

ANKARA ÜNİVERSİTESİ
MÜHENDİSLİK FAKÜLTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ



BLM4531 – AĞ TABANLI TEKNOLOJİLER VE UYGULAMALARI

“CHORD” PROJE FİNAL RAPORU

Canlı Demo: <https://chord.borak.dev>

Github Repo: <https://github.com/brckfrc/chord>

Video: <https://youtu.be/o0pG1QwQ9CI>

Bora KOCABIYIK

21290270

15/01/2026

İÇİNDEKİLER

İÇİNDEKİLER	i
1. GİRİŞ	1
2. PROJE MİMARİSİ VE TEKNOLOJİ YIĞINI	2
3. BACKEND MİMARİSİ	3
3.1. <u>Teknoloji Yığını</u>	3
3.2. <u>API Mimarisi ve Uç Noktalar (Endpoints)</u>	4
3.3. <u>Gerçek Zamanlı İletişim Mimarisi: SignalR</u>	4
3.4. <u>Ses ve Görüntü Mimarisi: LiveKit</u>	6
3.5. <u>Güvenlik Mekanizmaları</u>	6
4. FRONTEND MİMARİSİ	7
4.1. <u>Teknoloji Yığını</u>	7
4.2. <u>Durum Yönetimi ve Mimarisi</u>	8
5. VERİTABANI MİMARİSİ	9
6. GELİŞTİRME, TEST VE DAĞITIM SÜREÇLERİ	10
6.1. <u>Geliştirme Fazları ve Yönetimi</u>	10
6.2. <u>Test Stratejisi</u>	10
6.3. <u>Dağıtım (Deployment) Mimarisi ve CI/CD</u>	11
7. SONUÇ	13
KAYNAKÇA	14
EKLER	15
Ek 1 – <u>Mevcut Endpoint ve İşlevleri Tablosu</u>	16
Ek 2 – <u>Chord Web Arayüzü</u>	20

1. GİRİŞ

Chord projesi, yalnızca Discord benzeri bir platform oluşturma hedefinin ötesinde, modern, ölçeklenebilir ve dayanıklı bir gerçek zamanlı iletişim sisteminin mimari zorluklarına uygulamalı bir çözüm getirme vizyonu ile tasarlanmıştır.

Projenin temel amacı, dağıtık sistemlerde durum yönetimi, çoklu ortam akışları için NAT geçişi (NAT Traversal) ve yatay ölçeklenebilirlik gibi endüstri standardı problemlerin üstesinden gelmek için WebSocket (SignalR) ve WebRTC (LiveKit SFU) gibi teknolojilerin stratejik entegrasyonunu gerçekleştirmektir.

Projenin ana motivasyonu, yalnızca bir sohbet uygulaması geliştirmek değil, aynı zamanda güvenli kimlik doğrulama (JWT), rol tabanlı yetkilendirme, dağıtık sistemlerde durum yönetimi (Redis) ve sıfır kesintili dağıtım (Blue-Green Deployment) gibi profesyonel yazılım geliştirme yaşam döngüsünün kritik bileşenlerini uygulamaktır. Bu sayede, teorik ağ bilgisi ile endüstri standardı mühendislik pratikleri arasında bir köprü kurulması hedeflenmiştir.

Proje kapsamında başarıyla tamamlanan temel özellikler aşağıda listelenmiştir:

- **Gerçek Zamanlı Mesajlaşma:** SignalR WebSocket altyapısı ile anlık metin tabanlı iletişim.
- **Sunucular (Guilds) ve Kanallar:** Kullanıcıların topluluklar oluşturup metin ve ses kanalları üzerinden etkileşim kurabildiği yapı. Her sunucu oluşturulduğunda otomatik olarak “general” adında bir metin kanalı ve “Lobby” adında bir ses kanalı oluşturulur. Kanal pozisyonları, kanal tipine göre (Text, Voice, Announcement) ayrı ayrı yönetilir; her tip kendi pozisyon sıralamasına sahiptir (Text: 0,1,2... Voice: 0,1,2...). Sunucular, kullanıcının katılma tarihine göre sıralanır (en yeni katıldığı sunucu en üstte).
- **Ses ve Görüntü Sohbeti:** LiveKit SFU (Selective Forwarding Unit) mimarisi ile 10’dan fazla kullanıcıyı destekleyen, yüksek kaliteli ve ölçeklenebilir sesli/görüntülü sohbet odaları.
- **Direkt Mesajlar ve Arkadaşlık Sistemi:** Birebir özel mesajlaşma, arkadaş ekleme, kabul etme, reddetme ve engelleme işlevleri.
- **Dosya Yükleme:** Resim, video ve belgelerin (25MB limitli) paylaşılabilmesi için MinIO S3 uyumlu nesne depolama entegrasyonu. MinIO, Amazon S3 API’si ile uyumlu olup, kendi kendine barındırılabilen (self-hosted) bir nesne depolama çözümüdür. Yüklenen dosyalar otomatik olarak presigned URL’ler ile güvenli bir şekilde erişilebilir hale getirilir.
- **Rol Tabanlı İzin Sistemi:** Sunucu sahiplerinin özel roller oluşturarak üyelere granüler yetkiler atayabildiği esnek bir yetkilendirme sistemi.
- **Kullanıcı Varlığı (Presence):** Çevrimiçi, Boşta, Rahatsız Etmeyin ve Görünmez gibi durum bilgileri.
- **Gelişmiş Mesaj Özellikleri:** Mesajlara tepki verme (reaction), mesaj düzenleme/silme, @mention ile kullanıcılardan bahsetme ve yazıyor göstergesi.
- **Mesaj Sabitleme (Pinned Messages):** Önemli mesajların kanallarda sabitlenmesi ve kolay erişim için sabitlenmiş mesajların listelenmesi.
- **Okunmamış Mesaj Takibi:** Hem kanallar hem de direkt mesajlar için kullanıcı bazlı okunmamış mesaj sayısı takibi ve son okunan mesaj pozisyonu kaydı.

