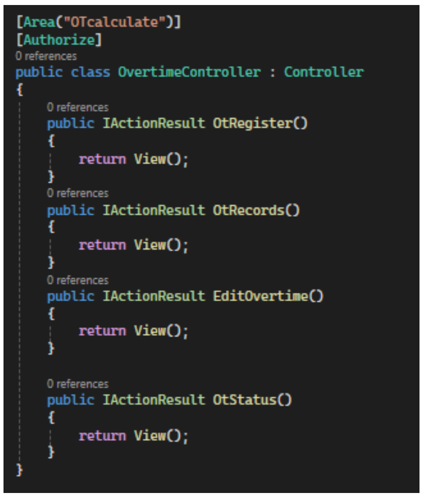
    Purpose: This function displays a page that allows the user to edit an existing registered overtime.

    How it works: When an employee chooses to modify a previously registered overtime in **OtRecords.cshtml**, this function runs. It tells the system to show the **EditOvertime.cshtml** web page, where they can make changes.

  - **OtStatus()** Function:

    Purpose: This function displays a page where users can check the approval status of their submitted overtime requests.

    How it works: When an employee clicks on "**OT Status**", this function runs. It simply tells the system to show the **OtStatus.cshtml** web page, providing an update on their request.



```
[Area("OTcalculate")]
[Authorize]
0 references
public class OvertimeController : Controller
{
    0 references
    public IActionResult OtRegister()
    {
        return View();
    }
    0 references
    public IActionResult OtRecords()
    {
        return View();
    }
    0 references
    public IActionResult EditOvertime()
    {
        return View();
    }

    0 references
    public IActionResult OtStatus()
    {
        return View();
    }
}
```

MODELS

- **ApprovalFlowModel.cs**

  This model is designed to store the details of an approval flow. Each "Approval Flow" record in the database will define a specific sequence of people who need to approve overtime request.

  - **ApprovalFlowId** (Unique ID):

    [Key] tells the database that this is the unique identifier for each approval flow. This is a number that helps us find and refer to a specific approval flow easily.

  - **ApprovalName** (Flow Name):

    This is a text field (like "Southern Air"). It gives a human-readable name to each approval flow.

  - **HoU, HoD, Manager, HR** (The Approvers):

    These are numbers (int?) that hold the **unique ID** of the specific person (from our **UserModel**) who will act as the:

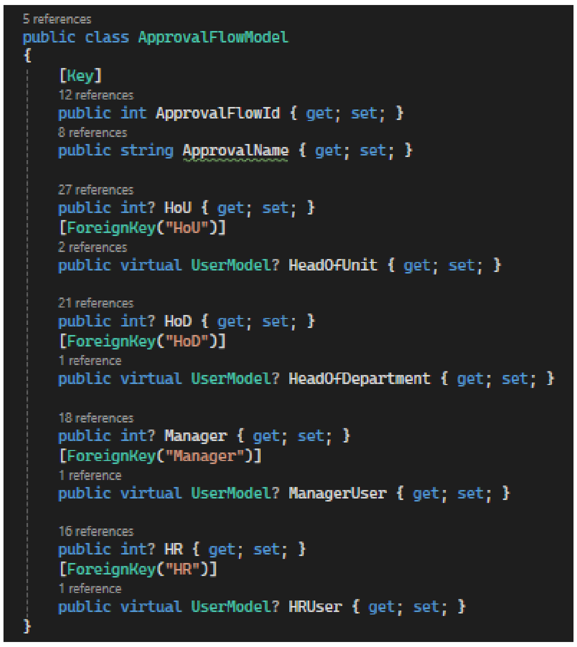
    HoU: Head of Unit

    HoD: Head of Department

    Manager: Manager

    HR: Human Resources

  - The **?** means that a person might not always be required for every step in every flow (e.g., some flows might skip the Manager).



```
5 references
public class ApprovalFlowModel
{
    [Key]
    12 references
    public int ApprovalFlowId { get; set; }
    8 references
    public string ApprovalName { get; set; }

    27 references
    public int? HoU { get; set; }
    [ForeignKey("HoU")]
    2 references
    public virtual UserModel? HeadOfUnit { get; set; }

    21 references
    public int? HoD { get; set; }
    [ForeignKey("HoD")]
    1 reference
    public virtual UserModel? HeadOfDepartment { get; set; }

    18 references
    public int? Manager { get; set; }
    [ForeignKey("Manager")]
    1 reference
    public virtual UserModel? ManagerUser { get; set; }

    16 references
    public int? HR { get; set; }
    [ForeignKey("HR")]
    1 reference
    public virtual UserModel? HRUser { get; set; }
}
```

- **CalendarModel.cs**

  This model is designed to hold details about each public holiday. It allows the system to know the name of a holiday, its date, and which state it applies to.

  - **HolidayId** (Unique ID):

    [Key] tells the database that this is the unique identifier for each holiday entry. This is a number that helps us find and refer to a specific holiday record easily.

  - **HolidayName** (Name of Holiday):

    required string: This is a text field that stores the name of the public holiday (e.g., "Hari Raya Aidilfitri," "Christmas"). required means this information must always be provided.

  - **HolidayDate** (Date of Holiday):

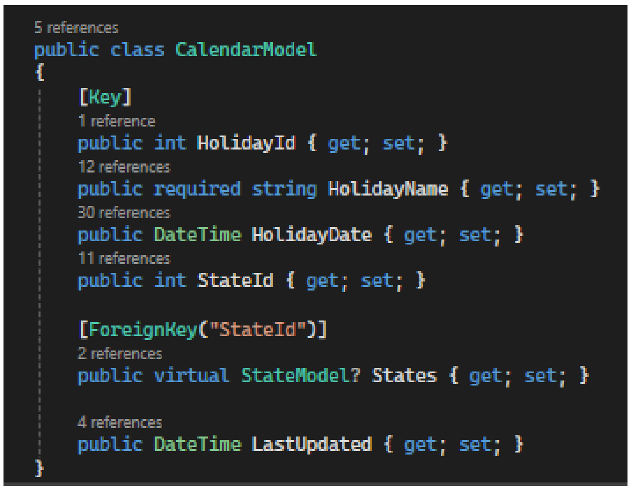
    This stores the actual date of the public holiday.

  - **StateId** (Which State):

    This is a number that indicates which Malaysian state this particular holiday applies to.

  - **LastUpdated** (Last Changed):

    This automatically records the date and time when this holiday entry was last updated or changed in the system.



```
5 references
public class CalendarModel
{
    [Key]
    1 reference
    public int HolidayId { get; set; }
    12 references
    public required string HolidayName { get; set; }
    30 references
    public DateTime HolidayDate { get; set; }
    11 references
    public int StateId { get; set; }

    [ForeignKey("StateId")]
    2 references
    public virtual StateModel? States { get; set; }

    4 references
    public DateTime LastUpdated { get; set; }
}
```

- **FlexiHourModel.cs**

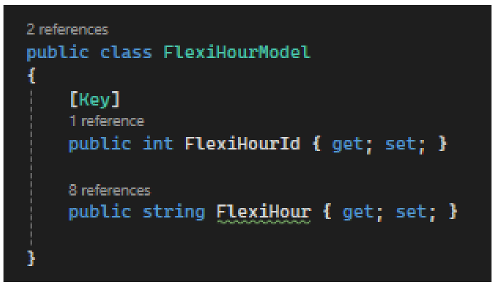
  This model is designed to hold various options for flexible working hours that can be selected by HR. This helps standardize the choices for overtime submission.

  - **FlexiHourId** (Unique ID):

    [Key] tells the database that this is the unique identifier for each flexible hour option. This is a number that helps us find and refer to a specific flexi-hour record easily.

  - **FlexiHour** (The Flexible Hour Value):

    This is a text field that stores the actual flexible hour value or a descriptive label (e.g., "7 AM - 4 PM", "9 AM - 6 PM", "Flexible"). This is the option that HR will see and choose from.



```
2 references
public class FlexiHourModel
{
    [Key]
    1 reference
    public int FlexiHourId { get; set; }

    8 references
    public string FlexiHour { get; set; }
}
```

- **HrUserSettingModel.cs**

  This model is designed to hold customized settings for each employee, managed by HR. This allows the system to handle variations in how overtime is processed for different individuals.

  - **HrUserSettingId** (Unique ID):

    [Key] tells the database that this is the unique identification number for each unique HR setting record.

  - **UserId** (Who is this setting for?):

    This number links directly to a specific employee's record in our UserModel blueprint.

  - **FlexiHourId** (Default Flexible Hours):

    This number links to a specific flexible hour option from our FlexiHourModel blueprint. It sets the default flexible working hours for this employee.

  - **FlexiHourUpdate** (When Flexi-Hours Last Changed):

    This records the date and time when the employee's flexible hour setting was last updated by HR.

- **StateId** (Which State's Rules Apply):

  This number links to a specific state from our StateModel blueprint. It defines which state's public holiday and weekend rules apply to this employee.

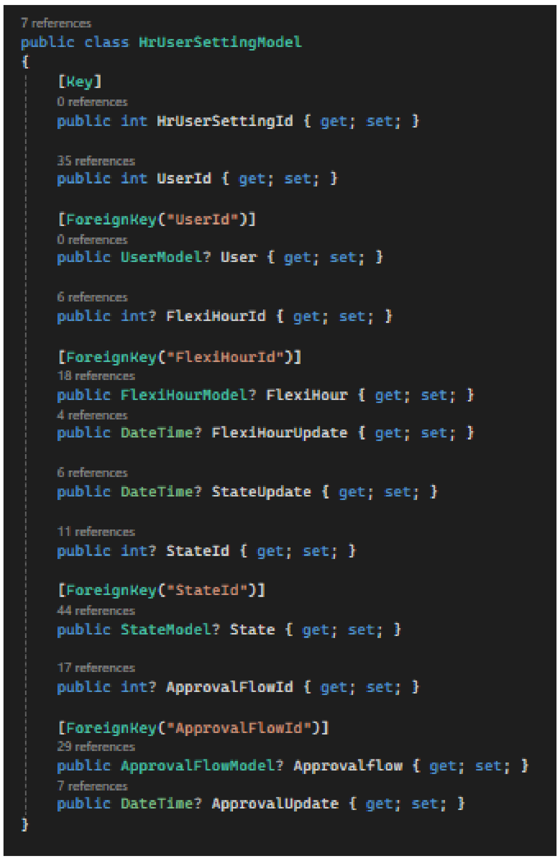
- **StateUpdate** (When State Setting Last Changed):

  This records the date and time when the employee's state setting was last updated by HR.

- **ApprovalFlowId** (Which Approval Path to Follow):

  This number links to a specific approval process from our ApprovalFlowModel blueprint. It determines the specific sequence of approvals an employee's overtime request will follow.

- **ApprovalUpdate** (When Approval Flow Last Changed): This records the date and time when the employee's assigned approval flow was last updated by HR.



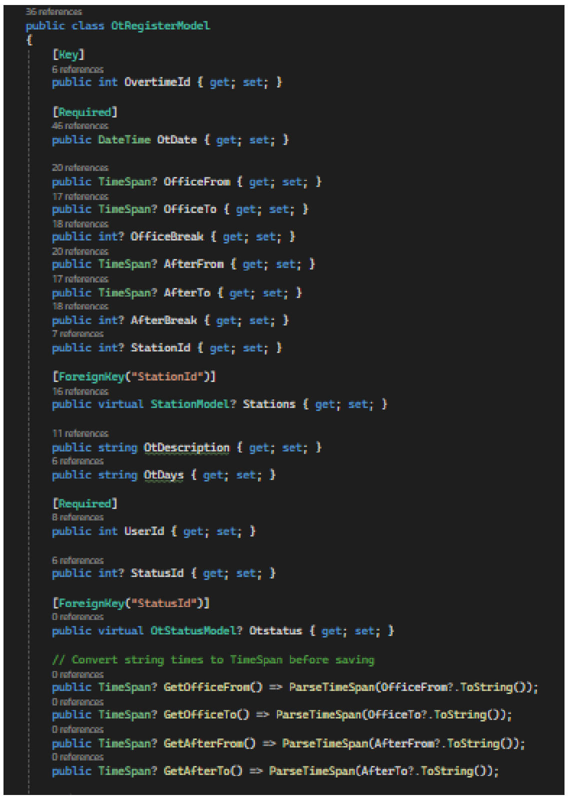
```
7 references
public class HrUserSettingModel
{
    [Key]
    0 references
    public int HrUserSettingId { get; set; }

    35 references
    public int UserId { get; set; }

    [ForeignKey("UserId")]
    0 references
    public UserModel? User { get; set; }

    6 references
    public int? FlexiHourId { get; set; }

    [ForeignKey("FlexiHourId")]
    18 references
    public FlexiHourModel? FlexiHour { get; set; }
    4 references
    public DateTime? FlexiHourUpdate { get; set; }

    6 references
    public DateTime? StateUpdate { get; set; }

    11 references
    public int? StateId { get; set; }

    [ForeignKey("StateId")]
    44 references
    public StateModel? State { get; set; }

    17 references
    public int? ApprovalFlowId { get; set; }

    [ForeignKey("ApprovalFlowId")]
    29 references
    public ApprovalFlowModel? Approvalflow { get; set; }
    7 references
    public DateTime? ApprovalUpdate { get; set; }
}
```

- **OtRegisterModel.cs**

  This modal is like a central hub for everything related to an overtime request. It contains the main blueprint for an overtime record and also blueprints for handling edits, updates, and keeping a log of changes.

  - **public class OtRegisterModel**
    This is the most important part of the file. It defines the main structure for a single overtime request that employees submit. Think of it as the form that gets filled out and then saved in our database.

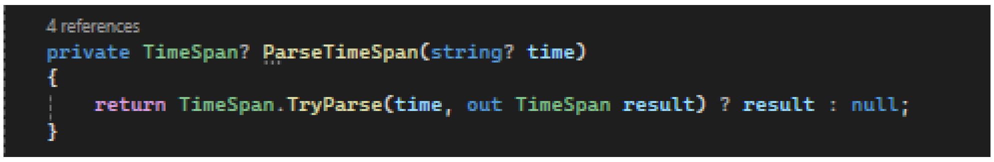
    - **OvertimeId** (Unique ID): This is a unique number that helps us identify each individual overtime request.
    - **OtDate** (Overtime Date): The specific date when the overtime was worked.
    - **OfficeFrom, OfficeTo, OfficeBreak** (Office Overtime Times): These store the start time, end time, and break duration for overtime worked during regular office hours (if applicable). TimeSpan? means it stores a duration, and ? means it can be left empty.
    - **AfterFrom, AfterTo, AfterBreak** (After-Office Overtime Times): Similar to above, but for overtime worked after regular office hours.
    - **StationId** (Location): This links the overtime request to a specific work location.
    - **OtDescription** (What was done): A brief explanation of the work performed during overtime.
    - **OtDays** (Days worked overtime): This might specify if it was a weekday, weekend, or public holiday.
    - **UserId** (Who registered it): This links the overtime request to the employee who registered it.
    - **StatusId** (Current Status): This shows the current approval status of the overtime request (e.g., pending, approved, rejected).
    - **GetOfficeFrom(), GetOfficeTo(), GetAfterFrom(), GetAfterTo()**: These are helper parts that make sure the time information (like "9:00 AM") is correctly read and stored in a format the system understands (TimeSpan).



```
36 references
public class OtRegisterModel
{
    [Key]
    6 references
    public int OvertimeId { get; set; }

    [Required]
    46 references
    public DateTime OtDate { get; set; }

    20 references
    public TimeSpan? OfficeFrom { get; set; }
    17 references
    public TimeSpan? OfficeTo { get; set; }
    18 references
    public int? OfficeBreak { get; set; }
    20 references
    public TimeSpan? AfterFrom { get; set; }
    17 references
    public TimeSpan? AfterTo { get; set; }
    18 references
    public int? AfterBreak { get; set; }
    7 references
    public int? StationId { get; set; }

    [ForeignKey("StationId")]
    16 references
    public virtual StationModel? Stations { get; set; }

    11 references
    public string OtDescription { get; set; }
    6 references
    public string OtDays { get; set; }

    [Required]
    8 references
    public int UserId { get; set; }

    6 references
    public int? StatusId { get; set; }

    [ForeignKey("StatusId")]
    0 references
    public virtual OtStatusModel? Otstatus { get; set; }

    // Convert string times to TimeSpan before saving
    0 references
    public TimeSpan? GetOfficeFrom() => ParseTimeSpan(OfficeFrom?.ToString());
    0 references
    public TimeSpan? GetOfficeTo() => ParseTimeSpan(OfficeTo?.ToString());
    0 references
    public TimeSpan? GetAfterFrom() => ParseTimeSpan(AfterFrom?.ToString());
    0 references
    public TimeSpan? GetAfterTo() => ParseTimeSpan(AfterTo?.ToString());
```

- **private TimeSpan? ParseTimeSpan(string? time)**
  This This is a private helper function tucked away inside OtRegisterModel.

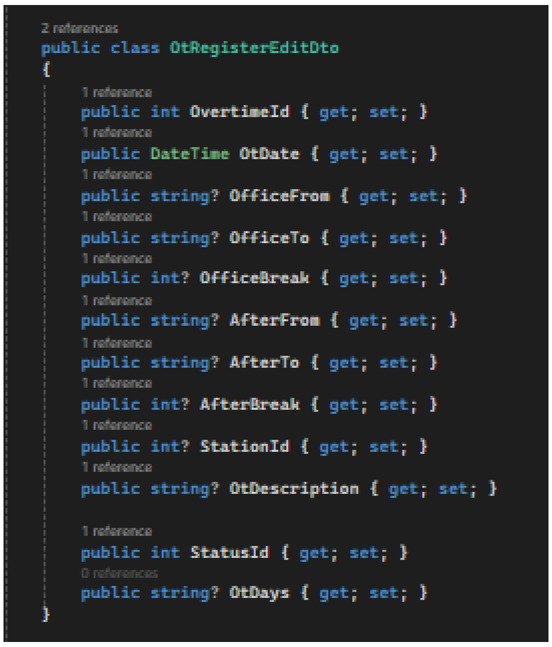
  - **Purpose:** Its job is to take a time that might be written as text (like "09:00") and try to convert it into a TimeSpan format, which is how our system likes to handle durations.
  - **private:** This means only code inside the OtRegisterModel can use this helper; it's not directly accessible from outside.



```
4 references
private TimeSpan? ParseTimeSpan(string? time)
{
    return TimeSpan.TryParse(time, out TimeSpan result) ? result : null;
}
```

- **public class OtRegisterEditDto**
  This is a "Data Transfer Object" (DTO), a simpler blueprint specifically used when a user is editing an existing overtime request on a web form. It helps keep the main OtRegisterModel clean and focuses on what's displayed or sent from a form rather than how it's stored in the database.

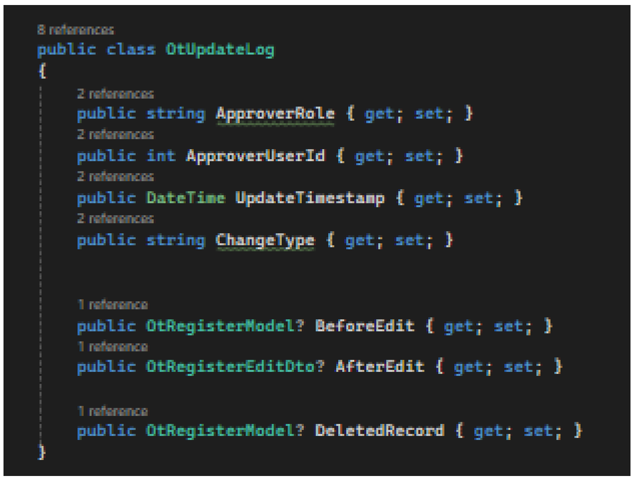
  - **Purpose:** It's designed to neatly package the information entered by the user on an edit form. Notice that times (like OfficeFrom) are string? here. This is common because web forms usually send time as text, which then needs to be converted later.



```
2 references
public class OtRegisterEditDto
{
    1 reference
    public int OvertimeId { get; set; }
    1 reference
    public DateTime OtDate { get; set; }
    1 reference
    public string? OfficeFrom { get; set; }
    1 reference
    public string? OfficeTo { get; set; }
    1 reference
    public int? OfficeBreak { get; set; }
    1 reference
    public string? AfterFrom { get; set; }
    1 reference
    public string? AfterTo { get; set; }
    1 reference
    public int? AfterBreak { get; set; }
    1 reference
    public int? StationId { get; set; }
    1 reference
    public string? OtDescription { get; set; }

    1 reference
    public int StatusId { get; set; }
    0 references
    public string? OtDays { get; set; }
}
```

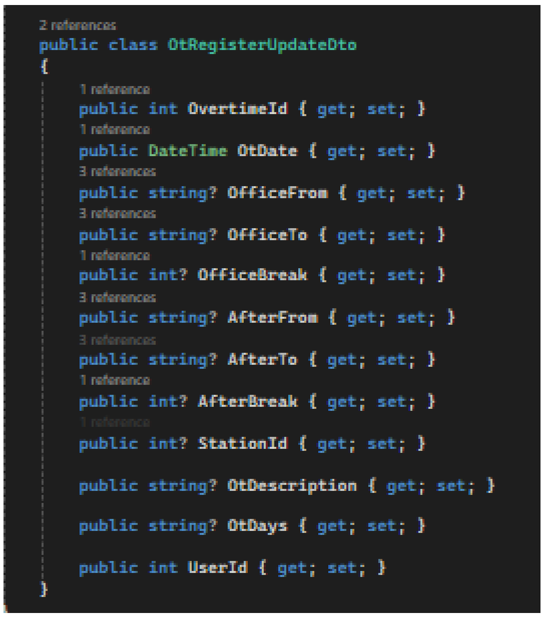
- **public class OtUpdateLog**
  This is a blueprint for creating a record of every change made to an overtime request. It's like a history book for each request.

  - **Purpose:** It helps us track who made a change, when they made it, and what kind of change it was. This is important for auditing and understanding the lifecycle of an overtime request.
  - **ApproverRole, ApproverUserId, UpdateTimestamp, ChangeType**: These fields capture details about the person who made the change, when it happened, and whether it was an edit or a deletion.
  - **BeforeEdit, AfterEdit, DeletedRecord:** These are special fields that can hold a snapshot of the overtime record before the change, after the change, or the record itself if it was deleted. This provides a full audit trail.



```
8 references
public class OtUpdateLog
{
    2 references
    public string ApproverRole { get; set; }
    2 references
    public int ApproverUserId { get; set; }
    2 references
    public DateTime UpdateTimestamp { get; set; }
    2 references
    public string ChangeType { get; set; }


    1 reference
    public OtRegisterModel? BeforeEdit { get; set; }
    1 reference
    public OtRegisterEditDto? AfterEdit { get; set; }

    1 reference
    public OtRegisterModel? DeletedRecord { get; set; }
}
```

- **public class OtRegisterUpdateDto**
  This is another "Data Transfer Object" (DTO), very similar to OtRegisterEditDto.



```
2 references
public class OtRegisterUpdateDto
{
    1 reference
    public int OvertimeId { get; set; }
    1 reference
    public DateTime OtDate { get; set; }
    3 references
    public string? OfficeFrom { get; set; }
    3 references
    public string? OfficeTo { get; set; }
    1 reference
    public int? OfficeBreak { get; set; }
    3 references
    public string? AfterFrom { get; set; }
    3 references
    public string? AfterTo { get; set; }
    1 reference
    public int? AfterBreak { get; set; }
    1 reference
    public int? StationId { get; set; }

    public string? OtDescription { get; set; }

    public string? OtDays { get; set; }

    public int UserId { get; set; }
}
```

- **OtStatusModel.cs**

  This is the main part that defines all the details for an OT status record.

  - **OtStatusModel**

    - **[Key] public int StatusId { get; set; }:** This is like the unique ID number for each OT status record. Every record will have a different StatusId, making it easy to find and refer to a specific one.

    - **public int UserId { get; set; }:** This stores the ID of the user who submitted the OT request.

    - **public int Month { get; set; }** and **public int Year { get; set; }:** These tell which month and year the OT request is for.

    - **public DateTime SubmitDate { get; set; }:** This records the exact date and time when the user first submitted their OT request.

    - **public string? HouStatus { get; set; }, public string? HodStatus { get; set; }, public string? ManagerStatus { get; set; }, public string? HrStatus { get; set; }:** These are fields to track the approval status from different levels or roles:

      HouStatus: Status from the Head of Unit.

      HodStatus: Status from the Head of Department.

      ManagerStatus: Status from a Manager.

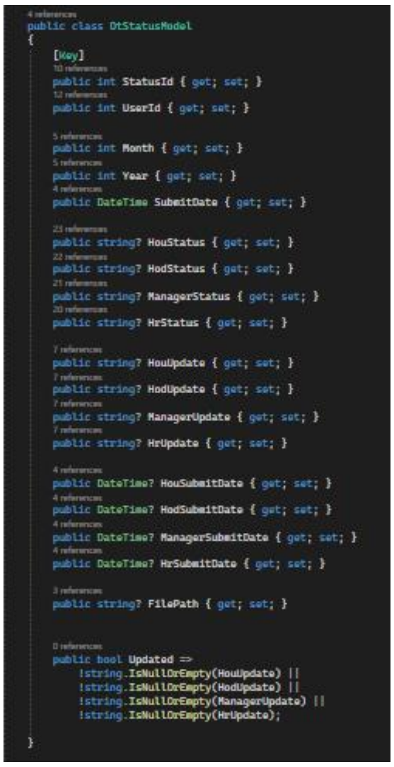
      HrStatus: Status from Human Resources.

    - **public string? HouUpdate { get; set; }**, **public string? HodUpdate { get; set; }**, etc.: These fields are specifically designed to store **JSON (JavaScript Object Notation) formatted text**. This JSON text will contain a detailed record of **what changes the respective approver made** to the OT request.

    - **public DateTime? HouSubmitDate { get; set; }**, **public DateTime? HodSubmitDate { get; set; }**, etc.: These record the **date and time when each specific person** (Head of Unit, Head of Department, etc.) made their decision or updated the status.

    - **public string? FilePath { get; set; }**: This stores the **location or path to any attached files** related to the OT request (like time sheet).

  - **public bool Updated**: This is a smart little helper. It automatically checks if **any** of the "update" fields (HouUpdate, HodUpdate, ManagerUpdate, HrUpdate) have any text in them. If even one of them has text, it means someone has made a update, and this Updated property will show true. Otherwise, it's false.



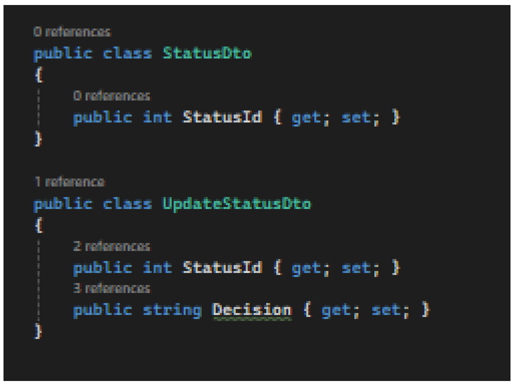
```
4 references
public class OtStatusModel
{
    [Key]
    10 references
    public int StatusId { get; set; }
    12 references
    public int UserId { get; set; }

    5 references
    public int Month { get; set; }
    5 references
    public int Year { get; set; }
    4 references
    public DateTime SubmitDate { get; set; }

    23 references
    public string? HouStatus { get; set; }
    22 references
    public string? HodStatus { get; set; }
    21 references
    public string? ManagerStatus { get; set; }
    20 references
    public string? HrStatus { get; set; }

    7 references
    public string? HouUpdate { get; set; }
    7 references
    public string? HodUpdate { get; set; }
    7 references
    public string? ManagerUpdate { get; set; }
    7 references
    public string? HrUpdate { get; set; }

    4 references
    public DateTime? HouSubmitDate { get; set; }
    4 references
    public DateTime? HodSubmitDate { get; set; }
    4 references
    public DateTime? ManagerSubmitDate { get; set; }
    4 references
    public DateTime? HrSubmitDate { get; set; }

    3 references
    public string? FilePath { get; set; }


    0 references
    public bool Updated =>
        !string.IsNullOrEmpty(HouUpdate) ||
        !string.IsNullOrEmpty(HodUpdate) ||
        !string.IsNullOrEmpty(ManagerUpdate) ||
        !string.IsNullOrEmpty(HrUpdate);

}
```

- **StatusDto and UpdateStatusDto Explained**

  These are smaller, simpler versions of your OtStatusModel, used for very specific tasks, usually when sending data between different parts of application (like from a webpage to the server). They're called **Data Transfer Objects (DTOs)**.

  - **public class StatusDto { public int StatusId { get; set; } }**: This is used when user only need to refer to an OT record by its **ID**, nothing else.

  - **public class UpdateStatusDto { public int StatusId { get; set; } public string Decision { get; set; } }**: This is used when user want to **update the status** of a specific OT record. User provide the StatusId to identify which record, and then user provide a Decision (like "Approve" or "Reject") to tell the system what action to take.



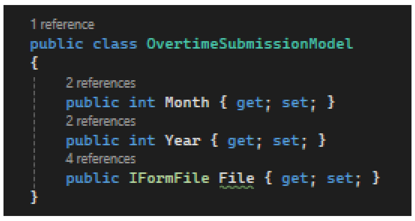
```
0 references
public class StatusDto
{
    0 references
    public int StatusId { get; set; }
}

1 reference
public class UpdateStatusDto
{
    2 references
    public int StatusId { get; set; }
    3 references
    public string Decision { get; set; }
}
```

- **OvertimeSubmissionModel.cs**

  This OvertimeSubmissionModel is a straightforward blueprint for handling **overtime submissions**. Think of it as a simple form that captures the essential details needed when someone wants to submit their overtime information.

  Here's a breakdown of what each part means:

  - **public int Month { get; set; }**: This field is where the system will store the **month** for which the overtime is being submitted.
  - **public int Year { get; set; }**: Similarly, this field holds the **year** corresponding to the overtime submission.
  - **public IFormFile File { get; set; }**: This is a crucial part. It's designed to handle an **uploaded file**. In the context of overtime, this File would typically containing the detailed overtime hours, calculations, or any other supporting documentation the user needs to submit. IFormFile is a standard way in web applications to represent a file that's been uploaded by a user.



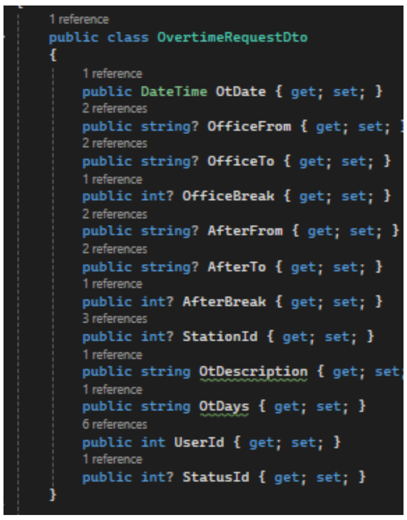
```
1 reference
public class OvertimeSubmissionModel
{
    2 references
    public int Month { get; set; }
    2 references
    public int Year { get; set; }
    4 references
    public IFormFile File { get; set; }
}
```

- **OvertimeRequestDto.cs**

  This OvertimeRequestDto (Data Transfer Object) is like a standardized form or package used to send **all the details for a single overtime request**. When someone fills out information about their overtime, this is the structure that collects and holds all that data before it's processed or saved.

  - **public DateTime OtDate { get; set; }**: This is the **specific date** for which the overtime is being requested.
  - **public string? OfficeFrom { get; set; }** and **public string? OfficeTo { get; set; }**: These fields likely record the **start and end times for overtime worked during normal office hours**.
  - **public int? OfficeBreak { get; set; }**: This would be the **duration of any break taken during office-hours overtime**, usually in minutes. It's optional, so it can be blank.
  - **public string? AfterFrom { get; set; }** and **public string? AfterTo { get; set; }**: Similar to the "Office" times, these likely record the **start and end times for overtime worked after regular office hours**.
  - **public int? AfterBreak { get; set; }**: This is the **duration of any break taken during after-hours overtime**, in minutes. Also optional.

- **public int? StationId { get; set; }**: If the overtime is tied to a specific location or work station, this field stores its **unique ID**.
- **public string OtDescription { get; set; }**: This is a **brief explanation or reason** for why the overtime was needed.
- **public string OtDays { get; set; }**: This likely specifies **which type of day** the overtime was worked on (e.g., "Weekday," "Weekend," "Public Holiday").
- **public int UserId { get; set; }**: This is the **ID of the user** who is submitting this overtime request.
- **public int? StatusId { get; set; }**: This field would hold the **ID of the current approval status** for this overtime request. The ? means it might be blank when the request is first submitted and later filled in as it goes through the approval process.



```
1 reference
public class OvertimeRequestDto
{
    1 reference
    public DateTime OtDate { get; set; }
    2 references
    public string? OfficeFrom { get; set;
    2 references
    public string? OfficeTo { get; set; }
    1 reference
    public int? OfficeBreak { get; set; }
    2 references
    public string? AfterFrom { get; set; }
    2 references
    public string? AfterTo { get; set; }
    1 reference
    public int? AfterBreak { get; set; }
    3 references
    public int? StationId { get; set; }
    1 reference
    public string OtDescription { get; set
    1 reference
    public string OtDays { get; set; }
    6 references
    public int UserId { get; set; }
    1 reference
    public int? StatusId { get; set; }
}
```

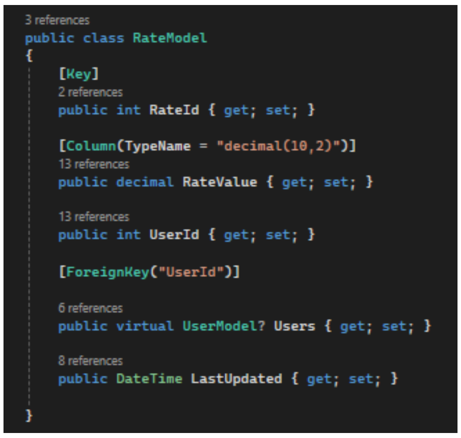
- **RateModel.cs (Salary)**

  This RateModel (which is **Salary**) is a blueprint for storing and managing **individual salary** within the system. Think of it as a record in a database that keeps track of a specific user's salary information and when it was last updated.

  - **[Key] public int RateId { get; set; }**: This is the **unique ID number** for each specific salary record. Every salary entry will have its own RateId to distinguish it from others.
  - **[Column(TypeName = "decimal(10,2)")] public decimal RateValue { get; set; }**: This field stores the **actual salary amount**.
    - decimal: This data type is perfect for currency because it handles exact decimal values, avoiding common rounding issues might find with other number types.
    - [Column(TypeName = "decimal(10,2)")]: This is a hint to database. It tells the database to create a column that can store numbers up to 10 digits in

total, with 2 of those digits being after the decimal point. This ensures the salary values are stored accurately.

  - **public int UserId { get; set; }**: This stores the **ID of the user** to whom this specific salary rate belongs. This is how it links a salary value to a particular person.
  - **public DateTime LastUpdated { get; set; }**: This records the **exact date and time when this salary record was last changed or updated**. This is very useful for auditing and tracking changes over time.

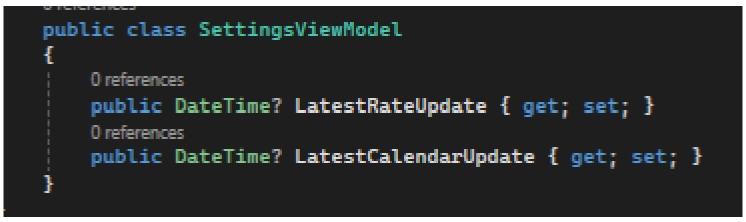


```
3 references
public class RateModel
{
    [Key]
    2 references
    public int RateId { get; set; }

    [Column(TypeName = "decimal(10,2)")]
    13 references
    public decimal RateValue { get; set; }

    13 references
    public int UserId { get; set; }

    [ForeignKey("UserId")]

    6 references
    public virtual UserModel? Users { get; set; }

    8 references
    public DateTime LastUpdated { get; set; }

}
```

- **SettingsModel.cs**

  This SettingsViewModel is a very simple blueprint designed to display **important dates related to the system's settings or configurations**. Think of it as a quick status board that shows when certain critical pieces of information were last updated.

  - **public DateTime? LatestRateUpdate { get; set; }**: This field tells the **most recent date and time when salary information was updated** in the system.
  - **public DateTime? LatestCalendarUpdate { get; set; }**: This field shows the **last date and time when the system's calendar data was updated**. This could include things like public holidays, weekends, or other.



```
public class SettingsViewModel
{
    0 references
    public DateTime? LatestRateUpdate { get; set; }
    0 references
    public DateTime? LatestCalendarUpdate { get; set; }
}
```

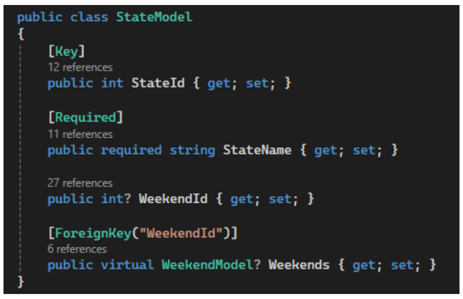
- **StateModel.cs**

  This StateModel serves as a blueprint for storing information about **Malaysian states**. It's designed to keep track of each state's name and its specific weekend settings. Think of it as a record in your database for each state, making it easy to manage state-specific configurations.

  - **[Key] public int StateId { get; set; }**: This is the **unique identification number** for each state record. Every state will have its own StateId, ensuring it can be uniquely identified in the database.

