Alright—let’s walk through this Angular component *line by line* (grouped by purpose so it’s readable) and then tie it all together with the control flow and “why” behind each choice. I’ll also point out pitfalls and easy improvements where relevant.

# **Imports**

import {  
 Component, OnInit, AfterViewInit,  
 Renderer2, ElementRef, ViewChild,  
 OnDestroy, ViewEncapsulation, HostBinding   
} from '@angular/core';

* **Component**: decorator to declare an Angular component.
* **OnInit / AfterViewInit / OnDestroy**: lifecycle interfaces; you implement their hooks (ngOnInit, ngAfterViewInit, ngOnDestroy).
* **Renderer2**: Angular-safe DOM API for creating listeners / setting attributes / classes without touching the native DOM directly (important for SSR and testability).
* **ElementRef**: reference to the host element of this component.
* **ViewChild**: pulls a template reference (#targetDiv) into the class.
* **HostBinding**: binds class properties to host element classes (adds/removes CSS classes).
* **ViewEncapsulation** is imported but not used—safe to remove.

import { CalendarResponsiveOptions } from 'primeng/calendar';

* Type from PrimeNG to configure responsive month layout.

import moment from 'moment';

* Date math helper; used heavily for ISO week handling.

import { isSameDay } from '@app/utils/helpers';

* Utility to compare two Dates ignoring time (your own helper).

# **Component decorator**

@Component({  
 selector: 'app-calendar',  
 templateUrl: './calendar.component.html',  
 styleUrls: ['./calendar.component.css'],  
})

* **selector**: HTML tag to use this component.
* **templateUrl** / **styleUrls**: external template and CSS.

# **Class signature and property fields**

export class CalendarComponent implements OnInit, AfterViewInit, OnDestroy {

Implements the lifecycle hooks the code uses.

@ViewChild('targetDiv', { static: false }) targetDiv!: ElementRef;

* Binds the element tagged #targetDiv from the template. static:false means it’s resolved after view init. The ! asserts it’ll exist.

### **Host-bound CSS flags (toggle classes on the host <app-calendar>)**

@HostBinding('class.booked-active') bookedActive: boolean = false;  
 @HostBinding('class.reserved-active') reservedActive: boolean = false;  
 @HostBinding('class.waiting-active') waitingActive: boolean = false;  
 @HostBinding('class.available-active') availableActive: boolean = false;  
 @HostBinding('class.leadtime-active') leadtimeActive: boolean = false; // Legacy support

* These booleans directly add/remove CSS classes on the host element depending on the selected week’s status. leadtime-active mirrors “booked” for legacy styles.

### **Core state**

today = new Date();  
 selectedRangeDates: Date[] | undefined;  
 fromDate: Date | undefined;  
 tillDate: Date | undefined;  
 maxDate: Date = new Date(new Date().getFullYear() + 2, 11, 31); // 2 years from now

* **today**: snapshot of current time when the component is constructed.
* **selectedRangeDates**: two-element array [from, to] for the user’s selection.
* **fromDate / tillDate**: same as selectedRangeDates but normalized to Monday–Sunday for week selection.
* **maxDate**: calendar upper bound (Dec 31 two years out).

### **Demo datasets**

bookingDates: Date[] = [];  
 reservedDates: Date[] = [];  
 waitingDates: Date[] = [];

* Arrays of dates that mark “booked”, “reserved”, “waiting list”. For demo, they get filled with weekly blocks.

### **UI flags and responsive options**

isLoading: boolean = false;  
 calendarResponsiveOptions: CalendarResponsiveOptions[] = [];

* **isLoading**: not used here but typical for async.
* **calendarResponsiveOptions**: PrimeNG responsive configuration.

### **Week selection trackers**

yearOfSelectedWeek: number = moment().year();  
 monthOfSelectedWeek: number = moment().month() + 1;  
 firstSelectedWeek: number | null = null;  
 lastSelectedWeek: number | null = null;

* Keep track of the calendar viewport’s **year** and **month** (month stored as 1–12 here because of +1).
* **firstSelectedWeek**: the week number when the user clicks the first time.
* **lastSelectedWeek**: the week number when the user clicks the second time to complete a range.

