Creating a cross-platform web application for the ACME Application using Domain-Driven Design (DDD) principles with authentication and authorization in .NET Core involves several steps. Below is a simplified example that demonstrates the structure of the application. Note that this example focuses on the structure and key components and might need adjustments based on your specific requirements.

### Step 1: Set Up a .NET Core Web Application

Create a new .NET Core web application:

```bash

dotnet new web -n AcmeWebApp

cd AcmeWebApp

```

### Step 2: Define Domain Entities and Aggregates

Define your domain entities and aggregates. For example:

```csharp

// Domain/Entities/User.cs

using System;

namespace AcmeWebApp.Domain.Entities

{

public class User

{

public Guid Id { get; private set; }

public string Username { get; private set; }

public string Password { get; private set; }

public string Role { get; private set; }

public User(string username, string password, string role)

{

Id = Guid.NewGuid();

Username = username;

Password = password;

Role = role;

}

}

}

```

### Step 3: Create Repositories

Create repositories to manage aggregates:

```csharp

// Infrastructure/Repositories/UserRepository.cs

using System;

using System.Collections.Generic;

using System.Linq;

using AcmeWebApp.Domain.Entities;

namespace AcmeWebApp.Infrastructure.Repositories

{

public class UserRepository

{

private static readonly List<User> \_users = new List<User>();

public User GetById(Guid id)

{

return \_users.FirstOrDefault(u => u.Id == id);

}

public User GetByUsername(string username)

{

return \_users.FirstOrDefault(u => u.Username == username);

}

public void Save(User user)

{

\_users.Add(user);

}

}

}

```

### Step 4: Implement Authentication

Use ASP.NET Core Identity for authentication:

```bash

dotnet add package Microsoft.AspNetCore.Identity

```

Configure Identity in `Startup.cs`:

```csharp

// Startup.cs

using Microsoft.AspNetCore.Identity;

public class Startup

{

// ... other configurations

public void ConfigureServices(IServiceCollection services)

{

// ... other configurations

services.AddIdentity<IdentityUser, IdentityRole>()

.AddEntityFrameworkStores<YourDbContext>()

.AddDefaultTokenProviders();

// ... other configurations

}

// ... other methods

}

```

### Step 5: Implement Authorization

Configure authorization policies in `Startup.cs`:

```csharp

// Startup.cs

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc.Authorization;

public class Startup

{

// ... other configurations

public void ConfigureServices(IServiceCollection services)

{

// ... other configurations

services.AddMvc(config =>

{

var policy = new AuthorizationPolicyBuilder()

.RequireAuthenticatedUser()

.Build();

config.Filters.Add(new AuthorizeFilter(policy));

});

// ... other configurations

services.AddAuthorization(options =>

{

options.AddPolicy("AdminOnly", policy => policy.RequireRole("Admin"));

// Add more policies as needed

});

}

// ... other methods

}

```

### Step 6: Create Controllers

Create controllers in the presentation layer:

```csharp

// Controllers/UserController.cs

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Identity;

using Microsoft.AspNetCore.Mvc;

using AcmeWebApp.Domain.Entities;

using AcmeWebApp.Infrastructure.Repositories;

[ApiController]

[Route("api/users")]

public class UserController : ControllerBase

{

private readonly UserManager<IdentityUser> \_userManager;

private readonly UserRepository \_userRepository;

public UserController(UserManager<IdentityUser> userManager, UserRepository userRepository)

{

\_userManager = userManager;

\_userRepository = userRepository;

}

[HttpPost]

[AllowAnonymous]

public IActionResult RegisterUser([FromBody] RegisterUserRequest request)

{

var user = new IdentityUser { UserName = request.Username, Email = request.Email };

var result = \_userManager.CreateAsync(user, request.Password).Result;

if (result.Succeeded)

{

\_userRepository.Save(new User(request.Username, request.Password, "User"));

return Ok(new { Message = "User registered successfully." });

}

return BadRequest(new { Message = "User registration failed." });

}

[HttpGet("{id}")]

[Authorize("AdminOnly")]

public IActionResult GetUser(Guid id)

{

var user = \_userRepository.GetById(id);

if (user == null)

{

return NotFound(new { Message = "User not found." });

}

return Ok(new { user.Id, user.Username, user.Role });

}

}

public class RegisterUserRequest

{

public string Username { get; set; }

public string Password { get; set; }

public string Email { get; set; }

}

```

### Step 7: Deploy Cross-Platform

.NET Core applications are inherently cross-platform. You can deploy the application to various platforms using the

following steps:

1. \*\*Publish the Application:\*\*

```bash

dotnet publish -c Release

```

2. \*\*Copy the Published Files:\*\*

Copy the contents of the `bin/Release/netcoreapp3.1/publish` directory to your deployment location.

3. \*\*Run the Application:\*\*

Execute the application on the target platform:

```bash

dotnet AcmeWebApp.dll

```

### Step 8: Test the Deployment

Test the deployed application on various platforms to ensure cross-platform compatibility.

Keep in mind that this is a basic example, and in a real-world scenario, you would need to handle security aspects, validation, error handling, and other considerations for a production application.

Creating a fully functional web application involves multiple steps, and providing a complete implementation within this format is not feasible. However, I can guide you through the high-level steps to create a .NET Core web application with the specified user flow.

### Step 1: Set Up a .NET Core Web Application

1. Create a new .NET Core web application:

```bash

dotnet new web -n AcmeRoomReservation

cd AcmeRoomReservation

```

2. Open the project in your preferred code editor (e.g., Visual Studio Code, Visual Studio).

### Step 2: Implement Domain Entities and Repositories

1. Define domain entities in the `Domain` folder (e.g., `User.cs`, `Room.cs`).

2. Implement repositories in the `Infrastructure` folder (e.g., `UserRepository.cs`, `RoomRepository.cs`).

### Step 3: Configure Authentication and Authorization

1. Install the necessary packages:

```bash

dotnet add package Microsoft.AspNetCore.Authentication

dotnet add package Microsoft.AspNetCore.Authorization

```2. Configure authentication and authorization in `Startup.cs`.

### Step 4: Implement Reservation and Recommendation Services

1. Implement reservation logic in the `ReservationService.cs`.

2. Create a recommendation service (e.g., `RecommendationService.cs`) for predictive and efficiency recommendations.

### Step 5: Create Controllers

1. Create controllers in the `Controllers` folder (e.g., `UserController.cs`, `RoomController.cs`).

2. Implement actions for user registration, room reservation, and fetching recommendations.

### Step 6: Configure Dependency Injection

1. Register services and repositories in the `Startup.cs` file.

### Step 7: Create Views (HTML/CSS) and UI Logic

1. Create views in the `Views` folder for user authentication, room reservation, and recommendations.

2. Implement UI logic using Razor syntax.

### Step 8: Configure Routing

1. Set up routing in `Startup.cs` to map URLs to controllers and actions.

### Step 9: Implement Notification Service

1. Integrate a notification service for sending emails or other notifications.

2. Handle notifications in the reservation process.

### Step 10: Testing

1. Write unit tests for services and repositories.

2. Test the user flow through the application.

### Step 11: Deployment

1. Publish the application:

```bash dotnet publish -c Release

2. Deploy the published files to your hosting environment.

### Considerations:

1. Use Entity Framework Core for data access.

2. Implement validation to ensure data integrity.

3. Enhance security by implementing HTTPS, data encryption, and secure coding practices.

4. Implement error handling and logging for better debugging and maintenance.

5. Consider using a front-end framework or library (e.g., Bootstrap, React, Angular) for a more interactive UI.

This is a simplified outline, and each step requires detailed implementation based on your specific requirements and design preferences. The choice of technologies and tools may vary based on your project constraints and goals.

Certainly! Let's start by implementing domain entities and repositories for the Room Reservation system in .NET Core. For simplicity, I'll provide examples for `User` and `Room` entities and their respective repositories.

### Step 1: Define Domain Entities

Create domain entities representing users and rooms in the `Domain` folder.

1. \*\*User Entity (`User.cs`):\*\*