  - **[Required] public required string StateName { get; set; }**: This field stores the **actual name of the state** (e.g., "Selangor", "Johor", "Perak").

  - **public int? WeekendId { get; set; }**: This field stores the **ID of the specific weekend configuration** that applies to this state. Since some states might have different weekend days (e.g., Friday/Saturday vs. Saturday/Sunday), this links the state to its relevant weekend schedule.



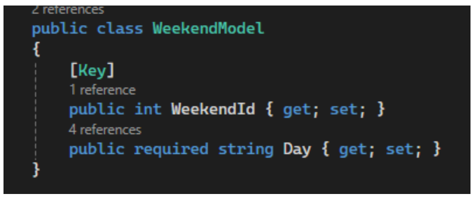
```
public class StateModel
{
    [Key]
    12 references
    public int StateId { get; set; }

    [Required]
    11 references
    public required string StateName { get; set; }

    27 references
    public int? WeekendId { get; set; }

    [ForeignKey("WeekendId")]
    6 references
    public virtual WeekendModel? Weekends { get; set; }
}
```

- **WeekendModel.cs**

  This WeekendModel is a very simple blueprint for defining **different weekend configurations**. It's designed to specify what day (or days) are considered a "weekend" for various purposes within system, especially when linked to things like states.

  - **[Key] public int WeekendId { get; set; }**: This is the **unique identification number** for each specific weekend configuration. For example, WeekendId 1 might be "Friday & Saturday", while WeekendId 2 could be "Saturday & Sunday".

  - **public required string Day { get; set; }**: This field stores the **name of the day (or days) that constitute the weekend** for this particular configuration.

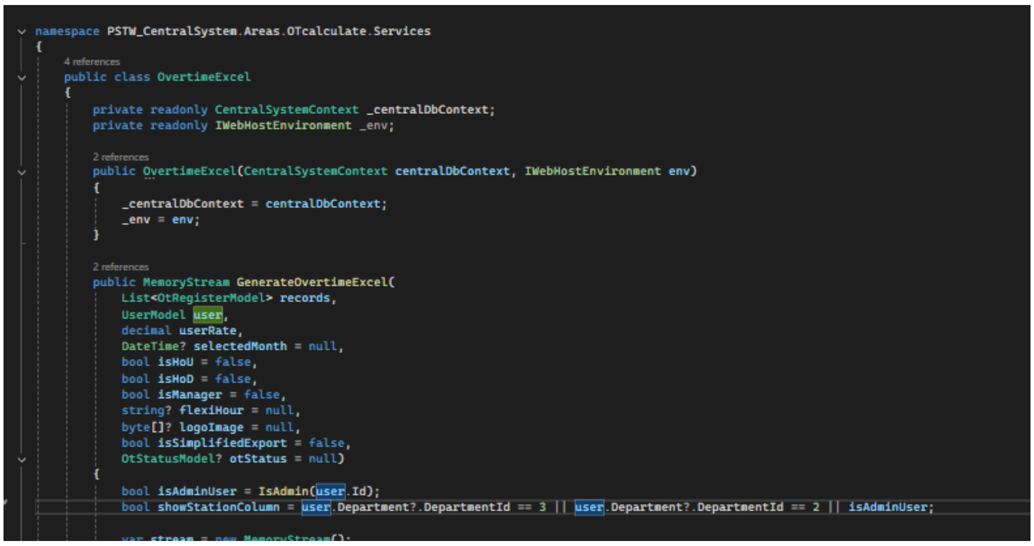


```
2 references
public class WeekendModel
{
    [Key]
    1 reference
    public int WeekendId { get; set; }
    4 references
    public required string Day { get; set; }
}
```

SERVICES & OVERTIME API

- **OvertimeExcel.cs**

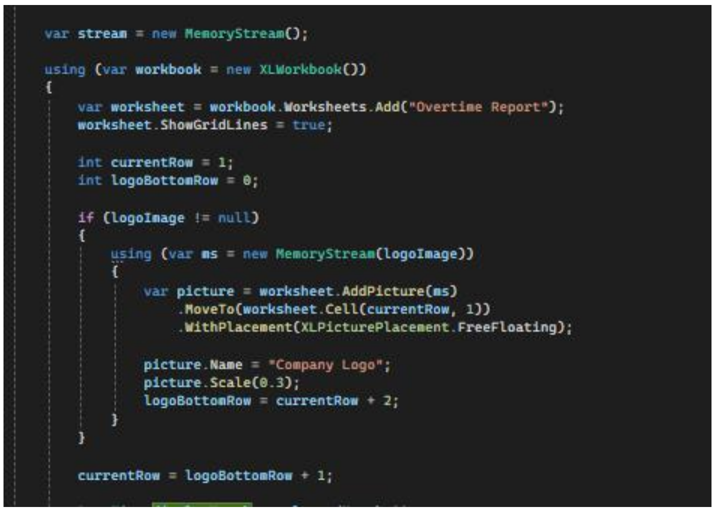
  This set of code is designed to **generate Excel reports for employee overtime**. It uses the **ClosedXML** library to create and manipulate Excel files. The system also interacts with a database to retrieve overtime records, user information, and approval statuses.



```
namespace PSTW_CentralSystem.Areas.OTcalculate.Services
{
    4 references
    public class OvertimeExcel
    {
        private readonly CentralSystemContext _centralDbContext;
        private readonly IWebHostEnvironment _env;

        2 references
        public OvertimeExcel(CentralSystemContext centralDbContext, IWebHostEnvironment env)
        {
            _centralDbContext = centralDbContext;
            _env = env;
        }

        2 references
        public MemoryStream GenerateOvertimeExcel(
            List<OtRegisterModel> records,
            UserModel user,
            decimal userRate,
            DateTime? selectedMonth = null,
            bool isHoU = false,
            bool isHoD = false,
            bool isManager = false,
            string? flexiHour = null,
            byte[]? logoImage = null,
            bool isSimplifiedExport = false,
            OtStatusModel? otStatus = null)
        {
            bool isAdminUser = IsAdmin(user.Id);
            bool showStationColumn = user.Department?.DepartmentId == 3 || user.Department?.DepartmentId == 2 || isAdminUser;
```

This section defines the OvertimeExcel class.

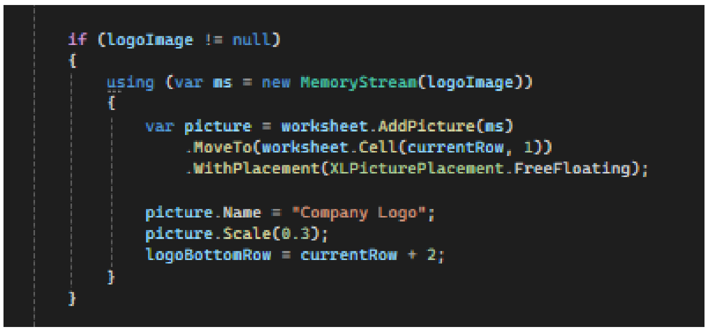
- **_centralDbContext**: This is used to connect to OT application's database (via Entity Framework Core) and fetch data like user details, overtime records, and public holidays.
- **_env**: This provides information about the web hosting environment, which is used to locate files like the company logo.
- **GenerateOvertimeExcel Method**: This is the main method responsible for creating the Excel file. It takes various parameters to customize the report, such as:
  - **records**: A list of overtime entries.
  - **user**: The employee for whom the report is being generated.
  - **userRate**: The user's basic salary.
  - **selectedMonth**: The month for the report.
  - **isHoU, isHoD, isManager**: Booleans to determine if the user viewing the report is a Head of Unit, Head of Department, or Manager, which influences what data (e.g., salary details) is visible.
  - **isSimplifiedExport**: A flag to generate a simpler version of the report.
  - **otStatus**: The approval status of the overtime.



```
var stream = new MemoryStream();

using (var workbook = new XLWorkbook())
{
    var worksheet = workbook.Worksheets.Add("Overtime Report");
    worksheet.ShowGridLines = true;

    int currentRow = 1;
    int logoBottomRow = 0;

    if (logoImage != null)
    {
        using (var ms = new MemoryStream(logoImage))
        {
            var picture = worksheet.AddPicture(ms)
                .MoveTo(worksheet.Cell(currentRow, 1))
                .WithPlacement(XLPicturePlacement.FreeFloating);

            picture.Name = "Company Logo";
            picture.Scale(0.3);
            logoBottomRow = currentRow + 2;
        }
    }

    currentRow = logoBottomRow + 1;
```

This is where the Excel file creation begins.

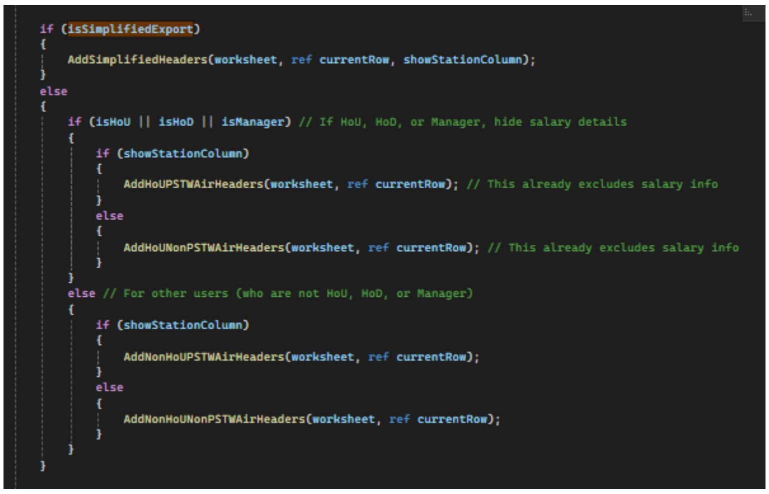
- **XLWorkbook()**: Initializes a new Excel workbook. The using statement ensures the workbook is properly disposed of after use, preventing resource leaks.
- **workbook.Worksheets.Add("Overtime Report")**: Adds a new sheet named "Overtime Report" to the workbook.
- **worksheet.ShowGridLines = true**: Makes the grid lines visible in the Excel sheet for better readability.



```
if (logoImage != null)
{
    using (var ms = new MemoryStream(logoImage))
    {
        var picture = worksheet.AddPicture(ms)
            .MoveTo(worksheet.Cell(currentRow, 1))
            .WithPlacement(XLPicturePlacement.FreeFloating);

        picture.Name = "Company Logo";
        picture.Scale(0.3);
        logoBottomRow = currentRow + 2;
    }
}
```

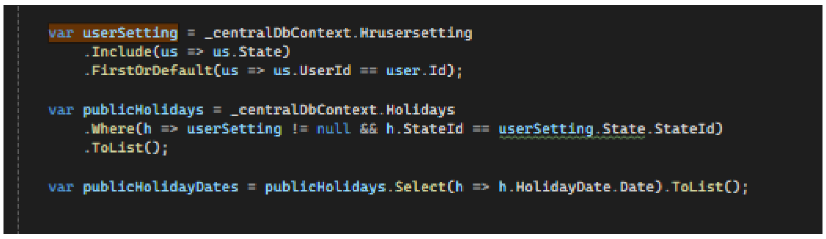
This part handles **adding a company logo** to the Excel report if an image is provided.

- It reads the logoImage (a byte array) into a MemoryStream.
- Then, it uses worksheet.AddPicture() to insert the image into the Excel sheet at a specific cell.


```
if (isSimplifiedExport)
{
    AddSimplifiedHeaders(worksheet, ref currentRow, showStationColumn);
}
else
{
    if (isHoU || isHoD || isManager) // If HoU, HoD, or Manager, hide salary details
    {
        if (showStationColumn)
        {
            AddHoUPSTWAirHeaders(worksheet, ref currentRow); // This already excludes salary info
        }
        else
        {
            AddHoUNonPSTWAirHeaders(worksheet, ref currentRow); // This already excludes salary info
        }
    }
    else // For other users (who are not HoU, HoD, or Manager)
    {
        if (showStationColumn)
        {
            AddNonHoUPSTWAirHeaders(worksheet, ref currentRow);
        }
        else
        {
            AddNonHoUNonPSTWAirHeaders(worksheet, ref currentRow);
        }
    }
}
```

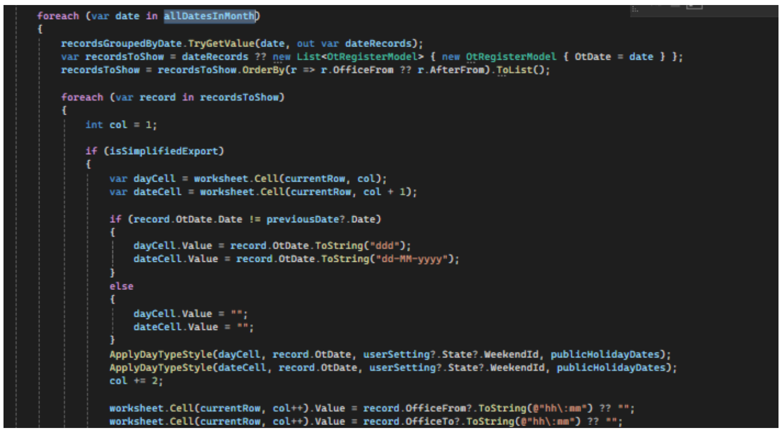
This crucial part **dynamically determines which set of headers to add** to the Excel report based on the user's role and the type of export.

- **isSimplifiedExport**: If true, a simpler set of headers is used.
- **isHoU || isHoD || isManager**: If the user viewing the report is a Head of Unit, Head of Department, or Manager, certain sensitive details like salary information are hidden.
- **showStationColumn**: This flag controls whether a "Station" column is included in the report.
- This branching logic ensures that the report adapts to different viewing permissions and export types.


```
var userSetting = _centralDbContext.Hrusersetting
    .Include(us => us.State)
    .FirstOrDefault(us => us.UserId == user.Id);

var publicHolidays = _centralDbContext.Holidays
    .Where(h => userSetting != null && h.StateId == userSetting.State.StateId)
    .ToList();

var publicHolidayDates = publicHolidays.Select(h => h.HolidayDate.Date).ToList();
```

This block retrieves **user-specific settings and public holidays** from the database.

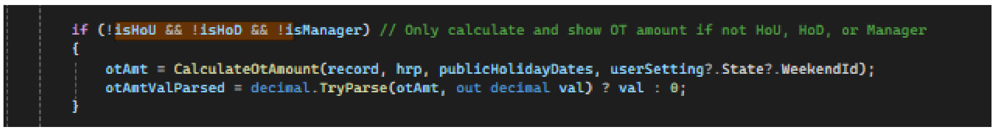
- It fetches Hrusersetting which might include the user's state.
- It then queries Holidays to get public holidays relevant to the user's state. These dates are used later to categorize overtime entries.



```
foreach (var date in allDatesInMonth)
{
    recordsGroupedByDate.TryGetValue(date, out var dateRecords);
    var recordsToShow = dateRecords ?? new List<OtRegisterModel> { new OtRegisterModel { OtDate = date } };
    recordsToShow = recordsToShow.OrderBy(r => r.OfficeFrom ?? r.AfterFrom).ToList();

    foreach (var record in recordsToShow)
    {
        int col = 1;

        if (isSimplifiedExport)
        {
            var dayCell = worksheet.Cell(currentRow, col);
            var dateCell = worksheet.Cell(currentRow, col + 1);

            if (record.OtDate.Date != previousDate?.Date)
            {
                dayCell.Value = record.OtDate.ToString("ddd");
                dateCell.Value = record.OtDate.ToString("dd-MM-yyyy");
            }
            else
            {
                dayCell.Value = "";
                dateCell.Value = "";
            }
            ApplyDayTypeStyle(dayCell, record.OtDate, userSetting?.State?.WeekendId, publicHolidayDates);
            ApplyDayTypeStyle(dateCell, record.OtDate, userSetting?.State?.WeekendId, publicHolidayDates);
            col += 2;

            worksheet.Cell(currentRow, col++).Value = record.OfficeFrom?.ToString(@"hh\:mm") ?? "";
            worksheet.Cell(currentRow, col++).Value = record.OfficeTo?.ToString(@"hh\:mm") ?? "";
```

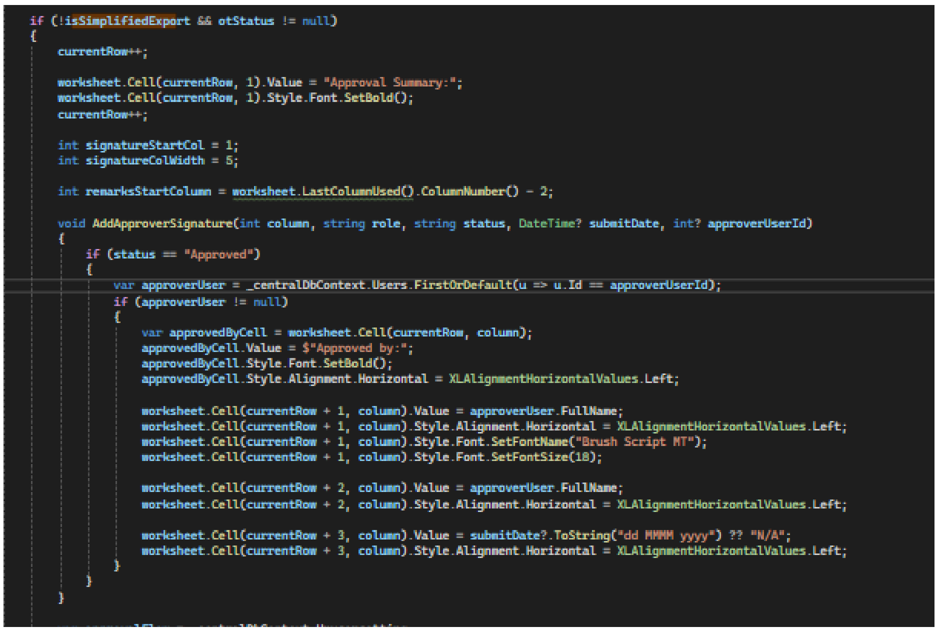
This is the **main loop that populates the Excel rows with overtime data**.

- It iterates through each day of the selected month (allDatesInMonth).
- For each day, it retrieves relevant overtime records (recordsGroupedByDate). If there are no records for a day, it still creates a row to ensure all dates in the month are represented.
- Inside the inner loop, it populates cells based on whether it's a simplified export or a detailed report, and applies specific styles (e.g., coloring weekend/holiday rows).



```
if (!isHoU && !isHoD && !isManager) // Only calculate and show OT amount if not HoU, HoD, or Manager
{
    otAmt = CalculateOtAmount(record, hrp, publicHolidayDates, userSetting?.State?.WeekendId);
    otAmtValParsed = decimal.TryParse(otAmt, out decimal val) ? val : 0;
}
```

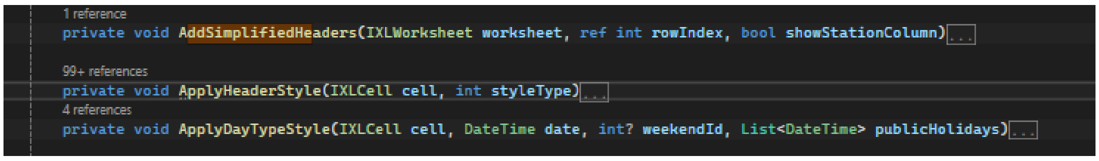
This condition ensures that the **Overtime Amount (OT Amt (RM)) is only calculated and displayed for users who are not HoU, HoD, or Manager**. This is a security or privacy measure to restrict sensitive financial information.



```
if (!isSimplifiedExport && otStatus != null)
{
    currentRow++;

    worksheet.Cell(currentRow, 1).Value = "Approval Summary:";
    worksheet.Cell(currentRow, 1).Style.Font.SetBold();
    currentRow++;

    int signatureStartCol = 1;
    int signatureColWidth = 5;

    int remarksStartColumn = worksheet.LastColumnUsed().ColumnNumber() - 2;

    void AddApproverSignature(int column, string role, string status, DateTime? submitDate, int? approverUserId)
    {
        if (status == "Approved")
        {
            var approverUser = _centralDbContext.Users.FirstOrDefault(u => u.Id == approverUserId);
            if (approverUser != null)
            {
                var approvedByCell = worksheet.Cell(currentRow, column);
                approvedByCell.Value = $"Approved by:";
                approvedByCell.Style.Font.SetBold();
                approvedByCell.Style.Alignment.Horizontal = XLAlignmentHorizontalValues.Left;

                worksheet.Cell(currentRow + 1, column).Value = approverUser.FullName;
                worksheet.Cell(currentRow + 1, column).Style.Alignment.Horizontal = XLAlignmentHorizontalValues.Left;
                worksheet.Cell(currentRow + 1, column).Style.Font.SetFontName("Brush Script MT");
                worksheet.Cell(currentRow + 1, column).Style.Font.SetFontSize(18);

                worksheet.Cell(currentRow + 2, column).Value = approverUser.FullName;
                worksheet.Cell(currentRow + 2, column).Style.Alignment.Horizontal = XLAlignmentHorizontalValues.Left;

                worksheet.Cell(currentRow + 3, column).Value = submitDate?.ToString("dd MMMM yyyy") ?? "N/A";
                worksheet.Cell(currentRow + 3, column).Style.Alignment.Horizontal = XLAlignmentHorizontalValues.Left;
            }
        }
    }
```

This block adds **additional sections to the report based on the export type and approval status**.

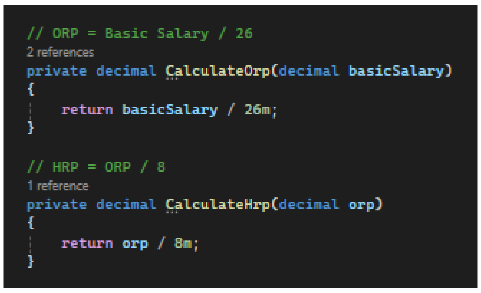
- For non-simplified exports with otStatus available, it includes an "Approval Summary" showing who approved the overtime and when, along with remarks about public holidays and weekends.
- For simplified exports, it only adds the remarks section.



```
1 reference
private void AddSimplifiedHeaders(IXLWorksheet worksheet, ref int rowIndex, bool showStationColumn)...

99+ references
private void ApplyHeaderStyle(IXLCell cell, int styleType)...
4 references
private void ApplyDayTypeStyle(IXLCell cell, DateTime date, int? weekendId, List<DateTime> publicHolidays)...
```

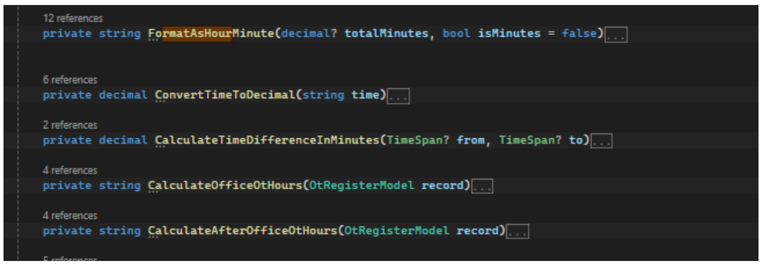
These are **helper methods for formatting the Excel sheet**.

- **AddSimplifiedHeaders / AddNonHoUPSTWAirHeaders / AddHoUPSTWAirHeaders**: These methods define the structure and content of the report headers based on the export type and user role. They merge cells and set initial column values.
- **ApplyHeaderStyle**: Applies consistent styling (bold font, alignment, borders, and background color) to header cells. The styleType parameter allows for different color schemes.
- **ApplyDayTypeStyle**: Applies specific background colors to cells for public holidays (pink) and weekends (light blue) to visually distinguish them.



```
// ORP = Basic Salary / 26
2 references
private decimal CalculateOrp(decimal basicSalary)
{
    return basicSalary / 26m;
}

// HRP = ORP / 8
1 reference
private decimal CalculateHrp(decimal orp)
{
    return orp / 8m;
}
```

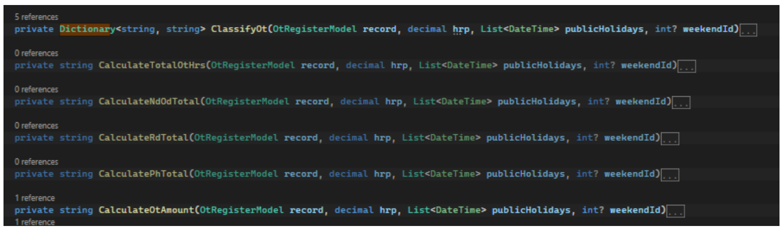
These methods define the **calculations for ORP (Ordinary Rate of Pay) and HRP (Hourly Rate of Pay)** based on the provided basicSalary. These rates are crucial for calculating overtime pay.



```
12 references
private string FormatAsHourMinute(decimal? totalMinutes, bool isMinutes = false)[...]


6 references
private decimal ConvertTimeToDecimal(string time)[...]

2 references
private decimal CalculateTimeDifferenceInMinutes(TimeSpan? from, TimeSpan? to)[...]

4 references
private string CalculateOfficeOtHours(OtRegisterModel record)[...]

4 references
private string CalculateAfterOfficeOtHours(OtRegisterModel record)[...]
```

These are various **utility methods for handling time and calculations**:

- **FormatAsHourMinute**: Converts a total number of minutes into a "hours:minutes" string format.
- **ConvertTimeToDecimal**: Converts a "hours:minutes" string back into a decimal representation of hours.
- **CalculateTimeDifferenceInMinutes**: Calculates the difference between two TimeSpan values in minutes, handling cases where the to time is on the next day.
- **CalculateOfficeOtHours / CalculateAfterOfficeOtHours**: Compute the net overtime hours for "office hours" and "after office hours" by subtracting any recorded breaks.

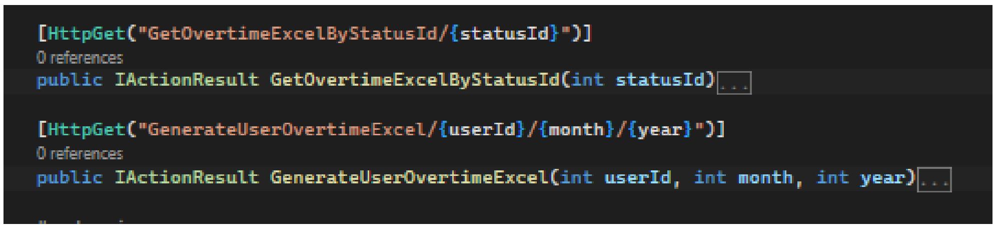


```
5 references
private Dictionary<string, string> ClassifyOt(OtRegisterModel record, decimal hrp, List<DateTime> publicHolidays, int? weekendId)...

0 references
private string CalculateTotalOtHrs(OtRegisterModel record, decimal hrp, List<DateTime> publicHolidays, int? weekendId)...

0 references
private string CalculateNdOdTotal(OtRegisterModel record, decimal hrp, List<DateTime> publicHolidays, int? weekendId)...

0 references
private string CalculateRdTotal(OtRegisterModel record, decimal hrp, List<DateTime> publicHolidays, int? weekendId)...

0 references
private string CalculatePhTotal(OtRegisterModel record, decimal hrp, List<DateTime> publicHolidays, int? weekendId)...

1 reference
private string CalculateOtAmount(OtRegisterModel record, decimal hrp, List<DateTime> publicHolidays, int? weekendId)...
1 reference
```

These methods are responsible for **classifying and calculating different types of overtime and their corresponding amounts**:

- **ClassifyOt**: Categorizes overtime hours into different types (Normal Day After Office, Off Day, Rest Day, Public Holiday) and their sub-categories (e.g., "OD < 4" for Off Day overtime less than 4 hours). This is a key part of how overtime is structured.
- **CalculateTotalOtHrs, CalculateNdOdTotal, CalculateRdTotal, CalculatePhTotal**: These methods sum up the classified overtime hours for different categories.
- **CalculateOtAmount**: This method calculates the total overtime pay based on the classified overtime hours, HRP, ORP, and the day type, applying different rates as per the defined logic (e.g., 1.5x for Normal Day after office, 2x for Rest Day after office, etc.).

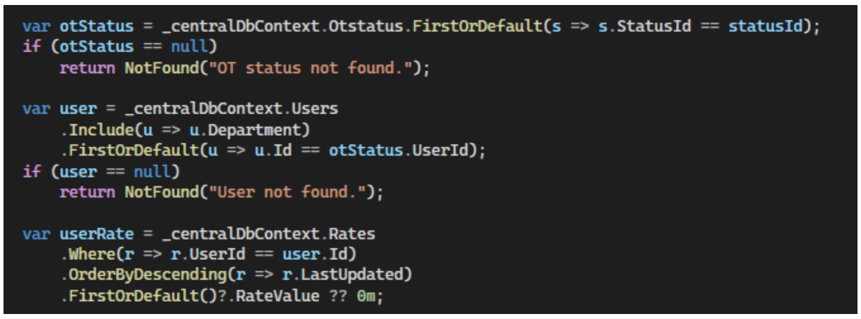
- **OvertimeAPI.cs (Excel)**



```
[HttpGet("GetOvertimeExcelByStatusId/{statusId}")]
0 references
public IActionResult GetOvertimeExcelByStatusId(int statusId)...

[HttpGet("GenerateUserOvertimeExcel/{userId}/{month}/{year}")]
0 references
public IActionResult GenerateUserOvertimeExcel(int userId, int month, int year)...
```

GetOvertimeExcelByStatusId/{statusId}: This endpoint generates an overtime Excel report for a specific overtime status ID. It's likely used when a user wants to download the report for an already submitted and approved/pending overtime request.

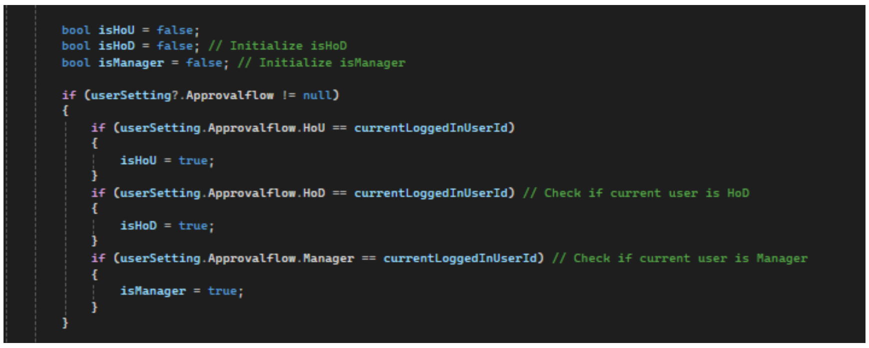
GenerateUserOvertimeExcel/{userId}/{month}/{year}: This endpoint generates an overtime Excel report for a specific user and month/year. This is probably used for general reporting or for a user to download their own monthly report.



```
var otStatus = _centralDbContext.Otstatus.FirstOrDefault(s => s.StatusId == statusId);
if (otStatus == null)
    return NotFound("OT status not found.");

var user = _centralDbContext.Users
    .Include(u => u.Department)
    .FirstOrDefault(u => u.Id == otStatus.UserId);
if (user == null)
    return NotFound("User not found.");

var userRate = _centralDbContext.Rates
    .Where(r => r.UserId == user.Id)
    .OrderByDescending(r => r.LastUpdated)
    .FirstOrDefault()?.RateValue ?? 0m;
```

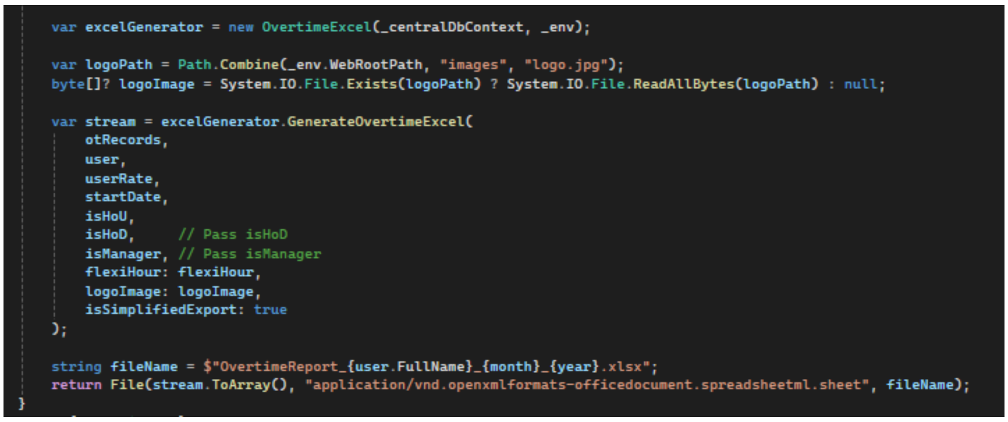
In both API methods, before generating the Excel, the code **retrieves all necessary data from the database**:

- **Otstatus**: Gets the approval status of the overtime request.
- **Users**: Fetches details of the user associated with the overtime, including their department.
- **Rates**: Retrieves the user's latest salary rate.
- **Otregisters**: Gets the actual overtime records.
- **Hrusersetting**: Retrieves user-specific settings, including flexi-hour details and approval flow information.



```
bool isHoU = false;
bool isHoD = false; // Initialize isHoD
bool isManager = false; // Initialize isManager

if (userSetting?.Approvalflow != null)
{
    if (userSetting.Approvalflow.HoU == currentLoggedInUserId)
    {
        isHoU = true;
    }
    if (userSetting.Approvalflow.HoD == currentLoggedInUserId) // Check if current user is HoD
    {
        isHoD = true;
    }
    if (userSetting.Approvalflow.Manager == currentLoggedInUserId) // Check if current user is Manager
    {
        isManager = true;
    }
}
```

This part **determines the role of the currently logged-in user (HoU, HoD, or Manager)** by comparing their ID with the approval flow settings. This information is then passed to the GenerateOvertimeExcel method to control the visibility of salary details in the report.



```
    var excelGenerator = new OvertimeExcel(_centralDbContext, _env);

    var logoPath = Path.Combine(_env.WebRootPath, "images", "logo.jpg");
    byte[]? logoImage = System.IO.File.Exists(logoPath) ? System.IO.File.ReadAllBytes(logoPath) : null;

    var stream = excelGenerator.GenerateOvertimeExcel(
        otRecords,
        user,
        userRate,
        startDate,
        isHoU,
        isHoD,      // Pass isHoD
        isManager, // Pass isManager
        flexiHour: flexiHour,
        logoImage: logoImage,
        isSimplifiedExport: true
    );

    string fileName = $"OvertimeReport_{user.FullName}_{month}_{year}.xlsx";
    return File(stream.ToArray(), "application/vnd.openxmlformats-officedocument.spreadsheetml.sheet", fileName);
}
```

This is the **final step in the API methods**:

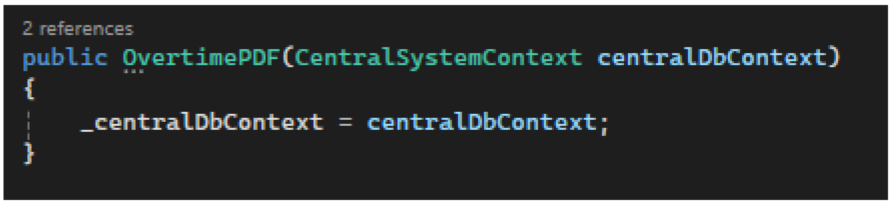
- An instance of the OvertimeExcel class is created.
- The **company logo** is loaded as a byte array from the wwwroot/images folder.
- The GenerateOvertimeExcel method is called with all the collected data and flags.
- The generated Excel file (as a MemoryStream) is then returned as a file download to the client with a dynamically generated filename.

- **OvertimePDF.cs**

  This file defines the **OvertimePDF class**, which is responsible for **generating PDF documents for overtime records**. It uses the QuestPDF library to construct professional-looking PDF reports, including detailed overtime tables, user information, and approval signatures.

  - **Constructor (public OvertimePDF(...))**

    - **Purpose**: This is how an OvertimePDF object is created. It needs access to the database (CentralSystemContext) so it can fetch necessary data like user settings, holidays, etc., when generating a PDF.

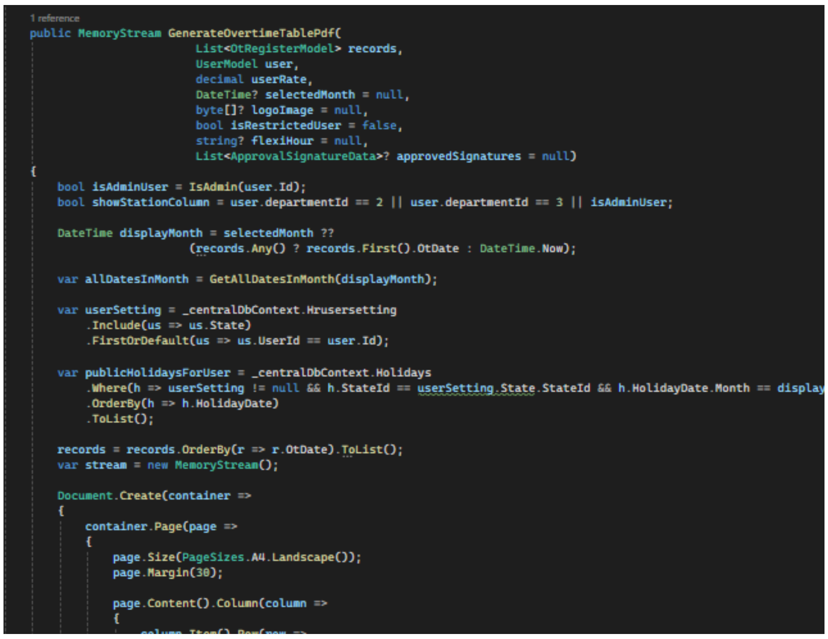
      ```
      2 references
      public OvertimePDF(CentralSystemContext centralDbContext)
      {
          _centralDbContext = centralDbContext;
      }
      ```

  - **GenerateOvertimeTablePdf** Method

    - **Purpose:** This is the core method for creating a detailed overtime PDF report, typically used for HR viewing as it includes salary-related calculations. It takes various inputs like the list of overtime records, user details, salary rate, and optional elements like a company logo and approval signatures.

