Tauri, Vite, React ve Rust ile Üretim Odaklı Masaüstü Uygulama Geliştirme Kılavuzu

Bölüm 1: Proje Temelleri ve İlk Yapılandırma

Bu temel bölüm, Tauri mimarisinin temel kavramlarının anlaşılmasını sağlayarak, tüm önkoşulların karşılandığı temiz ve iyi yapılandırılmış bir projenin oluşturulması sürecinde yol gösterecektir. Bu sağlam temel, uygulamanın sonraki geliştirme aşamalarında karşılaşılabilecek karmaşıklıkları yönetmeyi kolaylaştıracaktır.

1.1. Gerekli Geliştirme Ortamının Hazırlanması

Başarılı bir Tauri projesi geliştirmek için belirli araçların ve bağımlılıkların sisteme doğru bir şekilde kurulması zorunludur. Her bir bileşen, geliştirme ve derleme sürecinin farklı bir aşamasında kritik rol oynar.

- **Rust Kurulumu:** Tauri'nin arka planı (backend) Rust ile yazılmıştır. Rust derleyicisi ve paket yöneticisi Cargo'yu edinmenin en güvenilir yolu resmi rustup kurulum yöneticisini kullanmaktır. rustup, farklı Rust sürümleri arasında geçiş yapmayı ve araç zincirlerini (toolchains) yönetmeyi kolaylaştırır.[1]
- **Node.js ve Paket Yöneticisi:** Tauri'nin ön yüzü (frontend) modern web teknolojilerini kullanır ve bu teknolojilerin geliştirme ortamı Node.js üzerine kuruludur. Node.js ile birlikte gelen npm paket yöneticisi kullanılabilir, ancak bu kılavuzda verimliliği ve hızı nedeniyle pnpm tercih edilecektir. Proje bağımlılıklarını yönetmek ve geliştirme sunucusunu çalıştırmak için bir Node.js ortamı ve paket yöneticisi gereklidir.[2]
- **Sistem Bağımlılıkları:** Tauri, son kullanıcıya sunulacak olan nihai yerel uygulamayı derlemek için işletim sistemine özgü bazı kütüphanelere ve derleme araçlarına ihtiyaç duyar. Örneğin, Linux'ta webkit2gtk geliştirme kütüphaneleri, Windows'ta Microsoft Visual Studio C++ build tools ve macOS'ta Xcode Command Line Tools gereklidir. Bu bağımlılıklar, Tauri'nin web içeriğini göstermek için kullandığı sistemin yerel WebView'i ile etkileşime girmesini sağlar. Kurulum talimatları Tauri'nin resmi dokümantasyonunda platform bazında detaylandırılmıştır.[3]

Bu üç temel bileşenin doğru bir şekilde yapılandırılması, geliştirme sürecinin sorunsuz ilerlemesi için ilk ve en önemli adımdır.

1.2. create-tauri-app ile Proje İskeletini Oluşturma

Tauri, yeni bir projeye başlamayı önemli ölçüde hızlandıran create-tauri-app adında bir komut satırı aracı (CLI) sunar. Bu araç, farklı ön yüz kütüphaneleri için optimize edilmiş şablonlar kullanarak projenin temel iskeletini oluşturur.[4]
Projeyi oluşturmak için terminalde aşağıdaki komut çalıştırılır:

Bash

```
pnpm create tauri-app
```

Bu komut, interaktif bir kurulum süreci başlatır. Bu kılavuzun hedeflerine en uygun, modern ve sürdürülebilir bir yapı için aşağıdaki seçimlerin yapılması tavsiye edilir [2]:

1. **Project name:** Uygulamanız için anlamlı bir isim (örneğin, kurumsal-masaustu-uygulamasi).
2. **Choose which language to use for your frontend:** TypeScript / JavaScript seçeneği ile devam edilmelidir.
3. **Choose your package manager:** pnpm seçilerek hem disk alanı hem de kurulum hızı açısından avantaj sağlanır.
4. **Choose your UI template:** React seçilerek modern ve bileşen tabanlı bir arayüz altyapısı tercih edilir.
5. **Choose your UI flavor:** TypeScript seçeneği, büyük ölçekli ve bakımı kolay uygulamalar geliştirmek için kritik olan tür güvenliği (type safety) avantajını sunar. Bu, kodun daha güvenli ve öngörülebilir olmasını sağlar.[6]