```csharp

// Domain/Entities/User.cs

using System;

namespace AcmeRoomReservation.Domain.Entities

{

public class User

{

public Guid Id { get; set; }

public string Username { get; set; }

public string Password { get; set; }

public string Role { get; set; }

}

}

```

2. \*\*Room Entity (`Room.cs`):\*\*

```csharp

// Domain/Entities/Room.cs

using System;

namespace AcmeRoomReservation.Domain.Entities

{

public class Room

{

public Guid Id { get; set; }

public string Name { get; set; }

public int Capacity { get; set; }

public string Location { get; set; }

public bool HasChairs { get; set; }

}

}

```

### Step 2: Create Repositories

Create repositories for managing user and room entities in the `Infrastructure/Repositories` folder.

1. \*\*User Repository (`UserRepository.cs`):\*\*

```csharp

// Infrastructure/Repositories/UserRepository.cs

using System;

using System.Collections.Generic;

using System.Linq;

using AcmeRoomReservation.Domain.Entities;

namespace AcmeRoomReservation.Infrastructure.Repositories

{

public class UserRepository

{

private static readonly List<User> \_users = new List<User>();

public User GetById(Guid id)

{

return \_users.FirstOrDefault(u => u.Id == id);

}

public User GetByUsername(string username)

{

return \_users.FirstOrDefault(u => u.Username == username);

}

public void Save(User user)

{

user.Id = Guid.NewGuid();

\_users.Add(user);

}

}

}

```

2. \*\*Room Repository (`RoomRepository.cs`):\*\*

```csharp

// Infrastructure/Repositories/RoomRepository.cs

using System;

using System.Collections.Generic;

using System.Linq;

using AcmeRoomReservation.Domain.Entities;

namespace AcmeRoomReservation.Infrastructure.Repositories

{

public class RoomRepository

{

private static readonly List<Room> \_rooms = new List<Room>();

public Room GetById(Guid id)

{

return \_rooms.FirstOrDefault(r => r.Id == id);

}

public IEnumerable<Room> GetAvailableRooms(string location)

{

// Implement logic to fetch available rooms based on location and other criteria

return \_rooms.Where(r => r.Location.Equals(location, StringComparison.OrdinalIgnoreCase));

}

public void Save(Room room)

{

room.Id = Guid.NewGuid();

\_rooms.Add(room);

}

}

}

```

These examples provide a basic setup for domain entities (`User` and `Room`) and their corresponding repositories (`UserRepository` and `RoomRepository`). In a real-world scenario, you would likely use a database (e.g., Entity Framework Core) for data persistence and implement more sophisticated logic within the repositories.

To configure authentication and authorization in a .NET Core web application, you'll need to use ASP.NET Core Identity for authentication and configure authorization policies. Below are the steps to set up authentication and authorization in the `Startup.cs` file.

### Step 1: Install Required Packages

Open a terminal and run the following commands to install the required packages:

```bash

dotnet add package Microsoft.AspNetCore.Identity

dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore

```

### Step 2: Configure Identity and Authorization in `Startup.cs`

Update the `Startup.cs` file with the following configurations:

```csharp

// Startup.cs

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Hosting;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Hosting;

using Microsoft.AspNetCore.Identity;

using AcmeRoomReservation.Domain.Entities;

using AcmeRoomReservation.Infrastructure.Repositories;

namespace AcmeRoomReservation

{

public class Startup

{

// ... other configurations

public void ConfigureServices(IServiceCollection services)

{

// ... other configurations

// Configure Identity

services.AddIdentity<User, IdentityRole>()

.AddDefaultTokenProviders()

.AddEntityFrameworkStores<YourDbContext>(); // Make sure to replace YourDbContext with your actual database context

// Configure Authorization

services.AddAuthorization(options =>

{

options.AddPolicy("AdminOnly", policy => policy.RequireRole("Admin"));

// Add more policies as needed

});

// ... other configurations

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

// ... other configurations

// Use Authentication and Authorization

app.UseAuthentication();

app.UseAuthorization();

// ... other configurations

}

}

}

```

### Step 3: Update User Entity for Identity

Update the `User` entity in `Domain/Entities/User.cs` to inherit from `IdentityUser`:

```csharp

// Domain/Entities/User.cs

using Microsoft.AspNetCore.Identity;

using System;

namespace AcmeRoomReservation.Domain.Entities

{

public class User : IdentityUser

{

// Any additional properties or methods specific to your application

}

}

```

### Step 4: Create a Database Context

If you don't have a database context yet, create one to work with Entity Framework Core:

```csharp

// Infrastructure/Data/YourDbContext.cs

using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore;

using AcmeRoomReservation.Domain.Entities;

namespace AcmeRoomReservation.Infrastructure.Data

{

public class YourDbContext : IdentityDbContext<User>

{

public YourDbContext(DbContextOptions<YourDbContext> options)

: base(options)

{

}

// DbSet properties for other entities (e.g., Room) go here

protected override void OnModelCreating(ModelBuilder builder)

{

base.OnModelCreating(builder);

// Additional model configurations (if any) go here

}

}

}

```

### Step 5: Configure Connection String

Ensure you have a connection string in your `appsettings.json` or `appsettings.Development.json` file:

```json

{

"ConnectionStrings": {

"DefaultConnection": "YourConnectionString"

},

// ... other configurations

}

```

### Step 6: Apply Database Migrations

Run the following commands to apply migrations and update the database:

```bash

dotnet ef migrations add InitialCreate

dotnet ef database update

```

Make sure to replace `InitialCreate` with a meaningful migration name.

### Notes:

- Replace `YourDbContext` with the actual name of your database context.

- Adjust the namespace and folder structure based on your project organization.

This configuration sets up authentication using ASP.NET Core Identity and defines an "AdminOnly" authorization policy. Ensure you customize this based on your specific needs and integrate it with your user and role management logic.

Certainly! Let's implement the Reservation and Recommendation services for the Room Reservation system in .NET Core. Below are the steps for creating these services.

### Step 1: Create a Reservation Service

1. \*\*Define a Reservation model:\*\*

Create a `Reservation` class in the `Domain/Entities` folder:

```csharp

// Domain/Entities/Reservation.cs

using System;

namespace AcmeRoomReservation.Domain.Entities

{

public class Reservation

{

public Guid Id { get; set; }

public Guid UserId { get; set; }

public Guid RoomId { get; set; }

public DateTime StartTime { get; set; }

public DateTime EndTime { get; set; }

// Add other relevant properties

}

}

```

2. \*\*Create a Reservation Service:\*\*

Create a `ReservationService` class in the `Services` folder:

```csharp

// Services/ReservationService.cs

using System;

using AcmeRoomReservation.Domain.Entities;

using AcmeRoomReservation.Infrastructure.Repositories;

public class ReservationService

{

private readonly RoomRepository \_roomRepository;

private readonly UserRepository \_userRepository;

public ReservationService(RoomRepository roomRepository, UserRepository userRepository)

{

\_roomRepository = roomRepository;

\_userRepository = userRepository;

}

public bool ReserveRoom(Guid userId, Guid roomId, DateTime startTime, DateTime endTime)

{

// Validate reservation logic (e.g., check for overlapping appointments)

// Implement your logic here

// Save the reservation to the database

var reservation = new Reservation

{

UserId = userId,

RoomId = roomId,

StartTime = startTime,

EndTime = endTime

};