    - **Table Generation (ComposeTable)**: Calls a helper method to build the main overtime table with various columns including detailed time breakdowns, classified OT hours, and calculated amounts. **Crucially, it handles conditional display of salary details based on the isRestrictedUser flag.**

    - **Output**: Generates the complete PDF as a MemoryStream, which can then be sent to the user (e.g., for download).

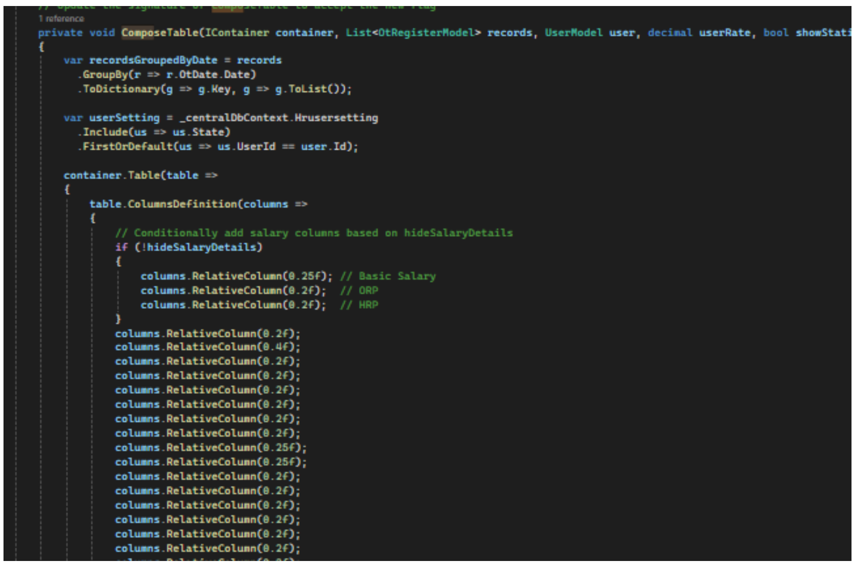


```
1 reference
public MemoryStream GenerateOvertimeTablePdf(
                        List<OtRegisterModel> records,
                        UserModel user,
                        decimal userRate,
                        DateTime? selectedMonth = null,
                        byte[]? logoImage = null,
                        bool isRestrictedUser = false,
                        string? flexiHour = null,
                        List<ApprovalSignatureData>? approvedSignatures = null)
{
    bool isAdminUser = IsAdmin(user.Id);
    bool showStationColumn = user.departmentId == 2 || user.departmentId == 3 || isAdminUser;

    DateTime displayMonth = selectedMonth ??
                        (records.Any() ? records.First().OtDate : DateTime.Now);

    var allDatesInMonth = GetAllDatesInMonth(displayMonth);

    var userSetting = _centralDbContext.Hrusersetting
        .Include(us => us.State)
        .FirstOrDefault(us => us.UserId == user.Id);

    var publicHolidaysForUser = _centralDbContext.Holidays
        .Where(h => userSetting != null && h.StateId == userSetting.State.StateId && h.HolidayDate.Month == display
        .OrderBy(h => h.HolidayDate)
        .ToList();

    records = records.OrderBy(r => r.OtDate).ToList();
    var stream = new MemoryStream();

    Document.Create(container =>
    {
        container.Page(page =>
        {
            page.Size(PageSizes.A4.Landscape());
            page.Margin(30);

            page.Content().Column(column =>
            {
```

- **ComposeTable Method**

  - **Purpose**: This private helper method is responsible for **constructing the detailed overtime data table** within the PDF. It iterates through each day of the month and each overtime record for that day, formatting the data into table cells.

  - **Key Feature**: **Conditional Display of Salary Information**. The **hideSalaryDetails** parameter (derived from **isRestrictedUser** in **GenerateOvertimeTablePdf**) controls whether columns like "Basic Salary", "ORP", "HRP", and "OT Amount" are included in the table, both in the headers and the individual rows. This is a critical security or access control feature.

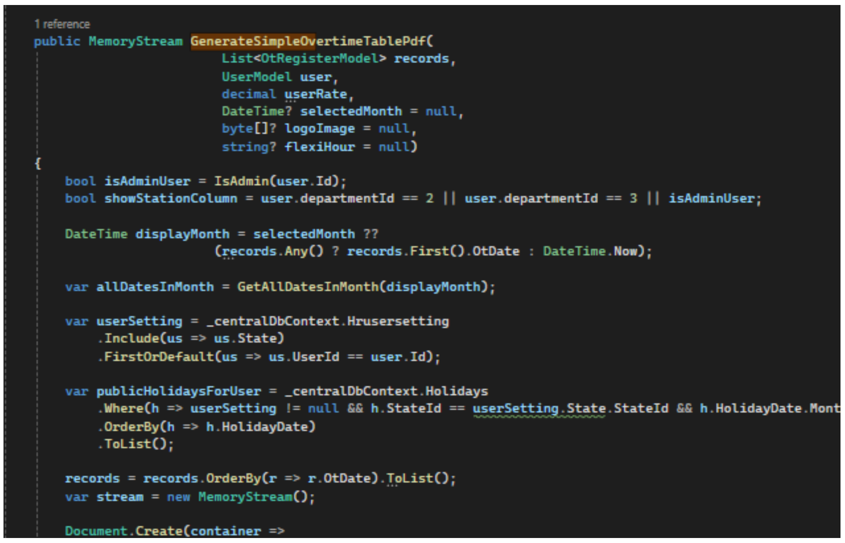


```
1 reference
private void ComposeTable(IContainer container, List<OtRegisterModel> records, UserModel user, decimal userRate, bool showStati
{
    var recordsGroupedByDate = records
      .GroupBy(r => r.OtDate.Date)
      .ToDictionary(g => g.Key, g => g.ToList());

    var userSetting = _centralDbContext.Hrusersetting
      .Include(us => us.State)
      .FirstOrDefault(us => us.UserId == user.Id);

    container.Table(table =>
    {
        table.ColumnsDefinition(columns =>
        {
            // Conditionally add salary columns based on hideSalaryDetails
            if (!hideSalaryDetails)
            {
                columns.RelativeColumn(0.25f); // Basic Salary
                columns.RelativeColumn(0.2f);  // ORP
                columns.RelativeColumn(0.2f);  // HRP
            }
            columns.RelativeColumn(0.2f);
            columns.RelativeColumn(0.4f);
            columns.RelativeColumn(0.2f);
            columns.RelativeColumn(0.2f);
            columns.RelativeColumn(0.2f);
            columns.RelativeColumn(0.2f);
            columns.RelativeColumn(0.2f);
            columns.RelativeColumn(0.2f);
            columns.RelativeColumn(0.25f);
            columns.RelativeColumn(0.25f);
            columns.RelativeColumn(0.2f);
            columns.RelativeColumn(0.2f);
            columns.RelativeColumn(0.2f);
            columns.RelativeColumn(0.2f);
            columns.RelativeColumn(0.2f);
            columns.RelativeColumn(0.2f);
```

- **GenerateSimpleOvertimeTablePdf Method**

  - **Purpose**: This method generates a **simplified overtime PDF report**, specifically designed *without* any salary-related columns. This is likely intended for general user viewing where salary details are not relevant or should not be exposed. Use in OtRecords.

  - **Key Difference**: It calls **ComposeSimpleTable** which explicitly excludes salary columns.

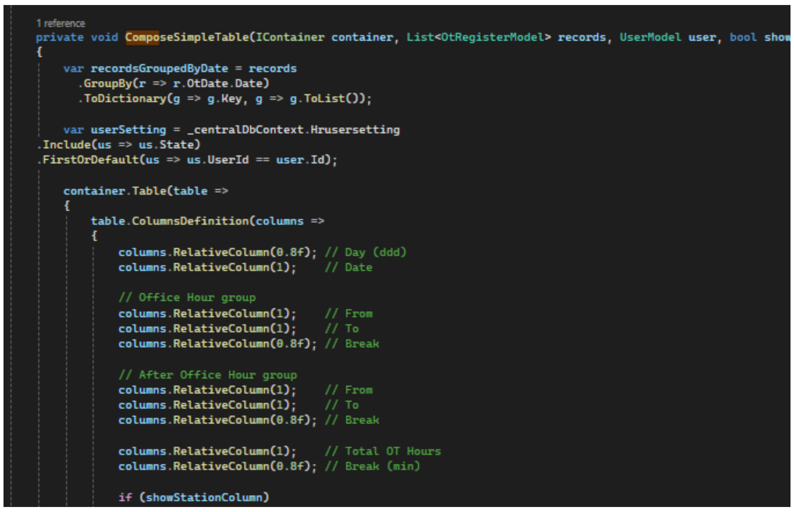


```
1 reference
public MemoryStream GenerateSimpleOvertimeTablePdf(
                        List<OtRegisterModel> records,
                        UserModel user,
                        decimal userRate,
                        DateTime? selectedMonth = null,
                        byte[]? logoImage = null,
                        string? flexiHour = null)
{
    bool isAdminUser = IsAdmin(user.Id);
    bool showStationColumn = user.departmentId == 2 || user.departmentId == 3 || isAdminUser;

    DateTime displayMonth = selectedMonth ??
                       (records.Any() ? records.First().OtDate : DateTime.Now);

    var allDatesInMonth = GetAllDatesInMonth(displayMonth);

    var userSetting = _centralDbContext.Hrusersetting
        .Include(us => us.State)
        .FirstOrDefault(us => us.UserId == user.Id);

    var publicHolidaysForUser = _centralDbContext.Holidays
        .Where(h => userSetting != null && h.StateId == userSetting.State.StateId && h.HolidayDate.Mont
        .OrderBy(h => h.HolidayDate)
        .ToList();

    records = records.OrderBy(r => r.OtDate).ToList();
    var stream = new MemoryStream();

    Document.Create(container =>
```

- **ComposeSimpleTable Method**

  - **Purpose**: This private helper method builds the simplified overtime data table, explicitly **excluding all salary-related columns and calculations**.

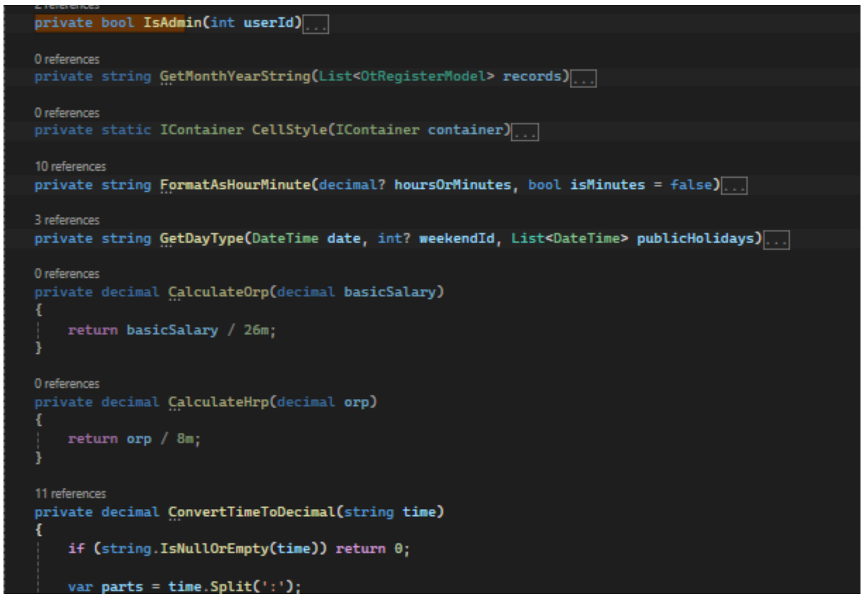
  - **Key Difference**: Unlike ComposeTable, it does not have a hideSalaryDetails parameter and its column and header definitions are hardcoded to *not* include salary information.



```
1 reference
private void ComposeSimpleTable(IContainer container, List<OtRegisterModel> records, UserModel user, bool show
{
    var recordsGroupedByDate = records
      .GroupBy(r => r.OtDate.Date)
      .ToDictionary(g => g.Key, g => g.ToList());

    var userSetting = _centralDbContext.Hrusersetting
.Include(us => us.State)
.FirstOrDefault(us => us.UserId == user.Id);

    container.Table(table =>
    {
        table.ColumnsDefinition(columns =>
        {
            columns.RelativeColumn(0.8f); // Day (ddd)
            columns.RelativeColumn(1);    // Date

            // Office Hour group
            columns.RelativeColumn(1);    // From
            columns.RelativeColumn(1);    // To
            columns.RelativeColumn(0.8f); // Break

            // After Office Hour group
            columns.RelativeColumn(1);    // From
            columns.RelativeColumn(1);    // To
            columns.RelativeColumn(0.8f); // Break

            columns.RelativeColumn(1);    // Total OT Hours
            columns.RelativeColumn(0.8f); // Break (min)

            if (showStationColumn)
```

- **Helper Methods (IsAdmin, GetDayType, CalculateOfficeOtHours, CalculateOtAmount, etc.)**

  - **Purpose**: These are various utility methods that OvertimePDF uses to perform calculations (like OT hours, amounts), classify days (Normal Day, Off Day, Rest Day, Public Holiday based on state settings), and determine user roles (e.g., if a user is an admin). They ensure the data presented in the PDF is accurate and correctly categorized.



```
private bool IsAdmin(int userId)[...]

0 references
private string GetMonthYearString(List<OtRegisterModel> records)[...]

0 references
private static IContainer CellStyle(IContainer container)[...]

10 references
private string FormatAsHourMinute(decimal? hoursOrMinutes, bool isMinutes = false)[...]

3 references
private string GetDayType(DateTime date, int? weekendId, List<DateTime> publicHolidays)[...]

0 references
private decimal CalculateOrp(decimal basicSalary)
{
    return basicSalary / 26m;
}

0 references
private decimal CalculateHrp(decimal orp)
{
    return orp / 8m;
}

11 references
private decimal ConvertTimeToDecimal(string time)
{
    if (string.IsNullOrEmpty(time)) return 0;

    var parts = time.Split(':');
```

- **OvertimeAPI.cs (PDF)**
    - **[HttpGet("GetOvertimePdfByStatusId/{statusId}")]**

        - **Purpose**: This API endpoint is used to generate a **detailed overtime PDF** for a specific overtime submission, identified by its statusId. It's designed to provide comprehensive data, including salary calculations, but with a **controlled visibility for salary details**.

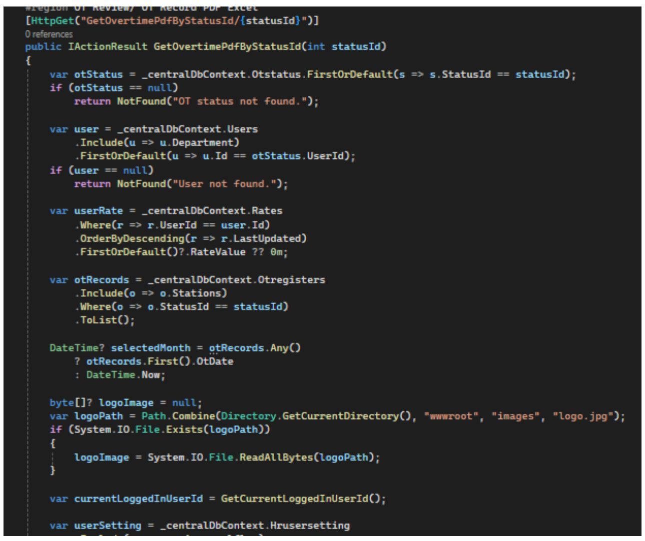
        - **Data Fetching**: It retrieves the OtStatus (overtime approval status), UserModel (employee details), RateModel (employee's basic salary), and OtRegisterModel (individual overtime entries) from the database using the provided statusId.

        - **Determining Salary Visibility (hideSalaryDetails)**: This is a critical security check. It inspects the Approvalflow associated with the user's settings. If the *currently logged-in user* is the Head of Unit (HoU), Head of Department (HoD), or Manager for the employee whose OT is being viewed, the hideSalaryDetails flag is set to true. This means these specific approvers will *not* see salary information in the generated PDF, even though the GenerateOvertimeTablePdf method *can* show it. This enforces a separation of duties or data access.

        - **Approval Signatures**: Collects information about who has approved the OT request and when, to be included in the PDF.

        - **PDF Generation**: It **instantiates OvertimePDF** (var pdfGenerator = new OvertimePDF(_centralDbContext);) and then calls its GenerateOvertimeTablePdf method, passing all the fetched data, including the calculated hideSalaryDetails flag.

        - **Response**: Returns the generated PDF file to the client (return File(stream.ToArray(), "application/pdf", fileName);).



```
#region OT Review/ OT Record PDF Excel
[HttpGet("GetOvertimePdfByStatusId/{statusId}")]
0 references
public IActionResult GetOvertimePdfByStatusId(int statusId)
{
    var otStatus = _centralDbContext.Otstatus.FirstOrDefault(s => s.StatusId == statusId);
    if (otStatus == null)
        return NotFound("OT status not found.");

    var user = _centralDbContext.Users
        .Include(u => u.Department)
        .FirstOrDefault(u => u.Id == otStatus.UserId);
    if (user == null)
        return NotFound("User not found.");

    var userRate = _centralDbContext.Rates
        .Where(r => r.UserId == user.Id)
        .OrderByDescending(r => r.LastUpdated)
        .FirstOrDefault()?.RateValue ?? 0m;

    var otRecords = _centralDbContext.Otregisters
        .Include(o => o.Stations)
        .Where(o => o.StatusId == statusId)
        .ToList();

    DateTime? selectedMonth = otRecords.Any()
        ? otRecords.First().OtDate
        : DateTime.Now;

    byte[]? logoImage = null;
    var logoPath = Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images", "logo.jpg");
    if (System.IO.File.Exists(logoPath))
    {
        logoImage = System.IO.File.ReadAllBytes(logoPath);
    }

    var currentLoggedInUserId = GetCurrentLoggedInUserId();

    var userSetting = _centralDbContext.Hrusersetting
```

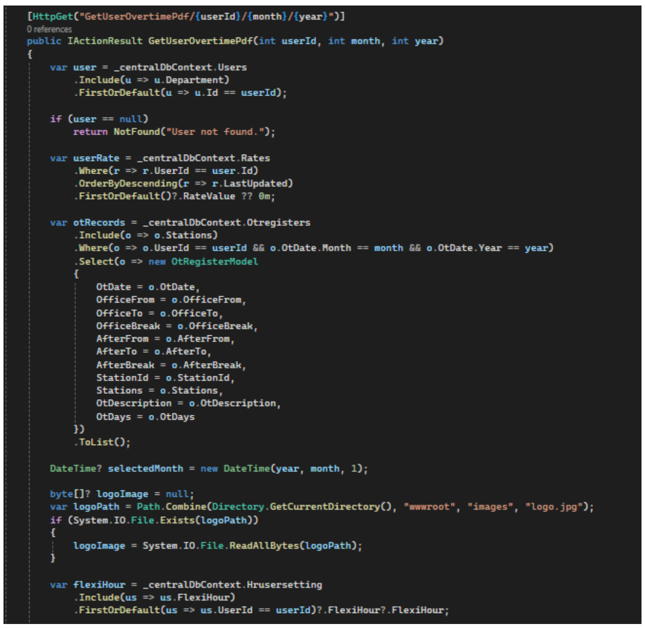
- **[HttpGet("GetUserOvertimePdf/{userId}/{month}/{year}")]**

  - **Purpose**: This API endpoint is used to generate a **simplified overtime PDF** for a specific user and month/year. This version **never includes salary details** and is likely for general employee access or simplified reporting.

  - **Data Fetching**: Retrieves user details and overtime records for the specified userId, month, and year.

  - **PDF Generation**: Instantiates OvertimePDF and calls its GenerateSimpleOvertimeTablePdf method. Notice that this method *does not* receive a hideSalaryDetails flag, as GenerateSimpleOvertimeTablePdf is designed to *always* exclude salary information.

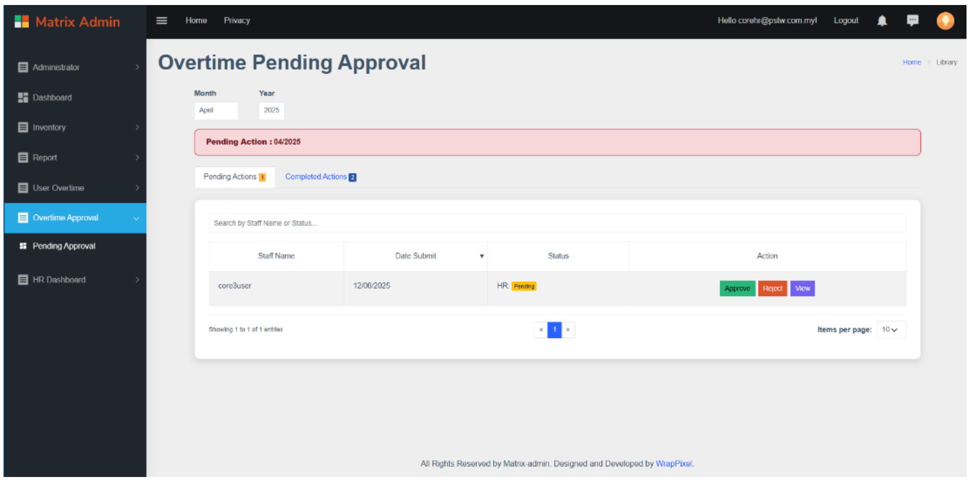
  - **Response**: Returns the generated PDF file to the client.



```
[HttpGet("GetUserOvertimePdf/{userId}/{month}/{year}")]
0 references
public IActionResult GetUserOvertimePdf(int userId, int month, int year)
{
    var user = _centralDbContext.Users
        .Include(u => u.Department)
        .FirstOrDefault(u => u.Id == userId);

    if (user == null)
        return NotFound("User not found.");

    var userRate = _centralDbContext.Rates
        .Where(r => r.UserId == user.Id)
        .OrderByDescending(r => r.LastUpdated)
        .FirstOrDefault()?.RateValue ?? 0m;

    var otRecords = _centralDbContext.Otregisters
        .Include(o => o.Stations)
        .Where(o => o.UserId == userId && o.OtDate.Month == month && o.OtDate.Year == year)
        .Select(o => new OtRegisterModel
        {
            OtDate = o.OtDate,
            OfficeFrom = o.OfficeFrom,
            OfficeTo = o.OfficeTo,
            OfficeBreak = o.OfficeBreak,
            AfterFrom = o.AfterFrom,
            AfterTo = o.AfterTo,
            AfterBreak = o.AfterBreak,
            StationId = o.StationId,
            Stations = o.Stations,
            OtDescription = o.OtDescription,
            OtDays = o.OtDays
        })
        .ToList();

    DateTime? selectedMonth = new DateTime(year, month, 1);

    byte[]? logoImage = null;
    var logoPath = Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images", "logo.jpg");
    if (System.IO.File.Exists(logoPath))
    {
        logoImage = System.IO.File.ReadAllBytes(logoPath);
    }

    var flexiHour = _centralDbContext.Hrusersetting
        .Include(us => us.FlexiHour)
        .FirstOrDefault(us => us.UserId == userId)?.FlexiHour?.FlexiHour;
```

VIEW & OVERTIME API

- **ApprovalDashboard**
  - **Approval.cshtml**



Overtime Pending Approval

Month: April
Year: 2025

Pending Action : 04/2025

Pending Actions 1 | Completed Actions 2

Search by Staff Name or Status...

Staff Name	Date Submit	Status	Action
core3user	12/06/2025	HR: Pending	Approve Reject View

Showing 1 to 1 of 1 entries

Items per page: 10

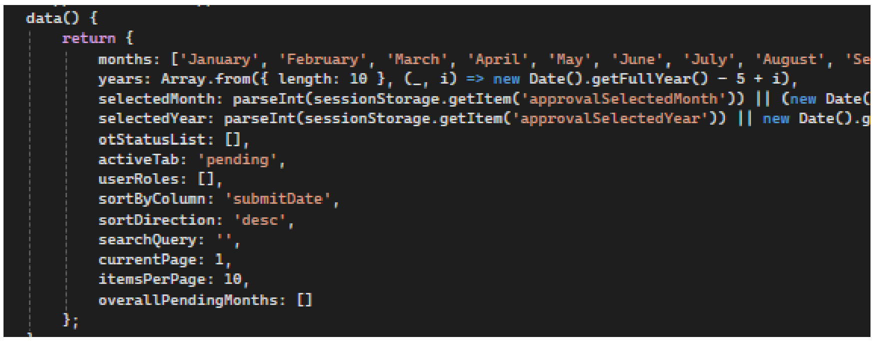
All Rights Reserved by Matrix-admin. Designed and Developed by WrapPixel.

This file (Approval.cshtml) represents the web page where approvers (like HoU, HoD, Manager, HR) can view and manage pending overtime requests.

➢ **<div id="app">...</div>**
- **Purpose**: This div acts as the **main container** for your Vue.js application. All the HTML elements and components inside this div will be managed by Vue.js. The id="app" matches the app.mount('#app') line in the script, linking the Vue instance to this part of the page.

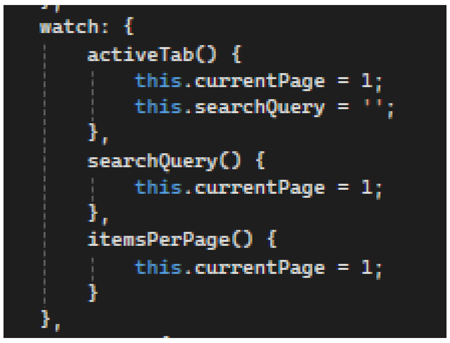
➢ **const app = Vue.createApp({...})**
- **Purpose**: This line **initializes your Vue.js application**. It creates a new Vue application instance and defines its core properties: data, watch, computed, and methods.

➢ **data()** function
- **months, years:** Arrays for dropdown options.
- **selectedMonth, selectedYear**: Current selected month/year, initialized from sessionStorage (to remember user's last selection) or current date.
- **otStatusList**: An array that will hold the list of overtime statuses fetched from the API.
- **activeTab**: Controls which tab ("Pending Actions" or "Completed Actions") is currently active.
- **searchQuery**: Stores the text typed into the search box.
- **currentPage, itemsPerPage**: Used for pagination (how many items to show per page).
- **overallPendingMonths**: A list of months that have any pending action for the current approver across all users they can approve for.



```
data() {
    return {
        months: ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'Se
        years: Array.from({ length: 10 }, (_, i) => new Date().getFullYear() - 5 + i),
        selectedMonth: parseInt(sessionStorage.getItem('approvalSelectedMonth')) || (new Date(
        selectedYear: parseInt(sessionStorage.getItem('approvalSelectedYear')) || new Date().g
        otStatusList: [],
        activeTab: 'pending',
        userRoles: [],
        sortByColumn: 'submitDate',
        sortDirection: 'desc',
        searchQuery: '',
        currentPage: 1,
        itemsPerPage: 10,
        overallPendingMonths: []
    };
```

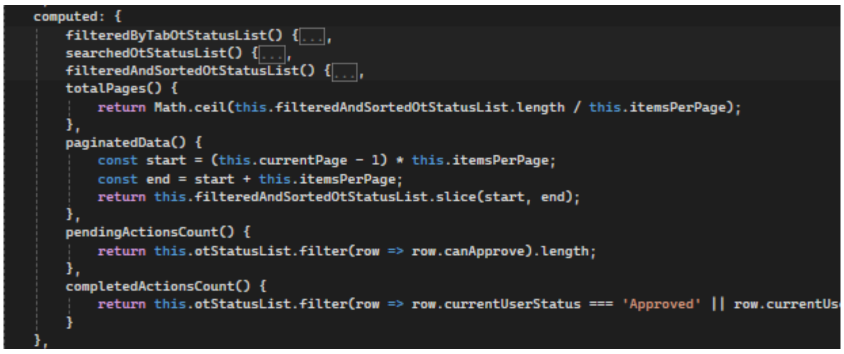
- ➢ **watch property**
  - **Purpose**: watch allows you to **perform actions when specific data properties change**. This is useful for reacting to user input or state changes that require more complex logic than simply updating the display.
  - **Functionality**:
    - When **activeTab** changes, it resets **currentPage** and **searchQuery** to ensure a fresh view of the new tab's data.
    - When **searchQuery** changes, it resets **currentPage** so the search starts from the first page.
    - When **itemsPerPage** changes, it also resets **currentPage** for consistent pagination.



```
watch: {
    activeTab() {
        this.currentPage = 1;
        this.searchQuery = '';
    },
    searchQuery() {
        this.currentPage = 1;
    },
    itemsPerPage() {
        this.currentPage = 1;
    }
},
```

- ➢ **computed properties**
  - **Purpose**: computed properties are like **dynamic values that are calculated based on other data properties**. They are "cached" and only re-calculated when their dependencies change, making them efficient for filtering, sorting, or complex display logic.
  - **Key Computed Properties**:
    - **filteredByTabOtStatusList:** Filters **otStatusList** to show either "pending" or "completed" items based on **activeTab**.

   - **searchedOtStatusList:** Further filters **filteredByTabOtStatusList** based on the text in **searchQuery** across various fields.
   - **filteredAndSortedOtStatusList:** Sorts **searchedOtStatusList** by the **sortByColumn** and **sortDirection**. It includes custom sorting logic for dates and status fields.
   - **totalPages**: Calculates the total number of pages needed for pagination.
   - **paginatedData:** Slices **filteredAndSortedOtStatusList** to show only the items for the current page.
   - **pendingActionsCount, completedActionsCount:** Count items for the badges on the tabs.



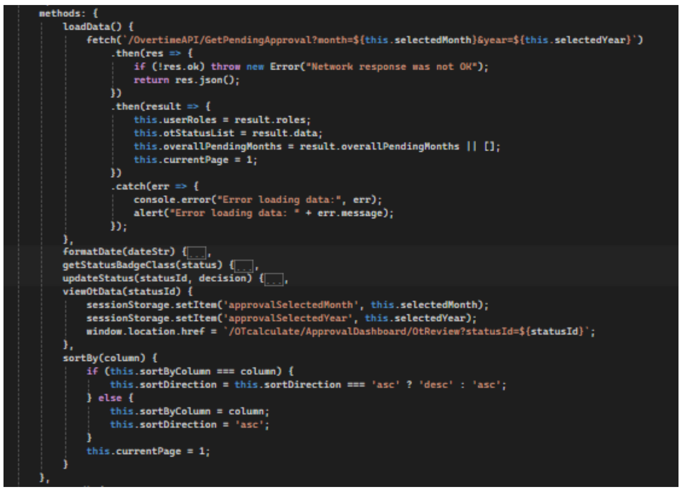
```
computed: {
    filteredByTabOtStatusList() {...},
    searchedOtStatusList() {...},
    filteredAndSortedOtStatusList() {...},
    totalPages() {
        return Math.ceil(this.filteredAndSortedOtStatusList.length / this.itemsPerPage);
    },
    paginatedData() {
        const start = (this.currentPage - 1) * this.itemsPerPage;
        const end = start + this.itemsPerPage;
        return this.filteredAndSortedOtStatusList.slice(start, end);
    },
    pendingActionsCount() {
        return this.otStatusList.filter(row => row.canApprove).length;
    },
    completedActionsCount() {
        return this.otStatusList.filter(row => row.currentUserStatus === 'Approved' || row.currentUs
    }
},
```

➢ **methods property**

- **Purpose**: methods are **functions that perform actions** in your Vue application, typically in response to user interactions (like clicks) or other events.
- **Key Methods**:
   - **loadData():** Crucially, this method fetches the overtime approval data from the backend API. It makes an fetch request to **/OvertimeAPI/GetPendingApproval**, processes the JSON response, and updates **otStatusList** and other data properties.
   - **formatDate(dateStr):** Formats date strings for display.
   - **getStatusBadgeClass(status):** Returns the appropriate CSS class for status badges (e.g., badge-approved, badge-rejected).
   - **updateStatus(statusId, decision):** Sends a request to the backend API **(/OvertimeAPI/UpdateApprovalStatus) to approve or reject an OT request.** It confirms with the user before sending the request and reloads the data upon success.
   - **viewOtData(statusId):** Navigates the user to another page **(/OTcalculate/ApprovalDashboard/OtReview)** to view the full

details of a specific OT request, remembering the current month/year selection.

