INFOTRACK TECH CHALLENGE

Background

implement a booking API that will accept a booking time and respond indicating

whether the reservation was successful or not.

Requirements:

- Assume that all bookings are for the same day (do not worry about handling dates)
- InfoTrack's hours of business are 9am-5pm, all bookings must complete by 5pm (latest booking
- is 4:00pm)
- Bookings are for 1 hour (booking at 9:00am means the spot is held from 9:00am to 9:59am)
- InfoTrack accepts up to 4 simultaneous settlements
- API needs to accept POST requests of the following format:
  {
  "bookingTime": "09:30",
  "name":"John Smith"
  }
- Successful bookings should respond with an OK status and a booking Id in GUID form
  {
  "bookingId": "d90f8c55-90a5-4537-a99d-c68242a6012b"
  }
- Requests for out of hours times should return a Bad Request status
- Requests with invalid data should return a Bad Request status
- Requests when all settlements at a booking time are reserved should return a Conflict status
- The name property should be a non-empty string
- The bookingTime property should be a 24-hour time (00:00 - 23:59)
- Bookings can be stored in-memory, it is fine for them to be forgotten when the application is
- Restarted

Further assumption:

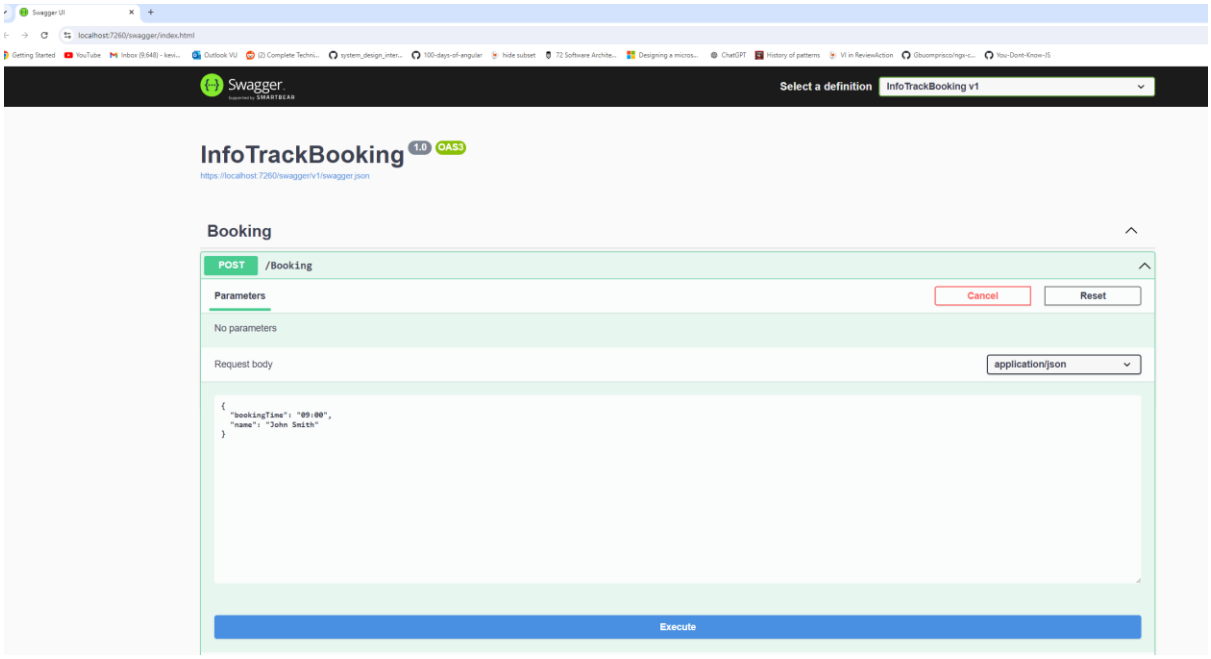
- Booking time must be of minute 00 or 30. Doesn't make sense when booking is made at 13:21.