// Add the reservation to the database

// Note: Implement the actual database logic based on your data access strategy

// \_reservationRepository.Save(reservation);

return true; // Successfully reserved the room

}

}

```

### Step 2: Create a Recommendation Service

1. \*\*Create a Recommendation Service:\*\*

Create a `RecommendationService` class in the `Services` folder:

```csharp

// Services/RecommendationService.cs

using System;

using System.Collections.Generic;

using AcmeRoomReservation.Domain.Entities;

using AcmeRoomReservation.Infrastructure.Repositories;

public class RecommendationService

{

private readonly RoomRepository \_roomRepository;

private readonly UserRepository \_userRepository;

public RecommendationService(RoomRepository roomRepository, UserRepository userRepository)

{

\_roomRepository = roomRepository;

\_userRepository = userRepository;

}

public IEnumerable<Room> GetRecommendedRooms(Guid userId, DateTime startTime, DateTime endTime)

{

// Implement your recommendation logic here

// You can consider user preferences, historical data, and meeting type

// For simplicity, let's return all available rooms for now

return \_roomRepository.GetAvailableRooms("Berlin");

}

}

```

### Step 3: Update Dependency Injection in `Startup.cs`

In your `Startup.cs` file, make sure to register these services in the `ConfigureServices` method:

```csharp

// Startup.cs

using AcmeRoomReservation.Infrastructure.Repositories;

using AcmeRoomReservation.Services;

public void ConfigureServices(IServiceCollection services)

{

// ... other configurations

// Register repositories

services.AddScoped<RoomRepository>();

services.AddScoped<UserRepository>();

// Register services

services.AddScoped<ReservationService>();

services.AddScoped<RecommendationService>();

// ... other configurations

}

```

### Notes:

- These services provide a basic structure, and you should customize them based on your specific business logic and data access strategy (e.g., using Entity Framework Core for database operations).

- Ensure proper error handling and validation based on your application's requirements.

- The `RecommendationService` is a placeholder; you can implement more sophisticated logic for generating recommendations based on historical data and user preferences.

Remember to integrate these services into your controllers or other parts of the application where they are needed.

To implement reservation logic and check for overlapping appointments, you need to ensure that a new reservation doesn't conflict with existing reservations for the same room and time frame. Below is an example of how you can enhance the `ReservationService` to include this validation.

```csharp

// Services/ReservationService.cs

using System;

using AcmeRoomReservation.Domain.Entities;

using AcmeRoomReservation.Infrastructure.Repositories;

public class ReservationService

{

private readonly RoomRepository \_roomRepository;

private readonly UserRepository \_userRepository;

public ReservationService(RoomRepository roomRepository, UserRepository userRepository)

{

\_roomRepository = roomRepository;

\_userRepository = userRepository;

}

public bool ReserveRoom(Guid userId, Guid roomId, DateTime startTime, DateTime endTime)

{

// Validate reservation logic (e.g., check for overlapping appointments)

if (!IsRoomAvailable(roomId, startTime, endTime))

{

return false; // Room is not available for the specified time frame

}

// Save the reservation to the database

var reservation = new Reservation

{

UserId = userId,

RoomId = roomId,

StartTime = startTime,

EndTime = endTime

};

// Add the reservation to the database

// Note: Implement the actual database logic based on your data access strategy

// \_reservationRepository.Save(reservation);

return true; // Successfully reserved the room

}

private bool IsRoomAvailable(Guid roomId, DateTime startTime, DateTime endTime)

{

// Check if the room is available for the specified time frame

var existingReservations = \_roomRepository.GetReservationsForRoom(roomId);

foreach (var reservation in existingReservations)

{

if (startTime < reservation.EndTime && endTime > reservation.StartTime)

{

return false; // Overlapping appointment found

}

}

return true; // Room is available

}

}

```

In the updated code:

- The `ReserveRoom` method now calls a new private method, `IsRoomAvailable`, to check for overlapping appointments before proceeding with the reservation.

- The `IsRoomAvailable` method queries existing reservations for the specified room using a hypothetical method `GetReservationsForRoom` (you need to implement this in your `RoomRepository`). It then checks if the proposed reservation time frame overlaps with any existing reservations.

- If an overlapping appointment is found, the method returns `false`, indicating that the room is not available for the specified time frame. Otherwise, it returns `true`, indicating that the room is available, and the reservation can proceed.

This logic ensures that reservations are validated against existing appointments for the same room and time frame, preventing conflicts. Adjust the code as needed based on your specific data access strategy and requirements.

To implement the `GetReservationsForRoom` method in the `RoomRepository`, you'll need to have a way to store and retrieve reservation data. For simplicity, let's assume you're using an in-memory collection to store reservations.

Below is an example of how you can implement the `GetReservationsForRoom` method in the `RoomRepository` class:

```csharp

// Infrastructure/Repositories/RoomRepository.cs

using System;

using System.Collections.Generic;

using System.Linq;

using AcmeRoomReservation.Domain.Entities;

namespace AcmeRoomReservation.Infrastructure.Repositories

{

public class RoomRepository

{

private static readonly List<Room> \_rooms = new List<Room>();

private static readonly List<Reservation> \_reservations = new List<Reservation>();

public Room GetById(Guid id)

{

return \_rooms.FirstOrDefault(r => r.Id == id);

}

public IEnumerable<Room> GetAvailableRooms(string location)

{

// Implement logic to fetch available rooms based on location and other criteria

return \_rooms.Where(r => r.Location.Equals(location, StringComparison.OrdinalIgnoreCase));

}

public void Save(Room room)

{

room.Id = Guid.NewGuid();

\_rooms.Add(room);

}

public IEnumerable<Reservation> GetReservationsForRoom(Guid roomId)

{

return \_reservations.Where(r => r.RoomId == roomId);

}

public void SaveReservation(Reservation reservation)

{

reservation.Id = Guid.NewGuid();

\_reservations.Add(reservation);

}

}

}

```

In this example:

- The `\_reservations` list is used to store reservations for various rooms.

- The `GetReservationsForRoom` method filters the list to retrieve reservations specific to a given room.

Please note that this is a simplified example, and in a real-world scenario, you would likely use a database (such as Entity Framework Core) for data persistence.

Make sure to update the `ReserveRoom` method in the `ReservationService` to use the `SaveReservation` method to store reservations:

```csharp

// Services/ReservationService.cs

public bool ReserveRoom(Guid userId, Guid roomId, DateTime startTime, DateTime endTime)

{

// Validate reservation logic (e.g., check for overlapping appointments)

if (!IsRoomAvailable(roomId, startTime, endTime))

{

return false; // Room is not available for the specified time frame

}

// Save the reservation to the database

var reservation = new Reservation

{

UserId = userId,

RoomId = roomId,

StartTime = startTime,

EndTime = endTime

};

// Add the reservation to the database

\_roomRepository.SaveReservation(reservation);

return true; // Successfully reserved the room

}

```

Adjust the code based on your actual data access strategy and requirements.

To add the reservation to a database, you'll typically use an Object-Relational Mapping (ORM) framework like Entity Framework Core. Below is an example of how you can modify the `RoomRepository` class to use Entity Framework Core for data access.

### Step 1: Install Entity Framework Core

```bash

dotnet add package Microsoft.EntityFrameworkCore

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

```

Replace `Microsoft.EntityFrameworkCore.SqlServer` with the appropriate package for your chosen database provider.

### Step 2: Update the `RoomRepository` for Entity Framework Core

```csharp

// Infrastructure/Repositories/RoomRepository.cs

using System;

using System.Collections.Generic;

using System.Linq;

using AcmeRoomReservation.Domain.Entities;

using Microsoft.EntityFrameworkCore;

namespace AcmeRoomReservation.Infrastructure.Repositories

{

public class RoomRepository

{

private readonly YourDbContext \_dbContext;

public RoomRepository(YourDbContext dbContext)

{

\_dbContext = dbContext;

}

public Room GetById(Guid id)

{

return \_dbContext.Rooms.FirstOrDefault(r => r.Id == id);

}

public IEnumerable<Room> GetAvailableRooms(string location)

{

return \_dbContext.Rooms

.Where(r => r.Location.Equals(location, StringComparison.OrdinalIgnoreCase));

}

public IEnumerable<Reservation> GetReservationsForRoom(Guid roomId)

{

return \_dbContext.Reservations

.Where(r => r.RoomId == roomId)

.ToList();

}

public void Save(Room room)

{

\_dbContext.Rooms.Add(room);

\_dbContext.SaveChanges();

}

public void SaveReservation(Reservation reservation)

{

\_dbContext.Reservations.Add(reservation);

\_dbContext.SaveChanges();

}

}

}

```

### Step 3: Define Entity Classes in Your DbContext

Ensure that you have defined the necessary entity classes in your `YourDbContext` class.