### **Status flags for the selected range**

invalidDate: boolean = false;  
 errorDate: boolean = false;  
 reservedDate: boolean = false;  
 waitingListDate: boolean = false;

* These represent outcome of validation:
  + **errorDate** ⇒ “booked” conflict for the selected week(s).
  + **reservedDate** ⇒ selected week(s) are reserved.
  + **waitingListDate** ⇒ waiting list.
  + **invalidDate** is defined but never set in this code.

# **Constructor**

constructor(  
 private renderer: Renderer2,  
 private el: ElementRef  
 ) {  
 console.log('CalendarComponent constructor called');  
 }

* Injects Renderer2 and the host ElementRef.
* Logs construction.

# **Lifecycle: ngOnInit**

ngOnInit(): void {  
 console.log('CalendarComponent ngOnInit called');  
 this.calendarResponsiveOptions = [  
 { breakpoint: '1528px', numMonths: 2 },  
 { breakpoint: '1527px', numMonths: 1 }  
 ];  
 console.log('Calendar responsive options set:', this.calendarResponsiveOptions);  
 this.generateDummyDateData();  
 }

* Sets responsive behavior for the PrimeNG Calendar (show 2 months above 1528px, otherwise 1).
* Builds sample “booked/reserved/waiting” weeks with generateDummyDateData().

# **Lifecycle: ngOnDestroy**

ngOnDestroy(): void {  
 console.log('CalendarComponent ngOnDestroy called');  
 }

* Just logs. (⚠️ Improvement: should also clean up event listeners & disconnect the MutationObserver. More on this below.)

# **Lifecycle: ngAfterViewInit**

ngAfterViewInit() {  
 console.log('CalendarComponent ngAfterViewInit called');  
 const observer = new MutationObserver((mutations) => {  
 console.log('MutationObserver detected changes:', mutations.length, 'mutations');  
 mutations.forEach((mutation) => {  
 if (mutation.addedNodes) {  
 this.handleAddedNodes(mutation.addedNodes);  
 }  
 });  
 });  
  
 observer.observe(this.el.nativeElement, { childList: true, subtree: true });  
 console.log('MutationObserver started watching for DOM changes');  
 }

* Creates a MutationObserver to watch the component subtree for DOM nodes being added.
* Why? PrimeNG’s Calendar generates week number cells dynamically; you hook into those once they appear to attach listeners and add hover styling.
* Calls handleAddedNodes for any added nodes.

⚠️ **Important improvement**: store observer on the instance and call observer.disconnect() in ngOnDestroy. Otherwise the observer keeps running and may leak.

# **Demo data generation**

private generateDummyDateData(): void {  
 console.log('Generating dummy date data...');  
 const today = new Date();  
   
 // booked: next Monday → Sunday  
 const nextMonday = this.getNextMonday(today);  
 for (let i = 0; i < 7; i++) { ... this.bookingDates.push(date); }  
  
 // reserved: 2 weeks later, Monday → Sunday  
 const reservedStart = this.getNextMonday(today);  
 reservedStart.setDate(reservedStart.getDate() + 14);  
 for (let i = 0; i < 7; i++) { ... this.reservedDates.push(date); }  
  
 // waiting: 3 weeks later, Monday → Sunday  
 const waitingStart = this.getNextMonday(today);  
 waitingStart.setDate(waitingStart.getDate() + 21);  
 for (let i = 0; i < 7; i++) { ... this.waitingDates.push(date); }  
  
 console.log('Dummy data generated:', ...);  
 }

* Builds three full weeks (Mon–Sun) of dates for the three statuses, spaced 0, 2, and 3 weeks ahead.

# **Week helpers: determine Monday/Sunday**

private getNextMonday(date: Date): Date {  
 const result = new Date(date);  
 const day = result.getDay();  
 const diff = day === 0 ? 1 : 8 - day;  
 result.setDate(result.getDate() + diff);  
 return result;  
 }

* JavaScript getDay() is 0(Sun)–6(Sat).
* Calculates days to the *next* Monday. If Sunday, add 1; otherwise add 8 - currentDay.

private getMondayOfWeek(date: Date): Date {  
 const result = new Date(date);  
 const day = result.getDay();  
 const diff = result.getDate() - day + (day === 0 ? -6 : 1);  
 result.setDate(diff);  
 return result;  
 }

* Returns the Monday of the week containing date. If Sunday (day===0), subtract 6 to land on previous Monday.

private getSundayOfWeek(date: Date): Date {  
 const monday = this.getMondayOfWeek(date);  
 const sunday = new Date(monday);  
 sunday.setDate(monday.getDate() + 6);  
 return sunday;  
 }

* Monday + 6 days.

