Restoran Yönetim Sistemi

Staj Projesi Teknik Dokümantasyonu

Onur Dalgıç

Bihl ve Wiedemann Otomasyon Sanayi

ve Ticaret Limited Sirketi

Yazılım Departmanı Stajı

Eskişehir Osmangazi Üniversitesi

Bilgisayar Mühendisliği

Ocak – Şubat 2026

İÇİNDEKİLER

1. GİRİŞ

Bu dokümanı hazırlama amacım, staj sürecim boyunca geliştirdiğim **RestaurantWeb** projesinde kullandığım ve büyük bir kısmını bu proje kapsamında ilk kez deneyimlediğim teknolojileri ve mimari yaklaşımları bilinçli bir şekilde kayıt altına almaktır.

Proje süresince, daha önce teorik olarak bildiğim ya da hiç kullanmadığım birçok kavramı **uygulama seviyesinde öğrenme** fırsatı buldum. Bu doküman, bu öğrenme sürecinin bir özeti olmanın yanı sıra, aldığım teknik kararların arkasındaki nedenleri açıkça ortaya koymayı hedeflemektedir.

RestaurantWeb, ASP.NET Core MVC ve PostgreSQL kullanılarak geliştirilmiş bir Restoran Yönetim Sistemi (POS) uygulamasıdır. Ancak proje geliştirilirken odağım, yalnızca "çalışan" bir sistem üretmekten ziyade; gerçek bir restoran ortamında karşılaşılabilecek yetkilendirme, eşzamanlılık (concurrency), veri tutarlılığı ve raporlama gibi problemleri anlamak ve bunlara mühendislik bakış açısıyla çözümler üretmek olmuştur. Bu nedenle projede kullanılan mimari yapı, basit CRUD işlemlerinin ötesine geçecek şekilde tasarlanmıştır.

Bu dokümanda, projede neden belirli bir mimari yaklaşımı tercih ettiğimi, hangi problemleri çözmek için hangi teknikleri kullandığımı ve bu kararların sisteme olan etkilerini kendi deneyimlerim üzerinden anlatıyorum. Controller, Service ve Repository katmanlarının ayrımı, transaction yönetimi, iş kurallarının konumlandırılması ve veritabanı seviyesinde alınan önlemler; yalnızca "nasıl yapıldığı" ile değil, aynı zamanda "neden bu şekilde yapıldığı" ile birlikte ele alınmaktadır. Amacım, projeyi okuyan bir kişinin yalnızca kodu değil, kodun arkasındaki **düşünce yapısını** da net bir şekilde anlayabilmesidir.

2. KATMANLI MİMARİ YAKLAŞIMI

İlk bakışta bir restoran uygulaması için temel CRUD işlemlerinin yeterli olacağı düşünülebilir. Ancak bu projeyi geliştirmekteki temel amacım, yalnızca çalışan bir uygulama ortaya koymak değil; daha geniş ölçekli bir yazılım projesinde mimari kararların nasıl alındığını, bu kararların sistem davranışını nasıl etkilediğini ve gerçek hayatta karşılaşılan problemlerle nasıl başa çıkıldığını **öğrenmekti.** Bu nedenle proje sürecinde, uygulamayı bilinçli olarak genişletmekten ve daha önce kullanmadığım mimari yaklaşımları ve teknikleri denemekten çekinmedim.

Proje geliştikçe, basit görünen işlemlerin dahi aslında birden fazla sorumluluğu beraberinde getirdiğini fark ettim. Örneğin bir siparişin kapatılması yalnızca tek bir kayıt güncellemesi değil;

- ödeme bilgisinin eklenmesi,
- sipariş durumunun değiştirilmesi,
- masa durumunun güncellenmesi,
- bazı durumlarda log ve rapor verilerinin etkilenmesi

gibi birden fazla adımı içermektedir. Bu tür işlemler arttıkça, tüm mantığın tek bir katmanda toplanmasının kodun okunabilirliğini ve sürdürülebilirliğini ciddi şekilde zorlaştırdığını gözlemledim.

Bu noktada, sorumlulukların net bir şekilde ayrılmasını sağlamak amacıyla Controller, Service ve Repository katmanlarından oluşan **katmanlı mimari** yaklaşımını benimsedim. Bu yaklaşım sayesinde:

- kullanıcıdan gelen isteklerin yönetimi,
- iş kurallarının uygulanması
- veritabanı erişimi

birbirinden ayrılarak daha kontrol edilebilir bir yapı oluşturuldu. Katmanlı mimari, projede kullanılan her teknolojinin ve her kuralın belirli bir bağlam içerisinde konumlandırılmasını sağlayarak, hem geliştirme sürecini hem de ileride yapılacak değişiklikleri daha öngörülebilir hale getirdi.

3. KATMANLARIN GENEL TANITIMI

Bu projede uygulama mimarisi;

- kullanıcıdan gelen isteklerin yönetimi,
- iş kurallarının uygulanması,
- veritabanı erişimi

olmak üzere üç temel sorumluluk alanına ayrılmıştır. Bu ayrım, sistemin hem okunabilirliğini artırmak hem de her bir katmanın yalnızca kendi problemiyle ilgilenmesini sağlamak amacıyla yapılmıştır. Böylece proje büyüdükçe, bir değişikliğin sistemin tamamını etkilemesi yerine yalnızca ilgili katmanda sınırlandırılması hedeflenmiştir.

En üstte yer alan **Controller katmanı**, uygulamanın dış dünya ile temas ettiği noktadır. Bu katman;

- tarayıcıdan gelen HTTP isteklerini karşılar,
- gelen verileri alır,
- yapılacak işlemi tanımlayarak alt katmanlara iletir.

Controller'ın temel sorumluluğu, işin nasıl yapılacağını belirlemekten ziyade, hangi işin yapılacağını ifade etmek ve elde edilen sonucu kullanıcı arayüzüne uygun şekilde sunmaktır. Bu yaklaşım sayesinde controller sınıfları mümkün olduğunca sade tutulmuş, karmaşık iş kurallarının bu katmanda yer almasının önüne geçilmiştir.

Controller'ın altında konumlanan **Service katmanı**, uygulamanın iş mantığının merkezini oluşturur. Gerçek dünyadaki restoran operasyonlarını temsil eden kurallar, durum geçişleri ve doğrulamalar bu katmanda ele alınır.

- Sipariş kapatma,
- rezervasyon oluşturma,
- masa yönetimi gibi işlemler,

birden fazla adımı ve kontrolü içerdiğinden, bu işlemlerin tek bir merkezden ve tutarlı bir şekilde yönetilmesi hedeflenmiştir. Service katmanı, bu yönüyle uygulamanın "nasıl davranması gerektiğini" tanımlayan katmandır.

En alt seviyede bulunan **Repository katmanı** ise veritabanı ile doğrudan iletişim kuran yapıdır. Bu katman;

- SQL sorgularının yazılması ve çalıştırılması,
- elde edilen verilerin uygun veri modellerine dönüştürülmesi

gibi sorumlulukları üstlenir. Repository katmanı sayesinde, verinin nasıl saklandığı veya nasıl alındığı bilgisi üst katmanlardan soyutlanmış olur. Böylece iş kuralları ile veritabanı erişimi birbirinden ayrılarak daha modüler bir yapı elde edilmiştir.

Bu katmanlı yapı, projede kullanılan her bileşenin belirli bir sınır içerisinde hareket etmesini sağlar. Controller, Service ve Repository katmanlarının her biri kendi sorumluluğuna odaklanarak, sistemin hem **geliştirme sürecinde** hem de **bakım aşamasında** daha öngörülebilir ve yönetilebilir olmasına katkı sunar.

3.1. Controller Katmanı

Controller katmanı, uygulamanın kullanıcı ile ilk temas kurduğu noktadır. Tarayıcıdan gelen her HTTP isteği, ASP.NET Core'un routing mekanizması aracılığıyla ilgili controller ve action metoduna yönlendirilir. Bu nedenle Controller'ları, sistemin "giriş kapısı" olarak konumlandırdım. Bu katmandaki temel amacım, kullanıcıdan gelen isteği doğru şekilde karşılamak ve bu isteğin işlenmesini uygun katmana devretmek olmuştur.

Projeyi geliştirirken Controller katmanının sorumluluklarını bilinçli olarak sınırladım. Controller'lar;

- form veya query üzerinden gelen verileri almak,
- gerekli temel doğrulamaları yapmak,
- kullanıcının bu işlemi gerçekleştirmeye yetkili olup olmadığını kontrol etmekle

ilgilenmektedir. Bunun dışında, işin nasıl yapılacağına dair kararların bu katmanda **verilmemesine** özellikle dikkat ettim. Bu yaklaşım, controller sınıflarının sade ve okunabilir kalmasını sağladı.

Controller katmanında alınan önemli tasarım kararlarından biri, **iş kurallarının bu katmanda yer almamasıdır.**

Örneğin bir siparişin kapatılması sırasında hangi tabloların güncelleneceği, hangi durum kontrollerinin yapılacağı veya işlem sırasında hata oluşursa nasıl bir yol izleneceği gibi kararlar controller içinde verilmemektedir. Bu tür kurallar **Service** katmanına bırakılmıştır. Controller yalnızca "siparişi kapat" isteğini başlatır ve bu isteğin sonucunu kullanıcı arayüzüne uygun bir şekilde yansıtır.

Ayrıca Controller'lar, uygulamanın HTTP ve UI dünyasına özgü detaylarını yönetmektedir. PRG (Post/Redirect/Get) yaklaşımı, TempData üzerinden kullanıcıya gösterilen bilgilendirme mesajları ve ViewModel'lerin hazırlanması bu katmanda ele alınmıştır. Bu sayede alt katmanlar, HTTP protokolü veya kullanıcı arayüzü gibi detaylardan tamamen soyutlanmıştır.

Bu yaklaşımın sağladığı **en önemli kazanım**, kodun okunabilirliği ve bakım kolaylığı olmuştur. Bir controller dosyasına bakıldığında, sistemin iş kurallarından ziyade kullanıcı akışının görülebilmesi hedeflenmiştir. Böylece ileride yeni bir ekran eklenmesi, mevcut bir akışın değiştirilmesi veya farklı bir arayüz (örneğin Web API) eklenmesi durumunda, iş mantığına **minimum müdahale** ile ilerlenebilecek bir yapı elde edilmiştir.

3.2. Service Katmanı

Service katmanı, bu projede uygulamanın iş mantığının merkezinde yer almaktadır. Projeyi geliştirirken en çok odaklandığım konulardan biri, sistemin her durumda **doğru ve tutarlı davranmasını** sağlamaktı. Bu nedenle, gerçek hayattaki restoran operasyonlarını temsil eden tüm iş kurallarının tek bir merkezde toplanması gerektiğini fark ettim. Service katmanı, bu ihtiyacın bir sonucu olarak konumlandırılmıştır.

Controller ve Repository katmanları daha çok mekanik sorumluluklar üstlenmektedir. Controller, kullanıcıdan gelen isteği alır ve ilgili işlemi başlatır; Repository ise veriyi okur veya yazar. Ancak;

- sistemin hangi koşullarda neye izin vereceği,
- hangi adımların hangi sırayla çalışacağı,
- bir işlem sırasında nasıl kararlar alınacağı

Service katmanında belirlenmektedir. Bu nedenle Service katmanını, uygulamanın davranışını tanımlayan ve yöneten **sistemin beyni** olarak ele aldım.

Projede ele alınan birçok işlem, ilk bakışta basit gibi görünse de arka planda birden fazla kontrol ve adımı içermektedir. Örneğin bir siparişin kapatılması işlemi;

- siparişin mevcut durumunun doğrulanması,
- ödemenin eklenmesi, masa durumunun güncellenmesi,
- bazı senaryolarda log ve rapor verilerinin etkilenmesi

gibi bir zincirden oluşmaktadır. Bu tür işlemlerin controller katmanında veya dağınık şekilde ele alınması, kodun zamanla karmaşıklaşmasına ve hata riskinin artmasına neden olmaktadır. Bu nedenle tüm bu kurallar Service katmanında merkezi olarak ele alınmıştır.

Service katmanının temel sorumluluğu, iş kurallarını uygulamak ve sistemin tutarlı davranmasını sağlamaktır. "Kapalı bir sipariş tekrar kapatılamaz", "aynı masa için aynı anda birden fazla aktif sipariş oluşturulamaz" veya "yetkisi olmayan bir kullanıcı belirli işlemleri

gerçekleştiremez” gibi kurallar bu katmanda tanımlanmıştır. Bu yaklaşım sayesinde, aynı iş kuralının farklı controller’lar veya farklı akışlar tarafından tekrar tekrar yazılmasının önüne geçilmiştir.

Bu katmanda ele aldığım önemli kavramlardan biri **idempotent davranıştır**. İdempotent davranış, aynı isteğin birden fazla kez gönderilmesi durumunda sistemin tutarsız bir duruma düşmemesini ifade eder. POS sistemlerinde bu durum özellikle kritiktir; kullanıcıların bir butona birden fazla kez basması veya istemcinin isteği tekrar göndermesi gibi senaryolar sıkça yaşanabilmektedir. Örneğin “siparişi kapat” işlemi ikinci kez tetiklendiğinde, sistemin hataya düşmesi veya veriyi bozması yerine, bu siparişin zaten kapalı olduğunu güvenli bir şekilde tespit edip işlemi etkisiz biçimde sonlandırması hedeflenmiştir. Bu tür kontroller Service katmanında merkezi olarak ele alınmıştır.

Service katmanında ele aldığım bir diğer önemli konu **transaction yönetimidir.** Birden fazla tabloyu etkileyen işlemlerde, tüm adımların **tek bir bütün olarak** ele alınması gerektiğini öğrendim. Service katmanında transaction sınırları belirlenerek, işlemin herhangi bir adımında hata oluşması durumunda tüm değişikliklerin geri alınması sağlanmıştır. Bu yaklaşım, özellikle finansal veriler ve sipariş kayıtları söz konusu olduğunda veri tutarlılığını korumak açısından kritik bir rol oynamaktadır.