- **Sunucu Davetleri (Guild Invites):** Paylaşılabilir davet kodları ile sunuculara katılım, davet süresi ve kullanım limiti yönetimi.
- **Profil Fotoğrafları ve İkonlar:** Kullanıcı avaturları ve sunucu ikonları için otomatik görüntü işleme (ImageSharp ile 256x256 WebP formatına dönüştürme ve kırpma).
- **Duyuru Kanalları (Announcement Channels):** Özel içerik yayınlamak için tasarlanmış duyuru tipi kanallar.
- **Denetim Kayıtları (Audit Logs):** Sunucu sahiplerinin sunucudaki önemli eylemleri (üye atılması, kanal oluşturulması vb.) takip edebilmesi için denetim kayıtları.
- **Health Check Sistemi:** Veritabanı, Redis ve MinIO servislerinin durumunu izleyen sağlık kontrolü endpoint'i (/health).

Bu raporun devamında, projenin teknik mimarisi, kullanılan teknoloji yığını, geliştirme süreçleri, test stratejileri ve dağıtım mimarisi detaylı bir şekilde ele alınacaktır.

2. PROJE MİMARİSİ VE TEKNOLOJİ YİĞİNİ

Chord projesinin mimarisi, ölçeklenebilirlik, sürdürülebilirlik ve platformlar arası uyumluluk hedefleri doğrultusunda stratejik olarak çok katmanlı bir yapıda tasarlanmıştır. Bu yapı; backend, frontend, veritabanı, gerçek zamanlı iletişim katmanı, depolama ve dağıtım gibi bileşenleri birbirinden bağımsız ancak uyum içinde çalışacak şekilde ayırır. Bu modüler yaklaşım, projenin farklı platformlara (web, mobil) kolayca entegre olabildiğini sağlamaktadır. Her katman, kendi alanında endüstri standardı olan teknolojiler kullanılarak inşa edilmiştir.

Aşağıdaki tablo, projenin ana katmanlarını ve bu katmanlarda kullanılan temel teknolojileri özetlemektedir:

Katman	Teknolojiler	Amaç
Backend	.NET 9, SignalR, EF Core 9	API, gerçek zamanlı iletişim, veri erişimi ve iş mantığı.
Frontend	React 18, TypeScript, Redux Toolkit	Kullanıcı arayüzü, bileşen tabanlı yapı ve merkezi durum yönetimi.
Veritabanı ve Önbellek	SQL Server, Redis	Kalıcı veri depolama, önbellekleme ve SignalR için yatay ölçeklendirme (backplane).
Ses/Görüntü	LiveKit (WebRTC SFU), Coturn (STUN/TURN)	Gerçek zamanlı, çok kullanıcı ve ölçeklenebilir ses ve video akışı.
Depolama	MinIO (S3 Uyumlu)	Dosya ve medya yüklemeleri için nesne tabanlı depolama çözümü.
CI/CD ve Dağıtım	GitHub Actions, Docker, Blue-Green	Otomatik derleme, test ve sıfır kesintili (zero-downtime) dağıtım süreçleri.
Mobil İstemci	Flutter (iOS)	iOS platformu için tek kod tabanlı ile geliştirilmiş native uygulama deneyimi.

Sonraki bölümlerde, bu mimariyi oluşturan her bir katmanın teknik detayları, tasarım kararları ve işlevleri ayrıntılı olarak incelenecektir.

3. BACKEND MİMARİSİ

Projenin backend mimarisi, yüksek verim ve platformlar arası uyumluluk hedefleri doğrultusunda stratejik olarak ASP.NET Core 9 üzerine tasarlanmıştır. Bu yapı, RESTful API uç noktalarını, gerçek zamanlı WebSocket iletişimini (SignalR) ve LiveKit, MinIO gibi harici servis entegrasyonlarını tek bir çatı altında verimli bir şekilde yönetir. Bu sayede hem web istemcisi hem de mobil uygulama için tutarlı ve güçlü bir altyapı sunulmaktadır.

3.1. Teknoloji Yığını

Backend, projenin gereksinimlerini karşılamak üzere özenle seçilmiş, kendini kanıtlamış teknolojiler ve kütüphanelerden oluşmaktadır. Bu teknolojiler, kategorize edilmiş bir şekilde aşağıda listelenmiştir:

- **Çerçeve ve Dil**
 - **.NET 9.0:** Web API altyapısı ve iş mantığının temelini oluşturan ana çerçeve.
- **Veri Erişimi**
 - **Entity Framework Core 9:** Veritabanı işlemleri için kullanılan nesne-ilişkisel eşleyici (ORM) aracı.
 - **SQL Server:** İlişkisel verilerin kalıcı olarak saklandığı veritabanı sistemi.
- **Gerçek Zamanlı İletişim ve Önbellek**
 - **SignalR:** WebSocket tabanlı anlık mesajlaşma, kullanıcı varlığı ve ses kanalı bildirimleri için kullanılan gerçek zamanlı iletişim kütüphanesi.
 - **Redis:** Yüksek performanslı bir anahtar-değer deposu. Uygulamada hem önbellekleme hem de SignalR'in birden fazla sunucu üzerinde ölçeklenmesini sağlayan "backplane" mekanizması için kullanılır.
- **Ses ve Görüntü**
 - **LiveKit:** WebRTC SFU (Selective Forwarding Unit) tabanlı, ölçeklenebilir ses ve video konferans sunucusu.
 - **Coturn:** NAT arkasındaki kullanıcıların bağlantı kurmasını sağlayan STUN/TURN sunucusu.
- **Depolama**
 - **MinIO:** Dosya yüklemeleri için kullanılan, S3 API uyumlu, kendi kendine barındırılabilen nesne depolama sistemi.
- **Güvenlik**
 - **JWT 8.2:** Kullanıcı kimlik doğrulama ve yetkilendirme için JSON Web Token tabanlı bir mekanizma.
 - **BCrypt:** Parolaların güvenli bir şekilde hash'lenerek veritabanında saklanması için kullanılan bir algoritma.

- **Yardımcı Araçlar**

- **Serilog 9:** Yapılandırılabilir ve esnek uygulama loglaması için kullanılır.
- **AutoMapper 12:** DTO (Data Transfer Object) ve veritabanı varlıkları (Entity) arasında nesne eşlemeyi otomatize eder.
- **FluentValidation 11:** Gelen isteklerin sunucu tarafında doğrulanması için akıcı ve zincirlenebilir bir doğrulama kütüphanesi.
- **ImageSharp:** Sunucu tarafında profil fotoğrafları gibi görselleri işlemek ve optimize etmek (örneğin, WebP formatına dönüştürme) için kullanılır.

