1 – Aşağıdaki belirtilen sayfada ilgili seçimler yapılmıştır.

Ekteki seçim;

Dışhat

Gelen

Uçuş Tarihi : 12 Haziran 2021

Uçuş Numarası : TK 0111

Yolcu adedi : Yetişkin -> 1 Çocuk - > 1 Bebek -> 0

Seçimleri yapılan hizmetin Toplam ücreti : 103.50 € + KDV dir.

Seçimlerde değişiklik yapıldıktan sonra Toplam ücretin değişmesi beklenmektedir ve ekranda anlık değiştirilmesi istenmektedir. Bu değişikliği sayfayı refresh etmeden nasıl ekranda güncelleriz ?

Java backend tarafında sayfayı yenilemeden ekranı güncellemek için genellikle WebSockets teknolojisi kullanılır.

WebSockets gerçek zamanlı iletişim için kullanılan bir protokoldür. İnternet tarayıcıları ve sunucular arasında sürekli ve etkileşimli iletişim kurmayı sağlar. WebSocket, HTTP protokolünü temel alırken, standart HTTP bağlantılarından farklı olarak kalıcı bir bağlantı sağlar.

HTTP protokolü, bir istek gönderildiğinde sunucu tarafından bir yanıt döndürür ve bağlantıyı sonlandırır. Bu durumda, sunucu herhangi bir veri göndermediği sürece, tarayıcı yeni bir istek gönderene kadar bilgi alışverişi mümkün değildir. Ancak WebSocket protokolü, sunucu ve tarayıcı arasında sürekli açık bir iletişim kanalı sağlayarak, gerçek zamanlı veri iletişimini kolaylaştırır.

WebSocket'in özellikleri şunlardır:

Tam çift yönlü iletişim: WebSocket ile tarayıcı ve sunucu arasında gerçek zamanlı veri alışverişi yapmak mümkündür. Hem sunucu hem de istemci tarafı mesaj gönderebilir ve alabilir.

Kalıcı bağlantı: WebSocket, HTTP bağlantılarından farklı olarak, kalıcı bir bağlantı sağlar. İstek ve yanıt alışverişi yapmak için tekrar tekrar bağlantı kurmak zorunda kalmazsınız.

Düşük gecikme: WebSocket, veri iletişiminde düşük gecikme sağlar. Veri iletimi anında gerçekleşir ve gereksiz tekrar bağlantı kurma işlemleri ortadan kalkar.

Gerçek zamanlı veri iletişimi: WebSocket, gerçek zamanlı uygulamalar için idealdir. Çevrimiçi sohbet uygulamaları, canlı bildirimler, hisse senedi takip sistemleri gibi uygulamalarda kullanılabilir.

Ticari satış uygulamalarında WebSocket kullanmak, sepet fiyatı gibi canlı verilerin anlık olarak güncellenmesi için yaygın bir yöntemdir. Özellikle birden fazla kullanıcının aynı anda seçim ve alışveriş yaptığı, seçilen ürünlerin fiyatlarının güncel tutulması gereken durumlarda WebSocket kullanımı oldukça faydalı olabilir. Bunun dışında WebSocket yapısı; canlı sohbet uygulamarında, gerçek zamanlı güncellemelerin olduğu sosyal medya akışlarında, hisse senedi takip sistemlerinde, çoklu oyunculu oyunlarda özetle gerçek zamanlı ve etkileşimli veri iletişimi gereken uygulamalarda yaygın olarak kullanılır.

2- Aşağıdaki bazı tablolar belirtilmiştir.

+ Tablo Adı: Car

Alanlar : id , brand name, model year , price , auto gallery

+ Tablo Adı: Auto Gallery

Alanlar : id , name , cars

2 tablo birbirleriyle bağı one to many olarak yapılmıştır.

Parametreler brand name , model year

2 ayrı koşul değeri alan bir sql query yazılmalı ve brand name ve model year aynı olan kayıtların sayısı 3 adetten fazla olanları çıktı olarak vermelidir.

```
SELECT brandname, modelyear, COUNT(*) AS record_count
FROM tbl_car
GROUP BY brandname, modelyear
HAVING COUNT(*) > 3;
```

Bu SQL sorgusu, "tbl_car" adlı tablodan verileri çeker ve "brandname" ve "modelyear" sütunlarına göre gruplandırır. Ardından, her grup için kayıt sayısını hesaplar ve "record_count" olarak adlandırılan bir alanla döndürür. Son olarak, "record_count" değeri 3'ten büyük olan grupları seçer ve çıktı olarak verir.

