



Ad-Soyad:Cemal Eren ekici Ekici

Numara: HR230033

Ders Adı: YAZILIM GELİŞTİRME ORTAM VE  
ARAÇLARI PROJE ÖDEVİ

# Yazılım Geliştirme Ortam Araçları Proje Dökümanı

## ▼ Proje Süreçleri ve Uygulanan Adımlar

Projemi, her aşamasını planlayarak ve adım adım gerçekleştirerek tamamladım. Aşağıda bu süreçleri açıklıyorum:

### 1. Başlangıç ve Planlama:

- Projeye başlamadan önce, gereksinimlerin tam olarak belirlenmesi için kapsamlı bir analiz yaptım. Proje gereksinimleri belirlendikten sonra, gerekli teknolojiler ve araçlar (Spring Boot, PostgreSQL, Docker, GitHub, JUnit, Postman, JMeter, GitHub Actions vb.) seçildi.
- Projeyi bir GitHub reposunda başlatarak sürüm kontrolünü sağladım.

### 2. Spring Boot Projesi Oluşturma:

- Proje için bir Spring Boot uygulaması oluşturdum. Başlangıçta, gerekli bağımlılıkları (Spring Web, JPA, PostgreSQL vb.) belirledim ve projeyi başlattım.
- Uygulama mimarisini belirleyip, controller, service, repository ve entity katmanlarını oluşturduktan sonra, tüm işlevsellikleri bu katmanlara ayırdım.

### 3. Servislerin Yazılması ve Test Edilmesi:

- İlk aşamada, projenin ana servislerini (çalışan oluşturma, görev kaydı.) yazdım. Bu servislerin doğru çalışıp çalışmadığını test etmek için Postman kullanarak manuel testler gerçekleştirdim.
- API'lerin beklenen şekilde çalıştığını doğruladıktan sonra, her servisi unit testlerle destekleyerek, test coverage raporları oluşturup servislerin kalitesini artırdım.

#### 4. **Sürekli Entegrasyon (CI) ve GitHub Actions:**

- Kod yazım ve test süreçlerinin otomatikleştirilmesi amacıyla GitHub Actions kullanarak sürekli entegrasyon (CI) sağladım. Her commit sonrasında testlerin çalıştırılmasını ve hataların zamanında fark edilmesini sağladım.
- CI pipeline'ını oluştururken, build ve test süreçlerinin başarıyla tamamlanması için gerekli yapılandırmaları yaptım.

#### 5. **Yük Testi:**

- Projenin son aşamasında, performans ve yük testleri yapmak için JMeter kullandım. JMeter ile API'leri belirli yükler altında test ederek, performansını ölçtüm ve uygulamanın ölçeklenebilirliğini değerlendirdim.

### **Yazılım Süreç Modeli Seçimi: Waterfall**

Projemde **Waterfall (Şelale)** yazılım süreç modelini tercih ettim. Sebeplerim ise şunlardı:

#### 1. **Kesin Gereksinimler ve Planlama:**

- Projeye başlarken gereksinimler tamamen netti ve değişiklikler beklenmiyordu. Waterfall modelinin bir özelliği olan, projede gereksinimlerin baştan sona kadar belirlenip netleştirilmesi, projenin her aşamasını daha belirgin ve sıralı hale getirdi.
- Her adımın tamamlanmasının ardından bir sonraki aşamaya geçilebileceği bir süreç izlemek, gereksinimlerimle uyumlu oldu.

#### 2. **Proje Sürecinin Düzgün İzlenebilirliği:**

- Waterfall modeli, aşamalar arasında net geçişler sağladığı için her aşamayı tamamladıktan sonra bir sonraki aşamaya geçmek için bir planım oldu. Bu da sürecin izlenebilirliğini artırarak, projenin tamamlanma süresini net bir şekilde öngörmeme yardımcı oldu.

### 3. Süreçlerin Sıralı İlerleyişi:

- Projenin ilk aşamasında tüm sistem gereksinimleri ve mimarisi belirlenip uygulama geliştirildikten sonra, sistem testlerine, entegrasyon testlerine ve son olarak yük testlerine geçtim. Her aşama bir önceki aşamanın sonucuna dayanarak daha sağlıklı bir ilerleyiş sağladı.

### 4. Test Süreci:

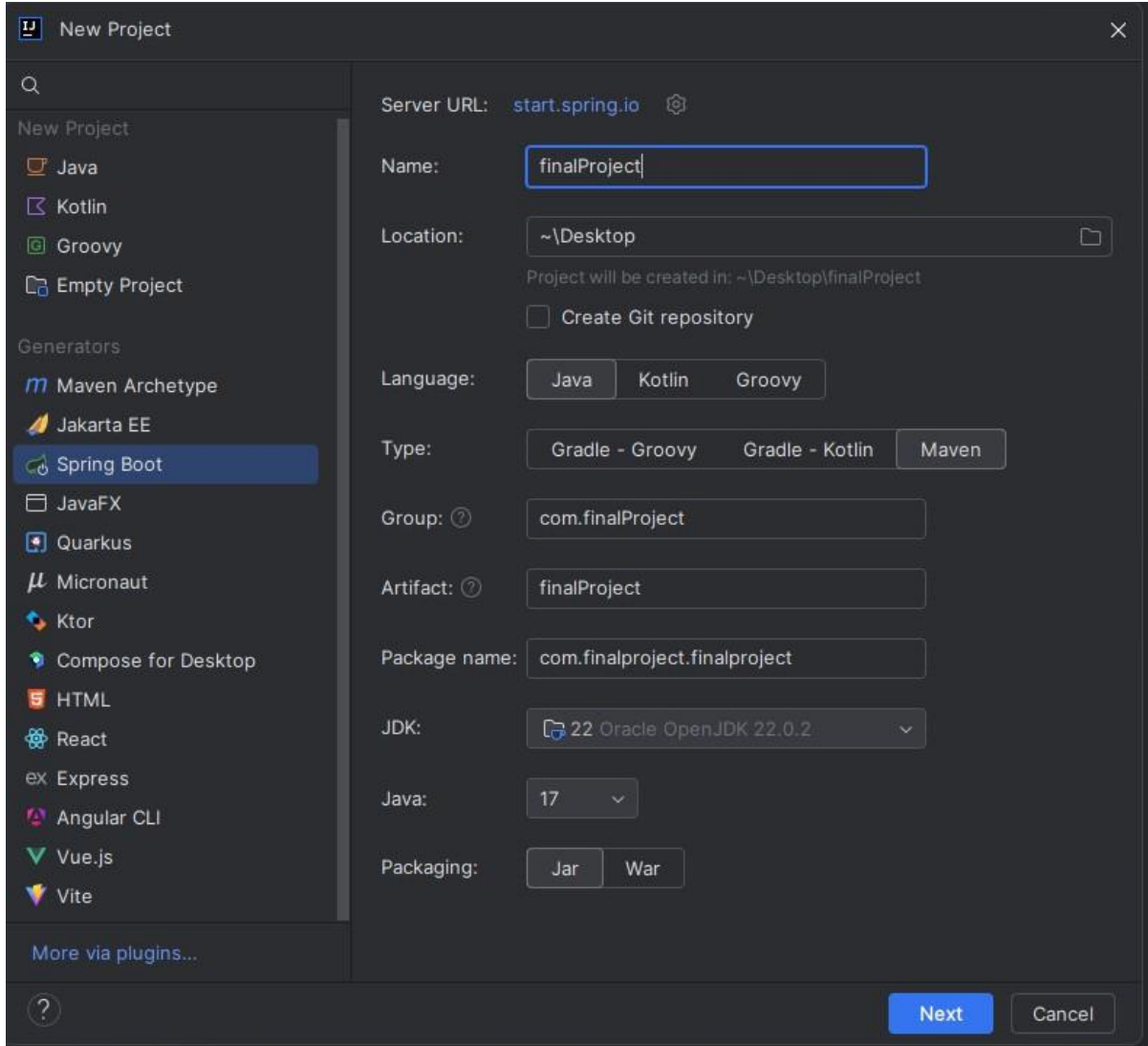
- Waterfall modelinde her aşama tamamlandıktan sonra test yapılması gerektiği için, yazılımımın her parçası için farklı test aşamaları oluşturduğumda, bu süreç oldukça verimli oldu.
- Unit testler ve yük testlerinin her biri belirli aşamalarda ve belirli kriterlerle gerçekleşti.

## ▼ Web Servis Geliştirmesi

Bu projeyi gerçekleştirirken kullandığım başlıca teknolojiler ; Java - Maven - Spring Boot, Docker - PostgreSQL ...

#### 1. İlk olarak bir Spring Boot projesi başlattım.

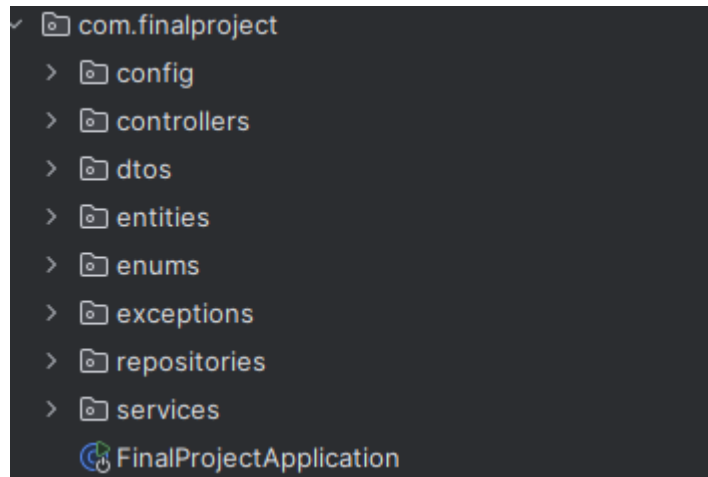
- a. Proje için uygun bir isim belirledim. Java 17 kullandım ve Maven yapı yönetim aracını seçtim. Projeyi JAR formatında paketlemek üzere yapılandırdım ve projeyi oluşturdum.