Service katmanı aynı zamanda **eşzamanlılık (concurrency)** problemlerinin ele alındığı yerdir. Gerçek bir restoran ortamında, birden fazla personelin aynı anda aynı masa veya sipariş üzerinde işlem yapması mümkündür. Bu tür senaryolarda oluşabilecek yarış durumlarını önlemek amacıyla, Service katmanında gerekli kontroller yapılmış ve veritabanı seviyesindeki kilitleme ve kısıt mekanizmaları ile birlikte çalışacak şekilde bir yapı kurulmuştur. Bu sayede, uygulamanın beklenmeyen durumlar üretmesinin önüne geçilmiştir. Dokümanın ilerleyen kısımlarında bu çözümlerden detaylı olarak bahsedilecektir.

Service katmanından dönen sonuçlar, projede standart bir yapı oluşturmak amacıyla **OperationResult<T> modeli** ile ifade edilmiştir. Bu yapı sayesinde, bir işlemin başarılı olup olmadığı, varsa hata mesajı ve işlem sonucunda elde edilen veri tek bir format üzerinden üst katmanlara iletilmiştir. Bu yaklaşım, hata yönetiminin tutarlı olmasını sağladığı gibi, controller katmanında daha sade ve okunabilir bir akış kurulmasına da katkı sağlamıştır.

Sonuç olarak Service katmanı, projede yalnızca bir “ara katman” değil; sistemin nasıl davranacağını tanımlayan ve yöneten **ana bileşen** olarak ele alınmıştır. Bu katman sayesinde, iş kuralları net bir şekilde ayrılmış, tekrar eden kodlar azaltılmış ve uygulamanın ileride genişletilmesine uygun bir temel oluşturulmuştur.

3.3. Repository Katmanı

Repository katmanı, projede veritabanı ile doğrudan iletişim kuran ve veri erişiminden sorumlu olan katman olarak konumlandırılmıştır. Bu katmanı tasarlarken temel hedefim, verinin nasıl saklandığı veya nasıl alındığı bilgisini üst katmanlardan tamamen soyutlamak olmuştur. Böylece iş kuralları ile veritabanı erişimi arasındaki sınır net bir şekilde çizilmiştir.

Projede **PostgreSQL** ile iletişim **Npgsql** kütüphanesi aracılığıyla sağlanmaktadır. Repository katmanı;

- SQL sorgularının yazılması,
- parametrik olarak çalıştırılması,
- elde edilen sonuçların uygulama içerisinde kullanılabilecek veri modellerine dönüştürülmesi

gibi sorumlulukları üstlenmektedir. Bu yaklaşım sayesinde, SQL ifadeleri sistemin tek bir bölümünde toplanmış ve farklı katmanlara dağılması engellenmiştir.

Repository katmanında bilinçli olarak iş kuralı bulundurulmamasına dikkat edilmiştir. Örneğin bir rezervasyonun çakışıp çakışmadığına karar vermek veya bir siparişin hangi durumda kapatılabileceğini belirlemek bu katmanın sorumluluğu değildir. Repository, yalnızca kendisinden istenen veriyi doğru ve güvenilir bir şekilde sağlamakla yükümlüdür. Bu nedenle bu katman, mümkün olduğunca sade ve öngörülebilir tutulmuştur.

Bu katmanda ele alınan bir diğer önemli konu, veritabanı sorgularının güvenli ve tekrar edilebilir olmasıdır. Parametrik sorgular kullanılarak SQL injection gibi güvenlik risklerinin önüne geçilmiştir. Ayrıca, raporlama ve listeleme gibi işlemler için yazılan sorguların performanslı çalışabilmesi adına, veritabanı tarafındaki **indeks** ve **kısıtlar** göz önünde bulundurularak tasarım yapılmıştır. Bu sayede, verinin doğru şekilde alınmasının yanı sıra, sistemin performansının da korunması hedeflenmiştir.

Repository katmanı, Service katmanı ile net bir sözleşme üzerinden çalışmaktadır. Service katmanı “hangi veriye ihtiyaç duyulduğunu” ifade ederken, Repository katmanı bu verinin “nasıl elde edileceğini” belirler. Bu ayrım sayesinde, ileride veritabanı yapısında veya sorgularda yapılacak bir değişikliğin, iş kurallarını doğrudan etkilemeden yönetilebilmesi mümkün hale gelmiştir.

Sonuç olarak Repository katmanı, projede sessiz ama kritik bir rol üstlenmektedir. Doğru tasarlanmış bir repository yapısı, üst katmanların veritabanı detaylarıyla ilgilenmeden iş mantığına odaklanmasını sağlar. Bu yaklaşım, projenin okunabilirliğini artırdığı gibi, bakım ve geliştirme süreçlerinde de önemli bir avantaj sunmuştur.

3.4. Katmanlı Mimari'nin Genel Değerlendirmesi (Artılar ve Bedeller)

Katmanlı mimarinin en önemli katkısı, sorumlulukların net bir şekilde ayrılmasını sağlamasıdır. Controller katmanında kullanıcı akışı ve HTTP detayları ele alınırken, iş kuralları Service katmanında toplanmış, veritabanı erişimi ise Repository katmanında izole edilmiştir.

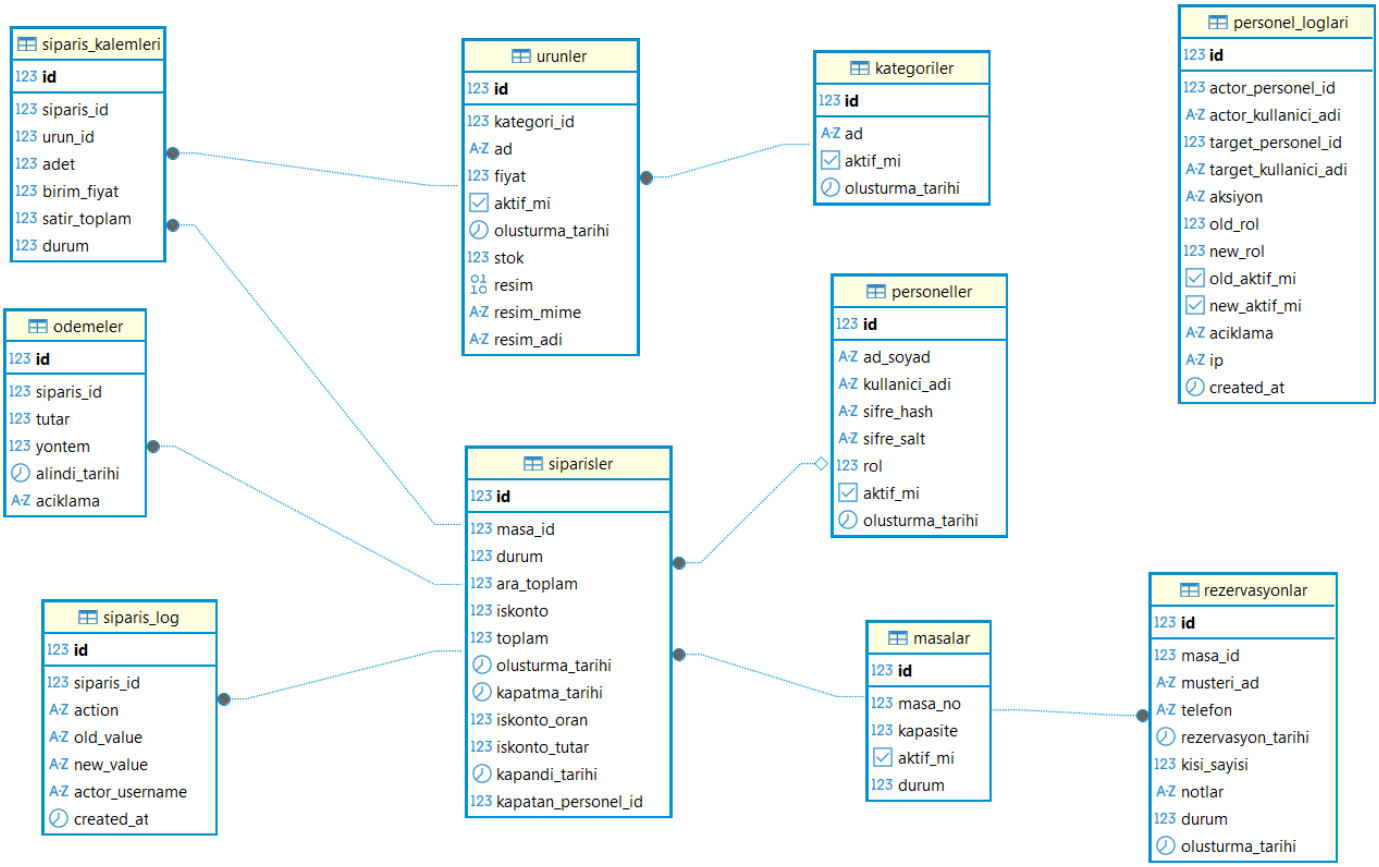
İş kurallarının tek bir merkezde toplanması, katmanlı mimarinin projeye sağladığı bir diğer önemli kazanımlardan biridir. Sipariş kapatma, rezervasyon oluşturma ve masa yönetimi gibi birden fazla adım içeren işlemler, Service katmanında ele alınarak sistemin tutarlı davranması sağlanmıştır. Bu sayede, aynı kuralların farklı yerlerde tekrar edilmesinin önüne geçilmiş ve hata riski azaltılmıştır.

Katmanlı mimari, iş kurallarının kullanıcı arayüzünden bağımsız olarak ele alınabilmesini de mümkün kılmıştır. Service katmanında toplanan bu yapı sayesinde, ileride farklı bir arayüz veya API eklenmesi durumunda mevcut iş mantığının yeniden kullanılabilir olması hedeflenmiştir.

Bununla birlikte, bu yaklaşımın belirli bedelleri bulunmaktadır. En belirgin bedel, dosya ve sınıf sayısındaki artış ve buna bağlı olarak takip maliyetinin yükselmesidir. Basit işlemler için dahi birden fazla katmanda düzenleme yapılması gerekebilmekte; bu durum özellikle küçük ölçekli işlemlerde gereksiz karmaşıklık hissi yaratabilmektedir.

Ancak proje ilerledikçe, bu maliyetin yerini düzenli bir yapı ve öngörülebilirlik aldığını gözlemledim. İş kurallarının yoğun olduğu ve hata toleransının düşük olduğu bir POS uygulaması için, katmanlı mimarinin getirdiği ek yük kabul edilebilir bir bedel olarak değerlendirilmiştir. Bu yaklaşım sayesinde, projenin yalnızca mevcut ihtiyaçlara değil, ileride yapılacak geliştirmelere de uygun bir temel üzerine kurulduğunu düşünüyorum.

4. VERİ MODELİ (ER DİYAGRAMI)



4.1. Sipariş Yönetimi – Örnek Akış ve Tasarım Kararları

Bu bölüm, RestaurantWeb uygulamasında sipariş yaşam döngüsünün uçtan uca nasıl çalıştığını ve bu akışın arkasındaki tasarım kararlarını (transaction sınırları, concurrency kontrolü, idempotency yaklaşımı, toplam hesaplama stratejisi, loglama) açıklar. Amaç; hem operasyonel (garson/kasa) kullanım senaryolarını desteklemek hem de veri bütünlüğünü PostgreSQL seviyesinde garanti altına almaktır.

4.1.1 Durum Modelleri (State Machine)

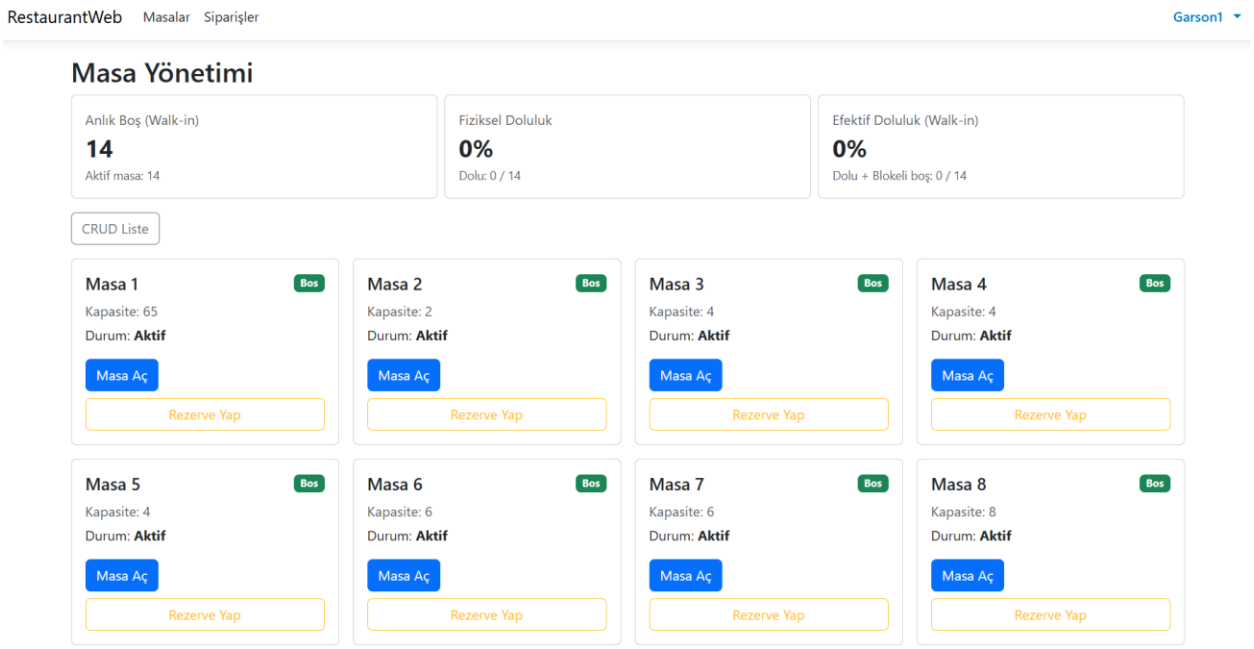
Sipariş, masa ve rezervasyon yönetimi; kısa ve deterministik state'ler ile modellenmiştir:

- **Sipariş Durumu (SiparisDurumu)**
  - Acik = 0
  - Kapali = 1
  - Iptal = 2

- **Sipariş Kalemi Durumu (SiparisKalemDurumu)**
  - Bekliyor = 0
  - Hazirlaniyor = 1
  - Hazir = 2
  - ServiseCikti = 3
- **Rezervasyon Durumu (RezervasyonDurumu)**
  - Aktif = 0
  - Iptal = 1
  - Kullanildi = 2
- **Masa Durumu (MasaDurumu)**
  - Bos = 0
  - Dolu = 1
  - Rezerve = 2

Bu state'ler, UI'daki "masa board" görünümü ile backend iş kurallarının ortak dilidir. Özellikle Masalar.Board ekranında "rezervasyon blokesi" gibi efektif durumlar hesaplanırken, tablo state'i ile rezervasyon state'i birlikte değerlendirilir.