Bu adımlar tamamlandığında, create-tauri-app aracı, seçilen teknolojilere uygun olarak yapılandırılmış, çalışmaya hazır bir proje dizini oluşturacaktır.[3]

1.3. Proje Yapısını Anlamak: Frontend ve Backend Ayrımı

Oluşturulan proje dizini, Tauri'nin mimari felsefesini yansıtan mantıksal bir yapıya sahiptir. Bu yapı, ön yüz ve arka plan katmanlarını net bir şekilde ayırarak bağımsız geliştirme ve yönetim imkanı tanır.[7]

- **Kök Dizin (Frontend Projesi):** Projenin kök dizini, standart bir Vite projesidir. Burada package.json, vite.config.ts, index.html gibi ön yüze ait dosyalar bulunur. Bu yapı, ön yüz geliştiricilerinin aşina oldukları bir ortamda çalışmalarını sağlar.[9]
- **src/ Dizini:** Bu dizin, React uygulamasının kaynak kodunu barındırır. Kullanıcı arayüzünü oluşturan .tsx uzantılı bileşenler, stiller ve diğer ön yüz varlıkları burada yer alır.
- **src-tauri/ Dizini (Backend Projesi):** Bu dizin, kendi içinde tam teşekküllü bir Rust

projesidir. Tauri'ye özgü bazı ek dosyalarla birlikte standart bir Cargo projesi yapısını takip eder.[8]

- **Cargo.toml:** Rust projesinin bağımlılıklarını (crate'ler) ve meta verilerini yöneten manifest dosyasıdır.
- **tauri.conf.json:** Tauri uygulamasının merkezi yapılandırma dosyasıdır. Uygulama kimliğinden pencere ayarlarına, eklentilerden derleme komutlarına kadar her şey bu dosyadan kontrol edilir.
- **src/main.rs ve src/lib.rs:** Rust uygulamasının giriş noktalarıdır. Modern Tauri yapısında, temel uygulama mantığı lib.rs içinde tanımlanır. Bu, kodun test edilebilirliğini artırır ve gelecekte mobil platformlara taşınabilirliği kolaylaştırır. main.rs ise sadece masaüstü uygulamasını başlatan basit bir sarmalayıcı (wrapper) görevi görür.[8]
- **icons/:** Farklı işletim sistemleri için kullanılacak uygulama ikonlarını içerir.
- **capabilities/:** Uygulamanın güvenlik modelinin merkezinde yer alır. Ön yüzün hangi Rust komutlarını çağırabileceği veya hangi API'lere erişebileceği gibi izinler bu dizindeki JSON dosyaları aracılığıyla tanımlanır.

Bu çift proje yapısı, Tauri'nin en güçlü yönlerinden biridir. Ön yüz ve arka plan ekiplerinin, aralarındaki tek sözleşme olan IPC (Inter-Process Communication) katmanı üzerinden, birbirlerinden bağımsız ve paralel olarak çalışmalarına olanak tanır. Bu, daha büyük ve karmaşık projelerde geliştirme verimliliğini önemli ölçüde artırır.[10]

1.4. tauri.conf.json: Uygulamanızın Kontrol Merkezi

tauri.conf.json dosyası, uygulamanın davranışını ve derleme sürecini tanımlayan en kritik dosyadır. Vite ve React ile oluşturulmuş bir proje için bu dosyadaki bazı ayarlar hayati öneme sahiptir.[6]

- **build Nesnesi:** Bu nesne, geliştirme ve üretim derleme süreçlerinin nasıl yürütüleceğini tanımlar.
  - beforeDevCommand: "pnpm dev" - Geliştirme modunda (tauri dev), Tauri'nin ön yüz geliştirme sunucusunu başlatmak için çalıştıracağı komuttur.
  - beforeBuildCommand: "pnpm build" - Üretim derlemesi (tauri build) öncesinde, ön yüz kodunu statik dosyalara (HTML, CSS, JS) dönüştürmek için çalıştırılacak komuttur.
  - devPath: "http://localhost:5173" - Vite geliştirme sunucusunun varsayılan olarak hizmet verdiği adrestir. Tauri'nin WebView'i, geliştirme sırasında içeriği bu adresten yükler.
  - distDir: "../dist" - pnpm build komutu çalıştırıldıktan sonra üretilen statik dosyaların bulunduğu dizine olan göreli yoldur. Tauri, üretim derlemesinde bu dizindeki dosyaları nihai uygulama paketine gömer.
- **package Nesnesi:** Uygulamanın adı (productName) ve sürümü (version) gibi paketleme bilgilerini içerir. Sürüm bilgisini package.json dosyasıyla senkronize tutmak için "version":

