Daily Event Scheduler: Full Project Breakdown

## Table of Contents

1. Project Overview

2. Backend Implementation

1. Design Choices

2. Event Struct

3. Scheduler Logic

4. Concurrency Management

5. Add Event and Overlap Check

3. Frontend Implementation

1. Angular Setup

2. Event Interface

3. Scheduler Service

4. Visual Timeline

4. Development Thought Process

1. Backend Thought Process

2. Frontend Thought Process

5. Results and Screenshots

# Project Overview

The Daily Event Scheduler project is designed to help users manage their daily schedules efficiently. Its goal is to prevent scheduling conflicts by ensuring no overlapping events. This documentation provides a detailed breakdown of the backend and frontend implementations, outlining the design choices, code structure, and overall development thought process that guided the creation of this application.

# Backend Implementation

## 1. Design Choices

We opted to use Go as the backend programming language for several reasons:

* Go’s minimalistic syntax allows developers to focus on building performant applications.
* Its native support for concurrency makes it ideal for handling multiple simultaneous user requests, especially when working with event scheduling where multiple users might interact with the server concurrently.

## 2. Event Struct

type Event struct {  
 StartTime int `json:"start\_time"`  
 EndTime int `json:"end\_time"`  
}

Reasoning: This structure uses integers for hours (0-23) to simplify time-based calculations and minimize complexity when checking for overlaps. By restricting the time to whole hours, the design prevents ambiguity and keeps the focus on daily scheduling without the need for precise minutes or seconds.

## 3. Scheduler Logic

Concurrency Management: A `sync.Mutex` is used to ensure that the list of events can be safely modified by multiple requests at once.

type Scheduler struct {  
 events [ ] Event  
 mutex sync.Mutex  
}

Purpose: The scheduler manages a list of events and ensures thread-safe operations. This was essential due to the nature of web applications where multiple users might interact with the server concurrently.

## 4. Add Event and Overlap Check

for \_, e := range s.events {  
 if event.StartTime < e.EndTime && e.StartTime < event.EndTime {  
 return false  
 }  
}

Purpose: The function iterates through existing events and ensures no time overlap exists. If an overlap is detected, the new event is rejected. The design prioritizes efficiency and simplicity, using a linear search approach that works well for the expected scale of daily events.

# Frontend Implementation

## 1. Angular Setup

Framework: Angular was selected because of its component-based architecture, and TypeScript's strong typing ensures consistency.

## 2. Event Interface

export interface Event {  
 start\_time: number;  
 end\_time: number;  
}

Reasoning: This interface mirrors the backend structure. The consistency between backend and frontend ensures that data flows smoothly and eliminates errors during API communication. Using TypeScript also adds type safety, making the development process more reliable.

## 3. Scheduler Service

@Injectable({  
 providedIn: 'root'  
})  
export class SchedulerService {  
 private apiUrl = 'http://localhost:8080/events';  
 constructor(private http: HttpClient) {}  
}

Purpose: This service encapsulates all HTTP requests to interact with the backend, allowing the frontend to add or fetch events. The separation of concerns (API logic isolated in a service) adheres to best practices in Angular, making the codebase clean and maintainable.

## 4. Visual Timeline

<div class="timeline">  
 <div class="event" \*ngFor="let event of events"  
 [style.top.%]="event.start\_time \* 100 / 24"  
 [style.height.%]="(event.end\_time - event.start\_time) \* 100 / 24"></div>  
</div>

Purpose: The timeline is rendered dynamically based on the event’s start and end times. This approach uses percentage-based styling to create a flexible, responsive UI that adapts to various screen sizes while providing a clear and intuitive representation of daily events.