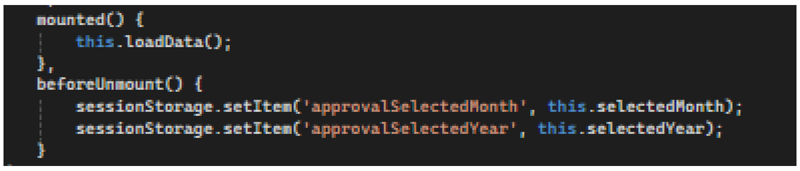
  - **sortBy(column):** Handles table header clicks for sorting, toggling sort direction.



```
methods: {
    loadData() {
        fetch(`/OvertimeAPI/GetPendingApproval?month=${this.selectedMonth}&year=${this.selectedYear}`)
            .then(res => {
                if (!res.ok) throw new Error("Network response was not OK");
                return res.json();
            })
            .then(result => {
                this.userRoles = result.roles;
                this.otStatusList = result.data;
                this.overallPendingMonths = result.overallPendingMonths || [];
                this.currentPage = 1;
            })
            .catch(err => {
                console.error("Error loading data:", err);
                alert("Error loading data: " + err.message);
            });
    },
    formatDate(dateStr) {...},
    getStatusBadgeClass(status) {...},
    updateStatus(statusId, decision) {...},
    viewOtData(statusId) {
        sessionStorage.setItem('approvalSelectedMonth', this.selectedMonth);
        sessionStorage.setItem('approvalSelectedYear', this.selectedYear);
        window.location.href = `/OTcalculate/ApprovalDashboard/OtReview?statusId=${statusId}`;
    },
    sortBy(column) {
        if (this.sortByColumn === column) {
            this.sortDirection = this.sortDirection === 'asc' ? 'desc' : 'asc';
        } else {
            this.sortByColumn = column;
            this.sortDirection = 'asc';
        }
        this.currentPage = 1;
    }
},
```

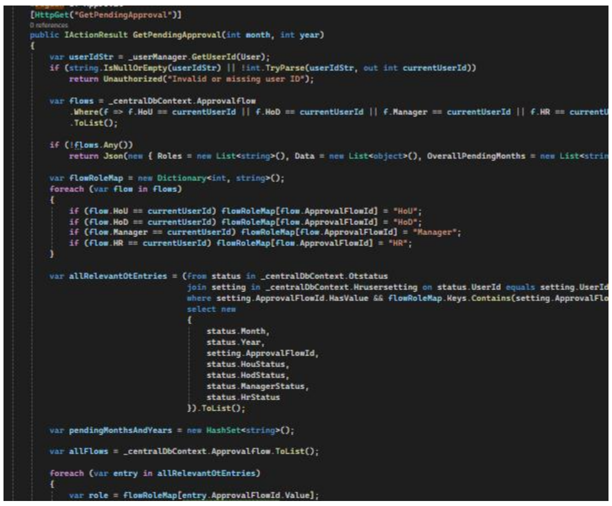
➢ **mounted & beforeUnmount**

- **Purpose**: These are special methods that Vue automatically calls at specific points in the component's lifecycle.
- **mounted()**:
  - **Purpose**: Executed once the Vue application has been "mounted" to the HTML element (i.e., the page is ready).
  - **Functionality**: Calls loadData() to fetch the initial data for display when the page first loads.
- **beforeUnmount()**:
  - **Purpose**: Executed just before the Vue application instance is destroyed.
  - **Functionality**: Saves the selectedMonth and selectedYear to sessionStorage so that when the user returns to this page, their previous selections are remembered.



```
mounted() {
    this.loadData();
},
beforeUnmount() {
    sessionStorage.setItem('approvalSelectedMonth', this.selectedMonth);
    sessionStorage.setItem('approvalSelectedYear', this.selectedYear);
}
```

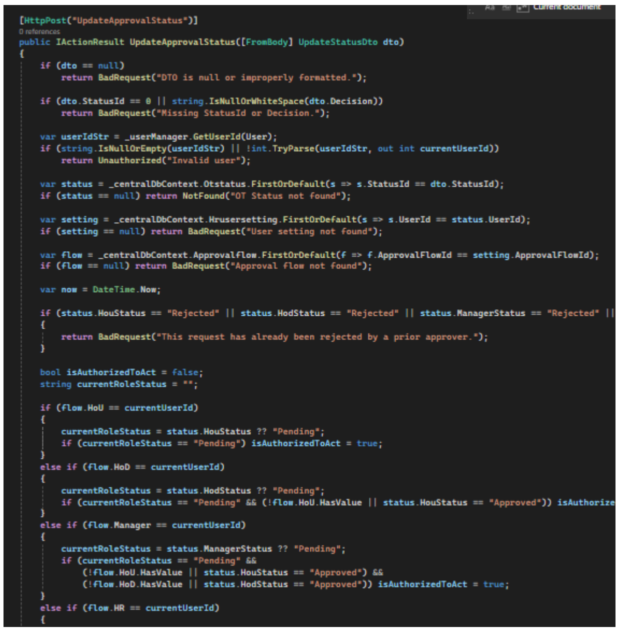
- **OvertimeAPI.cs (Approval.cshtml)**
  - **[HttpGet("GetPendingApproval")]**
    - **Purpose**: This API endpoint provides the **data displayed on the Approval.cshtml page**. It's designed to return a list of overtime status records that the currently logged-in user is authorized to approve, along with their current status in the approval flow.
    - **Key Functionality**:
      - **User Identification**: Determines the currentUserId from the authenticated user.
      - **Identify Approver Roles**: Finds all Approvalflow entries where the currentUserId is listed as an HoU, HoD, Manager, or HR. This defines what the logged-in user can approve.
      - **Filter Overtime Entries**: Queries Otstatus records, joining with Hrusersetting to find overtime requests associated with the identified approval flows for the month and year provided by the front-end.
      - **Determine CanApprove Logic**: For each retrieved overtime entry, it applies complex business rules to determine:
        - The currentUserStatus (e.g., "Pending," "Approved," "Rejected") from the perspective of the logged-in user's role.
        - Whether the logged-in user canApprove this particular request *at this stage* (e.g., an HoD can only approve if the HoU has already approved or if HoU approval isn't required).
        - If the request IsOverallRejected by any previous approver.
      - **Overall Pending Months**: Identifies any months (not just the currently selected one) where the user has *any* pending action across *any* employee under their approval flow. This is used for the "Pending Action" notice on the front-end.



```
[HttpGet("GetPendingApproval")]
0 references
public IActionResult GetPendingApproval(int month, int year)
{
    var userIdStr = _userManager.GetUserId(User);
    if (string.IsNullOrEmpty(userIdStr) || !int.TryParse(userIdStr, out int currentUserId))
        return Unauthorized("Invalid or missing user ID");

    var flows = _centralDbContext.Approvalflow
        .Where(f => f.HoU == currentUserId || f.HoD == currentUserId || f.Manager == currentUserId || f.HR == currentU
        .ToList();

    if (!flows.Any())
        return Json(new { Roles = new List<string>(), Data = new List<object>(), OverallPendingMonths = new List<strin

    var flowRoleMap = new Dictionary<int, string>();
    foreach (var flow in flows)
    {
        if (flow.HoU == currentUserId) flowRoleMap[flow.ApprovalFlowId] = "HoU";
        if (flow.HoD == currentUserId) flowRoleMap[flow.ApprovalFlowId] = "HoD";
        if (flow.Manager == currentUserId) flowRoleMap[flow.ApprovalFlowId] = "Manager";
        if (flow.HR == currentUserId) flowRoleMap[flow.ApprovalFlowId] = "HR";
    }

    var allRelevantOtEntries = (from status in _centralDbContext.Otstatus
                                join setting in _centralDbContext.Hrusersetting on status.UserId equals setting.UserId
                                where setting.ApprovalFlowId.HasValue && flowRoleMap.Keys.Contains(setting.ApprovalFlo
                                select new
                                {
                                    status.Month,
                                    status.Year,
                                    setting.ApprovalFlowId,
                                    status.HouStatus,
                                    status.HodStatus,
                                    status.ManagerStatus,
                                    status.HrStatus
                                }).ToList();

    var pendingMonthsAndYears = new HashSet<string>();

    var allFlows = _centralDbContext.ApprovalFlow.ToList();

    foreach (var entry in allRelevantOtEntries)
    {
        var role = flowRoleMap[entry.ApprovalFlowId.Value];
```

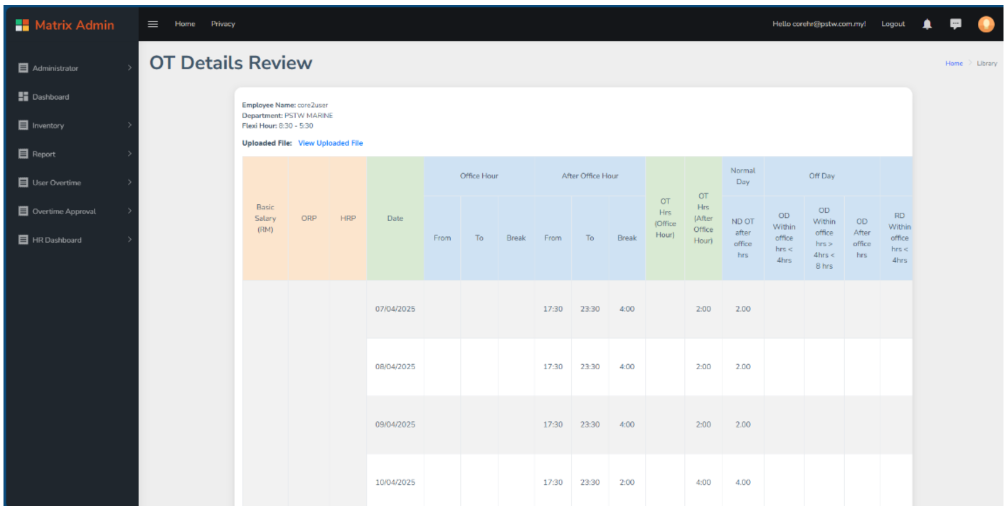
➢ **[HttpPost("UpdateApprovalStatus")]**

- **Purpose**: This API endpoint receives a request from the front-end to **update the approval status of a specific overtime request** (either "Approved" or "Rejected").
- **Key Functionality**:
  - **Input Validation**: Checks if the dto (Data Transfer Object containing StatusId and Decision) is valid.
  - **Authorization & Sequential Approval**:
    - Identifies the currentUserId and retrieves the Otstatus, Hrusersetting, and Approvalflow associated with the request.
    - It implements the **sequential approval logic**: it verifies that the currentUserId is indeed authorized to act on this specific request *at this particular stage* in the approval flow. For example, an HoD can only approve if the HoU has already approved (or if HoU is not part of the flow). It also prevents action if a previous approver has already rejected.
  - **Update Status**: If authorized, it updates the relevant status field (HouStatus, HodStatus, ManagerStatus, or HrStatus) and the corresponding SubmitDate for the current approver based on the Decision received.
  - **Database Save**: Saves the changes to the database.
  - **Response**: Returns an Ok response upon success or a BadRequest/NotFound/Unauthorized response with an appropriate error message if conditions are not met.



```
[HttpPost("UpdateApprovalStatus")]
0 references
public IActionResult UpdateApprovalStatus([FromBody] UpdateStatusDto dto)
{
    if (dto == null)
        return BadRequest("DTO is null or improperly formatted.");

    if (dto.StatusId == 0 || string.IsNullOrWhiteSpace(dto.Decision))
        return BadRequest("Missing StatusId or Decision.");

    var userIdStr = _userManager.GetUserId(User);
    if (string.IsNullOrEmpty(userIdStr) || !int.TryParse(userIdStr, out int currentUserId))
        return Unauthorized("Invalid user");

    var status = _centralDbContext.Otstatus.FirstOrDefault(s => s.StatusId == dto.StatusId);
    if (status == null) return NotFound("OT Status not found");

    var setting = _centralDbContext.Hrusersetting.FirstOrDefault(s => s.UserId == status.UserId);
    if (setting == null) return BadRequest("User setting not found");

    var flow = _centralDbContext.Approvalflow.FirstOrDefault(f => f.ApprovalFlowId == setting.ApprovalFlowId);
    if (flow == null) return BadRequest("Approval flow not found");

    var now = DateTime.Now;

    if (status.HouStatus == "Rejected" || status.HodStatus == "Rejected" || status.ManagerStatus == "Rejected" ||
    {
        return BadRequest("This request has already been rejected by a prior approver.");
    }

    bool isAuthorizedToAct = false;
    string currentRoleStatus = "";

    if (flow.HoU == currentUserId)
    {
        currentRoleStatus = status.HouStatus ?? "Pending";
        if (currentRoleStatus == "Pending") isAuthorizedToAct = true;
    }
    else if (flow.HoD == currentUserId)
    {
        currentRoleStatus = status.HodStatus ?? "Pending";
        if (currentRoleStatus == "Pending" && (!flow.HoU.HasValue || status.HouStatus == "Approved")) isAuthorize
    }
    else if (flow.Manager == currentUserId)
    {
        currentRoleStatus = status.ManagerStatus ?? "Pending";
        if (currentRoleStatus == "Pending" &&
            (!flow.HoU.HasValue || status.HouStatus == "Approved") &&
            (!flow.HoD.HasValue || status.HodStatus == "Approved")) isAuthorizedToAct = true;
    }
    else if (flow.HR == currentUserId)
    {
```

- **OtReview.cshtml**



OT Details Review

Employee Name: core2user
Department: PSTW MARINE
Flexi Hour: 8:30 - 5:30

Uploaded File: View Uploaded File

Basic Salary (RM)	ORP	HRP	Date	Office Hour			After Office Hour			OT Hrs (Office Hour)	OT Hrs (After Office Hour)	Normal Day	Off Day			
				From	To	Break	From	To	Break			ND OT after office hrs	OD Within office hrs < 4hrs	OD Within office hrs > 4hrs < 8 hrs	OD After office hrs	RD Within office hrs < 4hrs
			07/04/2025				17:30	23:30	4:00		2:00	2.00				
			08/04/2025				17:30	23:30	4:00		2:00	2.00				
			09/04/2025				17:30	23:30	4:00		2:00	2.00				
			10/04/2025				17:30	23:30	2:00		4:00	4.00				

- **<div id="reviewApp">...</div>**
  - **Purpose**: This div serves as the **root element** for the Vue.js application on the OT Review page. All dynamic content and interactions are confined within this boundary.
- **const reviewApp = Vue.createApp({...})**
  - **Purpose**: This line initiates the Vue.js application instance specifically for this review page, defining its reactive data, computed properties, methods, and lifecycle hooks.
- **data() function**
  - **Purpose**: This function defines the initial state for the OtReview page. These variables store all the information needed to render the page and respond to user interactions.
  - **Key Variables**:
    - **otRecords**: An array that will hold all the individual overtime entries for the selected submission.
    - **userInfo**: An object containing details about the employee whose OT is being reviewed (e.g., fullName, departmentName, basicSalary, flexiHour, filePath for uploaded documents).
    - **currentEditRecord:** A temporary object used to hold the data of an overtime record when it's being edited in the modal form.
    - **airStations, marineStations**: Lists of available stations for dropdowns in the edit modal.
    - **approverRole:** The role of the currently logged-in user (e.g., "HoU", "HoD") obtained from the backend.

    - **houStatus, hodStatus, managerStatus, hrStatus**: The current approval status for each approver level for this specific OT submission.
    - **userState, publicHolidays:** Information needed for calculating day types (Normal Day, Weekend, Public Holiday).
    - **breakOptions**: Predefined options for break durations in the edit modal.

➢ **computed properties**

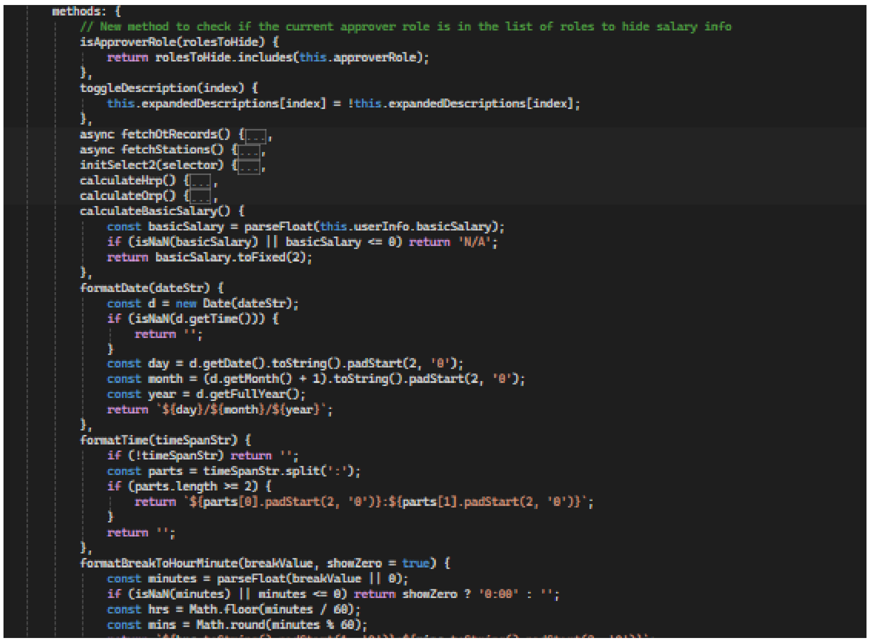
- **Purpose**: These properties dynamically calculate values based on the data properties. They ensure the UI is consistent and up-to-date with the underlying data, often performing filtering, sorting, or complex display logic.
- **Key Computed Properties**:
    - **showStationColumn, showAirStationDropdown, showMarineStationDropdown**: Determine visibility of station-related columns/dropdowns based on the employee's department or if the user is an admin.
    - **sortedOtRecords**: Sorts the otRecords by date for display.
    - **totals**: Calculates and aggregates totals for all relevant columns (breaks, classified OT hours, total OT hours, and OT amount) from all otRecords.
    - **hasApproverActedLocal**: A crucial property that determines if the **current approver has already made a decision** (Approved/Rejected) on this specific OT submission. This is used to disable "Edit" and "Delete" buttons for individual records if a decision has already been made by that approver.

➢ **watch property**

- **Purpose**: Watches for changes in specific data properties and executes a function when a change occurs.
- **Functionality**:
    - **currentEditRecord.stationId:** Updates the jQuery Select2 dropdown for stations when the stationId in the edit modal changes, ensuring the visual dropdown matches the model's value.
    - **airStations, marineStations:** Initializes or re-initializes the Select2 dropdowns when the station lists are loaded or updated.

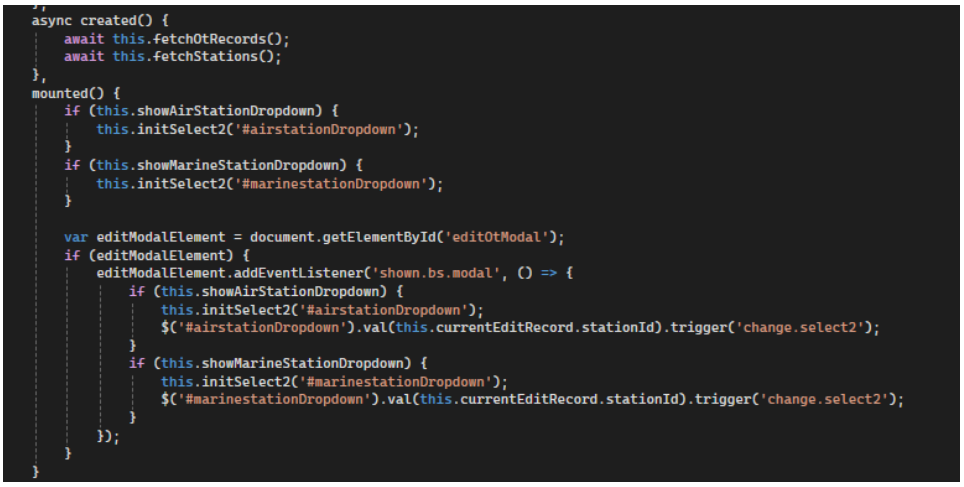
- **methods property**
  - **Purpose**: These are the functions that handle user interactions and perform the business logic on the client-side.
  - **Key Methods**:
    - **isApproverRole(rolesToHide):** Checks if the current approver's role is in a specified list of roles (used to conditionally hide salary info).
    - **toggleDescription(index):** Expands/collapses long descriptions in the table.
    - **fetchOtRecords():** Fetches all individual overtime records and related user/approval data for the current statusId from the OvertimeAPI. This is the primary data loading method for the page.
    - **fetchStations():** Fetches lists of Air and Marine stations from OvertimeAPI for use in the edit modal.
    - **initSelect2(selector):** Configures and initializes the jQuery Select2 plugin for dropdowns in the edit modal.
    - **Calculation Methods (calculateHrp, calculateOrp, calculateBasicSalary, calculateRawDuration, classifyOt, calculateTotalOtHrs, calculateNdOdTotal, calculateRdTotal, calculatePhTotal, calculateOtAmount, calculateModalTotalOtHrs, calculateModalTotalBreakHrs)**: These are crucial client-side functions that mirror the backend's logic to calculate various overtime hours (office, after office), categorize them by day type (Normal Day, Off Day, Rest Day, Public Holiday), and compute total hours and amounts. These ensure that the data displayed and the totals match the backend calculations.
    - **formatDate, formatTime, formatBreakToHourMinute, formatTimeFromDecimal, parseTimeSpanToDecimalHours, parseTime:** Utility methods for formatting and parsing date/time values for display and calculations.
    - **roundMinutes(fieldName):** Rounds minute values in time inputs (e.g., to the nearest 30 minutes) for consistency.
    - **editRecord(id):** Opens the edit modal and populates it with the data of the selected overtime record. It also disables editing if the approver has already acted.
    - **submitEdit():** Sends the updated overtime record data to the OvertimeAPI (UpdateOtRecordByApprover) after performing client-side validation.

- **deleteRecord(id):** Sends a request to the OvertimeAPI (DeleteOvertimeInOtReview) to delete an overtime record. It includes a confirmation step and checks if the approver has already acted.
- **fetchPublicHolidays(stateId):** Fetches public holidays relevant to the user's state from the API.
- **getDayType(dateStr):** Determines if a given date is a Normal Day, Off Day, Rest Day, or Public Holiday based on fetched state and holiday data.
- **saveAsPdf(), printPdf():** Triggers the generation and download/print of the full PDF report by making requests to the OvertimeAPI's PDF endpoints (GetOvertimePdfByStatusId).
- **exportToExcel():** Triggers the generation and download of an Excel report by making a request to the OvertimeAPI's Excel endpoint.
- **validateTimeRangeForSubmission, validateTimeFields:** Client-side validation for time inputs in the edit modal, ensuring logical time ranges.



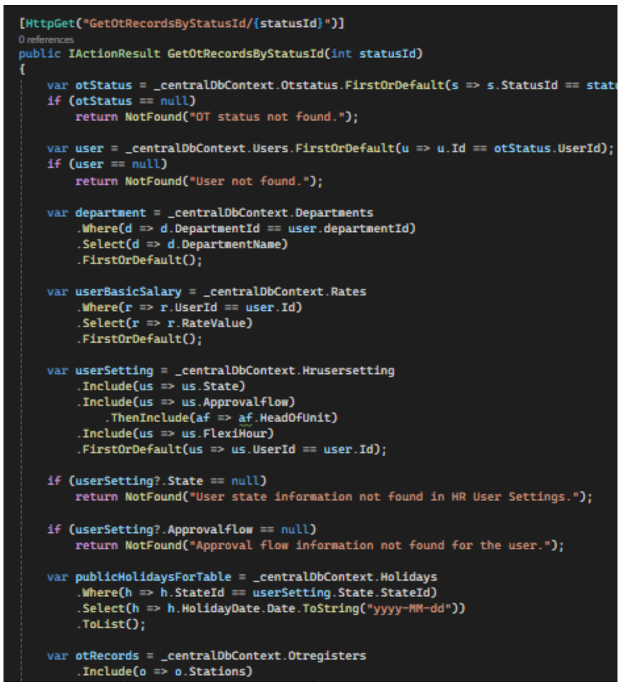
```
methods: {
    // New method to check if the current approver role is in the list of roles to hide salary info
    isApproverRole(rolesToHide) {
        return rolesToHide.includes(this.approverRole);
    },
    toggleDescription(index) {
        this.expandedDescriptions[index] = !this.expandedDescriptions[index];
    },
    async fetchOtRecords() {...},
    async fetchStations() {...},
    initSelect2(selector) {...},
    calculateHrp() {...},
    calculateOrp() {...},
    calculateBasicSalary() {
        const basicSalary = parseFloat(this.userInfo.basicSalary);
        if (isNaN(basicSalary) || basicSalary <= 0) return 'N/A';
        return basicSalary.toFixed(2);
    },
    formatDate(dateStr) {
        const d = new Date(dateStr);
        if (isNaN(d.getTime())) {
            return '';
        }
        const day = d.getDate().toString().padStart(2, '0');
        const month = (d.getMonth() + 1).toString().padStart(2, '0');
        const year = d.getFullYear();
        return `${day}/${month}/${year}`;
    },
    formatTime(timeSpanStr) {
        if (!timeSpanStr) return '';
        const parts = timeSpanStr.split(':');
        if (parts.length >= 2) {
            return `${parts[0].padStart(2, '0')}:${parts[1].padStart(2, '0')}`;
        }
        return '';
    },
    formatBreakToHourMinute(breakValue, showZero = true) {
        const minutes = parseFloat(breakValue || 0);
        if (isNaN(minutes) || minutes <= 0) return showZero ? '0:00' : '';
        const hrs = Math.floor(minutes / 60);
        const mins = Math.round(minutes % 60);
```

- **created & mounted**
  - **created()**:
    - **Purpose**: Executed immediately after the instance is created, before it's mounted to the DOM. This is a good place to fetch initial data.
    - **Functionality**: Calls fetchOtRecords() to get the overtime data and fetchStations() to get station lists as soon as the component is created.
  - **mounted()**:
    - **Purpose**: Executed after the Vue application has been rendered to the HTML. This is the ideal place for DOM-related operations (like initializing jQuery plugins).
    - **Functionality**: Initializes the Select2 dropdowns for stations. It also adds an event listener to the modal to re-initialize Select2 when the modal is shown, ensuring it works correctly.



```
async created() {
    await this.fetchOtRecords();
    await this.fetchStations();
},
mounted() {
    if (this.showAirStationDropdown) {
        this.initSelect2('#airstationDropdown');
    }
    if (this.showMarineStationDropdown) {
        this.initSelect2('#marinestationDropdown');
    }

    var editModalElement = document.getElementById('editOtModal');
    if (editModalElement) {
        editModalElement.addEventListener('shown.bs.modal', () => {
            if (this.showAirStationDropdown) {
                this.initSelect2('#airstationDropdown');
                $('#airstationDropdown').val(this.currentEditRecord.stationId).trigger('change.select2');
            }
            if (this.showMarineStationDropdown) {
                this.initSelect2('#marinestationDropdown');
                $('#marinestationDropdown').val(this.currentEditRecord.stationId).trigger('change.select2');
            }
        });
    }
}
```

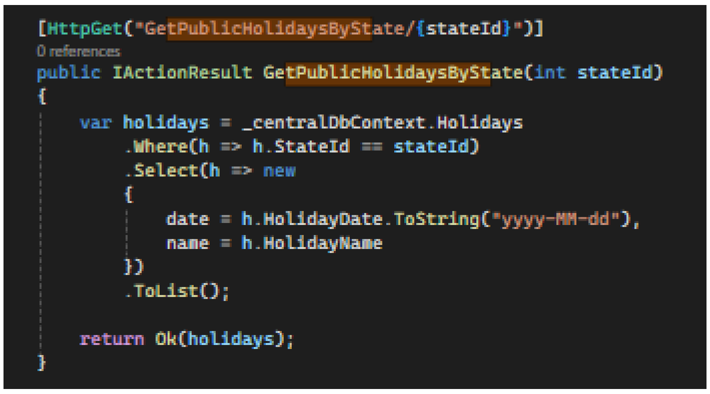
- **OvertimeAPI.cs (OtReview.cshtml)**
  - **[HttpGet("GetOtRecordsByStatusId/{statusId}")]**
    - **Purpose**: This is the primary API endpoint that the OtReview.cshtml page calls to **fetch all the detailed overtime entries** belonging to a specific OtStatusId. It also provides relevant user information, approval statuses, and day type definitions.
    - **Key Functionality**:
      - **Data Aggregation**: Retrieves the main Otstatus record, the associated User details (including department), the User's Rate (basic salary), and Hrusersetting (which includes State and Approvalflow for holiday and role-based logic).
      - **Public Holidays**: Fetches public holidays relevant to the user's state.
      - **Dynamic Day Type Calculation**: For each individual Otregister record, it dynamically determines its DayType ("Normal Day," "Off Day," "Rest Day," "Public Holiday") based on the date and the user's configured state and its weekend rules, and known public holidays. This calculated DayType is sent to the front-end to ensure correct display and calculations there.
      - **Approver Role & hasApproverActed**: Determines the specific role (HoU, HoD, Manager, HR) of the *currently logged-in user* in relation to *this specific OT submission's approval flow*. It then checks if this particular approver has already made a decision (Approved or Rejected) on the submission (hasApproverActed). This flag is crucial for the front-end to disable edit/delete buttons.



```
[HttpGet("GetOtRecordsByStatusId/{statusId}")]
0 references
public IActionResult GetOtRecordsByStatusId(int statusId)
{
    var otStatus = _centralDbContext.Otstatus.FirstOrDefault(s => s.StatusId == stat
    if (otStatus == null)
        return NotFound("OT status not found.");

    var user = _centralDbContext.Users.FirstOrDefault(u => u.Id == otStatus.UserId);
    if (user == null)
        return NotFound("User not found.");

    var department = _centralDbContext.Departments
        .Where(d => d.DepartmentId == user.departmentId)
        .Select(d => d.DepartmentName)
        .FirstOrDefault();

    var userBasicSalary = _centralDbContext.Rates
        .Where(r => r.UserId == user.Id)
        .Select(r => r.RateValue)
        .FirstOrDefault();

    var userSetting = _centralDbContext.Hrusersetting
        .Include(us => us.State)
        .Include(us => us.Approvalflow)
            .ThenInclude(af => af.HeadOfUnit)
        .Include(us => us.FlexiHour)
        .FirstOrDefault(us => us.UserId == user.Id);

    if (userSetting?.State == null)
        return NotFound("User state information not found in HR User Settings.");

    if (userSetting?.Approvalflow == null)
        return NotFound("Approval flow information not found for the user.");

    var publicHolidaysForTable = _centralDbContext.Holidays
        .Where(h => h.StateId == userSetting.State.StateId)
        .Select(h => h.HolidayDate.Date.ToString("yyyy-MM-dd"))
        .ToList();

    var otRecords = _centralDbContext.Otregisters
        .Include(o => o.Stations)
```

➢ **[HttpGet("GetPublicHolidaysByState/{stateId}")]**

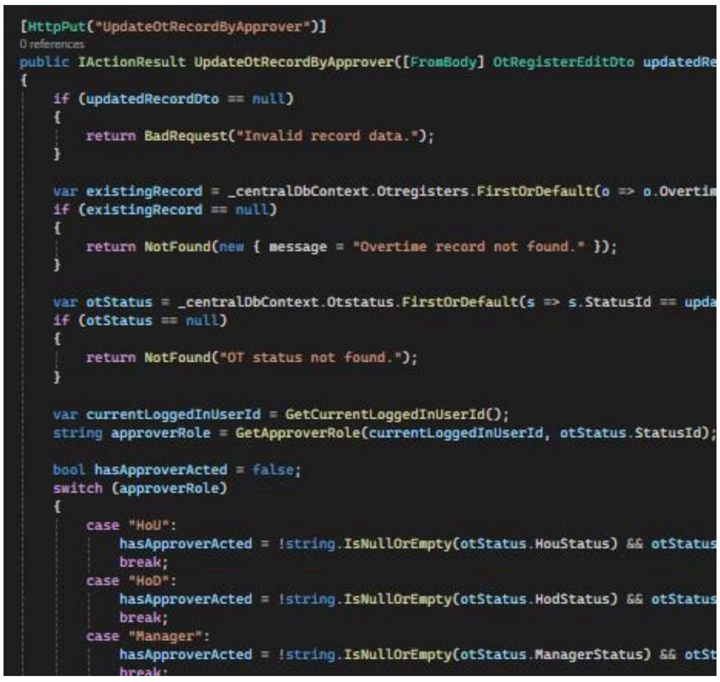
- **Purpose**: Provides a list of public holidays for a specific state. This is used by the front-end for accurate day type classification.



```
[HttpGet("GetPublicHolidaysByState/{stateId}")]
0 references
public IActionResult GetPublicHolidaysByState(int stateId)
{
    var holidays = _centralDbContext.Holidays
        .Where(h => h.StateId == stateId)
        .Select(h => new
        {
            date = h.HolidayDate.ToString("yyyy-MM-dd"),
            name = h.HolidayName
        })
        .ToList();

    return Ok(holidays);
}
```

➢ **[HttpPut("UpdateOtRecordByApprover")]**

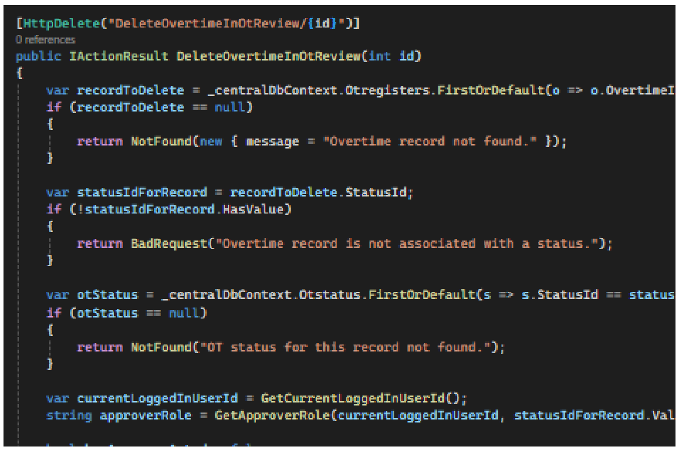
- **Purpose**: Allows an authorized approver to **modify an individual overtime record**.
- **Key Functionality**:
    - **Authorization Check**: Similar to DeleteOvertimeInOtReview, it verifies that the currentLoggedInUserId is the correct approver at the current stage of the approval flow and that they have *not yet acted* on this specific OT submission (i.e., their status is "Pending" or null). This prevents approvers from changing records after they've already approved or rejected the overall submission.
    - **Data Update**: Updates the existingRecord in the database with the new values from updatedRecordDto.
    - **Audit Logging**: Crucially, it logs the changes (before and after edit) in JSON format and appends this log to the respective approver's HouUpdate, HodUpdate, ManagerUpdate, or HrUpdate field in the OtStatusModel. This creates a detailed audit trail of any modifications made by approvers.



```
[HttpPut("UpdateOtRecordByApprover")]
0 references
public IActionResult UpdateOtRecordByApprover([FromBody] OtRegisterEditDto updatedRe
{
    if (updatedRecordDto == null)
    {
        return BadRequest("Invalid record data.");
    }

    var existingRecord = _centralDbContext.Otregisters.FirstOrDefault(o => o.Overtim
    if (existingRecord == null)
    {
        return NotFound(new { message = "Overtime record not found." });
    }

    var otStatus = _centralDbContext.Otstatus.FirstOrDefault(s => s.StatusId == upda
    if (otStatus == null)
    {
        return NotFound("OT status not found.");
    }

    var currentLoggedInUserId = GetCurrentLoggedInUserId();
    string approverRole = GetApproverRole(currentLoggedInUserId, otStatus.StatusId);

    bool hasApproverActed = false;
    switch (approverRole)
    {
        case "HoU":
            hasApproverActed = !string.IsNullOrEmpty(otStatus.HouStatus) && otStatus
            break;
        case "HoD":
            hasApproverActed = !string.IsNullOrEmpty(otStatus.HodStatus) && otStatus
            break;
        case "Manager":
            hasApproverActed = !string.IsNullOrEmpty(otStatus.ManagerStatus) && otSt
            break;
```

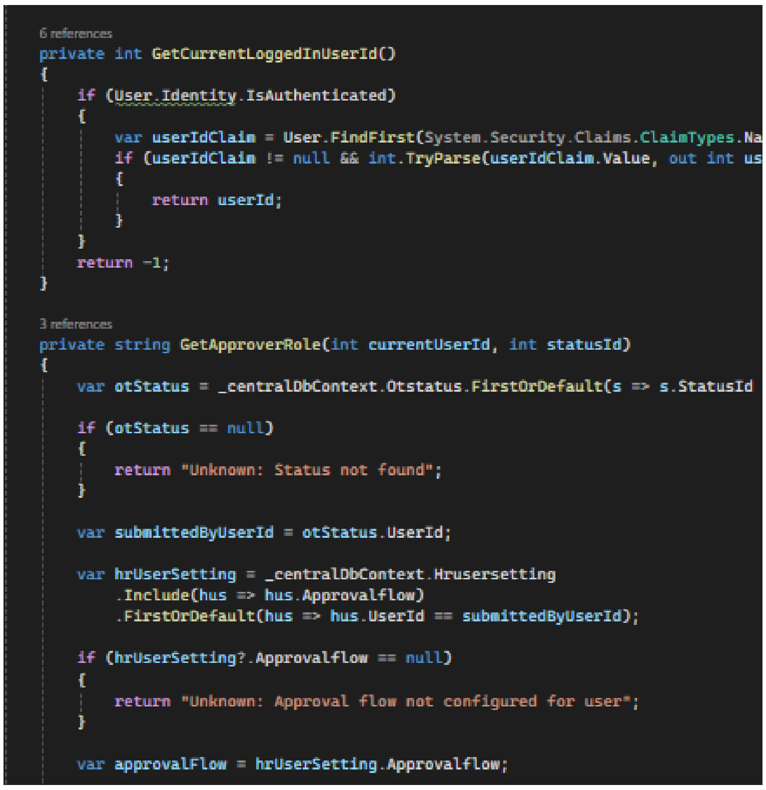
➢ **[HttpDelete("DeleteOvertimeInOtReview/{id}")]**

- **Purpose**: Allows an authorized approver to **delete an individual overtime record**.
- **Key Functionality**:
    - **Authorization Check**: Similar to the update, it first verifies that the currentLoggedInUserId is the correct approver and has *not yet acted* on the overall OT submission. This prevents deletion if a decision has already been made.
    - **Deletion**: Removes the OtRegister record from the database.
    - **Audit Logging**: Logs the deletion (the deleted record's data) and appends this log to the respective approver's HouUpdate, HodUpdate, ManagerUpdate, or HrUpdate field in the OtStatusModel. This ensures a record of what was deleted and by whom.



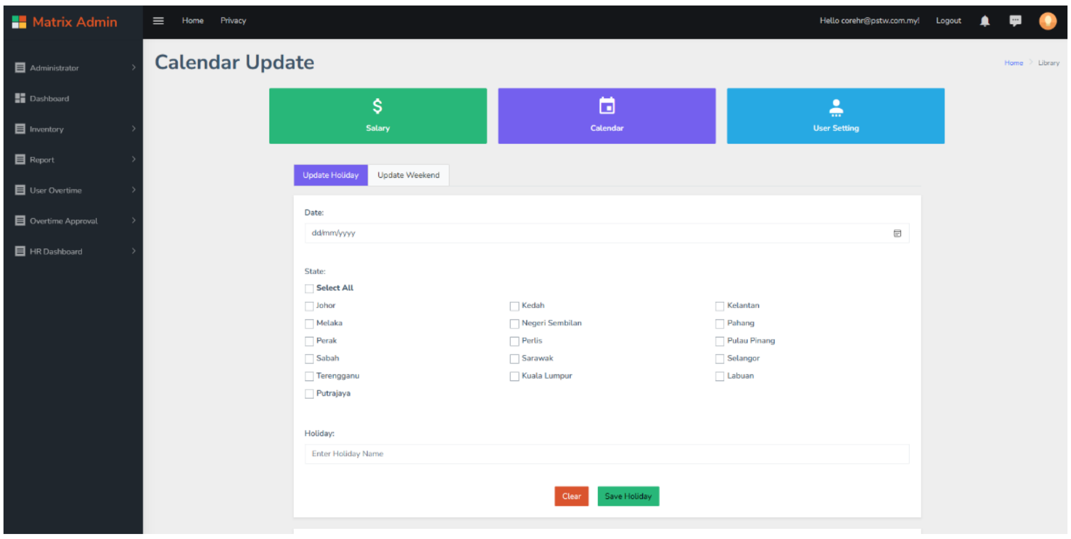
```
[HttpDelete("DeleteOvertimeInOtReview/{id}")]
0 references
public IActionResult DeleteOvertimeInOtReview(int id)
{
    var recordToDelete = _centralDbContext.Otregisters.FirstOrDefault(o => o.OvertimeI
    if (recordToDelete == null)
    {
        return NotFound(new { message = "Overtime record not found." });
    }

    var statusIdForRecord = recordToDelete.StatusId;
    if (!statusIdForRecord.HasValue)
    {
        return BadRequest("Overtime record is not associated with a status.");
    }

    var otStatus = _centralDbContext.Otstatus.FirstOrDefault(s => s.StatusId == status
    if (otStatus == null)
    {
        return NotFound("OT status for this record not found.");
    }

    var currentLoggedInUserId = GetCurrentLoggedInUserId();
    string approverRole = GetApproverRole(currentLoggedInUserId, statusIdForRecord.Val
```

- **Helper Methods (GetCurrentLoggedInUserId, GetApproverRole, AppendUpdateLog, ParseTimeStringToTimeSpan)**: These are internal helper methods used across the API to get the current user's ID, determine their specific approval role for a given OT submission (important for both access control and logging), manage the JSON-based update logs, and convert time strings.



```
6 references
private int GetCurrentLoggedInUserId()
{
    if (User.Identity.IsAuthenticated)
    {
        var userIdClaim = User.FindFirst(System.Security.Claims.ClaimTypes.Na
        if (userIdClaim != null && int.TryParse(userIdClaim.Value, out int us
        {
            return userId;
        }
    }
    return -1;
}

3 references
private string GetApproverRole(int currentUserId, int statusId)
{
    var otStatus = _centralDbContext.Otstatus.FirstOrDefault(s => s.StatusId

    if (otStatus == null)
    {
        return "Unknown: Status not found";
    }

    var submittedByUserId = otStatus.UserId;

    var hrUserSetting = _centralDbContext.Hrusersetting
        .Include(hus => hus.Approvalflow)
        .FirstOrDefault(hus => hus.UserId == submittedByUserId);

    if (hrUserSetting?.Approvalflow == null)
    {
        return "Unknown: Approval flow not configured for user";
    }

    var approvalFlow = hrUserSetting.Approvalflow;
```

- **HrDashboard**
  - **Calendar.cshtml**

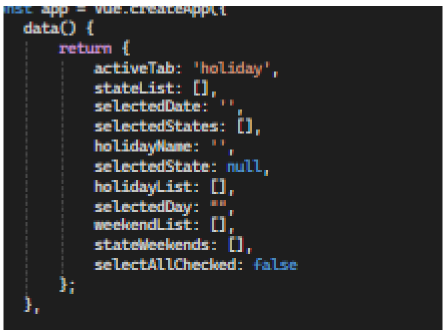


This file (Calendar.cshtml) provides a user interface for **HR personnel to manage holiday and weekend settings** within the system.

