ARAÇ SERVİS OTOMASYON SİSTEMİ

Yazılım Mühendisliği Ana Bilim Dalı

Dönem Projesi

ALİ EKŞİ

Proje Danışmanı: Prof. Dr. Merih PALANDÖKEN

Ocak 2024

Araç Servis Otomasyon Sistemi

Öz

Servis otomasyon sistemleri, işletmelerin rekabetçi iş ortamında ayakta kalabilmeleri ve başarı elde edebilmeleri adına kritik bir öneme sahiptir. Bu sistemler, işletmelerin servis süreçlerini optimize etme amacı güderken, aynı zamanda müşteri memnuniyetini artırma ve işletme verimliliğini maksimize etme hedeflerini desteklemektedir. Yapılan çalışma da servis otomasyon sistemlerinin işletmeler üzerindeki olumlu etkilerini çeşitli açılardan ele almayı amaçlamaktadır. İlk olarak, iş süreçlerinin nasıl optimize edildiği ve bu optimizasyonun işletme verimliliği üzerindeki etkileri detaylı bir şekilde incelenecektir. Servis otomasyonunun, manuel ve tekrarlayan görevleri ortadan kaldırarak iş süreçlerindeki hataları azaltması ve gecikmeleri minimize etmesi, işletmelerin daha etkili ve hızlı hizmet sunmalarına olanak tanımaktadır.

Anahtar Sözcükler: Servis Otomasyon Sistemleri, İş Süreç Optimizasyonu, İşletme Verimliliği, Servis yönetimi, Teknolojik çözümler

Vehicle Service Automation System

Abstract

Service automation systems hold critical significance for businesses to thrive and succeed in a competitive business environment. These systems not only aim to optimize business service processes but also support objectives of increasing customer satisfaction and maximizing operational efficiency. This study aims to comprehensively explore the positive impacts of service automation systems on businesses from various perspectives. Initially, it will meticulously examine how business processes are optimized and the detailed effects of this optimization on operational efficiency. Service automation, by eliminating manual and repetitive tasks, reduces errors in business processes and minimizes delays, enabling businesses to provide more effective and prompt services.

Keywords: Service Automation Systems, Business Process Optimization, Operational Efficiency, Service Management, Technological Solutions

Teşekkür

Bu projede sağladığı değerli katkılar ve proje rehberliği için Sayın Prof. Dr. Merih PALANDÖKEN, teşekkür etmek isterim. Projemizin başarılı bir şekilde tamamlanmasında sizin bilgi birikiminiz ve ilginiz çok önemli oldu. Bu süreçte sizinle çalışmak, benim için büyük bir öğrenme deneyimi oldu.

Gösterdiğiniz rehberlik ve ilgi için tekrar teşekkür ederim.

İçindekiler

Öz i

Abstract ii

Teşekkür iii

Şekiller Listesi vi

Kısaltmalar Listesi vii

1 Giriş 1

1.1 Problem 2

1.2 Amaç 3

1.3 Önem 4

1.4 Kullanılan Teknolojiler 5

2 Çok Katmanlı Mimari 6

2.1 Presentation Layer (UI, Sunum Katmanı) 7

2.2 Entity Layer (Varlık Katmanı) 8

2.3 Business Layer (İş Katmanı) 10

2.4 Data Access Layer(Veri Erişim Katmanı) 11

2.5 API Layer 13

2.6 Varlık İlişki Diyagramı 14

3 Araç Servis Otomasyon Sistemi 17

3.1 Projenin Mimarisi 19

3.2 Projenin Katmanları 21

4 Sonuç 28

Kaynaklar 31

Şekiller Listesi

Şekil 2.1 Katman Mimarisi 6
Şekil 2.2 Varlık İlişkisi Sembolleri 15
Şekil 2.3 Varlık İlişkisi Türleri 16
Şekil 2.4 Varlık İlişkisi Diyagramı 17

Şekil 3.1 Data Access Layer 21
Şekil 3.2 Login Controller 22
Şekil 3.3 Action Result 23
Şekil 3.4 Business Layer 23
Şekil 3.5 Business Layer Concrete 24
Şekil 3.6 Data Access Layer Migrations 25
Şekil 3.7 Data Access Layer Repositories 26
Şekil 3.8 Entity Layer Concrete 27
Şekil 3.9 Car Service 27
Şekil 3.10 Varlık İlişki Diyagramı 28

Kısaltmalar Listesi

VW	View
DB	Database
TB	Table
CRUD	Create, Read, Update, Delete
Auth	Authentication
UI	User Interface
ERD	Entity Relationship Diagram
ORM	Object Relational Mapping
API	Application Programming Interface
MVC	Model View Controller

Bölüm 1

Giriş

Günümüz hızla değişen iş dünyasında, işletmelerin varlık göstermeleri ve rekabet avantajı elde etmeleri, teknolojik çözümleri etkin bir şekilde kullanmalarına bağlıdır. Bu bağlamda, servis otomasyon sistemleri, işletmelerin operasyonel süreçlerini optimize etme, müşteri memnuniyetini artırma ve genel verimliliği maksimize etme konularında kritik bir role sahiptir. Bu çalışma, günümüz iş ortamında servis otomasyon sistemlerinin işletmelere sağladığı önemli avantajları detaylı bir şekilde incelemeyi amaçlamaktadır. Öncelikle, servis otomasyon sistemlerinin iş süreçlerini nasıl optimize ettiği ve bu optimizasyonun işletme verimliliği üzerindeki etkileri ele alınacaktır. Bu sistemlerin manuel ve tekrar eden görevleri ortadan kaldırarak iş süreçlerindeki hataları en aza indirme kapasitesi, işletmelere daha etkili ve hızlı hizmet sunma imkânı sağlamaktadır. Özellikle, günümüz müşteri beklentilerinin hızlı yanıt ve çözümler üzerine odaklandığı bir dönemde, servis otomasyonu işletmelerin bu beklentilere uygun olarak faaliyet göstermelerini desteklemektedir. Ayrıca, yapılan çalışma işletmelerin müşteri memnuniyetini artırma üzerindeki etkilerini de ele alacaktır. Servis otomasyon sistemleri, müşteri ilişkilerini daha etkili yönetme ve hızlı çözümler sunma konusunda işletmelere önemli bir avantaj sağlamaktadır. Müşteri memnuniyetinin artması, marka sadakati ve uzun vadeli müşteri ilişkilerinin güçlenmesine katkıda bulunarak işletmelerin rekabet avantajını artırmaktadır. Sonuç olarak, bu çalışma, servis otomasyon sistemlerinin günümüz iş dünyasında işletmeler için taşıdığı önemli rolü vurgulayarak, bu teknolojik çözümlerin işletmelerin rekabetçiliğini artırmada nasıl etkin bir araç olduğunu anlamamıza katkı sağlamayı amaçlamaktadır. Brown ve Taylor (2019) şu şekilde belirtmiştir: "Servis otomasyon sistemleri, iş süreçlerini optimize ederek manuel hataları en aza indirir ve bu durum işletmelerin genel verimliliğini artırır."

1.1 Problem

Dijitalleşme hızının yüksek olduğu günümüzde, dijitalleşmeyen süreçlerdeki aksaklıklar nedeniyle müşteri memnuniyeti düşmektedir. Servis otomasyon sistemlerinin uygulanması sırasında karşılaşılan en önemli sorunlardan biri, işletmelerin mevcut altyapılarıyla uyumsuzluk ve kullanıcı eğitimi zorlukları gibi faktörlerin, sistemin etkin ve sorunsuz bir şekilde entegre edilmesini engelleyebilecek potansiyel problemler olmasıdır.

Servis otomasyonuyla ilgili karşılaşılan karmaşık sorunlar ve zorluklar, işletmelerin günlük operasyonel süreçlerini etkin bir şekilde yönetme çabalarını büyük ölçüde etkilemektedir. Bu bağlamda, servis otomasyonu uygulamalarının yaygınlaşmasıyla birlikte ortaya çıkan çeşitli problemler, işletmelerin bu teknolojiyi benimseme ve entegre etme süreçlerini zorlamaktadır.

Birincil sorun, işletmelerin mevcut altyapılarıyla servis otomasyon sistemlerini uyumlu hale getirme zorunluluğudur. Birçok işletme, yıllar içinde geliştirdikleri altyapılarını ve yazılım sistemlerini, hali hazırda karmaşık olan servis otomasyon sistemleriyle bütünleştirmekte zorlanmaktadır. Bu uyumsuzluklar, iş süreçlerinde aksamalara ve verimlilik kayıplarına neden olabilir, ayrıca maliyetleri artırabilir.