# **DOM mutation handling (wiring up week cells)**

private handleAddedNodes(nodes: any) {  
 console.log('handleAddedNodes called with', nodes.length, 'nodes');  
 nodes.forEach((node: any) => {  
 if (node.nodeType === 1 && node.classList.contains('p-datepicker-weeknumber')) {  
 ...  
 }  
 });  
 }

* Iterates new nodes; filters to elements with p-datepicker-weeknumber (PrimeNG’s week number column).

Inside the if:

const weekNumber = parseInt(node.innerText, 10);  
 const currentYear = moment(this.today).year();  
 const currentWeekNumber = moment(this.today).isoWeek();  
  
 if (this.yearOfSelectedWeek <= currentYear && weekNumber < currentWeekNumber) {  
 this.renderer.setAttribute(node.querySelector('span'), 'style', 'font-weight: normal !important');  
 }

* Reads the ISO week number text.
* Finds “current week number” using Moment’s isoWeek().
* If this week is in the *past* relative to current year/week, it lightens the weight (visual cue that past weeks are disabled).

this.renderer.listen(node, 'click', this.handleWeekClick.bind(this));

* Adds click handler to select weeks. (Using Renderer2 is the Angular-safe way.)

### **Hover handlers**

this.renderer.listen(node, 'mouseover', (event) => {  
 const weekNumber = parseInt(event.target.innerText, 10);  
  
 const nodeDate = moment().year(this.yearOfSelectedWeek).isoWeek(weekNumber).endOf('isoWeek');  
 const todaysWeek = moment(this.today);  
 const currentDayOfWeek = moment(this.today).day();  
  
 // remove any hover classes first  
 this.renderer.removeClass(node, 'hover-week');  
 this.renderer.removeClass(node, 'hover-week-booked');  
 this.renderer.removeClass(node, 'hover-week-reserved');  
 this.renderer.removeClass(node, 'hover-week-waiting');  
  
 if (nodeDate.isBefore(todaysWeek) || (nodeDate.isSame(moment().endOf('isoWeek')) && currentDayOfWeek !== 1)) {  
 // disabled: past weeks or current week before Monday  
 return;  
 }

* Computes the end of the hovered ISO week.
* Clears previous hover classes.
* Blocks hover highlight if the week is in the past or it’s the current week but *not* Monday yet (so users can’t choose it prematurely).

const weekStart = moment().year(this.yearOfSelectedWeek).isoWeek(weekNumber).startOf('isoWeek').toDate();  
 const weekEnd = moment().year(this.yearOfSelectedWeek).isoWeek(weekNumber).endOf('isoWeek').toDate();  
  
 if (this.isWeekBooked(weekStart, weekEnd)) {  
 this.renderer.addClass(node, 'hover-week-booked');  
 } else if (this.isWeekReserved(weekStart, weekEnd)) {  
 this.renderer.addClass(node, 'hover-week-reserved');  
 } else if (this.isWeekWaiting(weekStart, weekEnd)) {  
 this.renderer.addClass(node, 'hover-week-waiting');  
 } else {  
 this.renderer.addClass(node, 'hover-week');  
 }

* Figures out the hovered week’s status (any day in that week matching a corresponding array) and adds a hover class accordingly.

this.renderer.listen(node, 'mouseout', () => {  
 this.renderer.removeClass(node, 'hover-week');  
 this.renderer.removeClass(node, 'hover-week-booked');  
 this.renderer.removeClass(node, 'hover-week-reserved');  
 this.renderer.removeClass(node, 'hover-week-waiting');  
 });

* Clears hover classes when leaving the week cell.

⚠️ **Improvement**: renderer.listen returns an **unlisten** function. Capture and call these in ngOnDestroy to prevent leaks.