"../package.json" şeklinde ayarlanabilir, bu da sürüm yönetimini merkezileştirir.[12]

- **tauri.bundle.identifier:** Uygulamanın işletim sistemi genelinde benzersiz kimliğidir. Bu kimlik, otomatik güncellemeler, bildirimler ve işletim sistemi entegrasyonu için kritik öneme sahiptir.[13] Genellikle ters alan adı notasyonu (reverse domain name notation) kullanılır (örneğin, com.sirketim.uygulamam).
- **tauri.windows:** Uygulamanın ana penceresinin başlığı, boyutu, yeniden boyutlandırılabilir olup olmadığı gibi özelliklerini tanımlar.

Aşağıdaki tablo, bir Vite+React projesi için temel tauri.conf.json ayarlarını özetlemektedir.

Tablo 1.1: tauri.conf.json Temel Geliştirme Yapılandırması

Ayar (Setting)	Değer (Value)	Açıklama (Description)
build.beforeDevCommand	"pnpm dev"	Geliştirme sunucusunu başlatır.
build.beforeBuildCommand	"pnpm build"	Üretim için ön yüzü derler.
build.devPath	"http://localhost:5173"	Geliştirme sunucusunun URL'si.
build.distDir	"../dist"	Üretim varlıklarının göreli yolu.
package.version	"../package.json"	Sürümü package.json'dan okur.
tauri.bundle.identifier	com.sirket.uygulama	Uygulamanın benzersiz kimliği.

Bu ayarların doğru yapılması, geliştirme ve derleme süreçlerinin hatasız çalışmasını garanti altına alır.[14]

1.5. Geliştirme Ortamını Başlatma ve İlk Çalıştırma

Tüm yapılandırmalar tamamlandıktan sonra, geliştirme ortamını başlatmak için tek bir komut yeterlidir.[2] Projenin kök dizininde aşağıdaki komut çalıştırılır:

Bash

```
pnpm tauri dev
```

Bu komut, arka planda bir dizi işlemi koordine eder:

1. tauri.conf.json dosyasında tanımlanan beforeDevCommand komutunu (pnpm dev) çalıştırarak Vite geliştirme sunucusunu başlatır. Bu sunucu, Hızlı Modül Değişimi (Hot Module Replacement - HMR) özelliği sayesinde kod değişikliklerinin anında arayüze yansımasını sağlar.
2. Yine tauri.conf.json'da belirtilen devPath adresinin (http://localhost:5173) erişilebilir olmasını bekler.
3. Rust arka planını debug modunda derler. İlk derleme, tüm bağımlılıkların (crate'ler)

indirilip derlenmesi gerektiği için biraz zaman alabilir. Ancak sonraki derlemeler, Cargo'nun önbellekleme mekanizması sayesinde çok daha hızlı olacaktır.[2]

4. Derleme tamamlandığında, işletim sisteminin yerel penceresini başlatır. Bu pencere, içeriğini Vite geliştirme sunucusundan yükleyerek uygulamayı görüntüler.

Bu entegre süreç, hem ön yüzün esnekliğini ve hızını hem de Rust'ın gücünü tek bir komutla birleştirerek verimli bir geliştirme döngüsü sunar.

Bölüm 2: Güvenli Frontend-Backend İletişimi

Bir Tauri uygulamasının kalbi, React ile yazılmış ön yüz ile Rust ile güçlendirilmiş arka plan arasındaki güvenli ve verimli iletişim katmanıdır. Bu bölüm, bu iletişimin temelini oluşturan IPC mimarisini, harici sunucularla güvenli bağlantı kurma yöntemlerini ve bu süreçlerde en iyi pratikleri detaylandıracaktır.

