Proje Genel Bakışı: Katmanlı Mimari ile Film ve Aktör Yönetimi API'si
Bu proje, Spring Boot kullanarak filmler ve aktörler için RESTful bir API geliştirmeyi amaçlamaktadır. Projenin ana hedefi, CRUD (Create, Read, Update, Delete) operasyonlarını destekleyen, iyi tasarlanmış, katmanlı ve hata yönetimini içeren bir backend servisi oluşturmaktır.

Proje Katmanları ve Anlamları
Projemiz, modern bir Spring Boot uygulamasında yaygın olarak kullanılan katmanlı (n-tier) mimari prensiplerine göre tasarlanmıştır. Her katmanın belirli bir sorumluluğu vardır ve bu sayede kod daha düzenli, bakımı kolay ve ölçeklenebilir olur.

1. Entity Katmanı 📚
- Anlamı: Veritabanındaki tabloların Java objeleri olarak temsil edildiği katmandır. Object-Relational Mapping (ORM) prensibine göre çalışır.
- Ne Kullandık:
  - Movie: Filmleri temsil eden entity sınıfı. id, name, directorName, rating, releaseDate alanlarına sahiptir.
  - Actor: Aktörleri temsil eden entity sınıfı. id, firstName, lastName, gender, birthDate alanlarına sahiptir.
  - Gender (Enum): Aktörlerin cinsiyetini belirten bir enum.
  - JPA Anotasyonları (@Entity, @Table, @Id, @GeneratedValue, @Column, @ManyToMany, @JoinTable): Java objelerini veritabanı tablolarına eşlemek için kullanıldı.
  - Lombok Anotasyonları (@Data, @NoArgsConstructor, @AllArgsConstructor): Getter, Setter, yapıcı metotlar gibi tekrar eden kodları otomatik oluşturarak kodu temiz tutmak için kullanıldı.
  - Jakarta Validation Anotasyonları (@NotNull, @NotBlank, @Size, @Positive): Veritabanına kaydedilmeden önce verilerin belirli kurallara uygunluğunu kontrol etmek için kullanıldı.
- İlişki: Movie ve Actor arasında Many-to-Many ilişki kuruldu. Bu, bir filmin birden çok aktörü olabileceği ve bir aktörün birden çok filmde rol alabileceği anlamına gelir. Movie tarafında mappedBy kullanılırken, Actor tarafında ilişki tablosu (movie_actor) @JoinTable ile açıkça tanımlandı.

2. Repository Katmanı (DAO) 💾
- Anlamı: Veritabanı ile doğrudan iletişim kuran, veri erişim işlemlerini soyutlayan katmandır. CRUD operasyonlarını gerçekleştirir.
- Ne Kullandık:
  - MovieRepository: Movie entity'si için veri erişim operasyonlarını sağlayan arayüz.
  - ActorRepository: Actor entity'si için veri erişim operasyonlarını sağlayan arayüz.
  - Spring Data JPA (JpaRepository): JDBC veya elle SQL yazmak yerine, temel CRUD operasyonlarını (findAll, findById, save, delete) arayüzler aracılığıyla otomatik olarak sağlayan güçlü bir Spring modülü. Bu sayede kod yazımımız minimuma indi.

3. Service Katmanı ⚙️

- Anlamı: Uygulamanın iş mantığını (business logic) barındıran katmandır. Controller'dan gelen istekleri alır, Repository katmanını kullanarak veri erişimi yapar ve gerektiğinde birden fazla Repository işlemini bir araya getirerek bir iş akışı oluşturur (transaction yönetimi gibi).
- Ne Kullandık:
  - MovieService (Arayüz) & MovieServiceImpl (İmplementasyon): Filmlerle ilgili iş mantığını ve operasyonları tanımlar/implement eder.
  - ActorService (Arayüz) & ActorServiceImpl (İmplementasyon): Aktörlerle ilgili iş mantığını ve operasyonları tanımlar/implement eder.
  - Bağımlılık Enjeksiyonu (Constructor Injection): Service sınıfları, ihtiyaç duydukları Repository'leri (örn: MovieServiceImpl'in MovieRepository'ye ihtiyacı var) constructor'ları aracılığıyla alırlar. Bu, test edilebilirliği artırır ve Spring IoC Container'ın yönetimini sağlar.
  - Hata Fırlatma (ApiException): findById gibi metotlarda, beklenen bir kaynak bulunamadığında custom ApiException fırlatarak hata yönetim sistemimizle entegrasyon sağlandı.