3.2. API Mimarisi ve Uç Noktalar (Endpoints)

Backend API'si, RESTful mimari prensiplerine uygun olarak tasarlanmıştır. API, temel kaynaklar (Kullanıcı, Sunucu, Kanal, Mesaj vb.) etrafında mantıksal olarak gruplandırılmış uç noktalardan oluşur. Bu yapı, API'nin anlaşılabilirliğini ve istemciler tarafından kolayca tüketilmesini sağlar.

Ana API kaynak gruplarından Authentication ile alakalı endpointler aşağıdaki tabloda gösterilmiştir:

- **Authentication**

Metot	Endpoint	Açıklama
POST	/api/Auth/register	Yeni bir kullanıcı kaydı oluşturur.
POST	/api/Auth/login	Kullanıcı girişi yaparak JWT token'ları döndürür.
POST	/api/Auth/refresh	Refresh token kullanarak yeni access token alır.
GET	/api/Auth/me	O anki oturum açmış kullanıcının bilgilerini getirir.
POST	/api/Auth/logout	Kullanıcı çıkışı yapar.
GET	/api/Auth/me/unread-summary	Tüm kanallar için okunmamış mesaj özetini getirir.
PATCH	/api/Auth/me/status	Kullanıcı durumunu günceller.

Bunun haricindeki tüm endpointler rapor sonunda bulunan EK-1'de tablo şeklinde görüntülenebilir.

3.3. Gerçek Zamanlı İletişim Mimarisi: SignalR

Projedeki anlık mesajlaşma, kullanıcı varlığı (presence) ve diğer gerçek zamanlı bildirimler SignalR ile sağlanmaktadır. SignalR, sunucu ve istemciler arasında kalıcı bir çift yönlü bağlantı kurarak sunucunun istemcilere anında veri göndermesine olanak tanır. Uygulamanın yatayda, yani birden fazla sunucuya dağıtılarak ölçeklenebilmesi için SignalR, mimarinin

vazgeçilmez bir parçası olan **Redis backplane** ile yapılandırılmıştır. Bu mekanizma olmadan, A Sunucusuna bağlı bir kullanıcıdan gelen mesaj, B Sunucusuna bağlı bir kullanıcıya asla ulaşamazdı. Redis, tüm sunucu örneklerinin paylaştığı bir mesajlaşma kanalı (message bus) görevi görerek her olayın tüm sisteme yayınlanmasını garanti eder ve böylece yatay ölçeklendirmeyi mümkün kılar.

Uygulamada iki ana SignalR Hub'ı bulunmaktadır:

1. **ChatHub**: Metin mesajlaşması, sesli kanal durum bildirimleri, yazıyor göstergeleri ve direkt mesajlaşma gibi ana iletişim işlevlerini yönetir.
2. **PresenceHub**: Kullanıcıların çevrimiçi/çevrimdışı durumlarını yönetir ve bu bilgiyi tüm istemcilere yayınlar.

ChatHub için Önemli Metotlar ve Olaylar:

- **İstemciden Sunucuya Çağrılan Metotlar (Client → Server):**

Metod	Açıklama
SendMessage	Bir kanala yeni bir mesaj gönderir.
JoinVoiceChannel	Bir sesli kanala katıldığını bildirir.
Typing	Kullanıcının bir kanalda yazdığını diğerlerine bildirir.
SendDMMessage	Bir direkt mesaj kanalına mesaj gönderir.
JoinDM	Bir direkt mesaj kanalına katılır.
LeaveDM	Bir direkt mesaj kanalından ayrılır.
TypingInDM	Direkt mesaj kanalında yazıyor göstergesi gönderir.
MarkDMAsRead	Direkt mesaj kanalını okundu olarak işaretler.
PinMessage	Bir mesajı kanalda sabitler.
UnpinMessage	Bir mesajın sabitlemesini kaldırır.

- **Sunucudan İstemciye Gönderilen Olaylar (Server → Client):**

Metod	Açıklama
ReceiveMessage	Yeni bir mesaj alındığında tetiklenir.
MessageEdited	Bir mesaj düzenlendiğinde tetiklenir.
MessagePinned	Bir mesaj sabitlendiğinde tetiklenir.
MessageUnpinned	Bir mesajın sabitlemesi kaldırıldığında tetiklenir.

UserJoinedVoiceChannel	Bir kullanıcı sesli kanala katıldığında tetiklenir.
UserLeftVoiceChannel	Bir kullanıcı sesli kanaldan ayrıldığında tetiklenir.
DMReceiveMessage	Direkt mesaj kanalında yeni bir mesaj alındığında tetiklenir.
DMMessagesEdited	Direkt mesaj düzenlendiğinde tetiklenir.
DMMessagesDeleted	Direkt mesaj silindiğinde tetiklenir.
DMUserTyping	Direkt mesaj kanalında bir kullanıcı yazıyor göstergesi.
DMMarkAsRead	Direkt mesaj kanalı okundu olarak işaretlendiğinde tetiklenir.
UserMentioned	Bir kullanıcı bir mesajda bahsedildiğinde tetiklenir.

3.4. Ses ve Görüntü Mimarisi: LiveKit

Projenin sesli ve görüntülü sohbet altyapısı, geleneksel P2P (Peer-to-Peer) mimarisi yerine **LiveKit SFU (Selective Forwarding Unit)** mimarisini kullanmaktadır. P2P mimarisinde her kullanıcı diğer tüm kullanıcılara ayrı bir bağlantı kurmak zorundadır, bu da katılımcı sayısı arttıkça istemci üzerindeki bant genişliği ve işlemci yükünü katlanarak artırır. SFU mimarisinde ise her istemci sadece sunucuya (LiveKit) tek bir bağlantı kurar. Sunucu, gelen medya akışlarını alır ve odadaki diğer katılımcılara seçici olarak iletir. Bu yaklaşım, istemci kaynak tüketimini önemli ölçüde azaltır ve 10'dan fazla kullanıcının aynı odada sorunsuzca iletişim kurmasını sağlayarak yüksek ölçeklenebilirlik sunar.