# Thought Process for Choosing Code Type and Functionality

## Backend Thought Process:

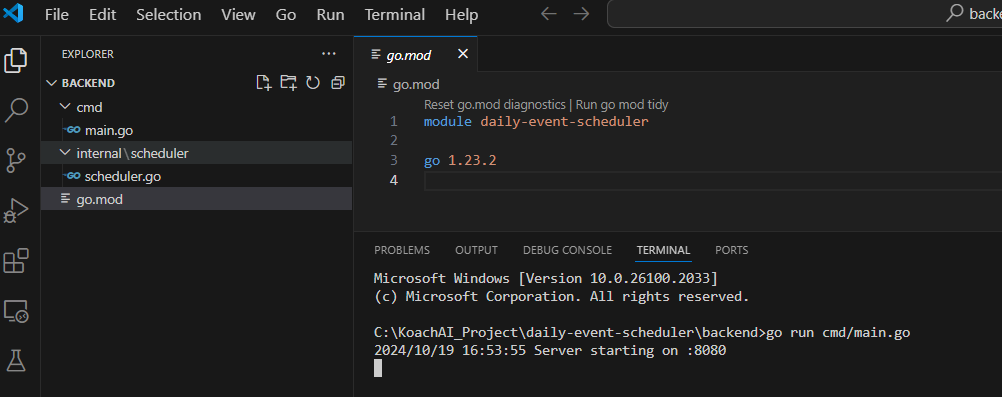
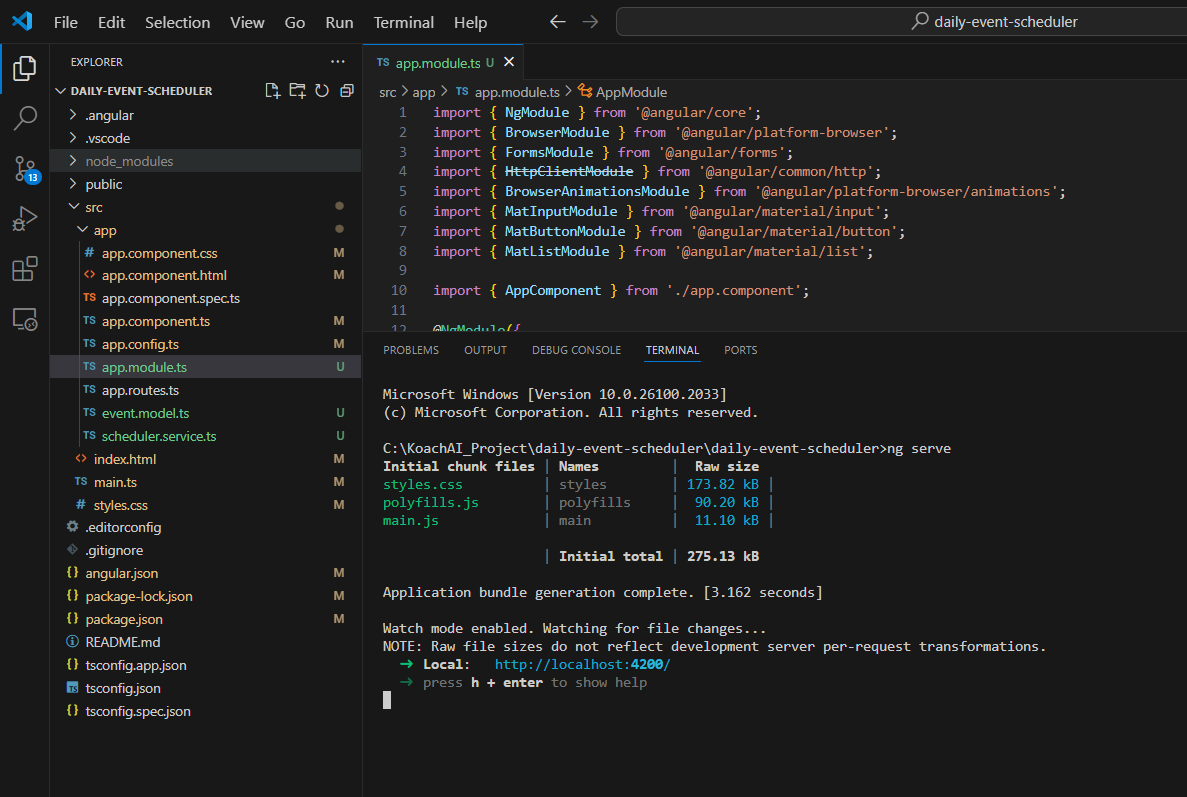
