Room Booking System Documentation

Prepared by: Harris Bin Mulsiham

Table of Contents

FOLDER STRUCTURE 3
DATABASE TABLES 4
DATABASE CONTEXT CONFIGURATION (CentralSystemContext.cs) 4
SIDEBAR NAVIGATION 5
MODEL 6
- BookingsModels.cs 6
- Booking.cs 7
- Room.cs 9

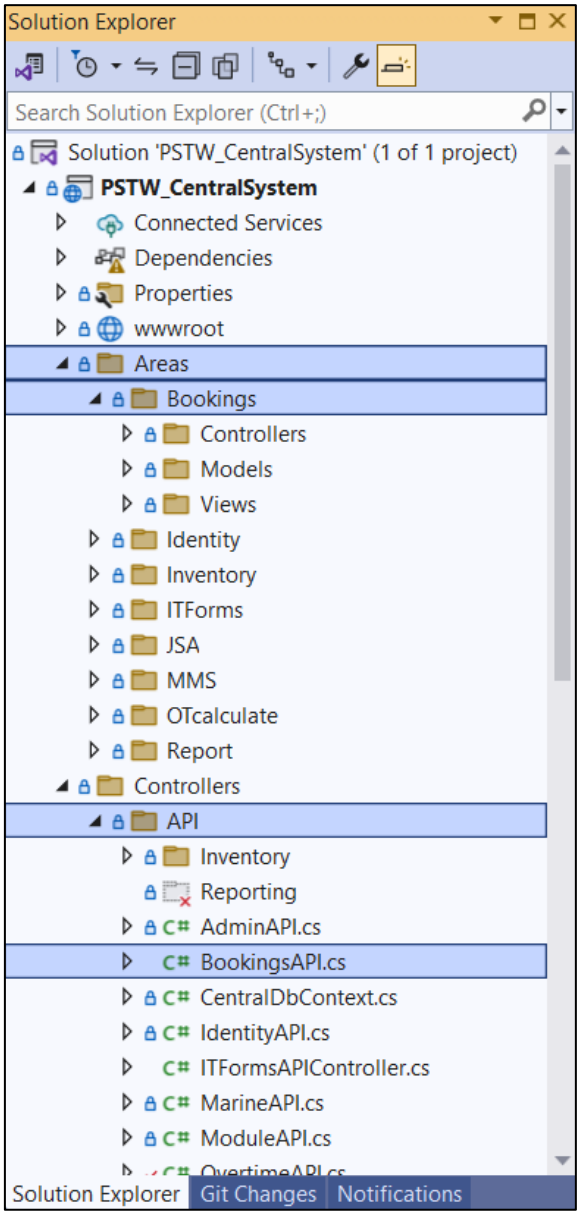
VIEWS & BOOKINGS API 11
- Bookings.cs 11
  - Index.cshtml 11
  - BookingsAPI(Index.cshtml) 21
  - Create.cshtml 29
  - BookingsAPI(Create.cshtml) 35
  - Calendar.cshtml 40
  - BookingsAPI(Calender.cshtml) 47
  - Managers.cshtml 50
  - BookingsAPI(Managers.cshtml) 52
  - Room.cshtml 54
  - BookingsAPI(Room.cshtml) 59

CONTROLLER 62
- BookingsController.cs 62

FOLDER STRUCTURE

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

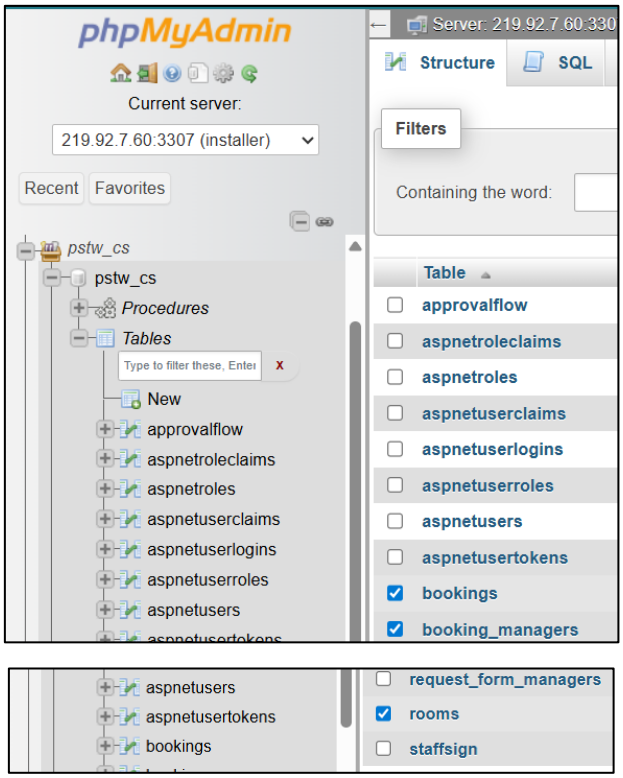
- **Bookings** Module: This module contains the **Controllers**, **Models**, and **Views** directories, designed to encapsulate the logic and presentation for the overtime system.
- **API Directory** (Controllers/API): Under the Controllers folder, within the API sub-directory, the **BookingsAPI.cs**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



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 **bookings**, **rooms, booking_managers**



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.

```
14 references
public DbSet<Booking> Bookings { get; set; }
10 references
public DbSet<Room> Rooms { get; set; }
6 references
public DbSet<BookingManager> BookingManager { get; set; }
```

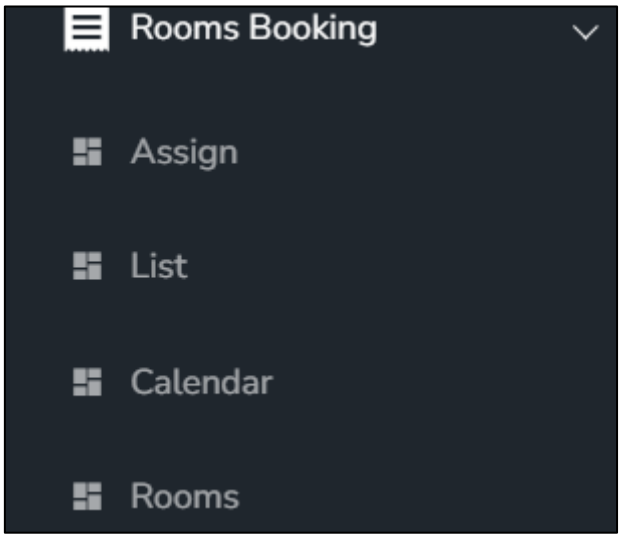
SIDEBAR NAVIGATION

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

Rooms Booking Menu Section: This section facilitates user-specific overtime functionalities. It includes a parent menu item, "User Overtime," which expands to reveal sub-links for:

- "List": Allows users to view a list of bookings made along with their booking status.
- "Calendar": Allows users to view bookings in a calendar format.
- "Rooms": Allows booking managers to view list of rooms with their statuses
- “Assign”: Allows booking managers to view list of users registered to be assigned as booking managers.

```
<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">Rooms Booking</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="Bookings" asp-controller="Bookings" asp-action="Managers" aria-expanded="false">
                <i class="mdi mdi-view-dashboard"></i><span class="hide-menu">Assign</span>
            </a>
        </li>
        <li class="sidebar-item">
            <a class="sidebar-link waves-effect waves-dark sidebar-link" asp-area="Bookings" asp-controller="Bookings" asp-action="Index" aria-expanded="false">
                <i class="mdi mdi-view-dashboard"></i><span class="hide-menu">List</span>
            </a>
        </li>
        <li class="sidebar-item">
            <a class="sidebar-link waves-effect waves-dark sidebar-link" asp-area="Bookings" asp-controller="Bookings" asp-action="Calendar" aria-expanded="false">
                <i class="mdi mdi-view-dashboard"></i><span class="hide-menu">Calendar</span>
            </a>
        </li>
        <li class="sidebar-item">
            <a class="sidebar-link waves-effect waves-dark sidebar-link" asp-area="Bookings" asp-controller="Bookings" asp-action="Room" aria-expanded="false">
                <i class="mdi mdi-view-dashboard"></i><span class="hide-menu">Rooms</span>
            </a>
        </li>
    </ul>
</li>
```



MODEL

- **BookingsModels.cs**

  This model is designed to represent a single room booking record in the system. It links each booking to a user, a room, and the organization context at the time of booking. The model also enforces validation rules such as booking duration and time overlap logic.

BookingStatus (Enum)

Purpose: Define the lifecycle states of a booking so the system can enforce rules (e.g., deletion limits, approvals).

How it works: Stored in Booking.CurrentStatus as an integer (0–3). UI and APIs read this value to decide what actions are allowed.

- **Pending = 0**

  Purpose: Represent a newly submitted booking awaiting a decision.

  How it works: Default on create. Can transition to Approved or Rejected; user may still cancel before start time, per your rules.

- **Approved = 1**

  Purpose: Mark a booking as confirmed so the room is considered reserved.

  How it works: Blocks time for the room. Deletion is generally not allowed (auditable history).

- **Rejected = 2**

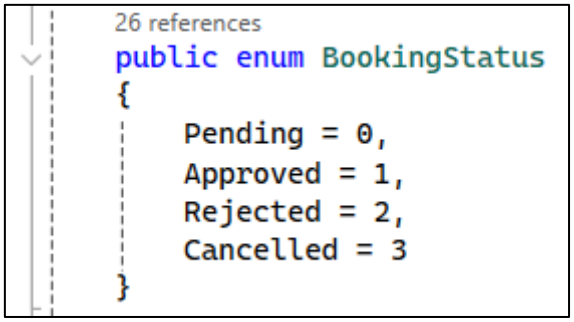
  Purpose: Indicate the request was declined so the slot is not reserved.

  How it works: Can be deleted (per your API rule) to keep lists clean; does not block the room.

- **Cancelled = 3**

  Purpose: Record that the requester withdrew the booking after submission/approval.

  How it works: Frees the room slot. Kept for history/reporting.



- **Booking.cs**

  Purpose: Represent a single room reservation request/record in the system.

  How it works: Maps to table bookings. Includes who requested, what room, time window, org snapshot, status, and server-side validation (min duration, end after start).

  - **BookingId (int)**

    Purpose: Unique identifier for each booking.

    How it works: [Key] primary key; generated by the database and used in API routes (Get/Update/Delete).

  - **RequestedByUserId (int)**

    Purpose: Identify the employee who submitted the booking.

    How it works: Required FK to aspnetusers(Id) (int). Used for filtering (e.g., "My bookings") and audit trails.

  - **TargetUserId (int?)**

    Purpose: Support "book on behalf of" scenarios.

    How it works: Optional FK to aspnetusers(Id). If null, the request is for the requester themselves.

  - **DepartmentId (int?)**

    Purpose: Capture the requester's department at submission time for reporting.

    How it works: Snapshot FK to departments(DepartmentId). Not auto-updated later, preserving history.

  - **CompanyId (int?)**

    Purpose: Capture the requester's company at submission time for reporting.

    How it works: Snapshot FK to companies(CompanyId). Remains as-is even if org changes later.

  - **RoomId (int)**

    Purpose: Specify which room is being reserved.

    How it works: Required FK to rooms(RoomId). Used by overlap checks and calendar rendering.

- **Title (string, max 150)**

  Purpose: Provide a short, readable name for the booking.

  How it works: Required. Shown in lists and calendar cells (e.g., “Weekly Sync”).

- **Purpose (string?, max 300)**

  Purpose: Explain the reason/context of the booking.

  How it works: Optional note to help admins/users understand the meeting’s intent.

- **StartUtc (DateTime)**

  Purpose: Define when the booking begins.

  How it works: Required UTC timestamp; used in overlap detection and filtering.

- **EndUtc (DateTime)**

  Purpose: Define when the booking ends.

  How it works: Required UTC timestamp; must be strictly after StartUtc. Also used in overlap detection.

- **Note (string?, max 500)**

  Purpose: Store any extra remarks or instructions (e.g., “Need projector”).

  How it works: Optional free-text displayed in detail views.

- **CreatedUtc (DateTime)**

  Purpose: Record when the booking was first created (audit).

  How it works: Required; defaults to DateTime.UtcNow on server.

- **LastUpdatedUtc (DateTime)**

  Purpose: Track the most recent modification time (audit).

  How it works: Required; updated on edits to help with troubleshooting and reports.

- **CurrentStatus (BookingStatus)**

  Purpose: Represent the booking’s current lifecycle state.

  How it works: Required; defaults to Pending. Drives UI/permission rules (e.g., deletable only if Pending/Rejected).

- **Validation (IValidatableObject.Validate)**

  Purpose: Prevent invalid times and impractical durations.

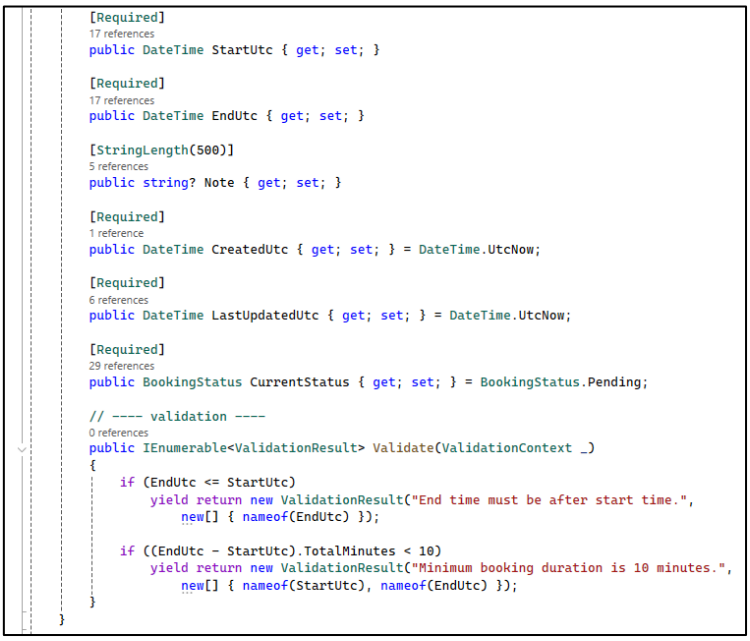
  How it works:

  - Throws validation error if EndUtc <= StartUtc.

- Throws validation error if duration < 10 minutes.

  These messages surface to the client for correction.

```
[Table("bookings")]
2 references
public class Booking : IValidatableObject
{
    [Key]
    [Column("BookingId")]
    22 references
    public int BookingId { get; set; }

    /// <summary>FK → aspnetusers(Id) (int)</summary>
    [Required]
    4 references
    public int RequestedByUserId { get; set; }

    /// <summary>If booking on behalf of someone else; else null.</summary>
    2 references
    public int? TargetUserId { get; set; }

    /// <summary>Snapshot of org at submission time.</summary>
    2 references
    public int? DepartmentId { get; set; }   // FK → departments(DepartmentId)
    2 references
    public int? CompanyId { get; set; }      // FK → companies(CompanyId)

    /// <summary>Room being booked.</summary>
    [Required]
    12 references
    public int RoomId { get; set; }          // FK → rooms(RoomId)

    [Required, StringLength(150)]
    5 references
    public string Title { get; set; } = string.Empty;

    [StringLength(300)]
    4 references
    public string? Purpose { get; set; }
```



```
    [Required]
    17 references
    public DateTime StartUtc { get; set; }

    [Required]
    17 references
    public DateTime EndUtc { get; set; }

    [StringLength(500)]
    5 references
    public string? Note { get; set; }

    [Required]
    1 reference
    public DateTime CreatedUtc { get; set; } = DateTime.UtcNow;

    [Required]
    6 references
    public DateTime LastUpdatedUtc { get; set; } = DateTime.UtcNow;

    [Required]
    29 references
    public BookingStatus CurrentStatus { get; set; } = BookingStatus.Pending;

    // ---- validation ----
    0 references
    public IEnumerable<ValidationResult> Validate(ValidationContext _)
    {
        if (EndUtc <= StartUtc)
            yield return new ValidationResult("End time must be after start time.",
                new[] { nameof(EndUtc) });

        if ((EndUtc - StartUtc).TotalMinutes < 10)
            yield return new ValidationResult("Minimum booking duration is 10 minutes.",
                new[] { nameof(StartUtc), nameof(EndUtc) });
    }
}
```

- **Room.cs**

  Purpose: Represent a physical/virtual room that can be reserved.

  How it works: Maps to table rooms. Admins manage the list; inactive rooms are hidden from booking forms but retained for historical integrity.

- **RoomId (int)**

  Purpose: Unique identifier for each room.

  How it works: [Key] primary key; referenced by Booking_RoomId.

- **RoomName (string, max 120)**

  Purpose: Human-friendly name used in dropdowns and calendar labels.

  How it works: Required; should be unique or clearly distinguishable (e.g., “Board Room L7”).

- **LocationCode (string?, max 40)**

  Purpose: Indicate where the room is located (building/floor/wing).

  How it works: Optional short code (e.g., “L7-EAST”). Helpful for filtering and signage.

- **Capacity (int?)**

  Purpose: Suggest the maximum number of attendees the room can accommodate.

  How it works: Optional numeric value; can be used for future capacity-based filtering.

- **IsActive (bool)**

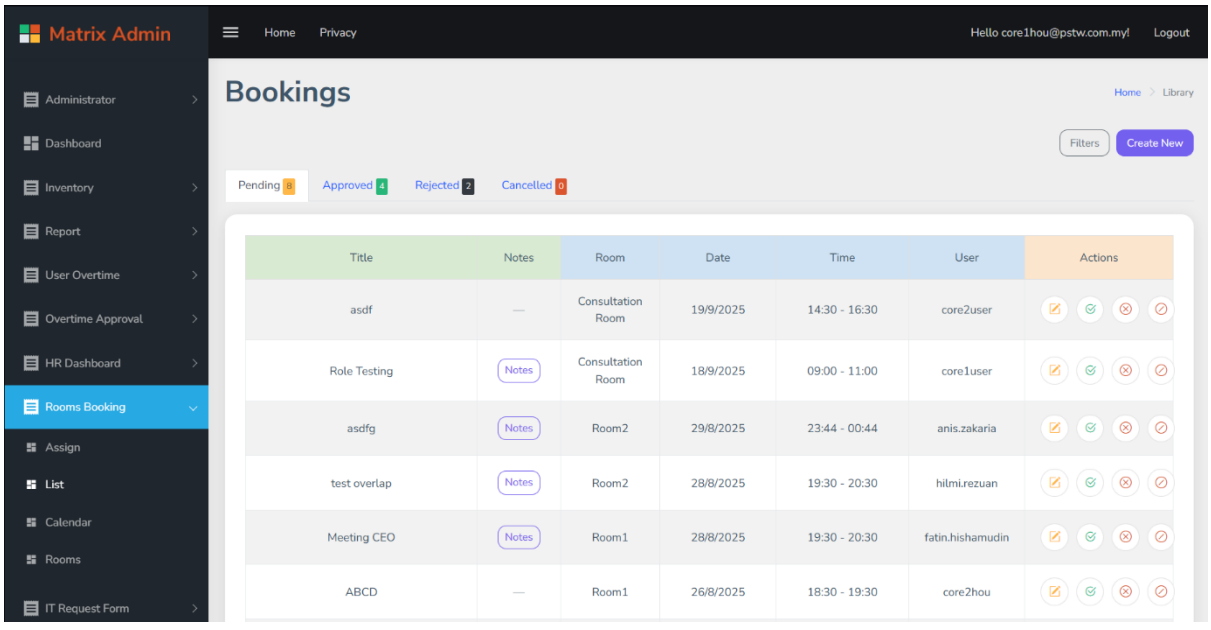
  Purpose: Control whether the room is available for new bookings.