➢ **data() function**

- **Purpose**: This function defines all the **reactive data** (variables) that the Vue application needs to store and display. Changes to these variables automatically update the relevant parts of the web page.
- **Key Variables**:
  - **activeTab:** Controls which section is currently visible: 'holiday' (for updating public holidays) or 'weekend' (for updating weekend settings).
  - **stateList:** A list of all Malaysian states, fetched from the backend, used in dropdowns and checkboxes.
  - **selectedDate, selectedStates, holidayName**: Variables for the "Update Holiday" form. selectedStates holds the IDs of states for which a holiday is being set.
  - **selectedState, holidayList:** Used to display and filter existing holidays for a selected state.
  - **selectedDay:** Stores the selected weekend day (e.g., "Saturday & Sunday") for states.
  - **dayList**: A list of available weekend day options (e.g., "Friday & Saturday", "Saturday & Sunday").
  - **stateWeekends:** A list showing which weekend day is assigned to which state.

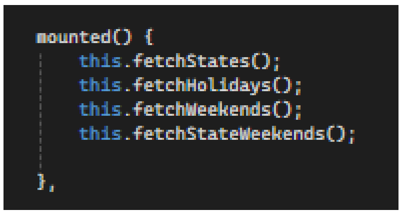
- **selectAllChecked:** A boolean that controls the "Select All" checkbox for states.



```
nst app = Vue.createApp({
  data() {
    return {
      activeTab: 'holiday',
      stateList: [],
      selectedDate: '',
      selectedStates: [],
      holidayName: '',
      selectedState: null,
      holidayList: [],
      selectedDay: "",
      weekendList: [],
      stateWeekends: [],
      selectAllChecked: false
    };
  },
```

➢ **mounted()**

- **Purpose**: This is a lifecycle hook that Vue.js calls automatically once the application has been "mounted" to the web page (meaning the HTML is ready). It's the ideal place to perform initial data loading.
- **Functionality**: When the page loads, it immediately calls methods to fetch:
  - stateList (all states)
  - holidayList (all holidays)
  - weekendList (all possible weekend patterns)
  - stateWeekends (the current weekend configuration for each state)

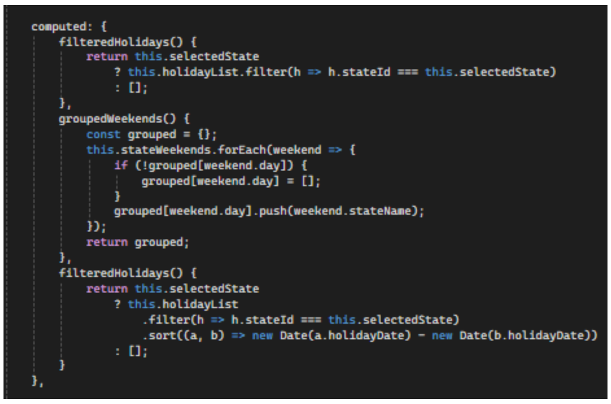


```
mounted() {
    this.fetchStates();
    this.fetchHolidays();
    this.fetchWeekends();
    this.fetchStateWeekends();

},
```

➢ **computed properties**

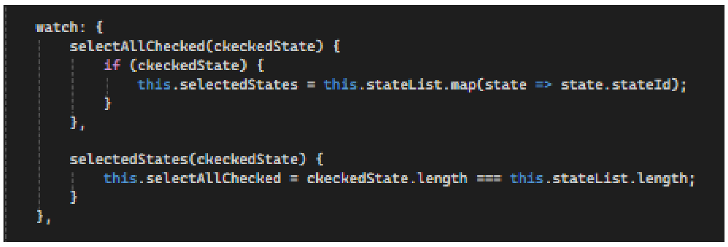
- **Purpose**: computed properties are dynamic values that Vue.js automatically recalculates whenever their underlying data dependencies change. They are efficient because they cache their results.
- **Key Computed Properties**:
  - **filteredHolidays:** Filters the holidayList to show only holidays for the selectedState and sorts them by date. This updates whenever selectedState changes.
  - **groupedWeekends:** Organizes the stateWeekends data into a more display-friendly format, grouping states by their assigned

weekend day (e.g., "Friday & Saturday: [Johor, Kedah]"). This makes the "Update Weekend" tab's display dynamic.



```
computed: {
    filteredHolidays() {
        return this.selectedState
            ? this.holidayList.filter(h => h.stateId === this.selectedState)
            : [];
    },
    groupedWeekends() {
        const grouped = {};
        this.stateWeekends.forEach(weekend => {
            if (!grouped[weekend.day]) {
                grouped[weekend.day] = [];
            }
            grouped[weekend.day].push(weekend.stateName);
        });
        return grouped;
    },
    filteredHolidays() {
        return this.selectedState
            ? this.holidayList
                .filter(h => h.stateId === this.selectedState)
                .sort((a, b) => new Date(a.holidayDate) - new Date(b.holidayDate))
            : [];
    }
},
```

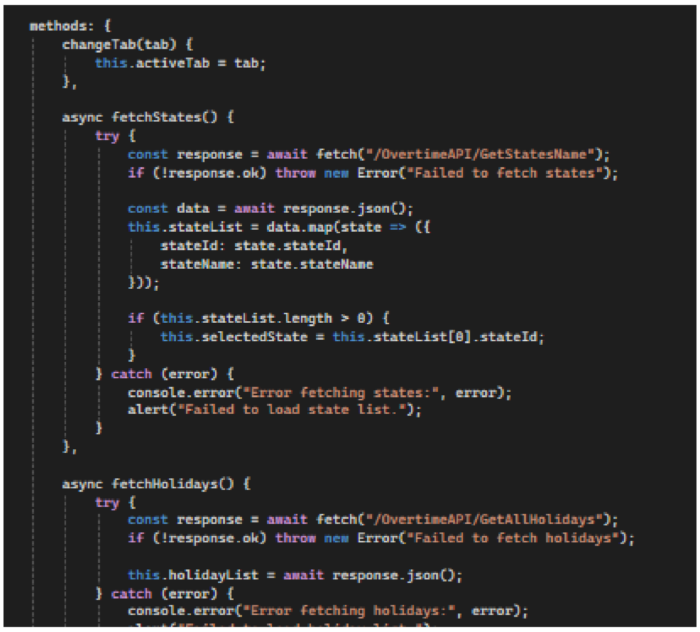
- **watch property**
  - **Purpose**: watch allows you to execute specific code when a particular data property changes. It's useful for more complex reactions than what computed properties handle.
  - **Functionality**:
    - **selectAllChecked:** When the "Select All" checkbox is toggled, this watcher updates the selectedStates array. If "Select All" is checked, selectedStates will include all state IDs; otherwise, it clears the array.
    - **selectedStates:** When the selectedStates array changes (e.g., individual checkboxes are clicked), this watcher updates the selectAllChecked checkbox to reflect whether all states are currently selected.



```
watch: {
    selectAllChecked(ckeckedState) {
        if (ckeckedState) {
            this.selectedStates = this.stateList.map(state => state.stateId);
        }
    },

    selectedStates(ckeckedState) {
        this.selectAllChecked = ckeckedState.length === this.stateList.length;
    }
},
```

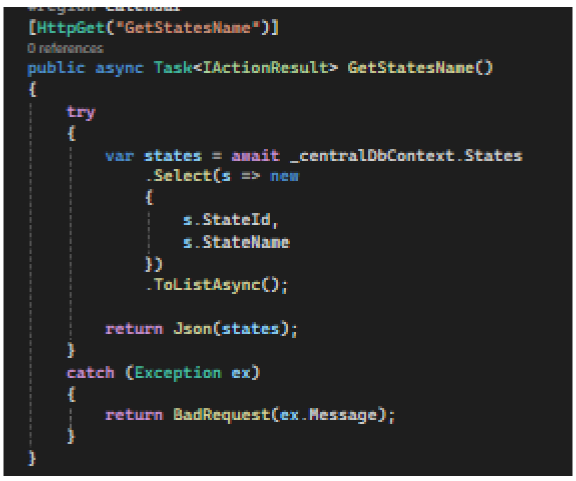
- **methods property**
  - **Purpose**: methods are functions that perform actions in response to user events (like button clicks) or other parts of the application.
  - **Key Methods**:
    - **changeTab(tab):** Updates the activeTab data property, switching between "Update Holiday" and "Update Weekend" sections.
    - fetchStates(): **Sends an HTTP GET request to /OvertimeAPI/GetStatesName** to retrieve a list of all Malaysian states and populates stateList.

- fetchHolidays(): **Sends an HTTP GET request to /OvertimeAPI/GetAllHolidays** to retrieve all existing public holidays and populates holidayList.
- **formatDate(date):** Formats a date string into a user-friendly format.
- updateHoliday(): **Sends an HTTP POST request to /OvertimeAPI/UpdateHoliday** with the selected date, states, and holiday name to add or update public holidays. It then re-fetches the holiday list to update the display.
- deleteHoliday(holidayId): **Sends an HTTP DELETE request to /OvertimeAPI/DeleteHoliday/{holidayId}** to remove a specific public holiday. It confirms with the user and then re-fetches the holiday list.
- fetchWeekends(): **Sends an HTTP GET request to /OvertimeAPI/GetWeekendDay** to retrieve predefined weekend day options (e.g., Friday & Saturday, Saturday & Sunday) and populates dayList.
- fetchStateWeekends(): **Sends an HTTP GET request to /OvertimeAPI/GetStateWeekends** to retrieve the current weekend settings for all states and populates stateWeekends.
- updateWeekend(): **Sends an HTTP POST request to /OvertimeAPI/UpdateWeekend** with selected states and a chosen weekend day to assign specific weekend patterns to states. It then re-fetches stateWeekends to update the display.
- **clearForm():** Resets the "Update Holiday" form fields.
- **clearWeekend():** Resets the "Update Weekend" form fields.



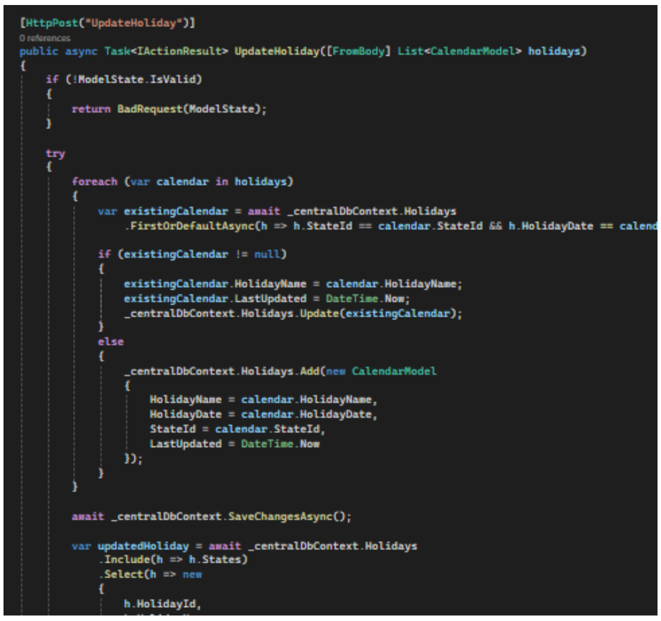
```
methods: {
    changeTab(tab) {
        this.activeTab = tab;
    },

    async fetchStates() {
        try {
            const response = await fetch("/OvertimeAPI/GetStatesName");
            if (!response.ok) throw new Error("Failed to fetch states");

            const data = await response.json();
            this.stateList = data.map(state => ({
                stateId: state.stateId,
                stateName: state.stateName
            }));

            if (this.stateList.length > 0) {
                this.selectedState = this.stateList[0].stateId;
            }
        } catch (error) {
            console.error("Error fetching states:", error);
            alert("Failed to load state list.");
        }
    },

    async fetchHolidays() {
        try {
            const response = await fetch("/OvertimeAPI/GetAllHolidays");
            if (!response.ok) throw new Error("Failed to fetch holidays");

            this.holidayList = await response.json();
        } catch (error) {
            console.error("Error fetching holidays:", error);
```

- **OvertimeAPI.cs (Calendar.cshtml)**
  - **[HttpGet("GetStatesName")]**
    - **Purpose**: Provides a list of all Malaysian states (ID and Name) to the front-end.
    - **Key Functionality**: Queries the database for all StateModel entries and returns their StateId and StateName.



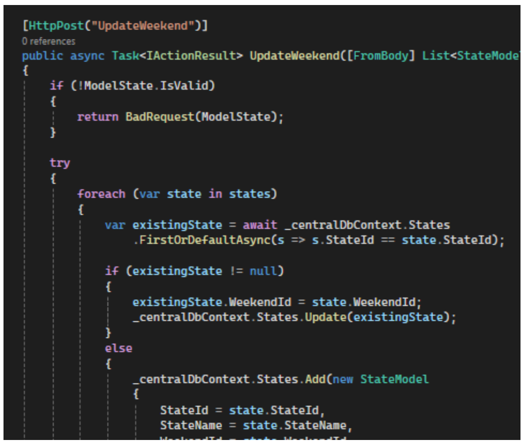
```
[HttpGet("GetStatesName")]
0 references
public async Task<IActionResult> GetStatesName()
{
    try
    {
        var states = await _centralDbContext.States
            .Select(s => new
            {
                s.StateId,
                s.StateName
            })
            .ToListAsync();

        return Json(states);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```

  - **[HttpPost("UpdateHoliday")]**
    - **Purpose**: Handles the creation or update of public holiday entries. This endpoint can receive a **list of holidays** to update multiple states at once for a given date and holiday name.
    - **Key Functionality**:
      - Receives a list of CalendarModel objects (each containing a HolidayDate, StateId, and HolidayName).
      - For each item in the list, it checks if a holiday already exists for that specific StateId and HolidayDate.
      - If it exists, the HolidayName and LastUpdated timestamp are updated.
      - If it doesn't exist, a new CalendarModel entry is added to the database.
      - Saves all changes to the database.



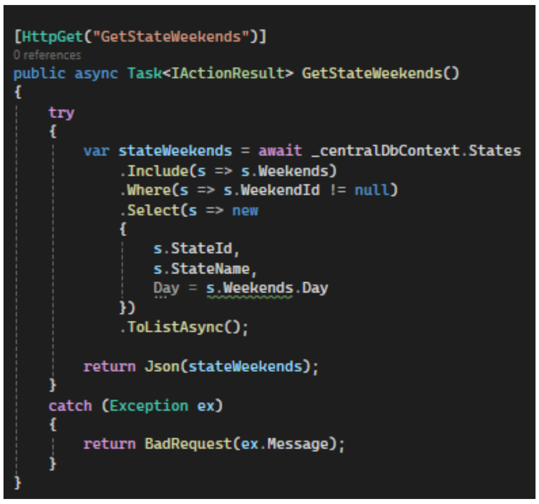
```csharp
[HttpPost("UpdateHoliday")]
0 references
public async Task<IActionResult> UpdateHoliday([FromBody] List<CalendarModel> holidays)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    try
    {
        foreach (var calendar in holidays)
        {
            var existingCalendar = await _centralDbContext.Holidays
                .FirstOrDefaultAsync(h => h.StateId == calendar.StateId && h.HolidayDate == calend

            if (existingCalendar != null)
            {
                existingCalendar.HolidayName = calendar.HolidayName;
                existingCalendar.LastUpdated = DateTime.Now;
                _centralDbContext.Holidays.Update(existingCalendar);
            }
            else
            {
                _centralDbContext.Holidays.Add(new CalendarModel
                {
                    HolidayName = calendar.HolidayName,
                    HolidayDate = calendar.HolidayDate,
                    StateId = calendar.StateId,
                    LastUpdated = DateTime.Now
                });
            }
        }

        await _centralDbContext.SaveChangesAsync();

        var updatedHoliday = await _centralDbContext.Holidays
            .Include(h => h.States)
            .Select(h => new
            {
                h.HolidayId,
```

- **[HttpGet("GetAllHolidays")]**
  - **Purpose**: Retrieves all existing holiday entries from the database.
- **[HttpDelete("DeleteHoliday/{id}")]**
  - **Purpose**: Deletes a specific holiday entry by its HolidayId.
- **[HttpGet("GetWeekendDay")]**
  - **Purpose**: Provides a list of predefined weekend day patterns (e.g., "Friday & Saturday", "Saturday & Sunday") to the front-end dropdown.
- **[HttpPost("UpdateWeekend")]**
  - **Purpose**: Updates the assigned weekend pattern for one or more states.
  - **Key Functionality**:
    - Receives a list of StateModel objects (each with StateId and WeekendId).
    - For each state in the list, it finds the existing state record in the database.
    - It then updates that state's WeekendId to the newly selected value.
    - Saves all changes to the database.



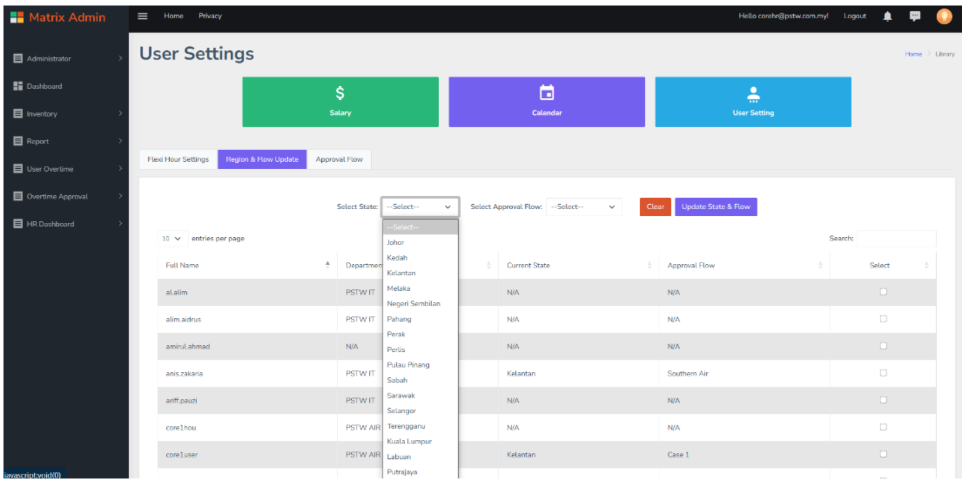
```
[HttpPost("UpdateWeekend")]
0 references
public async Task<IActionResult> UpdateWeekend([FromBody] List<StateMode
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    try
    {
        foreach (var state in states)
        {
            var existingState = await _centralDbContext.States
                .FirstOrDefaultAsync(s => s.StateId == state.StateId);

            if (existingState != null)
            {
                existingState.WeekendId = state.WeekendId;
                _centralDbContext.States.Update(existingState);
            }
            else
            {
                _centralDbContext.States.Add(new StateModel
                {
                    StateId = state.StateId,
                    StateName = state.StateName,
```

- **[HttpGet("GetStateWeekends")]**
    - **Purpose**: Retrieves the current weekend settings for all states that have a weekend defined, showing which day pattern is assigned to each state.



```
[HttpGet("GetStateWeekends")]
0 references
public async Task<IActionResult> GetStateWeekends()
{
    try
    {
        var stateWeekends = await _centralDbContext.States
            .Include(s => s.Weekends)
            .Where(s => s.WeekendId != null)
            .Select(s => new
            {
                s.StateId,
                s.StateName,
                Day = s.Weekends.Day
            })
            .ToListAsync();

        return Json(stateWeekends);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```

○ **HrUserSetting.cshtml**



Select State: --Select--

--Select--
Johor
Kedah
Kelantan
Melaka
Negeri Sembilan
Pahang
Perak
Perlis
Pulau Pinang
Sabah
Sarawak
Selangor
Terengganu
Kuala Lumpur
Labuan
Putrajaya

Select Approval Flow: --Select--

10 entries per page

Search:

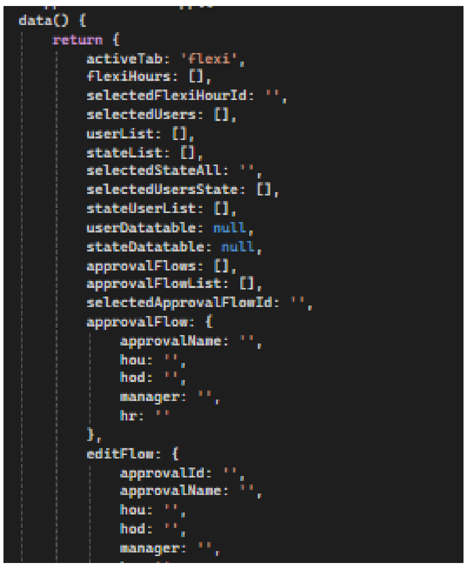
Full Name	Departmen	Current State	Approval Flow	Select
al.alim	PSTW IT	N/A	N/A	☐
alim.aidrus	PSTW IT	N/A	N/A	☐
amirul.ahmad	N/A	N/A	N/A	☐
anis.zakaria	PSTW IT	Kelantan	Southern Air	☐
ariff.pauzi	PSTW IT	N/A	N/A	☐
core1hou	PSTW AIR	N/A	N/A	☐
core1user	PSTW AIR	Kelantan	Case 1	☐

This file (HrUserSetting.cshtml) provides a user interface for **HR administrators** to manage various employee settings related to overtime calculations. This includes assigning **flexi-hours**, associating employees with a **state/region**, and assigning **approval flows**. It uses **Vue.js** to create an interactive experience with different tabs for each type of setting.

➢ **data() function**

- **Purpose**: This function defines all the **reactive state variables** for the Vue application. These variables hold the data displayed on the page and are automatically watched by Vue for changes, triggering UI updates.
- **Key Variables**:
  - **activeTab:** Controls which of the three main sections ("Flexi Hour Settings," "Region & Flow Update," "Approval Flow") is currently active.
  - **flexiHours, selectedFlexiHourId, selectedUsers:** Used for managing flexi-hour assignments. selectedUsers is an array of user IDs selected in the DataTables.
  - **userList, stateUserList:** Lists of users with their current flexi-hour/state/approval flow settings, fetched for display in DataTables.
  - **stateList, selectedStateAll, selectedUsersState:** Used for assigning states to users. selectedUsersState is an array of user IDs selected in the DataTables.
  - **approvalFlowList, selectedApprovalFlowId:** For assigning existing approval flows to users.
  - **approvalFlow:** An object representing the data for *creating* a new approval flow (name, HoU, HoD, Manager, HR).

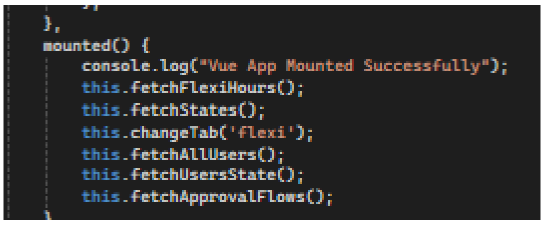
- **editFlow:** A temporary object used to hold the data of an approval flow when it's being *edited* in the modal form.
- **allUsers:** A list of all users, used in dropdowns for selecting approvers in an approval flow.
- **userDatatable, stateDatatable:** Variables to hold the DataTables instances for the user tables.



```
data() {
    return {
        activeTab: 'flexi',
        flexiHours: [],
        selectedFlexiHourId: '',
        selectedUsers: [],
        userList: [],
        stateList: [],
        selectedStateAll: '',
        selectedUsersState: [],
        stateUserList: [],
        userDatatable: null,
        stateDatatable: null,
        approvalFlows: [],
        approvalFlowList: [],
        selectedApprovalFlowId: '',
        approvalFlow: {
            approvalName: '',
            hou: '',
            hod: '',
            manager: '',
            hr: ''
        },
        editFlow: {
            approvalId: '',
            approvalName: '',
            hou: '',
            hod: '',
            manager: '',
```

➢ **mounted()**

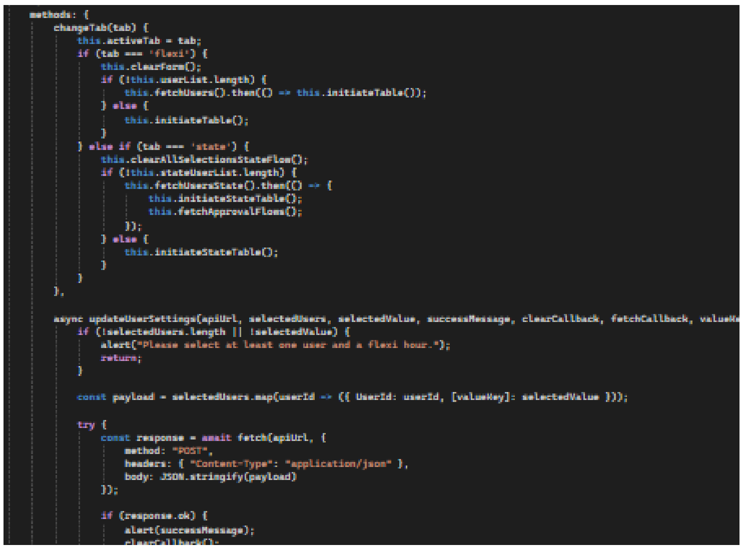
- **Purpose**: This lifecycle hook is called when the Vue application is fully loaded and attached to the HTML. It's the primary place for initial setup and data fetching.
- **Functionality**:
  - Logs a message to the console.
  - Immediately calls methods to fetch initial data: flexiHours, states, allUsers, usersState (users with state/flow info), and approvalFlows.
  - Sets the initial active tab to 'flexi'.
  - **Crucially, it calls initiateTable() (and later initiateStateTable()) after fetching the data.** This is because DataTables needs the data to be present in the Vue data before it can render the table.



```
},
mounted() {
    console.log("Vue App Mounted Successfully");
    this.fetchFlexiHours();
    this.fetchStates();
    this.changeTab('flexi');
    this.fetchAllUsers();
    this.fetchUsersState();
    this.fetchApprovalFlows();
```

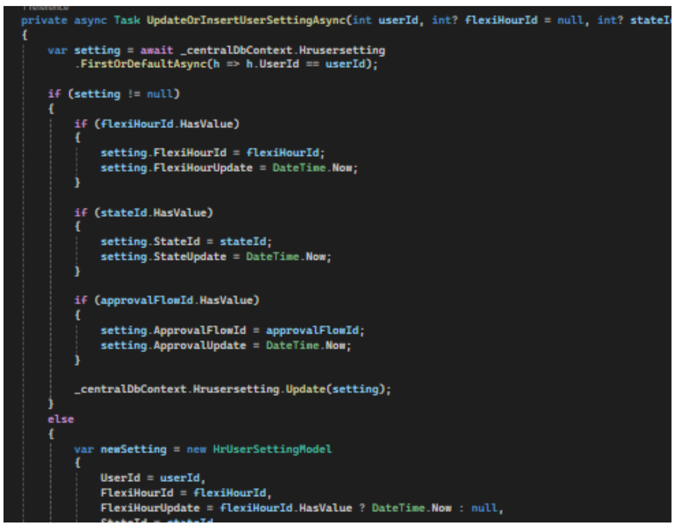
- **methods property**
  - **Purpose**: This section contains all the **functions that perform actions** in response to user interactions (like button clicks, tab changes) or programmatically.
  - **Key Methods**:
    - changeTab(tab): Updates the activeTab and conditionally calls initiateTable() or initiateStateTable() to load and render the correct DataTable for the selected tab.
    - updateUserSettings(...): A **generic helper method** used to send updates (for flexi-hours, states, or approval flows) to the backend API. It takes the API URL, selected user IDs, the value to update, success message, and callbacks for clearing forms and refreshing data. This promotes code reuse.
    - fetchFlexiHours(): **Fetches available flexi-hour options from /OvertimeAPI/GetFlexiHours**.
    - fetchUsers(): **Fetches the list of users with their current flexi-hour settings from /OvertimeAPI/GetUserFlexiHours** and prepares the data for the DataTable.
    - fetchStates(): **Fetches all state names from /OvertimeAPI/GetStatesName**.
    - fetchUsersState(): **Fetches the list of users with their current state and approval flow settings from /OvertimeAPI/GetUserStates** for the "Region & Flow Update" table.
    - initiateTable(), initiateStateTable(): These methods **initialize and populate the jQuery DataTables**. They handle setting up columns, rendering checkboxes, and attaching event listeners to the checkboxes within the table rows. This is where your raw data from userList or stateUserList gets displayed in an interactive grid.
    - handleCombinedUpdate(): Validates inputs and then calls updateUserSettings() to **update user states and/or approval flows** based on user selections in the "Region & Flow Update" tab.
    - updateFlexiHours(): Calls updateUserSettings() to **update flexi-hour settings** for selected users.
    - fetchAllUsers(): **Fetches a comprehensive list of all users from /OvertimeAPI/GetAllUsers**. This list is used for populating dropdowns where specific users need to be selected as approvers in an approval flow.

- fetchApprovalFlows(): **Fetches all *defined* approval flow templates from /OvertimeAPI/GetApprovalFlowList** to display them and allow selection/editing.
- submitApprovalFlow(): **Sends a POST request to /OvertimeAPI/CreateApprovalFlow** to create a new approval flow definition.
- deleteApprovalFlow(approvalId): **Sends a DELETE request to /OvertimeAPI/DeleteApprovalFlow/{id}** to remove an approval flow definition. It includes a confirmation and checks for dependencies (if the flow is currently assigned to users).
- clearForm(), clearAllSelectionsStateFlow(), clearFormApproval(): Reset various form fields and selected users/checkboxes in their respective tabs.
- openEditModal(flow): Populates the "Edit Approval Flow" modal with the data of the selected flow and then displays the modal.
- submitEditApprovalFlow(): **Sends a PUT request to /OvertimeAPI/EditApprovalFlow** to save changes made to an existing approval flow definition.



```
methods: {
    changeTab(tab) {
        this.activeTab = tab;
        if (tab === 'flexi') {
            this.clearForm();
            if (!this.userList.length) {
                this.fetchUsers().then(() => this.initiateTable());
            } else {
                this.initiateTable();
            }
        } else if (tab === 'state') {
            this.clearAllSelectionsStateFlow();
            if (!this.stateUserList.length) {
                this.fetchUsersState().then(() => {
                    this.initiateStateTable();
                    this.fetchApprovalFlows();
                });
            } else {
                this.initiateStateTable();
            }
        }
    },

    async updateUserSettings(apiUrl, selectedUsers, selectedValue, successMessage, clearCallback, fetchCallback, valueKe
        if (!selectedUsers.length || !selectedValue) {
            alert("Please select at least one user and a flexi hour.");
            return;
        }

        const payload = selectedUsers.map(userId => ({ UserId: userId, [valueKey]: selectedValue }));

        try {
            const response = await fetch(apiUrl, {
                method: "POST",
                headers: { "Content-Type": "application/json" },
                body: JSON.stringify(payload)
            });

            if (response.ok) {
                alert(successMessage);
```

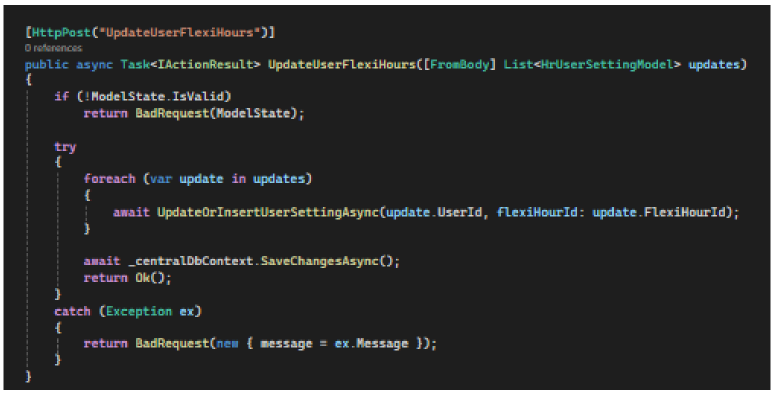
- **OvertimeAPI.cs (HrUserSetting.cshtml)**
  - **private async Task UpdateOrInsertUserSettingAsync(...)**
    - **Purpose**: This is a **private helper method** within the API controller. Its job is to efficiently update an existing HrUserSettingModel for a user or create a new one if it doesn't exist. This prevents duplicate code in other API methods.
    - **Key Functionality**: Takes a userId and optional flexiHourId, stateId, or approvalFlowId. It finds the HrUserSetting for that user. If found, it updates the provided fields and their corresponding Update timestamps. If not found, it creates a new HrUserSetting record.



```
private async Task UpdateOrInsertUserSettingAsync(int userId, int? flexiHourId = null, int? stateI
{
    var setting = await _centralDbContext.Hrusersetting
        .FirstOrDefaultAsync(h => h.UserId == userId);

    if (setting != null)
    {
        if (flexiHourId.HasValue)
        {
            setting.FlexiHourId = flexiHourId;
            setting.FlexiHourUpdate = DateTime.Now;
        }

        if (stateId.HasValue)
        {
            setting.StateId = stateId;
            setting.StateUpdate = DateTime.Now;
        }

        if (approvalFlowId.HasValue)
        {
            setting.ApprovalFlowId = approvalFlowId;
            setting.ApprovalUpdate = DateTime.Now;
        }

        _centralDbContext.Hrusersetting.Update(setting);
    }
    else
    {
        var newSetting = new HrUserSettingModel
        {
            UserId = userId,
            FlexiHourId = flexiHourId,
            FlexiHourUpdate = flexiHourId.HasValue ? DateTime.Now : null,
```

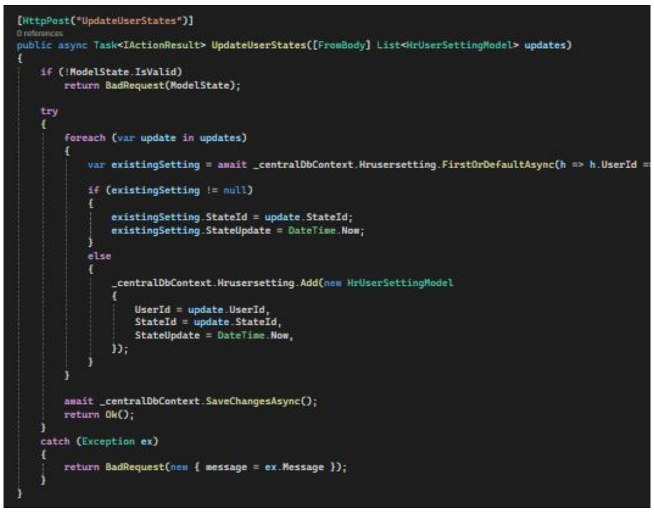
  - **[HttpGet("GetFlexiHours")]**
    - **Purpose**: Provides a list of available flexi-hour options (e.g., "9:00 - 6:00") to the front-end dropdown.
  - **[HttpGet("GetUserFlexiHours")]**
    - **Purpose**: Retrieves a list of all users, along with their current department and assigned flexi-hour. This data populates the table in the "Flexi Hour Settings" tab.
    - **Key Functionality**: Joins Users with Departments and HrUserSetting (including FlexiHour) to get a comprehensive view of each user's current flexi-hour assignment. It filters out "MAAdmin" and "SysAdmin" users.
  - **[HttpPost("UpdateUserFlexiHours")]**
    - **Purpose**: Handles updating the flexi-hour assignment for one or more selected users.
    - **Key Functionality**: Receives a list of HrUserSettingModel objects (each with UserId and FlexiHourId). It then uses the

UpdateOrInsertUserSettingAsync helper method to apply the new flexi-hour to each specified user.



```
[HttpPost("UpdateUserFlexiHours")]
0 references
public async Task<IActionResult> UpdateUserFlexiHours([FromBody] List<HrUserSettingModel> updates)
{
    if (!ModelState.IsValid)
        return BadRequest(ModelState);

    try
    {
        foreach (var update in updates)
        {
            await UpdateOrInsertUserSettingAsync(update.UserId, flexiHourId: update.FlexiHourId);
        }

        await _centralDbContext.SaveChangesAsync();
        return Ok();
    }
    catch (Exception ex)
    {
        return BadRequest(new { message = ex.Message });
    }
}
```