4. Controller Katmanı 🌐

- Anlamı: Dış dünyadan (web tarayıcıları, mobil uygulamalar vb.) gelen HTTP isteklerini karşılayan ve HTTP yanıtlarını döndüren katmandır. İş mantığını Service katmanına delege eder.
- Ne Kullandık:
  - MovieController: Filmlerle ilgili HTTP endpoint'lerini tanımlar (GET, POST, PUT, DELETE).
  - ActorController: Aktörlerle ilgili HTTP endpoint'lerini tanımlar (GET, POST, PUT, DELETE).
  - @RestController: Sınıfı bir REST kontrolcüsü olarak işaretler.
  - @RequestMapping: Endpoint'lerin temel URL yolunu belirler (örn: /workintech/movies, /workintech/actors).
  - @GetMapping, @PostMapping, @PutMapping, @DeleteMapping: CRUD operasyonları için HTTP metotlarını URL yollarıyla eşler.
  - @PathVariable: URL'deki değişkenleri metot parametrelerine bağlar (örn: /movies/{id}).
  - @RequestBody: HTTP isteğinin JSON gövdesini Java objelerine (DTO'lara) dönüştürür.
  - DTO'lar (MovieActorRequest, ActorRequest, ActorResponse): Kontrolcülerin istemciden aldığı veya istemciye döndürdüğü veri yapılarını tanımlar. Bu, entity sınıflarını doğrudan ifşa etmeden daha esnek ve kontrollü bir API sunar. ActorResponse için Java Record kullanıldı, bu modern ve temiz bir yaklaşımdır.
  - Loglama (@Slf4j): Her bir endpoint'te gelen istekleri ve gerçekleşen aksiyonları loglamak için kullanıldı, bu da hata ayıklama ve izleme için önemlidir.

5. Exceptions (Hata Yönetimi) Katmanı ⚠️

- Anlamı: Uygulamada meydana gelen hataları merkezi ve tutarlı bir şekilde yakalamak, işlemek ve istemciye uygun HTTP yanıtları olarak döndürmek için tasarlanmıştır.
- Ne Kullandık:
  - ApiException: Uygulamaya özgü özel bir RuntimeException sınıfı. Hata mesajı ve döndürülecek HTTP durum kodunu içerir. Servis katmanında kaynak bulunamadığı durumlarda fırlatıldı.
  - GlobalExceptionHandler: @ControllerAdvice ve @ExceptionHandler anotasyonları ile tüm Controller'lardan fırlayan hataları global olarak yakalar. ApiException'ları ve genel Exception'ları ayrı ayrı ele alarak istemciye ExceptionResponse formatında, uygun HTTP durum koduyla (örn: 404 Not Found, 500 Internal Server Error) yanıt döner.
  - ExceptionResponse: İstemciye dönülecek hata yanıtının yapısını (mesaj, durum kodu, zaman damgası) tanımlayan bir DTO.

6. Util (Yardımcı) Katmanı 🛠️

- Anlamı: Belirli bir katmana ait olmayan, ancak çeşitli katmanlarda ihtiyaç duyulan genel yardımcı metotları veya sınıfları barındırır.
- Ne Kullandık:
  - Converter: Entity objelerini DTO objelerine dönüştürmek için statik metotlar içeren bir yardımcı sınıf. Bu, dönüşüm mantığını merkezi bir yerde tutar ve kod tekrarını önler.