```csharp

// Infrastructure/Data/YourDbContext.cs

using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore;

using AcmeRoomReservation.Domain.Entities;

namespace AcmeRoomReservation.Infrastructure.Data

{

public class YourDbContext : IdentityDbContext<User>

{

public DbSet<Room> Rooms { get; set; }

public DbSet<Reservation> Reservations { get; set; }

public YourDbContext(DbContextOptions<YourDbContext> options)

: base(options)

{

}

// ... other configurations

}

}

```

### Step 4: Apply Migrations and Update Database

Run the following commands in the terminal to apply migrations and update the database:

```bash

dotnet ef migrations add InitialCreate

dotnet ef database update

```

Replace `InitialCreate` with a meaningful migration name.

With these changes, the `Save` and `SaveReservation` methods will add records to the database using Entity Framework Core.

Ensure that you've configured your database connection string in `appsettings.json` or `appsettings.Development.json`:

```json

{

"ConnectionStrings": {

"DefaultConnection": "YourConnectionString"

},

// ... other configurations

}

```

Replace `YourConnectionString` with the actual connection string for your database.

Adjust the code based on your specific needs and data access strategy.

To implement user registration logic using `\_reservationService`, you'll need to define the necessary functionality in the `ReservationService`. Below is an example of how you can enhance the `ReservationService` to include user registration logic.

First, add a new method to the `UserRepository` to handle user registration:

```csharp

// Infrastructure/Repositories/UserRepository.cs

using AcmeRoomReservation.Domain.Entities;

public class UserRepository

{

// ... other methods

public void RegisterUser(string username, string password)

{

var user = new User

{

UserName = username,

// Hash the password using a secure hashing algorithm

// Ensure to follow best practices for password hashing in a real-world scenario

PasswordHash = HashPassword(password)

};

Save(user);

}

private string HashPassword(string password)

{

// Implement password hashing logic using a secure hashing algorithm (e.g., BCrypt)

// Ensure to follow best practices for password hashing in a real-world scenario

// Example using BCrypt:

// return BCrypt.Net.BCrypt.HashPassword(password);

return password; // Replace with actual password hashing logic

}

}

```

Now, update the `ReservationService` to include the user registration logic:

```csharp

// Services/ReservationService.cs

using AcmeRoomReservation.Infrastructure.Repositories;

public class ReservationService

{

private readonly RoomRepository \_roomRepository;

private readonly UserRepository \_userRepository;

public ReservationService(RoomRepository roomRepository, UserRepository userRepository)

{

\_roomRepository = roomRepository;

\_userRepository = userRepository;

}

public bool ReserveRoom(Guid userId, Guid roomId, DateTime startTime, DateTime endTime)

{

// Validate reservation logic (e.g., check for overlapping appointments)

if (!IsRoomAvailable(roomId, startTime, endTime))

{

return false; // Room is not available for the specified time frame

}

// Save the reservation to the database

var reservation = new Reservation

{

UserId = userId,

RoomId = roomId,

StartTime = startTime,

EndTime = endTime

};

// Add the reservation to the database

\_roomRepository.SaveReservation(reservation);

return true; // Successfully reserved the room

}

public void RegisterUser(string username, string password)

{

// Implement user registration logic

// For simplicity, you can directly call the UserRepository method

\_userRepository.RegisterUser(username, password);

}

private bool IsRoomAvailable(Guid roomId, DateTime startTime, DateTime endTime)

{

// Check if the room is available for the specified time frame

var existingReservations = \_roomRepository.GetReservationsForRoom(roomId);

foreach (var reservation in existingReservations)

{

if (startTime < reservation.EndTime && endTime > reservation.StartTime)

{

return false; // Overlapping appointment found

}

}

return true; // Room is available

}

}

```

Now, you can use the `RegisterUser` method from the `ReservationService` in your `UserController`:

```csharp

// Controllers/UserController.cs

using Microsoft.AspNetCore.Mvc;

using AcmeRoomReservation.Services;

[ApiController]

[Route("api/[controller]")]

public class UserController : ControllerBase

{

private readonly ReservationService \_reservationService;

public UserController(ReservationService reservationService)

{

\_reservationService = reservationService;

}

[HttpPost("register")]

public IActionResult RegisterUser([FromBody] RegistrationModel model)

{

// Implement user registration logic using \_reservationService

\_reservationService.RegisterUser(model.Username, model.Password);

return Ok(new { Message = "User registered successfully" });

}

}

public class RegistrationModel

{

public string Username { get; set; }

public string Password { get; set; }

// Add other registration properties as needed

}

```

This example demonstrates how to integrate user registration logic into the existing services. Ensure that you have proper error handling and validation in a production scenario. Also, consider using a more secure password hashing mechanism (e.g., BCrypt) for storing passwords.

To implement password hashing using a secure hashing algorithm like BCrypt in a .NET Core application, you can use the BCrypt.Net library. Below is an example of how you can modify the `UserRepository` to incorporate BCrypt for password hashing.

### Step 1: Install BCrypt.Net Library

Add the BCrypt.Net library to your project:

```bash

dotnet add package BCrypt.Net-Next

```

### Step 2: Update `UserRepository` for Password Hashing

Modify the `UserRepository` class to include BCrypt for password hashing:

```csharp

// Infrastructure/Repositories/UserRepository.cs

using AcmeRoomReservation.Domain.Entities;

using BCrypt.Net;

public class UserRepository

{

// ... other methods

public void RegisterUser(string username, string password)

{

var user = new User

{

UserName = username,

PasswordHash = HashPassword(password)

};

Save(user);

}

private string HashPassword(string password)

{

// Use BCrypt to hash the password

return BCrypt.Net.BCrypt.HashPassword(password, BCrypt.Net.BCrypt.GenerateSalt());

}

// ... other methods

}

```

In this example, the `HashPassword` method generates a salt using `BCrypt.Net.BCrypt.GenerateSalt()` and then hashes the password using `BCrypt.Net.BCrypt.HashPassword(password, salt)`.

### Step 3: Update `User` Entity

Ensure that the `User` entity includes a property for storing the password hash:

```csharp

// Domain/Entities/User.cs

using Microsoft.AspNetCore.Identity;

public class User : IdentityUser

{

// Add other properties as needed

public string PasswordHash { get; set; }

}

```

### Step 4: Update `YourDbContext` to Include the `User` Entity

Make sure that your `YourDbContext` includes the `User` entity:

```csharp

// Infrastructure/Data/YourDbContext.cs

using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore;

using AcmeRoomReservation.Domain.Entities;

public class YourDbContext : IdentityDbContext<User>

{

// ... other configurations

public DbSet<User> Users { get; set; }

// ... other configurations

}

```

### Step 5: Update `Startup.cs` to Configure Identity

Ensure that the `ConfigureServices` method in `Startup.cs` configures Identity with your `User` entity:

```csharp

// Startup.cs

public void ConfigureServices(IServiceCollection services)