2.1. Tauri'nin IPC Mimarisi: invoke ve events

Tauri, paylaşılan bellek gibi daha riskli yöntemler yerine, doğası gereği daha güvenli olan Eşzamansız Mesajlaşma (Asynchronous Message Passing) adı verilen bir IPC modeli kullanır. Bu model, internetteki istemci-sunucu iletişimine benzer şekilde çalışır; süreçler, basit veri formatlarında serileştirilmiş istek ve yanıtları değiş tokuş eder. Bu yaklaşım, alıcı sürecin gelen istekleri doğrulayıp, kötü niyetli veya geçersiz bulduklarını reddetmesine olanak tanır.[15]
Tauri, bu mimari içinde iki temel iletişim mekanizması sunar:

- **Komutlar (invoke):** Bu, bir istek-yanıt modelidir. Ön yüzün, Rust'ta tanımlanmış bir fonksiyonu çağırması ve ondan bir sonuç beklemesi gerektiği durumlar için idealdir. invoke kullanımı, uygulamanın içinde tür güvenliğine sahip (type-safe) bir API çağrısı yapmaya benzer. Fonksiyon çağrılır, işlem tamamlanır ve bir veri veya hata geri döner.[16]
- **Olaylar (events):** Bu, "ateşle ve unut" (fire-and-forget) prensibine dayalı, tek yönlü bir mesajlaşma sistemidir. Arka planın, ön yüzden doğrudan bir istek gelmeksizin onu bilgilendirmesi gereken durumlar için kullanılır. Örneğin, uzun süren bir işlemin ilerleme durumunu bildirmek, sistem genelinde bir durum değişikliğini haber vermek veya anlık bildirimler göndermek gibi senaryolar için mükemmeldir.[18]

Bu iki mekanizma, farklı iletişim ihtiyaçları için optimize edilmiştir ve doğru senaryoda doğru mekanizmanın kullanılması, uygulamanın performansını ve sürdürülebilirliğini artırır.

2.2. Rust Tarafında Komutlar (#[tauri::command]) Oluşturma

Ön yüzden çağrılabilir bir fonksiyon oluşturmak için Rust tarafında #[tauri::command] makrosu kullanılır. Bu makro, sıradan bir Rust fonksiyonunu Tauri'nin IPC sistemi tarafından tanınabilir hale getirir.[17]

Örnek Komut Tanımlaması:

Rust

```
// src-tauri/src/lib.rs

#[tauri::command]
fn greet(name: String) -> String {
   format!("Merhaba, {}! Rust tarafından selamlar.", name)
}
```

Tanımlanan bu komutların ön yüz tarafından erişilebilir olması için, tauri::Builder'a kaydedilmeleri gerekir. Bu işlem tauri::generate_handler! makrosu ile yapılır:

Rust

```
// src-tauri/src/lib.rs

#[cfg_attr(mobile, tauri::mobile_entry_point)]
pub fn run() {
   tauri::Builder::default()
     .invoke_handler(tauri::generate_handler![greet])
     .run(tauri::generate_context!())
     .expect("error while running tauri application");
}
```

İleri Düzey Komut Özellikleri:

- **Argümanlar:** JavaScript'ten gönderilen argümanlar, Rust fonksiyon parametreleriyle eşleştirilir. Tauri'nin bu veriyi doğru bir şekilde seriden çıkarabilmesi (deserialize) için tüm argüman türlerinin serde::Deserialize özelliğini (trait) uygulaması gerekir.[16]
- **Özel Parametreler:** Tauri, komut fonksiyonlarına otomatik olarak bazı özel türleri enjekte edebilir. Bu, uygulama durumu ve bağlamına erişim için güçlü bir mekanizma sağlar:
    - tauri::Window: Komutu çağıran pencereye bir referans.
    - tauri::AppHandle: Uygulamanın geneline erişim sağlayan bir referans.
    - tauri::State<T>: Tauri tarafından yönetilen global bir duruma (state) erişim.[16]
- **Asenkron Komutlar:** Dosya okuma, ağ isteği yapma gibi uzun sürebilecek işlemler için komutlar async fn olarak tanımlanmalıdır. Bu, işlemin ayrı bir iş parçacığı havuzunda (thread pool) çalışmasını sağlayarak uygulamanın ana iş parçacığının (ve dolayısıyla kullanıcı arayüzünün) donmasını engeller.[16]

Rust

```
#[tauri::command]
async fn long_running_task() -> String {
    // Uzun süren bir işlemi simüle et
    tokio::time::sleep(std::time::Duration::from_secs(2)).await;
    "İşlem tamamlandı!".to_string()
}
```