İşlemlerin adımları şu şekildedir:

1. İlk olarak, "tbl_car" tablosundan veriler çekilir.

2. Ardından, "brandname" ve "modelyear" sütunlarına göre gruplandırılır.

3. Her grup için kayıt sayısı "COUNT(*)" fonksiyonuyla hesaplanır ve "record_count" olarak adlandırılan bir alanla döndürülür.

4. Son olarak, "HAVING COUNT(*) > 3" ifadesi kullanılarak, "record_count" değeri 3'ten büyük olan gruplar seçilir.

5. Seçilen gruplar, "brandname", "modelyear" ve "record_count" sütunlarından oluşan bir çıktı olarak verilir.

Bu sorgu, "tbl_car" tablosunda "brandname" ve "modelyear" değerlerine göre gruplandırılmış kayıtların sayısını hesaplar ve bu sayı 3'ten büyük olan grupları çıktı olarak verir.

```
SELECT tbl_car.brandname, tbl_car.modelyear, COUNT(*) AS car_count

FROM tbl_car

JOIN tbl_autogallery

ON tbl_car.autogalleri_id = tbl_autogallery.id

WHERE tbl_car.brandname = 'Volswagen' AND tbl_car.modelyear = '2021'

GROUP BY tbl_car.brandname, tbl_car.modelyear

HAVING COUNT(*) > 3;
```

Bu SQL sorgusu, "tbl_car" ve "tbl_autogallery" adlı iki tabloyu birleştirerek belirli koşullara göre verileri filtreler ve gruplar. Sorgunun adımları şu şekildedir:

1. İlk olarak, "tbl_car" ve "tbl_autogallery" tabloları arasında birleştirme işlemi gerçekleştirilir. Bu birleştirme, "tbl_car" tablosundaki "autogallery_id" sütunu ile "tbl_autogallery" tablosundaki "id" sütunu arasındaki eşleşmeye dayanır.

2. Ardından, sadece "tbl_car" tablosunda "brandname" sütunu "Volswagen" ve "modelyear" sütunu "2021" olan kayıtlar seçilir.

3. Sonra, "brandname" ve "modelyear" sütunlarına göre gruplandırılır.

4. Her grup için "COUNT(*)" fonksiyonu kullanılarak kayıt sayısı hesaplanır ve "car_count" olarak adlandırılan bir alanla döndürülür.

5. Son olarak, "HAVING COUNT(*) > 3" ifadesi kullanılarak, "car_count" değeri 3'ten büyük olan gruplar seçilir.

6. Seçilen gruplar, "brandname", "modelyear" ve "car_count" sütunlarından oluşan bir çıktı olarak verilir.

Bu sorgu, "tbl_car" tablosunda "brandname" sütunu "Volswagen" ve "modelyear" sütunu "2021" olan kayıtların sayısını hesaplar ve bu sayı 3'ten büyük olan grupları çıktı olarak verir. Sorgunun amacı, belirli bir marka ve model yılına sahip araçlardan en az 3 adet olanları filtrelemektir.

3- Sıklıkla kullandığınız spring annotasyonları nelerdir örnekleyiniz.

Spring Framework kapsamında en sık kullandığım temel anotasyonlardan bazıları şunlardır:

@Autowired: Bir sınıfın, bağımlılıklarını otomatik olarak enjekte etmek için kullanılır. Spring, ilgili bağımlılıkları otomatik olarak çözer ve enjekte eder.

@Controller: Bir sınıfın, bir MVC (Model-View-Controller) uygulamasında bir kontroller olarak işaretlenmesi için kullanılır. İstekleri işleyen ve uygun yanıtları döndüren bir bileşen olduğunu belirtir.

@RestController: Bir sınıfın HTTP isteklerini karşılayabilen ve HTTP yanıtlarını döndürebilen bir Controller sınıfı olduğunu belirtir. Bir RESTful web servisi, HTTP protokolünü kullanarak kaynakları temsil eden bir API sunar ve istemcilerle bu kaynaklar üzerinde etkileşimde bulunmayı sağlar.

@Service: Bir sınıfın, iş mantığı veya servis katmanı işlemlerini gerçekleştiren bir bileşen olduğunu belirtmek için kullanılır. İş mantığı işlemlerinin gerçekleştirildiği servis katmanının bir parçasıdır.

@Repository: Bir sınıfın, veritabanı erişimi veya veri erişim katmanı işlemlerini gerçekleştiren bir bileşen olduğunu belirtmek için kullanılır. Veritabanı işlemlerini yürütmek için kullanılan veri erişim katmanının bir parçasıdır.