{

// ... other configurations

// Configure Identity

services.AddIdentity<User, IdentityRole>()

.AddEntityFrameworkStores<YourDbContext>()

.AddDefaultTokenProviders();

// ... other configurations

}

```

### Notes:

- BCrypt is a secure and widely-used password hashing algorithm that incorporates a salt automatically.

- Ensure that you handle password-related operations securely and follow best practices.

- This example uses BCrypt.Net-Next. Ensure that you use the appropriate version based on the current package versions available.

After making these changes, your application should use BCrypt for hashing user passwords during registration. Adjust the code based on your specific requirements and security policies.

To implement user registration logic in a .NET Core application, you can use ASP.NET Core Identity for user management. Below is a step-by-step guide on how to set up user registration logic using ASP.NET Core Identity.

### Step 1: Install Required Packages

Make sure to install the necessary packages for ASP.NET Core Identity:

```bash

dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

dotnet add package Microsoft.AspNetCore.Identity.UI

```

### Step 2: Configure Your DbContext

Configure your DbContext to use ASP.NET Core Identity. Create or update your `YourDbContext` class:

```csharp

// Infrastructure/Data/YourDbContext.cs

using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore;

using AcmeRoomReservation.Domain.Entities;

public class YourDbContext : IdentityDbContext<User>

{

public YourDbContext(DbContextOptions<YourDbContext> options)

: base(options)

{

}

// ... other configurations

}

```

### Step 3: Configure Identity in `Startup.cs`

Configure Identity services in your `Startup.cs`:

```csharp

// Startup.cs

public void ConfigureServices(IServiceCollection services)

{

// ... other configurations

services.AddDbContext<YourDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

services.AddIdentity<User, IdentityRole>()

.AddEntityFrameworkStores<YourDbContext>()

.AddDefaultTokenProviders();

// ... other configurations

}

```

### Step 4: Create a RegistrationModel

Create a model to represent user registration data:

```csharp

// Models/RegistrationModel.cs

using System.ComponentModel.DataAnnotations;

public class RegistrationModel

{

[Required]

public string Username { get; set; }

[Required]

[DataType(DataType.Password)]

public string Password { get; set; }

[Required]

[DataType(DataType.Password)]

[Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]

public string ConfirmPassword { get; set; }

// Add other registration properties as needed

}

```

### Step 5: Create UserController for Registration

Create a `UserController` to handle user registration:

```csharp

// Controllers/UserController.cs

using Microsoft.AspNetCore.Identity;

using Microsoft.AspNetCore.Mvc;

using System.Threading.Tasks;

using AcmeRoomReservation.Domain.Entities;

[ApiController]

[Route("api/[controller]")]

public class UserController : ControllerBase

{

private readonly UserManager<User> \_userManager;

private readonly SignInManager<User> \_signInManager;

public UserController(UserManager<User> userManager, SignInManager<User> signInManager)

{

\_userManager = userManager;

\_signInManager = signInManager;

}

[HttpPost("register")]

public async Task<IActionResult> RegisterUser([FromBody] RegistrationModel model)

{

if (ModelState.IsValid)

{

var user = new User { UserName = model.Username };

var result = await \_userManager.CreateAsync(user, model.Password);

if (result.Succeeded)

{

await \_signInManager.SignInAsync(user, isPersistent: false);

return Ok(new { Message = "User registered successfully" });

}

foreach (var error in result.Errors)

{

ModelState.AddModelError(string.Empty, error.Description);

}

}

return BadRequest(new { Message = "Failed to register user. Please check the provided information." });

}

}

```

### Step 6: Update Authentication Schemes in `Startup.cs`

Update the authentication schemes in the `Startup.cs` file to use the Identity configuration:

```csharp

// Startup.cs

public void ConfigureServices(IServiceCollection services)

{

// ... other configurations

services.AddAuthentication(options =>

{

options.DefaultScheme = IdentityConstants.ApplicationScheme;

options.DefaultSignInScheme = IdentityConstants.ExternalScheme;

})

.AddIdentityCookies(options => { });

// ... other configurations

}

```

### Step 7: Enable User Registration Endpoint

Make sure to enable the registration endpoint in your `Configure` method in `Startup.cs`:

```csharp

// Startup.cs

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

// ... other configurations

app.UseRouting();

app.UseAuthentication();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

// ... other endpoint configurations

endpoints.MapControllers();

});

}

```

### Notes:

- Ensure that you have a valid database connection string configured in `appsettings.json` or `appsettings.Development.json`.

- Customize the user registration logic and model as needed for your application.

- The `RegisterUser` method in the `UserController` creates a new user using ASP.NET Core Identity, signs in the user, and returns a success message if registration is successful.

After these steps, you should have a basic user registration endpoint in your application. Adjust the code based on your specific requirements and security policies.

If you want to implement user registration logic using a custom `\_reservationService` instead of ASP.NET Core Identity, you'll need to handle user registration, including password hashing and storage, on your own. Below is a basic example of how you can achieve this using the `\_reservationService` and a UserRepository.

### Step 1: Create a UserRepository

Create a `UserRepository` class to handle user-related operations, including storing hashed passwords:

```csharp

// Infrastructure/Repositories/UserRepository.cs

using AcmeRoomReservation.Domain.Entities;

using BCrypt.Net;

public class UserRepository

{

public void RegisterUser(string username, string password)

{

var hashedPassword = HashPassword(password);

var user = new User

{

UserName = username,

PasswordHash = hashedPassword

};

// Save the user to the database or your data store

// Implement the actual database logic based on your data access strategy

// \_dbContext.Users.Add(user);

// \_dbContext.SaveChanges();

}

private string HashPassword(string password)

{

// Use BCrypt to hash the password

return BCrypt.Net.BCrypt.HashPassword(password, BCrypt.Net.BCrypt.GenerateSalt());

}

}

```

### Step 2: Update ReservationService to Use UserRepository

Update the `ReservationService` to use the `UserRepository` for user registration:

```csharp

// Services/ReservationService.cs

using AcmeRoomReservation.Infrastructure.Repositories;

public class ReservationService

{

private readonly RoomRepository \_roomRepository;

private readonly UserRepository \_userRepository;

public ReservationService(RoomRepository roomRepository, UserRepository userRepository)

{

\_roomRepository = roomRepository;

\_userRepository = userRepository;

}

public bool ReserveRoom(Guid userId, Guid roomId, DateTime startTime, DateTime endTime)

{

// Validate reservation logic (e.g., check for overlapping appointments)

if (!IsRoomAvailable(roomId, startTime, endTime))

{

return false; // Room is not available for the specified time frame

}

// Save the reservation to the database

var reservation = new Reservation

{

UserId = userId,

RoomId = roomId,

StartTime = startTime,

EndTime = endTime

};

// Add the reservation to the database

\_roomRepository.SaveReservation(reservation);

return true; // Successfully reserved the room

}

public void RegisterUser(string username, string password)

{

// Implement user registration logic using UserRepository

\_userRepository.RegisterUser(username, password);

}

private bool IsRoomAvailable(Guid roomId, DateTime startTime, DateTime endTime)

{

// Check if the room is available for the specified time frame

var existingReservations = \_roomRepository.GetReservationsForRoom(roomId);

foreach (var reservation in existingReservations)

{

if (startTime < reservation.EndTime && endTime > reservation.StartTime)

{

return false; // Overlapping appointment found

}

}

return true; // Room is available

}

}

```

### Step 3: Update UserController to Use ReservationService for Registration

Update the `UserController` to use the `ReservationService` for user registration:

```csharp

// Controllers/UserController.cs

using Microsoft.AspNetCore.Mvc;

using AcmeRoomReservation.Services;

[ApiController]

[Route("api/[controller]")]

public class UserController : ControllerBase

{

private readonly ReservationService \_reservationService;

public UserController(ReservationService reservationService)