# **Week status checks (for hover)**

private isWeekBooked(startDate: Date, endDate: Date): boolean {  
 let currentDate = new Date(startDate);  
 while (currentDate <= endDate) {  
 if (this.bookingDates.some(d => isSameDay(d, currentDate))) return true;  
 currentDate.setDate(currentDate.getDate() + 1);  
 }  
 return false;  
 }  
 private isWeekReserved(startDate: Date, endDate: Date): boolean { ... }  
 private isWeekWaiting(startDate: Date, endDate: Date): boolean { ... }

* Iterate from Monday → Sunday; return true if *any* day matches the array.
* Uses isSameDay to avoid time issues.

# **Respond to calendar month changes**

onMonthChange(event: any) {  
 console.log('onMonthChange called:', event);  
 this.yearOfSelectedWeek = event.year;  
 this.monthOfSelectedWeek = event.month;  
 console.log('Updated calendar view to:', this.monthOfSelectedWeek + '/' + this.yearOfSelectedWeek);  
 }

* Updates the calendar viewport’s year & month.
* **Note**: PrimeNG’s month is often **0–11**. Elsewhere you used moment().month() + 1. This mismatch can cause confusion. Here it’s only used for logging, so harmless—but be consistent.

# **Handling week clicks**

private handleWeekClick(event: any) {  
 const weekNumber = parseInt(event.target.innerText, 10);

* Reads the clicked week number (from the .p-datepicker-weeknumber cell).

const previousDay = moment(this.today).subtract(1, 'day');  
 const currentWeekNumber = previousDay.isoWeek();  
 const currentYear = moment().year();

* Takes “yesterday” and gets its ISO week number.

**Why subtract 1 day?**

To avoid edge cases around Mondays? Practically this makes the “current week” gate slightly stricter; it’s a heuristic to ensure you can’t select the current week *until* Monday (reinforced below).

this.monthOfSelectedWeek = moment().year(this.yearOfSelectedWeek).isoWeek(weekNumber).endOf('isoWeek').month() + 1;

* Stores the 1–12 month that contains the selected week’s **end**.

if ((weekNumber < currentWeekNumber && this.yearOfSelectedWeek === currentYear) || this.yearOfSelectedWeek < currentYear) {  
 // Disallow past weeks  
 return;  
 }  
  
 if (currentWeekNumber === weekNumber && !moment().isSame(moment().day("Monday").week(weekNumber))) {  
 // Disallow selecting current week before Monday  
 return;  
 }

* Prevent selecting past weeks.
* Prevent selecting the current week *before* Monday.

**Note**: moment().day("Monday").week(weekNumber) uses locale weeks, not ISO weeks; elsewhere you use ISO. Prefer isoWeekday(1) / isoWeek(weekNumber) for consistency.

const weekCell = (event.currentTarget as HTMLElement) ?? event.target.closest('.p-datepicker-weeknumber');  
 if (weekCell) {  
 this.renderer.removeClass(weekCell, 'hover-week');  
 this.renderer.addClass(weekCell, 'hover-week');  
 }

* Visually marks the clicked week.

if (this.firstSelectedWeek !== null && this.fromDate) {  
 this.handleSecondWeekClick(weekNumber);  
 } else {  
 this.handleFirstWeekClick(weekNumber);  
 }

* First click starts a range (one full week).
* Second click completes a multi-week range.

# **First & second week selection**

private handleFirstWeekClick(weekNumber: number) {  
 this.firstSelectedWeek = weekNumber;  
 const year = this.yearOfSelectedWeek;  
  
 const fromDate = moment().year(year).isoWeek(weekNumber).startOf('isoWeek').toDate();  
 const tillDate = moment().year(year).isoWeek(weekNumber).endOf('isoWeek').toDate();  
  
 this.calendarWeekBookingValidation(fromDate, tillDate);  
 }

* Always snaps to the full Monday–Sunday.

private handleSecondWeekClick(weekNumber: number) {  
 this.lastSelectedWeek = weekNumber;  
 const year = this.yearOfSelectedWeek;  
  
 const fromDate = moment(this.fromDate).startOf('isoWeek');  
 const tillDate = moment().year(year).isoWeek(this.lastSelectedWeek).endOf('isoWeek');  
   
 if (tillDate.isBefore(fromDate) || fromDate.isAfter(tillDate)) {  
 // If user clicked an earlier week second, swap logic: treat as a new single-week selection  
 [this.firstSelectedWeek, this.lastSelectedWeek] = [this.lastSelectedWeek, null];  
 const startWeek: number = this.firstSelectedWeek!;  
 const fromDate = moment().year(year).isoWeek(startWeek).startOf('isoWeek').toDate();  
 const tillDate = moment().year(year).isoWeek(startWeek).endOf('isoWeek').toDate();  
 this.calendarWeekBookingValidation(fromDate, tillDate);  
 return;  
 }  
   
 if (this.fromDate) {  
 this.calendarWeekBookingValidation(this.fromDate, tillDate.toDate());  
 }  
 this.firstSelectedWeek = null;  
 this.lastSelectedWeek = null;  
 }