4.1.2. Örnek Uçtan Uca Akış (Garson → Sipariş → Kasa)



Aşağıdaki senaryo, sistemin en tipik kullanımını temsil eder:

Adım A — Masa açma (sipariş oluşturma garantisi)

Garson, masa board ekranından ilgili masaya "Aç" işlemi uygular.

- **Controller:** MasalarController.Open(id)
- **Service:** MasaService.EnsureOpenTable(masaId, now)
- **Repository:** SiparisRepository.EnsureOpenOrderForTable(conn, tx, masaId)

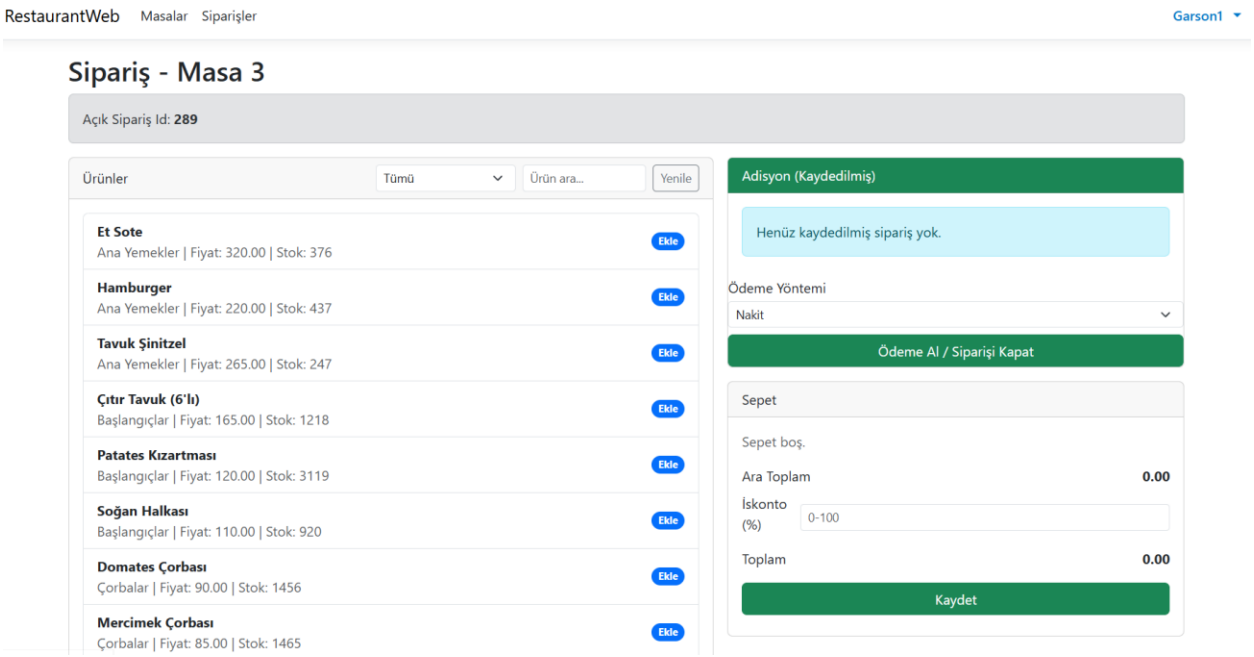
Bu adımın temel hedefi şudur:

- Masa aktif mi?
- Masa rezervasyon nedeniyle blokeli mi? (iş kuralı)
- Aynı masa için tek bir **açık sipariş** garantisi var mı? (DB + transaction)

EnsureOpenOrderForTable içinde masa satırı FOR UPDATE ile kilitlenir. Böylece aynı masaya eşzamanlı "aç" istekleri geldiğinde, **tek bir açık sipariş** üretilecek şekilde seri hale getirilir. Sipariş yoksa insert edilir; varsa mevcut açık sipariş id'si döndürülür. Son olarak masa Dolu durumuna alınır.

Bu tasarım, "sipariş alma ekranına gidildiğinde sipariş id'si hazır olsun" hedefini sağlar ve akışı deterministik hale getirir.

Adım B — Sipariş alma ekranına giriş (read ağırlıklı)

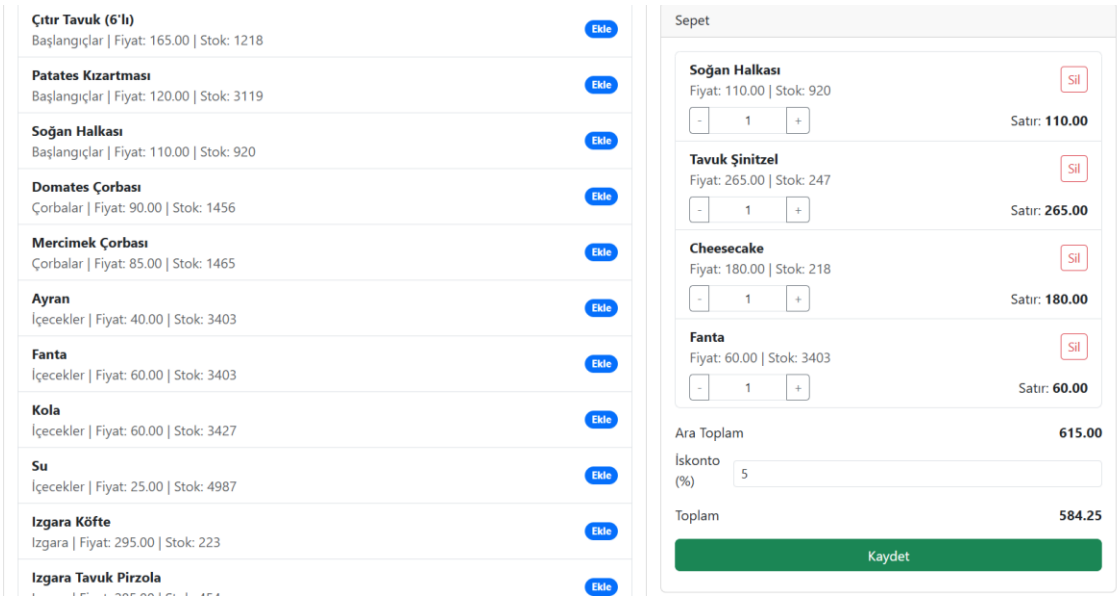


Masa açıldıktan sonra sistem Siparisler/Take?masaId=... ekranına yönlenir.

- Açık sipariş id'si GetOpenOrderId(masaId) ile alınır.
- Adisyon (header + kalemler) GetSiparisAdisyon(siparisId) ile çekilir.

Bu adımda yazma yoktur; temel amaç UI'yı oluşturmak ve mevcut adisyonu göstermek olduğu için okuma işlemleri bağımsız connection ile yapılır. Kritik yazma akışları ise transaction içinde yönetilir (bkz. 5.3).

Adım C — Ürün ekleme (kritik yazma akışı)



Garson sepete ürün ekleyip “Kaydet” dediğinde:

- **Controller:** SiparislerController.Submit
- **Service:** SiparisService.SubmitOrder (connection + tx açar)
- **Repository:** SiparisRepository.SubmitOrder(conn, tx, siparisId, items)

Bu işlem “kritik” kabul edilir, çünkü aynı anda:

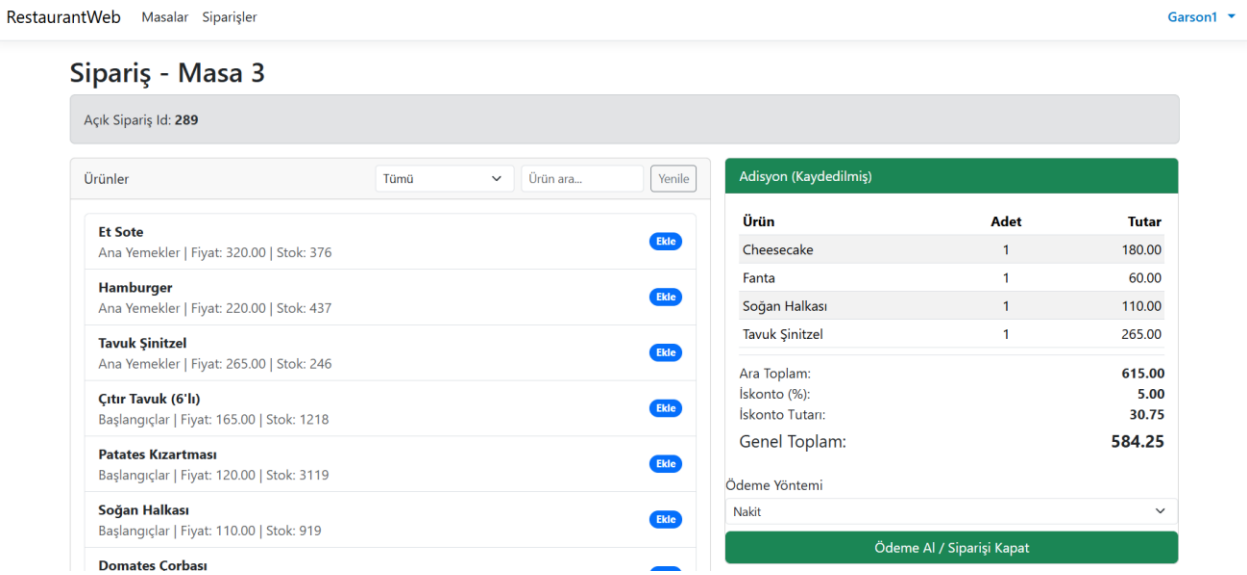
- sipariş kalemleri güncellenir (upsert)
- ürün stokları düşer
- sipariş toplamları hesaplanır

Bu yüzden tek transaction altında yapılır ve concurrency DB seviyesinde kontrol edilir:

- Sipariş satırı FOR UPDATE kilidi ile “sipariş açık mı?” kontrolü garanti altına alınır.
- Ürün satırları FOR UPDATE ile kilitlenir (stok yarışını engeller).
- Stok düşme adımı WHERE stok >= adet guard’ı ile yapılır; böylece negatif stok gibi durumlar DB tarafından engellenir.
- Kalemler ON CONFLICT (siparis_id, urun_id) ile biriktirilir (aynı ürün tekrar eklenirse tek satırda toplanır).
- Toplamlar tek kaynaktan RecalculateTotals ile güncellenir.

Sonuç: sistem, aynı anda birden fazla kullanıcı/istek geldiğinde dahi stok ve adisyon tutarlılığını korur.

Adım D — İskonto uygulama (kritik yazma + audit)



Kasa/garson iskonto uygular:

- **Controller:** SiparislerController.UpdateDiscount
- **Service:** SiparisService.UpdateDiscountRate (tx açar)
- **Repository:** SiparisRepository.UpdateDiscountRate(conn, tx, ...)

Burada önce siparişin açık olduğu FOR UPDATE ile doğrulanır; ardından iskonto_oran set edilir ve toplamlar yine RecalculateTotals üzerinden tek noktadan hesaplanır. Ayrıca siparis_log'a "DISCOUNT" aksiyonu ile audit kaydı düşülür (old/new).

Bu yaklaşımın amacı:

- Toplam hesaplama drift'ini önlemek (tek kaynak)
- İskonto değişikliklerinin izlenebilir olması (audit trail)

Adım E — Ödeme al ve siparişi kapat (kritik + idempotent)

Kasa "Kapat" dediğinde:

- **Controller:** SiparislerController.Close
- **Service:** SiparisService.CloseOrderWithPayment (tx açar)
- **Repository:** SiparisRepository.CloseOrderWithPayment(conn, tx, ...)

Bu akış idempotent tasarlanmıştır:

- Sipariş satırı FOR UPDATE ile kilitlenir; durum okunur.
- Sipariş zaten kapalıysa işlem **başarılı** kabul edilir (“zaten kapalı/ödenmiş”).
- Ödeme insert’i unique ihlaliyle tekrar gelirse 23505 yakalanır ve yine **başarılı** kabul edilir.
- Siparişi kapatma update’i yalnızca durum=0 iken yapılır; 0 satır etkilenirse yine başka bir isteğin kapattığı varsayılır ve işlem **başarılı** sayılır.
- Başarılı kapamada masa Bos yapılır ve siparis_log’a PAYMENT/CLOSE kayıtları atılır.

Bu tasarım, UI tarafında çift tıklama / tekrar istek / ağ retry gibi senaryolarda “kapatma başarısız oldu” algısını engeller; kasa operasyonunu güvenli hale getirir.

4.1.3. Tasarım Kararları

4.1..1.1. Transaction sınırları (Service orchestrator yaklaşımı)

Yazma içeren kritik akışlarda (Submit, Discount, Close, EnsureOpen) transaction sınırı Service katmanında yönetilir:

- Service: connection açar, BeginTransaction() başlatır, repository çağrısını yapar, commit/rollback uygular.
- Repository: sadece DB operasyonlarını yürütür; işin transaction/connection lifecycle’ını sahiplenmez.

Bu yaklaşım, çok adımlı DB değişikliklerini (stok + kalem + toplam + log) tek bir “atomic unit” olarak ele almayı sağlar.

4.1.1.2.Concurrency kontrolü (DB locking + guard)

POS benzeri sistemlerde concurrency kaçınılmazdır. Bu nedenle:

- FOR UPDATE satır kilitleri ile sipariş/masa/ürün seviyesinde yarışlar kontrollü hale getirilir.
- Stok düşme gibi kritik güncellemelerde WHERE stok >= adet guard’ı uygulanır.
- Kalem eklemede UPSERT ile tekrarlar konsolide edilir.