How to test

Configure json payload in Swagger UI https://localhost:7260/Booking

Or use curl

curl -X POST -H "Content-Type: application/json" -d "{ \"bookingTime\": \"09:00\", \"name\": \"John Smith\" }" https://localhost:7260/Booking



Architecture

Backend: C# .NET v8.0

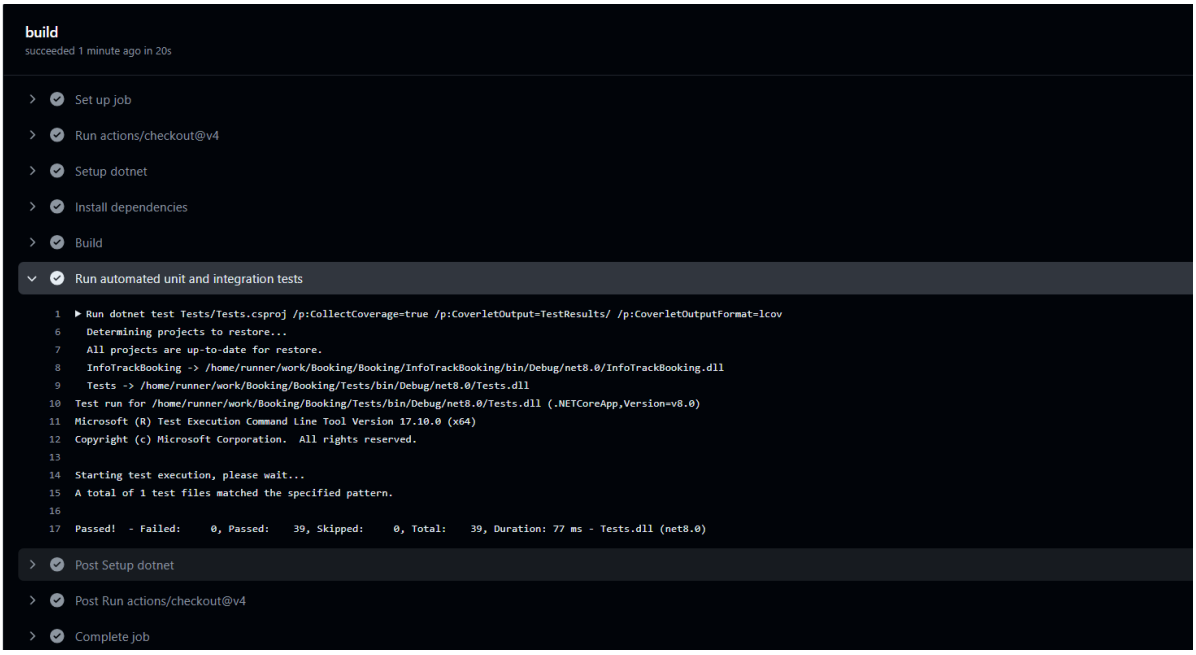
Unit test: XUnit

Database: EF In-Memory

Source control: GitHub

https://github.com/kevinnguyen2208/Booking

CI/CD workflow is configured to test build and run unit tests



Backend

Model

Model represents Data Structure.

BookingRequest represents the json payload coming from POST request.

```
public class BookingRequest
{
    1 reference
    public string BookingTime { get; set; }
    1 reference
    public string Name { get; set; }
}
```

BookingDetails represents what will be saved into the database. i.e. BookingId, Name, StartTime, EndTime.

```
public class BookingDetails
{
    5 references | 3/3 passing
    public Guid Id { get; set; }
    4 references | 3/3 passing
    public string Name { get; set; }
    4 references | 3/3 passing
    public string StartTime { get; set; }
    3 references | 3/3 passing
    public string EndTime { get; set; }
}
```

Controller

Controller handles POST request from the api and gets the response from corresponding service.

Validation returns BadRequest/Conflict/OK based on conditions set in service layer.

```
[HttpPost]
0 references
public async Task<IActionResult> ExecuteBooking([FromBody] BookingRequest request)
{
    ServiceResult<Guid> bookingValidation = await _bookingService.ExecuteBooking(request.BookingTime, request.Name);
    switch (bookingValidation.Validation)
    {
        case ValidationTypes.InvalidParameters:
        case ValidationTypes.InvalidTime:
            return BadRequest(bookingValidation.Message);
        case ValidationTypes.ReservedTime:
            return Conflict(bookingValidation.Message);
        case ValidationTypes.None:
        default:
            return Ok(bookingValidation.Value);
    }
}
```

Service

Service level performs any logic that we impose to reach the tasks' goals.

