Authentication Microservice

Authorization Microservice

Room Reservation Microservice

User Profile Microservice

Setting up a web frontend involves creating a user interface for interacting with your microservices. You can use various frontend technologies such as React, Angular, or Vue.js. Below, I'll provide a basic example using React with a simple project structure and components.

Setting up an Authentication Microservice involves creating a .NET Core project, configuring IdentityServer4, and implementing the necessary components for user authentication. Below are the steps to set up a basic Authentication Microservice using IdentityServer4:

1. Create a new .NET Core project:

- Open a terminal or command prompt and navigate to the directory where you want to create the project.

- Run the following command to create a new .NET Core Web API project:

dotnet new webapi -n AuthenticationMicroservice

- Change into the project directory:

cd AuthenticationMicroservice

2. Install necessary packages:

- Run the following commands to install the IdentityServer4 packages:

dotnet add package IdentityServer4

dotnet add package IdentityServer4.AccessTokenValidation

3. Configure IdentityServer4:

- Create a new class named Startup.cs in the project directory.

- Configure IdentityServer4 in the `Startup.cs` file:

```csharp

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Hosting;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Hosting;

using IdentityServer4;

namespace AuthenticationMicroservice

{

public class Startup

{

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

public IConfiguration Configuration { get; }

public void ConfigureServices(IServiceCollection services)

{

services.AddIdentityServer()

.AddInMemoryApiResources(new[]

{

new ApiResource("resourceApi", "Resource API")

})

.AddInMemoryClients(new[]

{

new Client

{

ClientId = "client",

AllowedGrantTypes = GrantTypes.ClientCredentials,

ClientSecrets =

{

new Secret("secret".Sha256())

},

AllowedScopes = { "resourceApi" }

}

});

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseRouting();

app.UseIdentityServer();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

}

}

}

4. Create a Controller for Token Generation:\*\*

- Create a new controller named `TokenController.cs`:

```csharp

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc;

using System.Threading.Tasks;

using IdentityServer4.Models;

using IdentityServer4.Test;

namespace AuthenticationMicroservice.Controllers

{

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

[ApiController]

public class TokenController : ControllerBase

{

[HttpPost]

[AllowAnonymous]

public async Task<IActionResult> GetTokenAsync()

{

var tokenResponse = await HttpContext

.GetTokenAsync(IdentityServerConstants.TokenRequest.ClientId);

if (tokenResponse != null)

{

return Ok(new { access\_token = tokenResponse });

}

return BadRequest("Error generating token.");

}

}

}

5. Run the Microservice:

- Use the following command to run the microservice:

dotnet run

- The IdentityServer4 will run on `https://localhost:5001`.

6. Test the Authentication Microservice:

- Use a tool like Postman or cURL to request a token from the authentication microservice.

curl --request POST \

--url https://localhost:5001/connect/token \

--header 'Content-Type: application/x-www-form-urlencoded' \

--data 'grant\_type=client\_credentials&client\_id=client&client\_secret=secret&scope=resourceApi'

- You should receive an access token in the response.

This is a basic setup for an Authentication Microservice using IdentityServer4. Depending on your application requirements, you might need to customize the configuration and integrate with a user database for more advanced scenarios.

Setting up an Authorization Microservice involves creating a .NET Core project, configuring authorization rules, and implementing components for validating user access to specific resources. Below are the steps to set up a basic Authorization Microservice:

\*\*1. Create a new .NET Core project:\*\*

- Open a terminal or command prompt and navigate to the directory where you want to create the project.

- Run the following command to create a new .NET Core Web API project

dotnet new webapi -n AuthorizationMicroservice

- Change into the project directory:

```bash

cd AuthorizationMicroservice

```

\*\*2. Install necessary packages:\*\*

- Run the following command to install required packages:

```bash

dotnet add package Microsoft.AspNetCore.Authorization

```

3. Configure Authorization:

- Open the `Startup.cs` file in the project directory.

- Configure authorization in the `Startup.cs` file:

```csharp

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Hosting;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Hosting;

using Microsoft.AspNetCore.Authorization;

namespace AuthorizationMicroservice

{

public class Startup

{

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

public IConfiguration Configuration { get; }

public void ConfigureServices(IServiceCollection services)

{

services.AddControllers();

services.AddAuthorization(options =>

{

options.AddPolicy("CanAccessResourceApi", policy =>

{

policy.RequireAuthenticatedUser();

policy.RequireClaim("scope", "resourceApi");

});

});

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseRouting();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

}

}

} ```

\*\*4. Create a Controller for Resource Access:\*\*

- Create a new controller named `ResourceController.cs`:

```csharp

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc;

namespace AuthorizationMicroservice.Controllers

{

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

[ApiController]

public class ResourceController : ControllerBase

{

[HttpGet]

[Authorize("CanAccessResourceApi")]

public IActionResult Get()

{

return Ok("Authorized to access the resource!");

}

}

}

```

\*\*5. Run the Microservice:\*\*

- Use the following command to run the microservice:

```bash

dotnet run

```

- The Authorization Microservice will run on `https://localhost:5001`.

\*\*6. Test the Authorization Microservice:\*\*

- Use a tool like Postman or cURL to test access to the protected resource.

```bash

curl --request GET \

--url https://localhost:5001/api/resource \

--header 'Authorization: Bearer <ACCESS\_TOKEN>'

```

- Replace `<ACCESS\_TOKEN>` with the access token obtained from the Authentication Microservice.

This is a basic setup for an Authorization Microservice using ASP.NET Core. Depending on your application's complexity and requirements, you may need to extend the authorization logic and integrate with IdentityServer4 or other identity providers. Additionally, you might want to use a more sophisticated authorization approach, such as policy-based authorization or custom requirements.

Setting up a Room Reservation Microservice involves creating a .NET Core project, implementing the business logic for room reservations, and setting up communication with other microservices, such as the Authentication and Authorization Microservices. Below are the steps to set up a basic Room Reservation Microservice:

\*\*1. Create a new .NET Core project:\*\*

- Open a terminal or command prompt and navigate to the directory where you want to create the project.

- Run the following command to create a new .NET Core Web API project:

```bash

dotnet new webapi -n RoomReservationMicroservice

```

- Change into the project directory:

```bash

cd RoomReservationMicroservice

```

\*\*2. Install necessary packages:\*\*

- Run the following command to install any required packages based on your specific needs. For example, you may need packages for working with databases, HTTP clients, or JWT validation.