* If the second click is “before” the first, it resets to a single-week selection (the second week).
* Otherwise it validates the multi-week range from the first week’s Monday to the second week’s Sunday.
* Clears the temporary week markers after completing the range.

# **Normalize selection to full weeks and validate**

private calendarWeekBookingValidation(fromDate: Date, tillDate: Date) {  
 const adjustedFromDate = this.getMondayOfWeek(fromDate);  
 const adjustedTillDate = this.getSundayOfWeek(tillDate);  
   
 this.fromDate = adjustedFromDate;  
 this.tillDate = adjustedTillDate;  
 this.selectedRangeDates = [adjustedFromDate, adjustedTillDate];  
  
 this.updateDateStatus();  
 }

* Regardless of inputs, clamp to Monday–Sunday.
* Sets internal selection state.
* Recomputes the “status” flags + host classes via updateDateStatus().

# **Handling manual date range changes (from the calendar UI)**

onChangeSelectedRangeDates(): void {  
 this.firstSelectedWeek = null;  
 this.lastSelectedWeek = null;  
   
 if (this.selectedRangeDates && this.selectedRangeDates.length === 2 && this.selectedRangeDates[1]) {  
 const adjustedFromDate = this.getMondayOfWeek(this.selectedRangeDates[0]);  
 const adjustedTillDate = this.getSundayOfWeek(this.selectedRangeDates[1]);  
 this.selectedRangeDates = [adjustedFromDate, adjustedTillDate];  
 this.fromDate = adjustedFromDate;  
 this.tillDate = adjustedTillDate;  
 this.calendarWeekBookingValidation(adjustedFromDate, adjustedTillDate);  
 } else if (this.selectedRangeDates && this.selectedRangeDates[0]) {  
 const adjustedFromDate = this.getMondayOfWeek(this.selectedRangeDates[0]);  
 const adjustedTillDate = this.getSundayOfWeek(this.selectedRangeDates[0]);  
 this.fromDate = adjustedFromDate;  
 this.tillDate = adjustedTillDate;  
 this.selectedRangeDates = [adjustedFromDate, adjustedTillDate];  
 this.updateDateStatus();  
 }  
 }

* If a user uses the datepicker to directly choose a range:
  + Snap it to full weeks and validate.
* If they pick only one date:
  + Auto-complete the full week containing that date.

# **Utility: is a given day within the selected range?**

private inSelectedRange(date: Date): boolean {  
 if (!this.fromDate || !this.tillDate) return false;  
 const d = new Date(date.getFullYear(), date.getMonth(), date.getDate()); // strip time  
 return d >= this.fromDate && d <= this.tillDate;  
 }

* Used by per-cell class/style computations.

# **Per-day class calculation (for the template)**

getDateClass(date: { day: number; month: number; year: number }): any {  
 const dateObj = new Date(date.year, date.month, date.day);

* Builds a Date from calendar’s date object.
  + ⚠️ **Month indexing**: new Date(year, month, day) expects **0–11**. PrimeNG typically provides a 0-based month in date templates, but if you feed 1–12 you’ll shift by one month. Make sure the template hands a 0-based month.

let classes: any = {};  
 const inRange = this.inSelectedRange(dateObj);  
  
 if (this.bookingDates.some(d => isSameDay(d, dateObj))) {  
 classes['date-booked'] = true;  
 if (inRange) classes['selected-booked'] = true;  
 } else if (this.reservedDates.some(d => isSameDay(d, dateObj))) {  
 classes['date-reserved'] = true;  
 if (inRange) classes['selected-reserved'] = true;  
 } else if (this.waitingDates.some(d => isSameDay(d, dateObj))) {  
 classes['date-waiting'] = true;  
 if (inRange) classes['selected-waiting'] = true;  
 } else {  
 classes['date-available'] = true;  
 if (inRange) classes['selected-available'] = true;  
 }  
 return classes;  
 }

* Priority order per day: **booked > reserved > waiting > available**.
* Adds a selected-\* variant if the day falls within the currently selected week range.