İkinci önemli bir problem, işletmelerin servis otomasyon sistemlerini etkili bir şekilde kullanabilmesi için personelinin uygun şekilde eğitilmesi gerekliliğidir. Servis otomasyonu, genellikle karmaşık bir yapıya sahip olduğundan, personelin bu sistemleri etkin bir şekilde kullanabilmesi için özel bilgi ve becerilere ihtiyaç duyar. Ancak, birçok işletme, personel eğitimi konusunda yeterli kaynak ve zamanı ayıramayabilir, bu da sistemin potansiyelini tam anlamıyla kullanma konusunda engel teşkil eder.

Üçüncü bir sorun, müşteri beklentilerine hızlı ve esnek bir şekilde yanıt verebilmek adına servis otomasyon sistemlerinin gerektirdiği sürekli güncellemeler ve iyileştirmelerdir. İş dünyasındaki dinamik değişimlere ayak uydurabilen bir servis otomasyonu sistemi, müşteri memnuniyetini artırabilir ve rekabet avantajı

sağlayabilir. Ancak, bu sürekli güncelleme ve iyileştirme ihtiyacı, işletmelerin bu teknolojiyi sürdürülebilir bir şekilde benimsemesini zorlaştırabilir.

1.2 Amaç

Servis otomasyonu, günümüzde işletmelerin rekabet avantajı elde etmeleri ve müşteri memnuniyetini en üst düzeye çıkarmaları için önemli bir stratejik araç haline gelmiştir. Bu bağlamda, servis otomasyonunun temel amacı, işletmelerin karmaşık ve dinamik servis süreçlerini daha etkili bir şekilde yönetmelerini sağlamaktır. Bu sistemler, müşteri taleplerini hızlı bir şekilde karşılamak, kaynakları daha verimli kullanmak ve işletme içindeki süreçleri optimize etmek amacıyla tasarlanmıştır.

Servis otomasyonunun temel amacı, müşteri odaklı bir yaklaşımla işletmelerin sunduğu hizmetleri iyileştirmektir. Müşteri memnuniyetinin gün geçtikçe daha kritik hale geldiği bu dönemde, servis otomasyonu işletmelere, müşteri taleplerini anında değerlendirme, çözüme kavuşturma ve etkileşim süreçlerini optimize etme imkânı sunar. Bu, müşteri sadakatinin artırılması, olumlu müşteri deneyimlerinin sağlanması ve marka itibarının güçlendirilmesi ile sonuçlanabilir.

Bunun yanı sıra, servis otomasyonu işletmelere operasyonel verimliliği artırma ve iş süreçlerini daha etkin bir şekilde yönetme fırsatı sağlar. Bu sistemler, manuel iş süreçlerini otomatikleştirme, veri analizi ile karar almayı destekleme ve kaynakları daha etkili bir biçimde kullanma imkânı sunar. Bu da işletmelerin zaman ve maliyet tasarrufu elde etmelerine, hataları en aza indirmelerine ve daha hızlı tepki vermelerine olanak tanır.

Ayrıca, servis otomasyonunun amacı, işletmelerin rekabet avantajını güçlendirmektir. Hızla değişen iş ortamında, rekabetçi kalabilmek için işletmelerin sürekli olarak iyileşme ve dönüşme ihtiyacı vardır. Servis otomasyonu, bu dönüşümü desteklemek adına işletmelere daha esnek, ölçeklenebilir ve yenilikçi bir hizmet sunma yeteneği kazandırır. Bu sayede, işletmeler pazardaki taleplere daha hızlı uyum sağlayabilir, yeni fırsatları değerlendirebilir ve rekabet avantajı elde edebilirler.

Sonuç olarak, servis otomasyonunun temel amacı, işletmelerin müşteri memnuniyetini artırma, operasyonel verimliliği optimize etme ve rekabet avantajı elde etme çabalarına önemli bir katkı sağlamaktır. Bu sistemler, modern iş dünyasının dinamik ihtiyaçlarına uyum sağlamak için tasarlanmıştır ve işletmelere geleceğe daha güvenle bakma imkânı sunar.

1.3 Önem

Servis otomasyonu, günümüz iş dünyasında önemli bir stratejik unsur olarak ortaya çıkarak, işletmelerin operasyonel süreçlerini dönüştürme, müşteri memnuniyetini artırma ve rekabet avantajı elde etme konularında kritik bir rol oynamaktadır. Bu sistem, işletmelerin karmaşık servis süreçlerini daha etkili yönetmelerini sağlamakla kalmayıp aynı zamanda verimliliklerini artırarak maliyetleri optimize etmelerine ve müşteri odaklı hizmet sunmalarına olanak tanımaktadır. Servis otomasyonunun önemi, işletmelerin bu teknolojik çözümü benimsemesi ve doğru bir şekilde uygulamasıyla birlikte ortaya çıkan çeşitli avantajlarla birlikte anlaşılabilir.

Birincil olarak, servis otomasyonu işletmelerin operasyonel süreçlerini optimize etme konusundaki ihtiyaçlarına yanıt vererek verimliliği artırmaktadır. Otomasyon, manuel süreçlere kıyasla daha hızlı ve hatasız işlemler sağlayarak, kaynakların daha etkin bir şekilde kullanılmasını mümkün kılar. Bu, işletmelerin zaman kaybını en aza indirmelerini ve operasyonel mükemmelliğe daha hızlı bir şekilde ulaşmalarını sağlar.

İkinci olarak, servis otomasyonu müşteri memnuniyetini artırmada kritik bir rol oynamaktadır. Müşteriler, hızlı ve etkili hizmetlere olan taleplerini her geçen gün artırmaktadır. Servis otomasyon sistemi, müşteri taleplerine daha hızlı yanıt verme, sorunları daha etkili bir şekilde çözme ve genel müşteri deneyimini geliştirme konusunda işletmelere önemli avantajlar sunar. Bu da uzun vadede müşteri sadakatini artırarak, marka itibarını güçlendirir.

Üçüncü olarak, servis otomasyonu işletmelerin rekabet avantajı elde etmelerine yardımcı olmaktadır. Rekabetin yoğun olduğu bir iş ortamında, hızlı tepki verme, esneklik ve inovasyon, işletmelerin ayakta kalabilmesi için kritik unsurlardır. Servis otomasyonu, iş süreçlerini daha esnek ve ölçeklenebilir hale getirerek, işletmelerin

değişen pazar koşullarına hızla adapte olmalarına olanak tanır. Bu da işletmelerin rekabet avantajını sürdürmelerine ve pazarda lider konumlarını güçlendirmelerine katkı sağlar.

Sonuç olarak, servis otomasyonu, işletmelerin günümüz dinamik iş ortamında başarılı olmalarını sağlamak adına kritik bir öneme sahiptir. Verimliliği artırma, müşteri memnuniyetini yükseltme ve rekabet avantajı elde etme potansiyeli, servis otomasyonunu benimseyen işletmeler için sadece bir teknolojik yatırım değil, aynı zamanda sürdürülebilir bir başarı stratejisi olarak ortaya çıkmaktadır. Bu nedenle, işletmelerin servis otomasyonunu doğru bir şekilde değerlendirmesi, uygulaması ve sürekli olarak iyileştirmesi, başarılı bir gelecek için kritik bir adım olacaktır.

1.4 Kullanılan Teknolojiler

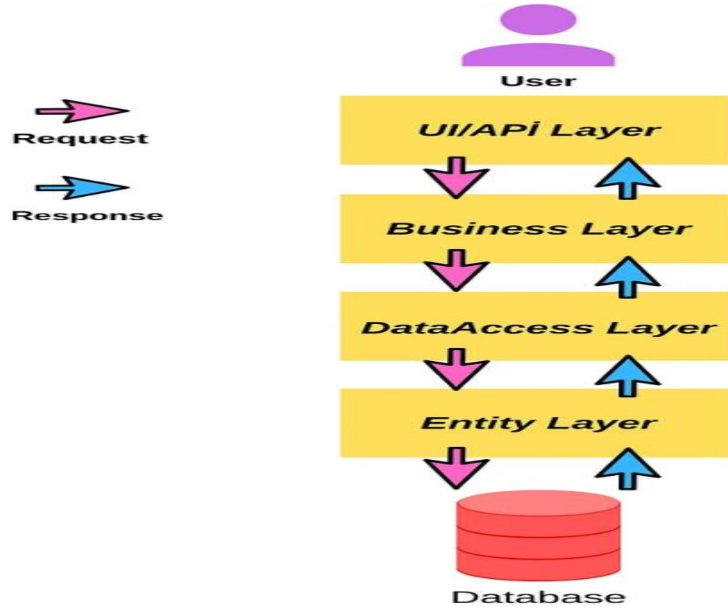
Projede kurumsal katmanlı mimari yapısı kullanılmış olup, code first yapısı kullanılmıştır. Kullanılan teknolojiler,