2. Daha sonrasında application.yaml dosyam da veri tabanı bağlantısı için gerekli verileri ekledim. Ve docker ile veritabanımı başlattım.

```
spring:
  application:
    name: final-project
  datasource:
    url: jdbc:postgresql://localhost:5432/final-project
    username: admin
    password: admin
    driver-class-name: org.postgresql.Driver
  jpa:
    database-platform: org.hibernate.dialect.PostgreSQLDialect
    hibernate:
      ddl-auto: update
server:
  port: 5000
  servlet:
    context-path: /api/v1
```

2. Projeyi gerçekleştirmek için Layered Architecture (Katmanlı Mimari) kullandım. Aşağıda bu mimariyi nasıl uyguladığımı ve projemdeki paketleri görebilirsiniz ;



3. Entitiy'leri oluşturmaya başladım.
- Çalışan ve Kadrolu Çalışan - Sözleşmeli çalışan olarak entitylermi oluşturdum. Burada Çalışan Sınıfını abstract bir sınıf olarak tanımlayıp

diğer iki sınıfı özelleştirdim.

- b. Görev ve Kalıcı Görev - Geçici Görev olmak üzere entitylerimi oluşturdum. Buarada Görev sınıfını abstract bir sınıf olarak tanımlayıp diğer iki sınıfı özelleştirdim.
- c. Daha sonrasında Çalışan Sınıfı ile Görev sınıfı arasında one to many ilişkisi kurdum.

```
@ManyToOne(fetch = FetchType.EAGER)
@JoinColumn(name = "employee_id", nullable = false)
private Employee employee;
```

- 4. DTO Sınıflarımı oluşturmaya başladım. Aşağıda örnek bir dto'nun göresi mevcuttur.
  - a. Dto controller katmanı ile service katmanı arasında veri taşımak için kullanılan nesnelerdir.
  - b. Dto sınıfımda validasyonları yapmak için "spring-boot-starter-validation" kütüphanesini kullandım ve gerekli bağımlılıkları pom.xml dosyasına ekledim.
  - c. Dto sınıfının değişkenleri private olduğu için getter ve setter fonksiyonlarını yazdım.

```

@NotBlank(message = "sicil no alanı zorunludur.") 3 usages
@Size(min = 8, max = 8, message = "sicil no alanı 8 haneli olmalıdır.")
private String sicilNo;

@NotBlank(message = "Ad alanı zorunludur.") 3 usages
@Size(min = 3, max = 20, message = "ad alanı 3 ile 20 harf arasında olmalıdır.")
private String ad;

@NotBlank(message = "Soyad alanı zorunludur.") 3 usages
@Size(min = 3, max = 20, message = "soyad alanı 3 ile 20 harf arasında olmalıdır.")
private String soyad;

@NotNull(message = "İşe giriş tarihi zorunludur.") 3 usages
@DateTimeFormat(pattern = "yyyy-MM-dd")
private LocalDate iseGirisTarihi;

@NotNull(message = "Sözleşme bitiş tarihi zorunludur.") 3 usages
@DateTimeFormat(pattern = "yyyy-MM-dd")
private LocalDate sozlesmeBitisTarihi;

@NotNull(message = "Aylık ücret alanı zorunludur.") 3 usages
@Positive(message = "Aylık ücret pozitif bir değer olmalıdır.")
private Float aylikUcret;

```

5. Controller Sınıflarımı oluşturmaya başladım. Burada Görevler (TasksController) ve Çalışanlar (EmployeeController) sınıflarını oluşturdum. Aşağıda bir post bir get örneğiyle spring botta controller katmanının nasıl çalıştığını anlatmaya çalıştım.
  - a. Burada controller sınıflarımızın başına @RestController anotasyonunu ekliyoruz. Spring Boot ile proje başlatıldığında bu anotasyon sayesinde EmployeeController'ın bir nesnesini üretip IoC container içerisine eklenmesini sağlıyoruz.
  - b. @RequestMapping anotasyonu, yazmış olduğumuz REST API'nin path kısmını ayarlıyoruz.
  - c. Constructor injection ile employeeService inject ediyoruz.
  - d. Çalışan eklemek için gerekli dto'yu alıp Service katmanına gönderiyoruz. Burada @PostMapping anotasyonu bu işlemin bir post işlemi olduğunu bildiriyoruz.
  - e. Service katmanından dönen veriyi ResponseEntity ile kullanıcıya dönüyoruz.



```

@RestController
@RequestMapping("/employees")
public class EmployeeController {

    private final EmployeeService employeeService; 4 usages
    @Autowired
    public EmployeeController (EmployeeService employeeService) { this.employeeService = employeeService; }

    @PostMapping("/permanent")
    public ResponseEntity<Employee> addPermanentEmployee(@Valid @RequestBody PermanentEmployeeDTO permanentEmployeeDTO) {
        Employee createdEmployee = employeeService.addPermanentEmployee(permanentEmployeeDTO);
        return new ResponseEntity<>(createdEmployee, HttpStatus.CREATED);
    }

    @PostMapping("/contract")
    public ResponseEntity<Employee> addContractEmployee(@Valid @RequestBody ContractEmployeeDTO contractEmployeeDTO) {
        Employee createdEmployee = employeeService.addContractEmployee(contractEmployeeDTO);
        return new ResponseEntity<>(createdEmployee, HttpStatus.CREATED);
    }

    @GetMapping("/")
    public List<Employee> listEmployees() {
        return employeeService.listEmployees();
    }
}

```

6. Service Sınıflarımı oluşturmaya başladım. Burada Görevler (TasksService) ve Çalışanlar (EmployeeService) sınıflarını oluşturdum.
  - a. Burada Service sınıflarımızın başına @Service anotasyonunu ekliyoruz. Spring Boot ile proje başlatıldığında bu anotasyon sayesinde EmployeeService'in bir nesnesini üretip IoC container içerisine eklmesini sağlıyoruz.
  - b. Constructor injection ile employeeRepository inject ediyoruz.
  - c. Öncelikle findEmployeeBySicilNo ile daha önce bu sicil no ile bir çalışan eklenmiş mi diye kontrol ediyoruz. Eğer eklenmişse hata fırlatıyoruz, eğer eklenmemişse dto'yu entiteye çeviriyoruz
  - d. Daha sonra bu entity'i veri tabanına kaydetmek için employeeRepository nesnesini kullanıyoruz.
  - e. Eğer kayıt başarılı olursa controller katmanına veriyi dönüyoruz.

```

@Service 11 usages
public class EmployeeService {

    private final EmployeeRepository employeeRepository; 6 usages

    public EmployeeService(EmployeeRepository employeeRepository) {
        this.employeeRepository = employeeRepository;
    }

    public Employee addPermanentEmployee(PermanentEmployeeDTO permanentEmployeeDTO) { 7 usages
        findEmployeeBySicilNo(permanentEmployeeDTO.getSicilNo());

        Employee employee = permanentEmployeeDTO.toEntity();
        employee.setIstenCikisTarihi(null);
        Employee savedEmployee = employeeRepository.save(employee);

        return savedEmployee;
    }
}

```

7. Repository sınıflarını oluşturmaya başladım. Burada Görevler (TaskRepository) ve Çalışanlar (EmployeeRepository) arayüzlerini (Interface) oluşturdum.
  - a. Repository katmanında ise veri tabanı ile haberleşmek için olan JPA'nın sağladığı temel CRUD işlemlerini kullandım.
  - b. Save, findAll, findById gibi metotları kullandım.
8. Son olarak Exceptions yönetmek için sınıflar tanımladım.
  - a. APIError bu sınıf hata mesajını belirli bir formatta kullanıcıya göstermek için tasarladım.
  - b. Global olarak tüm hataları yakalamak için GlobalExceptionHandler sınıfını tanımladım.
  - c. Spesifik errorları göstermek için SpecificExceptionHandler adında yeni bir sınıf tanımladım. Burada Dto ve Entity validasyon hatalarını yakalayıp, kullanıcıya gösterdim.

```

public class APIError extends RuntimeException {
    private final String message;
    private final int statusCode;

    public APIError(int statusCode, String message ) { 7 usages
        this.statusCode = statusCode;
        this.message = message;
    }
}