```bash

dotnet add package <package-name>

```

- Make sure to install packages for features like JWT validation (`Microsoft.AspNetCore.Authentication.JwtBearer`), HTTP client (`System.Net.Http.Json`), or Entity Framework if you plan to use a database.

\*\*3. Configure the Microservice:\*\*

- Open the `Startup.cs` file in the project directory.

- Configure the microservice in the `Startup.cs` file, including setting up middleware, services, and any necessary configurations.

```csharp

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Hosting;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Hosting;

namespace RoomReservationMicroservice

{

public class Startup

{

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

public IConfiguration Configuration { get; }

public void ConfigureServices(IServiceCollection services)

{

// Configure services, e.g., database context, authentication, etc.

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseRouting();

// Configure middleware, e.g., authentication, authorization, etc.

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

}

}

}

```

\*\*4. Implement Room Reservation Logic:\*\*

- Create controllers and services for handling room reservations, availability checks, and recommendations.

```csharp

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc;

namespace RoomReservationMicroservice.Controllers

{

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

[ApiController]

[Authorize("CanAccessResourceApi")]

public class ReservationController : ControllerBase

{

// Implement actions for room reservations, availability checks, etc.

}

}

```

\*\*5. Integrate with Authentication and Authorization Microservices:\*\*

- Ensure the microservice can communicate with the Authentication and Authorization Microservices. Use the appropriate mechanisms, such as JWT validation for authentication.

\*\*6. Configure Dependency Injection:\*\*

- Register services and dependencies needed by the microservice in the `ConfigureServices` method of `Startup.cs`.

```csharp

services.AddScoped<IReservationService, ReservationService>();

```

\*\*7. Implement Business Logic:\*\*

- Write the actual business logic for room reservations, ensuring that it adheres to the requirements outlined by Acme.

\*\*8. Test the Microservice:\*\*

- Run the microservice using the following command:

```bash

dotnet run

```

- Test the endpoints using tools like Postman or cURL.

This is a basic setup for a Room Reservation Microservice. Depending on your application requirements, you may need to integrate with additional services, databases, or external APIs. Additionally, consider implementing features such as recommendations and availability checks based on the constraints mentioned earlier.

Certainly! Below is a basic implementation of actions for room reservations, availability checks, and recommendations within a `ReservationController` in a .NET Core Web API project. This example assumes that you have a service (`ReservationService`) responsible for handling the business logic related to room reservations.

```csharp

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc;

using System;

using RoomReservationMicroservice.Services;

namespace RoomReservationMicroservice.Controllers

{

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

[ApiController]

[Authorize("CanAccessResourceApi")]

public class ReservationController : ControllerBase

{

private readonly IReservationService \_reservationService;

public ReservationController(IReservationService reservationService)

{

\_reservationService = reservationService;

}

[HttpPost("reserve")]

public IActionResult ReserveRoom([FromBody] ReservationRequest request)

{

try

{

// Validate the request model

if (!ModelState.IsValid)

{

return BadRequest("Invalid reservation request.");

}

// Call the reservation service to handle the reservation logic

var reservationResult = \_reservationService.ReserveRoom(request);

if (reservationResult.Success)

{

return Ok("Room reserved successfully.");

}

else

{

return BadRequest(reservationResult.ErrorMessage);

}

}

catch (Exception ex)

{

// Log the exception for debugging purposes

Console.WriteLine($"Error during room reservation: {ex.Message}");

return StatusCode(500, "An unexpected error occurred.");

}

}

[HttpGet("check-availability")]

public IActionResult CheckAvailability([FromQuery] AvailabilityCheckRequest request)

{

try

{

// Validate the request model

if (!ModelState.IsValid)

{

return BadRequest("Invalid availability check request.");

}

// Call the reservation service to check room availability

var availabilityResult = \_reservationService.CheckRoomAvailability(request);

if (availabilityResult)

{

return Ok("Room is available for the given date and time.");

}

else

{

return BadRequest("Room is not available for the given date and time.");

}

}

catch (Exception ex)

{

// Log the exception for debugging purposes

Console.WriteLine($"Error during availability check: {ex.Message}");

return StatusCode(500, "An unexpected error occurred.");

}

}

[HttpGet("recommendations")]

public IActionResult GetRecommendations([FromQuery] RecommendationRequest request)

{

try

{

// Validate the request model

if (!ModelState.IsValid)

{

return BadRequest("Invalid recommendation request.");

}

// Call the reservation service to get room recommendations

var recommendations = \_reservationService.GetRoomRecommendations(request);

return Ok(recommendations);

}

catch (Exception ex)

{

// Log the exception for debugging purposes

Console.WriteLine($"Error during room recommendations: {ex.Message}");

return StatusCode(500, "An unexpected error occurred.");

}

}

}

}

```

In this example, the `ReservationController` exposes three endpoints:

1. `POST /api/reservation/reserve`: To reserve a room based on the provided `ReservationRequest`.

2. `GET /api/reservation/check-availability`: To check the availability of a room based on the provided `AvailabilityCheckRequest`.

3. `GET /api/reservation/recommendations`: To get room recommendations based on the provided `RecommendationRequest`.

Make sure to replace the placeholders (`ReservationRequest`, `AvailabilityCheckRequest`, `RecommendationRequest`, and `IReservationService`) with your actual request and service implementations.

Additionally, remember to handle exceptions appropriately and include proper logging and error handling mechanisms in a production environment.

To integrate with Authentication and Authorization Microservices, you'll typically use JWT (JSON Web Token) for authentication and make use of the provided access token to authorize requests to your Room Reservation Microservice. Below are the steps to integrate with Authentication and Authorization Microservices:

\*\*1. Add Authentication Middleware:\*\*

- Configure your `Startup.cs` to use JWT-based authentication. You need to specify the authentication scheme, validate the issuer and audience, and set the token validation parameters.