  How it works: Defaults to true. Setting to false soft-deactivates the room (kept for history; hidden from user selection).

```
[Table("rooms")]
2 references
public class Room
{
    7 references
    [Key] public int RoomId { get; set; }

    [Required, StringLength(120)]
    7 references
    public string RoomName { get; set; } = string.Empty;

    [StringLength(40)]
    3 references
    public string? LocationCode { get; set; }

    3 references
    public int? Capacity { get; set; }
    6 references
    public bool IsActive { get; set; } = true;
}
}
```

VIEWS & BOOKINGS API

- **Bookings.cs**
  - **Index.cshtml**



This file (Index.cshtml) provides a dedicated web page for a user to **edit, cancel, or uncancel their own bookings** and the admin can **edit, cancel, uncancel, approve, and reject all bookings**. It displays all the bookings that has been established along with the parameters of each booking. The users are also able to filter bookings for better search purposes.

- ** … **
  - Purpose:
    Main container controlled by Vue. All filters, table rows, pager, and alerts inside this div are driven by the Vue instance mounted at the end of the file.

- **const app = Vue.createApp({ … })**
  - **Purpose:**Initializes the Vue application and defines data, watch, computed, methods, and mounted lifecycle logic that power the page.

- **data() (state)**
  - **Constants / refs**
    - api: Base URL to the Bookings API.
  - **Lookups / maps**
    - users, rooms: Raw arrays from ?lookups=1.
    - userMap, roomMap: Quick-lookup maps → id → display string used to show readable names in the table even if the list API only returns IDs.
  - **Filters state**
    - showFilters: Toggles the visibility of the filters panel.
    - fltFrom, fltTo: Date range (local date inputs) used to build from/to UTC ISO params for the API.
    - fltUser, fltCompany, fltDepartment: Client-side refiners.
      (Company/Department are shown but may be inactive until API returns those fields.)
  - **Table & UI**
    - rows: Full list returned by the API.
    - isLoading: Shows loading state for the table.
    - alert: { type, msg } used to render Bootstrap alerts in the #alerts container.
  - **Pagination**
    - currentPage: 1-based pagination index.
    - pageSize: Items per page (synced with the “Items per page” select).

```
data() {
    return {
        ctx: window.__ctx || { isManager: false, meId: 0 },
        api: `${window.location.origin}/api/BookingsApi`,

        users: [], rooms: [],
        userMap: new Map(), roomMap: new Map(),

        showFilters: false,
        fltFrom: "", fltTo: "", fltUser: "",

        activeStatus: "Pending",
        statuses: ["Pending", "Approved", "Rejected", "Cancelled"],

        rows: [],
        isLoading: false,
        alert: { type: "", msg: "" },

        currentPage: 1,
        pageSize: 10,

        notesById: new Map()
    };
},
```

- **watch property**
  - Purpose:
    Reacts to changes in reactive data with side effects.
  - **Implemented watcher**
    - showFilters(): Syncs the filters card DOM (#filtersCard) with Vue state. This keeps your existing static markup while controlling it from Vue.

```
watch: {
    showFilters() {
        const card = document.getElementById("filtersCard");
        if (card) card.style.display = this.showFilters ? "block" : "none";
    }
},
```

- **computed properties**
  - **Purpose:**
    Derived values that update automatically when dependencies change, without storing duplicate state.
  - **Key computed props**
    - **queryParams**
      Builds the query string for the API.
      - Always requests a large pageSize=500 from server (the page itself does client-side paging).
      - Converts local fltFrom / fltTo dates to UTC ISO boundaries via helpers.

- **filteredRows**
  Applies client-side refinement for fltUser, fltCompany, fltDepartment (until server supports them). It normalizes possible casing/shape differences (e.g., RequestedByUserId vs requestedByUserId).
- **totalPages**
  ceil(filteredRows.length / pageSize), with a minimum of 1.
- **pagedRows**
  Slices filteredRows for the active page.

```
computed: {
    queryParams() {
        const p = new URLSearchParams();
        p.set("pageSize", "500");

        const fromIso = this.localDateToUtcIsoStart(this.fltFrom);
        const toIso = this.localDateToUtcIsoEnd(this.fltTo);
        if (fromIso) p.set("from", fromIso);
        if (toIso) p.set("to", toIso);

        if (this.ctx.isManager && this.fltUser) p.set("userId", String(this.fltUser));

        return p.toString();
    },

    statusCounts() {
        const counts = { Pending: 0, Approved: 0, Rejected: 0, Cancelled: 0 };
        for (const r of this.rows) {
            const s = (r.status ?? r.Status ?? "").toString();
            if (counts[s] != null) counts[s]++;
        }
        return counts;
    },

    filteredRows() {
        const list = this.rows.filter(r => {
            const st = (r.status ?? r.Status ?? "").toString();
            if (st !== this.activeStatus) return false;
            if (this.ctx.isManager && this.fltUser) {
                const uid = Number(r.requestedByUserId ?? r.RequestedByUserId ?? r.userId ?? 0);
                if (String(uid) !== String(this.fltUser)) return false;
            }
            return true;
        });

        return list.slice().sort((a, b) => {
            const aC = this.getTimeNum(a.createdUtc ?? a.CreatedUtc);
            const bC = this.getTimeNum(b.createdUtc ?? b.CreatedUtc);
            if (aC !== bC) return (bC ?? -Infinity) - (aC ?? -Infinity);

            const aS = this.getTimeNum(a.startUtc ?? a.StartUtc);
            const bS = this.getTimeNum(b.startUtc ?? b.StartUtc);
            if (aS !== bS) return (bS ?? -Infinity) - (aS ?? -Infinity);

            const aId = this.getNum(a.id ?? a.Id ?? a.bookingId ?? a.BookingId);
            const bId = this.getNum(b.id ?? b.Id ?? b.bookingId ?? b.BookingId);
            return (bId ?? -Infinity) - (aId ?? -Infinity);
        });
    },

    totalPages() { return Math.max(1, Math.ceil(this.filteredRows.length / this.pageSize)); },

    pagedRows() {
        const start = (this.currentPage - 1) * this.pageSize;
        const end = start + this.pageSize;
        return this.filteredRows.slice(start, end);
    }
},
```

- **methods property**
  - **Purpose:**
    Functions that perform actions in response to user interaction or lifecycle events.

UI helpers

- **setAlert(type, msg)**
  Renders a Bootstrap alert into #alerts. Useful for action confirmations and errors.
- **escapeHtml(s)**
  Prevents HTML injection when writing dynamic text into the DOM.

Date helpers

- **formatLocal(s)**
  Formats an ISO string to YYYY-MM-DD HH:mm in the browser's local time.
- **localDateToUtcIsoStart(dateStr) / localDateToUtcIsoEnd(dateStr)**
  Convert a local date to the UTC ISO start/end of day. Ensures server filters the intended local day.

Lookups

- **loadLookups()**
  Calls GET /api/BookingsApi?lookups=1.
  - Populates users, rooms.
  - Rebuilds the User <select> options.
  - Builds userMap and roomMap so the table can show names even if the list lacks them.
  - Defensive against differing shapes: supports Id/id, UserName/userName, RoomId/roomId, Name/RoomName, etc.

Data load

- **loadList()**
  Fetches bookings using queryParams.
  - Shows a "Loading…" row while fetching.
  - **On success:**
    - Sets rows to the returned array.
    - Ensures currentPage is still within totalPages.
    - Renders only the current page (pagedRows) into <tbody> with:
      - Friendly Room name via roomMap (fallback to roomName or Room #id).
      - Friendly User name via userMap (fallback to userName or User #id).
      - Local time formatting for startUtc / endUtc.

        - Status badge coloring (Approved/Rejected/Cancelled/Pending).
        - Action buttons: Edit, Approve, Reject, Cancel/Un-cancel.
    - Calls updatePager() to sync pager texts/buttons.
- **On error:**
    - renders an error row and shows an alert.

Actions (POST to API)

- **Common helper:** postAction(action, id, opts)
    - Sends a POST to /api/BookingsApi?action={approve|reject|cancel}&id={id}[&undo=1].
    - Accepts optional JSON body for future extensibility.
- **handleApprove(id, btn) / handleReject(id, btn)**
  Disable the clicked button, post, show alert, and refresh the list.
- **handleToggleCancel(id, status, btn)**
  If current status is "Cancelled", posts with undo=1; otherwise performs a cancel. Refreshes list afterwards.

Filter controls

- **applyFilters()**
  Resets to page 1 and reloads the list (server date range + client refiners).
- **clearFilters()**
  Clears all filter states, resets DOM inputs for visual sync, resets to page 1, reloads.
- **toggleFilters()**
  Toggles the filters card.

Pagination helpers

- **updatePager()**
  Updates: "Page N of M", "(Showing X–Y of Z)", and enables/disables Prev/Next.
- **prevPage() / nextPage()**
  Navigates pages and reloads.
- **setPageSize(n)**
  Changes items per page, resets to first page, and reloads.

```
methods: {
    // ===== tabs & alerts =====
    setStatus(s) {
        if (this.activeStatus !== s) {
            this.activeStatus = s;
            this.currentPage = 1;
            this.renderTable();
        }
    },
    setAlert(type, msg) {
        this.alert = { type, msg };
        const wrap = document.getElementById("alerts");
        if (!wrap) return;
        wrap.innerHTML = `
            <div class="alert alert-${type} alert-dismissible fade show" role="alert">
              ${this.escapeHtml(msg)}
              <button type="button" class="btn-close" data-bs-dismiss="alert"></button>
            </div>`;
    },
    escapeHtml(s) {
        return String(s ?? "").replace(/[&<>\"']/g, m => ({
            "&": "&amp;", "<": "&lt;", ">": "&gt;", '"': "&quot;", "'": "&#39;"
        }[m]));
    },

    // ===== time & formatting helpers =====
    getTimeNum(v) { if (!v) return null; const t = new Date(v).getTime(); return Number.isNaN(t) ? null : t; },
    getNum(v) { const n = Number(v); return Number.isNaN(n) ? null : n; },

    /* e.g., 7/9/2025 (no leading zeros) */
    formatDateDMY(s) {
        if (!s) return "";
        const d = new Date(s); if (isNaN(d)) return this.escapeHtml(s);
        return `${d.getDate()}/${d.getMonth() + 1}/${d.getFullYear()}`;
    },
    /* e.g., 08:05 (zero-padded) */
    formatTimeHM(s) {
        if (!s) return "";
        const d = new Date(s); if (isNaN(d)) return "";
        const pad = n => String(n).padStart(2, "0");
        return `${pad(d.getHours())}:${pad(d.getMinutes())}`;
    },

    localDateToUtcIsoStart(dateStr) {
        if (!dateStr) return null;
        const d = new Date(dateStr + "T00:00");
        const utc = new Date(d.getTime() - d.getTimezoneOffset() * 60000);
        return utc.toISOString();
    },
    localDateToUtcIsoEnd(dateStr) {
        if (!dateStr) return null;
        const d = new Date(dateStr + "T23:59:59.999");
        const utc = new Date(d.getTime() - d.getTimezoneOffset() * 60000);
        return utc.toISOString();
    },
```

```
async loadLookups() {
    try {
        const res = await fetch(`${this.api}?lookups=1`);
        if (!res.ok) return;
        const { rooms = [], users = [] } = await res.json();

        this.users = users; this.rooms = rooms;

        const sel = document.getElementById("fltUser");
        if (sel) {
            const prev = sel.value ?? "";
            sel.innerHTML = '<option value="">All users</option>';
            users.forEach(u => {
                const id = Number(u.Id ?? u.id); if (!id) return;
                const name = u.UserName ?? u.userName ?? "";
                const opt = document.createElement("option");
                opt.value = String(id); opt.textContent = name;
                sel.appendChild(opt);
            });
            if (prev && [...sel.options].some(o => o.value === prev)) { sel.value = prev; this.fltUser = prev; }
        }

        this.userMap = new Map(users.map(u => [Number(u.Id ?? u.id), (u.UserName ?? u.userName ?? "")]));
        this.roomMap = new Map(rooms.map(r => {
            const id = Number(r.roomId ?? r.RoomId);
            const name = r.roomName ?? r.Name ?? r.RoomName ?? `Room ${id}`;
            return [id, name];
        }));
    } catch { }
},
```

```
        async loadList() {
            const $tbody = document.querySelector("#bookingsTable tbody");
            const cols = this.ctx.isManager ? 7 : 6;
            if ($tbody) $tbody.innerHTML = `<tr><td colspan="${cols}" class="text-center">Loading…</td></tr>`;
            this.isLoading = true;

            try {
                const res = await fetch(`${this.api}?${this.queryParams}`);
                if (!res.ok) throw new Error(await res.text());
                const data = await res.json();
                this.rows = Array.isArray(data) ? data : [];
                this.currentPage = 1;
                this.renderTable();
            } catch (err) {
                if ($tbody) $tbody.innerHTML = `<tr><td colspan="${cols}" class="text-danger text-center">Failed to load:
 ${this.escapeHtml(err.message)}</td></tr>`;
                this.setAlert("danger", err.message || "Failed to load.");
            } finally {
                this.isLoading = false;
            }
        },
```

```
renderTable() {
    const $tbody = document.querySelector("#bookingsTable tbody");
    if (!$tbody) return;

    const visible = this.pedRowsSafe();
    const cols = this.ctx.isManager ? 7 : 6;

    if (!visible.length) {
        $tbody.innerHTML = `<tr><td colspan="${cols}" class="text-center">No bookings found.</td></tr>`;
        this.updatePager();
        return;
    }

    $tbody.innerHTML = "";
    const showModeration = this.ctx.isManager && this.activeStatus === "Pending";
    this.notesById = new Map();

    for (const b of visible) {
        const id = b.id ?? b.Id ?? b.bookingId ?? b.BookingId;
        const title = b.title ?? "";
        const note = (b.note ?? b.description ?? "").toString();
        const hasNote = note.trim().length > 0;
        if (id != null) this.notesById.set(String(id), note);

        const start = b.startUtc ?? b.StartUtc;
        const end = b.endUtc ?? b.EndUtc;

        const dateDisp = this.formatDateDMY(start);
        const timeStart = this.formatTimeHM(start);
        const timeEnd = this.formatTimeHM(end);
        const timeDisp = (timeStart || timeEnd) ? `${timeStart || "—"} - ${timeEnd || "—"}` : "—";

        const ridRaw = (b.roomId ?? b.RoomId ?? null);
        const roomId = ridRaw == null ? null : Number(ridRaw);
        const roomDisp = (roomId && !Number.isNaN(roomId))
            ? (this.roomMap.get(roomId) ?? `Room #${roomId}`)
            : (b.roomName ?? "(unknown)");

        const uidRaw = (b.requestedByUserId ?? b.RequestedByUserId ?? null);
        const uid = uidRaw == null ? null : Number(uidRaw);
        const userDisp = (uid && !Number.isNaN(uid))
            ? (this.userMap.get(uid) ?? `User #${uid}`)
            : (b.userName ?? "(unknown)");

        const status = (b.status ?? b.Status ?? "").toString();
        const isCancelled = status === "Cancelled";
        const isApproved = status === "Approved";
        const isRejected = status === "Rejected";

        const canEdit = !(isApproved || isRejected || isCancelled);
        const canCancel = !isRejected;

        const cancelLabel = isCancelled ? "Un-cancel" : "Cancel";
        const approveDisabled = (isApproved || isCancelled) ? "disabled" : "";
        const rejectDisabled = (isRejected || isCancelled) ? "disabled" : "";
```

```
async postAction(action, id, opts = {}) {
    const url = new URL(`${this.api}`);
    url.searchParams.set("action", action);
    url.searchParams.set("id", id);
    if (opts.undo) url.searchParams.set("undo", "1");

    const res = await fetch(url.toString(), {
        method: "POST",
        headers: opts.body ? { "Content-Type": "application/json" } : undefined,
        body: opts.body ? JSON.stringify(opts.body) : undefined
    });

    const text = await res.text();
    if (!res.ok) throw new Error(text || `Failed (${res.status})`);
    try { return JSON.parse(text); } catch { return {}; }
},
async handleApprove(id, btn) {
    btn && (btn.disabled = true);
    try { await this.postAction("approve", id); this.setAlert("success", "Booking approved."); await this.loadList(); }
    catch (e) { this.setAlert("danger", e.message || "Approve failed."); }
    finally { btn && (btn.disabled = false); }
},
async handleReject(id, btn) {
    btn && (btn.disabled = true);
    try { await this.postAction("reject", id); this.setAlert("success", "Booking rejected."); await this.loadList(); }
    catch (e) { this.setAlert("danger", e.message || "Reject failed."); }
    finally { btn && (btn.disabled = false); }
},
async handleToggleCancel(id, status, btn) {
    btn && (btn.disabled = true);
    const undo = String(status || "").toLowerCase() === "cancelled";
    try { await this.postAction("cancel", id, { undo }); await this.loadList(); }
    catch (e) { this.setAlert("danger", e.message || "Cancel action failed."); }
    finally { btn && (btn.disabled = false); }
},
```

```
        showNoteCard(id) {
            const title = (this.filteredRows.find(r => String(r.id ?? r.Id ?? r.bookingId ?? r.BookingId) === String(id))?.title ??
"") + "";
            const note = this.notesById.get(String(id)) ?? "";

            const overlay = document.getElementById("noteOverlay");
            if (!overlay) return;

            const safeTitle = this.escapeHtml(title || "Notes");
            const safeNote = this.escapeHtml(note || "(No notes)");

            overlay.innerHTML = `
                <div class="note-card" role="dialog" aria-modal="true">
                    <div class="d-flex justify-content-between align-items-center mb-1">
                        <h6 class="mb-0">${safeTitle}</h6>
                        <button type="button" class="btn btn-sm btn-outline-secondary" data-note-close>Close</button>
                    </div>
                    <div class="note-body">${safeNote.replace(/\\n/g, '<br/>')}</div>
                </div>
            `;
            overlay.style.display = "flex";

            const closeAll = () => { overlay.style.display = "none"; overlay.innerHTML = ""; };
            overlay.onclick = (e) => { if (e.target === overlay) closeAll(); };
            overlay.querySelector("[data-note-close]")?.addEventListener("click", closeAll);
            document.addEventListener("keydown", function esc(e) { if (e.key === "Escape") { closeAll();
document.removeEventListener("keydown", esc); } });
        },

        // ===== filters & pager =====
        bindFilterInputs() {
            const bind = (id, setter) => {
                const el = document.getElementById(id); if (!el) return;
                const h = e => { this[setter] = e.target.value ?? ""; };
                el.addEventListener("change", h); el.addEventListener("input", h);
            };
            bind("fltFrom", "fltFrom"); bind("fltTo", "fltTo"); bind("fltUser", "fltUser");
        },
        syncFiltersFromDom() {
            const get = id => document.getElementById(id)?.value ?? "";
            this.fltFrom = get("fltFrom"); this.fltTo = get("fltTo"); this.fltUser = get("fltUser");
        },
        applyFilters() { this.syncFiltersFromDom(); this.currentPage = 1; this.loadList(); },
        clearFilters() {
            this.fltFrom = this.fltTo = this.fltUser = "";
            ["fltFrom", "fltTo", "fltUser"].forEach(id => { const el = document.getElementById(id); if (el) el.value = ""; });
            this.currentPage = 1; this.loadList();
        },
        toggleFilters() { this.showFilters = !this.showFilters; },
```

```
updatePager() {
    const total = this.filteredRows.length;
    const totalPages = this.totalPages;
    const pageInfo = document.getElementById("pageInfo");
    const rangeInfo = document.getElementById("rangeInfo");
    const btnPrev = document.getElementById("btnPrev");
    const btnNext = document.getElementById("btnNext");

    if (pageInfo) pageInfo.textContent = `Page ${this.currentPage} of ${totalPages}`;
    const startIdx = total ? (this.currentPage - 1) * this.pageSize + 1 : 0;
    const endIdx = Math.min(this.currentPage * this.pageSize, total);
    if (rangeInfo) rangeInfo.textContent = `(Showing ${startIdx}-${endIdx} of ${total})`;

    if (btnPrev) btnPrev.disabled = this.currentPage <= 1;
    if (btnNext) btnNext.disabled = this.currentPage >= totalPages;
},
prevPage() { if (this.currentPage > 1) { this.currentPage--; this.renderTable(); this.updatePager(); } },
nextPage() { if (this.currentPage < this.totalPages) { this.currentPage++; this.renderTable(); this.updatePager(); } },
setPageSize(n) {
    const newSize = Number(n) || 10;
    if (newSize !== this.pageSize) {
        this.pageSize = newSize; this.currentPage = 1; this.renderTable(); this.updatePager();
    }
}
```