- ASP.NET Core 5.0 Framework
- MsSQL Veri tabanı
- Microsoft Simple Authentication
- HTML/CSS3
- Entity Framework
- Identity Framework

Bölüm 2

Çok Katmanlı Mimari

Gelişen teknoloji ile birlikte, işletmeler ve kuruluşlar daha karmaşık ve büyük ölçekli uygulamalar geliştirmek zorunda kalıyorlar. Bu tür uygulamaların geliştirilmesi ve sürdürülebilirliği, doğru bir mimari deseni kullanıldığında daha etkili bir şekilde yönetilebilir. ASP.NET Core N-tier architecture (Katmanlı Mimari), bu ihtiyaçları karşılamak ve büyük ölçekli uygulamaların geliştirilmesini optimize etmek için kullanılan bir mimari desendir. N-tier mimarisi, yazılım uygulamalarının tasarımını ve geliştirilmesini düzenlemek amacıyla kullanılan bir mimari yaklaşımdır. Bu yaklaşım, bir uygulamayı farklı katmanlara ayırarak her bir katmanın belirli bir sorumluluğu üstlenmesini ve bu katmanların birbirleriyle etkileşimde bulunmasını sağlar. Her katman, uygulamanın belirli bir yönünden sorumlu olup, bu katmanlar genellikle sunum, iş mantığı, veri erişimi ve veritabanı katmanları olarak adlandırılır.



Şekil 2.1: Katman Mimarisi

Bu katmanlar, uygulamanın farklı bileşenlerini belirli bir düzen içinde sınıflandırarak bakımını, güncellemesini ve genişletmesini kolaylaştırır. N-tier mimarisi, büyük ve karmaşık yazılım projelerinde kullanıldığında daha iyi organizasyon ve sürdürülebilirlik sağlar. Her bir katmanın bağımsız olarak geliştirilebilmesi ve değiştirilebilmesi, yazılımın daha esnek olmasını ve değişen ihtiyaçlara daha hızlı cevap verebilmesini sağlar.

ASP.NET Core uygulamalarının genellikle N-tier mimari modelini takip ettiğini bilmekteyiz. N-tier mimarisi, uygulamayı farklı katmanlara ayırarak organize eder. Bu katmanlar, her biri belirli bir sorumluluğa sahip olan ve belirli iletişim kuralları setine tabi olan yapısal bir düzen sunar.

2.1 Presentation Layer (UI, Sunum Katmanı)

Bu katman, kullanıcı arayüzünü oluşturan ve kullanıcı ile etkileşime geçen bileşenleri içerir. Web sayfaları, kullanıcı arayüzü kontrolleri ve diğer sunum ile ilgili mantık burada bulunur. Sunum katmanı, bir yazılım uygulamasının dış dünyayla etkileşimde bulunan ve kullanıcı arayüzünü yöneten katmandır. Genellikle bu katman, bir uygulamanın web tarayıcıları, mobil cihazlar veya masaüstü uygulamalar gibi farklı istemci platformlarıyla nasıl etkileşimde bulunduğunu kontrol eder. ASP.NET Core uygulamalarında bu katman, genellikle web tabanlı kullanıcı arayüzlerini içerir. Sunum katmanının temel görevi şunlardır:

Kullanıcı Arayüzü Yönetimi: Sunum katmanı, kullanıcı arayüzünü oluşturur, düzenler ve yönetir. Bu, kullanıcının uygulama ile etkileşimde bulunurken gördüğü sayfalar, formlar, düğmeler ve diğer kullanıcı arayüzü elemanlarını içerir.

İstemci Tarafı İletişim: Web uygulamalarında, sunum katmanı genellikle istemci tarafa (tarayıcıya) HTML, CSS ve JavaScript gibi kaynakları ileterek ve bu kaynaklar aracılığıyla kullanıcı ile etkileşime geçer.

Controller/ViewModel Yönetimi (MVC Mimarisi): Sunum katmanı, Model-View-Controller (MVC) mimarisini takip eden uygulamalarda, Controller ve ViewModel'leri yönetir. Controller, gelen istekleri alır, işlemi başlatır ve sonuçları ViewModel'e dönüştürerek View'e gönderir.

Kullanıcı Girişi ve Doğrulama: Sunum katmanı, kullanıcı girişi ve kimlik doğrulama işlemlerini yönetir. Kullanıcıların giriş yapması, şifre sıfırlama, kullanıcı hesapları ve oturum yönetimi gibi işlevselliği içerir.

Client-Side Scripting ve Frameworks: İstemci tarafındaki (client-side) işlemleri yönetir. JavaScript veya TypeScript gibi client-side dilleri kullanarak sayfa içi etkileşimleri ve dinamik içerik güncellemelerini destekler. Ayrıca, popüler JavaScript çerçeveleri (Angular, React, Vue.js) gibi teknolojileri de entegre edebilir.

Internationalization ve Localization: Birden fazla dil veya bölgeyi destekleyen uygulamalarda, sunum katmanı internationalization (i18n) ve localization (l10n) işlemlerini yönetir.

CSS ve Tasarım Yönetimi: Kullanıcı arayüzünün görünümünü düzenler ve CSS gibi teknolojiler aracılığıyla tasarım yönetimini sağlar. Bu, uygulamanın kullanıcı dostu ve marka kimliğine uygun bir görünüme sahip olmasını sağlar.

Hata Yönetimi ve İletişimi: Sunum katmanı, hata mesajlarını yönetir ve kullanıcıya uygun bir şekilde iletilmesini sağlar. Ayrıca, kullanıcıya bilgi veren bildirimler, uyarılar ve diğer iletişim elemanlarını içerir.

Uygulama Durumu Yönetimi: Kullanıcı oturumları, sayfa geçmişi ve diğer uygulama durumu ile ilgili görevleri yönetir.

Sunum katmanı, genellikle uygulamanın dış dünyayla etkileşime girdiği ve kullanıcı deneyiminin şekillendiği bir nokta olarak öne çıkar. Bu katman, diğer katmanlardan gelen verileri kullanıcı dostu bir arayüze dönüştürür ve kullanıcıdan gelen istekleri iş mantığına yönlendirir. Aynı zamanda, modern web uygulamalarında dinamik ve etkileşimli kullanıcı arayüzleri oluşturmak için çeşitli teknolojileri entegre eder.

2.2 Entity Layer (Varlık Katmanı)

Entity Layer, uygulamanın veri modelini temsil eden ve genellikle veritabanı tabloları veya belirli veri nesneleriyle eşleşen bir katmandır. Entity Layer, genellikle uygulamanın iş mantığı katmanı ile veritabanı arasında bir köprü işlevi görerek, veri erişim işlemlerini yönetir. Varlık katmanının temel özellikleri şunlardır:

Veri Modelinin Temsili: Entity Layer, uygulamanın veri modelini temsil eder. Bu katman, genellikle uygulamanın veritabanındaki tabloları veya belirli veri nesnelerini, sınıfları veya varlıkları içerir. Bu sınıflar, uygulamanın iş mantığı tarafından kullanılır.

Varlık Sınıfları: Varlık katmanındaki sınıflar, genellikle veritabanındaki bir tabloyu temsil eder. Her bir varlık sınıfı, tablonun alanlarını (property) ve bu alanlarla ilişkilendirilmiş işlemleri içerir. Entity Framework gibi ORM (Object-Relational Mapping) araçları, bu varlık sınıflarını kullanarak veritabanı işlemlerini kolaylaştırır.

İlişkisel Veri Modeli: Entity Layer, veri modelini ilişkisel bir yapıyla temsil eder. Bu, farklı varlık sınıfları arasındaki ilişkileri ve bağlantıları içerir. Örneğin, bir e-ticaret uygulamasında "Sipariş" ve "Ürün" varlık sınıfları arasında bir ilişki olabilir.

Veritabanı İşlemleri: Entity Layer, genellikle veritabanı işlemlerini yönetir. Bu işlemler arasında veri ekleme, güncelleme, silme (CRUD), sorgulama ve ilişkisel veri manipülasyonu gibi temel veritabanı işlemleri bulunur.

Entity Framework (EF) ve ORM Araçları:Entity Layer, Entity Framework gibi ORM araçlarını içerebilir. ORM araçları, varlık sınıfları ile veritabanı tabloları arasındaki ilişkiyi otomatik olarak yönetir ve veri erişimini kolaylaştırır.