# **Per-day inline style calculation**

getDateStyle(date: { day: number; month: number; year: number }): any {  
 const dateObj = new Date(date.year, date.month, date.day);  
 const isInSelectedRange = this.inSelectedRange(dateObj);  
 if (isInSelectedRange) {  
 return {}; // Let host classes style the selected week block  
 }  
  
 // For non-selected dates, set default inline styles based on status  
 if (this.bookingDates.some((d) => isSameDay(d, dateObj))) {  
 return { color: 'red', 'text-decoration': 'line-through' };  
 } else if (this.waitingDates.some((d) => isSameDay(d, dateObj))) {  
 return { color: 'orange', 'text-decoration': 'line-through' };  
 } else if (this.reservedDates.some((d) => isSameDay(d, dateObj))) {  
 return { color: 'blue' };  
 } else {  
 return {};  
 }  
 }

* For the currently selected range, inline styles are suppressed so that the **host-bound classes** can uniformly style the week block.
* For dates outside the selection, applies inline styles for quick visual cues.

**Design note**: You’re mixing CSS classes and inline styles. That’s fine, but it’s cleaner to keep everything in classes if possible (easier theming and overrides).

# **Week-level status helpers (validate selected weeks)**

isDateBooked(dates: Date[]): boolean { ... }  
 isDateReserved(dates: Date[]): boolean { ... }  
 isDateInWaiting(dates: Date[]): boolean { ... }

* Each takes the [from, to] pair and scans Monday→Sunday for any matching days in the corresponding arrays.
* They internally normalize the input to Monday/Sunday again (defensive).

# **Compute and apply status (host-bound classes)**

private updateDateStatus(): void {  
 this.resetHostClasses();  
  
 if (this.selectedRangeDates?.[1]) {  
 const isBooked = this.isDateBooked(this.selectedRangeDates);  
 const isReserved = this.isDateReserved(this.selectedRangeDates);  
 const isWaiting = this.isDateInWaiting(this.selectedRangeDates);  
  
 // Reset flags  
 this.errorDate = false;  
 this.reservedDate = false;  
 this.waitingListDate = false;  
  
 // Priority: booked > waiting > reserved > available  
 if (isBooked) {  
 this.errorDate = true;  
 this.bookedActive = true;  
 this.leadtimeActive = true; // legacy mirroring  
 } else if (isWaiting) {  
 this.waitingListDate = true;  
 this.waitingActive = true;  
 } else if (isReserved) {  
 this.reservedDate = true;  
 this.reservedActive = true;  
 } else {  
 this.availableActive = true;  
 }  
 } else {  
 this.resetHostClasses();  
 }  
 }

* Determines the **overall status of the chosen week block** and flips host-bound classes accordingly.
* Note the priority order here is **booked > waiting > reserved > available** (slightly different from getDateClass which uses booked > reserved > waiting > available). If a week is both reserved and waiting in different days (weird in real life), “waiting” wins over “reserved” here.

private resetHostClasses(): void {  
 this.bookedActive = false;  
 this.reservedActive = false;  
 this.waitingActive = false;  
 this.availableActive = false;  
 this.leadtimeActive = false;  
 }

# **Clear selection**

clearSelection(): void {  
 this.selectedRangeDates = undefined;  
 this.fromDate = undefined;  
 this.tillDate = undefined;  
 this.firstSelectedWeek = null;  
 this.lastSelectedWeek = null;  
  
 this.errorDate = false;  
 this.reservedDate = false;  
 this.waitingListDate = false;  
 this.invalidDate = false;  
  
 this.resetHostClasses();  
 }

* Resets everything to the default, unselected state.