- **mounted() lifecycle**

  - Purpose:
    Runs once after Vue mounts, to wire DOM events and start loading data.

  - What it does

    1. Reads initial values from the filters inputs (#fltFrom, #fltTo, etc.) into Vue state.

    2. Attaches click handlers to:

       - **Filters:** toggle/apply/clear.

       - **Pager:** prev/next and page size select.

3. Delegates row action clicks at the document level for:
    - [data-action="approve"], "reject", "toggle-cancel".
4. Boots data: loadLookups().then(() => loadList()).

```
mounted() {
    this.fltFrom = document.getElementById("fltFrom")?.value ?? "";
    this.fltTo = document.getElementById("fltTo")?.value ?? "";
    this.fltUser = document.getElementById("fltUser")?.value ?? "";

    this.bindFilterInputs();

    document.getElementById("btnToggleFilters")?.addEventListener("click", this.toggleFilters);
    document.getElementById("btnApply")?.addEventListener("click", this.applyFilters);
    document.getElementById("btnClear")?.addEventListener("click", this.clearFilters);
    document.getElementById("btnPrev")?.addEventListener("click", this.prevPage);
    document.getElementById("btnNext")?.addEventListener("click", this.nextPage);
    const selPageSize = document.getElementById("selPageSize");
    if (selPageSize) {
        selPageSize.value = String(this.pageSize);
        selPageSize.addEventListener("change", e => this.setPageSize(e.target.value));
    }

    const table = document.getElementById("bookingsTable");
    if (table) {
        table.addEventListener("click", (e) => {
            const el = e.target instanceof Element ? e.target : null; if (!el) return;
            const btn = el.closest("[data-action]"); if (!btn) return;

            const action = btn.getAttribute("data-action");
            const id = btn.getAttribute("data-id");
            if (!action || !id) return;

            if (action === "approve") return this.handleApprove(id, btn);
            if (action === "reject") return this.handleReject(id, btn);
            if (action === "toggle-cancel") {
                const status = btn.getAttribute("data-status") || "";
                return this.handleToggleCancel(id, status, btn);
            }
            if (action === "show-note") { return this.showNoteCard(id); }
        });
    }

    document.addEventListener("click", (e) => {
        const el = e.target instanceof Element ? e.target : null; if (!el) return;
        const btn = el.closest("[data-action]"); if (!btn) return;

        const action = btn.getAttribute("data-action");
        const id = btn.getAttribute("data-id");
        if (!action || !id) return;

        if (action === "approve") return this.handleApprove(id, btn);
        if (action === "reject") return this.handleReject(id, btn);
        if (action === "toggle-cancel") {
            const status = btn.getAttribute("data-status") || "";
            return this.handleToggleCancel(id, status, btn);
        }
        if (action === "edit") {
            const url = '@Url.Action("Create", "Bookings", new { area = "Bookings" })' + `?id=${encodeURIComponent(id)}`;
            return window.location.assign(url);
        }

        if (action === "show-note") { return this.showNoteCard(id); }
    });

    this.loadLookups().then(() => this.loadList());
}
```

Table & Pager Markup

- **Table**
  Fixed columns with semantic headers (Title, Room, Start/End in local, Notes, User, Actions). The first row shows *No bookings found* until data loads.
- **Pager**
  Bootstrap-styled Prev/Next buttons, a live “Page N of M”, and a “Showing X–Y of Z”. Page size select (5/10/20/50) is synced to pageSize.

Styling Highlights

- cards & table: rounded corners and subtle shadow for modern look.
- header color bands: gentle grouping (green/blue/orange).

- filters layout: responsive CSS grid (6 → 3 → 2 columns).
- actions: white-space: nowrap to keep buttons aligned.

- **BookingsAPI(Index.cshtml)**

- **[HttpGet] ("Index data + lookup pack")**

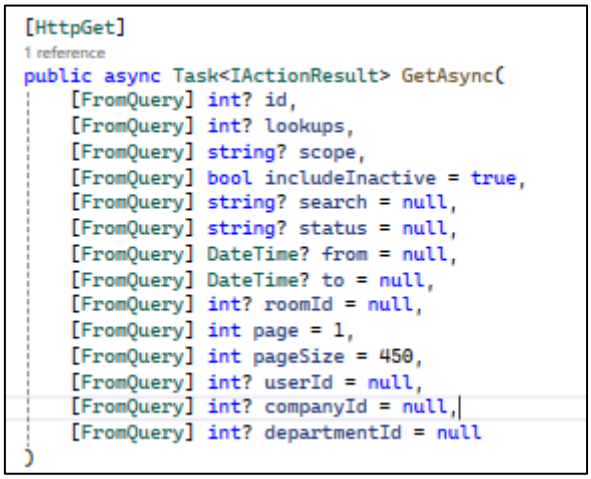
- **Purpose:** Powers the Bookings Index page. When lookups=1, it returns the dropdown data the page needs (rooms and users).
- **Key functionality:**
  - **Auth/roles:** Checks if the caller is a manager with IsManager(). Managers see all users. Non-managers only see themselves.
  - **Inputs (query):** Accepts many filters (id, scope, search, status, from, to, roomId, page, pageSize, userId, companyId, departmentId, includeInactive). In this branch, it only cares about lookups=1.
  - **Rooms lookup:** Pulls active rooms only (IsActive = true), sorts by RoomName, and returns { roomId, roomName }.
  - **Users lookup:**
    - If manager: returns all users sorted by FullName with { Id, UserName = FullName, Email }.
    - If not manager: resolves the current user id and returns only that single user. If the user id cannot be resolved, returns an empty list.
  - **Response shape:** 200 OK with { rooms: [...], users: [...] }.
  - **Edge handling:**
    - If there are no active rooms, rooms is an empty list so the UI can still render gracefully.
    - If the caller is not a manager and the current user cannot be identified, users is an empty list to avoid leaking other accounts.

```
[HttpGet]
1 reference
public async Task<IActionResult> GetAsync(
    [FromQuery] int? id,
    [FromQuery] int? lookups,
    [FromQuery] string? scope,
    [FromQuery] bool includeInactive = true,
    [FromQuery] string? search = null,
    [FromQuery] string? status = null,
    [FromQuery] DateTime? from = null,
    [FromQuery] DateTime? to = null,
    [FromQuery] int? roomId = null,
    [FromQuery] int page = 1,
    [FromQuery] int pageSize = 450,
    [FromQuery] int? userId = null,
    [FromQuery] int? companyId = null,
    [FromQuery] int? departmentId = null
)
```

```
if (lookups == 1)
{
    var rooms = await _db.Rooms
        .AsNoTracking()
        .Where(r => r.IsActive)
        .OrderBy(r => r.RoomName)
        .Select(r => new { roomId = r.RoomId, roomName = r.RoomName })
        .ToListAsync();

    object usersPayload;
    if (IsManager())
    {
        usersPayload = await _db.Users
            .AsNoTracking()
            .OrderBy(u => u.FullName)
            .Select(u => new { u.Id, UserName = u.FullName, u.Email })
            .ToListAsync();
    }
    else
    {
        var me = GetCurrentUserId();
        usersPayload = (me is int myId)
            ? await _db.Users.AsNoTracking()
                .Where(u => u.Id == myId)
                .Select(u => new { u.Id, UserName = u.FullName, u.Email })
                .ToListAsync()
            : new object[0];
    }

    return Ok(new { rooms, users = usersPayload });
}
```

- **[HttpGet] ("Index bookings list")**
- **Purpose:** Returns the paged list of bookings for the Index table. This is the main data feed your Razor page uses after the lookup pack has loaded.
- **Key functionality:**
    - **Auth and roles:** Uses IsManager() to decide visibility. Managers can see everything that matches the filters, while non-managers are restricted to bookings where they are either the requester or the target user. If the current user cannot be resolved for a non-manager, the action returns 401 Unauthorized.
    - **Inputs (query):** Accepts search, status, from, to, roomId, userId, companyId, departmentId, page, and pageSize. It also accepts id, scope, and includeInactive, but in this branch those are not used.

- **Text search:** If search is provided, filters on Title.Contains(search) to do a simple substring match on titles.
- **Status filter:** If status can be parsed into your BookingStatus enum, the query is narrowed to CurrentStatus == parsedStatus. Invalid values are ignored quietly so the grid still loads.
- **Date range filter:** Converts from and to to UTC using CoerceToUtc. Keeps bookings that overlap the window: EndUtc > fromUtc and StartUtc < toUtc. This means bookings that touch the range edges are included.
- **Entity filters:**
  - roomId: matches exact room.
  - userId: matches if the user is either RequestedByUserId or TargetUserId.
  - companyId and departmentId: exact matches on their respective columns.
- **RBAC narrowing:** For non-managers, the query is further tightened to the current user (RequestedByUserId == me or TargetUserId == me). If the current user id cannot be read, the action returns 401.
- **Paging and sort:** Sorts by StartUtc ascending, then applies Skip((page - 1) * pageSize) and Take(pageSize). Defaults are page=1 and pageSize=450.
- **Projection and UTC safety:** Projects each row to a lightweight shape for the grid and explicitly tags startUtc and endUtc with DateTimeKind.Utc via DateTime.SpecifyKind(...) to avoid timezone bugs on the client.



```
[HttpGet]
1 reference
public async Task<IActionResult> GetAsync(
    [FromQuery] int? id,
    [FromQuery] int? lookups,
    [FromQuery] string? scope,
    [FromQuery] bool includeInactive = true,
    [FromQuery] string? search = null,
    [FromQuery] string? status = null,
    [FromQuery] DateTime? from = null,
    [FromQuery] DateTime? to = null,
    [FromQuery] int? roomId = null,
    [FromQuery] int page = 1,
    [FromQuery] int pageSize = 450,
    [FromQuery] int? userId = null,
    [FromQuery] int? companyId = null,
    [FromQuery] int? departmentId = null
)
```

```
// BOOKINGS LIST
var q = _db.Bookings.AsNoTracking().AsQueryable();

if (!string.IsNullOrWhiteSpace(search)) q = q.Where(x => x.Title.Contains(search));

if (!string.IsNullOrWhiteSpace(status) && Enum.TryParse<BookingStatus>(status, true, out var st))
    q = q.Where(x => x.CurrentStatus == st);

DateTime? fromUtc = from.HasValue ? CoerceToUtc(from.Value) : null;
DateTime? toUtc = to.HasValue ? CoerceToUtc(to.Value) : null;

if (fromUtc.HasValue) q = q.Where(x => x.EndUtc > fromUtc.Value);
if (toUtc.HasValue) q = q.Where(x => x.StartUtc < toUtc.Value);

if (roomId.HasValue) q = q.Where(x => x.RoomId == roomId.Value);
if (userId.HasValue && userId.Value > 0)
    q = q.Where(x => x.RequestedByUserId == userId.Value || x.TargetUserId == userId.Value);

if (companyId.HasValue && companyId.Value > 0)
    q = q.Where(x => x.CompanyId == companyId.Value);

if (departmentId.HasValue && departmentId.Value > 0)
    q = q.Where(x => x.DepartmentId == departmentId.Value);

// RBAC: non-managers only see their own
var meList = GetCurrentUserId();
if (!IsManager())
{
    if (meList is int myId)
        q = q.Where(x => x.RequestedByUserId == myId || x.TargetUserId == myId);
    else
        return Unauthorized();
}

var items = await q.OrderBy(x => x.StartUtc)
    .Skip((page - 1) * pageSize)
    .Take(pageSize)
    .Select(b => new
    {
        id = b.BookingId,
        roomId = b.RoomId,
        requestedByUserId = b.RequestedByUserId,
        title = b.Title,
        description = b.Purpose,
        startUtc = DateTime.SpecifyKind(b.StartUtc, DateTimeKind.Utc),
        endUtc = DateTime.SpecifyKind(b.EndUtc, DateTimeKind.Utc),
        status = b.CurrentStatus.ToString(),
        note = b.Note
    })
    .ToListAsync();

return Ok(items);
```

- **[HttpPost] (action=approve) — "Approve booking" (Index page)**
- **Purpose:** Lets a manager approve a booking from the Index table actions. The client calls POST ?action=approve&id={bookingId} when the Approve button is clicked.
- **Key functionality:**
    - **Auth and roles:** Only managers can approve. If the caller is not a manager, the action returns 401 Unauthorized with a friendly message.
    - **Inputs (query/body):** Uses action=approve and id in the query string. scope and body are ignored here. If id is missing or invalid, returns 400 Bad Request.
    - **Record lookup:** Fetches the booking by BookingId. If not found, returns 404 Not Found.
    - **Status guards:**
        - If the booking is already **Cancelled**, returns 409 Conflict with “Cannot approve a cancelled booking.”
        - If the booking is already **Approved**, returns 200 OK with the current status so the UI can idempotently refresh.

- **Overlap check:** Before approving, verifies there is no time clash against other **Approved** bookings in the same room. The rule is interval overlap: x.StartUtc < b.EndUtc and b.StartUtc < x.EndUtc. If a clash exists, returns 409 Conflict with a clear message.
- **Approval write:** Sets CurrentStatus = Approved, updates LastUpdatedUtc = UtcNow, saves, and returns 200 OK with { id, status }.

```
[HttpPost]
0 references
public async Task<IActionResult> PostAsync(
    [FromQuery] string? action,
    [FromQuery] int? id,
    [FromQuery] string? scope,
    [FromBody] System.Text.Json.JsonElement? body // nullable is fine
)
{
```

```
// BOOKING: APPROVE (Manager only)
if (string.Equals(action, "approve", StringComparison.OrdinalIgnoreCase))
{
    if (!id.HasValue || id <= 0) return BadRequest("Missing id.");
    if (!IsManager())
        return Unauthorized("Not authorized to approve bookings.");

    var b = await _db.Bookings.FirstOrDefaultAsync(x => x.BookingId == id.Value);
    if (b is null) return NotFound();

    if (b.CurrentStatus == BookingStatus.Cancelled)
        return Conflict("Cannot approve a cancelled booking.");
    if (b.CurrentStatus == BookingStatus.Approved)
        return Ok(new { id = b.BookingId, status = b.CurrentStatus.ToString() });

    // prevent overlap with other APPROVED bookings
    var overlap = await _db.Bookings.AsNoTracking().AnyAsync(x =>
        x.RoomId == b.RoomId &&
        x.BookingId != b.BookingId &&
        x.CurrentStatus == BookingStatus.Approved &&
        x.StartUtc < b.EndUtc && b.StartUtc < x.EndUtc);

    if (overlap) return Conflict("Cannot approve: time slot overlaps an approved booking.");

    b.CurrentStatus = BookingStatus.Approved;
    b.LastUpdatedUtc = DateTime.UtcNow;
    await _db.SaveChangesAsync();
    return Ok(new { id = b.BookingId, status = b.CurrentStatus.ToString() });
}
```

- **[HttpPost] (action=reject) "Reject or un-reject a booking" (Index page)**
- **Purpose:** Lets a manager reject a booking, or undo a rejection back to Pending. The client calls POST ?action=reject&id={bookingId} and can include a JSON body with a note or an undo flag.
- **Key functionality:**
  - **Auth and roles:** Only managers can perform this. Non-managers get 401 Unauthorized.
  - **Inputs (query/body):**
    - Query: id is required and must be > 0.

- Body: optional JSON. If undo is true (checked via ShouldUndo(body)), the action attempts an un-reject. If a note is included, GetBodyNote(body) is used to store it in b.Note on reject.