@Entity: Bir sınıfın veritabanında bir tabloya karşılık geldiğini belirtmek için kullanılır. Bu anotasyonu bir sınıfa eklediğimizde, o sınıfın bir veritabanı tablosu olarak kullanılabileceğini belirtmiş oluruz. Bu sınıflar genellikle veritabanında kalıcı olarak saklanması gereken verileri temsil eder.

@RequestMapping: Bir metodun, belirli bir HTTP isteği URL'sine eşleştirildiğini ve bu isteğe yanıt verdiğini belirtmek için kullanılır. İstek türüne, URL'ye veya parametrelere göre özelleştirilebilir.

@GetMapping, @PostMapping, @PutMapping, @DeleteMapping: Bu anotasyonlar, sırasıyla GET, POST, PUT ve DELETE HTTP isteklerini işlemek için kullanılır. İlgili metodun, belirli bir URL'ye ve ilgili HTTP isteğine yanıt verdiğini belirtir.

@PathVariable: Bir metodun, URL'de yer alan dinamik değişkenleri (path variables) almak için kullanılır. Bu değişkenler, URL içindeki değişken parçalarını temsil eder.

@RequestBody: Bir metodun, gelen HTTP isteğinin gövdesindeki veriyi almak için kullanılır. JSON veya XML gibi formatlarda gelen veriyi ilgili nesne türüne dönüştürmek için kullanılır.

@ResponseBody: Bir metodun, dönüş değerini HTTP isteğinin gövdesine yerleştirmek için kullanılır. İlgili nesnenin JSON veya XML gibi formatlarda istemciye gönderilmesini sağlar.

4- Aşağıda verilmiş olan method exception oluşturur mu yoksa hatasız çalışırmı ?

Exception oluşturacak ise eğer bu sistem exception almadan (sayfada yada api de hata oluşmadan) süreci nasıl handle edebiliriz ?

Exception oluşturacak ise eğer neden exception oluşturur. Exception oluşturmaması için kod bloğunda nasıl bi refactor yapılmalıdır.

```
public boolean isGiftCart(CartModel cart) {
 return cart.getType().equals(CartType.GIFT);}
```

Verilen kod bloğu, NullPointerException hatası oluşturabilir. Eğer cart parametresi veya databasedeki enum classından gelecek olan verilerden biri veya her ikisi null ise NullPointerException atacaktır.

Bu durumu ele almak ve NullPointerException'ı önlemek için aşağıdaki gibi bir düzenleme yapılabilir:

```
public boolean isGiftCart(CartModel cart) {
   return cart != null && cart.getType() != null && cart.getType().equals(CartType.GIFT);
}
```

Öncelikle cart nesnesinin null olup olmadığı kontrol edilir. Ardından cart.getType() çağrısı yapılırken, null-check yapılır ve exception oluşması önlenir. Bu sayede hata oluşmadan işlem devam edebilir ve false değeri döndürülerek hata durumu işaretlenebilir.

Eğer bu sistemde hata oluşması durumunda exceptionları handle etmek ve hataları yönetmek istersek, aşağıdaki gibi try-catch bloğu kullanabiliriz:

```
public boolean isGiftCart(CartModel cart) {
 try {
   return cart.getType().equals(CartType.GIFT);
 } catch (NullPointerException e) {
   // Hata durumuyla ilgili işlemler burada yapılır
   // Örneğin, hata günlüğüne kaydedilebilir veya hata mesajı kullanıcıya gösterilebilir
   return false; // Hata durumunu işaretlemek için false değeri döndürülür
 }
} Bu şekilde, NullPointerException gibi exceptionları handle edebilir ve hata durumunu yönetebiliriz.
```

5- Local database kurulumu (mysql, postgresql veya herhangi bir database)

- Java spring uygulaması ayağa kaldırılması,

- İki adet tablo yer alamalı ve bu tabloların birbirleriyle bağı olmalıdır.

Order (create date, id , total price, customer,)

Customer (id , name , age , orders)

- Java spring uygulamasında ekleme,silme,güncelleme,listeleme gibi servisler yer almalıdır ve responseda yapılan işlem detayı return edilmelidir.

- Ekleme,silme,güncelleme,listeleme işlemlerini postman vb ile işlem yapılabilmelidir.

Bu adımlar sırasıyla izlenip java uygulaması üzerinden database' e kayıt atılmalı (Herhangi bir kayıt olabilir fark etmez. Database'de bir tablo açılıp o tabloya değer girilmesi java isteği üzerinden). Daha sonra aynı istek atılan uygulama ile (örnek postman ...) get ve list java spring endpointleri çağırılarak, database e atılan kayıt response olarak dönülmeli.