Sonuç: uygulama tarafında ekstra “lock” mekanizmalarına ihtiyaç duymadan PostgreSQL’in doğru kullanımına yaslanılır.

4.1.1.3.Idempotency (özellikle Close akışı)

“Kapat” gibi operasyonlar pratikte retry/double submit görür. Bu yüzden close akışı:

- “zaten kapalıysa OK” prensibiyle tasarlanır,

- unique ihlali ve UPDATE ... WHERE durum=0 gibi DB semantiklerini “idempotent outcome” üretmek için kullanır.

4.1.1.4.Audit / Loglama

Sipariş üzerinde yapılan kritik değişiklikler (iskonto, ödeme, kapama) siparis_log tablosuna yazılır.
Amaç:

- Kim, neyi, ne zaman değiştirdi?
- Operasyonel inceleme ve gerektiğinde geri dönük analiz.

5. ÖNEMLİ KOD PARÇALARI

5.1. Sipariş Gönderme Motoru (SiparisRepository.SubmitOrder)

Bu proje içinde "sipariş yönetimi"nin en kritik noktası, aynı anda birden fazla kullanıcı/istek çalışırken (garson ekranı, kasa, mutfak, yanlışlıkla çift tıklama, ağ gecikmesi vb.) stok, fiyat ve sipariş kalemlerinin tutarlı kalmasıdır. SubmitOrder metodu bu problemi; uygulama katmanında "şans eseri" değil, transaction + satır kilidi + DB guard kombinasyonuyla deterministik biçimde çözer.

Amaç

- Açık bir siparişe, sepetteki ürünleri eklemek/güncellemek
- Ürün stoklarını güvenli şekilde düşmek
- Siparişin finansal toplamlarını tek merkezden güncellemek
- Bunu yaparken **yarış koşullarını (concurrency)** ve **kısmi güncellemeleri** engellemek

Tasarım Kararları ve Gerekçeler

1) Transaction dışarıdan gelir: "tek iş = tek transaction"

- Metot imzasında NpgsqlConnection ve NpgsqlTransaction parametreleri alınıyor. Bunun anlamı:
- Bu akış, **service katmanında** transaction açılıp yönetiliyor.
- Repo, "iş mantığı akışını" yapıyor; commit/rollback kontrolü service'te.
- Böylece sipariş kalemi ekleme + stok düşme + total hesaplama **ya hep ya hiç** (atomic) çalışıyor.

Sonuç: Sipariş kalemleri yazılıp stok düşülmeden fail olma veya stok düşüp kalem yazılmama gibi "yarım kalan" senaryoların önü kesilir.

2) Sipariş satırı FOR UPDATE ile kilitlenir: açık sipariş garantisi

İlk adımda siparişin varlığı ve açık olması kontrol edilir ve **FOR UPDATE** kullanılır. Buradaki hedef:

- Aynı sipariş üzerinde başka bir işlem eş zamanlı çalışıyorsa, **satır kilidi** ile sıraya almak
- Kapalı/iptal siparişe ürün eklenmesini engellemek

Bu, "sipariş durumu değişirken ürün ekleme" gibi edge-case'leri temizler.

3) Ürün satırları FOR UPDATE ile kilitlenir: stok yarış problemi çözümü

Sepetteki ürünler için urunler tablosu **id = ANY(@ids)** ile çekilir ve **FOR UPDATE** ile kilitlenir. Bu sayede:

- Aynı ürünü aynı anda iki farklı sipariş düşmeye çalışsa bile,

- Stok okumaları ve stok düşme sırası **tutarlı** hale gelir.

Burada kilitlenen şey “tablo” değil; ilgili ürün satırlarıdır. Yük altında dahi doğru davranış sağlar.

4) Çift katman stok kontrolü:

Metot stok için iki aşamalı kontrol uygular:

A) Ön kontrol (in-memory):

- Kilitli ürün satırlarından stok okunur, istenen adetle karşılaştırılır.
- Uygunsuzsa hızlıca fail döner.

B) DB guard (UPDATE koşulu):

- Stok düşerken WHERE stok >= @adet koşulu kullanılır.
- ExecuteNonQuery() sonucu 0 ise fail döner.

5) Sipariş kalemleri UPSERT ile yönetilir:

Sipariş kalemleri eklenirken:

- (siparis_id, urun_id) unique mantığıyla aynı ürün aynı siparişte tek satırda tutulur.
- ON CONFLICT ... DO UPDATE ile adet artırılır, fiyat güncellenir.
- satir_toplam DB tarafında hesaplanır (adet * birim_fiyat).

Bu tasarım şunları sağlar:

- Sipariş kalemi ekleme “insert mi update mi?” ayrımı yapmadan tek yoldan ilerler.
- UI tekrar gönderse bile aynı kalem şişmez, **doğru şekilde toplanır**.
- satir_toplam gibi hesap alanlarında “client-side hesap” hataları minimize edilir.

6) Totaller tek otoritede hesaplanır: finansal doğruluk

Metot sonunda RecalculateTotals(...) çağrılır. Bu prensip şunu garanti eder:

- Ara toplam, iskonto, toplam gibi değerler **tek bir yerden** güncellenir.
- Sipariş kalemlerine farklı operasyonlar eklense bile (miktar azaltma, iptal, mutfak iadesi vb.) total hesaplama mantığı merkezi kalır.
- “Total hesaplama bir kez yazılır, her yerden çağrılır.” Bu, bakım maliyetini ve bug riskini ciddi düşürür.

5.2. Rezervasyon Zamanı Gelince Otomatik Sipariş Açma (MasaService.AutoOpenOrdersFromDueReservations)

Bu projede masa yönetimi sadece “boş/dolu” göstermekten ibaret değil; rezervasyon akışı masaya yansıtılmalı ve operasyonel olarak masanın doğru anda siparişe hazır hale gelmesi gerekir.

AutoOpenOrdersFromDueReservations metodu, rezervasyon saati yaklaşan aktif rezervasyonları tespit edip ilgili masa için siparişi otomatik açarak, personelin manuel olarak “masa aç” yapmasına gerek kalmadan süreci ilerletir. Bu davranış, özellikle yoğun saatlerde “rezervasyon var ama sistemde masa hâlâ boş görünüyor” gibi tutarsızlıkları engeller.

Amaç:

- Rezervasyon zamanı gelen (veya az gecikmiş) masaları otomatik olarak operasyona hazırlamak
- Masa satırında concurrency problemi yaşanmadan siparişi açmak
- Sipariş açılınca rezervasyonu “kullanıldı” durumuna geçirmek
- Tüm işlemleri tek transaction içinde tutup yarım kalmış senaryoları önlemek

Tasarım Kararları ve Gerekçeler

1) Konfigürasyon kontrollü “grace window” yaklaşımı

Metot, Reservation:AutoOpenGraceMinutes ile bir tolerans aralığı kullanır. Bu aralık yoksa 15 dakika kabul edilir, ayrıca 1–180 dakika aralığına zorlanır. Bu sayede yanlış config yüzünden sistemin hiç çalışmaması veya aşırı agresif çalışması engellenir. Rezervasyon saatine çok yakınken (veya küçük gecikmelerde) otomasyon devreye girer.

2) “Due” rezervasyonları seçip erken çıkış yapmak

İlk iş olarak _rezRepo.GetDueActiveReservations(now, grace) çağrılır. Burada hedef, yalnızca gerçekten işlenmesi gereken rezervasyonları çekmektir. Liste boşsa metot direkt biter; gereksiz DB bağlantısı/transaction maliyeti oluşmaz. Bu, board ekranı sık açıldığında performans açısından önemlidir.

3) “Toplu iş” tek transaction içinde yürütülür

Metot kendi connection/transaction’ını açar ve due rezervasyonların hepsini aynı transaction içinde işler. Buradaki amaç şudur: “sipariş açıldı” ve “rezervasyon kullanıldı” adımlarından biri başarısız olursa diğerinin tek başına kalmaması gerekir. Bu yaklaşım, otomasyonun sistemde yarım iz bırakmasını engeller ve **süreç bütünlüğünü artırır.**

4) Masa satırını FOR UPDATE ile kilitleyerek yarış koşullarını kontrol etmek

Her rezervasyon için önce _masaRepo.GetDurumAndLock(conn, tx, rez.MasaId) çalışır. Bu metot masa satırını **FOR UPDATE** ile kilitler. Böylece aynı masa üzerinde eş zamanlı işlemler (başka bir personelin masa açması, aynı anda iki board isteği gelmesi vb.) olduğunda, işlemler sıra ile gerçekleşir. Kilit "tabloyu" değil sadece ilgili masa satırını tuttuğu için de ölçeklenebilirlik korunur.

Ayrıca burada iki net filtre uygulanır:

- Masa bulunamazsa devam edilir (silinmiş/bozuk kayıt vb.)
- Masa pasifse devam edilir (operasyon dışı masaya dokunulmaz)

5) Rezervasyonu "kullanıldı" yapmak için açık bir koşul: önce sipariş açılmalı

Metodun en kritik doğruluk prensibi şudur: rezervasyonun durumu ancak sipariş açma işlemi başarılıysa "kullanıldı"ya çekilir. Bu küçük görünen karar, raporlama ve operasyonel güvenilirlik açısından büyük fark yaratır. Aksi halde sistemde "kullanıldı" görünen ama siparişi hiç açılmamış rezervasyonlar oluşabilir.

Sonuç

Bu metot, rezervasyon akışını masa ve sipariş yaşam döngüsüne otomatik biçimde entegre ederek uygulamayı daha "gerçek restoran operasyonu"na yaklaştırır. Özellikle FOR UPDATE kilidi ve "sipariş açıldıysa rezervasyonu kullanıldı yap" kuralı sayesinde, concurrency altında dahi deterministik ve tutarlı bir davranış elde edilir.

5.3. Finansal Toplamların Tek Otoritede Hesaplanması (SiparisRepository.RecalculateTotals)

Sipariş akışında en fazla hata üreten alanlardan biri **"toplam hesabı"dır.** Özellikle sipariş kalemi ekleme/çıkarma, iskonto güncelleme, kalem iptali, mutfak iadesi gibi operasyonlar arttıkça; ara toplam–iskonto–genel toplam hesapları farklı yerlerde yapılırsa sistem zamanla tutarsızlaşır. RecalculateTotals metodu bu riski, finansal hesapları tek bir otoritede toplayarak azaltmak için var.

Amaç:

- Siparişin **ara toplam** değerini sipariş kalemlerinden güvenilir şekilde üretmek
- Siparişin mevcut **iskonto oranını** okuyup iskonto tutarını deterministik hesaplamak
- siparisler tablosundaki ara_toplam, iskonto_tutar, toplam alanlarını **tek noktadan** güncellemek

- Bu işlemleri transaction içinde yaparak aynı sipariş üzerindeki diğer işlemlerle **tutarlı** kalmak

Tasarım Kararları ve Gerekçeler

1) **Ara toplam "kalemlerden" gelir: tek kaynak siparis_kalemleri**

Ara toplam, uygulama belleğinde biriken veriden değil; doğrudan DB'deki siparis_kalemleri.satir_toplam alanlarının toplamından üretilir.

- COALESCE(SUM(...), 0) kullanımı sayesinde **kalem yoksa bile** ara toplam 0 olur.
- Böylece "NULL toplam" gibi UI/rapor tarafında patlayan edge-case'ler engellenir.
- Daha önemlisi: ara toplamın kaynağı nettir → **kalemler doğruysa ara toplam da doğrudur.**

2) **İskonto oranı sipariş header'ından okunur: hesap "tek yerde" yapılır**

İskonto oranı **siparisler** tablosundan çekilir. Bu şu avantajı getirir:

- İskonto, "kalemlerin içine dağılmış" bir mantık olmaz.
- İskonto değiştiğinde yapılacak iş bellidir: oran güncellenir → **RecalculateTotals çağrılır** → her şey güncel kalır.
- İleride "kampanya indirimi + personel indirimi + kupon indirimi" gibi katmanlar eklense bile yine aynı felsefe korunur: oran/kurallar header'da, hesap tek otoritede.

3) **Güncelleme tek SQL ile yapılır: tutarlılık ve bakım kolaylığı**

Metot sonunda üç alan birlikte güncellenir:

- ara_toplam
- iskonto_tutar
- toplam

Bunların ayrı ayrı set edilmesi, "biri güncellendi biri kaldı" gibi yarım güncelleme riskini büyütür. Tek güncelleme ile bu risk düşer ve kodun bakım maliyeti azalır.

Sonuç

RecalculateTotals'in asıl katkısı "hesap yapmak" değil; **hesabın nerede yapıldığını standartlaştırmak**tır. Bu metot sayesinde siparişin finansal durumu, sistemde hangi operasyonlar olursa olsun aynı prensiple güncellenir:

Kalemler → Ara Toplam → İskonto → Toplam (tek otorite, transaction içinde)

6. Kimlik Doğrulama ve Yetkilendirme (Cookie Auth + Role Flags)

Bu projede güvenlik konusuna yalnızca “kullanıcı giriş yapabiliyor mu?” perspektifiyle yaklaşmadım. Gerçek bir restoran ortamında; kasa, mutfak ve garson ekranlarının aynı sistem üzerinde ama farklı yetkilerle çalışması gerektiğini göz önünde bulundurarak, kimlik doğrulama ve yetkilendirme mekanizmasını en baştan bilinçli şekilde kurguladım.

Amacım, hem güvenli hem de günlük kullanımda kullanıcıyı yormayan bir yapı oluşturmaktı. Bu nedenle ASP.NET Core’un yerleşik **Cookie Authentication** mekanizmasını tercih ettim ve rol bazlı yetkilendirme ile destekledim.

6.1. Uygulama Seviyesinde Güvenlik Altyapısı

Uygulama başlangıcında **(Program.cs)** kimlik doğrulama ve yetkilendirme altyapısı merkezi olarak tanımlanmıştır. Cookie tabanlı authentication yapılandırılarak, oturum yönetimi framework seviyesinde ele alınmıştır.