- **Lookup:** Loads the booking by BookingId. If not found, returns 404 Not Found.
- **Reject flow:**
  - Guard: if the booking is Cancelled, returns 409 Conflict with a clear message.
  - If body has a note, saves it to b.Note.
  - Sets CurrentStatus = Rejected.
- **Un-reject flow (undo):**
  - Only allowed if the current status is Rejected.
  - If it is not Rejected, returns 409 Conflict with "Booking is not rejected."
  - Sets CurrentStatus = Pending.
- **Audit fields:** Updates LastUpdatedUtc = UtcNow before saving.

```
[HttpPost]
0 references
public async Task<IActionResult> PostAsync(
    [FromQuery] string? action,
    [FromQuery] int? id,
    [FromQuery] string? scope,
    [FromBody] System.Text.Json.JsonElement? body // nullable is fine
)
{
```

```
// BOOKING: REJECT / UN-REJECT (Manager only; undo -> Pending)
if (string.Equals(action, "reject", StringComparison.OrdinalIgnoreCase))
{
    if (!id.HasValue || id <= 0) return BadRequest("Missing id.");
    if (!IsManager())
        return Unauthorized("Not authorized to reject bookings.");

    var b = await _db.Bookings.FirstOrDefaultAsync(x => x.BookingId == id.Value);
    if (b is null) return NotFound();

    bool undo = ShouldUndo(body);

    if (!undo)
    {
        if (b.CurrentStatus == BookingStatus.Cancelled)
            return Conflict("Cannot reject a cancelled booking.");
        if (body.HasValue && body.Value.ValueKind == JsonValueKind.Object)
            b.Note = GetBodyNote(body.Value);
        b.CurrentStatus = BookingStatus.Rejected;
    }
    else
    {
        if (b.CurrentStatus != BookingStatus.Rejected)
            return Conflict("Booking is not rejected.");
        b.CurrentStatus = BookingStatus.Pending;
    }

    b.LastUpdatedUtc = DateTime.UtcNow;
    await _db.SaveChangesAsync();
    return Ok(new { id = b.BookingId, status = b.CurrentStatus.ToString() });
}
```

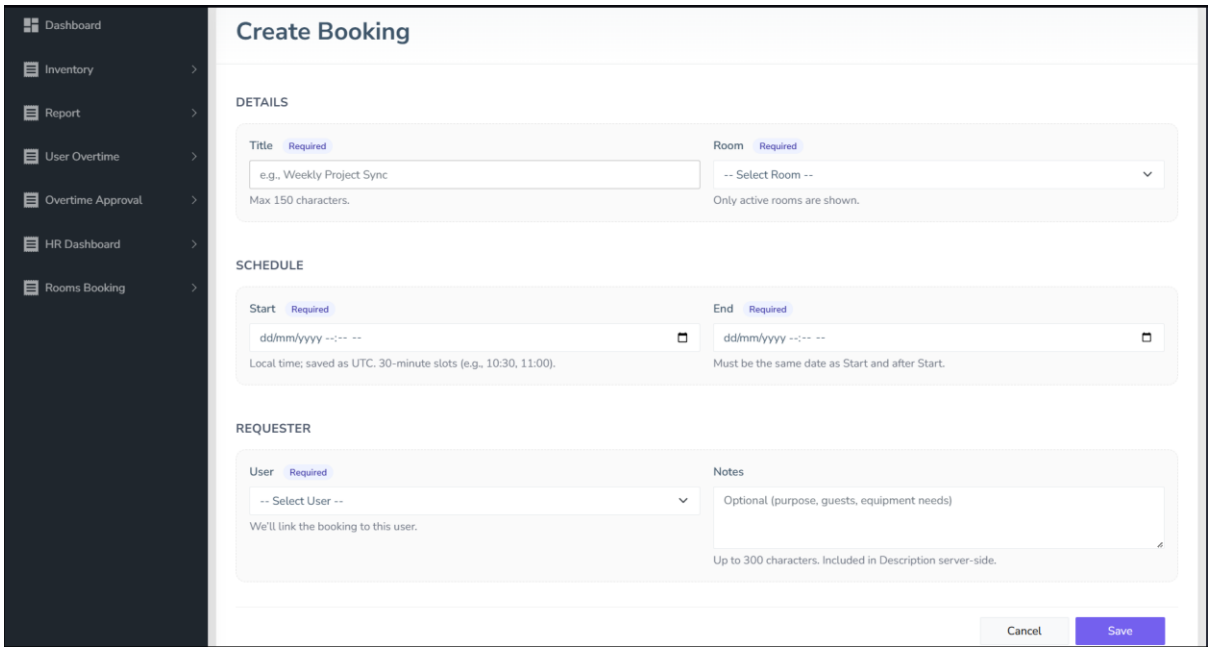
- **[HttpPost] (action=cancel) "Cancel or un-cancel a booking" (Index page)**
- **Purpose:** Lets the booking owner or a manager cancel an existing booking, or undo a previous cancellation back to Pending.
- **Key functionality:**
    - **Auth and roles:** Uses CanModify(b, me) so only the creator or a manager can act. If not allowed, returns Unauthorized with a clear message.
    - **Inputs (query/body):** Requires id > 0. Reads an optional JSON body. ShouldUndo(body) decides whether this is a normal cancel or an un-cancel.
    - **Guards:**
        - If the booking is Rejected, action is blocked. You cannot cancel or un-cancel a rejected booking.
        - For un-cancel, the booking must currently be Cancelled. Otherwise it returns a conflict.
    - **Cancel flow:** If the booking is not already Cancelled, sets CurrentStatus = Cancelled, stamps LastUpdatedUtc = UtcNow, saves, and returns the new status. If it is already Cancelled, just returns the current status so the call is safe to repeat.
    - **Un-cancel flow:**

- Checks for time overlap with any other Pending or Approved booking in the same room using interval overlap rules. If there is a clash, returns a conflict explaining why it cannot proceed.
- If clear, sets CurrentStatus = Pending, stamps LastUpdatedUtc = UtcNow, saves, and returns the new status.

```
[HttpPost]
0 references
public async Task<IActionResult> PostAsync(
    [FromQuery] string? action,
    [FromQuery] int? id,
    [FromQuery] string? scope,
    [FromBody] System.Text.Json.JsonElement? body // nullable is fine
)
{
```

```
// ----- BOOKING: CANCEL / UN-CANCEL (Owner or Manager) -------------------
if (string.Equals(action, "cancel", StringComparison.OrdinalIgnoreCase))
{
    if (!id.HasValue || id <= 0) return BadRequest("Missing id.");

    var b = await _db.Bookings.FirstOrDefaultAsync(x => x.BookingId == id.Value);
    if (b is null) return NotFound();

    var me = GetCurrentUserId();
    if (!CanModify(b, me))
        return Unauthorized("Only the creator or a manager can cancel/un-cancel this booking.");
    if (b.CurrentStatus == BookingStatus.Rejected)
        return Conflict("Rejected bookings cannot be cancelled or un-cancelled.");

    bool undo = ShouldUndo(body);

    if (!undo)
    {
        if (b.CurrentStatus != BookingStatus.Cancelled)
        {
            b.CurrentStatus = BookingStatus.Cancelled;
            b.LastUpdatedUtc = DateTime.UtcNow;
            await _db.SaveChangesAsync();
        }
        return Ok(new { id = b.BookingId, status = b.CurrentStatus.ToString() });
    }
    else
    {
        if (b.CurrentStatus != BookingStatus.Cancelled)
            return Conflict("Booking is not cancelled.");

        var overlap = await _db.Bookings.AsNoTracking().AnyAsync(x =>
            x.RoomId == b.RoomId &&
            x.BookingId != b.BookingId &&
            (x.CurrentStatus == BookingStatus.Pending || x.CurrentStatus == BookingStatus.Approved) &&
            x.StartUtc < b.EndUtc && b.StartUtc < x.EndUtc);

        if (overlap)
            return Conflict("Cannot un-cancel: time slot now overlaps another booking.");

        b.CurrentStatus = BookingStatus.Pending;
        b.LastUpdatedUtc = DateTime.UtcNow;
        await _db.SaveChangesAsync();
        return Ok(new { id = b.BookingId, status = b.CurrentStatus.ToString() });
    }
}
```

- **Create.cshtml**



This file (Create.cshtml) provides a dedicated web page for users to **create bookings** by filling in necessary details about it based on the provided fields. Once the user is satisfied they can save the booking and wait for approval. The users are also able to edit the booking using the same web page when clicking edit on the booking made which has been listed in the (Index.cshtml) web page

- **Constants & API base**
  - **Purpose:** app-wide limits for booking hours and the base API endpoint used for creates/updates/loads.
  - **Key values:**
    - DAY_START_HOUR = 8 (08:00)
    - DAY_END_HOUR = 20 (20:00)
    - api - absolute URL to BookingsApi

```
<script>
        // --- Constants ---
        const DAY_START_HOUR = 8;     // 08:00
        const DAY_END_HOUR = 20;      // 20:00
        const api = `${window.location.origin}/api/BookingsApi`;
```

- **UTC = datetime-local helpers**
  - **Purpose:** convert safely between ISO/UTC and the browser's datetime-local input value (which is local time). Handles timezone offsets correctly.

```
// ---------- UTC <-> datetime-local helpers ----------
function toLocalInputValue(iso) {
    if (!iso) return "";
    const hasTz = /(?:Z|[+\-]\d{2}:\d{2})$/i.test(String(iso));
    const d = new Date(iso);
    if (isNaN(d)) return "";
    const ms = hasTz
        ? d.getTime() - d.getTimezoneOffset() * 60000
        : d.getTime();
    return new Date(ms).toISOString().slice(0, 16); // "YYYY-MM-DDTHH:mm"
}

function fromLocalInputValue(localValue) {
    if (!localValue) return null;
    return new Date(localValue).toISOString(); // local -> UTC Z
}
```

- **UI helpers (alerts) & query parsing**
  - **Purpose:** show Bootstrap alerts at the page header and read the ?id= from querystring when editing.

```
// ---------- UI helpers ----------
function showAlert(type, msg) {
    document.getElementById("alerts").innerHTML = `
      <div class="alert alert-${type} alert-dismissible fade show mb-0" role="alert">
        ${msg}
        <button type="button" class="btn-close" data-bs-dismiss="alert"></button>
      </div>`;
}

function getQueryId() {
    const qs = new URLSearchParams(window.location.search);
    const qid = parseInt(qs.get("id") || "", 10);
    return Number.isInteger(qid) && qid > 0 ? qid : null;
}

function pad2(n) {
    return String(n).padStart(2, "0");
}

function formatLocalYmdHm(d) {
    return `${d.getFullYear()}-${pad2(d.getMonth() + 1)}-${pad2(
        d.getDate()
    )}T${pad2(d.getHours())}:${pad2(d.getMinutes())}`;
}
```

- **Time snapping & same-day enforcement**
  - **Purpose:** ensure start/end snap to 30-minute slots, enforce “same day” rule, and prevent invalid end < start situations.

```
function snapTo30(localValue) {
    if (!localValue) return localValue;
    const d = new Date(localValue);
    if (isNaN(d)) return localValue;
    d.setSeconds(0, 0);
    const m = d.getMinutes();
    let snappedMin;
    if (m < 15) snappedMin = 0;
    else if (m < 45) snappedMin = 30;
    else {
        d.setHours(d.getHours() + 1);
        snappedMin = 0;
    }
    d.setMinutes(snappedMin);
    return formatLocalYmdHm(d);
}

function enforceEndSameDay() {
    const startEl = document.getElementById("StartUtc");
    const endEl = document.getElementById("EndUtc");
    const sv = startEl.value,
        ev = endEl.value;
    if (!sv || !ev) return;

    const sd = new Date(sv);
    const ed = new Date(ev);
    if (isNaN(sd) || isNaN(ed)) return;

    const datesDiffer =
        sd.getFullYear() !== ed.getFullYear() ||
        sd.getMonth() !== ed.getMonth() ||
        sd.getDate() !== ed.getDate();

    if (datesDiffer) {
        ed.setFullYear(sd.getFullYear(), sd.getMonth(), sd.getDate());
        endEl.value = formatLocalYmdHm(ed);
        showAlert("warning", "End date was adjusted to the same day as Start.");
    }
}
```

```
function applyEndMinMaxForStartDay() {
    const startEl = document.getElementById("StartUtc");
    const endEl = document.getElementById("EndUtc");
    const sv = startEl.value;

    // compute local "now" rounded up to next :00/:30
    const nowLocalCeil = ceilToNext30Date(new Date());

    if (!sv) {
        // if empty, at least prevent any past time today
        startEl.min = formatLocalYmdHm(nowLocalCeil);
        endEl.min = formatLocalYmdHm(nowLocalCeil);
        startEl.removeAttribute("max");
        endEl.removeAttribute("max");
        return;
    }

    const sd = new Date(sv);
    if (isNaN(sd)) {
        startEl.min = formatLocalYmdHm(nowLocalCeil);
        endEl.min = formatLocalYmdHm(nowLocalCeil);
        startEl.removeAttribute("max");
        endEl.removeAttribute("max");
        return;
    }

    // the fixed day window
    const dayStart = new Date(sd); dayStart.setHours(DAY_START_HOUR, 0, 0, 0);
    const dayEnd = new Date(sd); dayEnd.setHours(DAY_END_HOUR, 0, 0, 0);

    // if the chosen start day is today, min is max(dayStart, nowLocalCeil); else min is dayStart
    const today = new Date();
    const isSameDay =
        sd.getFullYear() === today.getFullYear() &&
        sd.getMonth() === today.getMonth() &&
        sd.getDate() === today.getDate();

    const startMin = isSameDay && nowLocalCeil > dayStart ? nowLocalCeil : dayStart;
    const latestStart = new Date(dayEnd.getTime() - 30 * 60000); // 19:30

    // apply to Start picker
    startEl.min = formatLocalYmdHm(startMin);
    startEl.max = formatLocalYmdHm(latestStart);

    // clamp Start into allowed window
    if (new Date(startEl.value) < startMin) startEl.value = formatLocalYmdHm(startMin);
    if (new Date(startEl.value) > latestStart) startEl.value = formatLocalYmdHm(latestStart);

    // End is tied to (snapped) Start and capped by dayEnd
    const snappedStartStr = snapTo30(startEl.value);
    const snappedStart = new Date(snappedStartStr);

    endEl.min = formatLocalYmdHm(snappedStart);
    endEl.max = formatLocalYmdHm(dayEnd);

    // clamp End
    if (endEl.value) {
        const ed = new Date(endEl.value);
        if (ed < snappedStart) endEl.value = snappedStartStr;
        else if (ed > dayEnd) endEl.value = formatLocalYmdHm(dayEnd);
    }
```

- **Rounding "now" to the next slot & overall normalization**
  - **Purpose:** utility to ceil current time to next :00/:30 and a one-stop normalizer that snaps both fields, applies min/max, and fixes end < start.

```
function ceilToNext30Date(d) {
    const x = new Date(d);
    x.setSeconds(0, 0);
    const m = x.getMinutes();
    if (m === 0 || m === 30) return x;
    if (m < 30) { x.setMinutes(30); return x; }
    x.setHours(x.getHours() + 1); x.setMinutes(0); return x;
}

function normalizeTimes() {
    const startEl = document.getElementById("StartUtc");
    const endEl = document.getElementById("EndUtc");

    if (startEl.value) startEl.value = snapTo30(startEl.value);
    if (endEl.value) endEl.value = snapTo30(endEl.value);

    applyEndMinMaxForStartDay();

    if (startEl.value && endEl.value) {
        const sd = new Date(startEl.value);
        const ed = new Date(endEl.value);
        if (ed < sd) {
            const newEnd = new Date(sd);
            newEnd.setMinutes(sd.getMinutes() + 30);
            endEl.value = formatLocalYmdHm(newEnd);
        }
    }
}
```

- **Lookups (rooms & users) + dropdown population**
  - **Purpose:** fetch rooms/users once and fill the selects; supports preselecting items when editing.

```
async function loadLookups(selectedRoomId, selectedUserId) {
    const res = await fetch(`${api}?lookups=1`);
    if (!res.ok) throw new Error(await res.text());
    const { rooms = [], users = [] } = await res.json();

    const pick = (obj, ...keys) => {
        for (const k of keys)
            if (obj[k] !== undefined && obj[k] !== null) return obj[k];
        return null;
    };

    // Rooms
    const roomDdl = document.getElementById("RoomId");
    roomDdl.innerHTML = '<option value="">-- Select Room --</option>';
    rooms.forEach(r => {
        const id = pick(r, "RoomId", "roomId");
        if (id == null) return;

        const name =
            (pick(r, "Name", "name", "RoomName", "roomName") ?? `Room ${id}`)
                .toString()
                .trim();
        const loc = pick(r, "LocationCode", "locationCode");
        const cap = pick(r, "Capacity", "capacity");

        const opt = document.createElement("option");
        opt.value = id;
        opt.textContent = `${name}${loc ? ` @@ ${loc}` : ""}${cap ? ` — cap ${cap}` : ""
            }`;
        if (selectedRoomId && Number(selectedRoomId) === Number(id))
            opt.selected = true;
        roomDdl.appendChild(opt);
    });

    // Users
    const userDdl = document.getElementById("RequestedByUserId");
    userDdl.innerHTML = '<option value="">-- Select User --</option>';
    users.forEach(u => {
        const id = pick(u, "Id", "id");
        if (id == null) return;

        const name = pick(u, "UserName", "userName") ?? "";
        const email = pick(u, "Email", "email") ?? "";

        const opt = document.createElement("option");
        opt.value = id;
        opt.textContent = email ? `${name} (${email})` : name;
        if (selectedUserId && Number(selectedUserId) === Number(id))
            opt.selected = true;
        userDdl.appendChild(opt);
    });
}
```

- **Form submit (create vs update)**
  - **Purpose:** validate fields, enforce time rules, convert to UTC, then POST (create) or PUT (update). On success, navigate back to Index.
  - **Rules checked:**
    - Required fields, same day, within 08:00–20:00, start ≤ 19:30, end ≤ 20:00, end after start.

```
document.getElementById("bookingForm").addEventListener("submit", async e => {
    e.preventDefault();
    normalizeTimes();

    const id = document.getElementById("Id").value;
    const title = document.getElementById("Title").value.trim();
    const roomId = Number(document.getElementById("RoomId").value);
    const startLocal = document.getElementById("StartUtc").value;
    const endLocal = document.getElementById("EndUtc").value;
    const requestedByUserId = Number(
        document.getElementById("RequestedByUserId").value
    );
    const note = (document.getElementById("Note").value || "").trim();

    if (!title || !roomId || !startLocal || !endLocal || !requestedByUserId) {
        showAlert("danger", "Please fill all required fields.");
        return;
    }

    const sd = new Date(startLocal),
        ed = new Date(endLocal);
    const sameDay =
        sd.getFullYear() === ed.getFullYear() &&
        sd.getMonth() === ed.getMonth() &&
        sd.getDate() === ed.getDate();
    if (!sameDay) {
        showAlert("danger", "End must be on the same date as Start.");
        return;
    }

    const startMinutes = sd.getHours() * 60 + sd.getMinutes();
    const endMinutes = ed.getHours() * 60 + ed.getMinutes();
    const latestStartMinutes = DAY_END_HOUR * 60 - 30;

    if (startMinutes < DAY_START_HOUR * 60 || startMinutes > latestStartMinutes) {
        showAlert("danger", "Start must be between 08:00 and 19:30.");
        return;
    }
    if (endMinutes > DAY_END_HOUR * 60) {
        showAlert("danger", "End must be no later than 20:00.");
        return;
    }

    const startUtc = fromLocalInputValue(startLocal);
    const endUtc = fromLocalInputValue(endLocal);
    if (new Date(endUtc) <= new Date(startUtc)) {
        showAlert("danger", "End time must be after Start time.");
        return;
    }
```

```
    try {
        let res;
        if (id) {
            const payload = {
                RoomId: roomId,
                Title: title,
                StartUtc: startUtc,
                EndUtc: endUtc,
                Note: note || null
            };
            res = await fetch(`${api}?id=${encodeURIComponent(id)}`, {
                method: "PUT",
                headers: { "Content-Type": "application/json" },
                body: JSON.stringify(payload)
            });
        } else {
            const payload = {
                RoomId: roomId,
                RequestedByUserId: requestedByUserId,
                Title: title,
                StartUtc: startUtc,
                EndUtc: endUtc,
                Note: note || null,
                Description: note || null
            };
            res = await fetch(`${api}`, {
                method: "POST",
                headers: { "Content-Type": "application/json" },
                body: JSON.stringify(payload)
            });
        }

        if (!res.ok) {
            const t = await res.text();
            showAlert("danger", (id ? "Update" : "Create") + " failed: " + t);
            return;
        }
        window.location.href = '@Url.Action("Index", "Bookings", new { area = "Bookings" })';
    } catch (err) {
        showAlert("danger", (id ? "Update" : "Create") + " failed: " + err.message);
    }
```

- **Boot: edit mode vs create mode**
  - **Purpose:** on DOM ready, detect if ?id= exists.