```

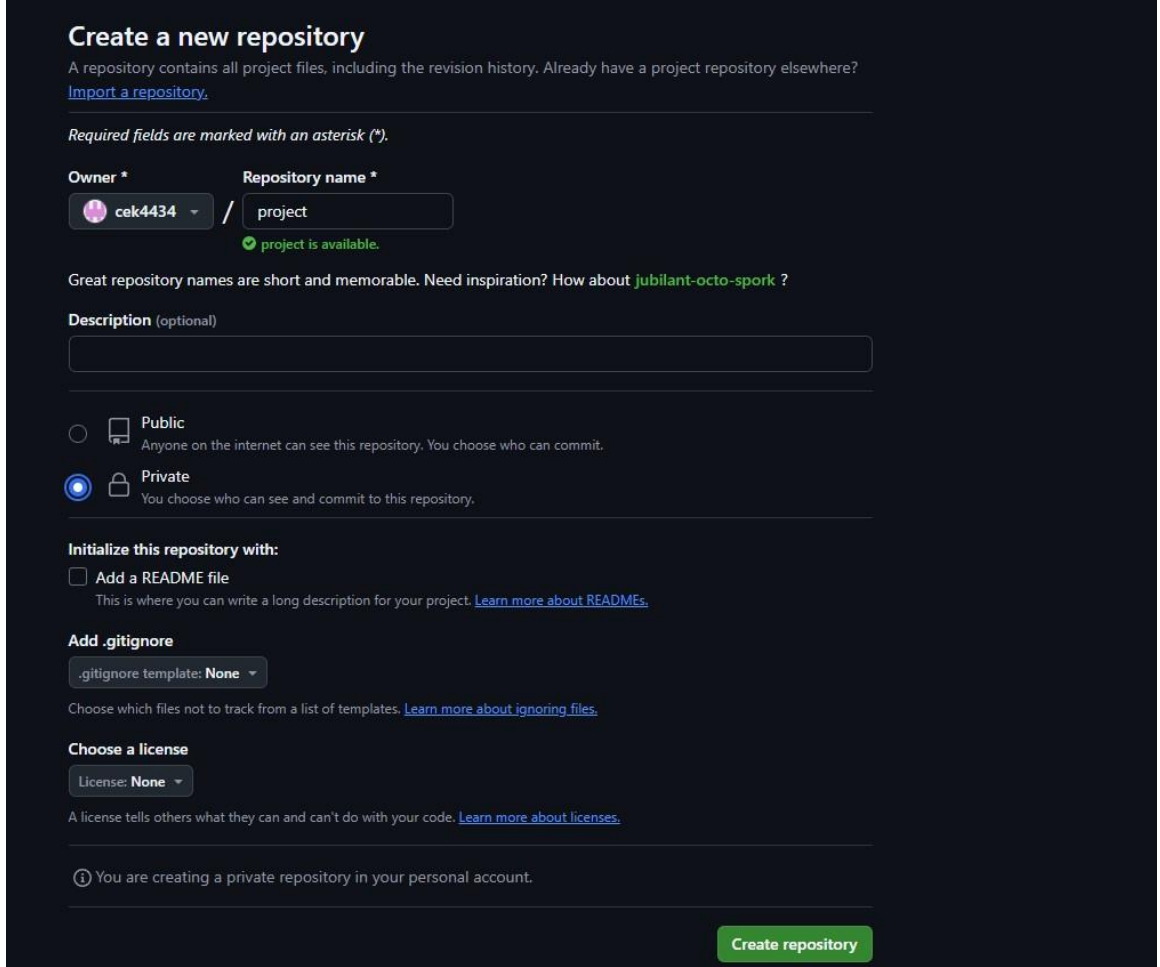
## Değerlendirme

- Mimari olarak **Layered Architecture** (Katmanlı Mimari) yaklaşımını benimsedim. Bu mimariyi, yazılımın daha yönetilebilir ve sürdürülebilir olmasını sağlamak için tercih ettim. Proje içinde, **Entity**, **DTO**, **Controller**, **Service**, ve **Repository** katmanlarını belirleyerek, her bir katmanın sorumluluklarını net bir şekilde ayırdım. Bu yapının uygulanması, projenin ilerleyen aşamalarında bakım ve geliştirme işlemlerini daha verimli hale getirdi.
- **Entity** sınıflarımı oluştururken, **Çalışan** ve **Görev** sınıflarını modelledim. **Çalışan** sınıfını abstract bir sınıf olarak tanımlayarak, **Kadrolu Çalışan** ve **Sözleşmeli Çalışan** sınıflarını özelleştirdim. Aynı şekilde, **Görev** sınıfını abstract olarak tanımlayıp, **Geçici Görev** ve **Kalıcı Görev** sınıflarını detaylandırıdım. Ayrıca, **Çalışan** ve **Görev** sınıfları arasında **One-to-Many** ilişkisi kurarak, veritabanındaki ilişkiyi doğru şekilde temsil ettim.
- **DTO** sınıflarını oluştururken, veri taşıma işlemleri için kullanılan nesneleri tasarladım ve bu sınıflarda gerekli **validation** kontrollerini sağlamak için **spring-boot-starter-validation** kütüphanesini entegre ettim. Böylece kullanıcıdan gelen verilerin doğruluğunu kontrol ederek, veritabanına geçişi sağlamadan önce hatalı verilerin engellenmesini sağladım.

- **Controller** sınıflarında, her iki ana işlevi gerçekleştirecek sınıfları (örneğin **EmployeeController** ve **TaskController**) oluşturup, **REST API**'lerini tasarladım. Burada **@RestController** ve **@RequestMapping** anotasyonları ile API endpoint'lerini tanımladım. Kullanıcıdan alınan verileri **Service** katmanına iletmek için **@PostMapping** ve **@GetMapping** gibi HTTP metodlarını kullandım. Ayrıca, **constructor injection** ile ilgili servisleri controller'a enjekte ederek **Spring Boot**'un IoC (Inversion of Control) mekanizmasını etkin bir şekilde kullandım.
- **Service** katmanında, iş mantığını ve veritabanı işlemlerini gerçekleştirdim. Çalışan kayıtlarını kontrol etmek, yeni çalışan eklemek, ve görev atamak gibi işlemleri burada yönettim. Ayrıca, **employeeRepository** ile veritabanına erişim sağladım ve **DTO** verilerini **Entity**'lere dönüştürerek veri tabanına kaydettim.
- **Repository** katmanında, **JPA** kullanarak veri tabanı ile haberleşmeyi sağladım ve temel CRUD (Create, Read, Update, Delete) işlemleri için gerekli metodları oluşturup veri tabanında işlem gerçekleştirdim.
- Son olarak, uygulamanın hata yönetimini sağlamak için **Exception Handling** mekanizmalarını kurdum. **GlobalExceptionHandler** sınıfıyla tüm hataları merkezi bir yerden yönettim, spesifik hatalar için ise **SpecificExceptionHandler** sınıfını oluşturarak DTO ve Entity validasyon hatalarını kullanıcıya doğru bir şekilde sundum. Bu sayede, hata mesajları kullanıcı dostu bir biçimde yönetilerek uygulamanın güvenilirliği arttı.
- Bu süreçte kullandığım **Docker** ile veritabanımın taşınabilirliğini sağladım ve çevresel farklardan bağımsız olarak projemin her ortamda aynı şekilde çalışmasını garanti ettim. **PostgreSQL** veritabanını Docker üzerinde çalıştırarak hızlı bir şekilde test ortamlarını oluşturabildim ve veri tabanımın doğru şekilde yapılandırıldığını kontrol ettim.
- Sonuç olarak, projeyi geliştirirken adım adım her katman için belirlediğim sorumluluklar ve teknolojilerle verimli ve sürdürülebilir bir uygulama geliştirmiş oldum. Proje, yazılım geliştirme süreçlerinde sıkça karşılaşılan sorunları minimize ederek, ileriye dönük geliştirmelere açık, güvenilir ve yönetilebilir bir altyapı sundu.

## ▼ Git - Github

- Öncellikle [github.com](https://github.com) adresine gidip giriş işlemi yaptım. Daha sonra kodlarımı ekleyeceğim repository açmak için new repository dedim.



- Burada repository'ni private olarak seçtim. Proje ismini verdim ve repository oluşturdum.
- Daha sonrasında kodlarımı bu repository atmak için git kullandım.
  1. "git init" diyerek projemde git bağlantısını oluşturdum.
  2. "git add ." komutuyla değişiklikleri git'e bildirdim.
  3. "git commit -m 'first commit' " ile ilk commitimi attım.

4. "git remote add origin https://github.com/cek4434/project.git" komutuyla git ve github arasında bağlantıyı kurdum.
5. "git push -u origin main" ile kodlarımı git repoma gönderdim.

```
commit 9bd2039c85a647d898098b0adcf31056b1f41af0 (HEAD → master, origin/master, origin/HEAD)
Author: cek4434 <cemal.ekici.eren@gmail.com>
Date: Tue Jan 14 14:44:06 2025 +0300

    task service finished

commit 0cf7b529f3cdd29235ebaad5e73f17ae95e1e9b5
Author: cek4434 <cemal.ekici.eren@gmail.com>
Date: Tue Jan 14 14:38:16 2025 +0300

    task controller and repository created

commit e6aafd25cc64b379924cba14c66803694d7f6302
Author: cek4434 <cemal.ekici.eren@gmail.com>
Date: Tue Jan 14 14:36:35 2025 +0300

    task entities and task dtos created

commit b1ccb5cdb4effcea439e1a235f28ca62c1c0e1b
Author: cek4434 <cemal.ekici.eren@gmail.com>
Date: Tue Jan 14 13:45:09 2025 +0300

    employee services finished

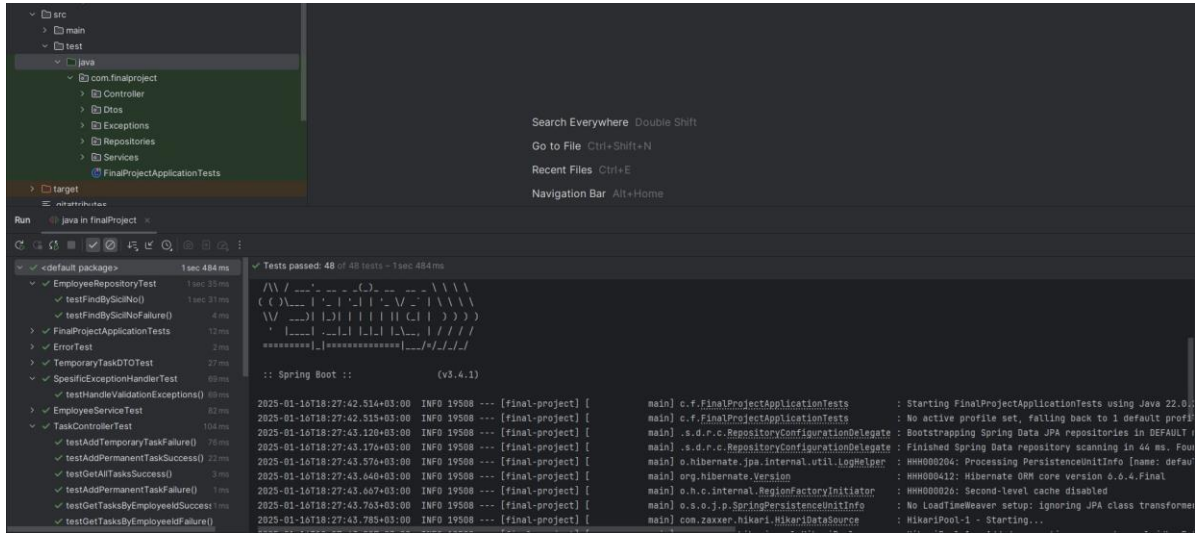
commit c9c7262c49ee305fc3c4a1359ef9cab046d30fc5
Author: cek4434 <cemal.ekici.eren@gmail.com>
Date: Tue Jan 14 13:43:53 2025 +0300

    employee dtos created
:|
```