Ayrıca, kullanıcıların farklı ağ yapılandırmaları (NAT, güvenlik duvarı vb.) arkasındayken bile birbirleriyle bağlantı kurabilmelerini sağlamak için bir **Coturn** sunucusu entegre edilmiştir. Coturn, STUN ve TURN protokollerini kullanarak NAT geçişini (NAT Traversal) mümkün kılar.

Sistemin düzgün çalışması için gereken ağ portları aşağıda belirtilmiştir:

Port	Protokol	Servis	Amaç
7880	TCP	LiveKit	WebSocket sinyalleşmesi
7881	UDP/TCP	LiveKit	RTC medya akışı
3478	UDP/TCP	Coturn	STUN/TURN

3.5. Güvenlik Mekanizmaları

Uygulamanın güvenliği, hem kullanıcı verilerini hem de sistem bütünlüğünü korumak amacıyla çok katmanlı bir yaklaşımla sağlanmıştır. Temel güvenlik önlemleri şunlardır:

- **Kimlik Doğrulama:** JWT (JSON Web Token) tabanlı bir sistem kullanılır. Kullanıcı giriş yaptığında kısa ömürlü bir Access Token ve daha uzun ömürlü bir Refresh Token alır. Access Token, korumalı API'lere erişim için kullanılırken, süresi dolduğunda Refresh Token ile yeni bir Access Token alınır.

- **Parola Güvenliği:** Kullanıcı parolaları, veritabanına kaydedilmeden önce **BCrypt** algoritması ile geri döndürülemez bir şekilde hash'lenir.
- **Global Exception Handler Middleware:** Tüm uygulama genelinde yakalanmamış hataları yakalayan ve tutarlı bir hata yanıtı formatı döndüren middleware. Development ortamında detaylı hata mesajları ve stack trace, production ortamında ise genel hata mesajları döndürür. Farklı exception tipleri (UnauthorizedAccessException, KeyNotFoundException, ArgumentException vb.) için uygun HTTP status kodları atanır.
- **Hız Sınırlama (Rate Limiting) Middleware:** Kötü niyetli otomatik saldırıları (Brute-force vb.) önlemek amacıyla, her IP adresi için dakikada 100 istek sınırı uygulanmıştır. Health check endpoint'leri rate limiting'den muaf tutulmuştur. Ayrıca, performans testleri için X-Load-Test: true header'ı ile isteklerde rate limiting bypass özelliği yapılandırılabilir şekilde sunulmuştur.
- **Audit Log Middleware:** HTTP isteklerinde IP adresi ve User-Agent bilgilerini yakalayıp audit log kayıtları için hazırlayan middleware. Bu bilgiler, sunucularda gerçekleşen önemli eylemlerin (üye atma, rol değiştirme vb.) denetim kayıtlarında kullanılır.
- **CORS (Cross-Origin Resource Sharing):** Sadece izin verilen istemci (frontend) adreslerinden gelen isteklere izin veren katı CORS politikaları uygulanır.
- **Sunucu Tarafı Doğrulama:** İstemciden gelen tüm veriler, sunucu tarafında **FluentValidation** kütüphanesi kullanılarak doğrulanır. Bu, geçersiz veya kötü niyetli verilerin işlenmesini engeller.
- **SQL Injection Koruması:** Entity Framework Core, parametrelili sorguları otomatik olarak kullanarak SQL enjeksiyon saldırılarına karşı doğal bir koruma sağlar.
- **Health Check Endpoint:** Sistemin kritik bileşenlerinin (veritabanı, Redis, MinIO) durumunu izlemek için /health endpoint'i sağlanmıştır. Bu endpoint, Docker container health check'leri ve CI/CD pipeline'larında sistem sağlığını doğrulamak için kullanılır.

Bu sağlam backend mimarisi, projenin kullanıcı arayüzünü oluşturan frontend katmanı için güvenilir bir temel oluşturmaktadır.

4. FRONTEND MİMARİSİ

Projenin kullanıcı arayüzü, modern web geliştirme standartlarına uygun olarak React 18 ve TypeScript kullanılarak geliştirilmiştir. Bileşen tabanlı (component-based) bir yaklaşımla inşa edilen frontend, hem geliştirme sürecini hızlandırmış hem de kodun yeniden kullanılabilirliğini ve bakımını kolaylaştırmıştır. Geliştirme ve derleme aracı olarak **Vite**'in tercih edilmesi, anlık sıcak yeniden yükleme (Hot Module Replacement) özelliği sayesinde geliştirici deneyimini önemli ölçüde iyileştirmiştir. Uygulama genelindeki karmaşık durumların (state) yönetimi ise **Redux Toolkit** ile merkezi ve öngörülebilir bir şekilde sağlanmıştır.

4.1. Teknoloji Yığını

Frontend katmanını oluşturan temel teknolojiler ve kütüphaneler şunlardır:

- **React 18 & TypeScript:** Kullanıcı arayüzünü oluşturmak için kullanılan temel kütüphane ve kodun tip güvenliğini sağlayarak hataları geliştirme aşamasında yakalamak için kullanılan dil.
- **Vite:** Geleneksel derleyicilere göre çok daha hızlı bir geliştirme sunucusu ve derleme aracı.
- **Redux Toolkit:** Uygulamanın genel durumunu (oturum bilgileri, sunucu listesi, aktif kanal, mesajlar vb.) yönetmek için kullanılan merkezi durum yönetimi kütüphanesi.
- **React Router:** Tek sayfa uygulamasında (SPA) sayfalar arası gezinmeyi ve yönlendirmeyi yönetir.
- **Tailwind CSS & shadcn/ui:** Hızlı ve tutarlı bir şekilde stil geliştirmek için kullanılan bir CSS çatısı ve bu çatı üzerine kurulu, yeniden kullanılabilir UI bileşenleri kütüphanesi.
- **SignalR Client:** Backend ile gerçek zamanlı WebSocket bağlantısı kurarak anlık veri akışını sağlar.
- **React Hook Form & Zod:** Formların yönetimini basitleştiren ve şema tabanlı doğrulama (validation) sağlayan kütüphaneler.