- **Edit:** load existing booking, prefill fields, run lookups with preselect, set min constraints, normalize, change button to “Update”.
- **Create:** load lookups fresh, set min constraints based on “now rounded to next slot”.

```
    document.addEventListener("DOMContentLoaded", async () => {
        const id = getQueryId();
        if (id) {
            const res = await fetch(`${api}?id=${encodeURIComponent(id)}`);
            if (!res.ok) {
                showAlert("danger", await res.text());
                return;
            }
            const b = await res.json();

            document.getElementById("Id").value = b.id ?? b.Id ?? b.bookingId;
            document.getElementById("Title").value = b.title ?? "";
            document.getElementById("StartUtc").value = snapTo30(
                toLocalInputValue(b.startUtc)
            );
            document.getElementById("EndUtc").value = snapTo30(
                toLocalInputValue(b.endUtc)
            );
            document.getElementById("Note").value = b.note ?? "";

            await loadLookups(b.roomId, b.requestedByUserId);
            const nowLocalCeil = ceilToNext30Date(new Date());
            document.getElementById("StartUtc").min = formatLocalYmdHm(nowLocalCeil);
            document.getElementById("EndUtc").min = formatLocalYmdHm(nowLocalCeil);

            applyEndMinMaxForStartDay();
            normalizeTimes();

            document.getElementById("submitBtn").textContent = "Update";
        } else {
            await loadLookups();
            const nowLocalCeil = ceilToNext30Date(new Date());
            document.getElementById("StartUtc").min = formatLocalYmdHm(nowLocalCeil);
            document.getElementById("EndUtc").min = formatLocalYmdHm(nowLocalCeil);
            applyEndMinMaxForStartDay();
        }
    });
</script>
```

- **BookingsAPI(Create.cshtml)**

- **[HttpGet] ("Create lookups pack")**

- **Purpose:** Feeds the Create page with the dropdown data it needs. When lookups=1, it returns active rooms and the allowed user list so the form can populate its selectors.
- **Key functionality:**
  - **Auth and roles:** Uses IsManager() to decide the user list. Managers get all users. Non-managers only get themselves. This keeps the form simple and avoids accidental impersonation.
  - **Inputs (query):** Accepts many query params, but in this branch it only cares about lookups=1. The rest (id, scope, search, status, from, to, roomId, page, pageSize, userId, companyId, departmentId, includeInactive) are ignored here.
  - **Rooms lookup:** Returns only active rooms, sorted by name, shaped as { roomId, roomName }.

- **Users lookup:**
  - Manager: all users { Id, UserName = FullName, Email }.
  - Non-manager: resolves current user id and returns just that one; if it cannot resolve, returns an empty list to keep the UI safe.
- **Response:** 200 OK with { rooms: [...], users: [...] }. Empty arrays are possible when there are no active rooms or the current user cannot be resolved.

```
[HttpGet]
1 reference
public async Task<IActionResult> GetAsync(
    [FromQuery] int? id,
    [FromQuery] int? lookups,
    [FromQuery] string? scope,
    [FromQuery] bool includeInactive = true,
    [FromQuery] string? search = null,
    [FromQuery] string? status = null,
    [FromQuery] DateTime? from = null,
    [FromQuery] DateTime? to = null,
    [FromQuery] int? roomId = null,
    [FromQuery] int page = 1,
    [FromQuery] int pageSize = 450,
    [FromQuery] int? userId = null,
    [FromQuery] int? companyId = null,
    [FromQuery] int? departmentId = null
)
```

```
if (lookups == 1)
{
    var rooms = await _db.Rooms
        .AsNoTracking()
        .Where(r => r.IsActive)
        .OrderBy(r => r.RoomName)
        .Select(r => new { roomId = r.RoomId, roomName = r.RoomName })
        .ToListAsync();

    object usersPayload;
    if (IsManager())
    {
        usersPayload = await _db.Users
            .AsNoTracking()
            .OrderBy(u => u.FullName)
            .Select(u => new { u.Id, UserName = u.FullName, u.Email })
            .ToListAsync();
    }
    else
    {
        var me = GetCurrentUserId();
        usersPayload = (me is int myId)
            ? await _db.Users.AsNoTracking()
                .Where(u => u.Id == myId)
                .Select(u => new { u.Id, UserName = u.FullName, u.Email })
                .ToListAsync()
            : new object[0];
    }

    return Ok(new { rooms, users = usersPayload });
}
```

- **[HttpGet] ("Create - load booking details by id")**
- **Purpose:** When the Create page is used to edit an existing booking, this branch returns the full details for that booking so the form can prefill.
- **Key functionality:**

- **Auth and roles:** Managers can view any booking. Non-managers can only view a booking if they are the requester or the target user. If not allowed, returns 401 Unauthorized with a clear message.
- **Inputs (query):** Requires id to be present. If the record is not found for that id, returns 404 Not Found.
- **Record fetch:** Reads the booking with AsNoTracking() to avoid EF change tracking during a simple read.
- **Projection and UTC safety:** Returns a lean object for the form with startUtc and endUtc explicitly tagged as UTC using DateTime.SpecifyKind(...) to prevent timezone glitches on the client.

```
[HttpGet]
1 reference
public async Task<IActionResult> GetAsync(
    [FromQuery] int? id,
    [FromQuery] int? lookups,
    [FromQuery] string? scope,
    [FromQuery] bool includeInactive = true,
    [FromQuery] string? search = null,
    [FromQuery] string? status = null,
    [FromQuery] DateTime? from = null,
    [FromQuery] DateTime? to = null,
    [FromQuery] int? roomId = null,
    [FromQuery] int page = 1,
    [FromQuery] int pageSize = 450,
    [FromQuery] int? userId = null,
    [FromQuery] int? companyId = null,
    [FromQuery] int? departmentId = null
)
```

```
// BOOKING DETAILS
if (id.HasValue)
{
    var b = await _db.Bookings.AsNoTracking().FirstOrDefaultAsync(x => x.BookingId == id.Value);
    if (b is null) return NotFound();

    var me = GetCurrentUserId();
    if (!IsManager() && me != b.RequestedByUserId && me != b.TargetUserId)
        return Unauthorized("Not allowed to view this booking.");

    return Ok(new
    {
        id = b.BookingId,
        roomId = b.RoomId,
        requestedByUserId = b.RequestedByUserId,
        title = b.Title,
        description = b.Purpose,
        startUtc = DateTime.SpecifyKind(b.StartUtc, DateTimeKind.Utc),
        endUtc = DateTime.SpecifyKind(b.EndUtc, DateTimeKind.Utc),
        status = b.CurrentStatus.ToString(),
        note = b.Note
    });
}
```

- **[HttpPost] ("Create booking")**
- **Purpose:** Handles the Create page form submission. Takes a JSON payload, builds a booking, applies role rules, runs validations, and saves it.
- **Key functionality:**

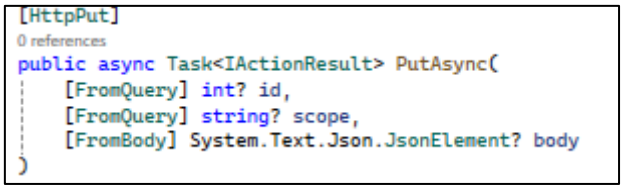
- **Auth and roles:** Non-managers can only create bookings for themselves. The code forces RequestedByUserId to the current user and sets TargetUserId to the same user if it was not provided. Managers can create for anyone. If the current user cannot be resolved, returns Unauthorized.
- **Inputs:** Expects a JSON body that maps to CreateBookingDto. Query params like action and scope are not used here. If deserialization fails or the payload is malformed, returns Bad Request.
- **Ownership coercion (non-manager):** After reading the current user id, the server rewrites the DTO so non-managers cannot spoof another user.
- **Validation and conflicts:** Although not shown in your snippet, this branch is where you typically validate required fields and check for time overlaps before insert. If any rule fails, it should return a clear error.
- **Insert:** Adds the new booking entity to _db.Bookings and saves.
- **Response:** Uses CreatedAtAction(nameof(GetAsync), new { id = entity.BookingId }, new { id = entity.BookingId }) so the client gets a 201 with the new id and a canonical location to fetch the created record.

```
[HttpPost]
0 references
public async Task<IActionResult> PostAsync(
    [FromQuery] string? action,
    [FromQuery] int? id,
    [FromQuery] string? scope,
    [FromBody] System.Text.Json.JsonElement? body // nullable is fine
)
{
```

```
// ------------------ BOOKING: CREATE ------------------------
var create = JsonSerializer.Deserialize<CreateBookingDto>(json, JsonOpts);
if (create is null) return BadRequest("Invalid payload.");

// For non-managers, force ownership to the current user
var meCreate = GetCurrentUserId();
if (!IsManager())
{
    if (meCreate is null) return Unauthorized();
    create = create with
    {
        RequestedByUserId = meCreate.Value,
        TargetUserId = create.TargetUserId ?? meCreate.Value
    };
}
```

```
    _db.Bookings.Add(entity);
    await _db.SaveChangesAsync();

    return CreatedAtAction(nameof(GetAsync),
        new { id = entity.BookingId },
        new { id = entity.BookingId });
}
```

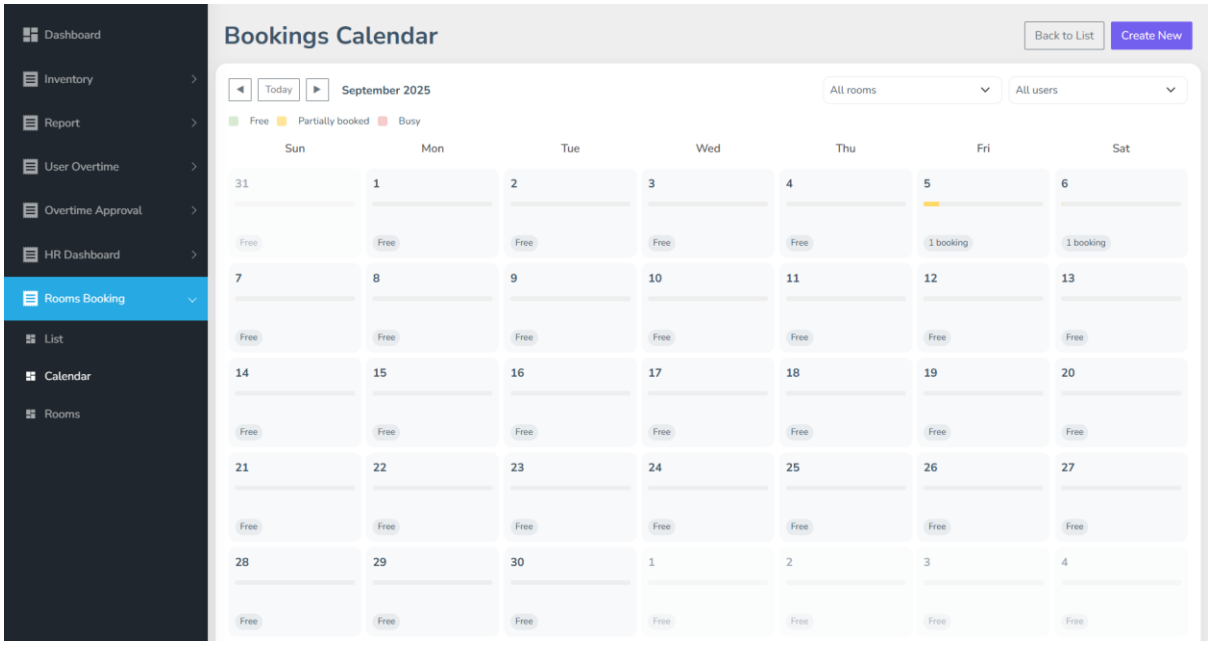
- **[HttpPut] ("Update booking")**
- **Purpose:** Saves edits from the Create/Edit page to an existing booking. The client calls PUT ?id={bookingId} with an UpdateBookingDto JSON body.
- **Key functionality:**
    - **Auth and roles:** Uses CanModify(entity, me) so only the creator or a manager can edit. If not allowed, returns Unauthorized with a clear message.
    - **Inputs (query/body):**
        - Query: id is required and must be greater than 0.
        - Body: JSON that deserializes to UpdateBookingDto. If the payload is missing or invalid, the action should return Bad Request.
    - **Record lookup:** Loads the booking by BookingId. If not found, returns Not Found.
    - **Editable statuses only:** Only bookings in Pending or Rejected can be updated. Any other status returns Conflict to protect approved and cancelled records.
    - **Apply changes:** This is where you assign the incoming fields to entity (title, purpose, room, start/end UTC, target user, note). You should also stamp LastUpdatedUtc = UtcNow.
    - **Overlap and rules:** Before saving, run the same interval overlap checks you use on create, plus any business rules (room required, start < end, valid room/user ids). On failure, return a clear error instead of saving.
    - **Save:** Persists the changes and returns a lightweight confirmation object with the id.

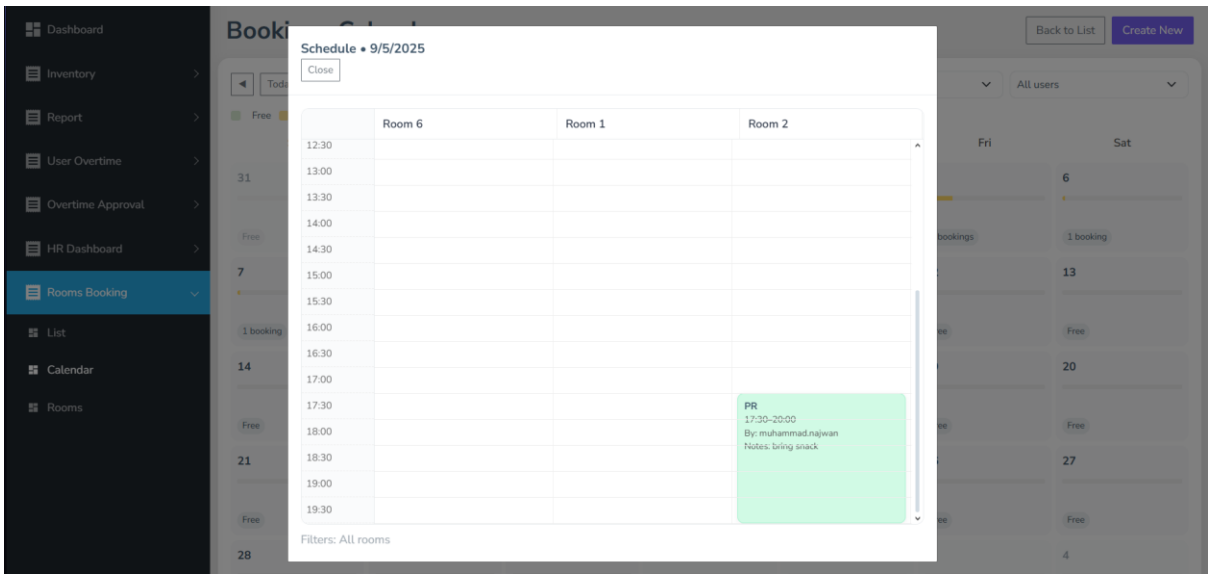


```
// BOOKING: UPDATE  (Owner or Manager)
var payload = JsonSerializer.Deserialize<UpdateBookingDto>(json, JsonOpts);

var entity = await _db.Bookings.FirstOrDefaultAsync(x => x.BookingId == id.Value);
if (entity is null) return NotFound();

var me = GetCurrentUserId();
if (!CanModify(entity, me))
    return Unauthorized("Only the creator or a manager can edit this booking.");

if (entity.CurrentStatus is not BookingStatus.Pending and not BookingStatus.Rejected)
    return Conflict("Only Pending or Rejected bookings can be updated.");

// optional: switch rooms
if (payload?.RoomId is int newRoomId && newRoomId != entity.RoomId)
{
    var roomOk2 = await _db.Rooms.AnyAsync(r => r.RoomId == newRoomId && r.IsActive);
    if (!roomOk2) return BadRequest("New room not found or inactive.");
    entity.RoomId = newRoomId;
}
```

```
await _db.SaveChangesAsync();
return Ok(new { id = entity.BookingId });
```

- **Calendar.cshtml**





This file (Calendar.cshtml) provides a dedicated web page for users to **view a calendar layout of the bookings** that have been **approved by the admin** so that they can identify when the most suitable slot is, to book for upcoming meetings or events. A schedule view is also available when users click on any date which shall showcase the bookings made on that specific day in more detail.