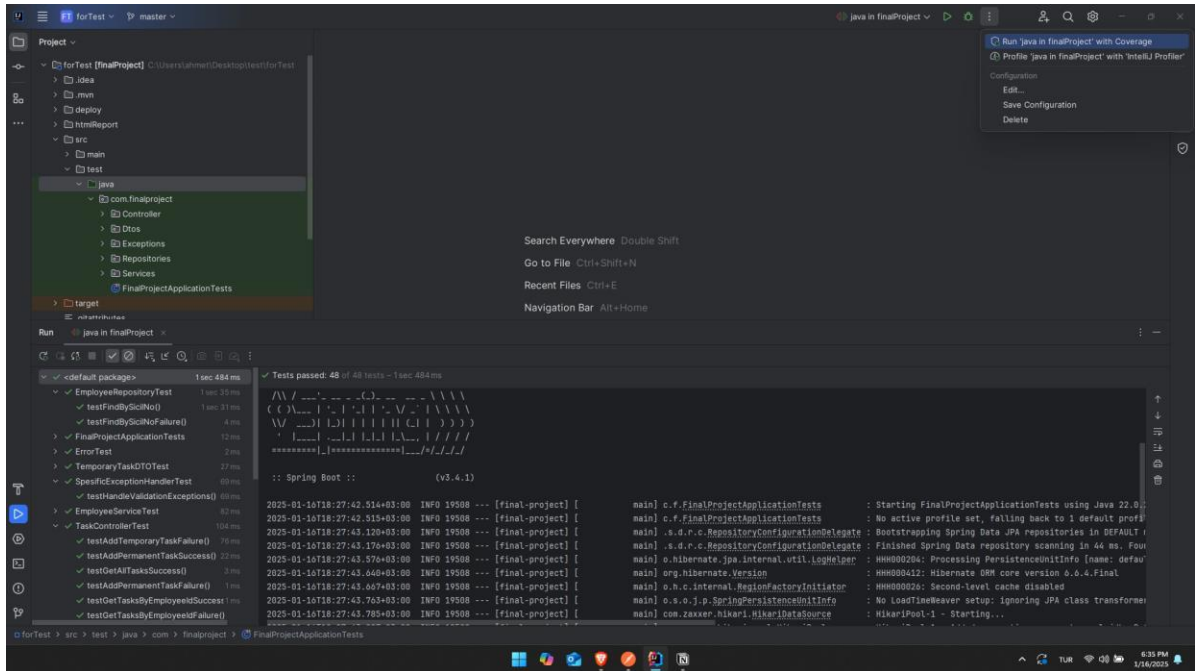
## ▼ Birim Testler - Test Coverage

- JUnit kullanarak projenin unit testlerini yazdım.
- Test coverage hesaplamak için IntelliJ IDEA içindeki yerleşik özellikleri kullandım. GitHub Actions kısmında ise test coverage hesaplamak için Jacoco kullanmaya karar verdim. GitHub Actions üzerindeki bu süreci daha detaylı olarak raporumda açıkladım.

- Yazmış olduğum tüm testleri çalıştırmak için, test klasörü altındaki java klasörüne sağ tıklayıp "Run" seçeneğini tıklayarak tüm testleri çalıştırdım.



- Yazdığımız birim testlerin coverage'ını hesaplamak için, aşağıdaki fotoğrafta görüldüğü üzere üç nokta simgesine tıklayıp test coverage'ı hesaplattım.



- Daha sonra açılan pencerede test coverage sonuçlarını detaylı bir şekilde inceledim.

Coverage java in finalProject				
Element	Class, %	Method, %	Line, %	Branch, % ^
com.finalproject	90% (20/22)	71% (78/109)	82% (153/186)	100% (10/10)
> entities	83% (5/6)	70% (24/34)	70% (24/34)	100% (0/0)
FinalProjectApplication	0% (0/1)	0% (0/1)	0% (0/1)	100% (0/0)
> dtos	100% (4/4)	54% (23/42)	71% (48/67)	100% (0/0)
> controllers	100% (2/2)	100% (9/9)	100% (13/13)	100% (0/0)
> config	100% (1/1)	100% (1/1)	100% (3/3)	100% (0/0)
> repositories	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
> enums	100% (2/2)	100% (4/4)	100% (6/6)	100% (0/0)
> exceptions	100% (4/4)	85% (6/7)	90% (20/22)	100% (2/2)
> services	100% (2/2)	100% (11/11)	97% (39/40)	100% (8/8)

- Bu raporu html olarak yazdırdım.

Current scope: all classes				
Overall Coverage Summary				
Package	Class, %	Method, %	Branch, %	Line, %
all classes	100% (22/22)	74.8% (92/123)	100% (10/10)	83.3% (147/200)
Coverage Breakdown				
Package	Class, %	Method, %	Branch, %	Line, %
com.finalproject	100% (1/1)	50% (1/2)	50% (1/2)	50% (1/2)
com.finalproject.config	100% (1/1)	100% (2/2)	100% (4/4)	100% (4/4)
com.finalproject.controllers	100% (2/2)	100% (9/9)	100% (13/13)	100% (13/13)
com.finalproject.dtos	100% (4/4)	58.7% (23/40)	73.2% (52/71)	73.2% (52/71)
com.finalproject.entities	100% (6/6)	79% (30/40)	79% (30/40)	79% (30/40)
com.finalproject.enums	100% (2/2)	100% (4/4)	100% (6/6)	100% (6/6)
com.finalproject.exceptions	100% (4/4)	88.9% (8/9)	100% (2/2)	91.7% (22/24)
com.finalproject.services	100% (2/2)	100% (11/11)	100% (8/8)	97.5% (39/40)

generated on 2023-01-16 21:06

## Değerlendirme

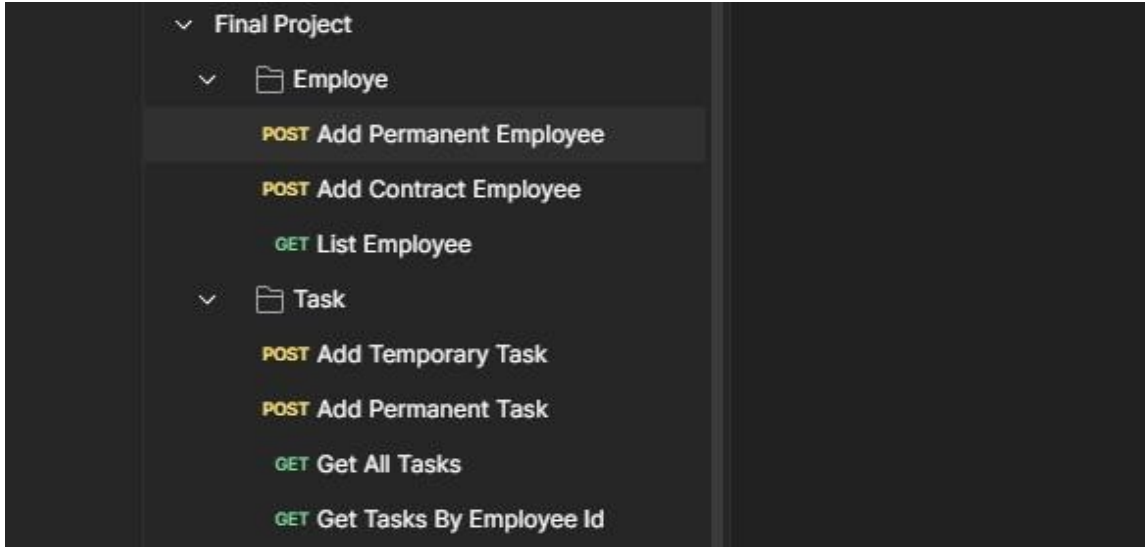


- Projemde, yazılımın doğruluğunu ve güvenilirliğini sağlamak amacıyla **JUnit** ile birim testlerimi yazdım. JUnit, Java uygulamalarında test yazma sürecini oldukça kolaylaştıran güçlü bir araçtır ve bu sayede her bir işlevin doğru çalışıp çalışmadığını hızlı bir şekilde test etme imkanı buldum. Yazdığım birim testleri ile uygulamamın temel fonksiyonlarını kapsamlı bir şekilde kontrol ettim ve potansiyel hataları erken aşamalarda tespit ettim.
- Test yazma sürecinin yanı sıra, **test coverage** hesaplamalarını yaparak, testlerimin ne kadarını uygulamanın kodunun kapsadığını analiz ettim. Test coverage, yazdığım testlerin ne kadar etkili olduğunu, hangi kod bölümlerinin test edilmediğini ve iyileştirilmesi gereken alanları belirlememi sağladı. Bu sayede yazılımın kalitesini arttırmak ve test kapsamını genişletmek için önemli veriler elde ettim.
- IntelliJ IDEA'nın sunduğu yerleşik özelliklerle test coverage'ı hesapladıktan sonra, testlerin başarı oranını ve kapsadığı kod oranını görsel olarak değerlendirdim. Bu araç, hataların ve eksik testlerin hızlı bir şekilde fark edilmesine yardımcı oldu. Sonrasında, **GitHub Actions** üzerinde Jacoco kullanarak CI sürecine entegre edilen test coverage raporlarını inceleyerek, her yeni commit sonrasında yazılımın kalitesini anlık olarak izledim.