```csharp

using Microsoft.AspNetCore.Authentication.JwtBearer;

using Microsoft.IdentityModel.Tokens;

using System.Text;

// ...

public void ConfigureServices(IServiceCollection services)

{

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(options =>

{

options.TokenValidationParameters = new TokenValidationParameters

{

ValidateIssuer = true,

ValidateAudience = true,

ValidateLifetime = true,

ValidateIssuerSigningKey = true,

ValidIssuer = "your-issuer", // Specify the issuer of the token

ValidAudience = "your-audience", // Specify the audience

IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("your-secret-key")) // Specify the secret key for validation

};

});

// ... Other service configurations

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

// ... Other middleware configurations

app.UseAuthentication(); // Add this line to enable authentication

// ... Other middleware configurations

}

```

\*\*2. Secure Actions with Authorization:\*\*

- Decorate the actions or controllers that require authorization with the `[Authorize]` attribute. You can specify roles or policies if needed.

```csharp

[Authorize("CanAccessResourceApi")]

public class ReservationController : ControllerBase

{

// ... Your actions

}

```

\*\*3. Extract User Information from Claims:\*\*

- You can access user information from the claims provided in the token. Extract necessary information (e.g., user ID, roles) for authorization purposes.

```csharp

var userId = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;

var userRoles = User.FindAll(ClaimTypes.Role).Select(c => c.Value).ToList();

```

\*\*4. Make Authorized Requests to Other Microservices:\*\*

- If your Authorization Microservice provides a role or scope claim in the token, use it to authorize access to specific actions or resources.

```csharp

[Authorize(Roles = "admin")]

public IActionResult AdminAction()

{

// ... Admin-specific logic

}

```

```csharp

[Authorize("CanAccessResourceApi")]

public class ReservationController : ControllerBase

{

// ... Your actions

}

```

\*\*5. Call the Authentication Microservice for User Information:\*\*

- If you need additional user information, you can call the Authentication Microservice to get user details based on the sub (subject) claim in the token.

```csharp

// Example: Call to Authentication Microservice

var user = await \_authenticationService.GetUserDetailsAsync(userId);

```

\*\*6. Handle Token Expiration and Refresh:\*\*

- Implement token expiration handling and, if needed, token refresh logic. You may need to redirect the user to the Authentication Microservice for token refresh.

\*\*Note:\*\*

- Replace placeholder values (`"your-issuer"`, `"your-audience"`, `"your-secret-key"`) with your actual values.

- Ensure that the Authorization Microservice issues tokens with the necessary claims (roles, scopes) for your authorization requirements.

This integration enables your Room Reservation Microservice to authenticate users using JWTs issued by the Authentication Microservice and authorize requests based on the provided claims (roles or scopes). Always follow security best practices, including secure key management and appropriate token validation settings.

To implement business logic in your Room Reservation Microservice, you'll typically define service interfaces, create service implementations, and use them in your controller actions. Below is a simple example that outlines how to structure the business logic for room reservations:

\*\*1. Define Service Interfaces:\*\*

```csharp

public interface IReservationService

{

ReservationResult ReserveRoom(ReservationRequest request);

bool CheckRoomAvailability(AvailabilityCheckRequest request);

List<RoomRecommendation> GetRoomRecommendations(RecommendationRequest request);

}

\*\*2. Create Service Implementations:\*\*

```csharp

public class ReservationService : IReservationService

{

private readonly IReservationRepository \_reservationRepository; // Assume you have a repository for database operations

public ReservationService(IReservationRepository reservationRepository)

{

\_reservationRepository = reservationRepository;

}

public ReservationResult ReserveRoom(ReservationRequest request)

{

// Implement reservation logic

// Check for overlapping appointments, room capacity, office hours, etc.

// Save reservation details to the database

// Example:

if (\_reservationRepository.CheckOverlappingAppointments(request))

{

return new ReservationResult { Success = false, ErrorMessage = "Overlapping appointments detected." };

}

// Additional checks and logic...

\_reservationRepository.CreateReservation(request);

return new ReservationResult { Success = true };

}

public bool CheckRoomAvailability(AvailabilityCheckRequest request)

{

// Implement availability check logic

// Check for overlapping appointments, office hours, etc.

// Example:

return !\_reservationRepository.CheckOverlappingAppointments(request);

}

public List<RoomRecommendation> GetRoomRecommendations(RecommendationRequest request)

{

// Implement recommendation logic

// Consider factors such as past reservations, room popularity, etc.

// Example:

return \_reservationRepository.GetRecommendedRooms(request);

}

}

```

\*\*3. Implement Repository for Database Operations:\*\*

```csharp

public interface IReservationRepository

{

bool CheckOverlappingAppointments(RequestBase request);

void CreateReservation(ReservationRequest request);

List<RoomRecommendation> GetRecommendedRooms(RecommendationRequest request);

}

```

\*\*4. Inject Services into Controller:\*\*

```csharp

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

[ApiController]

[Authorize("CanAccessResourceApi")]

public class ReservationController : ControllerBase

{

private readonly IReservationService \_reservationService;

public ReservationController(IReservationService reservationService)

{

\_reservationService = reservationService;

}

[HttpPost("reserve")]

public IActionResult ReserveRoom([FromBody] ReservationRequest request)

{

var result = \_reservationService.ReserveRoom(request);

if (result.Success)

{

return Ok("Room reserved successfully.");

}

else

{

return BadRequest(result.ErrorMessage);

}

}

[HttpGet("check-availability")]

public IActionResult CheckAvailability([FromQuery] AvailabilityCheckRequest request)

{

var isAvailable = \_reservationService.CheckRoomAvailability(request);

if (isAvailable)

{

return Ok("Room is available for the given date and time.");

}

else

{

return BadRequest("Room is not available for the given date and time.");

}

}

[HttpGet("recommendations")]

public IActionResult GetRecommendations([FromQuery] RecommendationRequest request)

{

var recommendations = \_reservationService.GetRoomRecommendations(request);

return Ok(recommendations);

}

}

`

\*\*5. Implement the Repository for Database Operations:\*\*

The repository should handle interactions with the database, such as checking for overlapping appointments, creating reservations, and fetching room recommendations.