- **Constants & API base**
  - **Purpose:** define the base API endpoint and the read-only scope used by the calendar.

```
<script>
        (() => {
            // --- Constants / API ---
            const api = `${window.location.origin}/api/BookingsApi`;
            const calendarScope = "calendar";
```

- **Small helpers & global error surface**
  - **Purpose:** tiny utilities for dates/strings + a lightweight alert surface that also logs to console; global listeners convert thrown errors into dismissible alerts.

```
const pad = n => String(n).padStart(2, "0");
function startOfMonth(d) { const x = new Date(d); x.setDate(1); x.setHours(0, 0, 0, 0); return x; }
function startOfWeek(d) { const x = new Date(d); const day = (x.getDay() + 7) % 7; x.setDate(x.getDate() - day); x.setHours(0, 0, 0, 0); return x; }
function addDays(d, n) { const x = new Date(d); x.setDate(x.getDate() + n); return x; }
function ymd(d) { return `${d.getFullYear()}-${pad(d.getMonth() + 1)}-${pad(d.getDate())}`; }
function toUtcIso(d) { const off = d.getTimezoneOffset(); return new Date(d.getTime() - off * 60000).toISOString(); }
function fmtLocal(d) { const x = new Date(d); return `${pad(x.getHours())}:${pad(x.getMinutes())}`; }
function esc(s) { return String(s ?? "").replace(/[&<>"']/g, m => ({ "&": "&amp;", "<": "&lt;", ">": "&gt;", '"': "&quot;", "'": "&#39;" }[m])); }
function alertMsg(msg) {
    document.getElementById("alerts").innerHTML = `
  <div class="alert alert-warning alert-dismissible fade show" role="alert">
    ${esc(String(msg))}
    <button type="button" class="btn-close" data-bs-dismiss="alert"></button>
  </div>`;
    console.warn("[Bookings Calendar]", msg);
}
window.addEventListener("error", e => alertMsg(e.message));
window.addEventListener("unhandledrejection", e => alertMsg(e.reason?.message || e.reason));
```

- **DOM references (calendar + day-board modal)**
  - **Purpose:** cache all frequently used DOM nodes for fast access.

```
const lblMonth = document.getElementById("lblMonth");
const calGrid = document.getElementById("calGrid");
const dowRow = document.getElementById("dowRow");
const ddlRoom = document.getElementById("ddlRoom");
const ddlUser = document.getElementById("ddlUser");

// Day board elements
const boardModalEl = document.getElementById("dayBoardModal"); let boardModal;
const boardTitle = document.getElementById("boardTitle");
const boardFilters = document.getElementById("boardFiltersInfo");
const timeCol = document.getElementById("timeCol");
const colsHead = document.getElementById("colsHeader");
const colsScroll = document.getElementById("colsScroll");
const canvas = document.getElementById("canvas");
const colGrid = document.getElementById("colGrid");
const nowLine = document.getElementById("nowLine");
```

- **Static DOW header (renders once)**
  - **Purpose:** write the “Sun–Sat” header row only once per page load.

```
if (!dowRow.dataset.done) {
    ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"].forEach(d => {
        const div = document.createElement("div");
        div.className = "dow";
        div.textContent = d;
        dowRow.appendChild(div);
    });
    dowRow.dataset.done = "1";
}
```

- **Lookups (rooms & users) + selects population + maps**
  - **Purpose:** fetch reusable lists, populate dropdown filters, and build userMap/roomMap for fast name resolution inside the day board.

```
let userMap = new Map(), roomMap = new Map(), users = [], rooms = [];
async function loadLookups() {
    try {
        const r = await fetch(`${api}?scope=${calendarScope}&lookups=1`);
        if (!r.ok) throw new Error(`Lookups ${r.status} ${r.statusText}`);
        const js = await r.json();
        rooms = js.rooms ?? [];
        users = js.users ?? [];
    } catch (e) {
        alertMsg(e.message);
        rooms = [];
        users = [];
    }

    // Fill selects
    ddlRoom.innerHTML = '<option value="">All rooms</option>';
    rooms.forEach(r => {
        const id = r.roomId;
        const name = r.roomName ?? `Room ${id}`;
        if (id == null) return;
        const opt = document.createElement("option");
        opt.value = id;
        opt.textContent = name;
        ddlRoom.appendChild(opt);
    });

    ddlUser.innerHTML = '<option value="">All users</option>';
    users.forEach(u => {
        const id = u.Id ?? u.id;
        const name = u.UserName ?? u.userName ?? "";
        const email = u.Email ?? "";
        const opt = document.createElement("option");
        opt.value = id;
        opt.textContent = email ? `${name} (${email})` : name;
        ddlUser.appendChild(opt);
    });

    userMap = new Map(
        users.map(u => [
            Number(u.Id ?? u.id),
            (u.UserName ?? u.userName ?? "") + (u.Email ? ` (${u.Email})` : "")
        ])
    );
    roomMap = new Map(
        rooms.map(r => {
            const id = Number(r.roomId ?? r.RoomId);
            const name = r.roomName ?? r.Name ?? r.RoomName ?? `Room ${id}`;
            return [id, name];
        })
    );
}
```

- **Month data fetch (server) + client-side filtering**
  - **Purpose:** load visible month's bookings in one go, then filter out cancelled and (optionally) limit to a specific user (requestor or target).
  - **Request params:** scope, from, to, pageSize, roomId?, userId?
  - **Client filter:** drop Cancelled, include rows where requestedByUserId or targetUserId matches selected user.

```
let curMonth = startOfMonth(new Date()), currentData = [];
async function fetchGridData(month) {
    const gridStart = startOfWeek(startOfMonth(month));
    const gridEnd = addDays(gridStart, 6 * 7 - 1);

    const params = new URLSearchParams();
    params.set("scope", calendarScope);               // <<— calendar scope (bypass RBAC for reads)
    params.set("from", toUtcIso(gridStart));
    params.set("to", toUtcIso(gridEnd));
    params.set("pageSize", "10000");
    if (ddlRoom.value) params.set("roomId", ddlRoom.value);
    if (ddlUser.value) params.set("userId", ddlUser.value);

    const r = await fetch(`${api}?${params.toString()}`);
    if (!r.ok) throw new Error(`API ${r.status}: ${await r.text().catch(() => r.statusText)}`);
    let data = await r.json();

    // client-side refine: drop Cancelled and apply user filter (requested/target)
    const userSel = ddlUser.value ? Number(ddlUser.value) : null;
    data = data.filter(b => {
        const status = (b.status ?? "").toString();
        if (status === "Cancelled") return false;
        if (userSel !== null) {
            const rq = Number(b.requestedByUserId ?? b.UserId ?? b.userId ?? NaN);
            const tg = Number(b.targetUserId ?? b.TargetUserId ?? NaN);
            if (!(rq === userSel || tg === userSel)) return false;
        }
        return true;
    });

    return { gridStart, data };
}
```

- **Month grid: skeleton, occupancy compute, render**
  - **Purpose:** draw 6×7 calendar cells, compute daily occupancy (% of minutes booked), color the cell (free/partial/busy), and show a quick count chip.

```
function renderSkeletonGrid(month) {
    lblMonth.textContent = month.toLocaleString(undefined, { month: "long", year: "numeric" });
    calGrid.innerHTML = "";
    const gridStart = startOfWeek(startOfMonth(month));
    for (let i = 0; i < 42; i++) {
        const d = addDays(gridStart, i);
        const cell = document.createElement("div");
        cell.className = `cell free ${d.getMonth() === month.getMonth() ? "" : "other-month"}`;
        cell.innerHTML = `
    <div class="daynum">${d.getDate()}</div>
    <div class="status-bar"><div style="width:0%"></div></div>
    <div class="chip">Free</div>`;
        cell.addEventListener("click", () => openDayBoard(d));
        calGrid.appendChild(cell);
    }
}
```

```
function computeDailyOccupancy(data, gridStart) {
    const start = new Date(gridStart), end = addDays(start, 41);
    const dayMap = new Map();

    for (let d = new Date(start); d <= end; d = addDays(d, 1)) {
        dayMap.set(ymd(d), { minutes: 0, items: [], date: new Date(d) });
    }

    for (const b of data) {
        const s = new Date(b.startUtc), e = new Date(b.endUtc);
        if (e < start || s > end) continue;

        let cur = new Date(Math.max(s, start)); cur.setHours(0, 0, 0, 0);
        while (cur <= end) {
            const dayStart = new Date(cur), dayEnd = new Date(cur); dayEnd.setHours(23, 59, 59, 999);
            const os = new Date(Math.max(s, dayStart)), oe = new Date(Math.min(e, dayEnd));
            const mins = Math.max(0, Math.ceil((oe - os) / 60000));
            if (mins > 0) {
                const key = ymd(cur), entry = dayMap.get(key);
                entry.minutes += Math.min(mins, 1440);
                if (entry.items.length < 5) entry.items.push({ full: b, start: os, end: oe });
            }
            cur = addDays(cur, 1);
            if (cur > e) break;
        }
    }

    for (const [, entry] of dayMap) {
        const pct = Math.max(0, Math.min(1, entry.minutes / 1440));
        entry.pct = pct;
        entry.className = pct === 0 ? "free" : (pct <= 0.6 ? "partial" : "busy");
    }
    return { start, dayMap };
}

let occCache = null;
function renderGrid(month) {
    lblMonth.textContent = month.toLocaleString(undefined, { month: "long", year: "numeric" });
    calGrid.innerHTML = "";
    const start = occCache.start;

    for (let i = 0; i < 42; i++) {
        const d = addDays(start, i), key = ymd(d);
        const entry = occCache.dayMap.get(key) || { pct: 0, className: "free", items: [], date: d };

        const cell = document.createElement("div");
        cell.className = `cell ${entry.className} ${d.getMonth() === month.getMonth() ? "" : "other-month"}`;
        cell.title = entry.items.length ? `${entry.items.length} booking(s)` : "Free day";
        cell.innerHTML = `
    <div class="daynum">${d.getDate()}</div>
    <div class="status-bar"><div style="width:${Math.round(entry.pct * 100)}%"></div></div>
    <div class="chip">${entry.items.length ? `${entry.items.length} booking${entry.items.length > 1 ? "s" : ""}` : "Free"}</div>`;
        cell.addEventListener("click", () => openDayBoard(d));
        calGrid.appendChild(cell);
    }
```

- **Day Board (room timeline view)**
  - **Purpose:** full-day vertical timeline (08:00–20:00, 30-min slots) with columns per room, showing bookings stacked in the correct positions, plus a live “nowline” when viewing today.
  - **Key values:**
    - DAY_START_HOUR = 8, DAY_END_HOUR = 20, SLOT_MIN = 30, SLOT_PX = 32.
    - TOTAL_INTERVALS — number of 30-min rows.
    - getRoomId, getUserId — normalize various possible field names from API.

```
function openDayBoard(dayDate) {
    boardTitle.textContent = `Schedule • ${dayDate.toLocaleDateString()}`;
    boardFilters.textContent = `Filters: ${ddlRoom.options[ddlRoom.selectedIndex]?.text || "All rooms"}`;

    // Left time labels
    timeCol.innerHTML = "";
    const timeInner = document.createElement("div");
    timeInner.id = "timeInner";
    timeCol.appendChild(timeInner);
    for (let i = 0; i <= TOTAL_INTERVALS; i++) {
        const mins = i * SLOT_MIN;
        const hr = DAY_START_HOUR + Math.floor(mins / 60);
        const mm = (mins % 60) === 0 ? "00" : "30";
        const row = document.createElement("div");
        row.className = "time-slot";
        row.textContent = `${String(hr).padStart(2, "0")}:${mm}`;
        timeInner.appendChild(row);
    }

    // Columns = rooms (respect filter)
    let columns = rooms
        .filter(r => !ddlRoom.value || String(r.roomId ?? r.RoomId) === ddlRoom.value)
        .map(r => {
            const id = Number(r.roomId ?? r.RoomId);
            const label = r.roomName ?? r.Name ?? r.RoomName ?? `Room ${id}`;
            return { key: id, label };
        });
    if (columns.length === 0) columns = [{ key: Number.MIN_SAFE_INTEGER, label: "Room" }];

    colsHead.innerHTML = "";
    colsHead.style.gridTemplateColumns = `repeat(${columns.length},1fr)`;
    columns.forEach(c => {
        const h = document.createElement("div");
        h.textContent = c.label;
        colsHead.appendChild(h);
    });

    canvas.style.height = `${TOTAL_INTERVALS * SLOT_PX}px`;

    // Build grid columns
    colGrid.innerHTML = "";
    colGrid.style.gridTemplateColumns = `repeat(${columns.length},1fr)`;
    for (let i = 0; i < columns.length; i++) {
        const g = document.createElement("div");
        g.className = "gcol";
        colGrid.appendChild(g);
        const container = document.createElement("div");
        container.style.position = "relative";
        container.style.height = "100%";
        g.appendChild(container);
    }
```

```
// Horizontal lines
[...canvas.querySelectorAll(".hline")].forEach(x => x.remove());
for (let i = 0; i <= TOTAL_INTERVALS; i++) {
    const l = document.createElement("div");
    l.className = "hline";
    l.style.top = `${i * SLOT_PX}px`;
    canvas.appendChild(l);
}

// Data rows for selected day
const dayStart = new Date(dayDate); dayStart.setHours(DAY_START_HOUR, 0, 0, 0);
const dayEnd = new Date(dayDate); dayEnd.setHours(DAY_END_HOUR, 0, 0, 0);
const rows = (currentData || []).filter(b => {
    const s = new Date(b.startUtc), e = new Date(b.endUtc);
    return e >= dayStart && s <= dayEnd;
});

function colIndexFor(b) {
    const id = getRoomId(b);
    const idx = columns.findIndex(c => c.key === id);
    return idx === -1 ? 0 : idx;
}

function posPx(s, e) {
    const clipS = new Date(Math.max(s.getTime(), dayStart.getTime()));
    const clipE = new Date(Math.min(e.getTime(), dayEnd.getTime()));
    const minutesFromStart = (clipS.getTime() - dayStart.getTime()) / 60000; // 0..720
    const dur = (clipE.getTime() - clipS.getTime()) / 60000;                 // 0..720
    return {
        topPx: (minutesFromStart / SLOT_MIN) * SLOT_PX,
        hPx: (dur / SLOT_MIN) * SLOT_PX
    };
}

for (const b of rows) {
    const s = new Date(b.startUtc), e = new Date(b.endUtc);
    const { topPx, hPx } = posPx(s, e);
    const idx = colIndexFor(b);
    const container = colGrid.children[idx].firstChild;

    const roomName = roomMap.get(getRoomId(b)) ?? (b.roomName ?? b.RoomName ?? "Room");
    const whoName = userMap.get(getUserId(b) ?? -1) ?? (b.userName ?? b.UserName ?? "—");
    const title = b.title ?? roomName;
    const note = (b.note ?? b.description ?? b.Notes ?? "").toString();

    const div = document.createElement("div");
    div.className = "booking";
    div.style.top = `${topPx}px`;
    div.style.height = `${hPx}px`;
    div.title = note ? `Notes: ${note}` : "";
    div.innerHTML = `
<div class="t">${esc(title)}</div>
<div class="meta">${esc(fmtLocal(s))}–${esc(fmtLocal(e))}</div>
<div class="meta">By: ${esc(whoName)}</div>
${note ? `<div class="meta">Notes: ${esc(note)}</div>` : ""}`;
    container.appendChild(div);
}
```

```
// Now-line
const isToday = ymd(new Date()) === ymd(dayDate);
if (isToday) {
    const now = new Date();
    const mins = (now.getHours() * 60 + now.getMinutes()) - (DAY_START_HOUR * 60);
    const px = Math.max(0, Math.min(TOTAL_INTERVALS * SLOT_PX, (mins / SLOT_MIN) * SLOT_PX));
    nowLine.classList.remove("d-none");
    nowLine.style.top = `${px}px`;
} else {
    nowLine.classList.add("d-none");
}

// Scroll sync
colsScroll.removeEventListener("scroll", onColsScroll);
function onColsScroll() {
    timeInner.style.transform = `translateY(-${colsScroll.scrollTop}px)`;
}
colsScroll.addEventListener("scroll", onColsScroll, { passive: true });
timeInner.style.transform = `translateY(-${colsScroll.scrollTop}px)`;
timeCol.scrollTop = colsScroll.scrollTop;

(boardModal ||= new bootstrap.Modal(boardModalEl)).show();
```

- **Navigation & filters**
  - **Purpose:** move month, jump to today, and re-render when filters change.

```
document.getElementById("btnPrev").onclick = async () => { curMonth.setMonth(curMonth.getMonth() - 1); await redraw(); };
document.getElementById("btnNext").onclick = async () => { curMonth.setMonth(curMonth.getMonth() + 1); await redraw(); };
document.getElementById("btnToday").onclick = async () => { curMonth = startOfMonth(new Date()); await redraw(); };
ddlRoom.onchange = redraw;
ddlUser.onchange = redraw;
```

- **Redraw pipeline (skeleton > fetch > occupancy > render)**
  - **Purpose:** orchestrate the refresh sequence; caches currentData and occCache for downstream use (e.g., day board).

```
async function redraw() {
    renderSkeletonGrid(curMonth);
    try {
        const { gridStart, data } = await fetchGridData(curMonth);
        currentData = data;
        occCache = computeDailyOccupancy(currentData, gridStart);
        renderGrid(curMonth);
    } catch (e) {
        alertMsg(e.message);
    }
}
```

- **Bootstrapping**
  - **Purpose:** initial page load: fetch lookups, then draw the current month.

```
(async function init() {
                await loadLookups();
                await redraw();
            })();
        })();
    </script>
```

  - **BookingsAPI(Calender.cshtml)**
  - **[HttpGet] ("Calendar lookups pack")**
  - **Purpose:** Feeds the Calendar page with dropdown data. When scope=calendar and lookups=1, it returns active rooms and all users so the calendar filters and chips can populate immediately.
  - **Key functionality:**
    - **Auth and roles:** No role gating here. Everyone who can reach the endpoint gets the full lists.

