System Architecture and Security Overview

**1. Identity and Authentication**

**1.1 Duende Identity Server Integration**

* **Authorization Code Flow**: We are using the Authorization Code Flow for authentication, ensuring secure communication between the client and the authorization server. The response type is set to code, where the client first requests an authorization code and later exchanges it for an access token.
* **PKCE (Proof Key for Code Exchange)**: PKCE is implemented to mitigate injection attacks. Each request includes a unique secret (code challenge) hashed and appended to the authentication request, providing enhanced security by preventing interception or manipulation during the authorization process. This approach is employed for both the front-end (Angular) and API components.

**1.2 API Resources and Scopes**

* **API Resource Approach**: Instead of traditional scopes, we are using API Resources. This allows each API resource to define multiple scopes, providing fine-grained control over which clients can access specific parts of the API.
* **JWT Confusion Attack Prevention**: To mitigate JWT (JSON Web Token) confusion attacks, we validate tokens using **TokenValidationParameters**, ensuring that the typ (type) header of the JWT matches the expected value. This prevents attackers from substituting one type of token for another (e.g., an access token in place of an ID token).

**1.3 Attribute-Based Access Control (ABAC)**

* **ABAC with Authorization Policies**: We have implemented ABAC to enforce access control based on user attributes. This provides dynamic and flexible access control decisions based on user properties, resource attributes, and environmental conditions.
* **Fine-Grained Policies**: In addition to ABAC, fine-grained policies are applied at the scope level, enabling interactions with specific clients (rather than users). This ensures that only authorized clients can perform certain actions or access specific resources.

**2. Token Management and Security**

**2.1 Reference Tokens and Token Revocation**

* **Reference Tokens**: To control token lifecycle and improve security, we use reference tokens instead of self-contained tokens. These tokens are opaque to the client and stored securely on the server. Reference tokens provide the ability to revoke tokens centrally and efficiently manage token lifetimes.
* **Token Revocation**: Token revocation is handled at the front end. When users log out, the front-end initiates a request to revoke their tokens, ensuring that they can no longer access the API after logout.

**2.2 BFF (Back-End for Front-End) Security and YARP**

* **BFF Security**: To strengthen security between the client and API, we employ the BFF Security framework. This architecture ensures that sensitive operations (e.g., token storage and management) are handled on the back end, minimizing the attack surface exposed to the front end.
* **YARP (Yet Another Reverse Proxy)**: YARP is used to proxy requests between the front end and the API, adding a layer of security that prevents XSS (Cross-Site Scripting) and other client-side attacks. This setup ensures that direct API access is restricted and filtered.

**3. Custom Account Handling and Federation**

* **Custom Password Security**: We secure user passwords using the IPasswordHasher interface, which ensures that passwords are hashed and salted appropriately before storage.
* **Federation**: Federation is enabled for third-party authentication. Users authenticated through external identity providers (such as Facebook) are treated the same as custom users. This ensures a seamless user experience and secure authentication regardless of the provider.

**4. Microservices Architecture**

**4.1 Microservices Setup**

* **Microservices Approach**: The system is built using a microservices architecture, where different functionalities (such as notifications, background services, and user management) are separated into independent services. This modular approach improves scalability, maintainability, and fault tolerance.

**4.2 Background Services (Hangfire Integration)**

* **Background Jobs**: A dedicated microservice handles background services using **Hangfire**. This setup allows for the scheduling and execution of background tasks (e.g., sending emails, processing data) in a reliable, fault-tolerant manner.

**4.3 Messaging with RabbitMQ and MassTransit**

* **Message Queueing**: RabbitMQ is used to handle messaging between microservices, ensuring that tasks and events are communicated efficiently between services.
* **MassTransit**: MassTransit is integrated with RabbitMQ to simplify message routing, error handling, and retries. We have implemented retry logic and circuit breakers to ensure that failed messages are retried and the system remains resilient under load.

**5. Logging and Monitoring**

* **Serilog**: For logging and monitoring, we use **Serilog** with structured logging. This ensures that all logs, particularly around email notifications and background jobs, are formatted in a way that allows for easy querying and analysis.

**6. API Design and Organization**

**6.1 Minimal API Approach**

* **Minimal API**: We use the Minimal API approach for building a modern REST API. Minimal APIs provide lightweight and high-performance endpoints, ensuring quick responses with minimal overhead.
* **Carter Modules**: To organize the Minimal APIs, we employ **Carter Modules**. This modular approach to organizing API endpoints simplifies routing, middleware configuration, and scalability.

**7. Testing and Quality Assurance**

* **xUnit Testing**: We use xUnit for unit testing our system components. All key features, including authentication, token management, messaging, and background tasks, are covered by tests to ensure reliability and correctness.

**8. Admin Panel**

8.1 **EF Core Integration**  
• **EF Core**: For the Admin Panel, we have integrated **Entity Framework Core (EF Core)** as the ORM (Object-Relational Mapping) solution. EF Core provides seamless interaction with the database, allowing for efficient querying and data manipulation.  
• **Result Pattern with Fluent Results**: Throughout the project, we are using the **Result Pattern** via **Fluent Results**. This design pattern ensures that each operation (whether success or failure) is returned in a consistent format, enhancing error handling and improving code readability. This approach ensures reliability, making it easier to track errors, failures, or successes in API calls.  
• **Secure Practices**: For securing the Admin Panel, we have employed various technologies such as **Duende IdentityServer** for authentication, **YARP** (Yet Another Reverse Proxy) for secure API communication, and strict security protocols to prevent vulnerabilities.

**9. Room Service**

9.1 **CosmosDB Integration**  
• **NoSQL with CosmosDB**: For the **Room Service**, we are utilizing **CosmosDB**, a NoSQL database. This allows for highly scalable, flexible, and low-latency data management, particularly suited for storing room-related information.  
• **Data Storage in JSON Format**: To ensure better data structuring and efficient storage, room service data is stored in **JSON format** within CosmosDB collections. This approach optimizes how data is retrieved and updated, allowing for more flexible querying.  
• **Performance and Scalability**: By leveraging CosmosDB, the Room Service is able to scale effortlessly, ensuring performance remains optimal, even under heavy loads or when handling complex data structures.