4.2. Durum Yönetimi ve Mimarisi

Uygulama genelindeki durum, Redux Toolkit kullanılarak yönetilmektedir. Bu mimaride, uygulamanın farklı bölümlerine ait durumlar (örneğin, kimlik doğrulama için authSlice, sunucular için guildsSlice) "**slice**" adı verilen modüler ve bağımsız birimlerde tutulur. Bu yapı, durum mantığının karmaşıklığını azaltır ve kodun daha organize olmasını sağlar.

Gerçek zamanlı olaylar, bu mimarinin merkezinde yer alır. Örneğin, yeni bir mesaj geldiğinde iş akışı şu şekilde işler:

1. **SignalR Olayı:** Backend'deki ChatHub, ReceiveMessage olayını tetikler ve yeni mesaj verisini istemciye gönderir.
2. **İstemci Dinleyicisi:** Frontend'deki SignalR istemcisi bu olayı yakalar.
3. **Redux Eylemi:** SignalR dinleyicisi, Redux store'u güncellemek için ilgili slice'taki bir eylemi (action) tetikler. Örneğin, messagesSlice içerisindeki addMessage eylemi çağrılır.
4. **Durum Güncellemesi:** Redux, messagesSlice'in durumunu yeni mesajı içerecek şekilde günceller.
5. **UI Güncellemesi:** React, Redux store'daki bu değişikliği otomatik olarak algılar ve sadece ilgili bileşeni (örneğin, MessageList bileşeni) yeniden render ederek yeni mesajı kullanıcı arayüzünde anlık olarak gösterir.

Bu reaktif veri akışı, kullanıcı deneyimini akıcı hale getirir ve arayüzün her zaman en güncel veriyi yansıtmasını sağlar. Bu sağlam frontend yapısı, uygulamanın veri katmanı olan veritabanı mimarisi üzerine kurulmuştur.

5. VERİTABANI MİMARİSİ

Projenin veri kalıcılığı, güvenilirliği ve esnekliği, Entity Framework Core 9 (EF Core) ve SQL Server veritabanı sistemi üzerine kurulmuştur. Geliştirme sürecinde benimsenen **"Code-First"** yaklaşımı sayesinde, veritabanı şeması doğrudan C# kodunda tanımlanan varlık (entity) sınıflarından otomatik olarak oluşturulmuştur. Bu yaklaşım, veritabanı şemasının uygulama koduyla senkronize bir şekilde evrimleşmesini sağlayarak geliştirme sürecine büyük bir esneklik katmıştır.

Aşağıdaki tablo, sistemin temelini oluşturan ana veri varlıklarını ve işlevsel açıklamalarını özetlemektedir:

Varlık (Entity)	Açıklama
User	Kullanıcıların kimlik doğrulama, profil (kullanıcı adı, e-posta, avatar) ve durum bilgilerini tutar.
Guild	Discord'daki sunuculara karşılık gelir; isim, ikon ve sahip bilgilerini içerir.
GuildMember	Bir kullanıcının bir sunucuya üyeliğini temsil eder; kullanıcı, sunucu ve takma ad bilgilerini bağlar.
Channel	Sunucular içinde yer alan metin, ses ve duyuru kanallarını tanımlar. Kanal pozisyonları, kanal tipine göre (Type) ayrı ayrı yönetilir; her tip kendi pozisyon sıralamasına sahiptir. Bu sayede, metin kanalları ve ses kanalları birbirinden bağımsız olarak sıralanabilir. Veritabanında (GuildId, Type, Position) unique index'i ile aynı tip içinde duplicate pozisyon engellenmiştir.
ChannelReadState	Kullanıcıların kanallardaki son okuma pozisyonunu ve okunmamış mesaj sayısını takip eder.
DirectMessageReadState	Direkt mesaj kanallarındaki okunmamış mesaj takibini yönetir.
MessageReaction	Mesajlara verilen tepkileri (emoji) saklar.
MessageMention	Mesajlarda kullanıcı bahsetmelerini (mentions) ve okunma durumlarını takip eder.
Message	Kanallarda gönderilen, metin ve ek dosyalar içeren sohbet mesajları.
Role	Sunuculardaki kullanıcı rollerini ve bu rollere atanmış izinleri (permissions) tanımlar.
GuildMemberRole	Bir üyenin hangi rollere sahip olduğunu belirten ilişki tablosu.
GuildInvite	Sunuculara katılım için oluşturulan davet kodlarını ve kurallarını (süre, kullanım limiti) saklar.

Veritabanı bütünlüğünü ve veri tutarlılığını sağlamak için çeşitli kısıtlamalar (constraints) etkin bir şekilde kullanılmıştır. Bu kısıtlamalardan bazıları şunlardır:

- **Benzersiz Kısıtlamalar (Unique Constraints):** User.Email ve User.Username gibi alanların tekrarlanmasını engelleyerek veri tekilliğini garanti eder.
- **Bileşik Anahtarlar (Composite Keys):** GuildMember tablosunda (GuildId, UserId) ikilisinin birincil anahtar olması, bir kullanıcının bir sunucuya yalnızca bir kez üye olabildiğini sağlar.
- **Zincirleme Silme Kuralları (Cascade Delete Rules):** Bir Guild silindiğinde, ona bağlı olan tüm Channels, GuildMembers, Roles ve AuditLogs kayıtlarının da otomatik olarak silinmesini sağlayarak veritabanında artık veri (orphan data) kalmasını önler.

Bu yapılandırılmış veritabanı mimarisi, projenin web platformundaki işlevselliğini desteklemenin yanı sıra, mobil platformlardaki varlığı için de sağlam bir zemin hazırlamaktadır.

6. GELİŞTİRME, TEST VE DAĞITIM SÜREÇLERİ

Chord projesi, sadece kod yazmaktan ibaret bir çalışma olmayıp, başından sonuna kadar planlı geliştirme fazları, kapsamlı test stratejileri ve modern CI/CD (Sürekli Entegrasyon/Sürekli Dağıtım) pratiklerini içeren profesyonel bir yazılım yaşam döngüsünü takip etmiştir. Bu yaklaşım, projenin kalitesini, güvenilirliğini ve sürdürülebilirliğini artırmıştır.