Burada özellikle dikkat ettiğim nokta, POS senaryosuna uygun bir oturum davranışı elde etmekti. Kullanıcının aktif olduğu sürece oturumunun düşmemesi, ancak belirli bir süre sonra güvenli şekilde sonlanması gerekiyordu. Bu nedenle cookie için **Sliding Expiration** aktif edildi ve **oturum süresi 8 saat** olarak belirlendi.

6.2. Giriş Akışı ve Güvenli Yönlendirme

Giriş işlemini tasarlarken, controller’ın yalnızca akışı yöneten bir rol üstlenmesini istedim. Kullanıcı adı ve şifre doğrulaması gibi iş kurallarını doğrudan controller içinde yapmak yerine, bu sorumluluğu **AuthService** katmanına taşıdım. Bu sayede:

- Giriş mantığı tek bir yerde toplandı.
- Controller sade ve okunabilir kaldı.
- İleride doğrulama kuralları değiştiğinde controller’lara dokunma ihtiyacı ortadan kalktı.

Login POST işleminde ayrıca **CSRF saldırılarına karşı koruma** sağlamak amacıyla **ValidateAntiForgeryToken** kullanıldı. Bunun özellikle form tabanlı uygulamalarda göz ardı edilmemesi gereken temel bir güvenlik önlemi olduğunu öğrendim.

Bir diğer önemli nokta, giriş sonrası yönlendirme sürecidir. Kullanıcı giriş yaptıktan sonra eğer bir returnUrl mevcutsa, bu adresin mutlaka uygulama içi (local) olup olmadığı kontrol edilir. Böylece sistemin dış bir adrese yönlendirilmesi engellenmiş olur. Bu küçük gibi görünen kontrol, pratikte ciddi bir güvenlik açığını kapatır.

6.3. Rol Modeli: Flags Enum Kullanımı

Projede en bilinçli aldığım kararlardan biri, kullanıcı rollerini klasik **"tek rol"** mantığıyla sınırlamamaktı. Gerçek hayatta bir personelin birden fazla sorumluluğu olabilir; örneğin bir kişi hem kasa hem de garson yetkisine sahip olabilir.

Bu ihtiyacı karşılamak için roller **Flags** attribute'u ile tanımlanan bir enum üzerinden modellenmiştir. Bu yaklaşım sayesinde:

- Roller veritabanında tek bir alan içinde bitmask olarak saklanabilmiştir.
- Bir kullanıcıya birden fazla rol atanabilmiştir.
- Kod tarafında rol kontrolleri sade ve genişletilebilir hale gelmiştir.

Giriş sırasında, kullanıcının sahip olduğu roller enum üzerinden kontrol edilerek dinamik olarak claim'lere dönüştürülmektedir. Böylece yetkilendirme mekanizması, sabit ve kırılgan if-else blokları yerine, genişlemeye açık bir yapı kazanmıştır.

Bu tasarım, ileride yeni bir rol eklenmesi gerektiğinde sistemin büyük ölçüde değişmeden çalışmaya devam etmesini sağlar.

[Flags]

public enum PersonelRol

{ None = 0, Admin = 1 << 0, Kasa = 1 << 1, Garson = 1 << 2, Mutfak = 1 << 3,}

6.4. Parola Güvenliği ve Hashleme Stratejisi

Kullanıcı şifrelerinin güvenliği için şifreleri veri tabanında düz metin olarak saklamadım; her kullanıcı için ayrı salt üretilir ve **PBKDF2** algoritması kullanılarak hashlenir. Doğrulama sırasında:

- Kullanıcının girdiği şifre, veritabanındaki salt ile yeniden hashlenir.
- Hash karşılaştırması FixedTimeEquals kullanılarak yapılır.

Bu yaklaşım, hem brute-force saldırılarını zorlaştırır hem de zamanlama tabanlı saldırı risklerini azaltır. Böylece şifre güvenliği, uygulamanın zayıf noktalarından biri olmaktan çıkarılmış olur.

6.5. Rol Bazlı Kullanıcı Deneyimi (Landing Mantığı)

Giriş sonrası kullanıcıyı tek bir ana sayfaya göndermek yerine, sahip olduğu role göre doğrudan ilgili ekrana yönlendirmeyi tercih ettim. Bunun sebebi tamamen operasyonel verimlilikti.

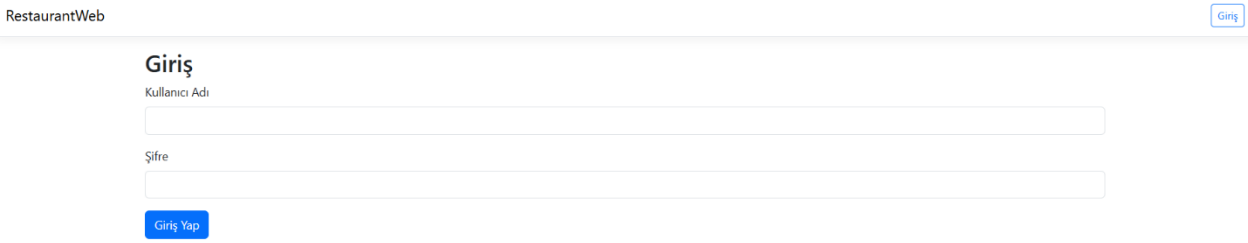
- Mutfak personeli doğrudan mutfak ekranına,
- Garson masa yönetim ekranına,
- Kasa ve admin rolleri raporlama ekranına yönlendirilir.

Bu yönlendirme mantığı service katmanında merkezi olarak ele alındı. Böylece controller'larda rol bazlı yönlendirme karmaşası oluşmadı ve kullanıcı deneyimi tutarlı hale geldi.

7. Genel Proje Uygulama Tanıtımı

Bu bölümde projedeki sayfaların temel amacını, kullancıların neler yapabileceğini anlatacağım.

7.1. Giriş Ekranı



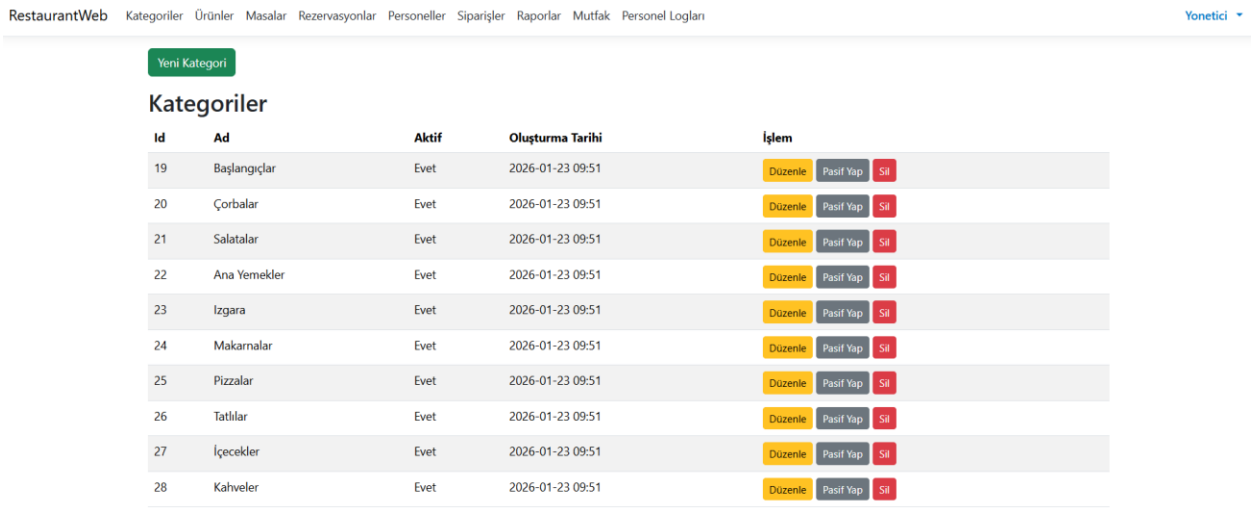
Bu ekran, uygulamaya erişimin tek giriş noktasıdır. Kullanıcılar kendilerine tanımlı **kullanıcı adı ve şifre** ile sisteme giriş yapar.

Giriş işlemi sırasında:

- Kullanıcının bilgileri doğrulanır,
- Hesabın aktif olup olmadığı kontrol edilir,
- Kullanıcının sahip olduğu role göre (Garson, Mutfak, Kasa, Admin) **doğrudan ilgili ekrana yönlendirme** yapılır.

Bu sayede kullanıcı sisteme girdikten sonra menüler arasında kaybolmadan, yalnızca kendi görev alanına ait ekranlarla çalışır.

7.2. Kategoriler Yönetimi

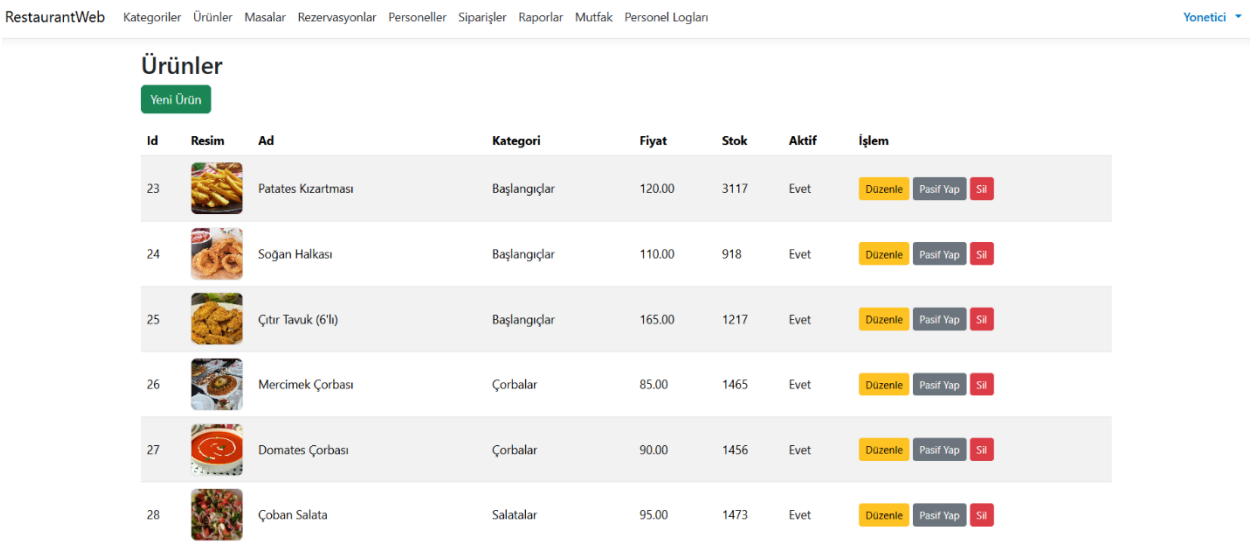


Bu ekran, restoranda sunulan ürünlerin **mantıksal olarak gruplandırıldığı** kategori yapısının yönetildiği alandır. Ürünlerin doğrudan serbest biçimde tanımlanması yerine, kategori bazlı bir yapı tercih edilmiştir. Bunun temel nedeni; sipariş alma, mutfak ekranı ve raporlama gibi süreçlerde verinin daha kontrollü ve anlamlı biçimde kullanılmasını sağlamaktır.

Bu ekranda kullanıcı:

- Yeni kategori ekleyebilir,
- Mevcut kategorilerin adını düzenleyebilir,
- Kategorileri **aktif / pasif** duruma alabilir,
- Kullanılmayan kategorileri silebilir.

7.3. Ürünler Yönetimi

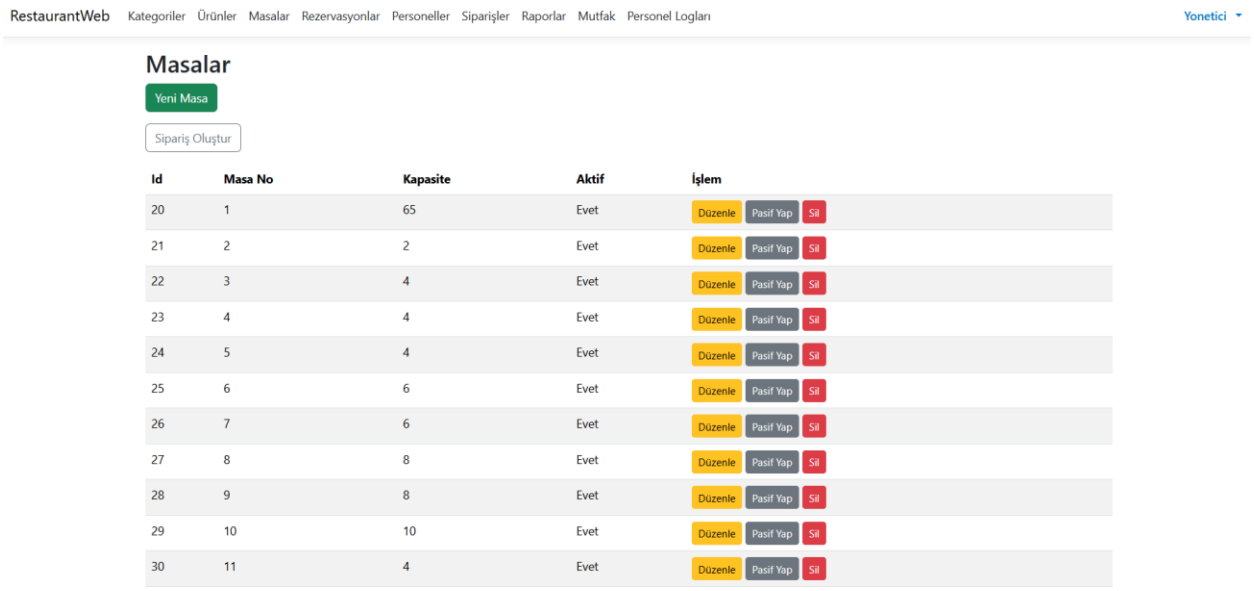


Bu ekran, restoranda satışı yapılan tüm ürünlerin yönetildiği ana paneldir. Ürünler; kategori, fiyat ve stok bilgileriyle birlikte tanımlanır ve sipariş süreçlerinin tamamı bu ekran üzerinden tanımlanan verilerle çalışır.