The logics often correlate to business requirements or any additional data handling that the system uses. In this case, the service class is used to perform data validation and data saving.

```
/// <summary>
/// validate booking parameters before saving
/// </summary>
11 references | 26/26 passing
public async Task<ServiceResult<Guid>> ExecuteBooking(string bookingTime, string name)
{
    //validate booking time or name is not null or empty
    if (string.IsNullOrEmpty(bookingTime) || string.IsNullOrEmpty(name))
    {
        return ServiceResult<Guid>.CreateErrorMessage("Booking time and/or name must not be empty or null.");
    }

    //validate booking time to be valid hh:mm format
    bool isValidTimeFormat = TimeHelper.CheckTimeFormat(bookingTime);
    bool isValidTime = TimeSpan.TryParse(bookingTime, out var startTime);
    if (!isValidTimeFormat || !isValidTime)
    {
        return ServiceResult<Guid>.CreateErrorMessage("Booking time must be in hh:mm format.");
    }

    //validate booking hours
    if (!ValidateBookingTime(startTime))
    {
        return ServiceResult<Guid>.CreateErrorMessage("Booking time must be between 09:00 and 16:00, or the minutes must be either 00 or 30.", ValidationTypes.InvalidTime);
    }

    //validate booking time reservation
    if (!await ValidateReservation(bookingTime))
    {
        return ServiceResult<Guid>.CreateErrorMessage("Booking time is fully booked.", ValidationTypes.ReservedTime);
    }

    //save booking when all validations have been verified
    Guid id = await _bookingRepository.SaveBooking(bookingTime, TimeHelper.CreateEndTime(startTime), name);
    return new ServiceResult<Guid>(id);
}
```

Repository

Repository level has access to the database where it retrieves the data from in-memory. This level retrieves data and doesn't have much logic, beside data collecting or sorting.

```
3 references
public async Task<List<BookingDetails>> GetExistingBookingsByStartTime(string startTime)
{
    List<BookingDetails> bookings = await _context.Bookings.ToListAsync();
    return bookings.Where(f => f.StartTime == startTime).ToList();
}

3 references
public async Task<Guid> SaveBooking(string startTime, string endTime, string name)
{
    BookingDetails booking = new BookingDetails()
    {
        BookingId = Guid.NewGuid(),
        Name = name,
        StartTime = startTime,
        EndTime = endTime
    };

    await _context.Bookings.AddAsync(booking);
    await _context.SaveChangesAsync();
    return booking.BookingId;
}
```

Interfaces

To ensure system follows OOP principles (i.e. abstraction and polymorphism) and help with maintainability, interfaces are added to be used as dependency injections instead of accessing the service/repository directly.

```
6 references
public interface IBookingRepository
{
    3 references
    Task<List<BookingDetails>> GetExistingBookingsByStartTime(string startTime);
    3 references
    Task<Guid> SaveBooking(string startTime, string endTime, string name);
}
```

```
public interface IBookingService
{

    Task<ServiceResult<Guid>> ExecuteBooking(string bookingTime, string name);
}
```

Helper class

This TimeHelper class is designed to handles any Time-related logic. In real life projects, helper class is used to determine a fixed logic or logic related to a specific data structure to be used widely across systems

```
4 references
public static class TimeHelper
{
    /// <summary>
    /// Validate booking time to be valid hh:mm format
    /// </summary>
    2 references | 9/9 passing
    public static bool CheckTimeFormat(string time)
    {
        string pattern = @"^([0-2][0-3]|[0-1][0-9]):[0-5][0-9]+$";
        bool isValidTimeFormat = Regex.IsMatch(time, pattern);
        return isValidTimeFormat;
    }

    /// <summary>
    /// Add 59 minutes to start of booking time
    /// </summary>
    2 references | 4/4 passing
    public static string CreateEndTime(TimeSpan start)
    {
        TimeSpan end = start.Add(new TimeSpan(0, 59, 0));
        return end.ToString(@"hh\:mm");
    }
}
```

Delegate class

This ServiceResult is a delegate class which accepts a generic type T. In this project, T is used as the Guid bookingId. Using a generic type allows reusability of a logic or handler across system with any data type e.g.

```
public class ServiceResult<T>
{
    2 references
    public ServiceResult(T value)
    {
        Value = value;
        Validation = ValidationTypes.None;
        Message = null;
    }

    3 references | 3/3 passing
    public T Value { get; set; }
    14 references | 23/23 passing
    public ValidationTypes Validation { get; set; }
    4 references
    public string Message { get; set; }

    4 references
    public static ServiceResult<T> CreateErrorMessage(string errorMessage, ValidationTypes validation = ValidationTypes.InvalidParameters)
    {
        var result = new ServiceResult<T>(default(T))
        {
            Validation = validation,
            Message = errorMessage
        };

        return result;
    }
}
```

Unit Tests

Unit testing uses XUnit to test the service class and helper class. Mocking is also done for Repository since Service injects Repository.