Validasyon ve İş Kuralları: Varlık katmanı genellikle gelen verileri doğrular ve belirli iş kurallarına uyan veri girişini sağlar. Bu, uygulamanın veri bütünlüğünü korumasına yardımcı olur.

DTO'lar (Data Transfer Objects): Varlık katmanı, veri transferi amacıyla kullanılan DTO'ları içerebilir. DTO'lar, veritabanından alınan veriyi, iş katmanına veya sunum katmanına iletmek için kullanılır.

Veritabanı Bağlantısı ve İletişim: Entity Layer, veritabanı ile iletişim için gerekli bağlantıları yönetir. Bu, veritabanına bağlanma, bağlantıyı açma, kapatma gibi işlemleri içerir.

Sorgu Dilleri ve Filtreleme: Entity Layer, genellikle LINQ (Language Integrated Query) gibi sorgu dillerini kullanarak veritabanından veri sorgular. Ayrıca, sorguları filtreleme ve düzenleme yeteneklerine sahiptir.

Varlık katmanı, uygulamanın veri yönetimiyle ilgili görevleri ele alarak iş mantığı katmanını veritabanıyla bağlamak için kullanılır. Bu sayede, veri erişimi ve manipülasyonu genellikle daha düzenli ve modüler bir yapıda olabilir.

2.3 Business Layer (İş Katmanı)

İş Katmanı, bir yazılım uygulamasının temel iş mantığını içeren ve genellikle sunum katmanı ile veri erişim katmanı arasında konumlanan bir katmandır. İş Katmanı, kullanıcı taleplerini işleyerek, uygulamanın temel işlevselliğini yürüten ve iş kurallarını uygulayan birimleri içerir. İş katmanının temel özellikleri şunlardır:

İş Mantığı Uygulama:İş Katmanı, uygulamanın temel iş mantığını içerir. Bu, uygulamanın temel amaçlarına uygun olarak belirlenen iş kurallarının ve süreçlerinin uygulanmasını içerir.

Kurumsal İş Kuralları: İş Katmanı, kurumsal düzeyde geçerli olan iş kurallarını içerir. Bu kurallar, genellikle uygulamanın amacına ve sektöre özgü olarak belirlenir.

Veri Doğrulama ve Validasyon: Gelen verilerin doğruluğunu kontrol eder ve belirli bir formata uymasını sağlar. Bu, veri bütünlüğünü korumak ve hatalı veri girişlerini önlemek için önemlidir.

İş Süreçleri Yönetimi: İş Katmanı, uygulamanın iş süreçlerini yönetir. Kullanıcı taleplerini alır, gerekli işlemleri başlatır ve veri erişim katmanı aracılığıyla veri tabanıyla etkileşimde bulunabilir.

DTO (Data Transfer Object) İşlemleri: İş Katmanı, veritabanıyla etkileşim halindeki verilerin sunum katmanına veya diğer katmanlara iletilmesi için DTO'ları yönetebilir.

İş Katmanı Servisleri: İş Katmanı, genellikle iş süreçlerini ve iş kurallarını uygulayan servis sınıflarını içerir. Bu servisler, veritabanı işlemlerini koordine eder ve uygulamanın çeşitli modüllerinin etkileşimini sağlar.

Transaction Yönetimi: İş Katmanı, işlemleri genellikle bir bütün olarak değerlendirir ve bu işlemleri veritabanında bir transaction içinde gerçekleştirir. Bu, veritabanındaki işlemlerin atomik olmasını sağlar.

Hata Yönetimi ve Günlükleme: İş Katmanı, uygulama içindeki hataları yönetir ve gerekli durumlarda hata günlüğü oluşturur. Bu, uygulamanın güvenilirliğini artırmak için önemlidir.

İş Katmanı Kurallarının Test Edilmesi: İş Katmanı, içerdiği iş kurallarının test edilmesini sağlayacak birim testlerini barındırabilir. Bu, uygulamanın güvenilirliğini ve istikrarını sağlamak için önemlidir.

Uygulama Bağımsızlığı: İş Katmanı, genellikle sunum katmanından ve veri erişim katmanından bağımsızdır. Bu, iş mantığının değiştirilmesi veya güncellenmesi durumunda diğer katmanlara minimal etki etmesini sağlar.

İş Katmanı, uygulamanın merkezinde yer aldığı için genellikle diğer katmanlarla sıkı bir bağımlılık içermez. Bu, uygulamanın modülerliğini artırır ve birim testlerin daha etkili bir şekilde uygulanabilmesini sağlar. Ayrıca, iş katmanının kurumsal düzeyde iş kurallarını uygulaması, uygulamanın güvenilirliği ve tutarlılığı açısından kritiktir.

2.4 Data Access Layer (Veri Erişim Katmanı)

Veri Erişim Katmanı (DAL), bir yazılım uygulamasının veritabanı veya başka bir veri deposu ile iletişim kurduğu ve veri manipülasyonu işlemlerini gerçekleştirdiği katmandır. Veri Erişim Katmanı, genellikle iş katmanı ile veritabanı arasındaki arayüzü sağlar, bu sayede uygulamanın veri depolama ve geri çekme işlemlerini yönetir. İşte Veri Erişim Katmanının temel özellikleri şunlardır:

Veritabanına Bağlantı Yönetimi: Veri Erişim Katmanı, veritabanına bağlantı yönetimini sağlar. Bu, bağlantı açma, kapatma ve gerekirse yeniden kullanma gibi işlemleri içerir.

CRUD İşlemleri (Create, Read, Update, Delete): Veri Erişim Katmanı, veritabanındaki verilerin oluşturulması, okunması, güncellenmesi ve silinmesi gibi temel CRUD işlemlerini yönetir.

SQL veya ORM Kullanımı: Veri Erişim Katmanı, veritabanı ile iletişim kurmak için SQL sorgularını kullanabilir veya ORM (Object-Relational Mapping) araçları ile

entegre çalışabilir. Örneğin, Entity Framework veya Dapper gibi ORM araçları kullanılabilir.

Transaction Yönetimi: Veri Erişim Katmanı, işlemleri genellikle bir transaction içinde gruplar. Bu, birden çok veri manipülasyonu işleminin bir bütün olarak ele alınmasını sağlar.

Performans İyileştirmeleri: Veri Erişim Katmanı, veritabanı ile etkileşimde bulunurken performansı artırmak için çeşitli optimizasyon tekniklerini uygular. Bu, indeksleme, sorgu optimizasyonu gibi işlemleri içerir.

Veri Validasyonu: Gelen verilerin doğruluğunu kontrol eder ve belirli bir formata uyup uymadığını doğrular. Bu, veri bütünlüğünü sağlamak ve hatalı veri girişlerini önlemek için önemlidir.

Bağlantı Yönetimi ve Havuzlama: Veri Erişim Katmanı, veritabanına bağlantıların yönetimini üstlenir. Bağlantı havuzlama gibi teknikler kullanarak, veritabanı bağlantıları daha etkili bir şekilde kullanılabilir.

Stored Procedure ve Views Yönetimi: Veri Erişim Katmanı, veritabanındaki stored procedure'leri (saklanmış prosedürleri) veya views'leri yönetebilir. Bu, özellikle karmaşık sorguların yönetilmesi veya performans iyileştirmeleri için kullanılır.

Veri Modelinin Dönüştürülmesi: Veri Erişim Katmanı, veritabanındaki veriyi iş katmanının kullanabileceği nesne veya veri modellerine dönüştürür. Bu, uygulama içinde daha tutarlı bir veri modeli sağlar.

Hata Yönetimi ve Günlükleme: Veri Erişim Katmanı, veritabanı işlemlerinde oluşan hataları yönetir ve gerekirse hata günlüğü oluşturur. Bu, uygulamanın güvenilirliğini artırmak için önemlidir.

Veri Erişim Katmanı, uygulamanın veri tabanı ile etkileşimini yöneterek, iş katmanının veri manipülasyonu için uygun bir arayüz sunmasını sağlar. Bu, uygulamanın genel veri yönetimi süreçlerini düzenler ve genellikle iş katmanını veritabanı detaylarından izole eder, bu da modüler ve bakımı kolay bir kod tabanı oluşturur.

2.5 API Layer

API Katmanı (API Layer), bir yazılım uygulamasının dış dünyayla, diğer sistemlerle veya kullanıcılarla etkileşimde bulunmasını sağlayan bir katmandır. Bu katman, uygulamanın dış dünyaya sunulan programlama arabirimlerini (API - Application Programming Interface) ve servisleri içerir. API Katmanı, genellikle bir sunucu tarafı uygulamanın dış dünyaya açılan kapısı olarak düşünülür.API Katmanının temel özellikleri şunlardır:

HTTP(S) Protokolü ile İletişim: API Katmanı, genellikle HTTP veya HTTPS gibi iletişim protokollerini kullanarak dış dünyayla etkileşimde bulunur. Bu, uygulamanın sunduğu servislerin web üzerinden erişilebilir olmasını sağlar.