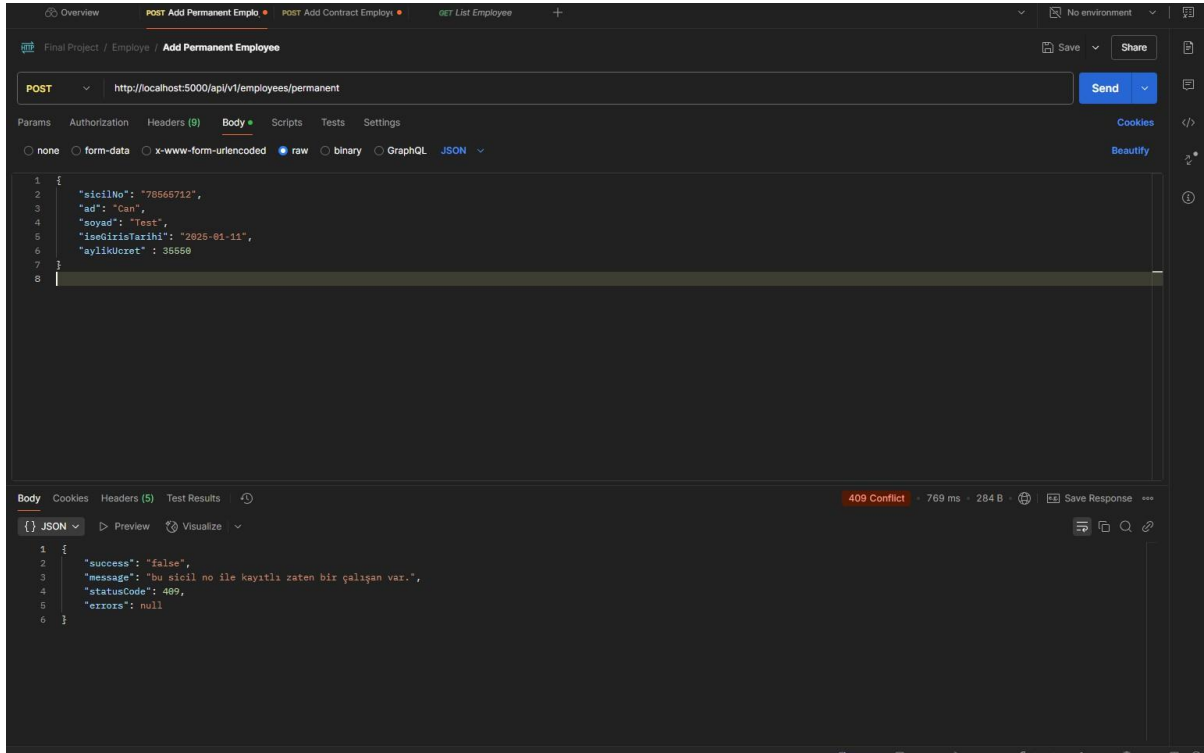
## ▼ Postman & JMeter Testleri

### Postman Testleri

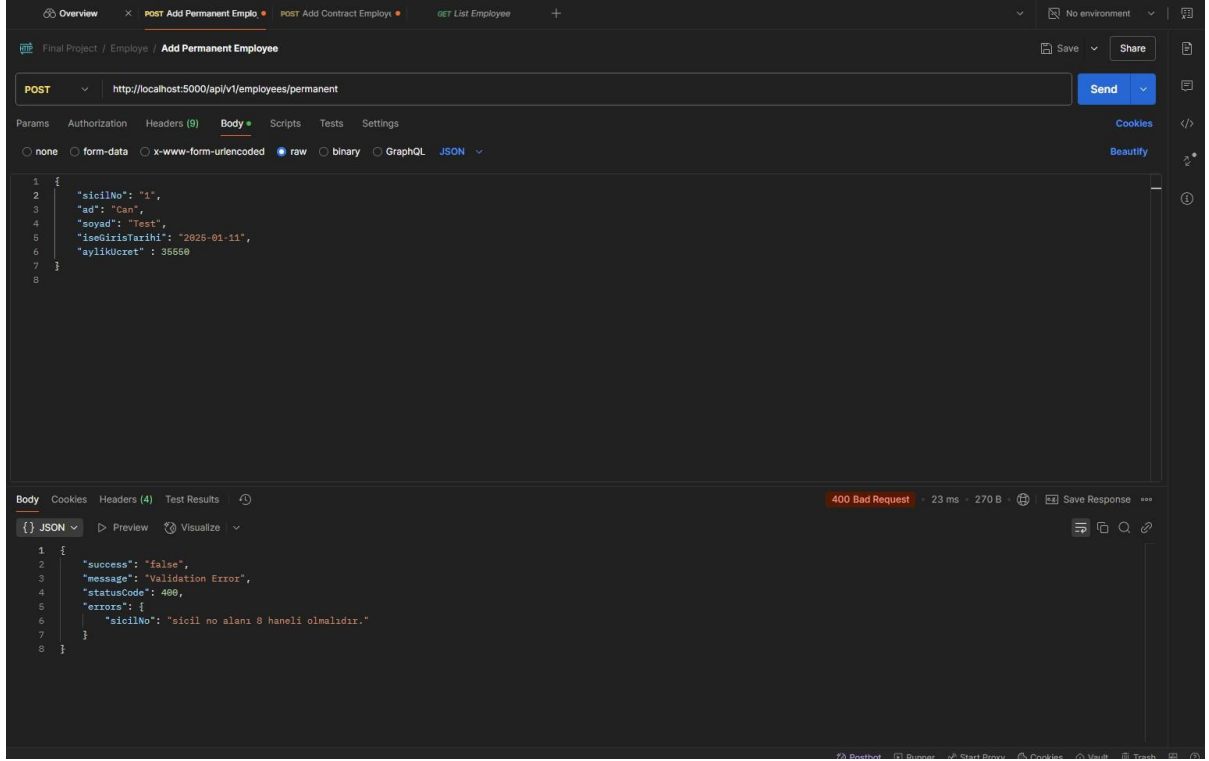
- Öncelikle postman'da yeni bir collection oluşturup, gerekli klasör ve endpointleri ekledim.



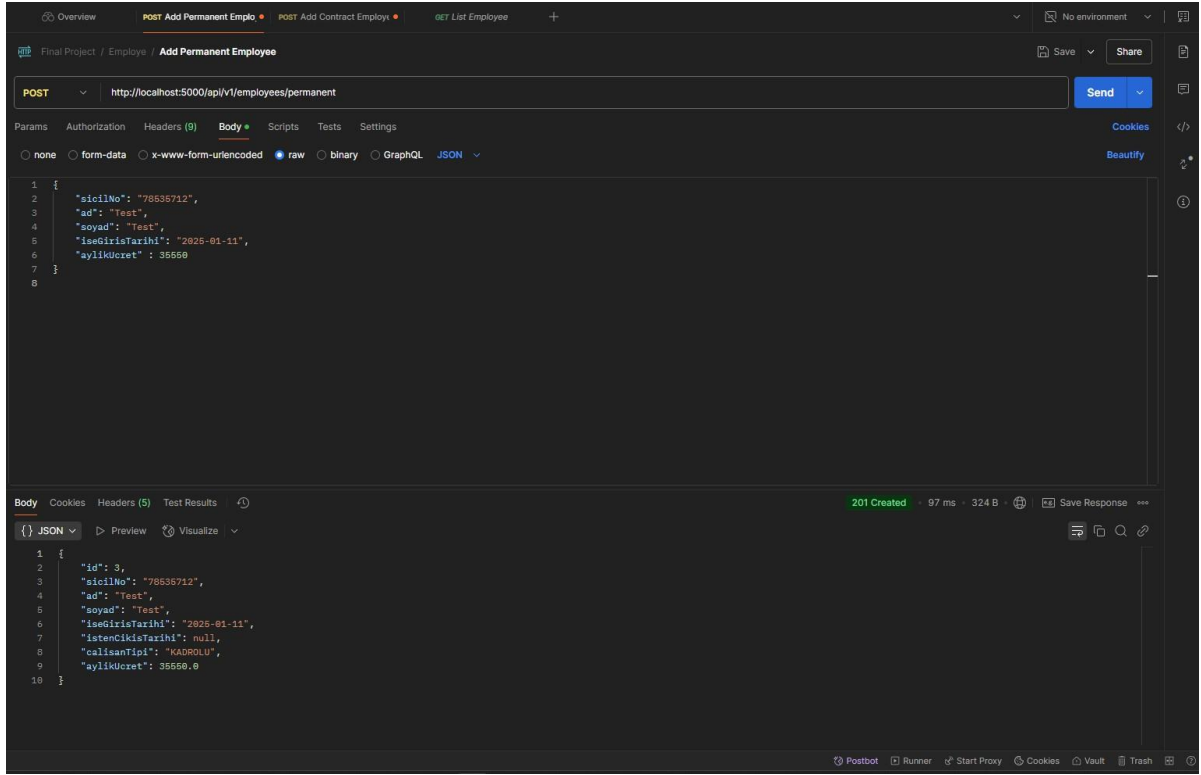
- Postman ile yaptığım bazı testlerin sonuçlarını aşağıdaki örneklerde göstermeye ve anlatmaya çalıştım.
- Bu örnekte bir employee eklemeye çalıştığımda aldığım hatayı göstermek istedim. Burada bu çalışan daha önce oluşturulduğu için hata aldım.



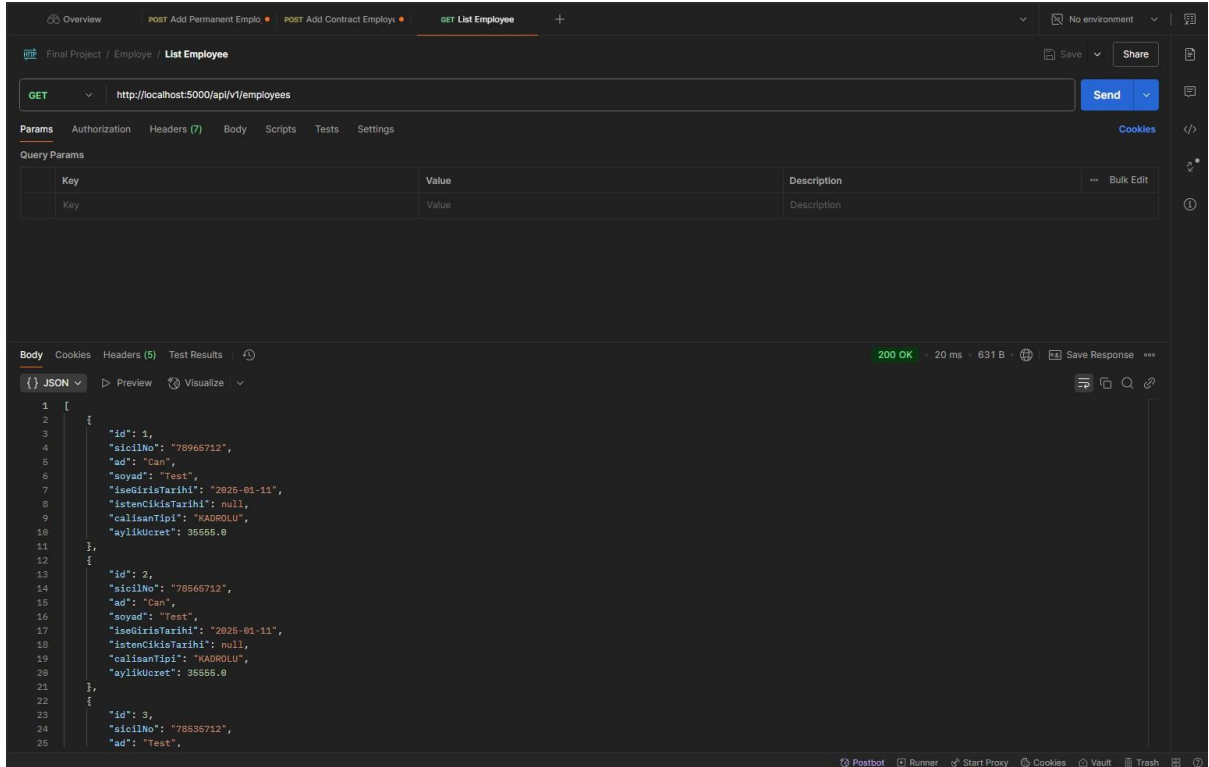
- Bu örnekte bir validasyon hatasını göstermek istedim. Burada sicil no 8 haneli olmak zorunda olduğu için hata aldık.