- **Inputs (query):** Uses scope=calendar together with lookups=1. Other query params (search, status, from, to, roomId, page, pageSize, userId, companyId, departmentId) are ignored in this branch.
- **Rooms list:** Pulls only rooms where IsActive is true, sorts by RoomName, and projects to { roomId, roomName }.
- **Users list:** Returns all users sorted by FullName, projected as { Id, UserName = FullName, Email }. This is intentionally not restricted by manager status, since the calendar may need to show bookings across users.
- **Response:** 200 OK with { rooms: [...], users: [...] }. Empty arrays are possible if there are no active rooms or no users, which allows the Calendar UI to render an empty state without breaking.

```
[HttpGet]
1 reference
public async Task<IActionResult> GetAsync(
    [FromQuery] int? id,
    [FromQuery] int? lookups,
    [FromQuery] string? scope,
    [FromQuery] bool includeInactive = true,
    [FromQuery] string? search = null,
    [FromQuery] string? status = null,
    [FromQuery] DateTime? from = null,
    [FromQuery] DateTime? to = null,
    [FromQuery] int? roomId = null,
    [FromQuery] int page = 1,
    [FromQuery] int pageSize = 450,
    [FromQuery] int? userId = null,
    [FromQuery] int? companyId = null,
    [FromQuery] int? departmentId = null
)
```

```
// ---------- CALENDAR LOOKUPS (everyone sees the full lists) ----------
// GET /api/BookingsApi?scope=calendar&lookups=1
if (string.Equals(scope, "calendar", StringComparison.OrdinalIgnoreCase) && lookups == 1)
{
    var roomsAll = await _db.Rooms
        .AsNoTracking()
        .Where(r => r.IsActive)
        .OrderBy(r => r.RoomName)
        .Select(r => new { roomId = r.RoomId, roomName = r.RoomName })
        .ToListAsync();

    var usersAll = await _db.Users
        .AsNoTracking()
        .OrderBy(u => u.FullName)
        .Select(u => new { u.Id, UserName = u.FullName, u.Email })
        .ToListAsync();

    return Ok(new { rooms = roomsAll, users = usersAll });
}
```

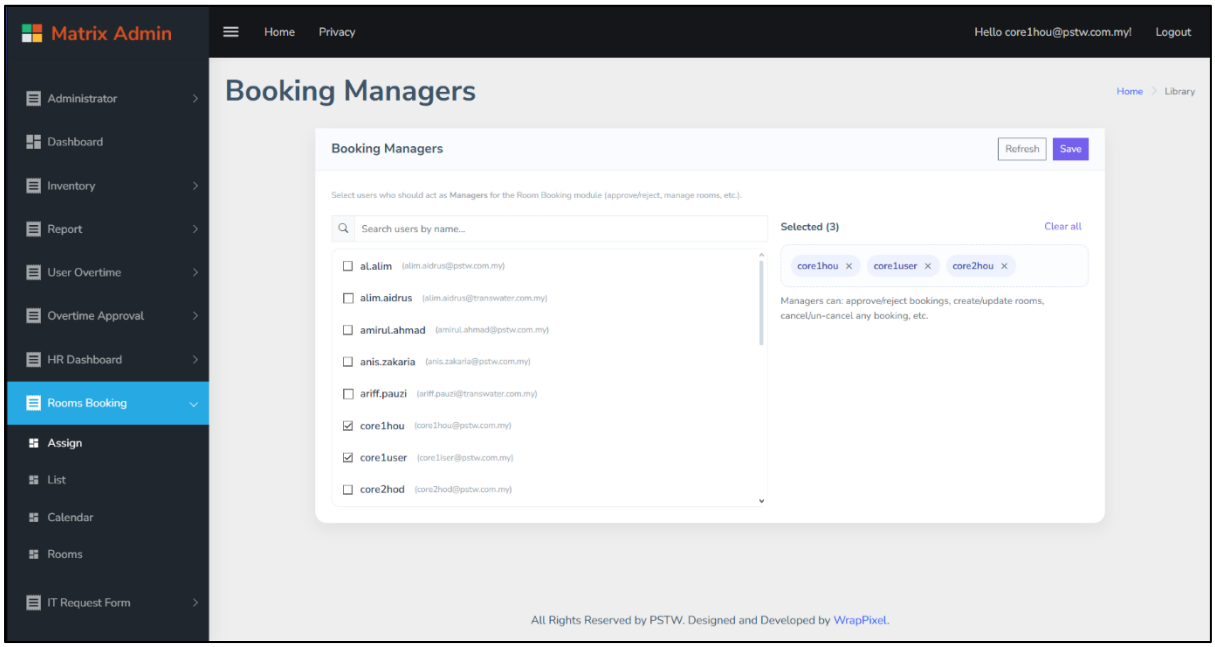
- **[HttpGet] ("Calendar bookings list")**
- **Purpose:** Returns the bookings to paint on the Calendar view. When scope=calendar and lookups!=1, it lists bookings in the requested window with optional filters.
- **Key functionality:**

- **Who can see it:** No role filtering here. The calendar shows all bookings that match the filters.
- **Inputs (query):** Uses scope=calendar plus optional search, status, from, to, roomId, userId, companyId, departmentId, page, and pageSize. Ignores id and lookups in this branch (as long as lookups != 1).
- **Text search:** If search is set, filters Title.Contains(search).
- **Status filter:** Tries to parse status into BookingStatus. On success, narrows to that status. Invalid values are ignored so the page still loads.
- **Date window:** Converts from and to to UTC with CoerceToUtc. Keeps any booking that overlaps the window (EndUtc > fromUtc and StartUtc < toUtc).
- **Entity filters:**
  - roomId: exact room match.
  - userId: booking where the user is either the requester or the target.
  - companyId, departmentId: exact matches on those columns.
- **Paging and sort:** Orders by StartUtc ascending, then applies Skip((page - 1) * pageSize) and Take(pageSize). Defaults are page=1, pageSize=450.
- **Projection and UTC safety:** Projects a lightweight shape for the calendar and tags startUtc and endUtc with DateTimeKind.Utc to prevent timezone bugs.

```
[HttpGet]
1 reference
public async Task<IActionResult> GetAsync(
    [FromQuery] int? id,
    [FromQuery] int? lookups,
    [FromQuery] string? scope,
    [FromQuery] bool includeInactive = true,
    [FromQuery] string? search = null,
    [FromQuery] string? status = null,
    [FromQuery] DateTime? from = null,
    [FromQuery] DateTime? to = null,
    [FromQuery] int? roomId = null,
    [FromQuery] int page = 1,
    [FromQuery] int pageSize = 450,
    [FromQuery] int? userId = null,
    [FromQuery] int? companyId = null,
    [FromQuery] int? departmentId = null
)
```

```
// ---------- CALENDAR LIST (everyone sees all bookings) ----------
// GET /api/BookingsApi?scope=calendar&from=...&to=...
if (string.Equals(scope, "calendar", StringComparison.OrdinalIgnoreCase) && !id.HasValue && lookups != 1)
{
    var qCal = _db.Bookings.AsNoTracking().AsQueryable();

    if (!string.IsNullOrWhiteSpace(search))
        qCal = qCal.Where(x => x.Title.Contains(search));

    if (!string.IsNullOrWhiteSpace(status) && Enum.TryParse<BookingStatus>(status, true, out var stCal))
        qCal = qCal.Where(x => x.CurrentStatus == stCal);

    DateTime? fromUtcCal = from.HasValue ? CoerceToUtc(from.Value) : null;
    DateTime? toUtcCal = to.HasValue ? CoerceToUtc(to.Value) : null;

    if (fromUtcCal.HasValue) qCal = qCal.Where(x => x.EndUtc > fromUtcCal.Value);
    if (toUtcCal.HasValue) qCal = qCal.Where(x => x.StartUtc < toUtcCal.Value);

    if (roomId.HasValue) qCal = qCal.Where(x => x.RoomId == roomId.Value);
    if (userId.HasValue && userId.Value > 0)
        qCal = qCal.Where(x => x.RequestedByUserId == userId.Value || x.TargetUserId == userId.Value);

    if (companyId.HasValue && companyId.Value > 0)
        qCal = qCal.Where(x => x.CompanyId == companyId.Value);

    if (departmentId.HasValue && departmentId.Value > 0)
        qCal = qCal.Where(x => x.DepartmentId == departmentId.Value);

    var list = await qCal
        .OrderBy(x => x.StartUtc)
        .Skip((page - 1) * pageSize)
        .Take(pageSize)
        .Select(b => new
        {
            id = b.BookingId,
            roomId = b.RoomId,
            requestedByUserId = b.RequestedByUserId,
            targetUserId = b.TargetUserId,
            title = b.Title,
            description = b.Purpose,
            startUtc = DateTime.SpecifyKind(b.StartUtc, DateTimeKind.Utc),
            endUtc = DateTime.SpecifyKind(b.EndUtc, DateTimeKind.Utc),
            status = b.CurrentStatus.ToString(),
            note = b.Note
        })
        .ToListAsync();

    return Ok(list);
}
```

- **Managers.cshtml**



This file (Managers.cshtml) is the simple admin screen to pick who counts as a “Manager” for the Room Booking module. It loads all users, shows a searchable list on the left, selected chips on the right, and saves the chosen user ids back to the API.

- **data()**

- **busy, saving, error**: Control loading states and surface any errors at the top of the card.
- **q:** The search keyword bound to the input field.
- **users:** Flat array of { id, name, email } from the server.
- **managerIds:** Array of selected user ids that represent the Manager list.

```
data() {
    return {
        busy: false,
        saving: false,
        error: null,
        q: '',
        users: [],        // {id, name, email}
        managerIds: []    // [int]
    };
},
```

➢ **computed**

- **filteredUsers**: Case-insensitive filter across name and email. Returns all users when the search box is empty.
- **selectedUsers:** Maps managerIds back to full user objects so the chip list can show names cleanly.

```
computed: {
    filteredUsers() {
        const k = (this.q || '').toLowerCase();
        return !k ? this.users : this.users.filter(u =>
            (u.name || '').toLowerCase().includes(k) || (u.email || '').toLowerCase().includes(k)
        );
    },
    selectedUsers() {
        const set = new Set(this.managerIds);
        return this.users.filter(u => set.has(u.id));
    }
},
```

➢ **methods**

- **loadUsers():** GET /api/BookingsApi/users. Expects a simple user list (id, name, email). Throws on non-OK.
- **loadManagers():** GET /api/BookingsApi/managers. Expects an array of ints (user ids) and assigns them to managerIds.
- **loadAll():** Sets busy, clears error, and runs both loaders in parallel. Restores busy in finally so the UI always re-enables.
- **remove(id):** Removes a single id from managerIds. The chip disappears immediately.
- **save():** POST /api/BookingsApi/managers with { userIds: managerIds }. Shows a simple alert on success. If the API returns a JSON error body with message, the method surfaces that string to the user.

```
methods: {
    async loadUsers() {
        const r = await fetch('/api/BookingsApi/users');
        if (!r.ok) throw new Error('Failed to load users');
        this.users = await r.json();
    },
    async loadManagers() {
        const r = await fetch('/api/BookingsApi/managers');
        if (!r.ok) throw new Error('Failed to load managers');
        this.managerIds = await r.json(); // array<int>
    },
    async loadAll() {
        try {
            this.busy = true; this.error = null;
            await Promise.all([this.loadUsers(), this.loadManagers()]);
        } catch (e) {
            this.error = e.message || 'Load failed.';
        } finally {
            this.busy = false;
        }
    },
    remove(id) {
        this.managerIds = this.managerIds.filter(x => x !== id);
    },
    async save() {
        try {
            this.saving = true; this.error = null;
            const r = await fetch('/api/BookingsApi/managers', {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
                body: JSON.stringify({ userIds: this.managerIds })
            });
            if (!r.ok) {
                const j = await r.json().catch(() => ({}));
                throw new Error(j.message || `Save failed (${r.status})`);
            }
            alert('Managers updated.');
        } catch (e) {
            this.error = e.message || 'Save failed.';
        } finally {
            this.saving = false;
        }
    }
},
```

- **mounted()**
  - Calls loadAll() on first paint so the list and selections appear together, minimizing flicker between users and selected chips.

```
async mounted() {
    await this.loadAll();
}
```

  - **BookingsAPI(Managers.cshtml)**

- **[HttpGet("users")] ("Managers – users list")**

- **Purpose:** Supplies the Managers page with the full user directory so you can pick who should become a bookings manager.
- **Key functionality:**
  - **Auth and roles:** No explicit check here. Usually the Razor page itself is protected; this endpoint returns data to whoever can reach it.
  - **Record set:** Reads all users with AsNoTracking(), sorts by FullName.

- **Projection:** Returns a clean shape { id, name, email } to keep the payload small and UI-friendly.
- **Response:** 200 OK with an array of users. Empty array if there are no users.

```
[HttpGet("users")]
0 references
public async Task<IActionResult> UsersAsync()
{
    var list = await _db.Users
        .AsNoTracking()
        .OrderBy(u => u.FullName)
        .Select(u => new { id = u.Id, name = u.FullName, email = u.Email })
        .ToListAsync();
    return Ok(list);
}
```

- **[HttpGet("managers")] ("Managers – current manager IDs")**
- **Purpose:** Tells the page who is currently marked as a bookings manager so the UI can pre-select those users in the list.
- **Key functionality:**
  - **Auth and roles:** No explicit check here. The assumption is the page is already behind your own access control.
  - **Record set:** Reads BookingManager rows where IsActive is true.
  - **Projection:** Returns only UserId values, e.g. [1, 2, 3], which is perfect for checkbox binding.
  - **Response:** 200 OK with an array of ints.

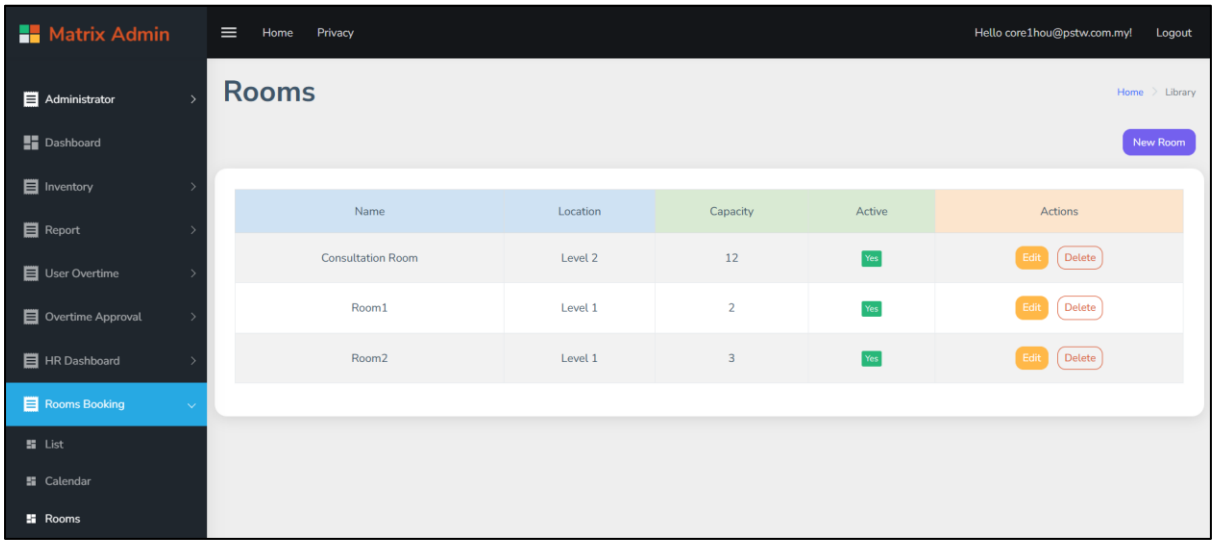
```
[HttpGet("managers")]
0 references
public async Task<IActionResult> GetManagersAsync()
{
    var ids = await _db.BookingManager
        .AsNoTracking()
        .Where(x => x.IsActive)
        .Select(x => x.UserId)
        .ToListAsync();
    return Ok(ids);
}
```

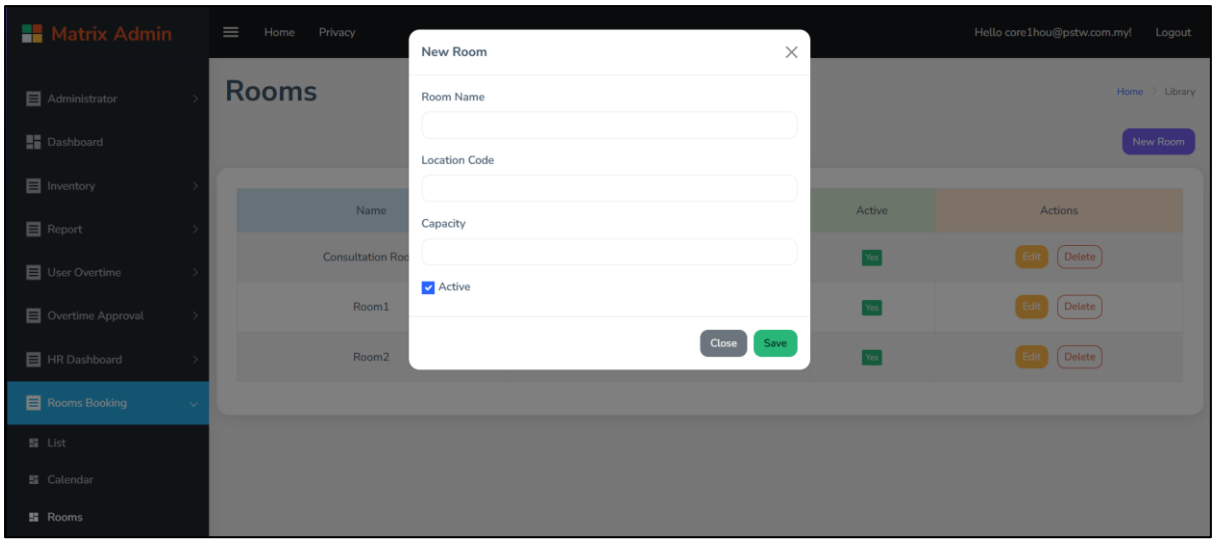
- **[HttpPost("managers")] ("Managers – save selection")**
- **Purpose**: Saves the chosen set of bookings managers from the page. You post a list of user IDs and the server updates BookingManager.