RESTful veya RPC Tabanlı Servisler: API Katmanı, genellikle RESTful (Representational State Transfer) veya RPC (Remote Procedure Call) tabanlı servisleri içerir. Bu servisler, belirli işlevselliği sağlamak için standart protokoller üzerinden çağrılabilir.

Endpoint'ler: API Katmanı, belirli işlevselliği temsil eden endpoint'leri içerir. Endpoint'ler, bir API'nin belirli bir servisine veya kaynağına erişim sağlayan URL'lerdir.

Veri Formatları: API Katmanı, genellikle belirli bir veri formatı üzerinden veri alışverişini yönetir. JSON veya XML gibi yaygın veri formatları kullanılabilir.

Güvenlik ve Kimlik Doğrulama: API Katmanı, dış dünyadan gelen istekleri güvenli bir şekilde yönetir. Bu, kimlik doğrulama ve yetkilendirme mekanizmalarını içerebilir.

Versioning (Sürümleme): API Katmanı, zaman içinde değişen API'lerle başa çıkabilmek için genellikle sürümleme stratejilerini destekler. Bu, eski ve yeni istemcilere uyumluluk sağlar.

Request ve Response İşleme: API Katmanı, gelen istekleri işler ve uygun servisleri çağırarak istemcilere yanıtlar. Bu işleme, gerekli validasyonları ve veri dönüşümlerini içerebilir.

Asenkron ve Senkron İşlemler: API Katmanı, istemcilere hem asenkron hem de senkron işlemleri destekleme yeteneği sağlayabilir. Bu, farklı kullanım senaryolarına uygun esneklik sağlar.

Dokümantasyon: API Katmanı, sunduğu servislerin kullanımını açıklamak için genellikle dokümantasyon içerir. Swagger veya OpenAPI gibi araçlar kullanılarak otomatik dokümantasyon oluşturmak yaygındır.

İzleme ve Günlükleme: API Katmanı, gelen istekleri izleyebilir, performans ölçümleri alabilir ve hata durumlarını günlüğe kaydedebilir. Bu, uygulamanın sağlığını izlemek ve sorunları hızlı bir şekilde tanımlamak için önemlidir.

API Katmanı, genellikle bir mikroservis mimarisinde veya daha büyük bir uygulama içinde belirli servislerin dış dünyaya açılması için kullanılır. Bu katman, uygulamanın ölçeklenebilirliğini artırabilir ve farklı platformlar arasında etkileşimi kolaylaştırabilir.

2.6 Varlık İlişki Diyagramı

Varlık İlişki Diyagramı (ERD), bir sistemde bulunan varlıkları (entity), bu varlıklar arasındaki ilişkileri ve varlıkların sahip olduğu nitelikleri gösteren bir görselleştirme aracıdır. ERD, veritabanı tasarımı sürecinde kullanılır ve genellikle Entity-Relationship Model (Varlık İlişki Modeli) notasyonuyla çizilir.

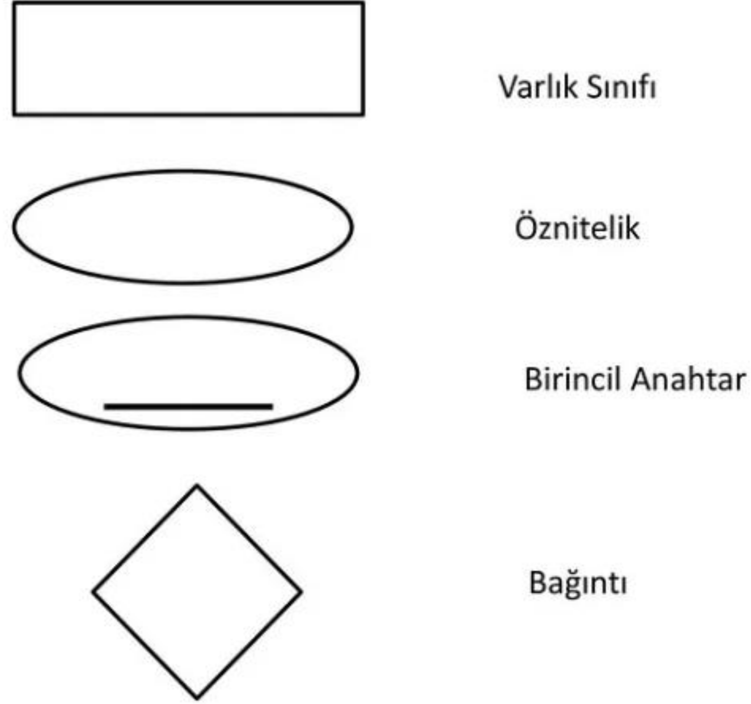
Varlık (Entity): Varlık, bir sistemde tanımlanabilir bir nesne, tablo veya sınıfı temsil eder. Örneğin, "Müşteri", "Ürün", "Sipariş" gibi iş süreçlerinde bulunan ana nesneler varlıkları temsil eder.

İlişki (Relationship): İlişki, varlıklar arasındaki bağlantıyı gösterir. Bu bağlantılar, genellikle "1'den çok", "çoktan çoka" veya "1'e 1" gibi kardinalite ifadeleri ile belirtilir. Örneğin, "Bir müşteri birden çok sipariş verebilir" şeklinde bir ilişki ifadesi.

Nitelik (Attribute): Varlıkların özelliklerini veya niteliklerini temsil eder. Örneğin, bir "Müşteri" varlığı için "ad", "soyad", "telefon numarası" gibi nitelikler bulunabilir.

Birincil Anahtar (Primary Key): Bir varlığın benzersiz bir şekilde tanımlanmasını sağlayan bir özelliktir. Her varlık birincil anahtara sahip olmalıdır. Örneğin, "Öğrenci" varlığında öğrenci numarası birincil anahtardır.

Yabancı Anahtar (Foreign Key): İki varlık arasındaki ilişkiyi sağlamak için kullanılır. Bir varlığın birincil anahtarı, diğer varlığın yabancı anahtarına referans olarak eklenir.



Şekil 2.2: Varlık İlişki Sembolleri

Varlık ilişki türleri, bir organizasyonun veritabanındaki veri yapısını anlamak ve modellemek için kullanılır. Bu modeller, veritabanı tasarımında kullanılacak tabloların, alanların ve ilişkilerin temelini oluşturur. Varlık ilişki türleri şunlardır;

Bir-Bir İlişkisi (One-to-One): Bu ilişki türünde, bir varlık türündeki her bir öğe, diğer varlık türündeki yalnızca bir öğe ile ilişkilidir. Örneğin, "Personel" varlık türü ile "Personel Bilgisi" varlık türü arasında bir-bir ilişkisi olabilir. Her personel kaydı, yalnızca bir personel bilgisi kaydına bağlıdır ve tersi de geçerlidir.

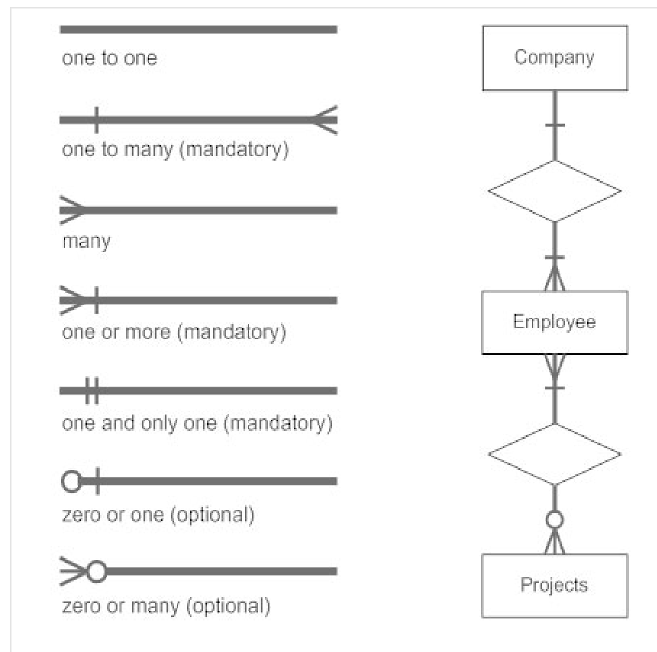
Bir-Çok İlişkisi (One-to-Many):Bu ilişki türünde, bir varlık türündeki her bir öğe, diğer varlık türündeki birden çok öğe ile ilişkilidir. Örneğin, "Departman" varlık türü

ile "Çalışan" varlık türü arasında bir-çok ilişkisi olabilir. Bir departman birden çok çalışana sahip olabilir, ancak bir çalışan sadece bir departmana ait olabilir.