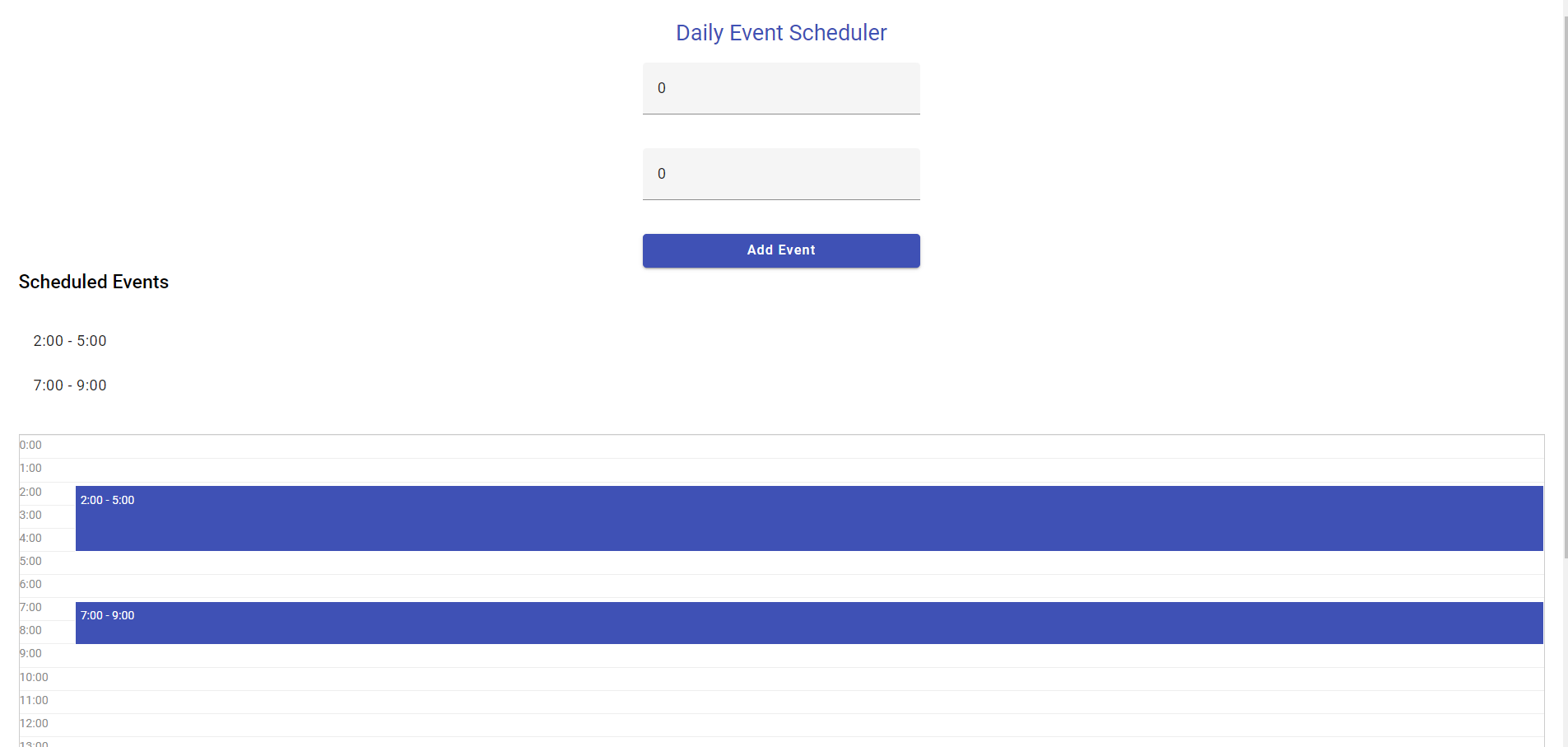
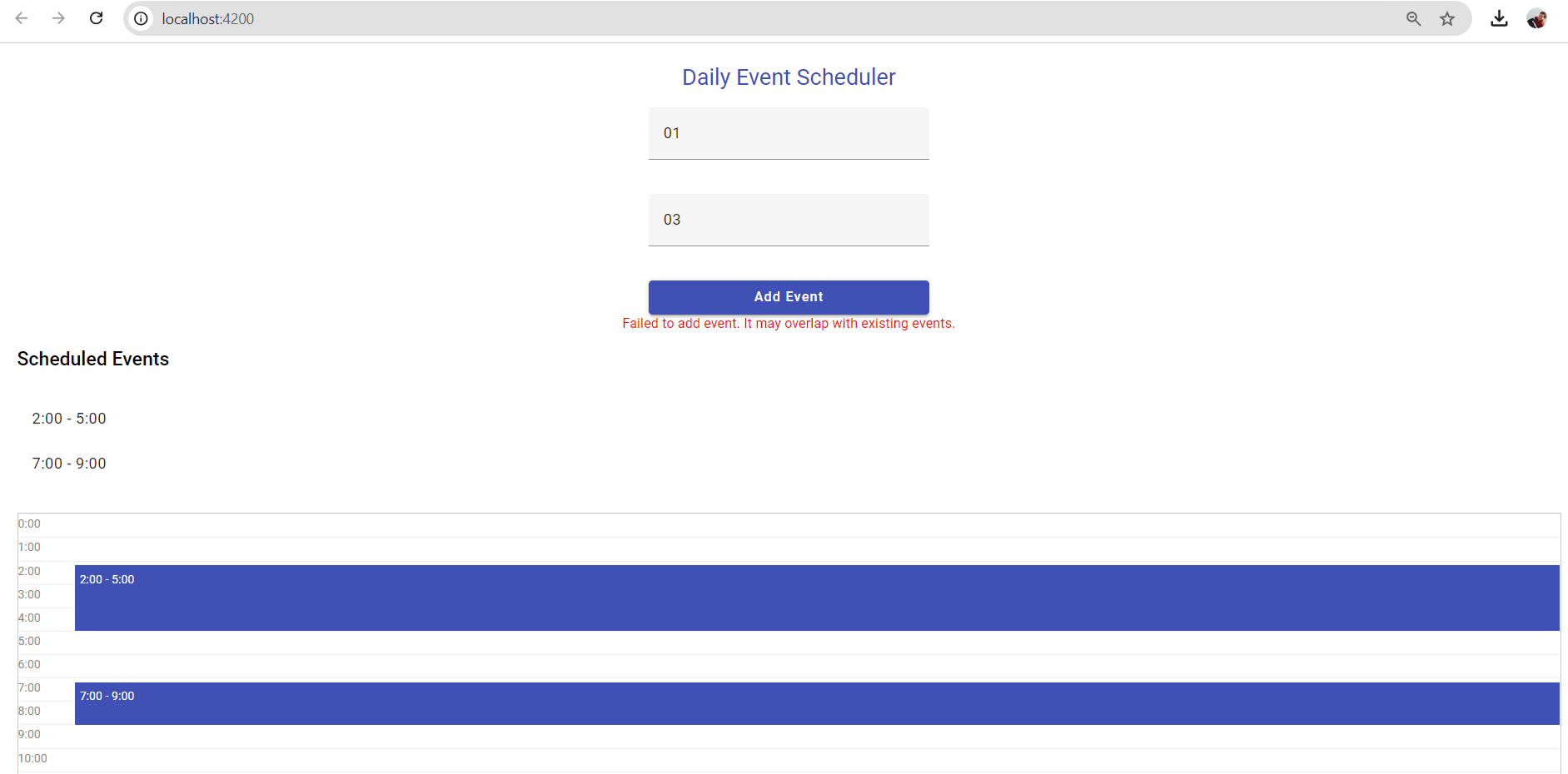
1. Language Choice (Go): Go was chosen due to its minimalistic syntax, which allows developers to focus on performance and efficiency. Its built-in support for concurrency made it ideal for handling multiple user requests concurrently, ensuring data integrity with the use of `sync.Mutex`.  
  