```csharp

public class ReservationRepository : IReservationRepository

{

private readonly YourDbContext \_dbContext; // Replace with your actual database context

public ReservationRepository(YourDbContext dbContext)

{

\_dbContext = dbContext;

}

public bool CheckOverlappingAppointments(RequestBase request)

{

// Implement logic to check for overlapping appointments in the database

// Example: Check if there are existing reservations for the same room and overlapping time

return \_dbContext.Reservations

.Any(r =>

r.RoomId == request.RoomId &&

r.StartTime < request.EndTime &&

r.EndTime > request.StartTime);

}

public void CreateReservation(ReservationRequest request)

{

// Implement logic to save the reservation details to the database

var reservation = new Reservation

{

// Map properties from the request to the reservation entity

RoomId = request.RoomId,

StartTime = request.StartTime,

EndTime = request.EndTime,

// Additional properties...

};

\_dbContext.Reservations.Add(reservation);

\_dbContext.SaveChanges();

}

public List<RoomRecommendation> GetRecommendedRooms(RecommendationRequest request)

{

// Implement logic to fetch recommended rooms based on historical data, popularity, etc.

// Example: Fetch rooms with the highest number of past reservations

return \_dbContext.Reservations

.GroupBy(r => r.RoomId)

.OrderByDescending(group => group.Count())

.Take(request.NumberOfRecommendations)

.Select(group => new RoomRecommendation

{

RoomId = group.Key,

ReservationCount = group.Count(),

// Additional properties...

})

.ToList();

}

}

\*\*Note:\*\*

- Adjust the examples based on your specific database structure and requirements.

- Replace `YourDbContext`, `Reservation`, and other types with your actual database context, entity types, and repository implementations.

This structure provides a separation of concerns between the business logic in the services, database operations in the repository, and the controller handling HTTP requests. It also facilitates unit testing and maintainability.

Setting up a User Profile Microservice involves creating a .NET Core project, defining service interfaces for user-related operations, implementing service logic, and exposing endpoints for user profile management. Below are the steps to set up a basic User Profile Microservice:

\*\*1. Create a new .NET Core project:\*\*

- Open a terminal or command prompt and navigate to the directory where you want to create the project.

- Run the following command to create a new .NET Core Web API project:

```bash

dotnet new webapi -n UserProfileMicroservice

- Change into the project directory:

cd UserProfileMicroservice

\*\*2. Install necessary packages:\*\*

- Run the following command to install any required packages based on your specific needs. For example, you may need packages for working with databases, HTTP clients, or JWT validation.

```bash

dotnet add package <package-name>

```

- Make sure to install packages for features like Entity Framework if you plan to use a database.

\*\*3. Configure the Microservice:\*\*

- Open the `Startup.cs` file in the project directory.

- Configure the microservice in the `Startup.cs` file, including setting up middleware, services, and any necessary configurations.

```csharp

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Hosting;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Hosting;

namespace UserProfileMicroservice

{

public class Startup

{

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

public IConfiguration Configuration { get; }

public void ConfigureServices(IServiceCollection services)

{

// Configure services, e.g., database context, authentication, etc.

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseRouting();

// Configure middleware, e.g., authentication, authorization, etc.

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

}

}

}

\*\*4. Define Service Interfaces:\*\*

```csharp

public interface IUserProfileService

{

UserProfile GetUserProfile(string userId);

void UpdateUserProfile(string userId, UserProfileUpdateRequest updateRequest);

}

```

\*\*5. Create Service Implementations:\*\*

public class UserProfileService : IUserProfileService

{

private readonly IUserProfileRepository \_userProfileRepository;

public UserProfileService(IUserProfileRepository userProfileRepository)

{

\_userProfileRepository = userProfileRepository;

}

public UserProfile GetUserProfile(string userId)

{

// Implement logic to get user profile details from the repository

return \_userProfileRepository.GetUserProfile(userId);

}

public void UpdateUserProfile(string userId, UserProfileUpdateRequest updateRequest)

{

// Implement logic to update user profile details in the repository

\_userProfileRepository.UpdateUserProfile(userId, updateRequest);

}

}

```

\*\*6. Implement Repository for Database Operations:\*\*

```csharp

public interface IUserProfileRepository

{

UserProfile GetUserProfile(string userId);

void UpdateUserProfile(string userId, UserProfileUpdateRequest updateRequest);

}

```

\*\*7. Implement the Repository for Database Operations:\*\*

```csharp

public class UserProfileRepository : IUserProfileRepository

{

private readonly YourDbContext \_dbContext; // Replace with your actual database context

public UserProfileRepository(YourDbContext dbContext)

{

\_dbContext = dbContext;

}

public UserProfile GetUserProfile(string userId)

{

// Implement logic to get user profile details from the database

return \_dbContext.UserProfiles.SingleOrDefault(u => u.UserId == userId);

}

public void UpdateUserProfile(string userId, UserProfileUpdateRequest updateRequest)

{

// Implement logic to update user profile details in the database

var user = \_dbContext.UserProfiles.SingleOrDefault(u => u.UserId == userId);

if (user != null)

{

// Update user properties based on the update request

user.FirstName = updateRequest.FirstName;

user.LastName = updateRequest.LastName;

// Additional properties...

\_dbContext.SaveChanges();

}

}

}

```

\*\*8. Inject Services into Controller:\*\*

```csharp

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

[ApiController]

[Authorize("CanAccessResourceApi")]

public class UserProfileController : ControllerBase

{

private readonly IUserProfileService \_userProfileService;

public UserProfileController(IUserProfileService userProfileService)

{

\_userProfileService = userProfileService;

}

[HttpGet("{userId}")]

public IActionResult GetUserProfile(string userId)

{

var userProfile = \_userProfileService.GetUserProfile(userId);

if (userProfile != null)

{

return Ok(userProfile);

}

else

{

return NotFound($"User profile not found for user with ID: {userId}");

}

}

[HttpPut("{userId}")]

public IActionResult UpdateUserProfile(string userId, [FromBody] UserProfileUpdateRequest updateRequest)

{

\_userProfileService.UpdateUserProfile(userId, updateRequest);

return Ok("User profile updated successfully.");