Çok-Bir İlişkisi (Many-to-One): Bu ilişki türünde, bir varlık türündeki birden çok öğe, diğer varlık türündeki yalnızca bir öğe ile ilişkilidir. Örneğin, "Çalışan" varlık türü ile "Departman" varlık türü arasında bir-çok ilişkisi olabilir. Bir çalışan sadece bir departmana ait olabilir, ancak bir departman birden çok çalışana sahip olabilir.

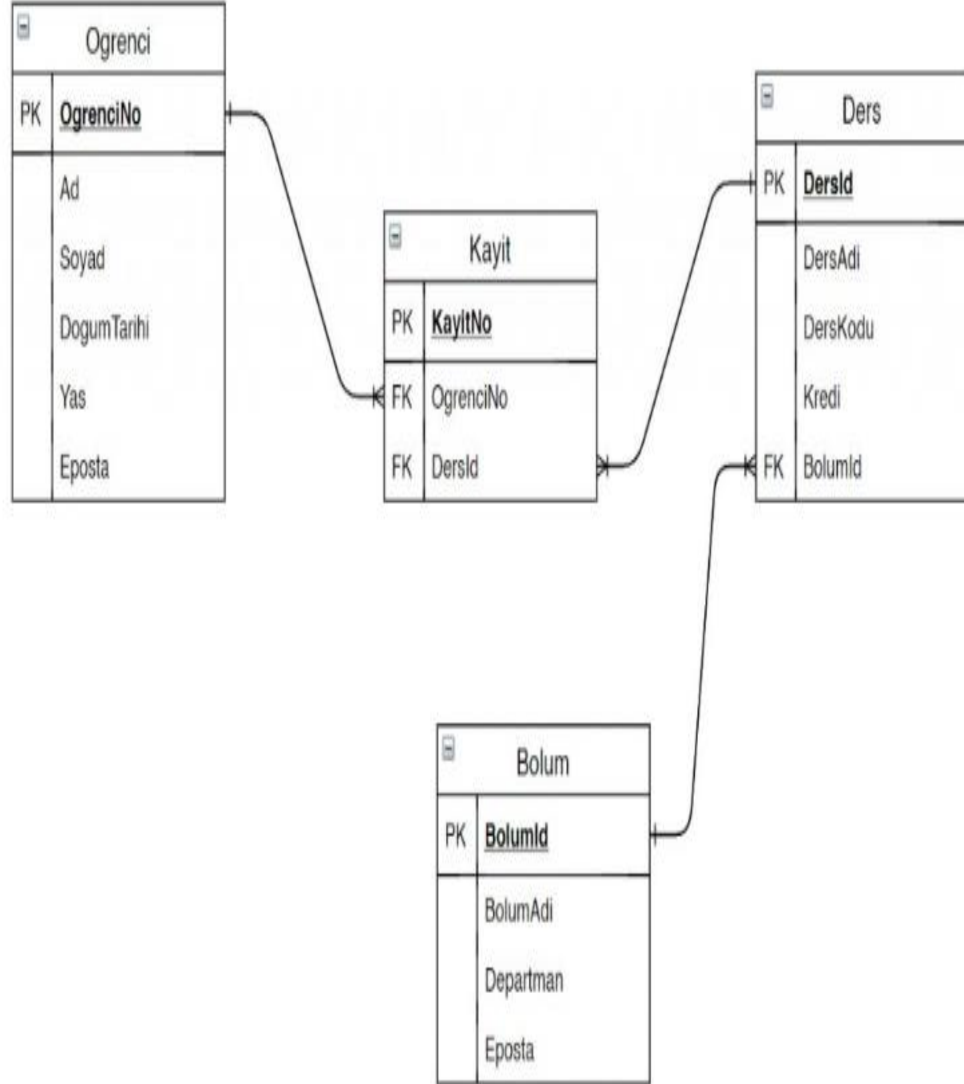
Çok-Çok İlişkisi (Many-to-Many): Bu ilişki türünde, bir varlık türündeki birden çok öğe, diğer varlık türündeki birden çok öğe ile ilişkilidir. Bu tür ilişkiler genellikle birleştirme tabloları veya ara tablolar kullanılarak ifade edilir. Örneğin, "Öğrenci" varlık türü ile "Ders" varlık türü arasında bir-çok ilişkisi olabilir. Bir öğrenci birden çok derse kaydolabilir ve bir ders birden çok öğrenciye sahip olabilir.

Zayıf Varlık İlişkisi (Weak Entity Relationship): Bu ilişki türü, bir varlığın başka bir varlık olmadan varlıkta kalma yeteneğini ifade eder. Zayıf varlık, kendi başına benzersiz bir şekilde tanımlanamaz ve bir güçlendirici varlık (owner entity) ile ilişkilidir. Bu güçlendirici varlık, zayıf varlığın kimliğini destekleyen birincil anahtarı sağlar.



Şekil 2.3: Varlık İlişki Türleri

Her bir ilişki türü, bir veritabanı tasarımının gereksinimlerine ve veri yapılarına uygun olarak seçilir ve tanımlanır. İlişkiler, veritabanındaki verilerin tutarlılığını ve bütünlüğünü sağlamak amacıyla dikkatlice tasarlanmalıdır.



Şekil 2.4: Varlık İlişki Diyagramı

ERD, genellikle çeşitli sembollerle çizilir ve bir veritabanının yapısını anlamak, tasarlamak ve iletmek için kullanılır.Varlık ilişki diyagramları, karmaşık sistemleri daha anlaşılır hale getirerek, projedeki ekip üyeleri ve paydaşlar arasında ortak bir anlayış oluşturulmasına yardımcı olur.

Bölüm 3

Araç Servis Otomasyon Sistemi

3.1 Projenin Mimari Yapısı

Projede kullanılan katmanlar,;

- **BusinessLayer**
  - Abstract
    - ICustomerCarService
    - ICustomerService
    - IReferenceDetailService
    - IReferenceService
    - IRoleService
    - IService
    - IServiceDetail
    - IWorkerService
  - Concrete
    - CustomerCarManager
    - CustomerManager
    - ReferenceDetailManager
    - ReferenceManager
    - RoleManager
    - ServiceDetailManager
    - ServiceManager
    - WorkerManager
- **CoreLayer**
- **DataAccessLayer**
  - Abstract
    - ICustomerDal

        - IGenericDal
        - ILogDal
        - IReferenceDal
        - IReferenceDetailDal
        - IRoleDal
        - IServiceDal
        - IServiceDetailDal
        - ISettingsDal
        - IWorkerDal
    - **Concrete**
        - Context
    - **EntityFramework**
        - EFCustomerCarRepository
        - EFCustomerRepository
        - EFReferenceDetailRepository
        - EFReferenceRepository
        - EFServiceDetailRepository
        - EFServiceRepository
        - EFSettingsRepository
        - EFWorkerRepository
    - **Migrations**
        - ContextModelSnapshot
    - **Repositories**
        - GenericRepository
- **EntityLayer**
    - Concrete
        - Customer
        - CustomerCar
        - Log
        - Reference
        - ReferenceDetail
        - Roles
        - Service

                - ServiceDetail
                - Settings
                - UserRoles
                - Worker
        - **Models**
                - CarService
                - CarServiceDetail
                - CustomerCarDetail
- **Vehicle Service Automation**
    - **Areas**
        - Admin
            - Controllers
            - Data
            - Models
            - Views
        - Customers
            - Controllers
            - Views
        - Workers
            - Controllers
            - Models
            - Views
    - **Controllers**
        - ErrorPage
        - Home
        - Login
    - **Models**
        - ErrorView
    - **Views**
        - ErrorPage
        - Home
        - Login
        - Shared