6.1. Geliştirme Fazları ve Yönetimi

Proje, karmaşıklığı yönetmek ve geliştirme sürecini öngörülebilir kılmak amacıyla "**FAZ**" olarak adlandırılan aşamalı bir yaklaşımla geliştirilmiştir. Her faz, belirli bir özellik setinin tamamlanmasına odaklanmış ve bir öncekinin üzerine inşa edilmiştir. Bu fazlı yaklaşım, projenin evrimini net bir şekilde ortaya koymaktadır:

- **FAZ 1-2:** Projenin temelleri atıldı. Backend altyapısı, veritabanı modeli, Docker entegrasyonu ve JWT tabanlı kimlik doğrulama sistemi (kayıt, giriş, token yenileme) bu aşamada tamamlandı. Sunucu (guild) ve kanal yönetimi için temel API uç noktaları oluşturuldu.
- **FAZ 3-6:** Gerçek zamanlı iletişim yetenekleri eklendi. SignalR entegrasyonu ile anlık mesajlaşma, kullanıcı varlığı (presence) ve Redis backplane ile ölçeklenebilirlik sağlandı. Paralel olarak, temel frontend yapısı React ile kuruldu ve backend API'sine entegre edilerek ilk kullanıcı arayüzleri hayata geçirildi.
- **FAZ 7-9.5:** Gelişmiş özellikler eklendi. MinIO ile dosya yükleme, LiveKit SFU entegrasyonu ile çok kullanıcılı sesli ve görüntülü sohbet, bitfield tabanlı rol ve izin sistemi, direkt mesajlaşma ve arkadaşlık sistemi gibi platformu zenginleştiren kritik işlevler bu fazlarda tamamlandı.

6.2. Test Stratejisi

Projenin kalitesini ve kararlılığını güvence altına almak için çok katmanlı bir test stratejisi uygulanmıştır. Bu strateji, hem backend hem de frontend katmanlarını kapsamaktadır.

- **Backend Testleri:**
 - **Birim Testleri (Unit Tests):** Backend iş mantığının doğruluğunu kontrol etmek amacıyla **xUnit** ve **Moq** kütüphaneleri kullanılarak **88 adet** birim testi yazılmıştır. Bu testler, özellikle AuthService (%100) ve GuildService (%95) gibi

kritik servislerde yüksek kod kapsamı (code coverage) oranlarına ulaşarak sistemin temel işlevlerinin güvenilirliğini sağlamıştır.

- **Frontend Testleri:**

- **Uçtan Uca (E2E) Testler:** Kullanıcı deneyimini gerçek bir tarayıcı ortamında simüle etmek için **Playwright** kullanılmıştır. Bu testler, docker-compose.test.yml ile oluşturulan izole bir Docker test ortamında çalıştırılır. Kayıt olma, sunucuya katılma, mesaj gönderme gibi kritik kullanıcı akışları otomatik olarak test edilerek arayüzdeki regresyon hataları erkenden tespit edilir.

- **Performans Testleri:**

- **Load Testing (K6):** API performansını yüksek eşzamanlı kullanıcı yükü altında doğrulamak için **K6** kullanılmıştır. Test senaryoları, 1.000 eşzamanlı kullanıcıya kadar ölçeklenebilir bir yük deseni ile kritik endpoint'leri (kimlik doğrulama, sunucu yönetimi, mesajlaşma, ses token alma) kapsar. Testler, p95 yanıt süresinin 500ms altında kalması, hata oranının %1'in altında olması ve minimum 100 istek/saniye throughput değerlerini doğrular.

6.3. Dağıtım (Deployment) Mimarisi ve CI/CD

Proje, farklı barındırma ihtiyaçlarına cevap verebilmek amacıyla esnek dağıtım seçenekleri sunmaktadır. Bu seçenekler, Standalone (tek başına çalışan sunucu), Standard VPS (mevcut bir ters proxy arkasında) ve YunoHost (kendi kendine barındırma platformu) olmak üzere üç farklı senaryo için yapılandırılmıştır. Tüm senaryolarda, custom domain (özel alan adı) yapılandırması desteklenmektedir. Standalone senaryoda Caddy reverse proxy otomatik olarak Let's Encrypt SSL sertifikası alır ve HTTPS sağlar. Standard VPS ve YunoHost senaryolarında ise mevcut reverse proxy (Nginx, Traefik, Apache) üzerinden custom domain yapılandırması yapılır. Proje, production ortamında chord.borak.dev custom domain'i ile yayınlanmaktadır; bu domain yapılandırması başlangıçta roadmap'te planlanmamış olup, deployment sürecinde ihtiyaç duyulduğu için eklenmiştir.

Projenin en gelişmiş dağıtım özelliği, **Blue-Green Dağıtım Stratejisi**'dir. Bu strateji, uygulamanın yeni bir versiyonu canlıya alınırken sıfır kesinti (zero-downtime) yaşanmasını sağlar. Süreç şu şekilde işler:

1. Mevcut versiyon ("Blue" ortamı) canlıda çalışmaya devam eder.
2. Yeni versiyon, canlı trafik almayan ikinci bir ortamda ("Green" ortamı) ayağa kaldırılır ve test edilir.
3. Testler başarılı olduğunda, gelen trafik anında "Green" ortamına yönlendirilir. Bu geçişin kullanıcılar için anlık ve kesintisiz olmasının sırrı, ters proxy (örneğin Nginx veya Caddy) yapılandırmasındaki tek ve atomik bir değişikliğe dayanmasıdır.
4. Eski "Blue" ortamı, bir sorun anında geri dönmek için bir süre bekletilir ve ardından kapatılır.

Bu tüm süreç, **GitHub Actions** kullanılarak kurulan bir CI/CD boru hattı ile otomatize edilmiştir. Pipeline, iki ayrı workflow dosyasından oluşur ve bu workflow yapılandırması başlangıçta roadmap'te detaylı olarak planlanmamış olup, deployment ihtiyaçları doğrultusunda geliştirme sürecinde eklenmiştir:

CI Workflow (ci.yml): Her push ve pull request'te tetiklenir ve şu adımları içerir: * Backend birim testleri çalıştırılır (xUnit, 88 test case). * Frontend lint ve build işlemleri yapılır (ESLint, Vite production build). * Docker imajları build edilir (cache ile optimize edilmiş, push edilmez). * Dokümantasyon değişiklikleri (**.md, docs/**, chord_roadmap.md) otomatik olarak CI'ı atlar (paths-ignore ile). * Commit mesajında [skip ci], [ci skip], [no ci] anahtar kelimelerinden biri varsa workflow tamamen atlanır.

