Projemiz, bir **bankacılık uygulaması için güvenli bir REST API** geliştirmeyi hedefliyor ve bu hedefe ulaşmak için Spring Boot, Spring Security ve JPA (Hibernate) gibi teknolojileri kullanıyor.

Proje Genel Mantığı ve Katmanlı Mimari

Proje, **katmanlı bir mimari** (layered architecture) prensibine göre tasarlanmıştır. Bu, kodun farklı sorumluluklara sahip katmanlara ayrılması anlamına gelir. Bu yaklaşım, kodun daha düzenli, bakımı kolay ve ölçeklenebilir olmasını sağlar. Temel katmanlarımız şunlardır:

1. **Entity Katmanı:** Veritabanındaki tabloları temsil eden Java sınıfları.
2. **Repository (DAO) Katmanı:** Veritabanı ile doğrudan etkileşimi sağlayan arayüzler.
3. **Service Katmanı:** İş mantığını içeren ve Repository katmanını kullanan sınıflar.
4. **Controller Katmanı:** Gelen HTTP isteklerini karşılayan ve yanıtları döndüren sınıflar.
5. **Config Katmanı:** Uygulamanın genel yapılandırmasını (özellikle güvenlik) yapan sınıflar.
6. **DTO Katmanı:** Veri transferi nesneleri (request/response modelleri).

Adım Adım Proje Geliştirme ve Mantığı

1. **Temel Spring Boot Yapılandırması ve Bağımlılıklar:**
    - **Ne Yaptık:** Projeyi Spring Boot çatısı altında başlattık ve `Spring Web`, `Spring Boot Security` gibi temel bağımlılıkları `pom.xml` dosyasına ekledik. Ayrıca `Project Lombok`'u da ekleyerek getter/setter gibi tekrar eden kodları otomatik oluşturmasını sağladık. Uygulamayı 9000 portundan çalışacak şekilde yapılandırdık.
    - **Bu Yüzden:** Spring Boot, REST API geliştirmeyi kolaylaştırır ve gerekli tüm bileşenleri otomatik yapılandırır. Spring Security, kimlik doğrulama ve yetkilendirme işlemlerini güvenli bir şekilde yönetmemizi sağlar. Lombok ise geliştirme hızımızı artırır ve kod kalabalığını azaltır.

2. **Veritabanı Şeması ve Entity Katmanı:**
    - **Ne Yaptık:** Veritabanında `bank` adında bir şema oluşturduk. Bu şema altında `member`, `role` ve `account` tablolarını temsil eden Entity sınıflarını (Member, Role, Account) tanımladık.
        - `Member`: Kullanıcı bilgilerini (email, password) tutar ve Spring Security entegrasyonu için `UserDetails` arayüzünü implement eder. `Role` ile Many-to-Many ilişkisi vardır.
        - `Role`: Kullanıcı rollerini (`ADMIN`, `USER` gibi) tutar ve Spring Security için `GrantedAuthority` arayüzünü implement eder.
        - `Account`: Banka hesap bilgilerini (id, name) tutar.
    - **Bu Yüzden:** JPA (Java Persistence API) kullanarak Java nesnelerimizi veritabanı tablolarına eşledik. Bu, doğrudan SQL sorguları yazmak yerine Java objeleriyle çalışarak veritabanı işlemlerini daha kolay ve Object-Oriented bir şekilde yapmamızı sağlar. `bank` şeması, veritabanı düzenimizi organize eder.

3. **Repository (DAO) Katmanı:**
    - **Ne Yaptık:** Her bir Entity (Member, Role, Account) için `JpaRepository`'den miras alan interface'ler (MemberRepository, RoleRepository, AccountRepository) oluşturduk. `MemberRepository` ve `RoleRepository` için özel sorgular (`findByEmail`, `findByAuthority`) ekledik.
    - **Bu Yüzden:** Spring Data JPA sayesinde, temel CRUD (Create, Read, Update, Delete) operasyonları için kod yazmamıza gerek kalmaz. `JpaRepository` bu metotları otomatik olarak sağlar. Özel sorgular ise belirli kriterlere göre veri çekmemizi mümkün kılar (örneğin, bir kullanıcının e-posta adresiyle bulunması).

4. **DTO (Data Transfer Object) Katmanı:**
    - **Ne Yaptık:** Veri transferi için `RegisterResponse` ve `RegistrationMember` adında iki adet Java `record` (DTO) tanımladık.
        - `RegistrationMember`: Kullanıcı kayıt isteği sırasında istemciden alınan email ve password bilgilerini taşır.
        - `RegisterResponse`: Kayıt işlemi başarılı olduğunda API'den istemciye dönülecek email ve mesaj bilgilerini taşır.
    - **Bu Yüzden:** DTO'lar, API istekleri ve yanıtları için belirli veri yapılarını tanımlar. Bu, hassas bilgilerin (örneğin şifrenin düz metin olarak) doğrudan entity nesneleri üzerinden dolaşmasını engeller ve API'nin daha temiz ve güvenli olmasını sağlar. Record'lar bu amaç için kısa ve değişmez bir yapı sunar.