3.2 Projenin Katmanları

Veritabanı bağlantısı ve tabloların projeye eklenmesi için data access katmanında bir dbcontext oluşturulup, tüm katmanlarda bu method kullanılarak veritabanına erişim sağlanır.

```
namespace DataAccessLayer.Concrete
{
    public class Context : DbContext
    {
        protected override void OnConfiguring(DbContextOptionsBuilder dbContextOptionsBuilder)
        {
            dbContextOptionsBuilder.UseSqlServer("server=localhost;database=db_vsa; User Id=SA; Password=**********;");
        }


        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<UserRoles>(
                x =>
                {
                    x.HasNoKey();
                });
            base.OnModelCreating(modelBuilder);
            modelBuilder.Entity<CarServiceDetail>().HasNoKey();

        }

        public DbSet<Reference> References { get; set; }
        public DbSet<ReferenceDetail> ReferenceDetails { get; set; }
        public DbSet<Worker> Workers { get; set; }
        public DbSet<Customer> Customers { get; set; }
        public DbSet<Service> Services { get; set; }
        public DbSet<ServiceDetail> ServiceDetails { get; set; }
        public DbSet<CustomerCar> CustomerCars { get; set; }
        public DbSet<Roles> Roles { get; set; }
        public DbSet<UserRoles> UserRoles { get; set; }

        public virtual DbSet<CarServiceDetail> CarServiceDetails { get; set; }
    }
}
```

Şekil 3.1: Data Access Layer

Autherization işlemleri için logincontroller sınıfı oluşturuldu ve giriş yapan kullanıcının yetkisine göre yönlendirmeler yapıldı.

```
[HttpPost]
public async Task<IActionResult> Index(Worker worker)
{

    var value = context.Workers.FirstOrDefault(x => x.WorkerEmail == worker.WorkerEmail && x.WorkerPass == worker.WorkerPass);
    if (value != null)
    {
        var claims = new List<Claim>
        {
            new Claim(ClaimTypes.Name,worker.WorkerEmail)
        };
        var useridentity = new ClaimsIdentity(claims, "vsa");
        ClaimsPrincipal claimsPrincipal = new ClaimsPrincipal(useridentity);
        await HttpContext.SignInAsync(claimsPrincipal);

        var userArea = context.Workers.Where(x => x.WorkerEmail == worker.WorkerEmail).Select(y => y.WorkerDepartment).FirstOrDefault();

        if (userArea.ReferenceDetailName == "admin")
        {
            return RedirectToAction("Admin", "Admin", new { Areas = "Admin" });
        }
        else if (userArea.ReferenceDetailName == "service")
        {
            return RedirectToAction("WorkersService", "Workers", new { Areas = "Workers" });
        }
        else
        {
            return View();
        }
    }
    else if(value==null)
    {
        var customerQuery = context.Customers.Where(x => x.CustomerEmail == worker.WorkerEmail).FirstOrDefault();

        if(customerQuery!=null)
        {
            var claims = new List<Claim>
            {
                new Claim(ClaimTypes.Name,worker.WorkerEmail)
            };
            var useridentity = new ClaimsIdentity(claims, "vsa");
            ClaimsPrincipal claimsPrincipal = new ClaimsPrincipal(useridentity);
            await HttpContext.SignInAsync(claimsPrincipal);
            return RedirectToAction("Customers", "Customers", new { Areas = "Customer" });
        }
        else
        {
            return View();
        }
    }
```

Şekil 3.2: Login Controller

Çıkış süreci için signoutasync kullanıldı. Sisteme giriş yapan kullanıcının çıkış yapması durumunda signoutasync kullanılarak kullanıcının oturum bilgileri sistemden silinir.

```
public async Task<IActionResult> Logout()
{
    try
    {
        await HttpContext.SignOutAsync();
        return RedirectToAction("Index", "Login");
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

Şekil 3.3: Action Result

Business katmanında abstract klasörünün altında her veritabanı objesi için bir interface tanımlandı. Bu interfaceler;

- Ekleme
- Güncelleme
- Silme
- Tüm veriyi getirme
- Identity bilgisine göre tekil veri getirme

olarak tanımlandı.

```
namespace BusinessLayer.Abstract
{
    public interface ICustomerCarService
    {
        void AddCustomerCar(CustomerCar customerCar);
        void UpdateCustomerCar(CustomerCar customerCar);
        void DeleteCustomerCar(CustomerCar customerCar);
        List<CustomerCar> GetAllCustomerCar();
        CustomerCar GetCustomerCar(int id);
    }
}
```

Şekil 3.4: Business Layer

Business katmanında Concrete klasörünün altında Interfaceden kalıtım alan sınıflar oluşturuldu. Controllerdan bu sınıflar çağrılarak veritabanı işlemleri gerçekleşir.

```
namespace BusinessLayer.Concrete
{
    public class RoleManage : ICustomerCarService
    {
        ICustomerCarDal _customerCarDal;

        public RoleManage(ICustomerCarDal customerCarDal)
        {
            _customerCarDal = customerCarDal;
        }

        public void AddCustomerCar(CustomerCar customerCar)
        {
            _customerCarDal.Insert(customerCar);
        }

        public void DeleteCustomerCar(CustomerCar customerCar)
        {
            _customerCarDal.Delete(customerCar);
        }

        public List<CustomerCar> GetAllCustomerCar()
        {
            return _customerCarDal.GetAllList();
        }

        public CustomerCar GetCustomerCar(int id)
        {
            return _customerCarDal.GetById(id);
        }

        public void UpdateCustomerCar(CustomerCar customerCar)
        {
            _customerCarDal.Update(customerCar);
        }
    }
}
```

Şekil 3.5: Business Layer Concrete

DataAccess katmanında Migrations klasörünün altında veritabanında objelerin oluşturulması için builder kullanıldı. Proje code first olduğundan dolayı dataaccess katmanında bulunan concrete klasöründeki her sınıftaki değişiklikten sonra migrate edilip sonrasında veritabanı update komutu çalıştırılarak koddaki tüm değişikliklerin veritabanına yansıtılması sağlanır.

```
namespace DataAccessLayer.Migrations
{
    [DbContext(typeof(Context))]
    [Migration("20240107020154_servicedetail")]
    partial class servicedetail
    {
        protected override void BuildTargetModel(ModelBuilder modelBuilder)
        {
#pragma warning disable 612, 618
            modelBuilder
                .HasAnnotation("Relational:MaxIdentifierLength", 128)
                .HasAnnotation("ProductVersion", "5.0.9")
                .HasAnnotation("SqlServer:ValueGenerationStrategy", SqlServerValueGenerationStrategy.IdentityColumn);

            modelBuilder.Entity("EntityLayer.Concrete.Customer", b =>
                {
                    b.Property<int>("CustomerID")
                        .ValueGeneratedOnAdd()
                        .HasColumnType("int")
                        .HasAnnotation("SqlServer:ValueGenerationStrategy", SqlServerValueGenerationStrategy.IdentityColumn);

                    b.Property<DateTime>("Created")
                        .HasColumnType("datetime2");

                    b.Property<string>("Creator")
                        .HasColumnType("nvarchar(max)");

                    b.Property<string>("CustomerAddress")
                        .HasColumnType("nvarchar(max)");

                    b.Property<bool>("CustomerCallPerm")
                        .HasColumnType("bit");

                    b.Property<string>("CustomerEmail")
                        .HasColumnType("nvarchar(max)");

                    b.Property<bool>("CustomerEmailPerm")
                        .HasColumnType("bit");

                    b.Property<string>("CustomerName")
                        .HasColumnType("nvarchar(max)");

                    b.Property<string>("CustomerPhone")
                        .HasColumnType("nvarchar(max)");

                    b.Property<bool>("CustomerSmsPerm")
                        .HasColumnType("bit");

                    b.Property<string>("CustomerSurName")
                        .HasColumnType("nvarchar(max)");
```

Şekil 3.6: Data Access Layer Migrations

Projede generic repository kullanıldı. Tüm crud işlemleri ayrı ayrı repositorylerden değil de tek bir repositoryden yönetilmesi sağlandı. Bu yapıda işlemlerin tek bir yerden yapılması projede ilerleyen süreçlerin daha kolay yönetilmesi için bu yapı tercih edildi.

```
namespace DataAccessLayer.Repositories
{
    public class GenericRepository<T> : IGenericDal<T> where T : class
    {
        public void Delete(T t)
        {
            using var c = new Context();
            c.Remove(t);
            c.SaveChanges();
        }

        public List<T> GetAllList()
        {
            using var c = new Context();
            return c.Set<T>().ToList();
        }

        public T GetById(int id)
        {
            using var c = new Context();
            return c.Set<T>().Find(id);
        }

        public void Insert(T t)
        {
            using var c = new Context();
            c.Add(t);
            c.SaveChanges();
        }

        public void Update(T t)
        {
            using var c = new Context();
            c.Update(t);
            c.SaveChanges();
        }
    }
}
```

Şekil 3.7: Data Access Layer Repositories

Örnek olarak Customer tablosunun veritabanı modeli. Burada virtual olarak kullanılan properyler sadedce ekranda gösterilmesi için yapıldı.