Deploy Workflow (deploy.yml): CI workflow başarıyla tamamlandıktan sonra otomatik olarak tetiklenir (workflow_run trigger): * CI workflow'un başarılı olduğu doğrulanır (manuel tetiklemede son CI run kontrol edilir). * Backend ve frontend için yeni Docker imajları oluşturulur ve GitHub Container Registry (GHCR)'a push edilir. * SSH üzerinden production sunucusuna bağlanılır ve blue-green deployment script'i (scripts/deploy.sh) çalıştırılır. * Health check'ler yapılır (/health endpoint'leri) ve başarısız olursa otomatik rollback gerçekleşir. * Commit mesajında [skip deploy] veya [deploy skip] anahtar kelimesi varsa deployment atlanır.

main branch'ine yapılan her bir kod itme (push) işlemi, otomatik olarak bu iki aşamalı süreci tetikler ve test edilmiş, doğrulanmış kodun production ortamına (chord.borak.dev) kesintisiz bir şekilde dağıtılmasını sağlar.

7. SONUÇ

Chord projesi, başlangıçta belirlenen hedeflere başarıyla ulaşarak modern ağ teknolojilerini ve ölçeklenebilir mimari prensiplerini etkin bir şekilde uygulayan, kapsamlı ve işlevsel bir gerçek zamanlı iletişim platformu ortaya koymuştur. Proje, Discord benzeri bir deneyim sunma vizyonunu, hem teknik derinlik hem de kullanıcı odaklı özellikler açısından karşılamıştır.

Projenin geliştirme süreci sonucunda, gerçek zamanlı mesajlaşma, çok kullanıcıli ses/görüntü sohbeti, esnek rol ve izin yönetimi, direkt mesajlar, arkadaşlık sistemi ve dosya paylaşımı gibi temel işlevler başarıyla tamamlanmıştır. SignalR ve LiveKit gibi teknolojilerin entegrasyonu, platformun yüksek performanslı ve ölçeklenebilir bir yapıya kavuşmasını sağlamıştır. Docker, GitHub Actions ve Blue-Green dağıtım stratejisi gibi modern DevOps pratiklerinin benimsenmesi ise projenin profesyonel standartlarda geliştirilip yönetilmesine olanak tanımıştır.

Bu süreçte elde edilen en önemli teknik ve mimari dersler şunlardır:

- Proje, çok kullanıcıli ses/görüntü iletişiminde **SFU mimarisinin** P2P'ye olan stratejik üstünlüğünü kanıtlamıştır. P2P, istemci tarafında N^2 karmaşıklığında bağlantı yönetimi gerektirirken, SFU bu yükü sunucuya taşıyarak istemci kaynaklarını serbest bırakmış ve 10'dan fazla katılımcıyla kararlı oturumları mümkün kılmıştır. Bu karar, projenin ölçeklenebilirlik hedeflerine ulaşmasında kritik rol oynamıştır.
- Yatay ölçeklenmenin kritik olduğu gerçek zamanlı uygulamalarda **Redis backplane**'in mimari bir zorunluluk olduğu teyit edilmiştir. Redis, birden fazla sunucu örneği arasında tutarlı bir durum ve olay yayılımı sağlayarak, sistemin tek bir sunucunun kapasitesinin ötesine geçmesine olanak tanımıştır. Bu bileşen olmadan, platform yatayda ölçeklenemez ve tek bir hata noktasına (single point of failure) mahkum kalırdı.
- **CI/CD otomasyonu ve Blue-Green dağıtım stratejisi**, yalnızca bir kolaylık değil, aynı zamanda projenin geliştirme hızını ve kararlılığını artıran temel bir mühendislik pratiği olmuştur. Otomatik test ve dağıtım süreçleri, insan hatasını en aza indirirken, sıfır kesintili güncellemeler son kullanıcı deneyimini korumuş ve geliştirme ekibinin yeni özellikleri güvenle canlıya almasını sağlamıştır.

Sonuç olarak Chord, mevcut haliyle sağlam, güvenilir ve zengin özelliklere sahip bir platformdur. Proje dokümantasyonunda belirtilen "Gelecek Özellikler" (Upcoming Features) başlığı altındaki bildirim ayarları ve mesaj arama gibi işlevlerin eklenmesiyle daha da geliştirilme potansiyeline sahiptir.

KAYNAKÇA

1. Microsoft. (2024). *ASP.NET Core Documentation*.
<https://learn.microsoft.com/en-us/aspnet/core/>
2. Microsoft. (2024). *SignalR Documentation*.
<https://learn.microsoft.com/en-us/aspnet/core/signalr/introduction>
3. LiveKit. (2024). *LiveKit Documentation*. <https://docs.livekit.io/>
4. Redis Labs. (2024). *Redis Documentation*. <https://redis.io/docs/>
5. Meta. (2024). *React Documentation*. <https://react.dev/>
6. Redux Toolkit. (2024). *Redux Toolkit Documentation*. <https://redux-toolkit.js.org/>
7. Docker Inc. (2024). *Docker Documentation*. <https://docs.docker.com/>
8. GitHub. (2024). *GitHub Actions Documentation*. <https://docs.github.com/en/actions>
9. MinIO. (2024). *MinIO Documentation*. <https://min.io/docs/>
10. Entity Framework Core. (2024). *EF Core Documentation*.
<https://learn.microsoft.com/en-us/ef/core/>
11. Playwright. (2024). *Playwright Documentation*. <https://playwright.dev/>
12. K6. (2024). *K6 Documentation*. <https://k6.io/docs/>

9. EKLER

EK-1 - Mevcut Endpoint ve İşlevleri Tablosu

EK-2 - Chord Web Arayüzü

EK-1 - Mevcut Endpoint ve İşlevleri Tablosu

- **Guilds (Sunucular)**

Metot	Endpoint	Açıklama
POST	/api/Guilds	Yeni bir sunucu oluşturur.
GET	/api/Guilds	Kullanıcının üye olduğu sunucuları listeler.
GET	/api/Guilds/{id}	Belirli bir sunucunun detaylarını getirir.
PUT	/api/Guilds/{id}	Sunucu bilgilerini günceller.
DELETE	/api/Guilds/{id}	Belirtilen sunucuyu siler (sadece sahip).
GET	/api/Guilds/{id}/members	Sunucudaki tüm üyeleri listeler.
POST	/api/Guilds/{id}/members	Sunucuya yeni bir üye ekler.
DELETE	/api/Guilds/{id}/members/{userId}	Sunucudan bir üyeyi çıkarır.