2.3. React Arayüzünden Rust Fonksiyonlarını Güvenle Çağırma

Ön yüz tarafında Rust komutlarını çağırmak için @tauri-apps/api paketi kullanılır. Bu paket, IPC iletişimi için gerekli fonksiyonları sağlar.[6]
@tauri-apps/api/core modülünden invoke fonksiyonu import edilir ve Rust'ta tanımlanan komutun adıyla çağrılır. invoke fonksiyonu, bir Promise döndürür.[17]
React Bileşeni İçinde invoke Kullanımı:

TypeScript

```
import { useState } from 'react';
import { invoke } from '@tauri-apps/api/core';

function Greeter() {
  const [greeting, setGreeting] = useState('');
  const [name, setName] = useState('');

  const callGreetCommand = async () => {
    try {
      const result = await invoke<string>('greet', { name });
      setGreeting(result);
    } catch (error) {
      console.error('Komut çağrılırken hata oluştu:', error);
      setGreeting('Bir hata oluştu.');
    }
  };

  return (
    <div>
```

```
      <input
        type="text"
        value={name}
        onChange={(e) => setName(e.target.value)}
        placeholder="Adınızı girin"
      />
      <button onClick={callGreetCommand}>Selamla</button>
      <p>{greeting}</p>
    </div>
  );
}

export default Greeter;
```

Güvenlik Katmanı: capabilities
Tauri, "en az ayrıcalık ilkesi" (principle of least privilege) prensibine sıkı sıkıya bağlıdır. Bir komutun ön yüzden çağrılabilmesi için, src-tauri/capabilities/default.json (veya ilgili yetenek dosyası) içinde açıkça izin verilmiş olması gerekir. Bu, potansiyel bir güvenlik açığı durumunda saldırı yüzeyini en aza indirir.[8]
Örnek capabilities Yapılandırması:

JSON

```
{
  "$schema": "../gen/schemas/desktop-schema.json",
  "identifier": "default",
  "description": "Varsayılan yetenekler",
  "windows": ["main"],
  "permissions": [
    "core:default",
    {
      "identifier": "allow-greet",
      "allow": [{ "name": "greet" }]
    }
  ]
}
```

Bu yapılandırma, sadece greet adlı komutun çağrılmasına izin verir.

2.4. Veri Aktarımı ve Kapsamlı Hata Yönetimi (Result<T, E>)

Rust ve JavaScript arasındaki veri aktarımı serde kütüphanesi aracılığıyla JSON formatında yapılır. Bu nedenle, Rust'tan döndürülen tüm veri yapılarının serde::Serialize özelliğini uygulaması gerekir.[17]
Hata yönetimi için en iyi pratik, Rust komutlarından Result<T, E> türü döndürmektir. Bu yapı, JavaScript Promise'leri ile mükemmel bir uyum içinde çalışır:

- Ok(T) döndürüldüğünde, JavaScript Promise'i T değeri ile resolve olur.
- Err(E) döndürüldüğünde, JavaScript Promise'i E değeri ile reject olur.

Bu sayede ön yüzde async/await ile try/catch blokları veya .then().catch() zincirleri kullanarak temiz ve öngörülebilir bir hata yönetimi yapılabilir.[16]
Örnek Hata Yönetimi:

Rust

```
// Rust tarafı
#
pub enum MyError {
  InvalidInput,
  OperationFailed(String),
}

#[tauri::command]
fn risky_operation(input: String) -> Result<String, MyError> {
  if input.is_empty() {
    Err(MyError::InvalidInput)
  } else {
    //... işlem...
    Ok("Başarılı!".to_string())
  }
}
```

TypeScript

```
// React tarafı
async function callRiskyOperation() {
 try {
  const result = await invoke('risky_operation', { input: '' });
  console.log(result);
 } catch (error) {
  // 'error' değişkeni MyError enum'ının serileştirilmiş halini içerir.
  console.error('İşlem başarısız:', error);
```

```
  }
}
```

Özelleştirilmiş hata türleri tanımlamak, uygulamanın farklı hata durumlarını daha anlamlı bir şekilde işlemesine olanak tanır.