- Bu örnekte başarılı bir şekilde çalışan eklemeyi gösterdim.

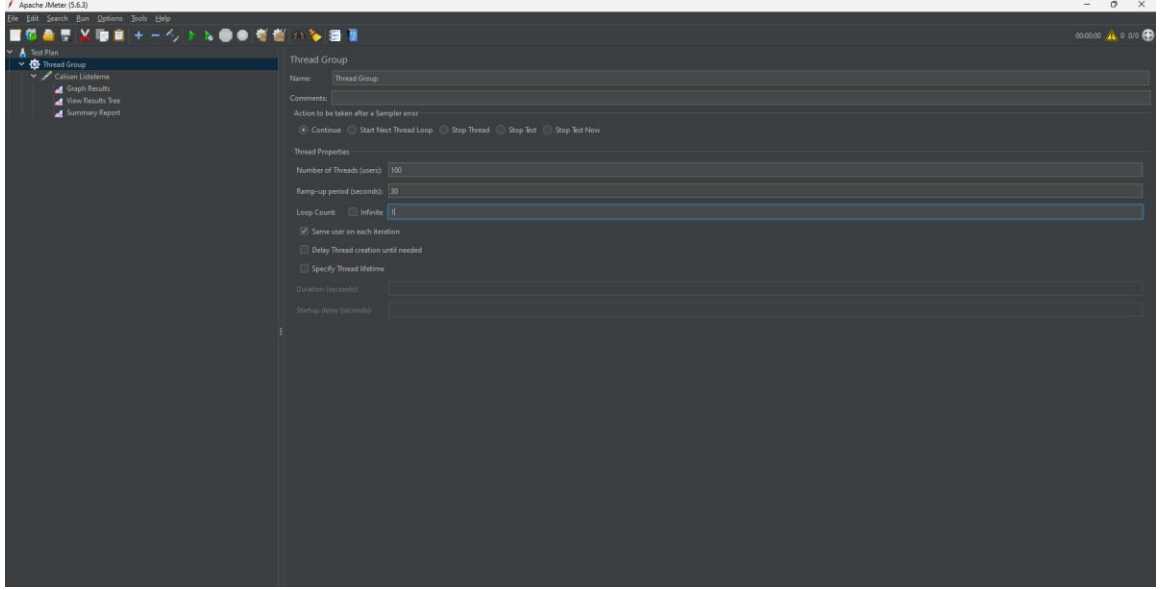


- Bu örnekte çalışanların tümünü listelemeyi göstermek istedim.

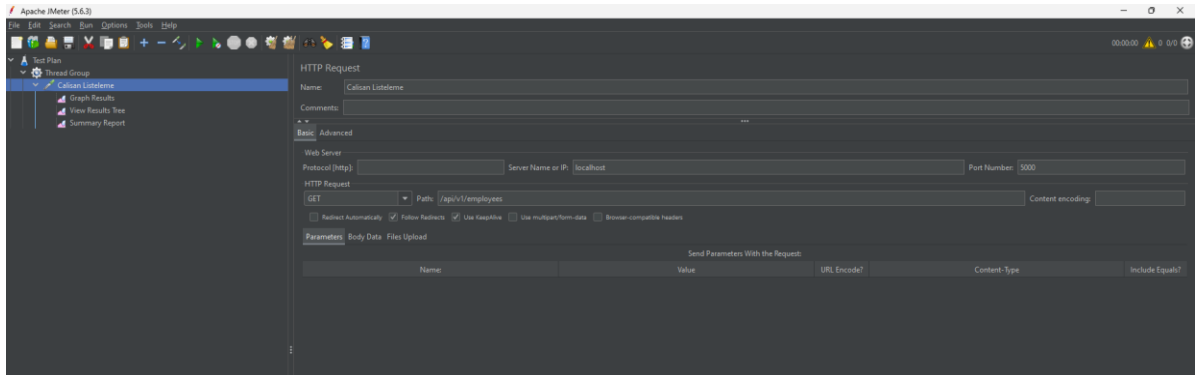


## JMeter Testleri

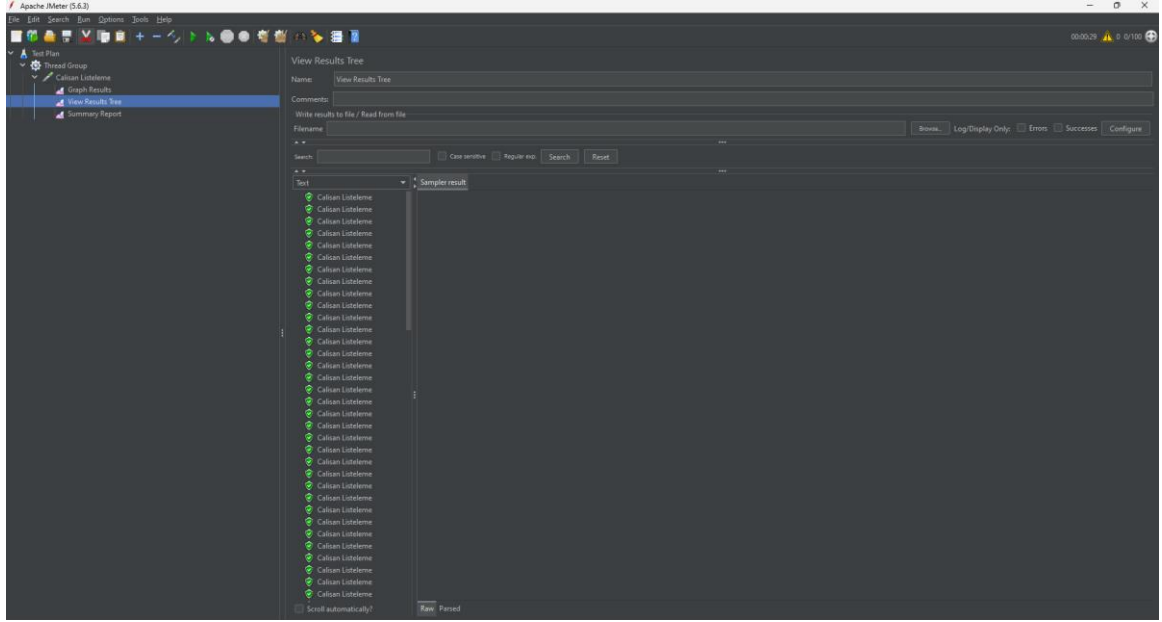
- Öncellikle bir 'Thread Group' oluşturun. Burada "Number of Threads" 100 ve "Ramp-up period" 30 olarak ayarlıyorum.
- "Number of Threads" değerini olarak 100 belirliyorum, bu aynı anda 100 kullanıcıyı simüle etmeye yarıyor. "Ramp-up period" süresini 30 saniye ayarlamam ise JMeter'in bu 100 kullanıcıyı sisteme yavaşça, 30 saniye boyunca eşit aralıklarla ekleyeceğini belirtiyor. Bu, ani yük yerine kontrollü bir yük testi yapmamıza olanak tanıyor.



- Daha sonrasında "Thread Group" → "Add" → "Sampler" \* HTTP Request oluştuyorum.
- Server Name or IP: Burada localhost kullanarak, testlerimi lokal geliştirme ortamında çalıştırıyorum.
- Port Number: API'm 5000 numaralı port üzerinde çalıştığı için doğru hedefe erişimi bu şekilde sağlıyorum.
- Path: API'nin "/api/v1/employees" gibi bir endpoint'ini test ederek bu endpoint'in tüm çalışanları listeleme amacına hizmet ettiğini belirtiyorum.
- HTTP Method: "GET" metodunu kullanarak yalnızca veri çekme operasyonunu test ediyorum.



- Burda JMeter ile yaptığımız testlerin bazı sonuçlarını almak için Graph Results, View Results Tree ve Summary Report kısımlarını ekliyorum.
- Daha sonrasında yukarıdaki “yeşil ok” simgesiyle testi çalıştırıyorum.



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Calisan Listeleme	100	29	6	506	66.68	0.00%	3.4/sec	5.33	0.43	1617.6
TOTAL	100	29	6	506	66.68	0.00%	3.4/sec	5.33	0.43	1617.6

- **Graph Results:** Performans sonuçlarını grafiksel olarak görselleştirir. Burada throughput (işlem hacmi), response time (cevap süresi) ve error rate (hata oranı) gibi metrikleri gözlemleyebiliriz.
- **View Results Tree:** Her bir isteğin detaylı sonuçlarını görüntüler. Bu kısımda, test esnasında gönderilen ve alınan HTTP başlıklarını ve gövdelerini kontrol edebiliriz.

- **Summary Report:** Tüm testlerin genel bir özetini verir. Ortalama cevap süresi, en uzun ve en kısa cevap süresi gibi önemli bilgiler buradan alınabilir.