}

}

```

\*\*Note:\*\*

- Adjust the examples

Setting up a web frontend involves creating a user interface for interacting with your microservices. You can use various frontend technologies such as React, Angular, or Vue.js. Below, I'll provide a basic example using React with a simple project structure and components.

### Step 1: Create a React App

```bash

npx create-react-app room-reservation-app

cd room-reservation-app

```

### Step 2: Install Required Packages

```bash

npm install axios react-router-dom

```

### Step 3: Project Structure

Place your components, styles, and other assets in appropriate folders. For simplicity, we'll create a basic structure:

```

src/

|-- components/

| |-- UserProfile.js

| |-- RoomReservation.js

|-- services/

| |-- ApiService.js

|-- App.js

|-- index.js

```

### Step 4: Implement Components

#### `UserProfile.js`

```jsx

import React, { useState, useEffect } from 'react';

import ApiService from '../services/ApiService';

const UserProfile = ({ userId }) => {

const [userProfile, setUserProfile] = useState(null);

useEffect(() => {

const fetchUserProfile = async () => {

try {

const response = await ApiService.getUserProfile(userId);

setUserProfile(response.data);

} catch (error) {

console.error('Error fetching user profile:', error);

}

};

fetchUserProfile();

}, [userId]);

return (

<div>

<h2>User Profile</h2>

{userProfile ? (

<div>

<p>Name: {userProfile.firstName} {userProfile.lastName}</p>

{/\* Add other user profile details \*/}

</div>

) : (

<p>Loading...</p>

)}

</div>

);

};

export default UserProfile;

```

#### `RoomReservation.js`

```jsx

import React, { useState } from 'react';

import ApiService from '../services/ApiService';

const RoomReservation = () => {

const [reservationData, setReservationData] = useState({

userId: 'user123', // Replace with the actual user ID from authentication

roomId: '',

startTime: '',

endTime: '',

// Add other reservation details

});

const handleReservationSubmit = async () => {

try {

await ApiService.reserveRoom(reservationData);

alert('Room reserved successfully!');

// Optionally, redirect or perform additional actions after successful reservation

} catch (error) {

console.error('Error reserving room:', error);

alert('Error reserving room. Please try again.');

}

};

return (

<div>

<h2>Room Reservation</h2>

{/\* Add form elements for room reservation details \*/}

<button onClick={handleReservationSubmit}>Reserve Room</button>

</div>

);

};

export default RoomReservation;

```

### Step 5: Implement `ApiService.js`

```jsx

import axios from 'axios';

const BASE\_URL = 'http://localhost:5000/api'; // Replace with your API endpoint

const ApiService = {

getUserProfile: async (userId) => {

return axios.get(`${BASE\_URL}/user-profile/${userId}`);

},

reserveRoom: async (reservationData) => {

return axios.post(`${BASE\_URL}/reservation/reserve`, reservationData);

},

};

export default ApiService;

```

### Step 6: Update `App.js`

```jsx

import React from 'react';

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

import UserProfile from './components/UserProfile';

import RoomReservation from './components/RoomReservation';

const App = () => {

return (

<Router>

<div>

<Switch>

<Route path="/user-profile/:userId" component={({ match }) => <UserProfile userId={match.params.userId} />} />

<Route path="/room-reservation" component={RoomReservation} />

</Switch>

</div>

</Router>

);

};

export default App;

```

### Step 7: Run the React App

```bash

npm start

```

Visit `http://localhost:3000` in your browser to view the web application. Adjust the code based on your actual requirements and integrate with your backend microservices as needed. Make sure your backend services are running and accessible from the frontend.

Remember to handle authentication and authorization, secure API calls, and manage state appropriately based on your application's needs.

Integrating microservices typically involves making HTTP requests between the frontend and backend services. Below, I'll provide a general guide on how to integrate the frontend React app with the microservices you've set up (User Profile, Room Reservation, Authentication, and Authorization).

### 1. Update `ApiService.js` with Authentication

If your microservices require authentication, you'll need to handle authentication in your frontend. Modify the `ApiService.js` file to include authentication headers. This might involve obtaining and storing a token after the user logs in.

```jsx

// ApiService.js

import axios from 'axios';

const BASE\_URL = 'http://localhost:5000/api'; // Replace with your API endpoint

const ApiService = {

getUserProfile: async (userId, token) => {

return axios.get(`${BASE\_URL}/user-profile/${userId}`, {

headers: {

Authorization: `Bearer ${token}`,

},

});

},

reserveRoom: async (reservationData, token) => {

return axios.post(`${BASE\_URL}/reservation/reserve`, reservationData, {

headers: {

Authorization: `Bearer ${token}`,

},

});

},

};

export default ApiService;

```

### 2. Implement Authentication in React

Use a state management solution (e.g., React Context, Redux) to manage authentication state and store the user token after a successful login.

### 3. Use Components in `App.js`

Update your `App.js` file to include routing for components and pass the authentication token as a prop.

```jsx

// App.js

import React, { useState } from 'react';

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

import UserProfile from './components/UserProfile';

import RoomReservation from './components/RoomReservation';

const App = () => {

const [authToken, setAuthToken] = useState(null);

const handleLogin = (token) => {

setAuthToken(token);

};

const handleLogout = () => {

setAuthToken(null);

// Optionally, redirect or perform other actions after logout

};

return (

<Router>

<div>

<Switch>

<Route

path="/user-profile/:userId"

render={({ match }) => (

<UserProfile userId={match.params.userId} token={authToken} />

)}

/>

<Route

path="/room-reservation"

render={() => <RoomReservation token={authToken} />}

/>

</Switch>

</div>

</Router>

);

};

export default App;

```

### 4. Integrate with Microservices in Components

In your components (e.g., `UserProfile.js`, `RoomReservation.js`), call the relevant methods from `ApiService.js` and pass the authentication token.