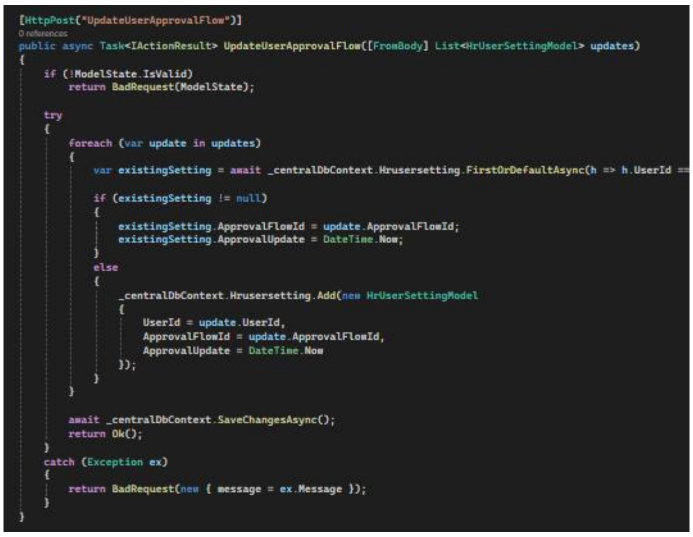
- **[HttpGet("GetUserStates")]**
  - **Purpose**: Retrieves a list of all users, along with their current department, assigned state, and assigned approval flow. This data populates the table in the "Region & Flow Update" tab.
  - **Key Functionality**: Joins Users with Departments, HrUserSetting (including State and Approvalflow) to provide a complete picture of each user's regional and approval flow assignments. It also filters out "MAAdmin" and "SysAdmin".
- **[HttpPost("UpdateUserStates")]**
  - **Purpose**: Handles updating the state/region assignment for one or more selected users.
  - **Key Functionality**: Receives a list of HrUserSettingModel objects (each with UserId and StateId). It uses the UpdateOrInsertUserSettingAsync helper to apply the new state to each user.



```
[HttpPost("UpdateUserStates")]
0 references
public async Task<IActionResult> UpdateUserStates([FromBody] List<HrUserSettingModel> updates)
{
    if (!ModelState.IsValid)
        return BadRequest(ModelState);

    try
    {
        foreach (var update in updates)
        {
            var existingSetting = await _centralDbContext.Hrusersetting.FirstOrDefaultAsync(h => h.UserId =

            if (existingSetting != null)
            {
                existingSetting.StateId = update.StateId;
                existingSetting.StateUpdate = DateTime.Now;
            }
            else
            {
                _centralDbContext.Hrusersetting.Add(new HrUserSettingModel
                {
                    UserId = update.UserId,
                    StateId = update.StateId,
                    StateUpdate = DateTime.Now,
                });
            }
        }

        await _centralDbContext.SaveChangesAsync();
        return Ok();
    }
    catch (Exception ex)
    {
        return BadRequest(new { message = ex.Message });
    }
}
```

➢ **[HttpPost("UpdateUserApprovalFlow")]**

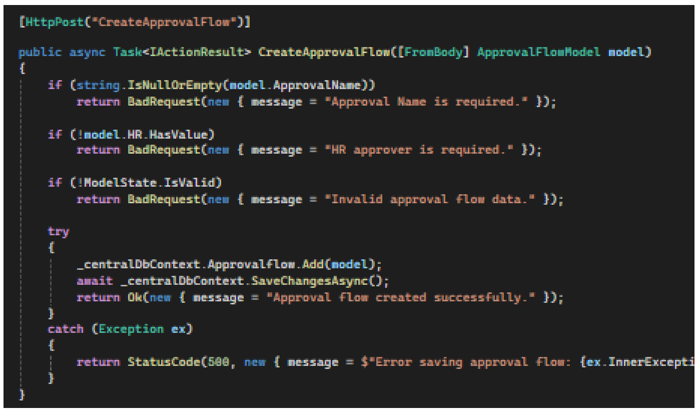
- **Purpose**: Handles updating the approval flow assignment for one or more selected users.
- **Key Functionality**: Receives a list of HrUserSettingModel objects (each with UserId and ApprovalFlowId). It uses the UpdateOrInsertUserSettingAsync helper to apply the new approval flow to each user.



```
[HttpPost("UpdateUserApprovalFlow")]
0 references
public async Task<IActionResult> UpdateUserApprovalFlow([FromBody] List<HrUserSettingModel> updates)
{
    if (!ModelState.IsValid)
        return BadRequest(ModelState);

    try
    {
        foreach (var update in updates)
        {
            var existingSetting = await _centralDbContext.Hrusersetting.FirstOrDefaultAsync(h => h.UserId ==

            if (existingSetting != null)
            {
                existingSetting.ApprovalFlowId = update.ApprovalFlowId;
                existingSetting.ApprovalUpdate = DateTime.Now;
            }
            else
            {
                _centralDbContext.Hrusersetting.Add(new HrUserSettingModel
                {
                    UserId = update.UserId,
                    ApprovalFlowId = update.ApprovalFlowId,
                    ApprovalUpdate = DateTime.Now
                });
            }
        }

        await _centralDbContext.SaveChangesAsync();
        return Ok();
    }
    catch (Exception ex)
    {
        return BadRequest(new { message = ex.Message });
    }
}
```

➢ **[HttpPost("CreateApprovalFlow")]**

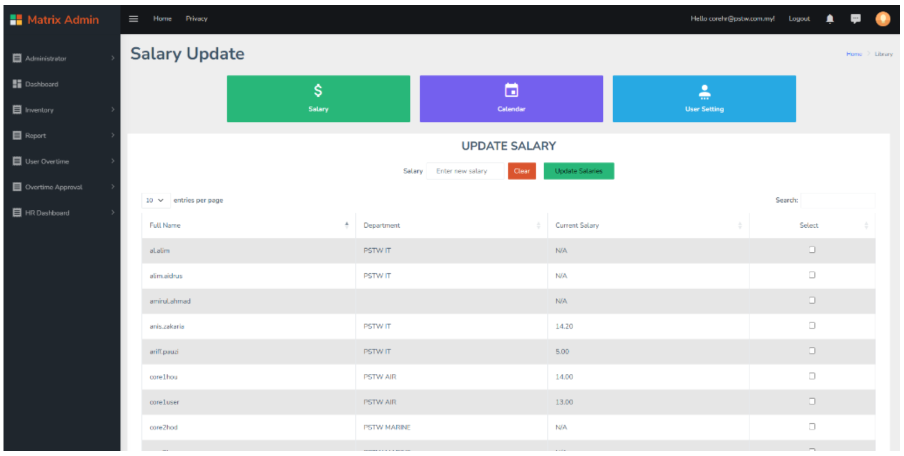
- **Purpose**: Allows HR to define a **new approval flow template**, specifying which user IDs serve as HoU, HoD, Manager, and HR for that flow.
- **Key Functionality**: Receives an ApprovalFlowModel (which includes ApprovalName and the IDs for each approver role). It adds this new flow definition to the database. It enforces that ApprovalName is present and HR approver is selected.



```
[HttpPost("CreateApprovalFlow")]

public async Task<IActionResult> CreateApprovalFlow([FromBody] ApprovalFlowModel model)
{
    if (string.IsNullOrEmpty(model.ApprovalName))
        return BadRequest(new { message = "Approval Name is required." });

    if (!model.HR.HasValue)
        return BadRequest(new { message = "HR approver is required." });

    if (!ModelState.IsValid)
        return BadRequest(new { message = "Invalid approval flow data." });

    try
    {
        _centralDbContext.Approvalflow.Add(model);
        await _centralDbContext.SaveChangesAsync();
        return Ok(new { message = "Approval flow created successfully." });
    }
    catch (Exception ex)
    {
        return StatusCode(500, new { message = $"Error saving approval flow: {ex.InnerExcepti
    }
}
```

- **[HttpGet("GetApprovalFlowList")]**
    - **Purpose**: Provides a list of all defined approval flow templates (their ID and Name) to the front-end, used in dropdowns.
- **[HttpPut("EditApprovalFlow")]**
    - **Purpose**: Allows HR to **modify an existing approval flow template**.
    - **Key Functionality**: Receives an ApprovalFlowModel with updated details for an existing ApprovalFlowId. It finds the existing flow and updates its properties.
- **[HttpDelete("DeleteApprovalFlow/{id}")]**
    - **Purpose**: Allows HR to **delete an approval flow template**.
    - **Key Functionality**: Finds the approval flow by its id. **Crucially, it checks if any users are currently assigned to this approval flow.** If users are still linked, it prevents deletion to maintain data integrity and returns a BadRequest error. Otherwise, it removes the flow from the database.
- **[HttpGet("GetAllUsers")]**
    - **Purpose**: Provides a simple list of all user IDs and Full Names. This is used specifically for populating dropdowns where individual users need to be selected (e.g., when defining approvers in an approval flow).

- **Rate.cshtml (Salary)**



Salary Update

UPDATE SALARY

Salary Enter new salary

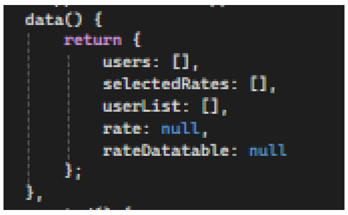
10 entries per page

Search:

Full Name	Department	Current Salary	Select
al.alim	PSTW IT	N/A	☐
alim.aidrus	PSTW IT	N/A	☐
amirul.ahmad		N/A	☐
anis.zakaria	PSTW IT	14.20	☐
ariff.pauzi	PSTW IT	5.00	☐
core1hou	PSTW AIR	14.00	☐
core1user	PSTW AIR	13.00	☐
core2hod	PSTW MARINE	N/A	☐

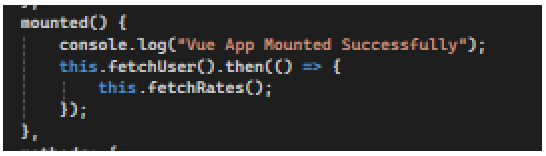
This file (Rate.cshtml) provides a user interface specifically for **HR administrators** to manage and update employee **salary**. It allows them to set a new salary value for one or more selected employees efficiently. The page uses **Vue.js** to make the table interactive and handle the salary update process.

- **data() function**
  - **Purpose**: This function defines the **initial state or data** for your Vue application. These variables hold the information that the page displays and interacts with. When these values change, Vue automatically updates the parts of the user interface that depend on them.
  - **Key Variables**:
    - users: This array is intended to hold a general list of users (though in this specific code, userList is primarily used after fetching).
    - selectedRates: An array that stores the **User IDs of the employees whose checkboxes are currently selected** in the table. This tells the application which users will have their salaries updated.
    - userList: This array holds the detailed information for each employee, including their fullName, departmentName, and importantly, their rateValue (current salary). This is the data displayed in the main table.
    - rate: This variable stores the **new salary value** entered by the user in the input field at the top of the form. It's initialized to null.
    - rateDatatable: This variable will hold the instance of the jQuery DataTables object, allowing Vue to control and interact with the table.



```
data() {
    return {
        users: [],
        selectedRates: [],
        userList: [],
        rate: null,
        rateDatatable: null
    };
},
```

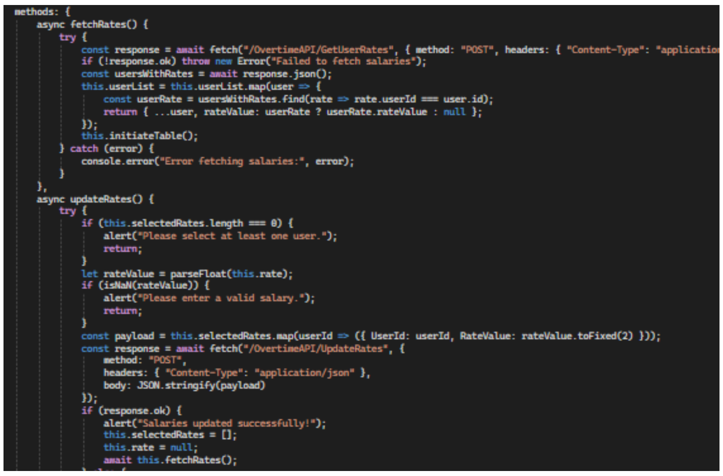
- **mounted()**
  - **Purpose**: This is a special lifecycle hook in Vue.js that runs automatically **after the Vue instance has been successfully attached to the HTML element** (#app in this case). It's the perfect place to perform initial setup tasks, like fetching data from an API.
  - **Functionality**:
    - It logs a message to the console to confirm successful mounting.
    - It calls this.fetchUser() first. Once fetchUser() completes, its .then() block calls this.fetchRates(). This ensures that the base user list is available before attempting to fetch and apply their current salary.



```
mounted() {
    console.log("Vue App Mounted Successfully");
    this.fetchUser().then(() => {
        this.fetchRates();
    });
},
```

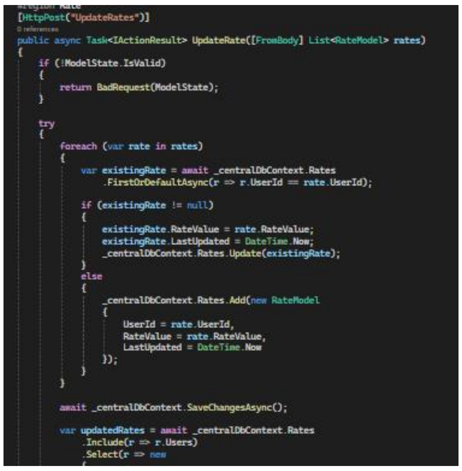
- **methods property**
  - **Purpose**: This section defines all the **functions that perform actions** within your Vue.js application. These methods are typically triggered by user interactions (e.g., clicking a button) or by other parts of the Vue application.
  - **Key Methods**:
    - fetchRates(): **Fetches the current salary for all users from the backend API** (/OvertimeAPI/GetUserRates). It then intelligently merges this salary data with the existing userList to update the rateValue for each user. Finally, it calls initiateTable() to render/refresh the interactive table with the updated salary information.
    - updateRates(): This method handles the **submission of the new salary updates**.
      - It first validates if any users are selected and if a valid salary amount is entered.
      - It then creates a payload (a list of objects, each containing a UserId and the RateValue to be applied).

        - It **sends an HTTP POST request to /OvertimeAPI/UpdateRates** with this payload.
        - Upon successful update, it displays an alert, clears the form, and calls fetchRates() again to refresh the displayed data.
    - fetchUser(): **Fetches a list of all general users (excluding "MAAdmin" and "SysAdmin") from an external API (/InvMainAPI/UserList/)**. This initial list provides basic user information (Full Name, Department) that fetchRates() then enriches with salary data.
    - initiateTable(): This method is responsible for **setting up and populating the jQuery DataTables plugin**. It takes the userList data and renders it into the interactive table, defining which data goes into which column and how the "Select" checkbox is rendered. It also sets up an event listener so that when a checkbox is clicked, the selectedRates array in Vue's data is updated.
    - clearForm(): Resets the input fields and deselects all checkboxes in the table, providing a clean slate for the next update.



```
methods: {
    async fetchRates() {
        try {
            const response = await fetch("/OvertimeAPI/GetUserRates", { method: "POST", headers: { "Content-Type": "applicatio
            if (!response.ok) throw new Error("Failed to fetch salaries");
            const usersWithRates = await response.json();
            this.userList = this.userList.map(user => {
                const userRate = usersWithRates.find(rate => rate.userId === user.id);
                return { ...user, rateValue: userRate ? userRate.rateValue : null };
            });
            this.initiateTable();
        } catch (error) {
            console.error("Error fetching salaries:", error);
        }
    },
    async updateRates() {
        try {
            if (this.selectedRates.length === 0) {
                alert("Please select at least one user.");
                return;
            }
            let rateValue = parseFloat(this.rate);
            if (isNaN(rateValue)) {
                alert("Please enter a valid salary.");
                return;
            }
            const payload = this.selectedRates.map(userId => ({ UserId: userId, RateValue: rateValue.toFixed(2) }));
            const response = await fetch("/OvertimeAPI/UpdateRates", {
                method: "POST",
                headers: { "Content-Type": "application/json" },
                body: JSON.stringify(payload)
            });
            if (response.ok) {
                alert("Salaries updated successfully!");
                this.selectedRates = [];
                this.rate = null;
                await this.fetchRates();
```

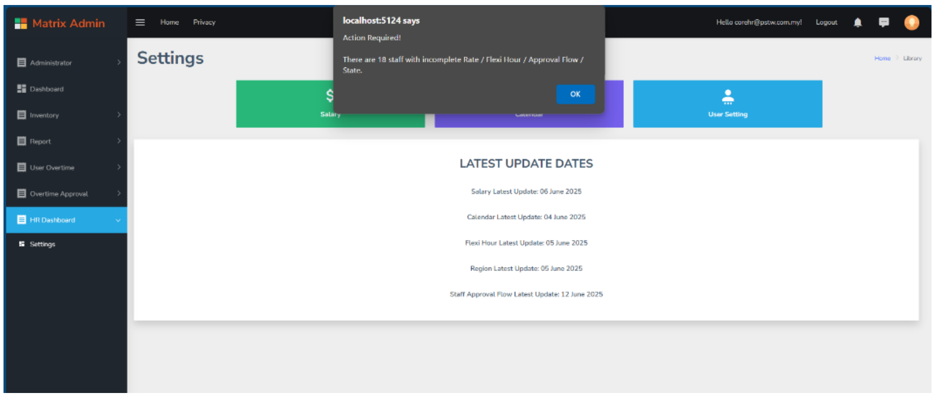
- **OvertimeAPI.cs (Rate.cshtml)**
  - **[HttpPost("UpdateRates")]**
    - **Purpose**: This API endpoint is responsible for **receiving new salary values from the front-end and updating them in the database**. It can handle updates for multiple users in a single request.
    - **Key Functionality**:
      - Receives a List<RateModel> from the client. Each RateModel in this list typically contains a UserId and the new RateValue.
      - For each RateModel in the list, it checks if a salary record already exists for that UserId.
      - If a record exists, it updates the RateValue and LastUpdated timestamp of the existing record.
      - If no record exists for that UserId, it creates a new RateModel entry in the database with the provided salary and the current timestamp.
      - Finally, it saves all changes to the database.
      - It then fetches the *updated* list of salary (including user names and departments) to send back to the front-end, allowing the UI to reflect the changes immediately.



```
[HttpPost("UpdateRates")]
0 references
public async Task<IActionResult> UpdateRate([FromBody] List<RateModel> rates)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    try
    {
        foreach (var rate in rates)
        {
            var existingRate = await _centralDbContext.Rates
                .FirstOrDefaultAsync(r => r.UserId == rate.UserId);

            if (existingRate != null)
            {
                existingRate.RateValue = rate.RateValue;
                existingRate.LastUpdated = DateTime.Now;
                _centralDbContext.Rates.Update(existingRate);
            }
            else
            {
                _centralDbContext.Rates.Add(new RateModel
                {
                    UserId = rate.UserId,
                    RateValue = rate.RateValue,
                    LastUpdated = DateTime.Now
                });
            }
        }

        await _centralDbContext.SaveChangesAsync();

        var updatedRates = await _centralDbContext.Rates
            .Include(r => r.Users)
            .Select(r => new
```

  - **[HttpPost("GetUserRates")]**
    - **Purpose**: This API endpoint provides the front-end with the **current salary rates for all users**.
    - **Key Functionality**: It queries the RateModel (salary) table, and importantly, it uses .Include() and .ThenInclude() to also fetch the related UserModel (employee details) and their Department information. This allows the front-end to display the employee's name and department alongside their salary.

- **Settings.cshtml**



Settings

localhost:5124 says

Action Required!

There are 18 staff with incomplete Rate / Flexi Hour / Approval Flow / State.

LATEST UPDATE DATES

Salary Latest Update: 06 June 2025

Calendar Latest Update: 04 June 2025

Flexi Hour Latest Update: 05 June 2025

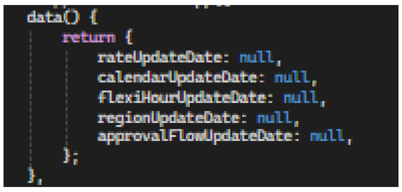
Region Latest Update: 05 June 2025

Staff Approval Flow Latest Update: 12 June 2025

This file (Settings.cshtml) serves as a **dashboard for HR administrators** to quickly see the latest update times for critical system configurations. It acts as an overview of when various HR-related data (like salaries, holidays, flexi hours, user regions, and approval flows) were last modified. It also provides a proactive alert if any staff members have incomplete settings.

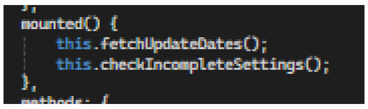
➢ **data() function**

- **Purpose**: This function defines all the **reactive state variables** that the Vue application needs to store and display. When these variables change, Vue automatically updates the relevant parts of the user interface.
- **Key Variables**:
  - rateUpdateDate: Stores the latest update timestamp for **salary** information.
  - calendarUpdateDate: Stores the latest update timestamp for **holiday** settings.
  - flexiHourUpdateDate: Stores the latest update timestamp for **flexi-hour** assignments.
  - regionUpdateDate: Stores the latest update timestamp for **user region/state** assignments.
  - approvalFlowUpdateDate: Stores the latest update timestamp for **user approval flow** assignments.
  - All these are initialized to null and will be populated by data fetched from the API.



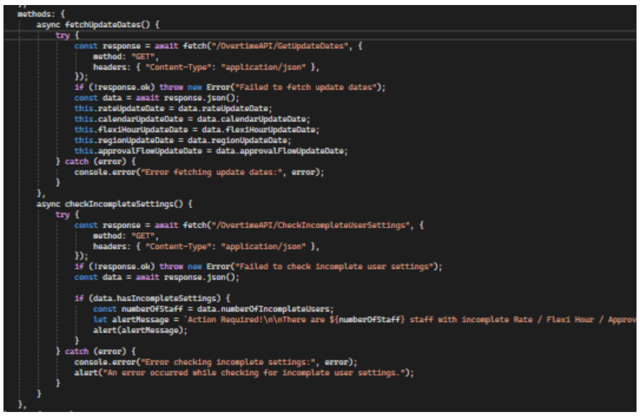
```
data() {
    return {
        rateUpdateDate: null,
        calendarUpdateDate: null,
        flexiHourUpdateDate: null,
        regionUpdateDate: null,
        approvalFlowUpdateDate: null,
    };
},
```

- **mounted()**
  - **Purpose**: This is a special lifecycle hook in Vue.js that automatically runs **after the Vue instance has been successfully attached to the HTML element**. It's the ideal place to perform initial setup tasks, like fetching data from an API.
  - **Functionality**: When the page loads:
    - It calls this.fetchUpdateDates() to retrieve the latest update timestamps for all settings categories.
    - It calls this.checkIncompleteSettings() to proactively check if any user settings are missing and alert the administrator.



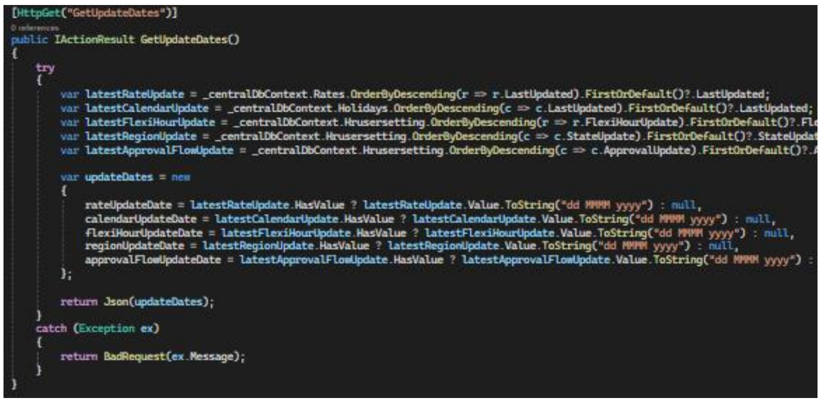
```
mounted() {
    this.fetchUpdateDates();
    this.checkIncompleteSettings();
},
```

- **methods property**
  - **Purpose**: This section defines the **functions that perform actions** within your Vue.js application, typically in response to the mounted hook or potential future user interactions (though this page is primarily for display).
  - **Key Methods**:
    - fetchUpdateDates(): **Sends an HTTP GET request to /OvertimeAPI/GetUpdateDates** to retrieve the latest update timestamps for various HR settings. Upon receiving the data, it updates the corresponding data properties, which in turn causes the display on the page to update.
    - checkIncompleteSettings(): **Sends an HTTP GET request to /OvertimeAPI/CheckIncompleteUserSettings** to determine if there are any users with missing salary, flexi-hour, region, or approval flow settings. If incomplete settings are found, it displays an alert message to the administrator, indicating how many staff members require attention.



```
methods: {
    async fetchUpdateDates() {
        try {
            const response = await fetch("/OvertimeAPI/GetUpdateDates", {
                method: "GET",
                headers: { "Content-Type": "application/json" },
            });
            if (!response.ok) throw new Error("Failed to fetch update dates");
            const data = await response.json();
            this.rateUpdateDate = data.rateUpdateDate;
            this.calendarUpdateDate = data.calendarUpdateDate;
            this.flexiHourUpdateDate = data.flexiHourUpdateDate;
            this.regionUpdateDate = data.regionUpdateDate;
            this.approvalFlowUpdateDate = data.approvalFlowUpdateDate;
        } catch (error) {
            console.error("Error fetching update dates:", error);
        }
    },
    async checkIncompleteSettings() {
        try {
            const response = await fetch("/OvertimeAPI/CheckIncompleteUserSettings", {
                method: "GET",
                headers: { "Content-Type": "application/json" },
            });
            if (!response.ok) throw new Error("Failed to check incomplete user settings");
            const data = await response.json();

            if (data.hasIncompleteSettings) {
                const numberOfStaff = data.numberOfIncompleteUsers;
                let alertMessage = `Action Required!\n\nThere are ${numberOfStaff} staff with incomplete Rate / Flexi Hour / Approv
                alert(alertMessage);
            }
        } catch (error) {
            console.error("Error checking incomplete settings:", error);
            alert("An error occurred while checking for incomplete user settings.");
        }
    }
},
```

- **OvertimeAPI.cs (Settings.cshtml)**
  - **[HttpGet("GetUpdateDates")]**
    - **Purpose**: This API endpoint is designed to **fetch the most recent "last updated" timestamps** for key data categories in the system, specifically: salary rates, public holidays (calendar), flexi-hour assignments, user state/region assignments, and user approval flow assignments.
    - **Key Functionality**:
      - It queries the _centralDbContext (your database context) for the Rates, Holidays, and Hrusersetting tables.
      - For each category, it finds the **single latest LastUpdated (or FlexiHourUpdate, StateUpdate, ApprovalUpdate) timestamp** by ordering records in descending order of their update time and taking the first one.
      - It formats these dates into a readable string (e.g., "15 June 2025") or sets them to null if no updates have occurred.

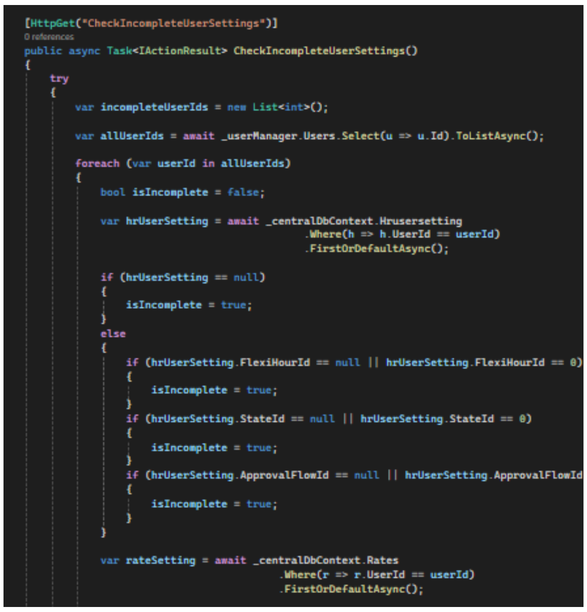


```
[HttpGet("GetUpdateDates")]
0 references
public IActionResult GetUpdateDates()
{
    try
    {
        var latestRateUpdate = _centralDbContext.Rates.OrderByDescending(r => r.LastUpdated).FirstOrDefault()?.LastUpdated;
        var latestCalendarUpdate = _centralDbContext.Holidays.OrderByDescending(c => c.LastUpdated).FirstOrDefault()?.LastUpdated;
        var latestFlexiHourUpdate = _centralDbContext.Hrusersetting.OrderByDescending(r => r.FlexiHourUpdate).FirstOrDefault()?.Fl
        var latestRegionUpdate = _centralDbContext.Hrusersetting.OrderByDescending(c => c.StateUpdate).FirstOrDefault()?.StateUpdat
        var latestApprovalFlowUpdate = _centralDbContext.Hrusersetting.OrderByDescending(c => c.ApprovalUpdate).FirstOrDefault()?.A

        var updateDates = new
        {
            rateUpdateDate = latestRateUpdate.HasValue ? latestRateUpdate.Value.ToString("dd MMMM yyyy") : null,
            calendarUpdateDate = latestCalendarUpdate.HasValue ? latestCalendarUpdate.Value.ToString("dd MMMM yyyy") : null,
            flexiHourUpdateDate = latestFlexiHourUpdate.HasValue ? latestFlexiHourUpdate.Value.ToString("dd MMMM yyyy") : null,
            regionUpdateDate = latestRegionUpdate.HasValue ? latestRegionUpdate.Value.ToString("dd MMMM yyyy") : null,
            approvalFlowUpdateDate = latestApprovalFlowUpdate.HasValue ? latestApprovalFlowUpdate.Value.ToString("dd MMMM yyyy") :
        };

        return Json(updateDates);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```

  - **[HttpGet("CheckIncompleteUserSettings")]**
    - **Purpose**: This API endpoint performs a critical check to identify if there are any **users in the system with incomplete HR settings** (missing salary, flexi-hour, state, or approval flow assignments).
    - **Key Functionality**:
      - It retrieves a list of all user IDs from the system.
      - For each user, it checks two main things:
        1. **HrUserSetting completeness**: Does the user have an HrUserSetting record? If not, it's incomplete. If they do, are their FlexiHourId, StateId, and ApprovalFlowId all set (not null or zero)?

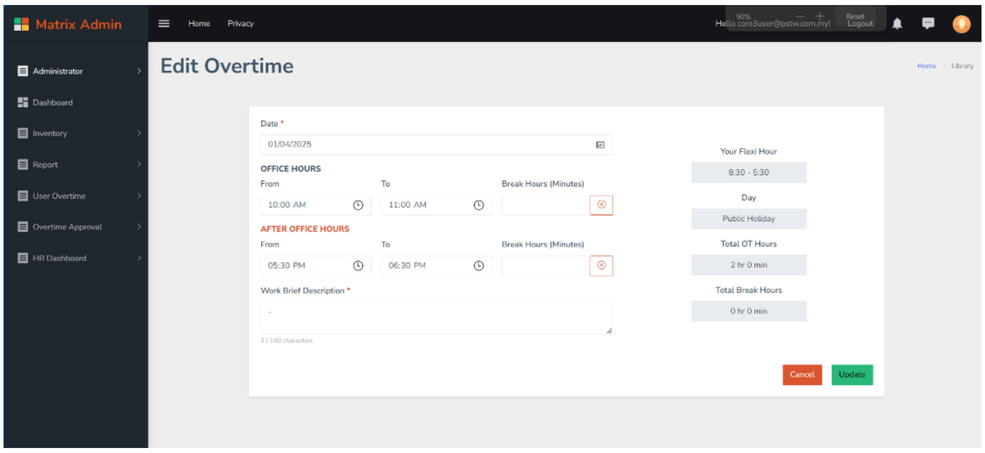
2. **Rate (salary) completeness**: Do they have a Rate (salary) record? If not, it's incomplete. If they do, is their RateValue greater than zero?

- If any of these conditions indicate incompleteness, the user's ID is added to a list of incompleteUserIds.
- It returns a JSON object indicating whether hasIncompleteSettings is true or false, and the numberOfIncompleteUsers.



```
[HttpGet("CheckIncompleteUserSettings")]
0 references
public async Task<IActionResult> CheckIncompleteUserSettings()
{
    try
    {
        var incompleteUserIds = new List<int>();

        var allUserIds = await _userManager.Users.Select(u => u.Id).ToListAsync();

        foreach (var userId in allUserIds)
        {
            bool isIncomplete = false;

            var hrUserSetting = await _centralDbContext.Hrusersetting
                                        .Where(h => h.UserId == userId)
                                        .FirstOrDefaultAsync();

            if (hrUserSetting == null)
            {
                isIncomplete = true;
            }
            else
            {
                if (hrUserSetting.FlexiHourId == null || hrUserSetting.FlexiHourId == 0)
                {
                    isIncomplete = true;
                }
                if (hrUserSetting.StateId == null || hrUserSetting.StateId == 0)
                {
                    isIncomplete = true;
                }
                if (hrUserSetting.ApprovalFlowId == null || hrUserSetting.ApprovalFlowId
                {
                    isIncomplete = true;
                }
            }

            var rateSetting = await _centralDbContext.Rates
                                    .Where(r => r.UserId == userId)
                                    .FirstOrDefaultAsync();
```

- **Overtime**
  - **EditOvertime.cshtml**



Edit Overtime

Date *
01/04/2025

OFFICE HOURS
From: 10:00 AM
To: 11:00 AM
Break Hours (Minutes):

AFTER OFFICE HOURS
From: 05:30 PM
To: 06:30 PM
Break Hours (Minutes):

Work Brief Description *
-
1 / 150 characters

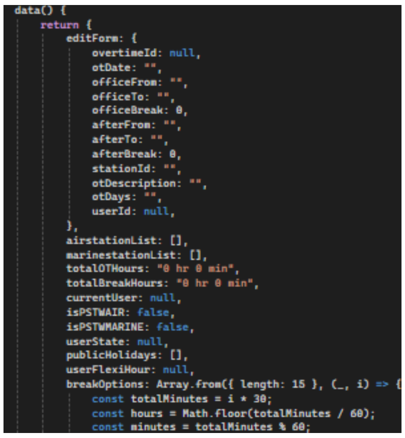
Your Flexi Hour: 8:30 - 5:30
Day: Public Holiday
Total OT Hours: 2 hr 0 min
Total Break Hours: 0 hr 0 min

This file (EditOvertime.cshtml) provides a dedicated web page for a user to **edit a single, existing overtime record**. It pre-fills a form with the current overtime details and allows the user to modify the date, times, breaks, station, and description. Use in OtRecords.

➢ **data() function**

- **Purpose**: This function defines all the reactive state variables that the Vue application needs to store and display. When these variables change, Vue automatically updates the relevant parts of the user interface.
- **Key Variables**:
  - editForm: This is the **main object that holds all the data for the overtime record being edited**. It's pre-populated from the API and then updated by user input.
  - airstationList, marinestationList: Lists of available stations for dropdowns, dynamically loaded based on the user's department.
  - totalOTHours, totalBreakHours: Computed display strings for the total calculated overtime and break hours.
  - currentUser: Object holding details about the currently logged-in user (department, roles).
  - isPSTWAIR, isPSTWMARINE: Boolean flags indicating if the user belongs to a department (or is an admin) that requires selecting Air or Marine stations.
  - userState, publicHolidays: Data about the user's assigned state and its public holidays, used for determining the OtDays (day type).
  - userFlexiHour: The user's assigned flexi-hour, used for display.

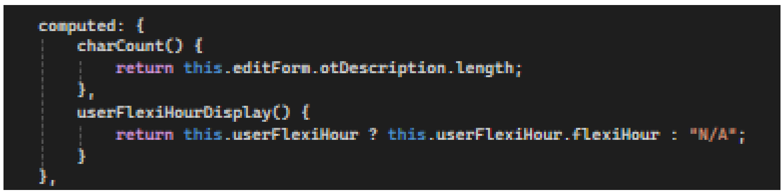
- breakOptions: Predefined options for break durations (e.g., "30 min", "1 hour").
- previousPage, returnMonth, returnYear: Used to navigate the user back to the correct previous page (e.g., OtRecords.cshtml) after editing.
- validationErrors: An object to store and display client-side validation messages for various form fields.



```
data() {
    return {
        editForm: {
            overtimeId: null,
            otDate: "",
            officeFrom: "",
            officeTo: "",
            officeBreak: 0,
            afterFrom: "",
            afterTo: "",
            afterBreak: 0,
            stationId: "",
            otDescription: "",
            otDays: "",
            userId: null,
        },
        airstationList: [],
        marinestationList: [],
        totalOTHours: "0 hr 0 min",
        totalBreakHours: "0 hr 0 min",
        currentUser: null,
        isPSTWAIR: false,
        isPSTWMARINE: false,
        userState: null,
        publicHolidays: [],
        userFlexiHour: null,
        breakOptions: Array.from({ length: 15 }, (_, i) => {
            const totalMinutes = i * 30;
            const hours = Math.floor(totalMinutes / 60);
            const minutes = totalMinutes % 60;
```

➢ **computed properties**

- **Purpose**: These properties dynamically calculate and return values based on other data properties. They are efficient because they are only re-calculated when their dependencies change.
- **Key Computed Properties**:
  - charCount: Returns the current character count of the otDescription field.
  - userFlexiHourDisplay: Formats and returns the user's flexi-hour for display, showing "N/A" if not available.