5. **Service Katmanı:**
    - **Ne Yaptık:** İş mantığını içeren servis sınıflarını ve arayüzlerini (AccountService, AccountServiceImpl, AuthenticationService, UserService) oluşturduk.
        - `AccountService` / `AccountServiceImpl`: Banka hesapları üzerinde `findAll` ve `save` gibi CRUD işlemlerini yönetir, Repository katmanı ile etkileşime geçer.
        - `AuthenticationService`: Yeni kullanıcı kaydı (`register`) işlemini yönetir. Gelen şifreleri `PasswordEncoder` ile şifreler, rolleri atar ve Member'ı veritabanına kaydeder.
        - `UserService`: Spring Security'nin `UserDetailsService` arayüzünü implement eder. `loadUserByUsername` metodunu kullanarak, kimlik doğrulama sırasında kullanıcıyı (Member) e-posta adresine göre veritabanından yükler.
    - **Bu Yüzden:** Servis katmanı, uygulamanın iş kurallarını barındırır. Repository katmanındaki düşük seviyeli veritabanı operasyonlarını soyutlar ve Controller katmanına daha yüksek seviyeli işlevler sunar. `AuthenticationService` ve `UserService`, güvenlik süreçlerinin yönetilmesinde kritik rol oynar.

6. **Config Katmanı (Spring Security Yapılandırması):**
    - **Ne Yaptık:** `SecurityConfig` sınıfını oluşturarak Spring Security'nin nasıl davranacağını yapılandırdık.
        - `PasswordEncoder` (BCryptPasswordEncoder) bean'ini tanımladık.
        - `AuthenticationManager` bean'ini tanımladık ve `DaoAuthenticationProvider` ile kendi `UserDetailsService` (UserService) ve `PasswordEncoder`'ımızı bağladık.
        - `SecurityFilterChain` bean'ini tanımlayarak CSRF'yi devre dışı bıraktık, `/auth/**` ve `/welcome/**` gibi yollara herkese açık erişim izni verdik.
        - `AccountController`'daki endpoint'ler için rol tabanlı yetkilendirme kuralları belirledik:
            - `GET /account/**`: ADMIN ve USER rollerine izinli.
            - `POST`, `PUT`, `DELETE /account/**`: Sadece ADMIN rolüne izinli.
        - Varsayılan form tabanlı giriş ve HTTP Basic kimlik doğrulamasını etkinleştirdik.
    - **Bu Yüzden:** Bu katman, API'mızın güvenliğini sağlar. Kullanıcıların nasıl kimlik doğrulaması yapacağını (JDBC authentication ile), şifrelerin nasıl şifreleneceğini ve hangi kullanıcının hangi role sahip olduğunda hangi kaynaklara erişebileceğini merkezi olarak yönetir. Bu sayede, hassas bankacılık işlemleri yetkisiz erişime karşı korunur.

7. **Controller Katmanı:**
    - **Ne Yaptık:** Gelen HTTP isteklerini işlemek ve yanıt döndürmek için `AccountController` ve `AuthController` sınıflarını tanımladık.
        - `AccountController`: Banka hesaplarıyla ilgili CRUD operasyonları için endpoint'ler (`GET /account`, `POST /account`) içerir. `AccountService`'i kullanarak iş mantığına erişir.
        - `AuthController`: Kullanıcı kayıt işlemi için `/auth/register` POST endpoint'ini sunar. `AuthenticationService`'i kullanarak yeni kullanıcıları kaydeder.
    - **Bu Yüzden:** Kontrolcüler, uygulamanın dış dünyaya açılan kapısıdır. İstemcilerden gelen HTTP isteklerini alır, servis katmanına yönlendirir, gerekli iş mantığı uygulandıktan sonra elde edilen sonuçları HTTP yanıtı olarak geri döndürür. Bu katman, RESTful prensiplerine uygun endpoint'ler tanımlar.

Eksik Kalan ve İyileştirilebilecek Noktalar (Daha Önce Belirttiğim gibi)

- `AccountService` ve `AccountController`'da eksik olan `findById`, `update` ve `delete` metotlarının eklenmesi gerekmektedir.
- `AuthenticationService`'deki `register` metodu şu an her kullanıcıya "ADMIN" rolü atama eğiliminde. Proje gereksinimine göre "bir user, bir admin" rolünde kullanıcı oluşturulması için bu kısım dinamikleştirilebilir veya başlangıçta varsayılan "USER" rolü atanabilir.
- Proje hedeflerinde belirtilen **OAuth2 entegrasyonu (GitHub ile)** henüz kodda görünmüyor. Bu, SecurityConfig'e ve muhtemelen yeni bir kimlik doğrulama sağlayıcısına eklenmesi gereken bir özelliktir.

Bu genel yapı ve her katmanın belirli bir sorumluluğu olması, projenin okunabilirliğini, yönetilebilirliğini ve gelecekteki geliştirmelere açıklığını büyük ölçüde artırır.