MVC deki model ve kontroller akışının güzel kurgulanması ve uygulama ayağı nasıl kaldırılıp servislerin nasıl kullanıldığına dair bir döküman hazırlanmalıdır. Bu proje için kaynak kodu ile iletilmesi gerekmektedir.

- Yukarıdaki ekleme silme güncelleme listeleme işlemlerine ek olarak

1- Bir servis olmalı ve parametre olarak verilen tarihten sonra oluşturulmuş siparişleri listelesin.

2- Bir servis olmalı ve bir kelime yada harf değerini parametre olarak alsın ve isminin içerisinde bu değer geçen müşteri ve müşteriye ait sipariş id sini getirsin.

3- Bir servis olmalı ve siparişi olmayan müşterileri listesin.

Proje İsmi : Cafe

Proje Java kullanılarak kurgulanan bir Spring Boot projesidir. Spring Boot, Java tabanlı bir web uygulama çatısıdır ve Spring Framework'ün üzerine inşa edilmiştir. Spring Boot, Spring projelerinin hızlı bir şekilde başlatılmasını, yapılandırılmasını, dağıtılmasını kolaylaştırır ve standart bir Spring uygulaması için gerekli olan birçok yapılandırmayı otomatik olarak sağlar, böylece geliştiricilerin daha az zaman harcamasını ve daha fazla odaklanmasını sağlar.

Projenin temel katmanları ve işleyişi:

Entity: Veritabanında saklanan verilerin sınıflarını içerir. Bu sınıflar genellikle ORM (Object Relational Mapping) teknolojisi kullanılarak oluşturulur.

Controller: Kullanıcı isteklerini karşılamak ve yanıtlamakla sorumludur. HTTP isteklerini alır, gerekli veri işlemlerini yapar ve HTTP yanıtlarını oluşturur. İsteklerin yönlendirilmesi, veri doğrulaması ve yanıtların oluşturulması gibi işlevler bu katmanda yer alır.

Service: İş mantığının gerçekleştirildiği katmandır. Controller'dan gelen istekleri alır, gerektiğinde veri erişim katmanına (Repository) erişir, iş mantığı işlemlerini gerçekleştirir ve veri manipülasyonu veya işlemleri koordine eder. Birden fazla Repository'nin kullanıldığı durumlarda Service katmanı, iş süreçlerinin yönetimi için kullanılır.

Repository: Veritabanı işlemlerinin gerçekleştirildiği katmandır. Veri erişim işlemlerini yapmak için kullanılır. Veritabanı tabloları ile iletişim kurmak, sorgular oluşturmak, veri ekleme, güncelleme, silme gibi işlemleri gerçekleştirmek bu katmanda yapılır. Spring Data JPA veya Spring JDBC gibi veri erişim teknolojileri kullanılarak gerçekleştirilebilir.

DTO (Data Transfer Object): Veri transferi için kullanılan sınıflardır. Controller ve Service katmanları arasında veri taşımak için kullanılır. DTO'lar, ilgili alanları içerir ve veri transferi sırasında istemci ve sunucu arasında veri uyumluluğunu sağlar. DTO'lar genellikle veritabanı entity sınıflarından farklıdır ve genellikle veri dönüşümü veya sadece belirli alanların aktarılması için kullanılır.

Öncelikle projenin tanımı, amacı ve katmanların yapısı ile ilgili bilgiler vermek ardından tüm bu adımların uygulandığı kodları adım adım açıklamaları ile göstermek istiyorum.

Projede Customer ve Order olmak üzere iki Entity Class ve bu classların Controller, Service , Repository, katmanları bulunmaktadır. Projenin veritabanı işlemleri PostgreSQL veritabanı kullanarak gerçekleştirildi. Customer ve Order bilgilerinin tutulması için “t_customer” ve “t_order” isimli iki farklı tablo oluşturuldu. “t_customer” tablosu id , name , age , orders fieldlarına sahiptir. “t_order” tablosu ise create date, id , total price, customer fieldlarına sahiptir. Tabloların birbiriyle etkileşim içerisinde olması Hibernate ile sağlandı. JPQL ile sorgulamalar yapıldı. Manuel testler için PostmanAPI kullanıldı.

Customer Sınıfı ile ilgili yapılan işlemler :

- Gerekli anotasyonlar konuldu ve injectionlar yapıldı.
- Post Mapping ile Customer kaydı yapıldı.
- Get Mapping ile belirli Id’ye sahip Customer sorgulaması yapıldı ve DTO olarak response edildi.
- Get Mapping ile Pageable yapıda bütün Customerların sorgulaması yapıldı ve DTO olarak response edildi.
- Get Mapping ile veritabanındaki bütün Customerların sorgulaması yapıldı ve List yapısında listelendi.
- Put Mapping ile Customer güncellemesi yapıldı.
- Delete Mapping ile Customer silindi.
- Postman ile manuel testler gerçekleştirildi.