```
computed: {
    charCount() {
        return this.editForm.otDescription.length;
    },
    userFlexiHourDisplay() {
        return this.userFlexiHour ? this.userFlexiHour.flexiHour : "N/A";
    }
},
```

➢ **mounted()**

- **Purpose**: This lifecycle hook runs automatically **after the Vue instance is attached to the HTML element (#app)**. It's the primary place for initial setup and data fetching.
- **Functionality**:

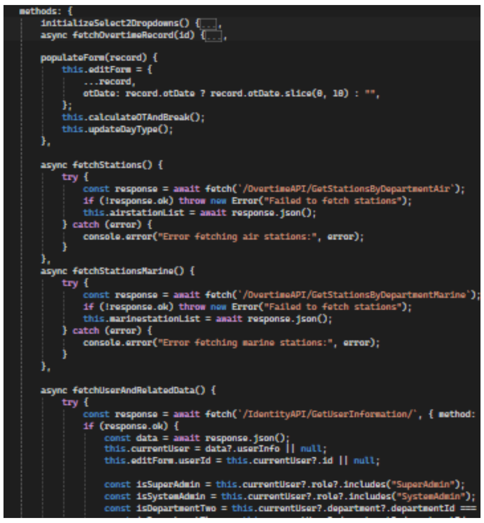
- Retrieves the overtimeId from the URL, which tells the page *which* specific overtime record to edit. It also captures month and year for smart navigation back.
- If an overtimeId is found, it calls fetchOvertimeRecord() to load the data for that record.
- It then calls fetchUserAndRelatedData() to get information about the current logged-in user, their department, state, and flexi-hour. This is important for conditional display (e.g., showing specific station dropdowns) and calculations.
- Based on the user's department, it conditionally fetches either Air or Marine station lists.
- Finally, it calls initializeSelect2Dropdowns() to set up the interactive station dropdowns, ensuring they display correctly and reflect the pre-loaded stationId.

➢ **methods property**

- **Purpose**: This section defines all the **functions that perform actions** in response to user input (like typing, clicking buttons, changing dropdowns) or other programmatic triggers.
- **Key Methods**:
  - initializeSelect2Dropdowns(): Sets up the jQuery Select2 plugin for the station dropdowns, making them searchable and more user-friendly. It also attaches change listeners to update Vue's editForm.stationId.
  - fetchOvertimeRecord(id): **Fetches the details of a specific overtime record from /OvertimeAPI/GetOvertimeRecordById/{id}**. After fetching, it calls populateForm() to load this data into the editForm.
  - populateForm(record): Takes the fetched overtime record data and assigns it to the editForm properties. It then triggers calculations and day type updates.
  - fetchStations(), fetchStationsMarine(): **Fetch lists of Air or Marine stations from /OvertimeAPI/GetStationsByDepartmentAir or GetStationsByDepartmentMarine** respectively, based on the user's department.
  - fetchUserAndRelatedData(): **Fetches the currently logged-in user's information (including department, roles) from /IdentityAPI/GetUserInformation/**. This determines isPSTWAIR and isPSTWMARINE flags, which control station dropdown

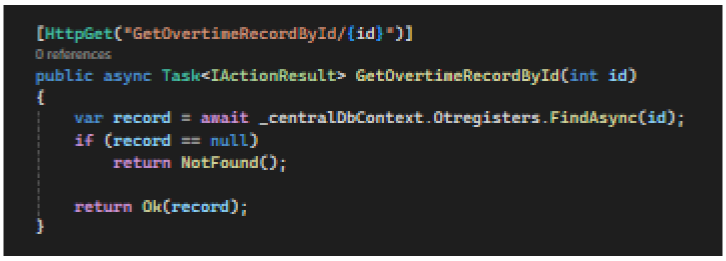
visibility. It also triggers fetchUserStateAndHolidays() and fetchUserFlexiHour().

- fetchUserStateAndHolidays(): **Fetches the user's assigned state and its public holidays from /OvertimeAPI/GetUserStateAndHolidays/{userId}**. This data is essential for the updateDayType() function.
- fetchUserFlexiHour(): **Fetches the user's assigned flexi-hour from /OvertimeAPI/GetUserFlexiHour/{userId}**.
- limitCharCount(event): Restricts the length of the otDescription text area to 150 characters.
- calculateOTAndBreak(): **Calculates the total overtime hours and total break hours** based on the times and breaks entered in the form. It updates totalOTHours and totalBreakHours for display and also calls updateDayType().
- updateTime(fieldName): Rounds the input time (From/To) to the nearest 30 minutes for consistency and then triggers time range validation and recalculation.
- validateTimeRanges(): Performs **client-side validation** on the "From" and "To" times for both "Office Hours" and "After Office Hours" to ensure valid time ranges (e.g., "To" is after "From," handling midnight crossings, and specific constraints for midnight "To" times). It shows alerts if validation fails.
- updateDayType(): Determines the type of day (Weekday, Weekend, Public Holiday) based on the selected otDate, the user's userState, and fetched publicHolidays.
- validateForm(): Performs **overall form validation** (required fields, time entries, station for specific departments). It consolidates validation errors and shows an alert with all issues.
- validateAndUpdate(): The button handler that first calls validateForm(). If the form is valid, it then calls updateRecord().
- updateRecord(): **Sends an HTTP POST request to /OvertimeAPI/UpdateOvertimeRecord** with the complete editForm data. If successful, it displays an alert and navigates the user back to the previous page.
- goBack(): Navigates the user back to the page they came from, or to a specific "OtRecords" page if a returnMonth and returnYear were available.
- clearOfficeHours(), clearAfterHours(): Helper methods to clear the input fields and reset breaks for either office or after-office hours.



```
methods: {
    initializeSelect2Dropdowns() {...},
    async fetchOvertimeRecord(id) {...},

    populateForm(record) {
        this.editForm = {
            ...record,
            otDate: record.otDate ? record.otDate.slice(0, 10) : "",
        };
        this.calculateOTAndBreak();
        this.updateDayType();
    },

    async fetchStations() {
        try {
            const response = await fetch(`/OvertimeAPI/GetStationsByDepartmentAir`);
            if (!response.ok) throw new Error("Failed to fetch stations");
            this.airstationList = await response.json();
        } catch (error) {
            console.error("Error fetching air stations:", error);
        }
    },
    async fetchStationsMarine() {
        try {
            const response = await fetch(`/OvertimeAPI/GetStationsByDepartmentMarine`);
            if (!response.ok) throw new Error("Failed to fetch stations");
            this.marinestationList = await response.json();
        } catch (error) {
            console.error("Error fetching marine stations:", error);
        }
    },

    async fetchUserAndRelatedData() {
        try {
            const response = await fetch(`/IdentityAPI/GetUserInformation/`, { method:
            if (response.ok) {
                const data = await response.json();
                this.currentUser = data?.userInfo || null;
                this.editForm.userId = this.currentUser?.id || null;

                const isSuperAdmin = this.currentUser?.role?.includes("SuperAdmin");
                const isSystemAdmin = this.currentUser?.role?.includes("SystemAdmin");
                const isDepartmentTwo = this.currentUser?.department?.departmentId ===
```

- **OvertimeAPI.cs (EditOvertime.cshtml)**
  - **[HttpGet("GetStationsByDepartmentAir")]** and **[HttpGet("GetStationsByDepartmentMarine")]**
    - **Purpose**: These endpoints provide lists of specific stations (Air or Marine) based on their associated DepartmentId. This allows the front-end to show relevant station options only when needed.
  - **[HttpGet("GetOvertimeRecordById/{id}")]**
    - **Purpose**: This is the primary endpoint to **fetch a single, existing overtime record** by its OvertimeId. This data is used to pre-fill the editing form on the front-end.
    - **Key Functionality**: Takes an id (the OvertimeId) and uses _centralDbContext.Otregisters.FindAsync(id) to quickly retrieve the record. If found, it returns the record; otherwise, it returns a "Not Found" response.

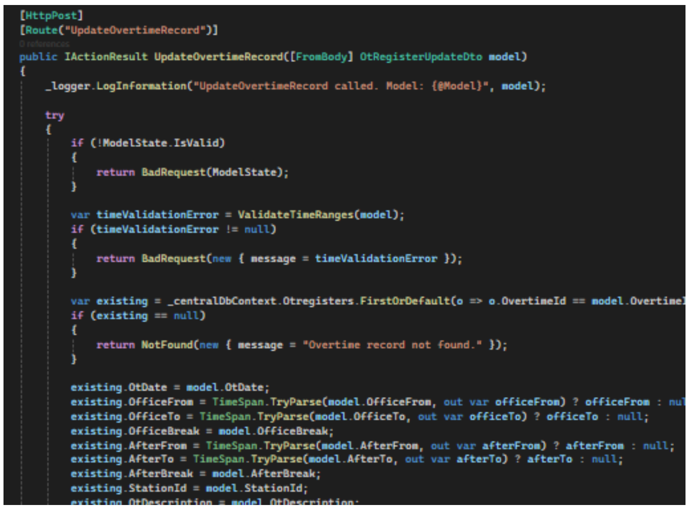


```
[HttpGet("GetOvertimeRecordById/{id}")]
0 references
public async Task<IActionResult> GetOvertimeRecordById(int id)
{
    var record = await _centralDbContext.Otregisters.FindAsync(id);
    if (record == null)
        return NotFound();

    return Ok(record);
}
```

  - **[HttpGet("GetUserFlexiHour/{userId}")]**
    - **Purpose**: Retrieves the flexi-hour assigned to a specific user. This is displayed on the edit form as reference.
    - **Key Functionality**: Queries Hrusersetting and includes FlexiHour to get the details. Handles cases where flexi-hour might not be found.

➢ **[HttpPost("UpdateOvertimeRecord")]**

- **Purpose**: This is the core endpoint for **receiving the updated overtime record data from the front-end and saving it to the database**.
- **Key Functionality**:
  - Receives an OtRegisterUpdateDto containing all the edited fields of an overtime record.
  - **Server-Side Validation (ValidateTimeRanges)**: Before saving, it performs comprehensive server-side validation on the time ranges (Office From/To, After From/To) to ensure data integrity and apply the same business rules as the client-side validation. This is a critical security measure as client-side validation can be bypassed.
  - If time validation passes, it finds the existing OtRegister record by OvertimeId.
  - It updates all the relevant fields of the existing record with the values from the model.
  - It handles converting the time strings (OfficeFrom, OfficeTo, etc.) back into TimeSpan objects for database storage.
  - It saves the changes using _centralDbContext.SaveChanges().
  - Returns a success message or detailed error messages if validation fails or the record isn't found.

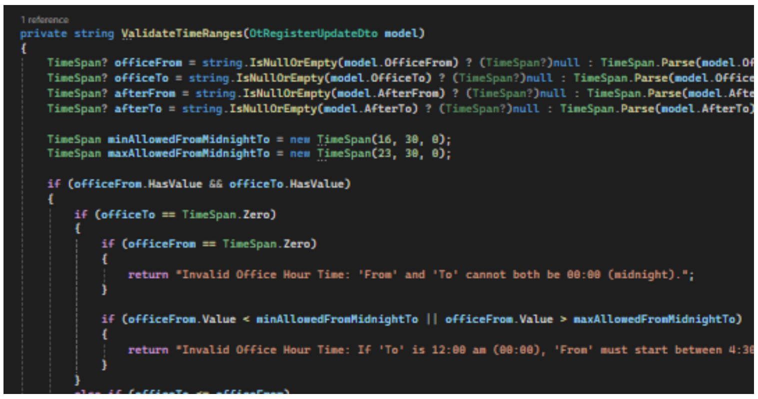


```
[HttpPost]
[Route("UpdateOvertimeRecord")]
0 references
public IActionResult UpdateOvertimeRecord([FromBody] OtRegisterUpdateDto model)
{
    _logger.LogInformation("UpdateOvertimeRecord called. Model: {@Model}", model);

    try
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }

        var timeValidationError = ValidateTimeRanges(model);
        if (timeValidationError != null)
        {
            return BadRequest(new { message = timeValidationError });
        }

        var existing = _centralDbContext.Otregisters.FirstOrDefault(o => o.OvertimeId == model.Overtime
        if (existing == null)
        {
            return NotFound(new { message = "Overtime record not found." });
        }

        existing.OtDate = model.OtDate;
        existing.OfficeFrom = TimeSpan.TryParse(model.OfficeFrom, out var officeFrom) ? officeFrom : nul
        existing.OfficeTo = TimeSpan.TryParse(model.OfficeTo, out var officeTo) ? officeTo : null;
        existing.OfficeBreak = model.OfficeBreak;
        existing.AfterFrom = TimeSpan.TryParse(model.AfterFrom, out var afterFrom) ? afterFrom : null;
        existing.AfterTo = TimeSpan.TryParse(model.AfterTo, out var afterTo) ? afterTo : null;
        existing.AfterBreak = model.AfterBreak;
        existing.StationId = model.StationId;
        existing.OtDescription = model.OtDescription;
```

➢ **private string ValidateTimeRanges(OtRegisterUpdateDto model)**

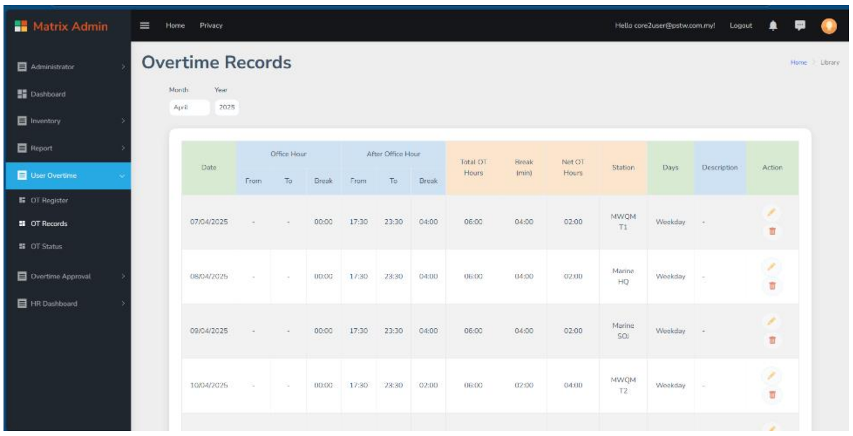
- **Purpose**: This is a **private helper method** within the API controller that centralizes the server-side validation logic for overtime time entries. It mirrors the client-side validation.

- **Key Functionality**:
    - Parses the string time inputs from the model into TimeSpan objects.
    - Applies a series of checks for both office hours and after-office hours:
        - Ensures "From" and "To" times are not both midnight (00:00).
        - For times spanning across midnight (e.g., 23:00 to 00:00), it has specific rules for the "From" time (must be within a certain late-evening range).
        - Ensures "To" time is later than "From" time for same-day durations.
        - Checks if at least one of "Office Hours" or "After Office Hours" has been provided.
    - Returns an error message string if validation fails, or null if all validations pass.



```
1 reference
private string ValidateTimeRanges(OtRegisterUpdateDto model)
{
    TimeSpan? officeFrom = string.IsNullOrEmpty(model.OfficeFrom) ? (TimeSpan?)null : TimeSpan.Parse(model.Of
    TimeSpan? officeTo = string.IsNullOrEmpty(model.OfficeTo) ? (TimeSpan?)null : TimeSpan.Parse(model.Office
    TimeSpan? afterFrom = string.IsNullOrEmpty(model.AfterFrom) ? (TimeSpan?)null : TimeSpan.Parse(model.Afte
    TimeSpan? afterTo = string.IsNullOrEmpty(model.AfterTo) ? (TimeSpan?)null : TimeSpan.Parse(model.AfterTo)

    TimeSpan minAllowedFromMidnightTo = new TimeSpan(16, 30, 0);
    TimeSpan maxAllowedFromMidnightTo = new TimeSpan(23, 30, 0);

    if (officeFrom.HasValue && officeTo.HasValue)
    {
        if (officeTo == TimeSpan.Zero)
        {
            if (officeFrom == TimeSpan.Zero)
            {
                return "Invalid Office Hour Time: 'From' and 'To' cannot both be 00:00 (midnight).";
            }

            if (officeFrom.Value < minAllowedFromMidnightTo || officeFrom.Value > maxAllowedFromMidnightTo)
            {
                return "Invalid Office Hour Time: If 'To' is 12:00 am (00:00), 'From' must start between 4:30
            }
        }
```

- **OtRecords.cshtml**



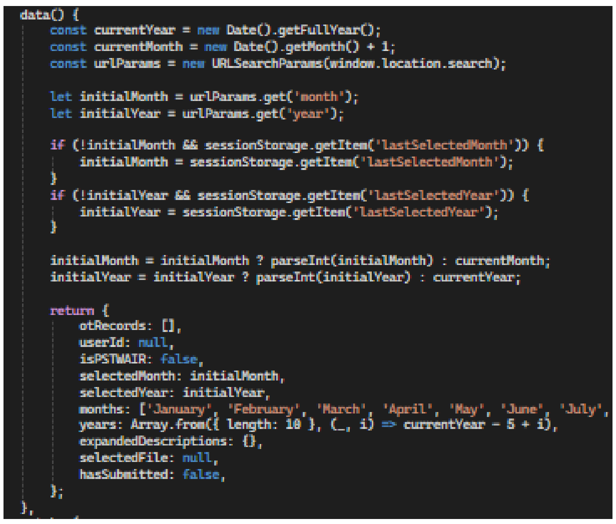
Overtime Records

Month: April
Year: 2025

Date	Office Hour			After Office Hour			Total OT Hours	Break (min)	Net OT Hours	Station	Days	Description	Action
	From	To	Break	From	To	Break							
07/04/2025	-	-	00:00	17:30	23:30	04:00	06:00	04:00	02:00	MWQM T1	Weekday	-	
08/04/2025	-	-	00:00	17:30	23:30	04:00	06:00	04:00	02:00	Marine HQ	Weekday	-	
09/04/2025	-	-	00:00	17:30	23:30	04:00	06:00	04:00	02:00	Marine SOI	Weekday	-	
10/04/2025	-	-	00:00	17:30	23:30	02:00	06:00	02:00	04:00	MWQM T2	Weekday	-	

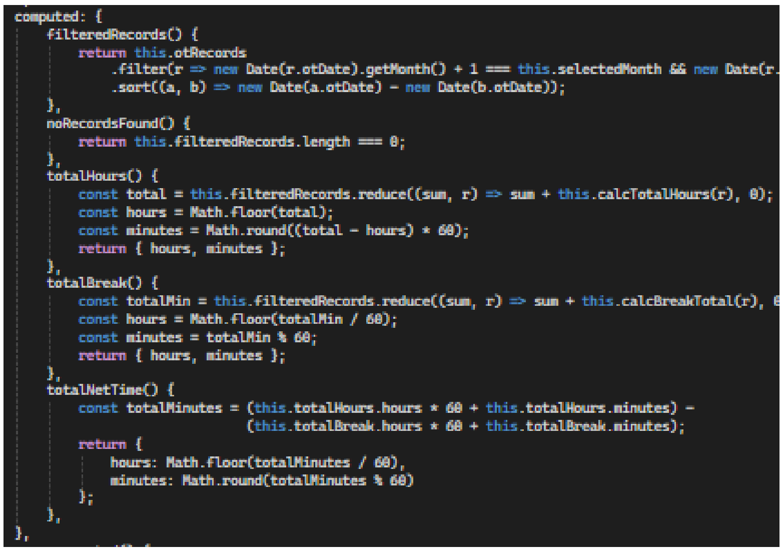
This file (OtRecords.cshtml) is the **main page where individual users view and manage their submitted overtime records**. Users can see their overtime entries for a selected month and year, perform basic calculations (total hours, breaks), edit or delete individual entries, and importantly, **submit their compiled overtime for approval**.

- **data() function**
  - **Purpose**: This function defines all the **reactive state variables** for the Vue application. These variables hold the data that the page displays and interacts with. When these values change, Vue automatically updates the parts of the user interface that depend on them.
  - **Key Variables**:
    - otRecords: An array that will hold all the individual overtime entries fetched from the API for the current user.
    - userId: The ID of the currently logged-in user.
    - isPSTWAIR: A boolean flag indicating if the user's department requires a "Station" column to be displayed (common for specific departments like AIR or MARINE).
    - selectedMonth, selectedYear: The month and year currently selected by the user, influencing which records are displayed. These are initialized from URL parameters or session storage to remember user preferences.
    - months, years: Arrays used to populate the month and year dropdowns.
    - expandedDescriptions: An object used to track which overtime descriptions are currently expanded in the table.
    - selectedFile: Stores the file selected by the user for submission.
    - hasSubmitted: A boolean flag that indicates whether overtime for the selected month/year has already been submitted for approval. This controls the "Submit" button's state.



```
data() {
    const currentYear = new Date().getFullYear();
    const currentMonth = new Date().getMonth() + 1;
    const urlParams = new URLSearchParams(window.location.search);

    let initialMonth = urlParams.get('month');
    let initialYear = urlParams.get('year');

    if (!initialMonth && sessionStorage.getItem('lastSelectedMonth')) {
        initialMonth = sessionStorage.getItem('lastSelectedMonth');
    }
    if (!initialYear && sessionStorage.getItem('lastSelectedYear')) {
        initialYear = sessionStorage.getItem('lastSelectedYear');
    }

    initialMonth = initialMonth ? parseInt(initialMonth) : currentMonth;
    initialYear = initialYear ? parseInt(initialYear) : currentYear;

    return {
        otRecords: [],
        userId: null,
        isPSTWAIR: false,
        selectedMonth: initialMonth,
        selectedYear: initialYear,
        months: ['January', 'February', 'March', 'April', 'May', 'June', 'July',
        years: Array.from({ length: 10 }, (_, i) => currentYear - 5 + i),
        expandedDescriptions: {},
        selectedFile: null,
        hasSubmitted: false,
    };
},
```

- **watch property**
  - **Purpose**: watch allows you to **perform actions when specific data properties change**. This is particularly useful for triggering data reloads when user selections change.
  - **Functionality**:
    - Whenever selectedMonth or selectedYear changes (because the user selected a new month or year from the dropdowns), it triggers getSubmissionStatus(). This ensures that the UI updates the "Submit" button's state to reflect whether OT for the newly selected period has already been submitted.
    - It also saves the selectedMonth and selectedYear to sessionStorage to remember the user's last choice.
- **computed properties**
  - **Purpose**: computed properties are like **dynamic values that are calculated based on other data properties**. They are cached and only re-calculated when their dependencies change, making them efficient for filtering, sorting, or complex display logic.
  - **Key Computed Properties**:
    - filteredRecords: Filters otRecords to show only the entries for the currently selectedMonth and selectedYear, and sorts them by date. This is the data displayed in the main table.
    - noRecordsFound: A boolean flag that is true if filteredRecords is empty, used to display a "No records found" message and disable PDF/Excel buttons.
    - totalHours, totalBreak, totalNetTime: These calculate the overall sum of total overtime hours, total break minutes, and net overtime hours for all records currently displayed in the table. They return objects with hours and minutes for formatted display.



```
computed: {
    filteredRecords() {
        return this.otRecords
            .filter(r => new Date(r.otDate).getMonth() + 1 === this.selectedMonth && new Date(r.
            .sort((a, b) => new Date(a.otDate) - new Date(b.otDate));
    },
    noRecordsFound() {
        return this.filteredRecords.length === 0;
    },
    totalHours() {
        const total = this.filteredRecords.reduce((sum, r) => sum + this.calcTotalHours(r), 0);
        const hours = Math.floor(total);
        const minutes = Math.round((total - hours) * 60);
        return { hours, minutes };
    },
    totalBreak() {
        const totalMin = this.filteredRecords.reduce((sum, r) => sum + this.calcBreakTotal(r), 0
        const hours = Math.floor(totalMin / 60);
        const minutes = totalMin % 60;
        return { hours, minutes };
    },
    totalNetTime() {
        const totalMinutes = (this.totalHours.hours * 60 + this.totalHours.minutes) -
                             (this.totalBreak.hours * 60 + this.totalBreak.minutes);
        return {
            hours: Math.floor(totalMinutes / 60),
            minutes: Math.round(totalMinutes % 60)
        };
    },
},
```

➢ **mounted()**

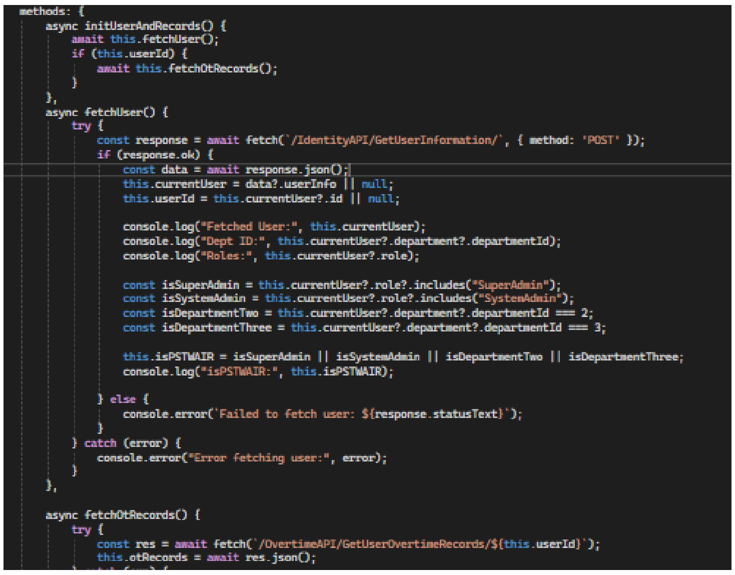
- **Purpose**: This lifecycle hook runs automatically **after the Vue instance has been successfully attached to the HTML element (#app)**. It's the primary place for initial setup and data fetching.
- **Functionality**:
  - It calls initUserAndRecords() to fetch information about the logged-in user and their overtime records.
  - After user and records are fetched, it calls getSubmissionStatus() to determine if the overtime for the initially selected month/year has already been submitted.
  - It stores the initial selectedMonth and selectedYear in sessionStorage.

➢ **methods property**

- **Purpose**: This section defines all the **functions that perform actions** in response to user events (like button clicks, dropdown changes) or programmatic triggers.
- **Key Methods**:
  - initUserAndRecords(): A helper to ensure fetchUser() is called before fetchOtRecords(), so the userId is available.
  - fetchUser(): **Fetches the currently logged-in user's information from /IdentityAPI/GetUserInformation/**. This populates this.currentUser and this.userId, and sets the isPSTWAIR flag based on the user's department/roles, which controls the visibility of the "Station" column.

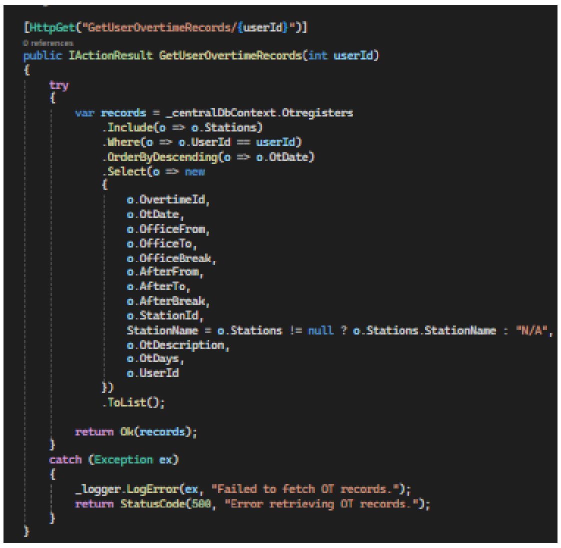
- fetchOtRecords(): **Fetches all overtime records for the current user from /OvertimeAPI/GetUserOvertimeRecords/{userId}**. This populates the otRecords array.
- getSubmissionStatus(): **Checks if overtime for the selectedMonth and selectedYear has already been submitted for approval by calling /OvertimeAPI/CheckOvertimeSubmitted/{userId}/{month}/{year}**. It updates the hasSubmitted flag, which enables/disables the "Submit" button.
- toggleDescription(index): Toggles the expanded class for a description, allowing users to read full descriptions in the table.
- **Formatting/Calculation Helpers (formatDate, formatTime, formatMinutesToHourMinute, getTimeDiff, calcTotalTime, calcTotalHours, calcBreakTotal, calcNetHours, formatHourMinute)**: These methods handle parsing and formatting time and date strings for display, and perform the necessary calculations for total hours, breaks, and net overtime.
- editRecord(index): Redirects the user to the EditOvertime.cshtml page, passing the overtimeId of the selected record, along with the current selectedMonth and selectedYear for a seamless return.
- deleteRecord(index): Prompts for confirmation and then **sends an HTTP DELETE request to /OvertimeAPI/DeleteOvertimeRecord/{id}** to remove a specific overtime entry. If successful, it removes the record from the otRecords array, updating the table.
- printPdf(), downloadPdf(): **Generate a simplified PDF report** of the current month's overtime records by calling /OvertimeAPI/GetUserOvertimePdf/{userId}/{month}/{year}. printPdf opens it in a new window for printing, while downloadPdf triggers a file download.
- downloadExcel(month, year): **Generates an Excel report** of the current month's overtime records by calling /OvertimeAPI/GenerateUserOvertimeExcel/{userId}/{month}/{year} and triggers a download.
- openSubmitModal(): Opens a Bootstrap modal for the user to upload a supporting file before submitting.
- handleFileUpload(event): Captures the file selected by the user in the modal's file input.
- submitOvertime(): **Sends an HTTP POST request to /OvertimeAPI/SubmitOvertime**, including the selectedMonth, selectedYear, and the selectedFile (e.g., an Excel sheet). This

action initiates the approval process for the month's overtime. After successful submission, it closes the modal and refreshes the hasSubmitted status.



```javascript
methods: {
    async initUserAndRecords() {
        await this.fetchUser();
        if (this.userId) {
            await this.fetchOtRecords();
        }
    },
    async fetchUser() {
        try {
            const response = await fetch(`/IdentityAPI/GetUserInformation/`, { method: 'POST' });
            if (response.ok) {
                const data = await response.json();
                this.currentUser = data?.userInfo || null;
                this.userId = this.currentUser?.id || null;

                console.log("Fetched User:", this.currentUser);
                console.log("Dept ID:", this.currentUser?.department?.departmentId);
                console.log("Roles:", this.currentUser?.role);

                const isSuperAdmin = this.currentUser?.role?.includes("SuperAdmin");
                const isSystemAdmin = this.currentUser?.role?.includes("SystemAdmin");
                const isDepartmentTwo = this.currentUser?.department?.departmentId === 2;
                const isDepartmentThree = this.currentUser?.department?.departmentId === 3;

                this.isPSTWAIR = isSuperAdmin || isSystemAdmin || isDepartmentTwo || isDepartmentThree;
                console.log("isPSTWAIR:", this.isPSTWAIR);

            } else {
                console.error(`Failed to fetch user: ${response.statusText}`);
            }
        } catch (error) {
            console.error("Error fetching user:", error);
        }
    },

    async fetchOtRecords() {
        try {
            const res = await fetch(`/OvertimeAPI/GetUserOvertimeRecords/${this.userId}`);
            this.otRecords = await res.json();
```

- **OvertimeAPI.cs (OtRecords.cshtml)**
  - **[HttpGet("GetUserOvertimeRecords/{userId}")]**
    - **Purpose**: This endpoint provides all the **individual overtime entries for a specific user**. It's the primary data source for the OtRecords.cshtml table.
    - **Key Functionality**:
      - Takes a userId as input.
      - Queries the _centralDbContext.Otregisters (your overtime registration table).
      - Includes related Stations data (if an entry has an associated station).
      - Filters records by the provided userId.
      - Orders the records by OtDate in descending order (most recent first).
      - Selects specific fields to return, including a StationName if available.



```
[HttpGet("GetUserOvertimeRecords/{userId}")]
0 references
public IActionResult GetUserOvertimeRecords(int userId)
{
    try
    {
        var records = _centralDbContext.Otregisters
            .Include(o => o.Stations)
            .Where(o => o.UserId == userId)
            .OrderByDescending(o => o.OtDate)
            .Select(o => new
            {
                o.OvertimeId,
                o.OtDate,
                o.OfficeFrom,
                o.OfficeTo,
                o.OfficeBreak,
                o.AfterFrom,
                o.AfterTo,
                o.AfterBreak,
                o.StationId,
                StationName = o.Stations != null ? o.Stations.StationName : "N/A",
                o.OtDescription,
                o.OtDays,
                o.UserId
            })
            .ToList();

        return Ok(records);
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Failed to fetch OT records.");
        return StatusCode(500, "Error retrieving OT records.");
    }
}
```

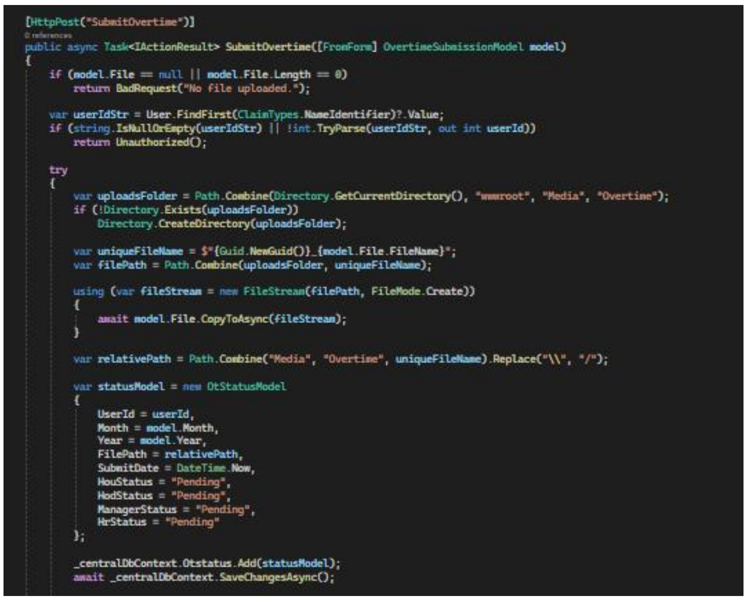
➢ **[HttpDelete("DeleteOvertimeRecord/{id}")]**

- **Purpose**: Allows a user to **delete one of their individual overtime records**.
- **Key Functionality**:
    - Takes an id (the OvertimeId of the record to delete).
    - Finds the record in _centralDbContext.Otregisters.
    - If the record is found, it's removed from the database using _centralDbContext.Otregisters.Remove(record).
    - _centralDbContext.SaveChanges() persists the deletion.
    - Returns a success message or a "Not Found" / "Error" response.

➢ **[HttpPost("SubmitOvertime")]**

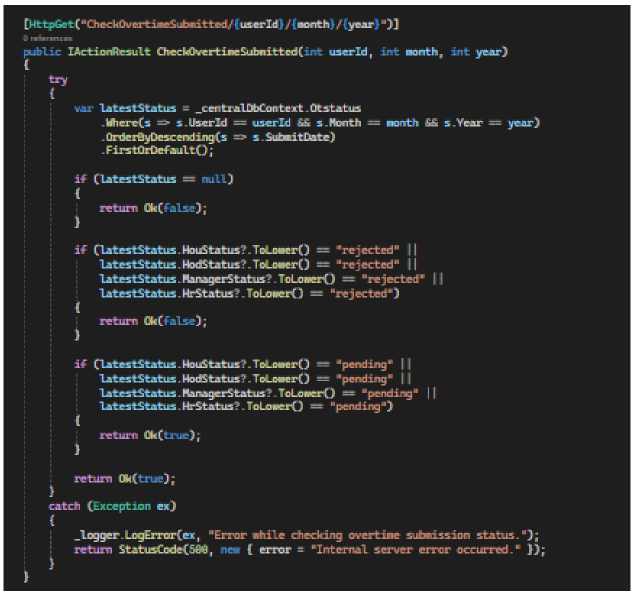
- **Purpose**: This crucial endpoint allows a user to **formally submit their overtime records for a given month and year for approval**. It also handles the upload of an accompanying file (e.g., a signed form or supporting document).
- **Key Functionality**:
    - Receives an OvertimeSubmissionModel from the front-end, which includes the Month, Year, and the File itself (an IFormFile).
    - **File Upload**:
        - Validates that a file has been uploaded.
        - Determines the server path where the file will be saved (e.g., wwwroot/Media/Overtime).
        - Generates a unique filename to prevent conflicts.

        - Saves the uploaded File to the specified server path.
        - Constructs a relativePath to the saved file for database storage.
    - **Create OtStatusModel**: Creates a new OtStatusModel record, linking it to the UserId, Month, Year, and the FilePath of the uploaded document. It initializes all approval statuses (HoU, HoD, Manager, HR) to "Pending". This record represents the *overall submission* for the month.
    - **Link OtRegisterModel to OtStatusModel**: After the OtStatusModel is saved (and gets its StatusId), it finds all individual OtRegisterModel records for that user and month/year that were just submitted. It then updates their StatusId field to link them to the newly created OtStatusModel.
    - **Update HrUserSetting Timestamp**: Updates the ApprovalUpdate timestamp in the user's HrUserSetting record, indicating when their approval flow was last touched.
    - Saves all changes to the database.
    - Returns an Ok response upon success or error messages if anything goes wrong.



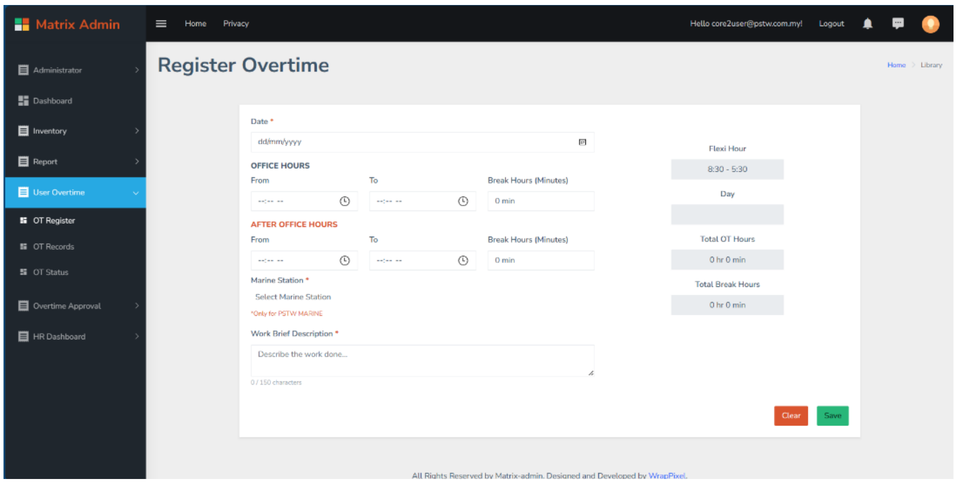
```
[HttpPost("SubmitOvertime")]
0 references
public async Task<IActionResult> SubmitOvertime([FromForm] OvertimeSubmissionModel model)
{
    if (model.File == null || model.File.Length == 0)
        return BadRequest("No file uploaded.");

    var userIdStr = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
    if (string.IsNullOrEmpty(userIdStr) || !int.TryParse(userIdStr, out int userId))
        return Unauthorized();

    try
    {
        var uploadsFolder = Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "Media", "Overtime");
        if (!Directory.Exists(uploadsFolder))
            Directory.CreateDirectory(uploadsFolder);

        var uniqueFileName = $"{Guid.NewGuid()}_{model.File.FileName}";
        var filePath = Path.Combine(uploadsFolder, uniqueFileName);

        using (var fileStream = new FileStream(filePath, FileMode.Create))
        {
            await model.File.CopyToAsync(fileStream);
        }

        var relativePath = Path.Combine("Media", "Overtime", uniqueFileName).Replace("\\", "/");

        var statusModel = new OtStatusModel
        {
            UserId = userId,
            Month = model.Month,
            Year = model.Year,
            FilePath = relativePath,
            SubmitDate = DateTime.Now,
            HouStatus = "Pending",
            HodStatus = "Pending",
            ManagerStatus = "Pending",
            HrStatus = "Pending"
        };

        _centralDbContext.Otstatus.Add(statusModel);
        await _centralDbContext.SaveChangesAsync();
```

- **[HttpGet("CheckOvertimeSubmitted/{userId}/{month}/{year}")]**
  - **Purpose**: This endpoint checks whether a user's overtime for a specific month and year has already been submitted for approval (and if it's currently pending or approved, as opposed to rejected). This is used by the front-end to enable or disable the "Submit" button.
  - **Key Functionality**:
    - Queries _centralDbContext.Otstatus for the latest submission for the given userId, month, and year.
    - If no submission is found, it returns false (not submitted).
    - If a submission exists and *any* approver (HoU, HoD, Manager, HR) has Rejected it, it also returns false, allowing the user to re-submit.
    - If the latest submission is Pending or Approved at any level, it returns true, indicating that a submission is active and new submissions are not allowed until the current one is fully rejected.