```jsx

// UserProfile.js

import React, { useState, useEffect } from 'react';

import ApiService from '../services/ApiService';

const UserProfile = ({ userId, token }) => {

const [userProfile, setUserProfile] = useState(null);

useEffect(() => {

const fetchUserProfile = async () => {

try {

const response = await ApiService.getUserProfile(userId, token);

setUserProfile(response.data);

} catch (error) {

console.error('Error fetching user profile:', error);

}

};

fetchUserProfile();

}, [userId, token]);

// ... rest of the component

};

```

```jsx

// RoomReservation.js

import React, { useState } from 'react';

import ApiService from '../services/ApiService';

const RoomReservation = ({ token }) => {

const [reservationData, setReservationData] = useState({

userId: 'user123', // Replace with the actual user ID from authentication

roomId: '',

startTime: '',

endTime: '',

// Add other reservation details

});

const handleReservationSubmit = async () => {

try {

await ApiService.reserveRoom(reservationData, token);

alert('Room reserved successfully!');

// Optionally, redirect or perform additional actions after successful reservation

} catch (error) {

console.error('Error reserving room:', error);

alert('Error reserving room. Please try again.');

}

};

// ... rest of the component

};

```

### 5. Test Integration

Ensure that your microservices are running, and test the integration by logging in, navigating between components, and making reservations. Monitor network requests in your browser's developer tools to ensure the frontend correctly communicates with the backend.

### 6. Error Handling and Loading States

Enhance error handling by displaying appropriate messages to users in case of failures. Additionally, consider implementing loading states to provide feedback to users during asynchronous operations.

This is a high-level guide, and the specifics may vary based on your application architecture and microservice design. Consider additional factors such as error handling, security, and performance as you integrate the frontend with your microservices.

Implementing authentication and authorization in the frontend typically involves user login, storing authentication tokens securely, and controlling access to certain features based on the user's role or permissions. Below, I'll provide a basic example using React and a common approach with JSON Web Tokens (JWT) for authentication.

### Step 1: Install Required Packages

```bash

npm install axios react-router-dom

```

### Step 2: Create a Simple Authentication Service

Create a service to handle authentication-related operations, including login, logout, and storing tokens securely.

#### `AuthService.js`

```jsx

import axios from 'axios';

const BASE\_URL = 'http://localhost:5000/api'; // Replace with your API endpoint

const AuthService = {

login: async (credentials) => {

try {

const response = await axios.post(`${BASE\_URL}/auth/login`, credentials);

const { token } = response.data;

localStorage.setItem('token', token); // Store token in localStorage (you may use a more secure storage)

return token;

} catch (error) {

throw new Error('Invalid credentials');

}

},

logout: () => {

localStorage.removeItem('token');

},

getToken: () => {

return localStorage.getItem('token');

},

isAuthenticated: () => {

const token = localStorage.getItem('token');

return !!token;

},

};

export default AuthService;

```

### Step 3: Implement Login Component

Create a simple login form component.

#### `Login.js`

```jsx

import React, { useState } from 'react';

import AuthService from '../services/AuthService';

const Login = ({ onLogin }) => {

const [credentials, setCredentials] = useState({

username: '',

password: '',

});

const handleInputChange = (e) => {

setCredentials({

...credentials,

[e.target.name]: e.target.value,

});

};

const handleLogin = async (e) => {

e.preventDefault();

try {

const token = await AuthService.login(credentials);

onLogin(token);

} catch (error) {

alert('Invalid credentials');

}

};

return (

<div>

<h2>Login</h2>

<form onSubmit={handleLogin}>

<label>

Username:

<input

type="text"

name="username"

value={credentials.username}

onChange={handleInputChange}

/>

</label>

<br />

<label>

Password:

<input

type="password"

name="password"

value={credentials.password}

onChange={handleInputChange}

/>

</label>

<br />

<button type="submit">Login</button>

</form>

</div>

);

};

export default Login;

```

### Step 4: Implement App Component

Integrate authentication into your main `App.js` component.

#### `App.js`

```jsx

import React, { useState } from 'react';

import { BrowserRouter as Router, Route, Switch, Redirect } from 'react-router-dom';

import UserProfile from './components/UserProfile';

import RoomReservation from './components/RoomReservation';

import Login from './components/Login';

import AuthService from './services/AuthService';

const PrivateRoute = ({ component: Component, ...rest }) => {

return (

<Route

{...rest}

render={(props) =>

AuthService.isAuthenticated() ? (

<Component {...props} />

) : (

<Redirect to="/login" />

)

}

/>

);

};

const App = () => {

const [authToken, setAuthToken] = useState(AuthService.getToken());

const handleLogin = (token) => {

setAuthToken(token);

};

const handleLogout = () => {

AuthService.logout();

setAuthToken(null);

};

return (

<Router>

<div>

<Switch>

<PrivateRoute

path="/user-profile/:userId"

component={({ match }) => (

<UserProfile userId={match.params.userId} token={authToken} />

)}

/>

<PrivateRoute

path="/room-reservation"

component={() => <RoomReservation token={authToken} />}

/>

<Route

path="/login"

render={() =>

AuthService.isAuthenticated() ? (

<Redirect to="/user-profile" />

) : (

<Login onLogin={handleLogin} />

)

}

/>

<Route

path="/logout"

render={() => {

handleLogout();

return <Redirect to="/login" />;

}}

/>

<Route path="/">

<Redirect to="/user-profile" />

</Route>

</Switch>

</div>

</Router>

);

};

export default App;

```

### Step 5: Implement Authorization in Components

Update your components to handle authorization based on the user's role or permissions.

#### Example: `UserProfile.js`

```jsx

import React, { useState, useEffect } from 'react';

import ApiService from '../services/ApiService';

const UserProfile = ({ userId, token }) => {

const [userProfile, setUserProfile] = useState(null);

useEffect(() => {

const fetchUserProfile = async () => {

try {

const response = await ApiService.getUserProfile(userId, token);

setUserProfile(response.data);

} catch (error) {

console.error('Error fetching user profile:', error);

}

};

fetchUserProfile();

}, [userId, token]);

return (

<div>

<h2>User Profile</h2>

{userProfile ? (

<div>

<p>Name: {userProfile.firstName} {userProfile.lastName}</p>

{/\* Add other user profile details \*/}

</div>

) : (

<p>Loading...</p>

)}

</div>

);