2. Scheduling Logic: The backend needed to manage daily events without overlapping time slots. A straightforward struct (`Event`) was used to store time information, keeping the system lean. By focusing on whole-hour events, we reduced complexity while meeting the needs of a typical scheduler.  
  
The key functionality (adding events) was designed to reject any overlap, ensuring that once an event was scheduled, no conflicting times could be added. The choice of code reflected a balance between simplicity and accuracy.

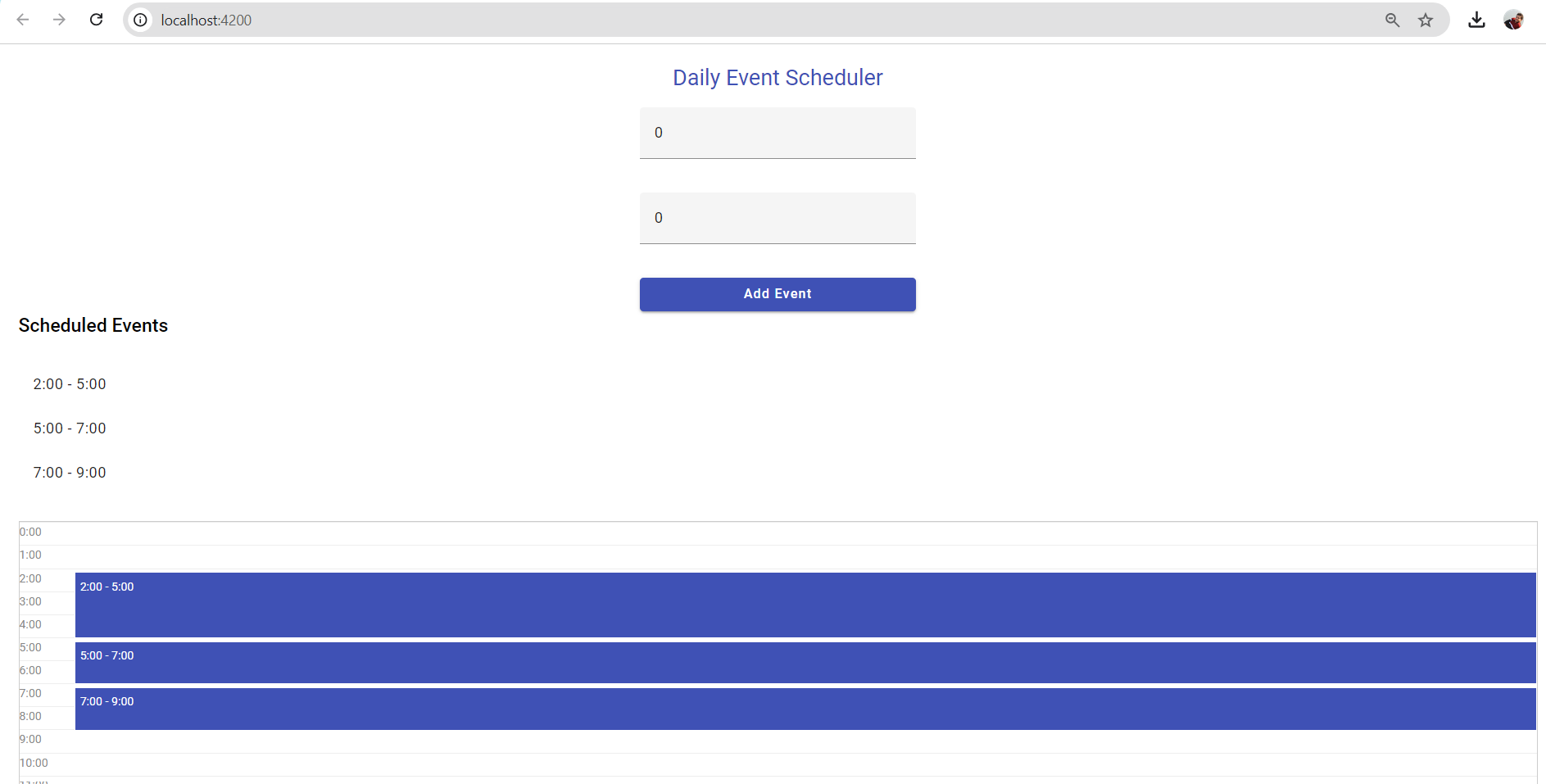
## Frontend Thought Process:

1. Framework Choice (Angular): Angular’s structure helps in breaking down the application into manageable components. The reason for selecting Angular was its robust tools for building dynamic, single-page applications, alongside features like two-way data binding, which simplifies handling user inputs.  
  
2. Data Handling: Using TypeScript provides strict type-checking, ensuring consistent data structures between frontend and backend. The service pattern in Angular isolates API logic from UI components, keeping the code modular and easier to maintain.  
  
3. UI (Timeline View): The visual timeline is key to making the application user-friendly. By using relative CSS positioning with dynamic percentages, the timeline visually represents the start and end of each event, making it easy for users to grasp their schedule at a glance. The visual clarity and flexibility of this approach enhance user experience while maintaining simplicity in code implementation.

# Results

In this section, I have added screenshots of the results for each case.

Part 1: Starting the backend server  
  
  
  
Part 2:Starting the frontend server  
  
  
  
  
Part 3: Opening of http://localhost:4200/  
  
Case 1: events (2, 5) and (7, 9) are already scheduled:  
  
  
  
Case 2: Trying to add (1, 3) should fail (overlaps with 2-5)  
  


Case 3: Trying to add (5, 7) should succeed (fits between existing events)  
  
  
  
  
  
Case 4: Trying to add (8, 10) should fail (overlaps with 7-9)  
  