{

\_reservationService = reservationService;

}

[HttpPost("register")]

public IActionResult RegisterUser([FromBody] RegistrationModel model)

{

// Implement user registration logic using \_reservationService

\_reservationService.RegisterUser(model.Username, model.Password);

return Ok(new { Message = "User registered successfully" });

}

}

public class RegistrationModel

{

public string Username { get; set; }

public string Password { get; set; }

// Add other registration properties as needed

}

```

In this example, the `UserRepository` is responsible for storing user data, including the hashed password. The `ReservationService` orchestrates the user registration process and interacts with the `UserRepository`. Adjust the code based on your specific needs and data access strategy.

To implement room reservation logic using the `\_reservationService`, you'll need to enhance the `\_reservationService` to include functionality for room reservations. Below is an example of how you can achieve this:

### Step 1: Update ReservationService

Update the `ReservationService` to include room reservation logic:

```csharp

// Services/ReservationService.cs

using System;

using AcmeRoomReservation.Infrastructure.Repositories;

public class ReservationService

{

private readonly RoomRepository \_roomRepository;

private readonly UserRepository \_userRepository;

public ReservationService(RoomRepository roomRepository, UserRepository userRepository)

{

\_roomRepository = roomRepository;

\_userRepository = userRepository;

}

public bool ReserveRoom(Guid userId, Guid roomId, DateTime startTime, DateTime endTime)

{

// Validate reservation logic (e.g., check for overlapping appointments)

if (!IsRoomAvailable(roomId, startTime, endTime))

{

return false; // Room is not available for the specified time frame

}

// Save the reservation to the database

var reservation = new Reservation

{

UserId = userId,

RoomId = roomId,

StartTime = startTime,

EndTime = endTime

};

// Add the reservation to the database

\_roomRepository.SaveReservation(reservation);

return true; // Successfully reserved the room

}

public void RegisterUser(string username, string password)

{

// Implement user registration logic using UserRepository

\_userRepository.RegisterUser(username, password);

}

private bool IsRoomAvailable(Guid roomId, DateTime startTime, DateTime endTime)

{

// Check if the room is available for the specified time frame

var existingReservations = \_roomRepository.GetReservationsForRoom(roomId);

foreach (var reservation in existingReservations)

{

if (startTime < reservation.EndTime && endTime > reservation.StartTime)

{

return false; // Overlapping appointment found

}

}

return true; // Room is available

}

}

```

In this example, the `ReserveRoom` method checks if the room is available for the specified time frame, and if so, it creates and saves a new reservation to the database using the `RoomRepository`. The actual logic may vary based on your application's requirements and data access strategy.

### Step 2: Update RoomController

Update the `RoomController` to handle room reservation requests:

```csharp

// Controllers/RoomController.cs

using Microsoft.AspNetCore.Mvc;

using System;

using AcmeRoomReservation.Services;

[ApiController]

[Route("api/[controller]")]

public class RoomController : ControllerBase

{

private readonly ReservationService \_reservationService;

public RoomController(ReservationService reservationService)

{

\_reservationService = reservationService;

}

[HttpPost("reserve")]

public IActionResult ReserveRoom([FromBody] ReservationRequest model)

{

// Implement room reservation logic using \_reservationService

var isSuccess = \_reservationService.ReserveRoom(model.UserId, model.RoomId, model.StartTime, model.EndTime);

if (isSuccess)

{

return Ok(new { Message = "Room reserved successfully" });

}

return BadRequest(new { Message = "Failed to reserve the room. Please choose a different time." });

}

}

public class ReservationRequest

{

public Guid UserId { get; set; }

public Guid RoomId { get; set; }

public DateTime StartTime { get; set; }

public DateTime EndTime { get; set; }

// Add other reservation properties as needed

}

```

In this example, the `ReserveRoom` endpoint in `RoomController` calls the `ReserveRoom` method in `ReservationService` and returns appropriate responses based on the result.

Adjust the code based on your specific requirements and data access strategy.

To implement logic to fetch room recommendations based on the `\_recommendationService`, you'll need to enhance the `\_recommendationService` to include functionality for recommending rooms. Below is an example of how you can achieve this:

### Step 1: Update RecommendationService

Update the `RecommendationService` to include room recommendation logic:

```csharp

// Services/RecommendationService.cs

using System;

using System.Collections.Generic;

using AcmeRoomReservation.Infrastructure.Repositories;

public class RecommendationService

{

private readonly RoomRepository \_roomRepository;

public RecommendationService(RoomRepository roomRepository)

{

\_roomRepository = roomRepository;

}

public List<Room> GetRecommendedRooms(Guid userId, DateTime startTime, DateTime endTime)

{

// Implement logic to fetch recommended rooms based on user preferences, history, etc.

// For simplicity, this example returns all available rooms, but you should customize this based on your requirements.

var allRooms = \_roomRepository.GetAllRooms();

return allRooms;

}

}

```

In this example, the `GetRecommendedRooms` method retrieves a list of all available rooms from the `RoomRepository`. You should customize this method based on your specific recommendation logic, which might involve considering user preferences, historical data, or other factors.

### Step 2: Update RecommendationController

Update the `RecommendationController` to handle room recommendation requests:

```csharp

// Controllers/RecommendationController.cs

using Microsoft.AspNetCore.Mvc;

using System;

using AcmeRoomReservation.Services;

[ApiController]

[Route("api/[controller]")]

public class RecommendationController : ControllerBase

{

private readonly RecommendationService \_recommendationService;

public RecommendationController(RecommendationService recommendationService)

{

\_recommendationService = recommendationService;

}

[HttpGet("recommendations")]

public IActionResult GetRecommendations(Guid userId, DateTime startTime, DateTime endTime)

{

// Implement logic to fetch room recommendations using \_recommendationService

var recommendations = \_recommendationService.GetRecommendedRooms(userId, startTime, endTime);

return Ok(recommendations);

}

}

```

In this example, the `GetRecommendations` endpoint in `RecommendationController` calls the `GetRecommendedRooms` method in `RecommendationService` and returns the list of recommended rooms.

Adjust the code based on your specific recommendation logic and data access strategy. You might want to consider factors such as room availability, user preferences, historical data, and other criteria when recommending rooms.

Keep in mind that the example provided here is simplified, and you should tailor it to meet the specific requirements and complexity of your recommendation system.

To implement logic to fetch room recommendations based on the `\_recommendationService`, you'll need to enhance the `\_recommendationService` to include functionality for recommending rooms. Below is an example of how you can achieve this:

### Step 1: Update RecommendationService

Update the `RecommendationService` to include room recommendation logic:

```csharp

// Services/RecommendationService.cs

using System;

using System.Collections.Generic;

using AcmeRoomReservation.Infrastructure.Repositories;

public class RecommendationService

{

private readonly RoomRepository \_roomRepository;

public RecommendationService(RoomRepository roomRepository)

{

\_roomRepository = roomRepository;

}

public List<Room> GetRecommendedRooms(Guid userId, DateTime startTime, DateTime endTime)

{

// Implement logic to fetch recommended rooms based on user preferences, history, etc.