## Değerlendirme

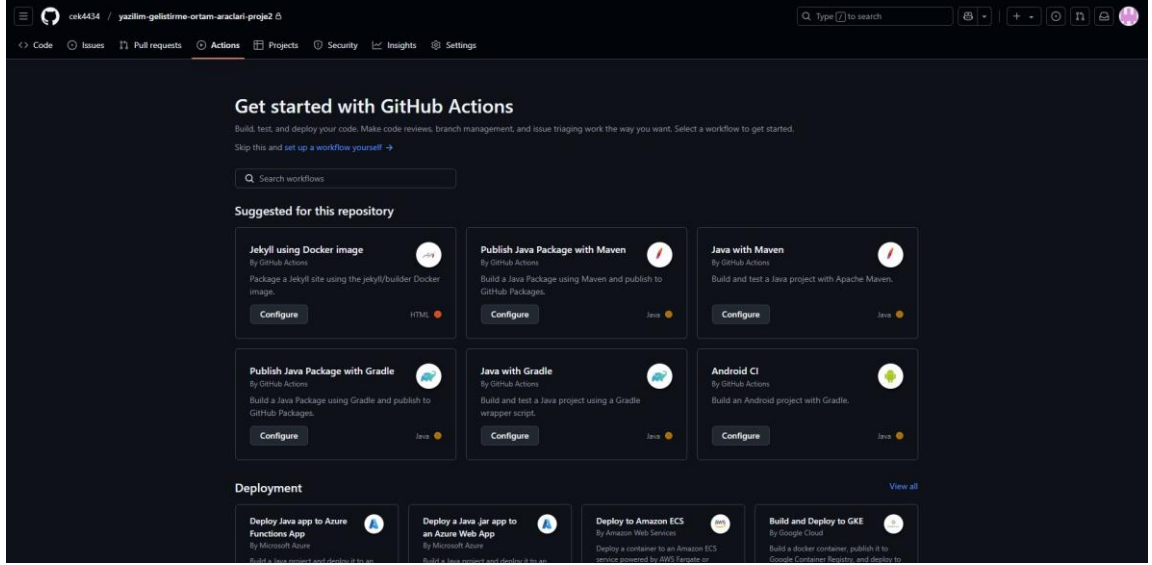
- Projemin doğru çalıştığını ve işlevselliğini test etmek için **Postman**'ı kullandım. Bu araç, API'nin çeşitli özelliklerini test etme ve sonuçları görselleştirme konusunda oldukça faydalı oldu
- İlk olarak, API'nin temel işlevlerinin doğru çalışıp çalışmadığını test ettim. Bu adımda, çalışan oluşturma,görev atama gibi temel CRUD işlemlerini gerçekleştirdim. Her API endpoint'inin doğru yanıtları döndürüp döndürmediğini kontrol ettim. Yanıt kodlarını ve içeriklerini inceledim. API'nin **200 OK** ve **201 Created** gibi doğru HTTP statü kodlarını döndürdüğünden emin oldum. Ayrıca, API'nin hata durumları için de doğru mesajlar ve statü kodları sunduğunu doğruladım.
- Projenin performansını analiz etme için JMeter kullandım.
- API, daha fazla istek geldiğinde ne kadar hızlı yanıt veriyor? Yük arttıkça yanıt sürelerinde bir artış oldu mu? Bu soruları cevaplamak için, belirli bir süre içinde kaç istek yapılabildiğini ve yanıt sürelerini ölçtüm.
- Kullanıcı sayısı arttıkça, hata oranının yükselip yükselmediğini kontrol ettim.
- API'nin kaynak kullanımını (CPU, bellek, vb.) ölçmek ve bu ölçümlerle daha yüksek yüklerde nasıl davrandığını analiz ettim.

## ▼ Sürekli Entegrasyon (Github Actions)

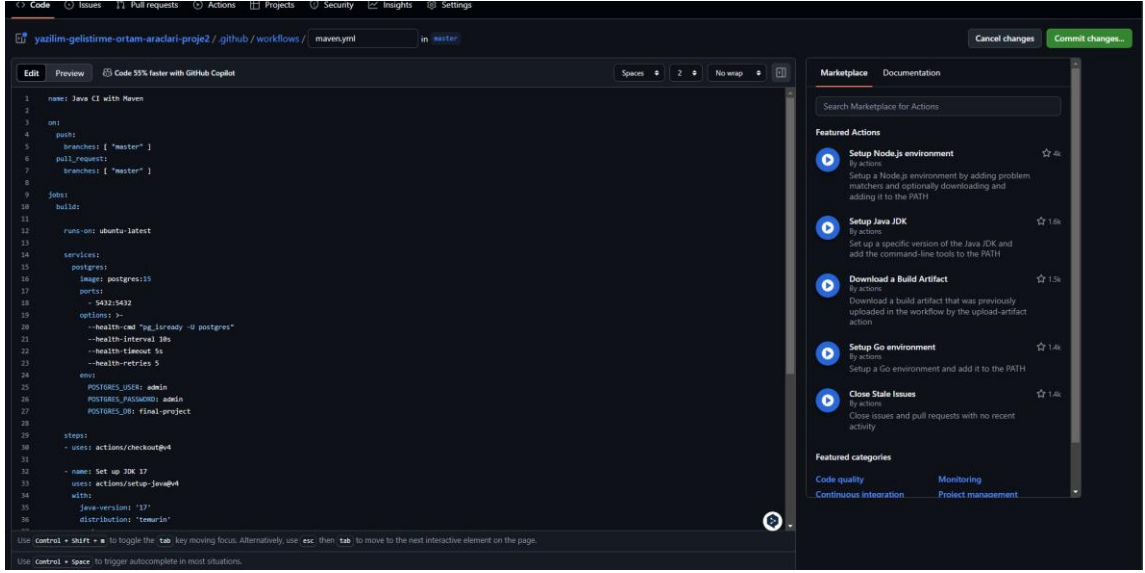
Projemde Github Actions ile CI sürecini nasıl oluşturduğumu aşağıda adım adım açıklıyorum.



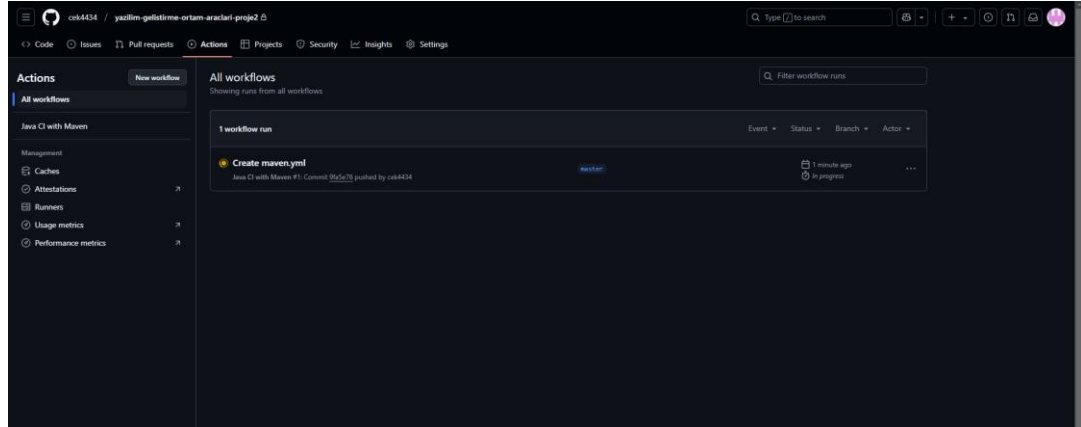
1. Öncellikle github repository'ime gidip "Actions" kısmına tıkladım. Burada "Java with Maven" seçeneğini seçtim. (Projem bir Maven projesi olduğu için bunu seçtim.)

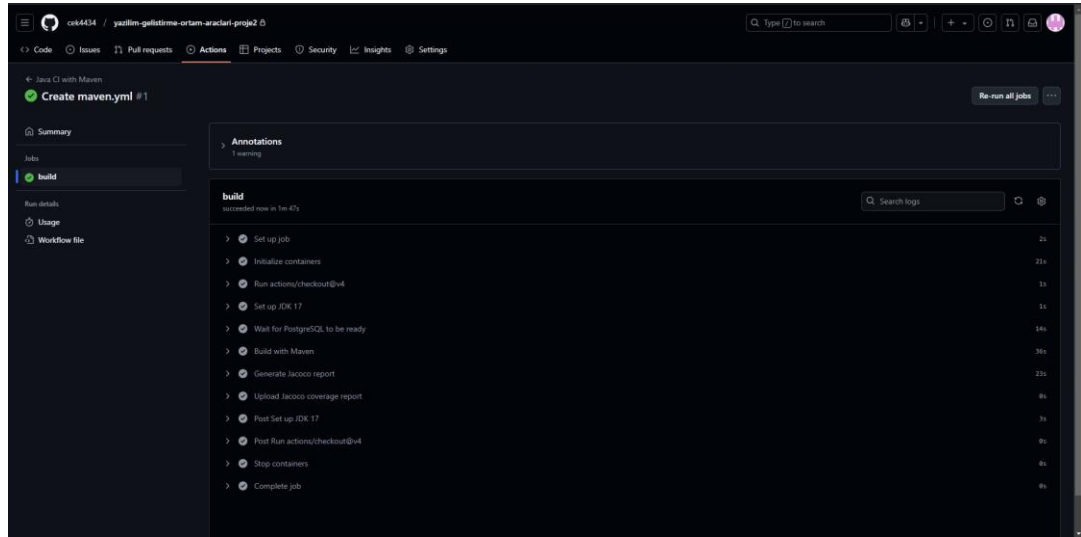
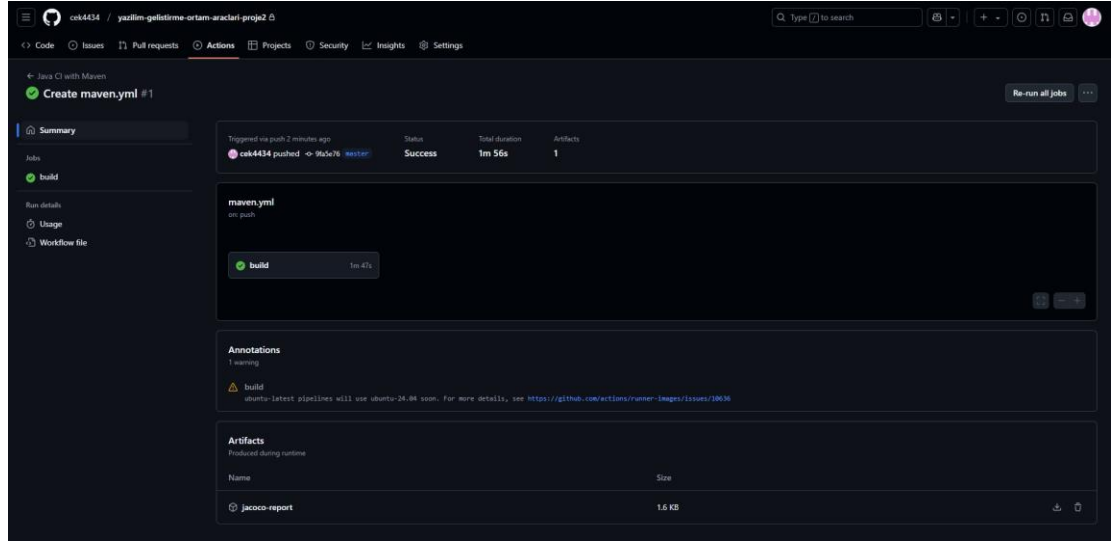