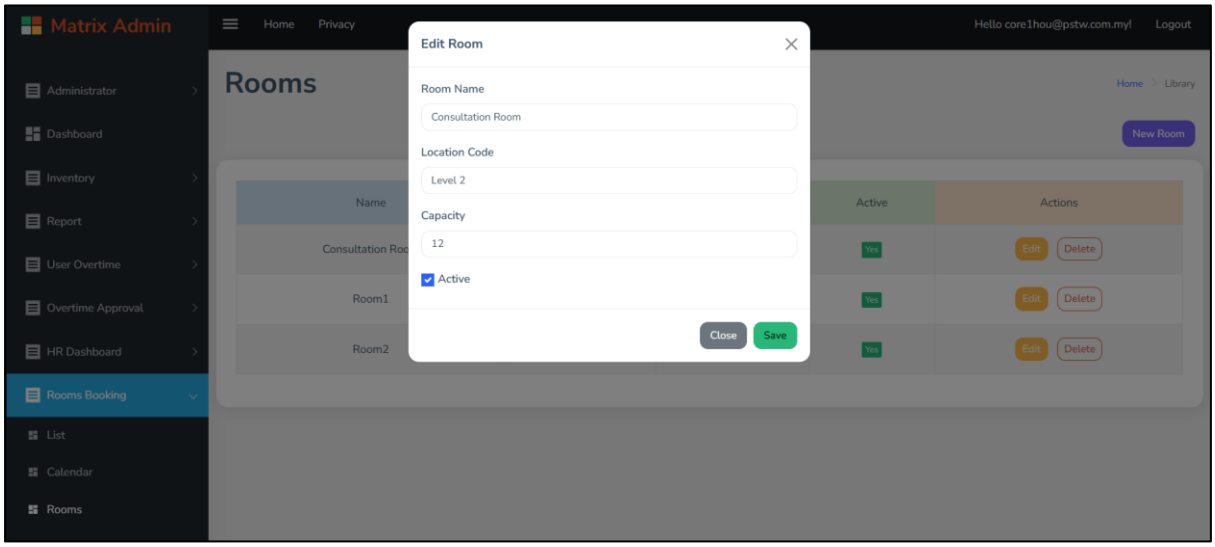
- **Key functionality:**
    - **Auth and roles**: Only a current manager can change the list. If IsManager() is false, returns 401 Unauthorized with “Managers only.”
    - **Inputs (body):** Expects { "userIds": [1,2,3] }. If userIds is missing, returns 400 Bad Request.
    - **Soft reset strategy:**
        - Loads all BookingManager rows and marks them inactive.
        - Builds a distinct set from the posted userIds (ignores non-positive values).
        - For each selected id: upsert. If a row exists, set IsActive = true. If not, insert a new row with IsActive = true, CreatedUtc = now, and CreatedByUserId = me.
    - **Side effects:** Writes to BookingManager only. Does not modify AspNetUsers or roles.
    - **Response:** 200 OK with { saved: true, count: <number selected> }.

```
[HttpPost("managers")]
0 references
public async Task<IActionResult> SaveManagersAsync([FromBody] SaveManagersDto dto)
{
    if (dto?.UserIds is null) return BadRequest("userIds required.");
    // Only an existing manager can edit the list (bootstrap by seeding one row)
    if (!IsManager()) return Unauthorized("Managers only.");

    var now = DateTime.UtcNow;
    var me = GetCurrentUserId();

    // Soft-reset approach: mark all inactive then upsert selected as active
    var all = await _db.BookingManager.ToListAsync();
    foreach (var bm in all) bm.IsActive = false;

    var selected = new HashSet<int>(dto.UserIds.Where(id => id > 0));

    foreach (var uid in selected)
    {
        var existing = all.FirstOrDefault(x => x.UserId == uid);
        if (existing is null)
        {
            _db.BookingManager.Add(new BookingManager
            {
                UserId = uid,
                IsActive = true,
                CreatedUtc = now,
                CreatedByUserId = me
            });
        }
        else
        {
            existing.IsActive = true;
        }
    }

    await _db.SaveChangesAsync();
    return Ok(new { saved = true, count = selected.Count });
}
```

- **Room.cshtml**


Matrix Admin
Home
Privacy
Hello core1hou@pstw.com.my!
Logout
Administrator
Dashboard
Inventory
Report
User Overtime
Overtime Approval
HR Dashboard
Rooms Booking
List
Calendar
Rooms
Rooms
Home
Library
New Room
Name
Location
Capacity
Active
Actions
Consultation Room
Level 2
12
Yes
Edit
Delete
Room1
Level 1
2
Yes
Edit
Delete
Room2
Level 1
3
Yes
Edit
Delete


Matrix Admin
Home
Privacy
Hello core1hou@pstw.com.my!
Logout
Rooms
New Room
Name
Active
Actions
Room1
Room2
Edit
Delete
New Room
Room Name
Location Code
Capacity
Active
Close
Save


Matrix Admin
Home
Privacy
Hello core1hou@pstw.com.my!
Logout
Rooms
New Room
Name
Active
Actions
Room1
Room2
Edit
Delete
Edit Room
Room Name
Consultation Room
Location Code
Level 2
Capacity
12
Active
Close
Save

This file (Room.cshtml) provides a dedicated web page for the admin to **create, edit, and delete rooms**. It displays rooms that have been established in a list and admins can also activate or deactivate a certain room only if that room isn't being used.

- **Constants & DOM refs**
  - **Purpose:** define the API base and cache the table body element used throughout.

```
@section Scripts {
    <script>
        (() => {
            // --- Constants / refs ---
            const api = `${window.location.origin}/api/BookingsApi`;
            const tbody = document.querySelector("#roomsTable tbody");
```

- **UI helpers (alerts & HTML escaping)**
  - **Purpose:** show Bootstrap alerts in the header and safely escape any dynamic strings before injecting into HTML.

```
function showAlert(type, msg) {
    document.getElementById("alerts").innerHTML = `
  <div class="alert alert-${type} alert-dismissible fade show" role="alert">
    ${msg}
    <button type="button" class="btn-close" data-bs-dismiss="alert"></button>
  </div>`;
}

function escapeHtml(s) {
    return String(s ?? "").replace(/[&<>\"']/g, m =>
    ({
        "&": "&amp;",
        "<": "&lt;",
        ">": "&gt;",
        '"': "&quot;",
        "'": "&#39;"
    }[m])
    );
}
```

- **Data load - list rooms**
  - **Purpose:** fetch the rooms list (scope=rooms), render rows, and handle empty/error states. Uses badges for Active/Inactive and wires Edit/Delete buttons inline.

```
async function loadRooms() {
    tbody.innerHTML = `<tr><td colspan="5" class="text-center">Loading…</td></tr>`;
    try {
        const res = await fetch(`${api}?scope=rooms`);
        if (!res.ok) throw new Error(await res.text());

        const data = await res.json();
        if (!Array.isArray(data) || data.length === 0) {
            tbody.innerHTML = `<tr><td colspan="5" class="text-center">No rooms yet.</td></tr>`;
            return;
        }

        tbody.innerHTML = "";
        for (const r of data) {
            const tr = document.createElement("tr");
            tr.innerHTML = `
      <td>${escapeHtml(r.roomName)}</td>
      <td>${escapeHtml(r.locationCode ?? "")}</td>
      <td>${r.capacity ?? ""}</td>
      <td>
        ${r.isActive
                    ? '<span class="badge bg-success">Yes</span>'
                    : '<span class="badge bg-secondary">No</span>'
                }
      </td>
      <td>
        <button class="btn btn-sm btn-warning me-2"
          onclick='openEdit(${r.roomId},"${escapeHtml(
                    r.roomName
                )}","${escapeHtml(r.locationCode ?? "")}",${r.capacity ?? "null"
                },${r.isActive})'>
          Edit
        </button>
        <button class="btn btn-sm btn-outline-danger"
          onclick="deleteRoom(${r.roomId})">
          Delete
        </button>
      </td>`;
            tbody.appendChild(tr);
        }
    } catch (err) {
        tbody.innerHTML = `<tr><td colspan="5" class="text-danger text-center">Failed: ${escapeHtml(
            err.message
        )}</td></tr>`;
    }
}
```

- **Modal helpers - open create & open edit**
  - **Purpose:** reset the form for a new room or prefill values for editing, and (for edit) show the Bootstrap modal immediately.

```
window.openCreate = function () {
    document.getElementById("roomModalTitle").textContent = "New Room";
    document.getElementById("RoomId").value = "";
    document.getElementById("RoomName").value = "";
    document.getElementById("LocationCode").value = "";
    document.getElementById("Capacity").value = "";
    document.getElementById("IsActive").checked = true;
};

window.openEdit = function (id, name, loc, cap, active) {
    document.getElementById("roomModalTitle").textContent = "Edit Room";
    document.getElementById("RoomId").value = id;
    document.getElementById("RoomName").value = name;
    document.getElementById("LocationCode").value = loc === "null" ? "" : loc;
    document.getElementById("Capacity").value = cap === null ? "" : cap;
    document.getElementById("IsActive").checked = !!active;

    const modal = new bootstrap.Modal(document.getElementById("roomModal"));
    modal.show();
};
```

- **Save & Delete actions**

- **Purpose:** submit create/update payloads via POST/PUT and delete rooms via DELETE; all actions reload the list and surface success/error alerts.

```
window.saveRoom = async function () {
    const id = document.getElementById("RoomId").value;
    const roomName = document.getElementById("RoomName").value.trim();
    const locationCode = document.getElementById("LocationCode").value.trim();
    const capacity = document.getElementById("Capacity").value
        ? Number(document.getElementById("Capacity").value)
        : null;
    const isActive = document.getElementById("IsActive").checked;

    if (!roomName) {
        showAlert("danger", "Room Name is required.");
        return;
    }

    try {
        let res;
        const payload = {
            RoomName: roomName,
            LocationCode: locationCode || null,
            Capacity: capacity,
            IsActive: isActive
        };

        if (id) {
            res = await fetch(`${api}?scope=rooms&id=${encodeURIComponent(id)}`, {
                method: "PUT",
                headers: { "Content-Type": "application/json" },
                body: JSON.stringify(payload)
            });
        } else {
            res = await fetch(`${api}?scope=rooms`, {
                method: "POST",
                headers: { "Content-Type": "application/json" },
                body: JSON.stringify(payload)
            });
        }

        if (!res.ok) throw new Error(await res.text());

        showAlert("success", "Room saved.");
        bootstrap.Modal.getInstance(document.getElementById("roomModal"))?.hide();
        await loadRooms();
    } catch (err) {
        showAlert("danger", "Save failed: " + err.message);
    }
};

window.deleteRoom = async function (id) {
    if (!confirm("Delete this room?")) return;
    try {
        const res = await fetch(`${api}?scope=rooms&id=${encodeURIComponent(id)}`, {
            method: "DELETE"
        });
        if (!res.ok) throw new Error(await res.text());

        showAlert("success", "Room deleted.");
        await loadRooms();
    } catch (err) {
        showAlert("danger", "Delete failed: " + (err?.message || ""));
    }
};
```

- **Bootstrapping**
  - **Purpose:** load the rooms list when the DOM is ready.

```
        document.addEventListener("DOMContentLoaded", loadRooms);
    })();
</script>
```

- **BookingsAPI(Room.cshtml)**

- **[HttpGet] Rooms - list**

- **Purpose:** Returns rooms for the Room page grid and dropdowns. Works as admin tooling but is open to any authenticated user.
- **Key functionality:**
  - **Inputs (query):** Uses scope=rooms. Respects includeInactive.
  - **Filtering:** If includeInactive is false, only active rooms are returned.
  - **Projection:** { roomId, roomName, locationCode, capacity, isActive }, ordered by roomName.
  - **Response:** 200 OK with an array. Empty array if there are no rooms.

```
[HttpGet]
1 reference
public async Task<IActionResult> GetAsync(
    [FromQuery] int? id,
    [FromQuery] int? lookups,
    [FromQuery] string? scope,
    [FromQuery] bool includeInactive = true,
    [FromQuery] string? search = null,
    [FromQuery] string? status = null,
    [FromQuery] DateTime? from = null,
    [FromQuery] DateTime? to = null,
    [FromQuery] int? roomId = null,
    [FromQuery] int page = 1,
    [FromQuery] int pageSize = 450,
    [FromQuery] int? userId = null,
    [FromQuery] int? companyId = null,
    [FromQuery] int? departmentId = null
)
```

```
// ROOMS LIST (admin tooling / open to authenticated)
if (string.Equals(scope, "rooms", StringComparison.OrdinalIgnoreCase))
{
    var rq = _db.Rooms.AsNoTracking().AsQueryable();
    if (!includeInactive) rq = rq.Where(r => r.IsActive);
    var rooms = await rq
        .OrderBy(r => r.RoomName)
        .Select(r => new
        {
            roomId = r.RoomId,
            roomName = r.RoomName,
            locationCode = r.LocationCode,
            capacity = r.Capacity,
            isActive = r.IsActive
        })
        .ToListAsync();
    return Ok(rooms);
}
```

- **[HttpPost]Rooms - create**
- **Purpose:** Lets managers add a new room from the Room page.
- **Key functionality:**
    - **Auth and roles:** Managers only. Returns Unauthorized if caller is not a manager.
    - **Inputs (body):** CreateRoomDto with RoomName required. Optional LocationCode, Capacity, IsActive.
    - **Uniqueness check:** Rejects if a room with the same RoomName already exists.
    - **Insert:** Trims strings, normalizes LocationCode to null if empty, then inserts and saves.
    - **Response:** 200 OK with { roomId } on success. 400 Bad Request when RoomName is missing. 409 Conflict when name already exists.

```
[HttpPost]
0 references
public async Task<IActionResult> PostAsync(
    [FromQuery] string? action,
    [FromQuery] int? id,
    [FromQuery] string? scope,
    [FromBody] System.Text.Json.JsonElement? body // nullable is fine
)
{
```

```
// ROOMS: CREATE (Managers only)
if (string.Equals(scope, "rooms", StringComparison.OrdinalIgnoreCase))
{
    if (!IsManager()) return Unauthorized("Managers only.");

    var dto = JsonSerializer.Deserialize<CreateRoomDto>(json, JsonOpts);
    if (dto is null || string.IsNullOrWhiteSpace(dto.RoomName))
        return BadRequest("RoomName is required.");

    var exists = await _db.Rooms.AnyAsync(r => r.RoomName == dto.RoomName);
    if (exists) return Conflict("A room with this name already exists.");

    var room = new Room
    {
        RoomName = dto.RoomName.Trim(),
        LocationCode = string.IsNullOrWhiteSpace(dto.LocationCode) ? null : dto.LocationCode.Trim(),
        Capacity = dto.Capacity,
        IsActive = dto.IsActive
    };
    _db.Rooms.Add(room);
    await _db.SaveChangesAsync();
    return Ok(new { roomId = room.RoomId });
}
```

- **[HttpPut]Rooms - update**
- **Purpose:** Lets managers edit an existing room's details.
- **Key functionality:**

- **Auth and roles:** Managers only.
- **Inputs (query/body):** Requires id in query and an UpdateRoomDto body. All fields are optional and only provided fields are updated.
- **Record lookup:** Returns NotFound if the room does not exist.
- **Update rules:** Trims text fields, normalizes LocationCode to null when blank, updates Capacity and IsActive if provided.
- **Response:** 200 OK with { roomId } after save.

```
[HttpPut]
0 references
public async Task<IActionResult> PutAsync(
    [FromQuery] int? id,
    [FromQuery] string? scope,
    [FromBody] System.Text.Json.JsonElement? body
)
```

```
// ROOMS: UPDATE (Managers only)
if (string.Equals(scope, "rooms", StringComparison.OrdinalIgnoreCase))
{
    if (!IsManager()) return Unauthorized("Managers only.");

    var dto = JsonSerializer.Deserialize<UpdateRoomDto>(json, JsonOpts);

    var r = await _db.Rooms.FirstOrDefaultAsync(x => x.RoomId == id.Value);
    if (r is null) return NotFound();

    if (dto is not null)
    {
        if (dto.RoomName is not null) r.RoomName = dto.RoomName.Trim();
        if (dto.LocationCode is not null) r.LocationCode = string.IsNullOrWhiteSpace(dto.LocationCode) ? null : dto.LocationCode.Trim();
        if (dto.Capacity.HasValue) r.Capacity = dto.Capacity.Value;
        if (dto.IsActive.HasValue) r.IsActive = dto.IsActive.Value;
    }
    await _db.SaveChangesAsync();
    return Ok(new { roomId = r.RoomId });
}
```

- **[HttpDelete]Rooms - hard delete**
- **Purpose:** Lets managers permanently delete a room. Use with care.
- **Key functionality:**
  - **Auth and roles:** Managers only.
  - **Inputs (query):** Requires id.
  - **Delete behavior:** Attempts a hard delete. If the room is referenced by bookings or blocked by constraints, returns a conflict message advising to deactivate instead.
  - **Response:**
    - 200 OK with { deleted: true } on success.
    - 409 with { error: "Cannot delete: room is in use or active. Deactivate it instead." } when foreign keys or constraints prevent deletion.
    - 500 with { error: "Delete failed." } for unexpected errors.

```
[HttpDelete]
0 references
public async Task<IActionResult> DeleteAsync([FromQuery] int? id, [FromQuery] string? scope)
{
    if (!id.HasValue || id <= 0) return BadRequest("Missing id.");

    // ROOMS: HARD DELETE (Managers only)
    if (string.Equals(scope, "rooms", StringComparison.OrdinalIgnoreCase))
    {
        if (!IsManager()) return Unauthorized("Managers only.");

        var r = await _db.Rooms.FirstOrDefaultAsync(x => x.RoomId == id.Value);
        if (r is null) return NotFound();

        try
        {
            _db.Rooms.Remove(r);
            await _db.SaveChangesAsync();
            return Ok(new { deleted = true });
        }
        catch (DbUpdateException)
        {
            // Likely foreign key in use (existing bookings) or DB constraint
            return StatusCode(409, new { error = "Cannot delete: room is in use or active. Deactivate it instead." });
        }
        catch (Exception)
        {
            return StatusCode(500, new { error = "Delete failed." });
        }
    }
```

CONTROLLER

- **BookingsController.cs**

  The controller has "Action Methods" that handle specific tasks when a user navigates through the booking system.

  Index( )

  Purpose: This function is to show the main bookings list page.

  How it works: When someone clicks on “Rooms Booking” in the sidebar then click on “List”, this function runs. It simply tells the system to display the Index.cshtml page. The page itself loads booking records from the BookingsAPI endpoint, with options for filtering and pagination.

  Room( )

  Purpose: This function is to show the main room overview page.

  How it works:

When a user navigates to “Rooms” in the sidebar, this function runs. It displays the Room.cshtml page, where users can see available rooms and their details. Data is retrieved dynamically from the BookingsAPI endpoint.

RoomsCreate( )

Purpose: This function is to show the page for creating or editing rooms.

How it works:

When an admin selects “Add Room” or chooses to edit an existing room, this function runs. It opens the RoomsCreate.cshtml page. When editing, the page fetches room details via BookingsAPI/Get and pre-fills the form for updating. When adding, the form remains blank for new data entry.

Calendar( )

Purpose: This function is to show the calendar view of bookings.

How it works:

When someone clicks on “Calendar” in the sidebar, this function runs. It displays the Calendar.cshtml page, which renders a monthly view of all bookings. Users can click a specific date, and the system will fetch bookings for that date from the BookingsAPI/List endpoint.

Create(int? id)

Purpose: This function is to show the page for creating or editing a booking.

How it works:

When a user clicks “Create New” or chooses to edit an existing booking, this function runs. It shows the Create.cshtml page. If no id is provided, the page opens as a blank form for creating a new booking. If an id is provided, it will be passed into ViewBag.Id and used by the JavaScript in the view to call BookingsAPI/Get and load the booking’s details for editing.

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace PSTW_CentralSystem.Areas.Bookings.Controllers
{
    [Area("Bookings")]
    [Authorize]
    0 references
    public class BookingsController : Controller
    {
        0 references
        public IActionResult Index() => View();

        0 references
        public IActionResult Room() => View();

        // Rooms create/edit
        0 references
        public IActionResult RoomsCreate() => View();

        0 references
        public IActionResult Calendar() => View();

        // Create/Edit page. If id is provided, page loads
        0 references
        public IActionResult Create(int? id)
        {
            ViewBag.Id = id;
            return View();
        }

    }
}
```