};

export default UserProfile;

```

In this example, the `PrivateRoute` component ensures that only authenticated users can access certain routes. The `AuthService` handles login, logout, and token storage. The `UserProfile` component demonstrates making authenticated API requests.

Please note that this is a basic example, and in a real-world application, you would likely need a more sophisticated authentication and authorization mechanism, possibly involving role-based access control (RBAC), user roles, and permissions. Also, ensure secure token storage practices and consider additional security measures based on your application's requirements.

To implement room reservation logic in the frontend, you need to create a component for room reservation and integrate it with your existing React application. Below, I'll provide an example of how you can create a simple Room Reservation component.

### Step 1: Create a Room Reservation Component

#### `RoomReservation.js`

```jsx

import React, { useState } from 'react';

import ApiService from '../services/ApiService';

const RoomReservation = ({ token }) => {

const [reservationData, setReservationData] = useState({

roomId: '',

startTime: '',

endTime: '',

// Add other reservation details

});

const handleInputChange = (e) => {

setReservationData({

...reservationData,

[e.target.name]: e.target.value,

});

};

const handleReservationSubmit = async (e) => {

e.preventDefault();

try {

await ApiService.reserveRoom(reservationData, token);

alert('Room reserved successfully!');

// Optionally, redirect or perform additional actions after successful reservation

} catch (error) {

console.error('Error reserving room:', error);

alert('Error reserving room. Please try again.');

}

};

return (

<div>

<h2>Room Reservation</h2>

<form onSubmit={handleReservationSubmit}>

<label>

Room ID:

<input

type="text"

name="roomId"

value={reservationData.roomId}

onChange={handleInputChange}

/>

</label>

<br />

<label>

Start Time:

<input

type="datetime-local"

name="startTime"

value={reservationData.startTime}

onChange={handleInputChange}

/>

</label>

<br />

<label>

End Time:

<input

type="datetime-local"

name="endTime"

value={reservationData.endTime}

onChange={handleInputChange}

/>

</label>

<br />

<button type="submit">Reserve Room</button>

</form>

</div>

);

};

export default RoomReservation;

```

### Step 2: Integrate Room Reservation Component in `App.js`

Update your `App.js` file to include the `RoomReservation` component and pass the authentication token.

#### `App.js`

```jsx

import React, { useState } from 'react';

import { BrowserRouter as Router, Route, Switch, Redirect } from 'react-router-dom';

import UserProfile from './components/UserProfile';

import RoomReservation from './components/RoomReservation';

import Login from './components/Login';

import AuthService from './services/AuthService';

const PrivateRoute = ({ component: Component, ...rest }) => {

return (

<Route

{...rest}

render={(props) =>

AuthService.isAuthenticated() ? (

<Component {...props} />

) : (

<Redirect to="/login" />

)

}

/>

);

};

const App = () => {

const [authToken, setAuthToken] = useState(AuthService.getToken());

const handleLogin = (token) => {

setAuthToken(token);

};

const handleLogout = () => {

AuthService.logout();

setAuthToken(null);

};

return (

<Router>

<div>

<Switch>

<PrivateRoute

path="/user-profile/:userId"

component={({ match }) => (

<UserProfile userId={match.params.userId} token={authToken} />

)}

/>

<PrivateRoute

path="/room-reservation"

component={() => <RoomReservation token={authToken} />}

/>

<Route

path="/login"

render={() =>

AuthService.isAuthenticated() ? (

<Redirect to="/user-profile" />

) : (

<Login onLogin={handleLogin} />

)

}

/>

<Route

path="/logout"

render={() => {

handleLogout();

return <Redirect to="/login" />;

}}

/>

<Route path="/">

<Redirect to="/user-profile" />

</Route>

</Switch>

</div>

</Router>

);

};

export default App;

```

### Step 3: Implement API Service Methods

Make sure to implement the corresponding API service methods in your `ApiService.js` file to handle room reservation requests.

#### `ApiService.js`

```jsx

// Add the following method to ApiService.js

reserveRoom: async (reservationData, token) => {

return axios.post(`${BASE\_URL}/reservation/reserve`, reservationData, {

headers: {

Authorization: `Bearer ${token}`,

},

});

},

```

### Step 4: Test Room Reservation

Run your React application (`npm start`) and navigate to the Room Reservation page. Fill in the required details and submit the form to make a room reservation. Ensure that the backend microservice is properly configured to handle room reservation requests.

This example provides a basic implementation, and you may need to enhance it based on your specific requirements, such as handling responses, error messages, and loading states. Additionally, consider incorporating feedback to users about the success or failure of the room reservation request.

Securing APIs involves implementing proper authentication and authorization mechanisms. Here's a general guide on how to secure your APIs and implement authorization checks in a .NET Core environment. I'll provide an example using JWT (JSON Web Tokens) for authentication and custom authorization policies for authorization checks.

### Step 1: Secure API Endpoints with JWT Authentication

#### Install Required NuGet Packages

```bash

dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer

dotnet add package System.IdentityModel.Tokens.Jwt

```

#### Configure Authentication in `Startup.cs`

```csharp

// Startup.cs

using Microsoft.AspNetCore.Authentication.JwtBearer;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.IdentityModel.Tokens;

using System.Text;

public class Startup

{

private readonly IConfiguration \_configuration;

public Startup(IConfiguration configuration)

{

\_configuration = configuration;

}

public void ConfigureServices(IServiceCollection services)

{

// Other configurations...

// Configure JWT Authentication

var jwtSettings = \_configuration.GetSection("JwtSettings");

var secretKey = Encoding.UTF8.GetBytes(jwtSettings["Secret"]);

var tokenValidationParameters = new TokenValidationParameters

{

ValidateIssuerSigningKey = true,

IssuerSigningKey = new SymmetricSecurityKey(secretKey),

ValidateIssuer = false,

ValidateAudience = false,

RequireExpirationTime = true,

ValidateLifetime = true

};

services.AddAuthentication(options =>

{

options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

})

.AddJwtBearer(options =>

{

options.TokenValidationParameters = tokenValidationParameters;

});

// Authorization policies

services.AddAuthorization(options =>

{

options.AddPolicy("CanReserveRoom", policy =>

{

policy.RequireAuthenticatedUser();

policy.RequireClaim("Role", "User");

});

// Add more policies as needed

});

// Other configurations...