// For simplicity, this example returns all available rooms, but you should customize this based on your requirements.

var allRooms = \_roomRepository.GetAllRooms();

return allRooms;

}

}

```

In this example, the `GetRecommendedRooms` method retrieves a list of all available rooms from the `RoomRepository`. You should customize this method based on your specific recommendation logic, which might involve considering user preferences, historical data, or other factors.

### Step 2: Update RecommendationController

Update the `RecommendationController` to handle room recommendation requests:

```csharp

// Controllers/RecommendationController.cs

using Microsoft.AspNetCore.Mvc;

using System;

using AcmeRoomReservation.Services;

[ApiController]

[Route("api/[controller]")]

public class RecommendationController : ControllerBase

{

private readonly RecommendationService \_recommendationService;

public RecommendationController(RecommendationService recommendationService)

{

\_recommendationService = recommendationService;

}

[HttpGet("recommendations")]

public IActionResult GetRecommendations(Guid userId, DateTime startTime, DateTime endTime)

{

// Implement logic to fetch room recommendations using \_recommendationService

var recommendations = \_recommendationService.GetRecommendedRooms(userId, startTime, endTime);

return Ok(recommendations);

}

}

```

In this example, the `GetRecommendations` endpoint in `RecommendationController` calls the `GetRecommendedRooms` method in `RecommendationService` and returns the list of recommended rooms.

Adjust the code based on your specific recommendation logic and data access strategy. You might want to consider factors such as room availability, user preferences, historical data, and other criteria when recommending rooms.

Keep in mind that the example provided here is simplified, and you should tailor it to meet the specific requirements and complexity of your recommendation system.

Implementing logic to fetch recommended rooms based on user preferences, history, and other factors can be a complex task that requires careful consideration of various criteria. Below is a simplified example that you can use as a starting point. Keep in mind that in a real-world scenario, you might need to incorporate more sophisticated algorithms and data to provide meaningful recommendations.

### Step 1: Enhance RecommendationService

Update the `RecommendationService` to include more sophisticated recommendation logic:

```csharp

// Services/RecommendationService.cs

using System;

using System.Collections.Generic;

using AcmeRoomReservation.Infrastructure.Repositories;

public class RecommendationService

{

private readonly RoomRepository \_roomRepository;

private readonly UserRepository \_userRepository;

public RecommendationService(RoomRepository roomRepository, UserRepository userRepository)

{

\_roomRepository = roomRepository;

\_userRepository = userRepository;

}

public List<Room> GetRecommendedRooms(Guid userId, DateTime startTime, DateTime endTime)

{

// Get user preferences and history (this is a simplified example, customize as needed)

var userPreferences = \_userRepository.GetUserPreferences(userId);

var userHistory = \_userRepository.GetUserHistory(userId);

// Get all available rooms

var allRooms = \_roomRepository.GetAllRooms();

// Filter and score rooms based on user preferences, history, etc.

var scoredRooms = ScoreRooms(allRooms, userPreferences, userHistory);

// Sort rooms by score (highest score first)

scoredRooms.Sort((a, b) => b.Score.CompareTo(a.Score));

// Return the top recommended rooms

var topRecommendedRooms = scoredRooms.Take(5).Select(room => room.Room).ToList();

return topRecommendedRooms;

}

private List<ScoredRoom> ScoreRooms(List<Room> rooms, UserPreferences userPreferences, UserHistory userHistory)

{

var scoredRooms = new List<ScoredRoom>();

foreach (var room in rooms)

{

// Implement scoring logic based on user preferences, history, etc.

// This is a simplified example; adjust the scoring algorithm based on your requirements

var score = CalculateRoomScore(room, userPreferences, userHistory);

scoredRooms.Add(new ScoredRoom { Room = room, Score = score });

}

return scoredRooms;

}

private double CalculateRoomScore(Room room, UserPreferences userPreferences, UserHistory userHistory)

{

// Implement scoring algorithm based on user preferences, history, etc.

// This is a simplified example; adjust the scoring algorithm based on your requirements

// Example: Score higher if the room has preferred resources

var resourceScore = room.Resources.Count(r => userPreferences.PreferredResources.Contains(r.Type)) \* 0.5;

// Example: Score higher if the room has been frequently used by the user

var usageScore = userHistory.FrequentRooms.ContainsKey(room.Id) ? userHistory.FrequentRooms[room.Id] \* 0.3 : 0;

// Combine scores

var totalScore = resourceScore + usageScore;

return totalScore;

}

}

public class ScoredRoom

{

public Room Room { get; set; }

public double Score { get; set; }

}

```

### Step 2: Update User and Room Entities

Make sure that your `User` and `Room` entities include relevant properties such as preferences, history, and resources:

```csharp

// Domain/Entities/User.cs

public class User

{

// ... other properties

public UserPreferences Preferences { get; set; }

public UserHistory History { get; set; }

}

public class UserPreferences

{

public List<ResourceType> PreferredResources { get; set; }

// Add other preference properties as needed

}

public class UserHistory

{

public Dictionary<Guid, int> FrequentRooms { get; set; }

// Add other history properties as needed

}

```

```csharp

// Domain/Entities/Room.cs

public class Room

{

// ... other properties

public List<Resource> Resources { get; set; }

}

public class Resource

{

public ResourceType Type { get; set; }

// Add other resource properties as needed

}

public enum ResourceType

{

Whiteboard,

TV,

Projector,

// Add other resource types as needed

}

```

### Step 3: Update UserRepository

Update the `UserRepository` to include methods for fetching user preferences and history:

```csharp

// Infrastructure/Repositories/UserRepository.cs

public class UserRepository

{

// ... other methods

public UserPreferences GetUserPreferences(Guid userId)

{

// Implement logic to fetch user preferences from the database

// This is a simplified example; replace with actual database access

return new UserPreferences { PreferredResources = new List<ResourceType> { ResourceType.Projector, ResourceType.Whiteboard } };

}

public UserHistory GetUserHistory(Guid userId)

{

// Implement logic to fetch user history from the database

// This is a simplified example; replace with actual database access

return new UserHistory { FrequentRooms = new Dictionary<Guid, int> { { Guid.NewGuid(), 5 }, { Guid.NewGuid(), 3 } } };

}

}

```

### Step 4: Update RecommendationController

Update the `RecommendationController` to handle room recommendation requests:

```csharp

// Controllers/RecommendationController.cs

using Microsoft.AspNetCore.Mvc;

using System;

using AcmeRoomReservation.Services;

[ApiController]

[Route("api/[controller]")]

public class RecommendationController : ControllerBase

{

private readonly RecommendationService \_recommendationService;

public RecommendationController(RecommendationService recommendationService)

{

\_recommendationService = recommendationService;

}

[HttpGet("recommendations")]

public IActionResult GetRecommendations(Guid userId, DateTime startTime, DateTime endTime)

{

// Implement logic to fetch room recommendations using \_recommendationService

var recommendations = \_recommendationService.GetRecommendedRooms(userId, startTime, endTime);

return Ok(recommendations);

}

}

```

In this example, the `GetRecommendedRooms` method in `RecommendationService` fetches user preferences and history, scores each room based on the scoring algorithm, and returns the top recommended rooms.

Please note that this is a simplified example, and you may need to adapt and expand it based on your specific use case and data model. The scoring algorithm and the factors considered in the example are placeholders, and you should adjust them based on the real criteria for room recommendations in your application.

Configuring Dependency Injection (DI) in a .NET Core application involves registering services and dependencies so that they can be injected into the classes that require them. Here's how you can configure DI in your ACME application:

### Step 1: ConfigureServices method in Startup.cs

In the `Startup.cs` file, you'll find the `ConfigureServices` method. This is where you configure DI for your application.

```csharp

// Startup.cs

public void ConfigureServices(IServiceCollection services)

{

// ... other configurations

// Register your repositories

services.AddScoped<RoomRepository>();

services.AddScoped<UserRepository>();

// Register your services

services.AddScoped<ReservationService>();

services.AddScoped<RecommendationService>();

// ... other configurations

services.AddControllers();

}

```

In this example:

- `AddScoped`: It specifies that a new instance of the service is created for each scope (usually per request in a web application).

### Step 2: Controller Injection

Controllers in ASP.NET Core automatically get their dependencies injected through the constructor. Make sure your controllers have the necessary dependencies in their constructors.