Bu ekranda kullanıcı:

- Yeni ürün ekleyebilir,
- Ürün bilgilerini (ad, kategori, fiyat, stok) güncelleyebilir,
- Ürünleri **aktif / pasif** duruma alabilir,
- Ürünlere görsel ekleyerek sipariş alma ekranında daha okunabilir bir yapı oluşturabilir.

7.4. Masalar Yönetimi



Bu ekran, restorandaki **fiziksel masa yapısının** sistemde birebir temsil edildiği yönetim ekranıdır. Masalar; masa numarası, kapasite ve aktiflik durumu ile tanımlanır ve sipariş, rezervasyon ve doluluk hesaplamalarının tamamı bu yapı üzerinden yürür.

Bu ekranda kullanıcı:

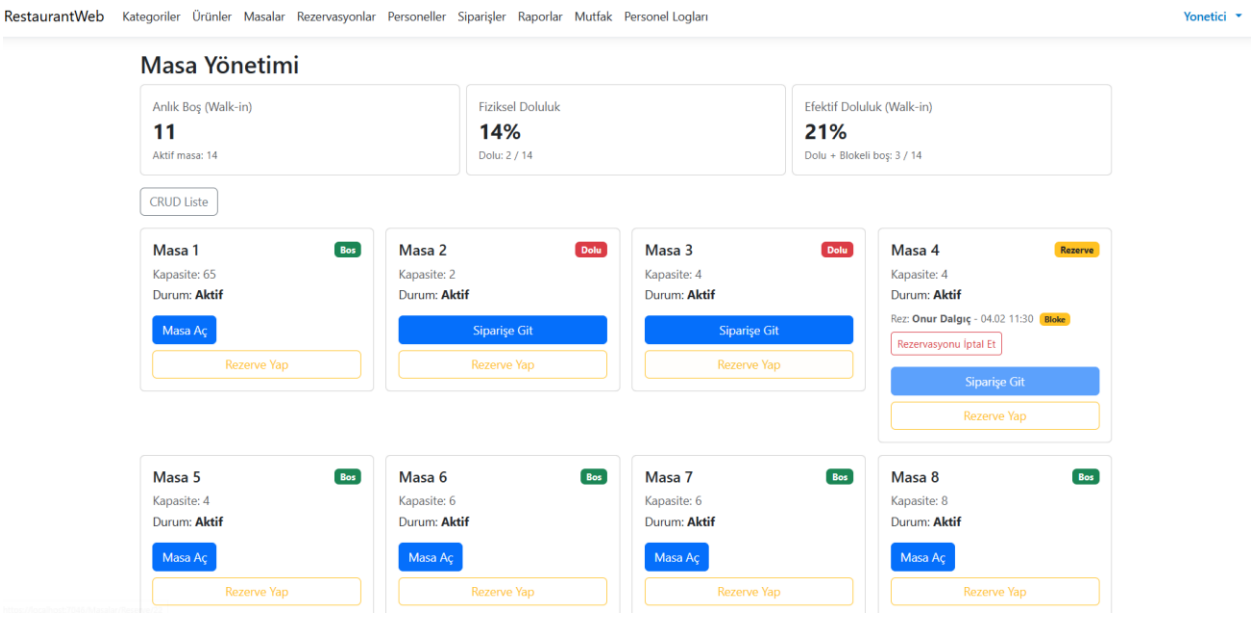
Yeni masa ekleyebilir,

Masaların kapasite bilgilerini düzenleyebilir,

Masaları **aktif / pasif** duruma alabilir,

Kullanılmayan masaları sistemden kaldırabilir.

7.5. Masa Yönetimi (Operasyonel Ekran)



Bu ekran, sistemde tanımlı masaların **anlık operasyonel durumunu** tek bakışta görebilmek için tasarlandı. CRUD amaçlı masa tanım ekranından farklı olarak, burası doğrudan **günlük servis akışının yönetildiği** ana ekranlardan biridir.

Bu ekranda kullanıcı:

- Masaların **boş / dolu / rezerve** durumunu anlık olarak görebilir,
- Boş bir masa için doğrudan **sipariş açabilir**,
- Mevcut bir siparişe tek tıkla geçiş yapabilir,
- Masalar için **rezervasyon oluşturabilir veya iptal edebilir**.

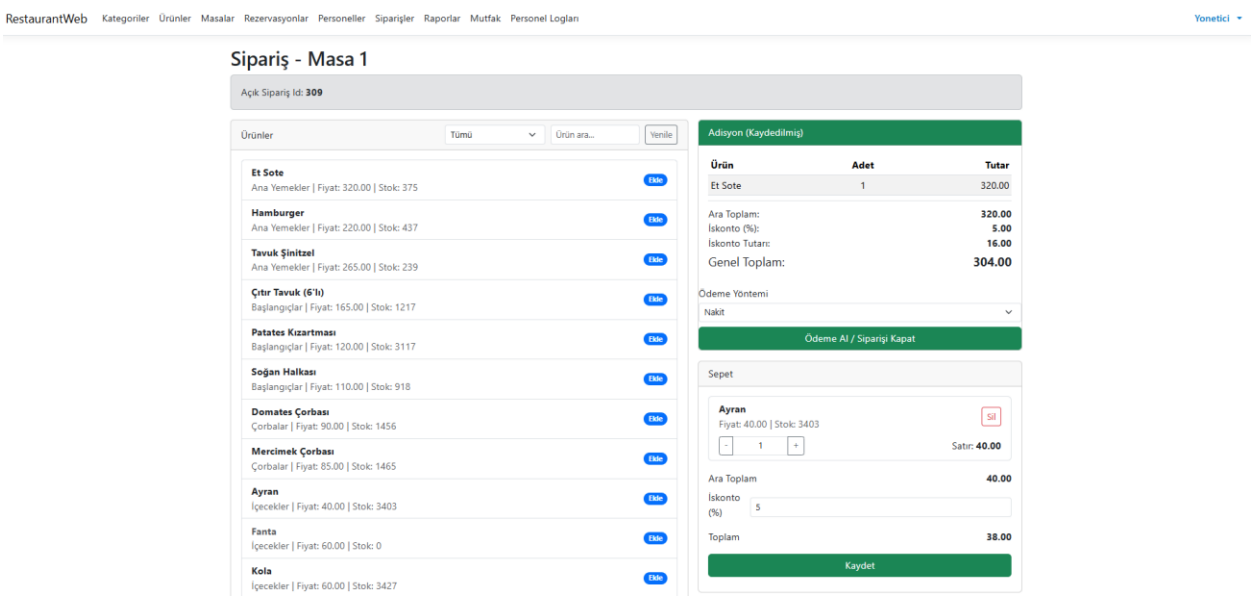
Üst bölümde yer alan özet kartlar sayesinde:

- Anlık walk-in boş masa sayısı,
- Fiziksel doluluk oranı,
- Rezervasyonlar dahil edilerek hesaplanan **efektif doluluk oranı**

tek bakışta izlenebilir. Bu sayede yalnızca “kaç masa dolu?” sorusu değil, “operasyonel olarak ne kadar doluyuz?” sorusu da cevaplanmış olur.

Bu ekranı tasarlarken amacım, garsonun veya yönetici kullanıcının **liste okumak zorunda kalmadan**, renkler ve aksiyon butonları üzerinden hızlı karar verebilmesini sağlamaktı. Bu nedenle masa kartları, hem durum bilgisini hem de yapılabilecek işlemleri aynı yerde sunacak şekilde kurgulandı.

7.6. Sipariş Alma ve Adisyon Yönetimi Ekranı



Bu ekran, bir masaya ait **aktif siparişin (adisyonun)** uçtan uca yönetildiği ana çalışma ekranıdır. Garsonun sipariş alma, güncelleme ve ödeme öncesi tüm işlemleri tek bir akış içinde yapabilmesi hedeflenmişti

Ekran üç ana bölümden oluşur:

7.6.1. Ürün Listesi (Sol Panel)

Bu alanda:

- Sistemde aktif olan tüm ürünler listelenir,
- Ürünler kategoriye göre filtrelenebilir veya isimle aranabilir,
- Her ürün için **anlık fiyat ve stok bilgisi** doğrudan görüntülenir.

“Ekle” butonu ile ürünler sepete alınır. Stok bilgisi görünür olduğu için, garson sipariş sırasında stok kaynaklı hataları en baştan fark edebilir.

7.6.2. Sepet (Alt Sağ Panel)

Sepet bölümü, henüz **veritabanına kaydedilmemiş** sipariş kalemlerini temsil eder.

Bu alanda kullanıcı:

- Ürün adetlerini artırıp azaltabilir,
- Ürünleri sepetten silebilir,
- İskonto oranını girerek ara toplam ve toplam tutarı anlık olarak görebilir.

Sepette görünen tutarlar, kullanıcıya hızlı geri bildirim sağlamak içindir. Asıl hesaplama ve doğrulama işlemleri, sipariş kaydedildiğinde **sunucu tarafında** tekrar yapılır.

7.6.3. Adisyon (Kaydedilmiş Sipariş) Paneli (Üst Sağ)

Bu panel, veritabanında kayıtlı olan mevcut siparişin özetini gösterir:

- Kaydedilmiş ürünler ve adetleri,
- Ara toplam, iskonto tutarı ve genel toplam,
- Seçilen ödeme yöntemi.

Bu ayrım sayesinde:

- Sepet (geçici durum) ile adisyon (kalıcı durum) net biçimde ayrılır,
- Kullanıcı, hangi ürünlerin sisteme kaydedildiğini açıkça görür.

7.6.4. Ödeme ve Sipariş Kapatma

Ödeme yöntemi seçildikten sonra:

"Ödeme Al / Siparişi Kapat" ile ödeme alınır, sipariş kapatılır ve masa otomatik olarak boş duruma çekilir.

Bu ekranı tasarlarken amacım; **sipariş alma, düzenleme ve ödeme süreçlerini tek bir ekranda**, minimum karmaşa ile yönetilebilir hale getirmekti. Böylece hem operasyon hızı artıyor hem de kullanıcı hataları belirgin şekilde azalıyor.

7.7. Rezervasyonlar Ekranı

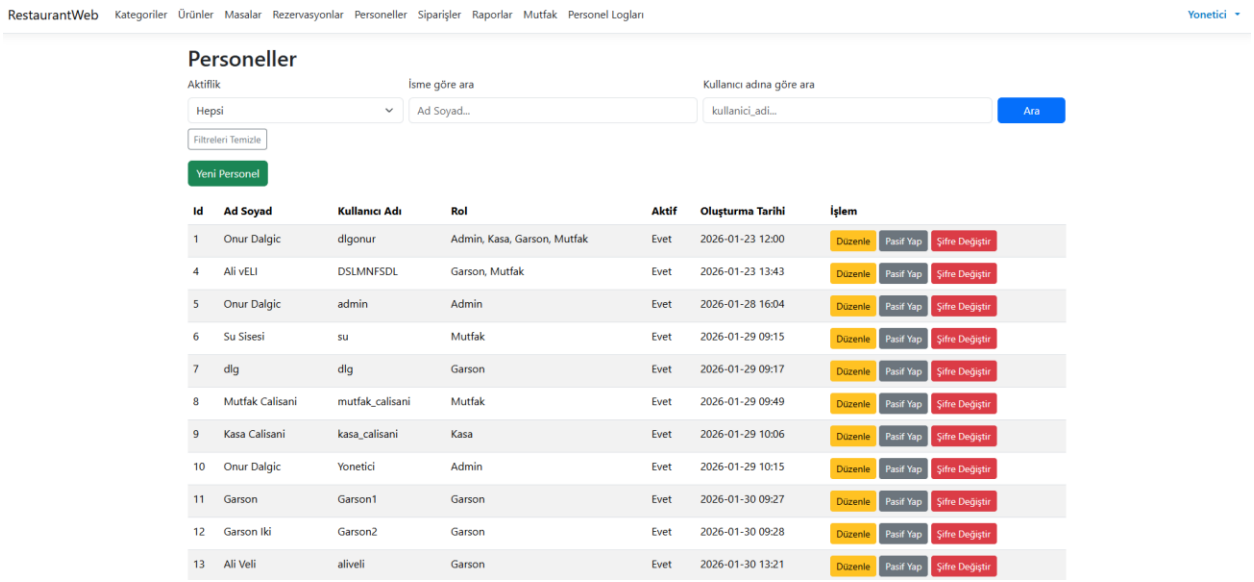
Rezervasyonlar ekranı, restoranın **ileri tarihli masa planlamasını** ve rezervasyon geçmişini merkezi bir noktadan yönetmek amacıyla tasarlanmıştır. Bu ekran yalnızca kayıt göstermek için değil, operasyonel kararları desteklemek için filtrelenebilir ve aksiyon alınabilir bir yapı sunar.

Ekranın üst bölümünde yer alan filtreler ile kullanıcı:

- Tarih aralığına göre rezervasyonları seçebilir,
- Rezervasyon durumuna göre (Aktif / Kullanıldı / İptal) filtreleme yapabilir,
- Belirli bir masa numarasına ait rezervasyonları görüntüleyebilir,
- Müşteri adına göre arama yapabilir,
- Listelenecek kayıt sayısını kontrol edebilir.

Bu yapı sayesinde hem **gelecek rezervasyonlar** hem de **geçmiş kayıtlar** tek ekran üzerinden rahatça incelenebilir.

7.8. Personeller Ekranı



Personeller ekranı, sistemde tanımlı tüm kullanıcıların **yetki, erişim ve hesap durumlarının** merkezi olarak yönetildiği alandır. Bu ekran, uygulamanın güvenlik ve rol bazlı erişim modelinin yönetim yüzünü temsil eder.

Filtreleme ve Arama

Ekranın üst bölümünde yer alan filtreler sayesinde:

- Aktif / pasif personeller listelenebilir,
- Ad–soyad üzerinden arama yapılabilir,
- Kullanıcı adına göre filtreleme uygulanabilir.

Bu yapı, özellikle çok sayıda personelin bulunduğu ortamlarda ilgili kullanıcıya hızlı erişim sağlar.

Personel Listesi ve Roller

Listede her personel için:

- Ad–soyad ve kullanıcı adı,
- Sahip olduğu roller (Admin, Kasa, Garson, Mutfak vb.),
- Hesabın aktif/pasif durumu,
- Oluşturulma tarihi

açık şekilde görüntülenir.