}

// Other methods...

}

```

#### Configure JWT Settings in `appsettings.json`

```json

// appsettings.json

{

"JwtSettings": {

"Secret": "your\_secret\_key\_here",

"Issuer": "your\_issuer\_here",

"Audience": "your\_audience\_here"

},

// Other settings...

}

```

### Step 2: Implement Authorization Checks in Controllers

#### Example: `RoomReservationController.cs`

```csharp

// RoomReservationController.cs

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc;

[ApiController]

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

public class RoomReservationController : ControllerBase

{

[HttpPost("reserve")]

[Authorize(Policy = "CanReserveRoom")]

public IActionResult ReserveRoom([FromBody] ReservationRequest request)

{

// Your logic for room reservation

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

}

// Other actions...

public class ReservationRequest

{

public string RoomId { get; set; }

public DateTime StartTime { get; set; }

public DateTime EndTime { get; set; }

// Other reservation details...

}

}

```

In this example, the `[Authorize]` attribute with the specified policy ensures that only authenticated users with the "User" role can access the `ReserveRoom` endpoint.

### Step 3: Generate JWT Tokens on User Login

#### Example: `AuthenticationService.cs`

```csharp

// AuthenticationService.cs

using Microsoft.IdentityModel.Tokens;

using System;

using System.IdentityModel.Tokens.Jwt;

using System.Security.Claims;

using System.Text;

public class AuthenticationService

{

private readonly IConfiguration \_configuration;

public AuthenticationService(IConfiguration configuration)

{

\_configuration = configuration;

}

public string GenerateJwtToken(string userId, string role)

{

var jwtSettings = \_configuration.GetSection("JwtSettings");

var secretKey = Encoding.UTF8.GetBytes(jwtSettings["Secret"]);

var claims = new[]

{

new Claim(ClaimTypes.NameIdentifier, userId),

new Claim(ClaimTypes.Role, role),

// Add more claims as needed

};

var tokenDescriptor = new SecurityTokenDescriptor

{

Subject = new ClaimsIdentity(claims),

Expires = DateTime.UtcNow.AddHours(Convert.ToDouble(jwtSettings["ExpirationHours"])),

SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(secretKey), SecurityAlgorithms.HmacSha256Signature)

};

var tokenHandler = new JwtSecurityTokenHandler();

var token = tokenHandler.CreateToken(tokenDescriptor);

return tokenHandler.WriteToken(token);

}

}

```

### Step 4: Use JWT Token in Frontend

Modify your frontend authentication logic to handle JWT tokens. Store the token securely and include it in the Authorization header of your API requests.

### Step 5: Test Authorization

Ensure that the entire flow works correctly by testing user logins, obtaining JWT tokens, and using those tokens to access protected API endpoints.

This is a basic setup, and you may need to customize it based on your specific requirements and security policies. Additionally, consider using HTTPS to secure communication between your frontend and backend. Always follow security best practices to protect sensitive information and prevent common vulnerabilities.

Testing is a crucial part of the software development process to ensure that your application works as expected, catches potential issues, and remains maintainable. I'll provide a brief overview of testing in the context of a .NET Core application, covering unit testing and integration testing.

### 1. Unit Testing

#### Setting Up Unit Tests

1. Create a separate test project for your application. In Visual Studio, you can add a new project of type "xUnit Test Project" or "NUnit Test Project."

2. Add references to your main project and any other projects being tested.

3. Install a testing framework like xUnit or NUnit using NuGet.

```bash

dotnet add package xunit

dotnet add package xunit.runner.visualstudio

```

4. Create test classes and methods for your application's units (individual methods, classes, etc.).

#### Writing Unit Tests

Example: Testing a simple class method.

```csharp

// Main project code

public class MathOperations

{

public int Add(int a, int b)

{

return a + b;

}

}

// Test project code

public class MathOperationsTests

{

[Fact]

public void Add\_ShouldReturnCorrectSum()

{

// Arrange

var mathOperations = new MathOperations();

// Act

var result = mathOperations.Add(2, 3);

// Assert

Assert.Equal(5, result);

}

}

```

#### Running Unit Tests

Use your preferred test runner, such as the one integrated into Visual Studio, Rider, or the command line:

```bash

dotnet test

```

### 2. Integration Testing

Integration tests verify that different parts of your system work together correctly. In the context of a .NET Core application, you can use tools like `WebApplicationFactory` to host your application during tests.

#### Setting Up Integration Tests

1. Create a separate test project for integration tests.

2. Install required NuGet packages:

```bash

dotnet add package Microsoft.AspNetCore.Mvc.Testing

```

3. Write integration tests that use the `WebApplicationFactory` to create a test server.

#### Writing Integration Tests

Example: Testing a controller.

```csharp

public class RoomReservationControllerTests : IClassFixture<WebApplicationFactory<YourStartup>>

{

private readonly WebApplicationFactory<YourStartup> \_factory;

public RoomReservationControllerTests(WebApplicationFactory<YourStartup> factory)

{

\_factory = factory;

}

[Fact]

public async Task ReserveRoom\_ShouldReturnSuccess()

{

// Arrange

var client = \_factory.CreateClient();

// Act

var response = await client.PostAsync("/api/roomreservation/reserve", new StringContent("YourReservationData"));

// Assert

response.EnsureSuccessStatusCode();

// Add more assertions as needed

}

}

```

#### Running Integration Tests

```bash

dotnet test

```

### 3. Automated Testing Tools

Consider using tools like [Postman](https://www.postman.com/) for API testing and [Selenium](https://www.selenium.dev/) for web UI testing.

### 4. Continuous Integration (CI)

Integrate testing into your CI/CD pipeline to automate testing on every code change.

### 5. Mocking

Use mocking frameworks (e.g., [Moq](https://github.com/Moq/moq4)) to isolate units being tested from external dependencies.

### 6. Code Coverage

Consider using code coverage tools (e.g., [Coverlet](https://github.com/coverlet-coverage/coverlet)) to ensure your tests cover a sufficient portion of your code.

Remember to tailor your testing strategy based on your specific application requirements and the nature of your project.