```csharp

// Controllers/RoomController.cs

[ApiController]

[Route("api/[controller]")]

public class RoomController : ControllerBase

{

private readonly ReservationService \_reservationService;

public RoomController(ReservationService reservationService)

{

\_reservationService = reservationService;

}

// ... actions

}

```

```csharp

// Controllers/UserController.cs

[ApiController]

[Route("api/[controller]")]

public class UserController : ControllerBase

{

private readonly ReservationService \_reservationService;

public UserController(ReservationService reservationService)

{

\_reservationService = reservationService;

}

// ... actions

}

```

```csharp

// Controllers/RecommendationController.cs

[ApiController]

[Route("api/[controller]")]

public class RecommendationController : ControllerBase

{

private readonly RecommendationService \_recommendationService;

public RecommendationController(RecommendationService recommendationService)

{

\_recommendationService = recommendationService;

}

// ... actions

}

```

### Step 3: Service Injection

If your services have dependencies, make sure they are injected into the service's constructor.

```csharp

// Services/ReservationService.cs

public class ReservationService

{

private readonly RoomRepository \_roomRepository;

private readonly UserRepository \_userRepository;

public ReservationService(RoomRepository roomRepository, UserRepository userRepository)

{

\_roomRepository = roomRepository;

\_userRepository = userRepository;

}

// ... methods

}

```

```csharp

// Services/RecommendationService.cs

public class RecommendationService

{

private readonly RoomRepository \_roomRepository;

private readonly UserRepository \_userRepository;

public RecommendationService(RoomRepository roomRepository, UserRepository userRepository)

{

\_roomRepository = roomRepository;

\_userRepository = userRepository;

}

// ... methods

}

```

### Notes:

- Adjust the scope of the services (`AddScoped`, `AddTransient`, or `AddSingleton`) based on your application's requirements. For web applications, `AddScoped` is often appropriate.

- This example assumes you have repositories and services for `Room` and `User` entities. Adjust the registration and injection based on your actual classes and dependencies.

With these steps, your dependencies are configured for injection, and the ASP.NET Core DI system will take care of providing the necessary instances to your controllers and services.

In an ASP.NET Core application, routing is configured in the `Startup.cs` file. The routing configuration defines how incoming requests are mapped to specific controllers and actions. Here's how you can configure routing for your ACME application:

### Step 1: Configure Routes in `Startup.cs`

Open the `Startup.cs` file, and locate the `ConfigureServices` and `Configure` methods.

In the `ConfigureServices` method, ensure that the `AddControllers` method is called to enable controller discovery and configuration.

```csharp

// Startup.cs

public void ConfigureServices(IServiceCollection services)

{

// ... other configurations

services.AddControllers();

// ... other configurations

}

```

In the `Configure` method, configure routing using `UseEndpoints`. You can specify the routes for your controllers within the `MapControllerRoute` method.

```csharp

// Startup.cs

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

// ... other production environment configurations

}

app.UseHttpsRedirection();

app.UseRouting();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute(

name: "default",

pattern: "{controller}/{action}/{id?}",

defaults: new { controller = "Home", action = "Index" });

});

}

```

In the `MapControllerRoute` method:

- `name`: The name of the route.

- `pattern`: The URL pattern to match.

- `defaults`: Default values for the controller, action, and other parameters.

### Step 2: Controller and Action Naming Conventions

Ensure that your controllers and actions follow the naming conventions to be easily matched by the routing system. By default, controllers should end with "Controller", and actions should be named following RESTful conventions.

For example:

```csharp

// Controllers/RoomController.cs

[ApiController]

[Route("api/[controller]")]

public class RoomController : ControllerBase

{

private readonly ReservationService \_reservationService;

public RoomController(ReservationService reservationService)

{

\_reservationService = reservationService;

}

[HttpPost("reserve")]

public IActionResult ReserveRoom([FromBody] ReservationRequest model)

{

// ... reserve room logic

}

}

```

### Step 3: Test Your Routes

After configuring routing, you can test your routes by making HTTP requests to your application's endpoints. Ensure that the URLs in your requests match the configured routes.

For example, assuming your application is running locally, you can use tools like Postman or curl to test the routes:

```bash

# Example curl command for reserving a room

curl -X POST -H "Content-Type: application/json" -d '{"userId": "yourUserId", "roomId": "yourRoomId", "startTime": "2024-01-25T09:00:00", "endTime": "2024-01-25T10:00:00"}' http://localhost:5000/api/room/reserve

```

This command sends a POST request to the `api/room/reserve` endpoint. Adjust the URL and payload based on your actual routes and request structure.

Ensure that the routes are correctly mapped to your controllers and actions, and the corresponding logic is executed as expected.

To implement a Notification Service in your ACME application, you can follow these steps. A Notification Service typically involves sending notifications (emails, messages, etc.) to users based on certain events or triggers in your system.

### Step 1: Define a Notification Model

Create a model to represent the information needed for a notification. This could include details such as the recipient, message content, subject, etc.

```csharp

// Models/NotificationModel.cs

public class NotificationModel

{

public string RecipientEmail { get; set; }

public string Subject { get; set; }

public string Message { get; set; }

}

```

### Step 2: Create a Notification Service Interface

Define an interface for the Notification Service. This will allow for different implementations if needed in the future.

```csharp

// Services/INotificationService.cs

public interface INotificationService

{

void SendNotification(NotificationModel notification);

}

```

### Step 3: Implement an Email Notification Service

Create an implementation of the `INotificationService` interface for sending email notifications. You may use a third-party library like `SmtpClient` or a dedicated email service.

```csharp

// Services/EmailNotificationService.cs

using System.Net;

using System.Net.Mail;

public class EmailNotificationService : INotificationService

{

public void SendNotification(NotificationModel notification)

{

using (var client = new SmtpClient("your-smtp-server.com"))

{

// Configure SMTP client settings

client.UseDefaultCredentials = false;

client.Credentials = new NetworkCredential("your-email@example.com", "your-email-password");

client.Port = 587;

client.EnableSsl = true;

// Create and send email

var mailMessage = new MailMessage("your-email@example.com", notification.RecipientEmail)

{

Subject = notification.Subject,

Body = notification.Message,

IsBodyHtml = true

};

client.Send(mailMessage);

}

}

}

```

### Step 4: Register the Notification Service

Register the Notification Service in the `Startup.cs` file to make it available for dependency injection.

```csharp

// Startup.cs

public void ConfigureServices(IServiceCollection services)

{

// ... other configurations

services.AddScoped<INotificationService, EmailNotificationService>();

// ... other configurations

}

```

### Step 5: Use the Notification Service in your Application Logic

Inject the `INotificationService` into the class where you want to send notifications and use it as needed.

```csharp

// Services/ReservationService.cs

public class ReservationService

{

private readonly RoomRepository \_roomRepository;

private readonly UserRepository \_userRepository;

private readonly INotificationService \_notificationService;

public ReservationService(RoomRepository roomRepository, UserRepository userRepository, INotificationService notificationService)

{

\_roomRepository = roomRepository;

\_userRepository = userRepository;

\_notificationService = notificationService;

}

public bool ReserveRoom(Guid userId, Guid roomId, DateTime startTime, DateTime endTime)

{

// Your reservation logic here

// Send notification to the user

var user = \_userRepository.GetUserById(userId);

var notification = new NotificationModel

{

RecipientEmail = user.Email,

Subject = "Room Reservation Confirmation",

Message = "Your room reservation has been confirmed. Details: ..."

};

\_notificationService.SendNotification(notification);

return true; // Successfully reserved the room

}

}

```

In this example, when a room reservation is confirmed, the `ReservationService` sends a confirmation email to the user using the `INotificationService`. Adjust the notification content and format based on your requirements.

Please note that the email sending logic in `EmailNotificationService` is a basic example, and in a production scenario, you may want to use a dedicated email service or a more feature-rich library for sending emails securely and efficiently.