Roller **bit-mask (flags enum)** mantığıyla tanımlandığı için bir kullanıcı birden fazla rolü aynı anda taşıyabilir.

İşlemler

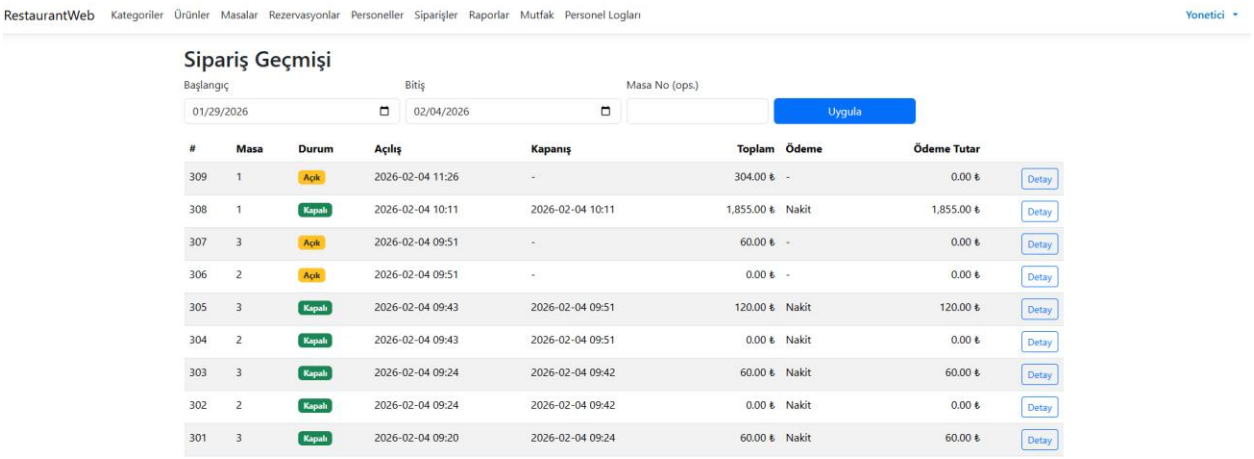
Her personel satırı üzerinden:

- Personel bilgileri güncellenebilir,
- Hesap pasif hale getirilebilir,
- Şifre değişikliği yapılabilir.

Şifre yönetimi, güvenlik gereği ayrı bir akış olarak ele alınmış ve düz metin şifre tutulmadan, hash + salt yaklaşımıyla uygulanmıştır.

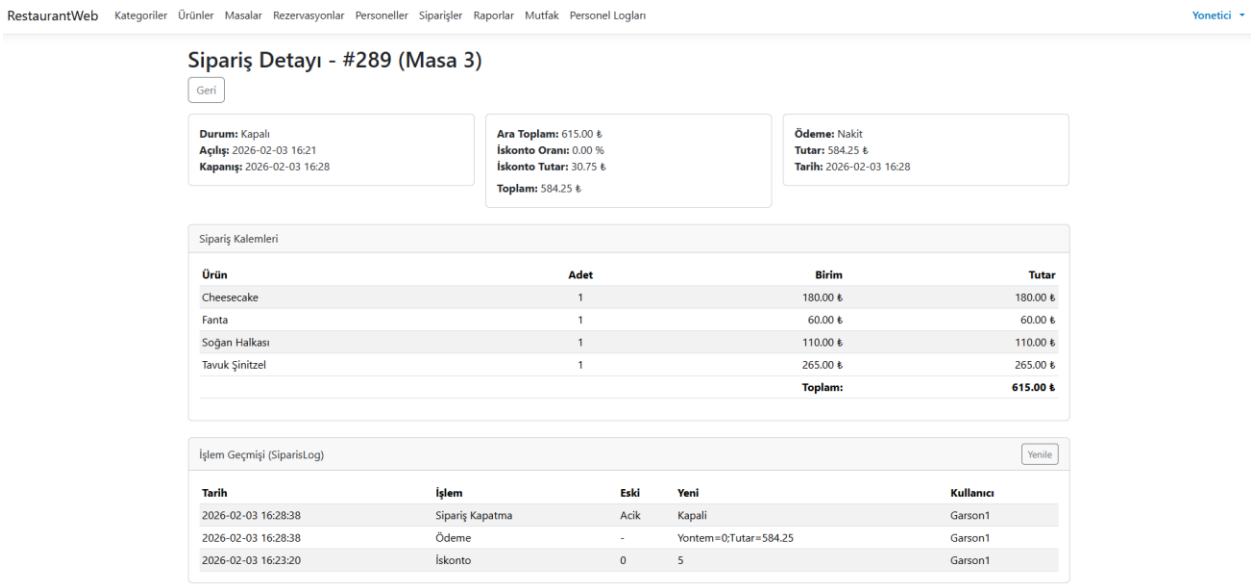
Herhangi bir personelin kaydı silinemez. Onun yerine pasif hale getirilmesi bilinçli olarak bir tercihimdir.

7.9. Sipariş Geçmiş Ekranı



Sipariş Geçmişi ekranı, sistemde açılmış tüm siparişlerin **zamansal, finansal ve operasyonel geçmişinin** izlenmesini sağlar. Bu ekran özellikle kasa, yönetici ve denetim amaçlı kullanımlar için tasarlanmıştır.

7.10. Sipariş Detayı Ekranı



Sipariş Geçmişi ekranında yer alan **Detay** butonuna tıklanarak girilen bu ekran, seçilen siparişe ait tüm bilgilerin **tek bir yerde, okunabilir ve şeffaf** biçimde görüntülenmesini sağlar.

Bu ekranda kullanıcı:

- Siparişte yer alan ürünleri,
- Ürün bazında adet ve tutarları,
- Ara toplam, iskonto ve genel toplam,
- Ödeme yöntemi ve kapanış durumu

net ve değiştirilemez bir özet olarak sunulur. Bu alan, “son durum” bilgisini temsil eder.

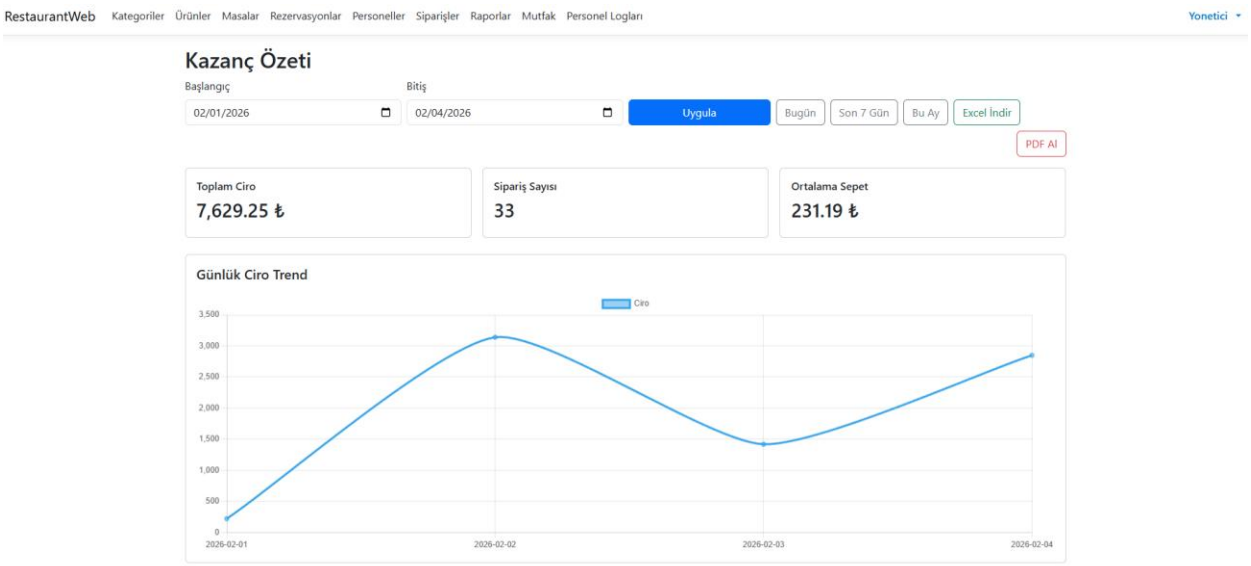
İşlem Geçmişi (Sipariş Logları)

Ekranın alt kısmında yer alan **İşlem Geçmişi**, bu projenin standart CRUD uygulamalarından ayrıldığı en kritik noktalardan biridir. Bu bölümde kullanıcı:

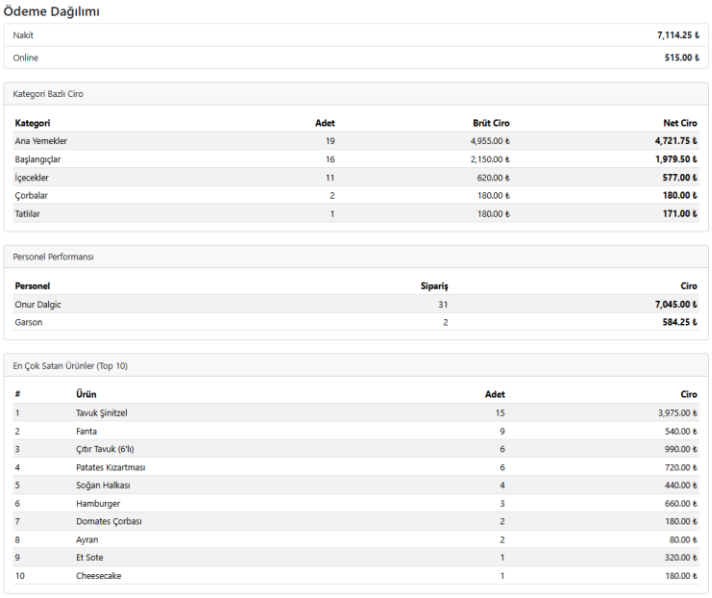
- Siparişin **ne zaman açıldığını**,
- Hangi ürünün **kim tarafından eklendiğini veya çıkarıldığını**,
- İskonto değişikliklerini,
- Siparişin **hangi kullanıcı tarafından, hangi ödeme yöntemiyle kapatıldığını**

zaman damgalı ve kullanıcı bazlı olarak görebilir.

7.11. Raporlar (Kazanç Özeti) Ekranı



Bu ekran, belirli bir tarih aralığında restoranın **finansal ve operasyonel performansını tek bir**



Ödeme Dağılımı

Nakit	7,114.25 ₺
Online	515.00 ₺

Kategori Bazlı Ciro

Kategori	Adet	Brüt Ciro	Net Ciro
Ana Yemekler	19	4,955.00 ₺	4,721.75 ₺
Başlangıçlar	16	2,150.00 ₺	1,979.50 ₺
İçecekler	11	620.00 ₺	577.00 ₺
Çorbalar	2	180.00 ₺	180.00 ₺
Tatlılar	1	180.00 ₺	171.00 ₺

Personel Performansı

Personel	Sipariş	Ciro
Onur Dalgic	31	7,045.00 ₺
Garson	2	584.25 ₺

En Çok Satan Ürünler (Top 10)

#	Ürün	Adet	Ciro
1	Tavuk Şinitzel	15	3,975.00 ₺
2	Fanta	9	540.00 ₺
3	Çıtır Tavuk (6'lı)	6	990.00 ₺
4	Patates Kızartması	6	720.00 ₺
5	Soğan Halkası	4	440.00 ₺
6	Hamburger	3	660.00 ₺
7	Domates Çorbası	2	180.00 ₺
8	Ayran	2	80.00 ₺
9	Et Sote	1	320.00 ₺
10	Cheesecake	1	180.00 ₺

panel üzerinden analiz edebilmek amacıyla tasarlanmıştır.

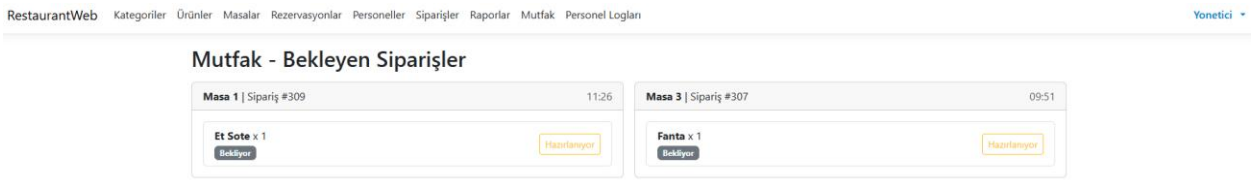
Kullanıcı bu ekran üzerinden:

- Belirlenen tarih aralığı için **toplam ciro**, **sipariş sayısı** ve **ortalama sepet tutarını** özet olarak görüntüleyebilir.

- Günlük bazda **ciro trendini grafik üzerinde** inceleyerek yoğun ve zayıf günleri analiz edebilir.
- **Ödeme yöntemlerine göre dağılımı** (nakit / online) ayrı ayrı görebilir.
- **Kategori bazlı ciro kırılımı** sayesinde hangi ürün gruplarının daha fazla gelir ürettiğini analiz edebilir.
- **Personel performansını**, alınan sipariş sayısı ve üretilen ciro üzerinden karşılaştırmalı olarak inceleyebilir.
- En çok satılan ürünleri **Top 10 listesi** halinde görerek menü optimizasyonu için veri elde edebilir.
- Aynı raporu **Excel formatında dışa aktarabilir**, gerektiğinde yönetsel raporlama için kullanabilir.

Bu yapı sayesinde rapor ekranı yalnızca geçmişi listeleyen bir sayfa olmaktan çıkartılarak, **karar destek mekanizması** olarak kullanılabilecek bütüncül bir yönetim paneline dönüştürülmüştür.