- **Channels (Kanallar)**

Metot	Endpoint	Açıklama
POST	/api/guilds/{guildId}/Channels	Belirtilen sunucuya yeni bir kanal ekler.
GET	/api/guilds/{guildId}/Channels	Sunucudaki kanalları listeler.
GET	/api/guilds/{guildId}/Channels/{id}	Belirli bir kanalın detaylarını getirir.
PUT	/api/guilds/{guildId}/Channels/{id}	Kanal bilgilerini günceller.
DELETE	/api/guilds/{guildId}/Channels/{id}	Kanalı siler.

- **Messages (Mesajlar)**

Metot	Endpoint	Açıklama
POST	/api/channels/{channelId}/Messages	Belirtilen kanala yeni bir mesaj gönderir.
GET	/api/channels/{channelId}/Messages	Kanalın mesajlarını sayfalı olarak getirir.

GET	/api/channels/{channelId}/Messages/{messageId}	Belirli bir mesajın detaylarını getirir.
PUT	/api/channels/{channelId}/Messages/{messageId}	Mesajı düzenler.
DELETE	/api/channels/{channelId}/Messages/{messageId}	Mesajı siler.
GET	/api/channels/{channelId}/Messages/last-read	Son okunan mesaj ID'sini getirir.
POST	/api/channels/{channelId}/messages/{messageId}/pin	Mesajı sabitlet.
DELETE	/api/channels/{channelId}/messages/{messageId}/pin	Mesaj sabitlemeyi kaldırır.
GET	/api/channels/{channelId}/pins	Sabitlenmiş mesajları listeler.
POST	/api/channels/{channelId}/mark-read	Kanalı okundu olarak işaretler.
GET	/api/channels/{channelId}/unread-count	Okunmamış mesaj sayısını getirir.

- **Reactions (Tepkiler)**

Metot	Endpoint	Açıklama
GET	/api/messages/{messageId}/Reactions	Mesajdaki tepkileri getirir.
POST	/api/messages/{messageId}/Reactions	Mesaja bir tepki ekler.
DELETE	/api/messages/{messageId}/Reactions/{emoji}	Mesajdan bir tepkiyi kaldırır.

- **Mentions (@Bahsetme)**

Metot	Endpoint	Açıklama
GET	/api/Mentions	Kullanıcının bahsedildiği mesajları listeler.
GET	/api/Mentions/unread-count	Okunmamış bahsetme sayısını getirir.

PATCH	/api/Mentions/{id}/mark-read	Bir bahsetmeyi okundu olarak işaretler.
PATCH	/api/Mentions/mark-all-read	Tüm bahsetmeleri okundu olarak işaretler.

- **Invites (Davetler)**

Metot	Endpoint	Açıklama
POST	/api/Invites/guilds/{guildId}	Yeni bir davet kodu oluşturur.
GET	/api/Invites/{code}	Davet bilgilerini getirir.
POST	/api/Invites/{code}/accept	Daveti kabul ederek sunucuya katılır.
GET	/api/Invites/guilds/{guildId}	Sunucudaki tüm davetleri listeler.
DELETE	/api/Invites/{inviteId}	Bir daveti iptal eder.

- **Friends (Arkadaşlar)**

Metot	Endpoint	Açıklama
POST	/api/Friends/request	Başka bir kullanıcıya arkadaşlık isteği gönderir.
GET	/api/Friends	Kabul edilmiş arkadaşları listeler.
GET	/api/Friends/pending	Bekleyen arkadaşlık isteklerini listeler.
GET	/api/Friends/blocked	Engellenmiş kullanıcıları listeler.
POST	/api/Friends/{id}/accept	Arkadaşlık isteğini kabul eder.
POST	/api/Friends/{id}/decline	Arkadaşlık isteğini reddeder.
POST	/api/Friends/{userId}/block	Bir kullanıcıyı engeller.
DELETE	/api/Friends/{userId}/block	Bir kullanıcının engelini kaldırır.
DELETE	/api/Friends/{friendId}	Bir arkadaş listeden çıkarır.

- **Direct Messages (Direkt Mesajlar)**

Metot	Endpoint	Açıklama
-------	----------	----------

GET	/api/dms	Kullanıcının tüm DM kanallarını listeler.
GET	/api/dms/{id}	Belirli bir DM kanalını getirir.
POST	/api/dms/users/{userId}	Bir kullanıcı ile DM kanalı oluşturur veya mevcut kanalı getirir.
GET	/api/dms/{id}/messages	DM kanalındaki mesajları sayfalı olarak getirir.
POST	/api/dms/{id}/messages	DM kanalına mesaj gönderir.
PUT	/api/dms/{id}/messages/{messageId}	DM mesajını düzenler.
DELETE	/api/dms/{id}/messages/{messageId}	DM mesajını siler.
POST	/api/dms/{id}/mark-read	DM kanalını okundu olarak işaretler.

- **Upload (Dosya & Video Yükleme)**

Metot	Endpoint	Açıklama
POST	/api/Upload	Genel dosya yükleme.
DELETE	/api/Upload	Yüklenmiş bir dosyayı siler.
POST	/api/Upload/avatar	Kullanıcı profil fotoğrafı yükler.
POST	/api/Upload/guild/{guildId}/icon	Sunucu ikonu yükler.

- **Voice (Ses/Video)**

Metot	Endpoint	Açıklama
POST	/api/voice/token	LiveKit SFU için erişim token'ı oluşturur.
GET	/api/voice/room/{channelId}	Sesli kanal odasının durumunu getirir.

- **Audit Logs (Denetim Kayıtları)**

Metot	Endpoint	Açıklama
GET	/api/guilds/{guildId}/audit-logs	Sunucudaki denetim kayıtlarını getirir.

EK-2 - Chord Web Arayüzü