## **How the logic flows (end-to-end)**

1. **Init** (ngOnInit)
   1. Sets responsive options.
   2. Builds demo data: 3 weekly blocks (booked/reserved/waiting).
2. **View ready** (ngAfterViewInit)
   1. Starts a MutationObserver on the host element.
   2. As the PrimeNG calendar renders week-number cells (.p-datepicker-weeknumber), handleAddedNodes runs and:
      1. Lightens styling for past weeks.
      2. Attaches click, mouseover, and mouseout listeners.
3. **User hovers a week**
   1. If week is past or it’s the current week but not Monday yet ⇒ no hover effect.
   2. Otherwise, checks that whole week’s status and applies a hover class (booked/reserved/waiting/neutral).
4. **User clicks a week**
   1. If it’s a past week or current week before Monday ⇒ do nothing.
   2. First click ⇒ select that full Monday–Sunday, validate it (compute booked/reserved/waiting state), and set host classes.
   3. Second click ⇒ extend to a multi-week range (first Monday to second Sunday), or if the second is earlier, treat it as a new single-week selection. Validate and set host classes.
5. **User interacts with the date range input directly**
   1. onChangeSelectedRangeDates enforces Monday–Sunday snapping and triggers validation.
6. **Template rendering**
   1. For each date cell, getDateClass and getDateStyle determines the badge/styling:
      1. Inside the selected range, inline styles are suppressed so **host classes** can style the block uniformly.
      2. Outside the selected range, inline styles provide quick cues (red strike for booked, etc.).
7. **Clear**
   1. clearSelection() resets everything.

## **Why this design?**

* **Weeks as atomic units**: Business logic cares about *weeks*. You snap *all* interactions to Monday–Sunday to avoid partial-week ambiguity and make validation simpler and predictable.
* **DOM observer approach**: PrimeNG generates its DOM after Angular change detection; the MutationObserver lets you attach behavior to elements the library creates (week number cells) without forking the library.
* **HostBinding flags**: Styling the **selected block** (not each individual cell) is easier by toggling classes at the host level. This cleanly separates the “selected block’s theme” from the per-day default styles.

## **Potential pitfalls & improvements (quick wins)**

1. **Observer & listener cleanup**
   1. Store the MutationObserver (this.observer = new MutationObserver(...)) and call this.observer.disconnect() in ngOnDestroy.
   2. Capture unlisten callbacks from renderer.listen (push them to this.unlistenFns[]) and call each in ngOnDestroy.
2. **Consistency of ISO week APIs**
   1. In handleWeekClick, use moment().isoWeek(weekNumber).isoWeekday(1) instead of .day("Monday").week(weekNumber) to avoid mixing locale-week and ISO-week concepts.
3. **Month indexing**
   1. Ensure the template date.month is **0-based** when building new Date(y, m, d). If it’s 1-based, subtract 1. Also, be consistent when logging monthOfSelectedWeek.
4. **Priority order uniformity**
   1. getDateClass uses booked > reserved > waiting; updateDateStatus uses booked > waiting > reserved. Make them the same so the UI is predictable.
5. **Performance**
   1. some(isSameDay) inside loops is fine for demo arrays but if these arrays get large, consider:
      1. Precomputing a Set<string> of yyyy-mm-dd for booked/reserved/waiting to get O(1) lookups.
      2. Or storing ranges by week number instead of individual dates.
6. **Accessibility**
   1. The week number cells get click/hover behavior; ensure this is keyboard accessible too (e.g., keydown handlers, tabindex=0, ARIA roles if needed).
7. **Type safety**
   1. Avoid any for events where you can define a known shape, and consider creating a small interface for the date object { day: number; month: number; year: number } to reuse across methods.
8. **Remove unused imports/flags**
   1. ViewEncapsulation is unused.
   2. invalidDate is never set—either use it or remove it.

## **How you’d hook it up in the template (at a glance)**

You didn’t include the HTML, but typically with PrimeNG Calendar you’d:

* Enable **week numbers** ([showWeek]="true").
* Provide **range selection** (selectionMode="range").
* Use **day templates** to call getDateClass() and getDateStyle() on each cell.
* Bind [(ngModel)]="selectedRangeDates" and (onSelect)="onChangeSelectedRangeDates()" or (onInput)="onChangeSelectedRangeDates()".
* Bind (onMonthChange)="onMonthChange($event)".

If you want, I can refactor this to:

* add proper cleanup for the observer/listeners,
* align ISO week logic everywhere,
* and convert the date arrays to faster lookup sets—just say the word and I’ll drop a cleaned-up version.