2. Daha sonra aşağıdaki gibi bir workflows oluşturdum.
  - a. Uygulamamda PostgreSQL veritabanını kullandığım için service kısmında gerekli bağımlılıkları ekledim. Postgresql Docker ile çalıştırdığım için port ayarı, veritabanı gibi isimleri aşağıda belirttim.
  - b. Bunun dışında yazmış olduğum unit testleri çalıştırmak ve test coverage hesaplamak için Jacoco bağımlılıklarını ekledim.
  - c. Daha sonrasında "commit changes" diyerek commit atma işlemini gerçekleştirdim.



3. Workflow oluřtu ve alıřmaya bařladı. Ařağıda bu workflow'un detaylarını "build success" mesajlarını gsterdim.





4. Ayrıca başarıyla tamamlanan workflow altında Jacoco ile hesaplanan test sonucunu indirip görüntüledim.

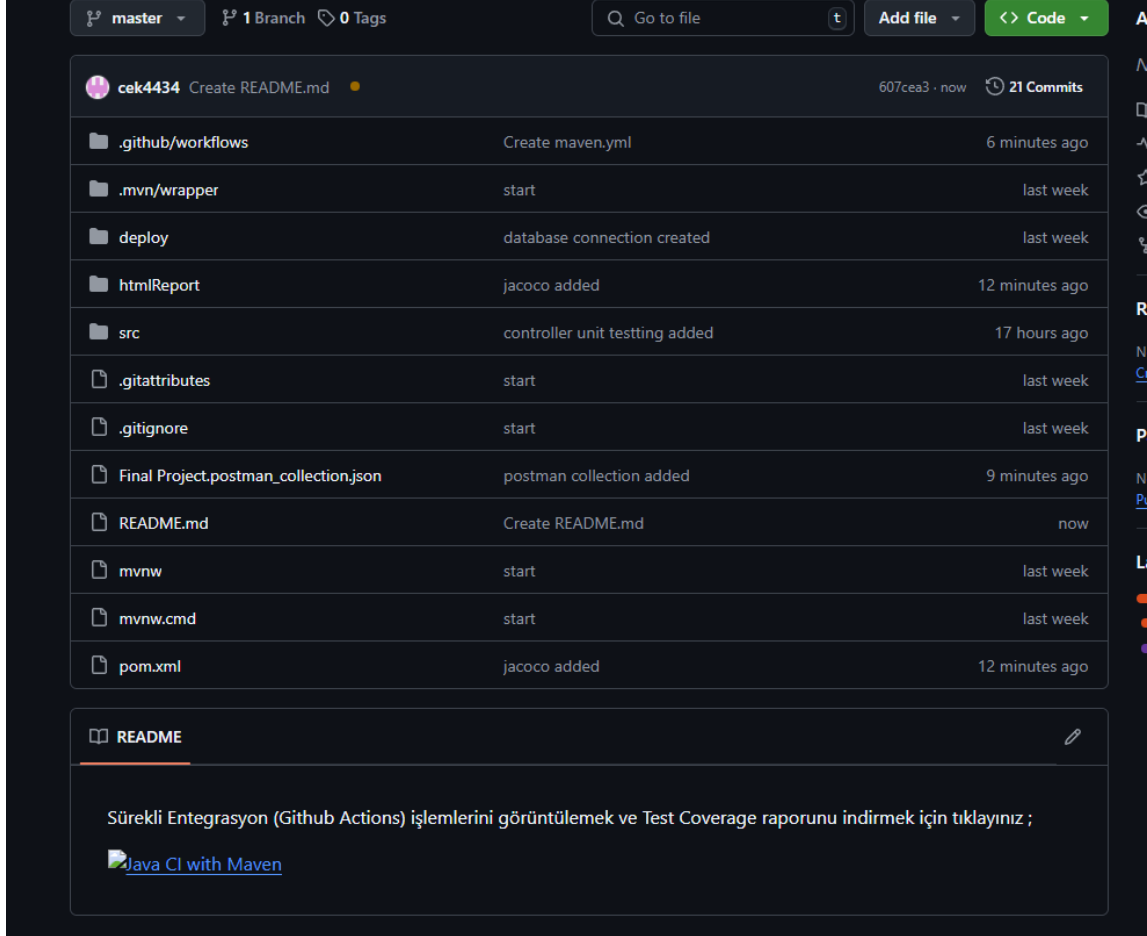
finalProject

finalProject

Element	Missed Instructions	Cov	Missed Branches	Cov	Missed	Cov	Missed	Lines	Missed	Methods	Missed	Classes
com.finalproject.dto	77%		n/a	19	46	19	90	19	46	0	4	
com.finalproject.entities	75%		n/a	10	40	14	57	10	40	0	6	
com.finalproject.exceptions	96%		100%	1	10	2	28	1	9	0	4	
com.finalproject.services	96%		100%	0	16	1	42	0	12	0	2	
com.finalproject	37%		n/a	1	2	2	3	1	2	0	1	
com.finalproject.controllers	100%		n/a	0	9	0	15	0	9	0	2	
com.finalproject.enums	100%		n/a	0	2	0	6	0	2	0	2	
com.finalproject.config	100%		n/a	0	2	0	4	0	2	0	1	
Total	112 of 776	85%	0 of 10	100%	31	127	38	245	31	122	0	22

Created with JaCoCo 0.8.12.202003131800

5. Daha sonrasında bir "Readme.md" oluşturup pull request adımını gösterdim. Bu sayede uzak repodaki yaptığım değişiklikleri local repo'ma çektim. Ayrıca workflow'un nasıl tetiklendiğini göstermeye çalıştım.



```
Desktop\project>git pull origin master
```

```
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 8 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (8/8), 2.64 KiB | 300.00 KiB/s, done.
From https://github.com/cek4434/yazilim-gelistirme-ortam-araclari-proje2
* branch          master      -> FETCH_HEAD
  ab65e99..607cea3  master      -> origin/master
Updating ab65e99..607cea3
Fast-forward
 .github/workflows/maven.yml | 56 ++++++++++++++++++++++++++++++++++++++
 README.md                   |  3 +++
 2 files changed, 59 insertions(+)
 create mode 100644 .github/workflows/maven.yml
 create mode 100644 README.md
```

## Değerlendirme

- **Otomatik Test ve Build Süreci:**

GitHub Actions, her commit sonrası otomatik olarak Maven ile projeyi derleyip test etmeye olanak tanıdı. Bu süreç sayesinde manuel müdahale gereksinimi ortadan kalktı ve her yeni değişiklik ile birlikte uygulamanın düzgün çalışıp çalışmadığını anında kontrol etme fırsatı buldum. Bu, yazılım geliştirme sürecini hızlandırarak, hataların erken aşamalarda tespit edilmesini sağladı.

- **PostgreSQL Bağımsız Test Ortamı:**

PostgreSQL veritabanı, Docker container'ı üzerinden çalıştırıldı. Bu sayede, veritabanı yapılandırmasını her geliştirme ortamında tekrarlamama gerek kalmadı ve aynı ortamda sürekli entegrasyon testleri yapılabilirdi. Docker kullanımı, veritabanı ve uygulama arasındaki bağımsızlığı artırarak, testlerimin izolasyonunu sağladı.

- **Test Coverage ve Kalite İzleme:**

Jacoco kullanarak, uygulamanın test kapsamını izleme ve raporlama sürecini otomatikleştirdim. GitHub Actions üzerinden yapılan testler sonucunda otomatik olarak oluşturulan coverage raporları sayesinde, yazılım kalitesini sürekli izleyebilmekteyim. Bu, testlerin kapsamını artırarak uygulamanın güvenilirliğini sağlamama yardımcı oldu.

- **Hata Tespiti ve Hızlı Geri Bildirim:**

Workflow'un her aşamasında meydana gelen hatalar, anında GitHub

Actions arayüzü üzerinden tespit edilebiliyor. Hata mesajları ve loglar, sorunların kaynağını hızlıca bulmamı sağladı ve çözüm sürecini hızlandırdı. Bu da sürekli entegrasyon sürecini daha verimli hale getirdi.

- **Yapılandırma ve İzlenebilirlik:**

CI süreci, GitHub repository'sine entegre edilerek her değişiklikte birlikte sistemin nasıl çalıştığını ve hangi testlerin geçtiğini takip edebilme imkanı sağladı. GitHub Actions arayüzü üzerinden yapılan her değişiklik sonrası workflow sonuçları ve test raporlarına kolayca erişebildim. Bu, projemin ilerlemesini ve kod kalitesini her aşamada izlememi kolaylaştırdı.