Projeyi Yapım Sırası ve Dikkat Edilenler
Projeyi adım adım aşağıdaki mantıkla inşa ettim:

1. Proje Kurulumu ve Temel Paket Yapısı: Spring Initializr ile temel Spring Boot projesi oluşturuldu. Bağımlılıklar (Web, JPA, Lombok, PostgreSQL) eklendi. Gerekli paketler (entity, repository, service, controller, exceptions, dto, util) oluşturularak projenin iskeleti hazırlandı.
2. Entity Katmanı: Movie ve Actor entity'leri tanımlandı, Many-to-Many ilişki kuruldu. JPA ve Lombok anotasyonları ile veritabanı eşlemesi ve boilerplate kod azaltma sağlandı. Validasyon kuralları (@NotNull, @Size vb.) eklendi.
3. Veritabanı Ayarları: application.properties dosyası ile PostgreSQL bağlantısı ve Hibernate DDL otomatik güncelleme ayarları yapıldı. SQL logları aktif edildi.
4. Repository Katmanı: MovieRepository ve ActorRepository arayüzleri, Spring Data JPA'nın JpaRepository'sini extend ederek temel CRUD operasyonlarını otomatik olarak elde etti.
5. Service Katmanı: MovieService, MovieServiceImpl, ActorService, ActorServiceImpl sınıfları oluşturuldu. İş mantığı burada konumlandırıldı ve Repository katmanı ile etkileşim sağlandı. findById metotlarında custom hata (ApiException) fırlatma entegre edildi.
6. Hata Yönetimi Katmanı: ApiException özel hata sınıfı ve GlobalExceptionHandler sınıfı ile merkezi hata yönetimi kuruldu. ExceptionResponse DTO'su ile istemciye dönülecek hata formatı belirlendi.
7. DTO Katmanı: MovieActorRequest, ActorRequest, ActorResponse gibi DTO'lar oluşturuldu. Bu, API'nin istemciyle daha esnek ve kontrollü iletişim kurmasını sağladı. Özellikle ActorResponse için Java Record kullanılarak modern bir yaklaşım sergilendi.
8. Util Katmanı: Converter sınıfı, entity-DTO dönüşümlerini merkezi bir noktada topladı.
9. Controller Katmanı: MovieController ve ActorController sınıfları implement edildi. Tüm CRUD endpoint'leri tanımlandı, Service katmanı çağrılarak iş mantığına delege edildi ve DTO'lar kullanılarak veri iletişimi sağlandı.

Katmanlar Arasındaki İlişkiler ve Bağımlılıklar

- Controller -> Service: Controller katmanı, iş mantığını yürütmek için Service katmanına bağımlıdır. Doğrudan Repository veya Entity katmanlarıyla etkileşime girmez.
- Service -> Repository: Service katmanı, veri erişimi (CRUD) için Repository katmanına bağımlıdır. Doğrudan Entity objeleriyle çalışır.
- Repository -> Entity: Repository katmanı, belirli bir Entity sınıfı üzerinde veri erişimi sağlar.
- Entity: Veritabanı yapısını tanımlar ve diğer katmanlar tarafından veri modeli olarak kullanılır.
- Exceptions: Service katmanından fırlatılır ve Controller katmanındaki GlobalExceptionHandler tarafından yakalanır.
- DTO: Controller katmanında istekleri almak ve yanıtları döndürmek için kullanılır. Converter sınıfı, Entity ile DTO arasında dönüşüm yapar.
-

Bu yapı sayesinde, her katmanın belirli bir sorumluluğu vardır ve katmanlar arasındaki bağımlılıklar tek yönlüdür (üst katman alt katmanı kullanır). Bu, projenin modülerliğini, test edilebilirliğini ve bakım kolaylığını artırır.