7.12. Mutfak (Bekleyen Siparişler) Ekranı



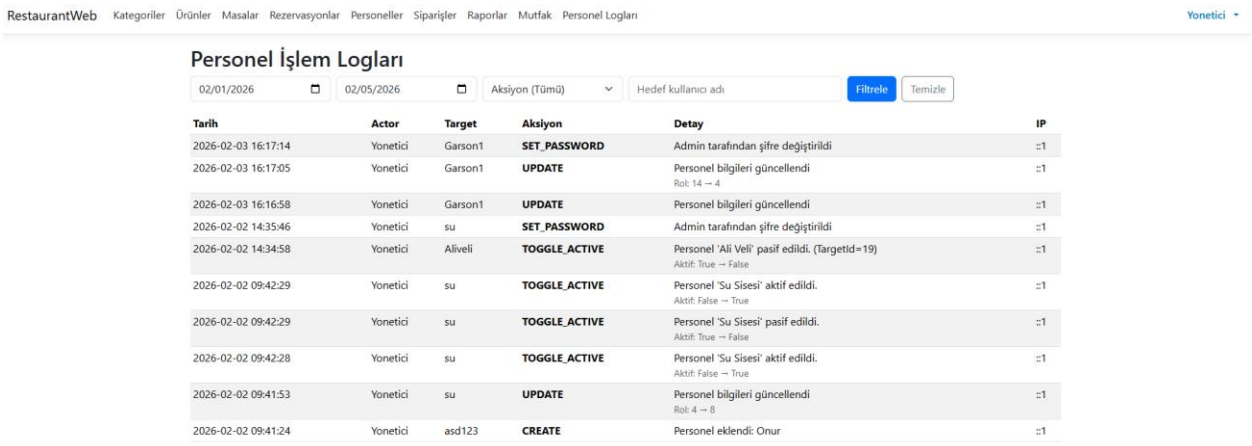
Bu ekran, mutfak personelinin **aktif siparişleri anlık olarak takip edebilmesi** ve sipariş kalemlerinin durumunu yönetebilmesi amacıyla tasarlanmıştır.

Bu ekran üzerinden:

- Açık olan siparişler **masa ve sipariş numarası bazında** listelenir.
- Her siparişin içerisindeki ürünler **kalem kalem** görüntülenir.
- Sipariş kalemleri için durum takibi yapılır:
- **Bekliyor → Hazırlanıyor → Hazır → Servise Çıktı**
- Siparişlerin **zaman bilgisi** görüntülenerek önceliklendirme yapılabilir.

Bu yapı sayesinde mutfak tarafında siparişler sözlü iletişime gerek kalmadan, **dijital ve izlenebilir bir akış** içerisinde yönetilir. Garson–mutfak koordinasyonu netleşir ve operasyonel gecikmeler minimize edilir.

7.13. Personel İşlem Loglar (Audit Log) Ekranı



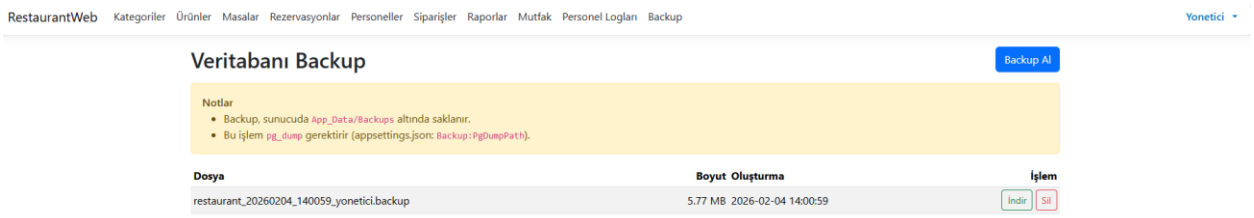
Bu ekran, sistem üzerinde yapılan **kritik personel işlemlerinin izlenebilirliğini** sağlamak amacıyla geliştirilmiştir. Amaç; kim, ne zaman, kime, hangi işlemi yaptı sorularının net ve geriye dönük olarak cevaplanabilmesidir.

Bu ekran üzerinden:

- Personeller üzerinde yapılan tüm işlemler **tarihsel olarak** listelenir.
- İşlemi yapan kullanıcı (**Actor**) ve işlemin hedefi (**Target**) açıkça görüntülenir.
- Yapılan işlem türü standartlaştırılmış aksiyon kodlarıyla tutulur (CREATE, UPDATE, TOGGLE_ACTIVE, SET_PASSWORD vb.).
- İşleme ait açıklayıcı **detay bilgisi** (rol değişimi, aktif/pasif geçişi gibi) gösterilir.
- İşlemin hangi **IP adresinden** gerçekleştirildiği kaydedilir.
- Tarih aralığı, aksiyon türü ve hedef kullanıcı adına göre **filtreleme** yapılabilir.

Bu yapı sayesinde sistem, yalnızca işlevsel değil aynı zamanda **denetlenebilir ve kurumsal standartlara uygun** bir hale getirilmiştir. Yetkili kullanıcıların yaptığı tüm kritik işlemler şeffaf biçimde kayıt altına alınmaktadır.

7.14. Veritabani Backup (Yedekleme) Ekranı



Bu ekran, restoran yönetim sisteminde tutulan tüm operasyonel ve finansal verilerin güvenliğini sağlamak amacıyla tasarlanmıştır. Sistem yöneticisinin (Admin) veritabanının anlık bir kopyasını alabilmesine, bu yedekleri saklayabilmesine ve gerektiğinde dışa aktarabilmesine imkân tanır.

Bu ekran üzerinden kullanıcı:

- Mevcut veritabanının **tek tıklama ile tam yedeğini** alabilir.
- Daha önce alınmış yedekleri **tarih, dosya adı ve boyut bilgileriyle** birlikte liste halinde görüntüleyebilir.
- Alınan bir yedeği **dosya olarak indirerek** harici ortamlarda saklayabilir.
- Artık ihtiyaç duyulmayan yedekleri **kontrollü şekilde silebilir**.

Yedekleme işlemi, PostgreSQL'in yerleşik ve endüstri standardı aracı olan **pg_dump** kullanılarak gerçekleştirilmektedir. Bu sayede:

- Sadece tablo verileri değil; **constraint'ler, index'ler, ilişkiler ve veri bütünlüğü** de yedeğe dahil edilir.
- Alınan yedekler, PostgreSQL üzerinde **eksiksiz ve güvenli şekilde geri yüklenebilir** niteliktedir.
- Yedek dosyaları, uygulama sunucusunda App_Data/Backups dizini altında saklanarak uygulama dışından da erişilebilir hale getirilir.

8. Süreci Hızlandıran Otomasyon Scriptleri

Bir projeyi geliştirirken benim için mesele yalnızca istenen özelliklerin çalışması değildir. Asıl önemli olan, bu özelliklere **nasıl bir süreçle** ulaşıldığıdır. Aynı sonucu elde etmenin onlarca yolu olabilir; ancak mühendislik bakış açısı, bu yollar arasından **zamanı, eforu ve hata ihtimalini en iyi yöneten** olanı seçmeyi gerektirir.

Bu nedenle, tekrar eden veya manuel yapıldığında hem yavaşlatıcı hem de hata üretmeye açık olan her adımı, mümkün olduğunca otomasyona dökmeye çalışırım. Bunu "ekstra" bir uğraş olarak değil, doğrudan geliştirmenin bir parçası olarak görürüm. Çünkü bana göre mühendis, yalnızca problemi çözen kişi değil; **problemi sistematik, ölçeklenebilir ve sürdürülebilir biçimde çözen kişidir**.

Bu projeyi geliştirirken de aynı yaklaşımla hareket ettim ve geliştirme sürecini ciddi şekilde hızlandıran iki yardımcı script yazdım. Bu scriptler doğrudan uygulamanın bir parçası olmasa da, projenin ortaya çıkış sürecinde önemli bir rol oynadı.

Script 1 – Otomatik Ürün Görseli Seedleme (image_seeder.py)

Bu projede ürün yönetimi yalnızca isim, kategori ve fiyat bilgisinden ibaret değil; kullanıcı deneyimini iyileştirmek için her ürünün görselinin de sisteme eklenmesi gerekiyor. Ancak onlarca ürünü tek tek internetten görsel bulup indirerek, uygun formatta düzenleyip, veritabanına manuel olarak yüklemek hem zaman kaybı hem de hata üretmeye son derece açık bir süreç.

Bu noktada, bu süreci otomatikleştirilmesi gereken bir süreç olarak ele aldım ve bu ihtiyacı karşılamak amacıyla image_seeder.py scriptini yazdım.

Script'in temel amacı; veritabanında görseli olmayan ürünleri tespit edip, her ürün için anlamlı bir arama terimi üreterek otomatik şekilde görsel bulmak ve bu görselleri doğrudan veritabanına kaydetmektir. Script çalıştığında:

- Veritabanındaki **resmi olmayan ürünler** tespit edilir.
- Ürün adı ve kategori bilgisine göre **bağlama uygun bir arama terimi** üretilir.
- Her ürün için Bing üzerinden **tek ve yeterli bir görsel** indirilir.
- İndirilen görsel, uygun MIME tipiyle birlikte **binary (byte) olarak veritabanına yazılır**.
- Geçici dosyalar temizlenir ve süreç tamamen izole şekilde tamamlanır.

Script 2 – Proje Dizin ve Metot Haritalayıcı (dizin_olusturucu.py)

Bu proje büyüdükçe, yalnızca "kaç dosya var" değil; **hangi dosyada hangi sorumluluklar var, hangi sınıf hangi metotları barındırıyor** soruları da önem kazanmaya başladı. Özellikle mimariyi anlatırken, projeyi devralacak bir geliştiriciye ya da değerlendiren bir kişiye yapıyı net göstermek manuel olarak oldukça zahmetliydi.

Bu ihtiyacı karşılamak için dizin_olusturucu.py scriptini geliştirdim. Script'in amacı, projeyi yüzeysel bir klasör ağacı olarak değil; **metot seviyesine kadar inen okunabilir bir mimari harita** haline getirmektir. Script çalıştırıldığında:

- Proje dizini recursive olarak taranır.
- bin, obj, wwwroot, Migrations gibi **gürültü oluşturan klasörler bilinçli olarak hariç tutulur**.
- Yalnızca mimariyi anlatan .cs dosyalarına odaklanılır.
- Her C# dosyası satır satır analiz edilerek:
- public / private / async / override gibi erişim belirleyicilere sahip **gerçek metotlar** tespit edilir.
- if, for, while gibi yanlış pozitif oluşturabilecek yapılar ayıklanır.
- Ortaya çıkan yapı, klasör → dosya → metot hiyerarşisini koruyacak şekilde **tek bir okunabilir çıktı dosyası** haline getirilir.

Bu script'i özellikle:

- Dokümantasyon hazırlarken
- Mimariyi tartışırken
- Projeyi sonradan gözden geçirirken

aktif olarak kullandım. Buradaki temel kazanım, kodu yazarken değil; **kodu anlatırken ve anlamlandırırken zaman kazanmak** oldu.

9. Sonuç ve Değerlendirme

Bu proje, benim için yalnızca staj kapsamında geliştirilen bir restoran yönetim sistemi değil; yazılım geliştirme sürecine bakış açımın ciddi şekilde şekillendiği bir öğrenme alanı olmuştur. Daha önce .NET, web tabanlı geliştirme ve PostgreSQL ile doğrudan bir proje geliştirmemiş olmama rağmen, bu çalışma sayesinde bu teknolojileri gerçek bir problem alanı üzerinde bir araya getirme fırsatı buldum.

Proje sürecinde en önemli kazanımımdan biri, bir yazılım sisteminin yalnızca "çalışıyor" olmasının yeterli olmadığıdır. Sipariş, stok, ödeme ve rezervasyon gibi birden fazla aktörün aynı anda işlem yaptığı bir yapıda; **eş zamanlılık (concurrency)**, **veri tutarlılığı**, **iş kuralı bütünlüğü** ve **hata yönetimi** gibi kavramların sistemin merkezinde yer alması gerektiğini deneyimleyerek öğrendim. Özellikle transaction yönetimi, row-level locking (FOR UPDATE), partial unique constraint ve UPSERT gibi veritabanı özelliklerini, teoride kalmadan pratikte kullanma imkânı buldum.

Katmanlı mimari **(Controller–Service–Repository)** yaklaşımı sayesinde; HTTP akışı, iş kuralları ve veri erişimi net biçimde ayrılmıştır. Controller'ların yalnızca kullanıcı etkileşimi ve yönlendirme ile ilgilenmesi, Service katmanının iş akışlarını ve transaction sınırlarını yönetmesi, Repository katmanının ise SQL ve veritabanı detaylarını izole etmesi; kodun okunabilirliğini ve sürdürülebilirliğini ciddi ölçüde artırmıştır. Bu yapı, ileride yeni bir ekran, yeni bir iş kuralı ya da farklı bir arayüz eklenmesi durumunda sistemin kontrollü biçimde genişletilebilmesini mümkün kılmaktadır.

Hata yönetimi konusunda **OperationResult** yaklaşımını benimsemek, benim için önemli bir dönüm noktası olmuştur. Her hatanın exception olmadığı, bazı durumların iş kuralı ihlali olarak ele alınması gerektiği fikrini bu projede net biçimde içselleştirdim. Bu sayede controller katmanı, veritabanı hata kodları veya teknik detaylar yerine yalnızca işlem sonucu ve kullanıcıya gösterilecek mesaj ile ilgilenir hale gelmiştir.

Kullanıcı deneyimi tarafında ise klasik MVC yaklaşımını, **Fetch/AJAX** tabanlı etkileşimlerle destekledim. Sipariş alma ekranında ürün ekleme, çıkarma ve miktar güncelleme işlemlerinin sayfa yenilenmeden yapılması; hem performansı hem de kullanım hissini belirgin biçimde iyileştirmiştir. Buna karşın, istemci tarafı manipülasyon riskine karşı tüm kritik kontrollerin server tarafında tekrar yapılması gerektiğini de bu süreçte öğrendim.

Sonuç olarak bu proje; benim için yalnızca yeni teknolojiler öğrenilen bir çalışma değil, aynı zamanda **nasıl düşünerek yazılım geliştirilmesi gerektiğini** öğreten bir deneyim olmuştur. Aldığım kararların büyük bölümü kısa vadeli çözümlerden ziyade, sistemin uzun vadeli davranışı ve bakım maliyeti göz önünde bulundurularak verilmiştir. Bu açıdan bakıldığında, proje hem teknik hem de mühendislik bakış açısı açısından benim için önemli bir kazanım olmuştur.