Order Sınıfı ile ilgili yapılan işlemler :

- Gerekli anotasyonlar konuldu ve injectionlar yapıldı.
- Post Mapping ile Order kaydı yapıldı.
- Get Mapping ile belirli Id'ye sahip Order sorgulaması yapıldı ve DTO olarak response edildi.
- Get Mapping ile Pageable yapıda bütün Orderların sorgulaması yapıldı ve DTO olarak response edildi.
- Get Mapping ile veritabanındaki bütün Orderların sorgulaması yapıldı ve List yapısında listelendi.
- Put Mapping ile Order güncellemesi yapıldı.
- Delete Mapping ile Order silindi.
- Postman ile manuel testler gerçekleştirildi.

Database Bağlantısı:

Projenin veritabanı işlemleri PostgreSQL kullanılarak gerçekleştirildi. PostgreSql içerisinde oluşturmuş olduğum database yol haritamı application.yml dosyamın içerisinde belirttim.

```yaml
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/cafe_db
    username: db_user
    password: db_password
    driver-class-name: org.postgresql.Driver
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
    database-platform: org.hibernate.dialect.PostgreSQL81Dialect
    properties:
      '[hibernate.format_sql]': true
```

Tablo Oluşturma:

Hibernate Injection yardımı ile domain classlarım ile database'imin içerisindeki "t_customer" ve "t_order" isimli iki farklı tablomu oluşturdum.

```java
@Entity
@AllArgsConstructor
@NoArgsConstructor
@Getter
@Setter
@Table(name = "t_order")
@SuperBuilder(toBuilder = true)
@JsonInclude(JsonInclude.Include.NON_EMPTY)
public class Order {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private Double totalprice;

    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd HH:mm")
    private LocalDateTime localDate;

    @ManyToOne
    @JsonIgnore
    @JoinColumn(name="customer_id")
    private Customer customer;
```

```java
@Entity
@AllArgsConstructor
@NoArgsConstructor
@Getter
@Setter
@Table(name = "t_customer")
@SuperBuilder (toBuilder = true)
@JsonInclude(JsonInclude.Include.NON_EMPTY)
public class Customer {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    private Integer age;

    @OneToMany(mappedBy = "customer", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    private List<Order> orders;
```

Controller:

Controller katmanları içerisinde Customer ve Order ile ilgili yukarıda belirttiğim POST, GET, UPDATE, DELETE ve listeleme gibi tüm işlemler yapıldı. Bu işlemlerin gerçekleşmesi için gerekli injectionlar ve anotasyonlar konuldu.

```java
@RestController
@RequestMapping("/customer")
@RequiredArgsConstructor
public class CustomerController {

    private final CustomerService customerService;
```

```java
@RestController
@RequestMapping("/orders")
@RequiredArgsConstructor
public class OrderController {


    private final OrderService orderService;
```

Service:

Projenin iş mantığının gerçekleştirildiği katmandır. Controllerdan gelen tüm istekler ve görevler bu katman içerisinde gerçekleştirildi ve client'a response olarak dönüldü.

```java
@Service
@RequiredArgsConstructor
public class OrderService {

    private final OrderRepository orderRepository;
    private final CustomerService customerService;
```

```java
@Service
@RequiredArgsConstructor
public class CustomerService {

    private final CustomerRepository customerRepository;
```

Repository:

Controller ve Service katmanlarımızda bu işlemlerin gerçekleştirilebilmesi için Repository katmanına ihtiyacımız vardır. Veritabanı işlemlerinin gerçekleştirildiği katmandır. Repository ile bağlantı sağlarken Repository interface'ini JpaRepository'den extend ederek tüm crud operationları'nı gerçekleştiririz.

```java
@Repository
public interface CustomerRepository extends JpaRepository<Customer, Long> {

    1 usage
    @Query("SELECT c FROM Customer c WHERE lower(c.name) LIKE %:pName%")
    List<Customer> findAllByNameLike(@Param("pName") String name);

    1 usage
    List<Customer> findCustomerByOrders_Empty();
```

```java
@Repository
public interface OrderRepository extends JpaRepository<Order, Long> {


}
```

Projenin katmanları, gerçekleştirilen işlemleri ve projenin işleyişi ilgili detayları GitHub bağlantısı üzerinden bulabilirsiniz.

https://github.com/kesermustafa/CafeApp.git