```
[HttpGet("CheckOvertimeSubmitted/{userId}/{month}/{year}")]
0 references
public IActionResult CheckOvertimeSubmitted(int userId, int month, int year)
{
    try
    {
        var latestStatus = _centralDbContext.Otstatus
            .Where(s => s.UserId == userId && s.Month == month && s.Year == year)
            .OrderByDescending(s => s.SubmitDate)
            .FirstOrDefault();

        if (latestStatus == null)
        {
            return Ok(false);
        }

        if (latestStatus.HouStatus?.ToLower() == "rejected" ||
            latestStatus.HodStatus?.ToLower() == "rejected" ||
            latestStatus.ManagerStatus?.ToLower() == "rejected" ||
            latestStatus.HrStatus?.ToLower() == "rejected")
        {
            return Ok(false);
        }

        if (latestStatus.HouStatus?.ToLower() == "pending" ||
            latestStatus.HodStatus?.ToLower() == "pending" ||
            latestStatus.ManagerStatus?.ToLower() == "pending" ||
            latestStatus.HrStatus?.ToLower() == "pending")
        {
            return Ok(true);
        }

        return Ok(true);
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error while checking overtime submission status.");
        return StatusCode(500, new { error = "Internal server error occurred." });
    }
}
```

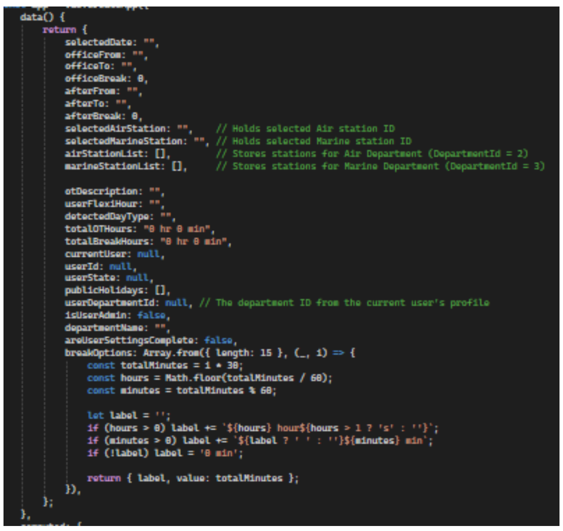
- **OtRegister.cshtml**



This file (OtRegister.cshtml) provides the **main form for users to register new overtime entries**. It's a single-page form where employees input details like the date, time worked (both during and after office hours), breaks, any associated station, and a description of the work done. The page dynamically calculates total hours and identifies the day type (weekday, weekend, public holiday) while providing clear validation.

- **data() function**
  - **Purpose**: This function defines all the **reactive state variables** for the Vue application. These variables hold the data that the form captures and displays. When these values change, Vue automatically updates the relevant parts of the user interface.
  - **Key Variables**:
    - selectedDate, officeFrom, officeTo, officeBreak, afterFrom, afterTo, afterBreak, otDescription: These directly correspond to the input fields on the form for capturing overtime details. officeBreak and afterBreak default to 0.
    - selectedAirStation, selectedMarineStation: Store the IDs of chosen stations from dropdowns.
    - airStationList, marineStationList: Arrays holding the lists of available Air and Marine stations, fetched from the API.
    - userFlexiHour, detectedDayType, totalOTHours, totalBreakHours: Display-only fields that show calculated or fetched information.
    - currentUser, userId, userDepartmentId, isUserAdmin, departmentName: Information about the currently logged-in user, crucial for determining which station dropdowns to show and for linking the overtime entry.

- userState, publicHolidays: Data (fetched from API) about the user's assigned state and its public holidays, used to correctly identify the detectedDayType.
- areUserSettingsComplete: A crucial flag that determines if the user has all necessary HR settings configured (flexi-hour, salary, state, approval flow). If false, the "Save" button is disabled.
- breakOptions: A pre-defined array of options for break duration dropdowns (e.g., "30 min", "1 hour").
- validationErrors: An object to store client-side validation error messages, which are displayed dynamically below the relevant input fields.



```
data() {
    return {
        selectedDate: "",
        officeFrom: "",
        officeTo: "",
        officeBreak: 0,
        afterFrom: "",
        afterTo: "",
        afterBreak: 0,
        selectedAirStation: "",    // Holds selected Air station ID
        selectedMarineStation: "", // Holds selected Marine station ID
        airStationList: [],        // Stores stations for Air Department (DepartmentId = 2)
        marineStationList: [],     // Stores stations for Marine Department (DepartmentId = 3)

        otDescription: "",
        userFlexiHour: "",
        detectedDayType: "",
        totalOTHours: "0 hr 0 min",
        totalBreakHours: "0 hr 0 min",
        currentUser: null,
        userId: null,
        userState: null,
        publicHolidays: [],
        userDepartmentId: null, // The department ID from the current user's profile
        isUserAdmin: false,
        departmentName: "",
        areUserSettingsComplete: false,
        breakOptions: Array.from({ length: 15 }, (_, i) => {
            const totalMinutes = i * 30;
            const hours = Math.floor(totalMinutes / 60);
            const minutes = totalMinutes % 60;

            let label = '';
            if (hours > 0) label += `${hours} hour${hours > 1 ? 's' : ''}`;
            if (minutes > 0) label += `${label ? ' ' : ''}${minutes} min`;
            if (!label) label = '0 min';

            return { label, value: totalMinutes };
        }),
    };
},
```

- **computed properties**
  - **Purpose**: computed properties are dynamic values that Vue.js automatically calculates based on other data properties. They are efficient because they are "cached" and only re-calculated when their dependencies change.
  - **Key Computed Properties**:
    - charCount: Returns the current character count of the otDescription field.
    - showAirDropdown, showMarineDropdown: Booleans that control the visibility of the Air or Marine station dropdowns, based on the isUserAdmin flag or userDepartmentId.
    - stationIdForSubmission: Determines which station ID (Air or Marine) should be used when submitting the form, or null if none is selected. This ensures only one station is submitted.
    - requiresStation: A boolean indicating if *any* station dropdown is visible, which implies a station *might* be required for validation.

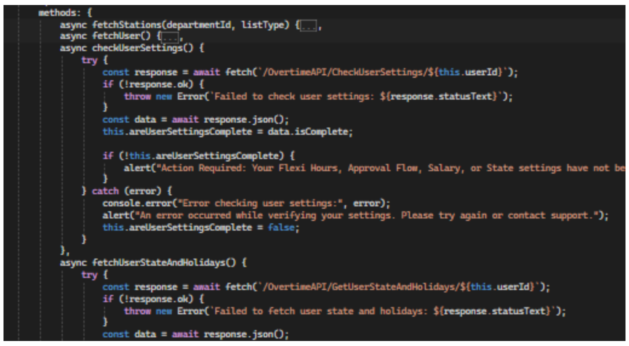
- **watch property**
  - **Purpose**: watch allows you to **execute specific code when a data property changes**. This is used here to synchronize jQuery's Select2 dropdowns with Vue's data.
  - **Functionality**: When airStationList or marineStationList are updated (meaning the station data has been fetched), these watchers ensure that the corresponding jQuery Select2 dropdowns are correctly initialized or re-initialized, and their selected values are synchronized with selectedAirStation/selectedMarineStation.

- **mounted()**
  - **Purpose**: This lifecycle hook runs automatically **after the Vue instance has been successfully attached to the HTML element (#app)**. It's the primary place for initial setup and data fetching.
  - **Functionality**:
    - Calls fetchUser() to get information about the logged-in user (ID, department, roles).
    - Once the userId is available, it calls checkUserSettings() to verify if the user's essential HR settings are complete. This is crucial as saving overtime is disabled if settings are incomplete.
    - Based on the user's department/admin status (showAirDropdown, showMarineDropdown), it conditionally fetches the relevant station lists (fetchStations).
- **methods property**
  - **Purpose**: This section defines all the **functions that perform actions** in response to user input (like typing, clicking buttons, changing dropdowns) or other programmatic triggers.
  - **Key Methods**:
    - fetchStations(departmentId, listType): **Fetches a list of stations for a specific departmentId from /OvertimeAPI/GetStationsByDepartment**. It populates either airStationList or marineStationList.
    - fetchUser(): **Fetches detailed information about the currently logged-in user from /IdentityAPI/GetUserInformation/**. This populates currentUser, userId, userDepartmentId, isUserAdmin, and departmentName. It then triggers fetchUserStateAndHolidays() and updates userFlexiHour.
    - checkUserSettings(): **Calls /OvertimeAPI/CheckUserSettings/{userId} to verify if the user has complete flexi-hour, state, approval flow, and salary**

settings. It updates areUserSettingsComplete and displays an alert if settings are missing, disabling the save button.

- fetchUserStateAndHolidays(): **Fetches the user's assigned state and its public holidays for the current year from /OvertimeAPI/GetUserStateAndHolidays/{userId}**. This data is crucial for determining the detectedDayType.
- roundToNearest30(timeStr): Rounds a given time string (e.g., "08:17") to the nearest 30-minute interval (e.g., "08:30"). This ensures consistency in time entries.
- limitCharCount(event): Restricts the length of the otDescription text area to 150 characters.
- calculateOTAndBreak(): **Calculates the total overtime hours and total break hours** based on the officeFrom/To, afterFrom/To times, and officeBreak/afterBreak values. It updates totalOTHours and totalBreakHours for display and also calls updateDayType().
- calculateTimeDifference(startTime, endTime, breakMinutes): A helper method for calculateOTAndBreak() that calculates the duration between two times, accounting for breaks and handling time ranges that cross midnight.
- parseTime(timeString), formatTime(timeString): Utility methods for converting time strings to objects/numbers and back for calculations and display.
- handleDateChange(): Triggered when the selectedDate changes. It calls updateDayType() and calculateOTAndBreak() to re-evaluate the day type and hours.
- addOvertime(): This is the **main method for submitting a new overtime record**.
  - It first performs **client-side validation**: checks if user settings are complete, date/description/station (if required) are filled, and time ranges are valid (including complex midnight-crossing rules).
  - If validation passes, it compiles all the form data into a requestData object.
  - It **sends an HTTP POST request to /OvertimeAPI/AddOvertime** with this requestData.
  - Upon successful save, it displays an alert, clears the form, and re-fetches user settings and holidays to ensure the form is ready for a new entry.

        - updateDayType(): Determines if the selectedDate is a "Public Holiday," "Weekend," or "Weekday" based on the user's state's weekend configuration and fetched public holidays.
        - clearForm(): Resets all form fields, calculated totals, and selected stations to their initial empty states, preparing the form for a new entry. It also resets the jQuery Select2 dropdowns.



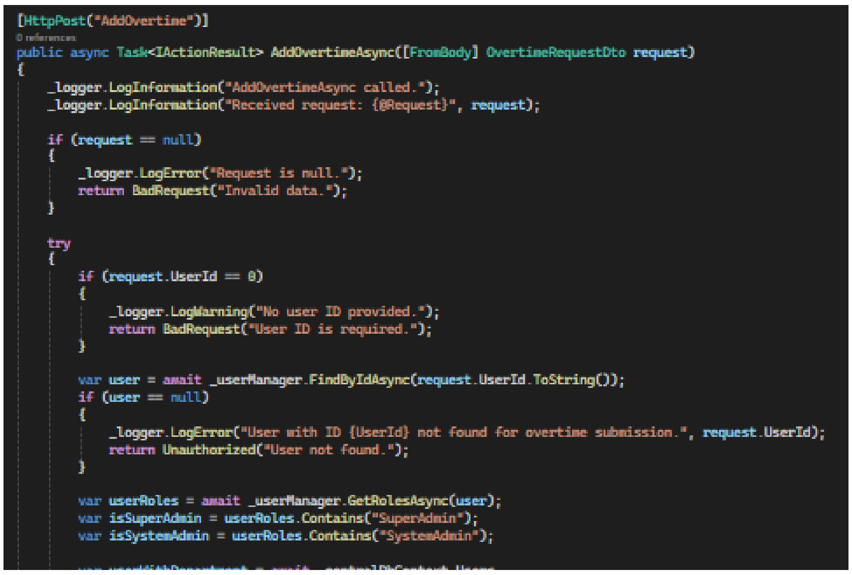
```
methods: {
    async fetchStations(departmentId, listType) {...},
    async fetchUser() {...},
    async checkUserSettings() {
        try {
            const response = await fetch(`/OvertimeAPI/CheckUserSettings/${this.userId}`);
            if (!response.ok) {
                throw new Error(`Failed to check user settings: ${response.statusText}`);
            }
            const data = await response.json();
            this.areUserSettingsComplete = data.isComplete;

            if (!this.areUserSettingsComplete) {
                alert("Action Required: Your Flexi Hours, Approval Flow, Salary, or State settings have not be
            }
        } catch (error) {
            console.error("Error checking user settings:", error);
            alert("An error occurred while verifying your settings. Please try again or contact support.");
            this.areUserSettingsComplete = false;
        }
    },
    async fetchUserStateAndHolidays() {
        try {
            const response = await fetch(`/OvertimeAPI/GetUserStateAndHolidays/${this.userId}`);
            if (!response.ok) {
                throw new Error(`Failed to fetch user state and holidays: ${response.statusText}`);
            }
            const data = await response.json();
```

- **OvertimeAPI.cs (OtRegister.cshtml)**
    - **[HttpGet("GetStationsByDepartment")]**
        - **Purpose**: Provides a list of stations filtered by a specific departmentId. This is a general endpoint that OtRegister.cshtml (and other pages) can use to populate station dropdowns relevant to the user's department.
        - **Key Functionality**: Takes an optional departmentId. If provided, it queries the _centralDbContext.Stations table and returns stations that match that department ID.
    - **[HttpPost("AddOvertime")]**
        - **Purpose**: This is the core endpoint responsible for **receiving new overtime record data from the front-end and saving it to the database**.
        - **Key Functionality**:
            - Receives an OvertimeRequestDto from the client, which contains all the details of the new overtime entry.
            - **User Validation**: Ensures a UserId is provided and that the user actually exists.
            - **Station Requirement Check**: Dynamically checks if a station is required for the user based on their department (DepartmentId == 2 for Air, 3 for Marine) or if they are an "Admin" role. If required and missing, it returns a BadRequest.
            - **Time Validation (Server-Side)**: Contains robust server-side validation for OfficeFrom/To and AfterFrom/To time ranges. This

ensures data integrity even if client-side validation is bypassed. It checks for:

    ▪ Both "From" and "To" times being provided if one is entered.
    ▪ "To" time being later than "From" time for same-day durations.
    ▪ Special rules for time ranges that cross midnight (e.g., if "To" is 00:00, "From" must be within a specific late-evening window).
    ▪ Ensuring at least one set of hours (Office or After Office) is provided.

- **Data Mapping**: Converts string-based time inputs (e.g., "08:30") from the request into TimeSpan objects, which are suitable for database storage.
- **Database Insertion**: Creates a new OtRegisterModel object using the validated and parsed data, then adds it to the _centralDbContext.Otregisters table.
- **Saving Changes**: Calls await _centralDbContext.SaveChangesAsync() to persist the new record in the database.
- **Response**: Returns a success message (Ok) or a detailed error message (BadRequest or StatusCode(500)).



```
[HttpPost("AddOvertime")]
0 references
public async Task<IActionResult> AddOvertimeAsync([FromBody] OvertimeRequestDto request)
{
    _logger.LogInformation("AddOvertimeAsync called.");
    _logger.LogInformation("Received request: {@Request}", request);

    if (request == null)
    {
        _logger.LogError("Request is null.");
        return BadRequest("Invalid data.");
    }

    try
    {
        if (request.UserId == 0)
        {
            _logger.LogWarning("No user ID provided.");
            return BadRequest("User ID is required.");
        }

        var user = await _userManager.FindByIdAsync(request.UserId.ToString());
        if (user == null)
        {
            _logger.LogError("User with ID {UserId} not found for overtime submission.", request.UserId);
            return Unauthorized("User not found.");
        }

        var userRoles = await _userManager.GetRolesAsync(user);
        var isSuperAdmin = userRoles.Contains("SuperAdmin");
        var isSystemAdmin = userRoles.Contains("SystemAdmin");
```

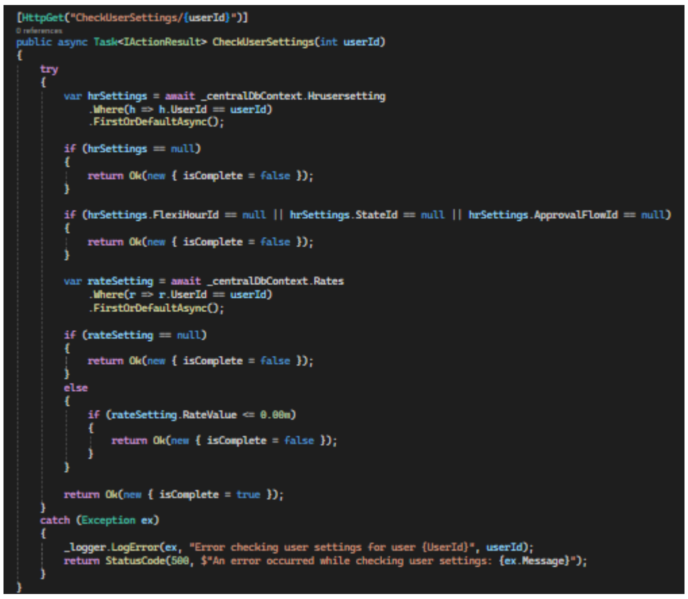
➢ **[HttpGet("GetUserStateAndHolidays/{userId}")]**

- **Purpose**: Provides essential user-specific settings (like their assigned state, its weekend configuration, and public holidays) to the front-end. This data is critical for the front-end to correctly determine the "Day Type" (Weekday, Weekend, Public Holiday) for the selected overtime date.

- **Key Functionality**: Takes a userId. It queries _centralDbContext.Hrusersetting, including linked State and Weekends information, as well as FlexiHour. It then fetches relevant Holidays for that state.

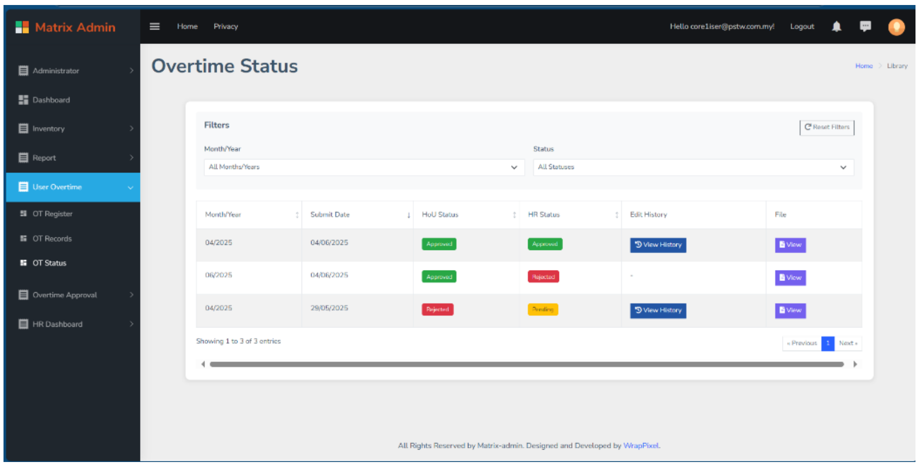
➢ **[HttpGet("CheckUserSettings/{userId}")]**

- **Purpose**: This endpoint performs a critical check to see if a user has all their **essential HR settings (Flexi-Hour, State, Approval Flow, Salary) configured**. This is used to enable/disable the "Save" button on the front-end to prevent users from submitting incomplete data.
- **Key Functionality**:
    - Queries _centralDbContext.Hrusersetting and _centralDbContext.Rates for the given userId.
    - Returns false (isComplete = false) if:
        - HrUserSetting is missing.
        - FlexiHourId, StateId, or ApprovalFlowId within HrUserSetting are null or zero.
        - Rate (salary) record is missing.
        - RateValue is zero or less.
    - Returns true (isComplete = true) otherwise.



```
[HttpGet("CheckUserSettings/{userId}")]
0 references
public async Task<IActionResult> CheckUserSettings(int userId)
{
    try
    {
        var hrSettings = await _centralDbContext.Hrusersetting
            .Where(h => h.UserId == userId)
            .FirstOrDefaultAsync();

        if (hrSettings == null)
        {
            return Ok(new { isComplete = false });
        }

        if (hrSettings.FlexiHourId == null || hrSettings.StateId == null || hrSettings.ApprovalFlowId == null)
        {
            return Ok(new { isComplete = false });
        }

        var rateSetting = await _centralDbContext.Rates
            .Where(r => r.UserId == userId)
            .FirstOrDefaultAsync();

        if (rateSetting == null)
        {
            return Ok(new { isComplete = false });
        }
        else
        {
            if (rateSetting.RateValue <= 0.00m)
            {
                return Ok(new { isComplete = false });
            }
        }

        return Ok(new { isComplete = true });
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error checking user settings for user {UserId}", userId);
        return StatusCode(500, $"An error occurred while checking user settings: {ex.Message}");
    }
}
```

- **OtStatus.cshtml**



Overtime Status

Filters

Month/Year: All Months/Years

Status: All Statuses

Month/Year	Submit Date	HoU Status	HR Status	Edit History	File
04/2025	04/06/2025	Approved	Approved	View History	View
06/2025	04/06/2025	Approved	Rejected	-	View
04/2025	29/05/2025	Rejected	Pending	View History	View

Showing 1 to 3 of 3 entries

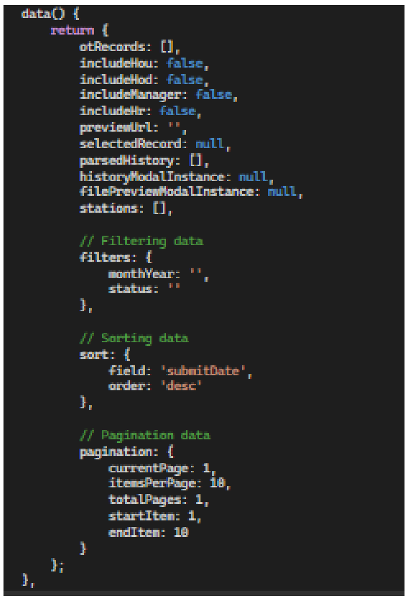
All Rights Reserved by Matrix-admin. Designed and Developed by WrapPixel.

This file (OtStatus.cshtml) provides a **dashboard for individual users to track the approval status of their submitted overtime requests**. It shows a list of their overtime submissions, detailing the approval status at each stage (HoU, HoD, Manager, HR), when it was submitted, if it has a linked file, and if any changes were made during the approval process.

➢ **data() function**

- **Purpose**: This function defines all the **reactive state variables** for the Vue application. These variables hold the data displayed on the page and are automatically watched by Vue for changes, triggering UI updates.
- **Key Variables**:
  - otRecords: The main array containing all overtime submission status records for the current user, fetched from the API.
  - includeHou, includeHod, includeManager, includeHr: Boolean flags that determine which approval columns (HoU, HoD, Manager, HR) should be displayed in the table. These are based on the user's configured approval flow.
  - previewUrl: The URL for the file to be displayed in the file preview modal.
  - selectedRecord: Holds the details of the specific status record whose edit history is being viewed.
  - parsedHistory: An array that will store the parsed and formatted edit/delete history logs for a specific submission.
  - historyModalInstance, filePreviewModalInstance: References to the Bootstrap modal instances, used to show and hide the modals programmatically.
  - stations: A list of all stations, fetched to correctly display station names in the edit history.

- filters: An object containing the current filter selections (monthYear, status).
- sort: An object defining the current sorting criteria (field, order).
- pagination: An object holding pagination-related data (currentPage, itemsPerPage, totalPages, startItem, endItem).



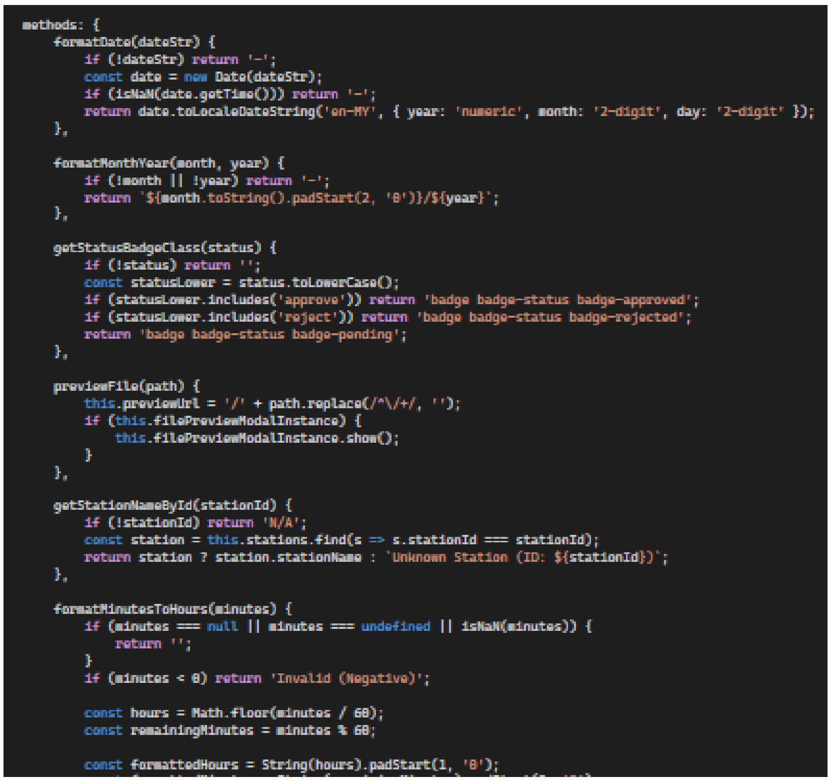
```
data() {
    return {
        otRecords: [],
        includeHou: false,
        includeHod: false,
        includeManager: false,
        includeHr: false,
        previewUrl: '',
        selectedRecord: null,
        parsedHistory: [],
        historyModalInstance: null,
        filePreviewModalInstance: null,
        stations: [],

        // Filtering data
        filters: {
            monthYear: '',
            status: ''
        },

        // Sorting data
        sort: {
            field: 'submitDate',
            order: 'desc'
        },

        // Pagination data
        pagination: {
            currentPage: 1,
            itemsPerPage: 10,
            totalPages: 1,
            startItem: 1,
            endItem: 10
        }
    };
},
```

➢ **computed properties**

- **Purpose**: computed properties dynamically calculate and return values based on other data properties. They are optimized to only re-calculate when their dependencies change.
- **Key Computed Properties**:
  - columnCount: Calculates the total number of columns to display in the table header, dynamically adjusting based on which approver roles are included (includeHou, etc.).
  - monthYearOptions: Generates a list of unique "Month/Year" strings from otRecords to populate the filter dropdown, ensuring they are sorted chronologically.
  - statusOptions: Defines the available status filters ("Pending", "Approved", "Rejected") and conditionally adds "Partially Approved" and "Fully Approved" if there's a multi-stage approval flow (i.e., more than one approver column is included).
  - filteredRecords: This is the **most complex computed property**. It takes the raw otRecords, applies filters (by month/year and status), and then applies sorting based on sort.field and sort.order. It also handles pagination, slicing the data to only show the items for the currentPage and calculating totalPages, startItem, and endItem for the pagination display.

- **mounted()**
  - **Purpose**: This lifecycle hook runs automatically **after the Vue instance has been successfully attached to the HTML element (#app)**. It's the primary place for initial data loading and setting up external JavaScript libraries (like Bootstrap modals).
  - **Functionality**:
    - Uses Promise.all to fetch data from two API endpoints concurrently:
      1. /OvertimeAPI/GetUserOtStatus: This is the **primary data source**, fetching the user's overtime submission statuses and information about which approval columns to include.
      2. /OvertimeAPI/GetAllStations: Fetches a list of all stations, needed for displaying station names correctly in the edit history.
    - Once both API calls complete, it populates includeHou, includeHod, includeManager, includeHr flags, and the otRecords array.
    - It checks if the user's approval flow is configured and displays an alert if not.
    - It stores references to the Bootstrap modal instances for later use (historyModalInstance, filePreviewModalInstance).
- **methods property**
  - **Purpose**: This section defines all the **functions that perform actions** in response to user events (like clicking on table headers for sorting, clicking "View History" buttons, etc.) or for data formatting.
  - **Key Methods**:
    - formatDate(dateStr), formatMonthYear(month, year): Utility methods for formatting dates and month/year strings for display.
    - getStatusBadgeClass(status): Returns the appropriate CSS class for status badges (e.g., badge-approved, badge-rejected) based on the approval status.
    - previewFile(path): Sets the previewUrl for the iframe in the file preview modal and then shows the modal.
    - getStationNameById(stationId): Looks up and returns a station name based on its ID from the stations array. Used primarily for displaying history.
    - formatMinutesToHours(minutes): Converts a total number of minutes into an "HH:MM" format. Used for displaying breaks in history.

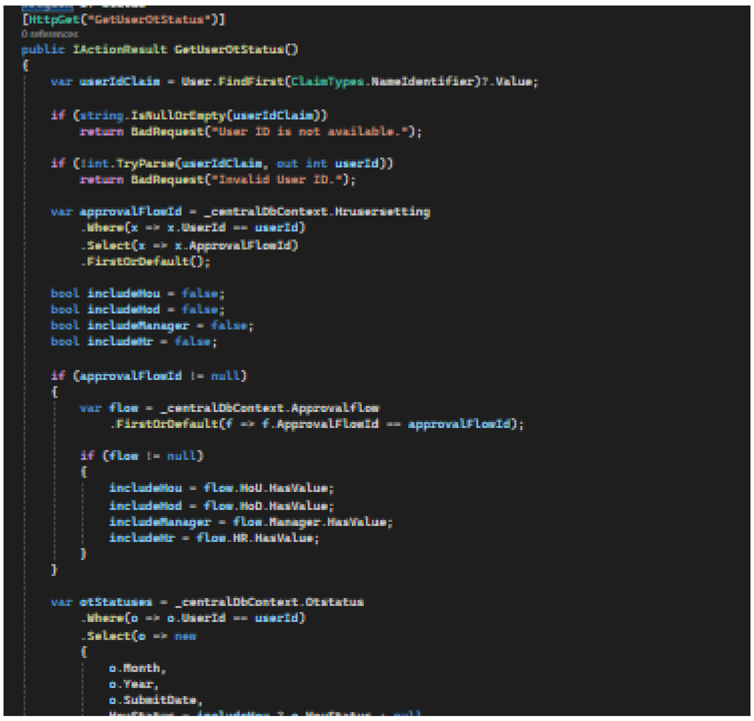
- getChanges(before, after): **Parses and compares "before" and "after" states of an overtime record (from the JSON log strings)** to identify and format specific changes. This is a complex helper for the edit history feature.
- showEditHistory(record): Triggered when "View History" is clicked. It takes the otStatus record, extracts the JSON HouUpdate, HodUpdate, etc., strings, parses them, and then uses getChanges() to format the audit trail for display in the editHistoryModal. It then shows the modal.
- closeEditHistory(): Hides the edit history modal and resets related data.
- sortBy(field): Handles table header clicks for sorting. It toggles the sort.order (asc/desc) if the same column is clicked again, otherwise sets a new sort.field and resets order to 'asc'. It also resets pagination to the first page.
- resetFilters(): Clears all applied filters and resets pagination to the first page.
- prevPage(), nextPage(), goToPage(page): Implement the client-side pagination logic, updating pagination.currentPage and re-rendering the filteredRecords.



```
methods: {
    formatDate(dateStr) {
        if (!dateStr) return '-';
        const date = new Date(dateStr);
        if (isNaN(date.getTime())) return '-';
        return date.toLocaleDateString('en-MY', { year: 'numeric', month: '2-digit', day: '2-digit' });
    },

    formatMonthYear(month, year) {
        if (!month || !year) return '-';
        return `${month.toString().padStart(2, '0')}/${year}`;
    },

    getStatusBadgeClass(status) {
        if (!status) return '';
        const statusLower = status.toLowerCase();
        if (statusLower.includes('approve')) return 'badge badge-status badge-approved';
        if (statusLower.includes('reject')) return 'badge badge-status badge-rejected';
        return 'badge badge-status badge-pending';
    },

    previewFile(path) {
        this.previewUrl = '/' + path.replace(/^\/+/, '');
        if (this.filePreviewModalInstance) {
            this.filePreviewModalInstance.show();
        }
    },

    getStationNameById(stationId) {
        if (!stationId) return 'N/A';
        const station = this.stations.find(s => s.stationId === stationId);
        return station ? station.stationName : `Unknown Station (ID: ${stationId})`;
    },

    formatMinutesToHours(minutes) {
        if (minutes === null || minutes === undefined || isNaN(minutes)) {
            return '';
        }
        if (minutes < 0) return 'Invalid (Negative)';

        const hours = Math.floor(minutes / 60);
        const remainingMinutes = minutes % 60;

        const formattedHours = String(hours).padStart(1, '0');
```

- **OvertimeAPI.cs (OtStatus.cshtml)**
  - **[HttpGet("GetUserOtStatus")]**
    - **Purpose**: This is the primary endpoint that OtStatus.cshtml calls to **retrieve all overtime submission statuses for the currently logged-in user**. It also dynamically determines which approval columns (HoU, HoD, Manager, HR) should be visible on the front-end based on the user's assigned approval flow.
    - **Key Functionality**:
      - **User Identification**: Extracts the userId from the authenticated user's claims.
      - **Determine Approval Flow Structure**:
        - It queries _centralDbContext.Hrusersetting to find the ApprovalFlowId assigned to the current user.
        - If an ApprovalFlowId exists, it then retrieves the specific Approvalflow definition from _centralDbContext.Approvalflow.
        - Based on this flow, it sets boolean flags (includeHou, includeHod, includeManager, includeHr) to true if a corresponding approver (HoU, HoD, Manager, HR) is defined in that flow. These flags are then returned to the front-end to control column visibility.
      - **Fetch OtStatus Records**: It queries _centralDbContext.Otstatus to get all overtime submission status records specifically for the current userId.
      - **Conditional Status Selection**: When selecting data for otStatuses, it **conditionally includes the HouStatus, HodStatus, ManagerStatus, and HrStatus fields based on the includeHou, etc., flags**. If an approver role is *not* included in the user's flow, null is returned for that status, ensuring the front-end doesn't display irrelevant columns or data.
      - **Updated Flag**: Calculates a Updated boolean flag, which is true if any of the HouUpdate, HodUpdate, ManagerUpdate, or HrUpdate (JSON log) fields are not empty. This tells the front-end whether to show the "View History" button.

- **Response**: Returns a JSON object containing:
  - The includeHou, includeHod, includeManager, includeHr flags.
  - The list of otStatuses records for the user (with conditionally populated approval statuses and the Updated flag).
  - A hasApprovalFlow boolean indicating if any approval flow is assigned to the user.



```csharp
[HttpGet("GetUserOtStatus")]
0 references
public IActionResult GetUserOtStatus()
{
    var userIdClaim = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;

    if (string.IsNullOrEmpty(userIdClaim))
        return BadRequest("User ID is not available.");

    if (!int.TryParse(userIdClaim, out int userId))
        return BadRequest("Invalid User ID.");

    var approvalFlowId = _centralDbContext.Hrusersetting
        .Where(x => x.UserId == userId)
        .Select(x => x.ApprovalFlowId)
        .FirstOrDefault();

    bool includeHou = false;
    bool includeHod = false;
    bool includeManager = false;
    bool includeHr = false;

    if (approvalFlowId != null)
    {
        var flow = _centralDbContext.Approvalflow
            .FirstOrDefault(f => f.ApprovalFlowId == approvalFlowId);

        if (flow != null)
        {
            includeHou = flow.HoU.HasValue;
            includeHod = flow.HoD.HasValue;
            includeManager = flow.Manager.HasValue;
            includeHr = flow.HR.HasValue;
        }
    }

    var otStatuses = _centralDbContext.Otstatus
        .Where(o => o.UserId == userId)
        .Select(o => new
        {
            o.Month,
            o.Year,
            o.SubmitDate,
```