```
namespace EntityLayer.Concrete
{
    public class Customer
    {
        public int CustomerID { get; set; }
        public string CustomerName { get; set; }
        public string CustomerSurName { get; set; }
        public string CustomerEmail { get; set; }
        public string CustomerPhone { get; set; }
        public string CustomerAddress { get; set; }
        public bool CustomerSmsPerm { get; set; }
        public bool CustomerEmailPerm { get; set; }
        public bool CustomerCallPerm { get; set; }
        public string Creator { get; set; }
        public string Modifier { get; set; }
        public DateTime Created { get; set; }
        public DateTime Modified { get; set; }
        public virtual string CustomerSmsPermDesc => CustomerSmsPerm ? "Evet" : "Hayır";
        public virtual string CustomerEmailPermDesc => CustomerEmailPerm ? "Evet" : "Hayır";
        public virtual string CustomerCallPermDesc => CustomerCallPerm ? "Evet" : "Hayır";
    }
```

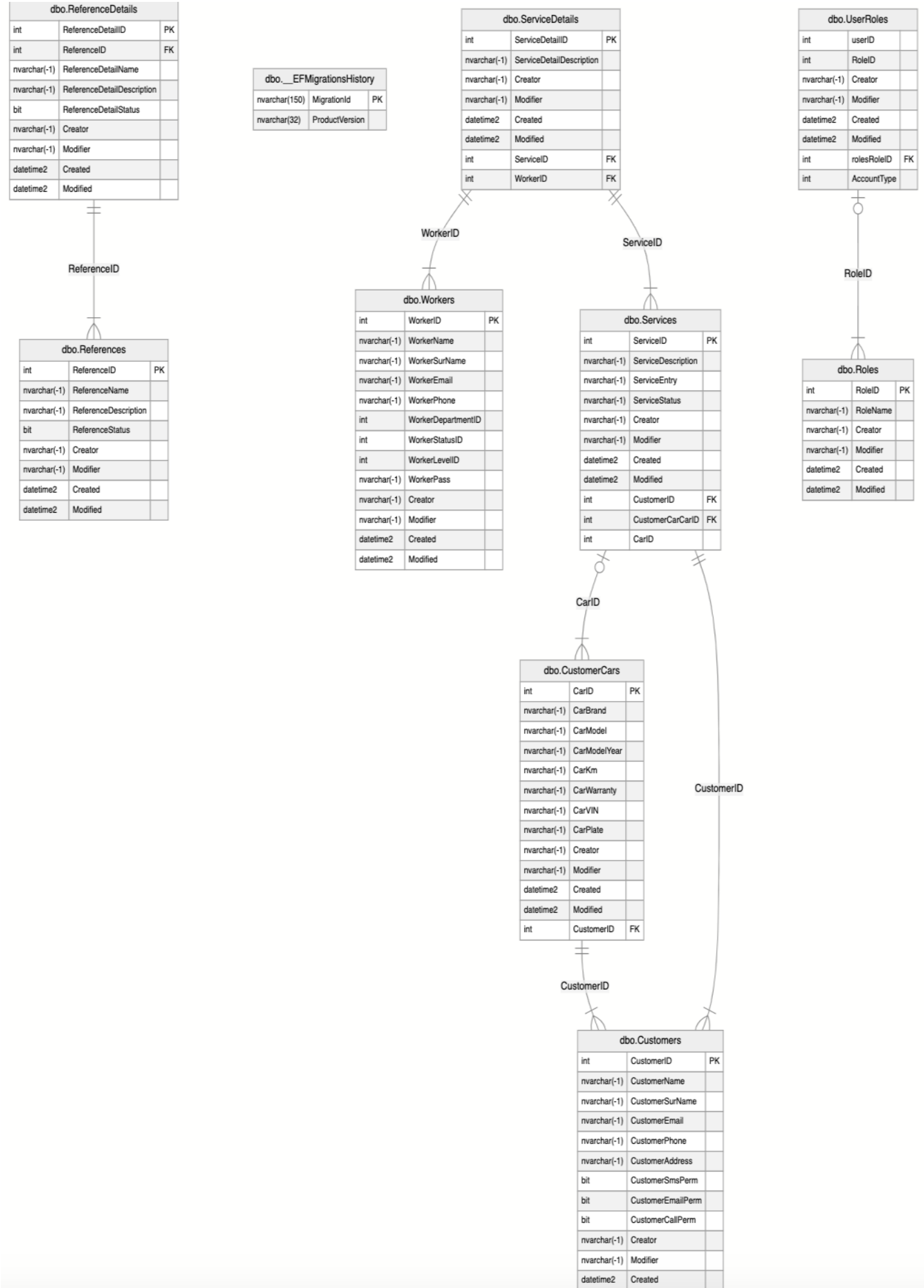
Şekil 3.8: Entity Layer Concrete

Birden fazla tablonun birleştirilip istenilen alanların bir model haline getirilmesi için modeller kullanıldı.

```
namespace Vehicle_Service_Automation.Areas.Workers
{
    public class CarService
    {
        public int ServiceID { get; set; }
        public string ServiceDescription { get; set; }
        public string CarBrand { get; set; }
        public string CarModel { get; set; }
        public string CarModelYear { get; set;}
        public string Modifier { get; set; }
        public string ServiceStatus { get; set; }
    }
}
```

Şekil 3.9: Car Service

Oluşturulan varlık ilişki diyagramıyla (ERD) sistemin varlıklarını (nesneler, tablolar, sınıflar) ve bu varlıklar arasındaki ilişkileri gösterilmiştir. Bu sayede projenin veri modeli daha açık bir şekilde anlaşılabilir ve geliştirme sürecinde veritabanı tasarımı daha etkili bir şekilde gerçekleştirilebilir.



Şekil 3.10: Varlık İlişki Diyagramı (ERD)

Bölüm 4

Sonuç

Bu çalışmamızda, araç otomasyon sistemi, MVC (Model-View-Controller) mimarisinin etkili bir şekilde uygulanmasıyla işletme verimliliği üzerinde olumlu bir etki sağlamıştır. Bu çerçevede, işletmelerin bu tip bir sistem entegrasyonu öncesinde düşünmeleri gereken kilit faktörler ve dikkat edilmesi gereken önemli noktalar şunlardır:

1. **Model**
   - Otomasyon sisteminin temel veri yapıları, iş süreçlerini detaylı bir şekilde analiz ederek oluşturulmalıdır.
   - Modelin doğru bir şekilde tasarlanması, işletmenin spesifik ihtiyaçlarına uyum sağlamak adına özel veri gereksinimlerini içermelidir.
2. **View (Görünüm)**
   - Kullanıcı arayüzü (UI) tasarımı, kullanıcıların sistemle etkileşimini kolaylaştırmalı ve kullanıcı deneyimini artırmalıdır.
   - Görünüm katmanı, işletme süreçlerinin anlaşılabilir ve kullanıcı dostu bir arayüzle temsil edilmesini sağlamalıdır.
3. **Controller (Denetleyici)**
   - Denetleyici, işletme mantığını yönetmeli ve model ile görünüm arasındaki etkileşimi düzenlemelidir.
   - İş süreçlerinin mantıksal akışını kontrol etmek, sistem performansını artırmak adına dikkatlice planlanmalıdır.

4. **Özelleştirme ve Uyumluluk:**

    - MVC mimarisinin sunduğu özelleştirme imkanları kullanılarak, sistem işletmenin spesifik gereksinimlerine uyumlu hale getirilmelidir.
    - Uyumluluk, sistemdeki değişikliklerin sorunsuz bir şekilde entegre edilebilmesi için sürekli gözetilmelidir.

5. **Personel Eğitimi ve Kabul Süreci:**

    - Kullanıcıların sistemi etkili bir şekilde kullanabilmesi için düzenli eğitim programları düzenlenmeli ve sistem kullanımı konusunda bilinçlendirme sağlanmalıdır.
    - Personel, yeni sistemle ilgili olarak benimsemeye yönelik bir kabul süreci ile desteklenmelidir.

Bu değerlendirmeler sonucunda, araç otomasyon sistemi için başarılı bir MVC mimarisi uygulamasının işletme verimliliği üzerindeki olumlu etkisini görülmektedir.

Kaynaklar

Mücerret, R. (2019). Yazılım Mimarisi: Tasarım Desenleri ve İyi Uygulama İlkeleri. Birsen Yayınevi.

Kaya, C. (2019). ASP.NET Core MVC ve SignalR ile Gerçek Zamanlı Uygulama Geliştirme. Seçkin Yayıncılık.

Varol, E. (2020). ASP.NET Core 3.1 ile MVC ve Razor Pages. Umuttepe Yayıncılık.

Toprak, O. (2021). ASP.NET Core MVC 5 ve C# ile Web Programlama. Kodlab Yayınları.