2.5. Harici Sunucularla Güvenli İletişim: tauri-plugin-http ve reqwest

Uygulamanın harici API'ler veya sunucularla iletişim kurması gerektiğinde, bu istekleri doğrudan ön yüzden yapmak yerine Rust arka planı üzerinden yapmak güvenlik açısından daha iyidir. Bu yaklaşım, API anahtarları gibi hassas bilgilerin ön yüz kodunda ifşa olmasını engeller. Tauri, bu amaçla resmi tauri-plugin-http eklentisini sunar. Bu eklenti, Rust ekosisteminin en popüler ve güçlü HTTP istemcisi olan reqwest kütüphanesini temel alır ve onun tüm yeteneklerini Tauri uygulamalarına sunar.[21]

Kurulum ve Kullanım:

1. Eklentiyi kurun: pnpm tauri add http
2. lib.rs dosyasında eklentiyi başlatın: .plugin(tauri_plugin_http::init())
3. Rust komutu içinde reqwest kullanarak istek yapın:

Rust

```
use tauri_plugin_http::reqwest;

#[tauri::command]
async fn fetch_data_from_api() -> Result<String, String> {
    let client = reqwest::Client::new();
    let response = client
        .get("https://api.github.com/repos/tauri-apps/tauri")
        .header("User-Agent", "Tauri-App") // GitHub API'si User-Agent gerektirir
        .send()
        .await
        .map_err(|e| e.to_string())?;

    if response.status().is_success() {
        response.text().await.map_err(|e| e.to_string())
    } else {
        Err(format!("API'den hata alındı: {}", response.status()))
    }
}
```

HTTP İstemcisi için Güvenlik Kapsamı (scope)

tauri-plugin-http eklentisi de Tauri'nin güvenlik modeline tabidir. capabilities/default.json dosyasında bir scope tanımlayarak uygulamanın hangi alan adlarına (domain) istek yapabileceğini kısıtlayabilirsiniz. Bu, bir ön yüz bağımlılığında ortaya çıkabilecek bir güvenlik açığının (supply chain attack), verileri kötü niyetli bir sunucuya sızdırmasını engeller.[22]
Örnek scope Yapılandırması:

JSON

```
{
  "permissions": [
    {
      "identifier": "http:default",
      "allow": [{ "url": "https://api.github.com/repos/tauri-apps/*" }],
      "deny": [{ "url": "https://api.github.com/repos/tauri-apps/private-repo" }]
    }
  ]
}
```

Bu yapılandırma, uygulamanın sadece tauri-apps organizasyonundaki genel GitHub depolarına istek yapmasına izin verirken, private-repo adlı depoya erişimi engeller. Bu katmanlı güvenlik yaklaşımı, Tauri'yi kurumsal düzeyde güvenli uygulamalar geliştirmek için güçlü bir seçenek haline getirir. reqwest gibi güçlü bir aracı sunarken, onu varsayılan olarak güvenli bir sanal alan (sandbox) içinde sınırlandırarak hem yetenek hem de güvenlik sağlar.[24]

Bölüm 3: Kurumsal Düzeyde Loglama Altyapısı

Üretim ortamındaki bir uygulamanın hatalarını ayıklamak ve performansını izlemek için kapsamlı ve yapılandırılabilir bir loglama sistemi vazgeçilmezdir. Bu bölüm, Tauri'nin resmi loglama eklentisini kullanarak, hem geliştirme hem de üretim için sağlam bir loglama altyapısının nasıl kurulacağını adım adım anlatmaktadır.

3.1. tauri-plugin-log Kurulumu ve Temel Yapılandırma

tauri-plugin-log, uygulamanın hem Rust arka planından hem de JavaScript ön yüzünden gelen logları tek bir merkezde toplayan ve yöneten güçlü bir eklentidir.[26]
Kurulum Adımları:

1. **Bağımlılıkları Ekleme:** Eklentiyi projeye eklemek için Tauri CLI kullanılır:
   Bash
   pnpm tauri add log

Bu komut, hem src-tauri/Cargo.toml dosyasına Rust crate'ini (tauri-plugin-log) hem de package.json dosyasına JavaScript paketini (@tauri-apps/plugin-log) ekleyecektir.[27]

2. **Eklentiyi Başlatma:** Eklentinin aktif hale gelmesi için src-tauri/src/lib.rs dosyasındaki tauri::Builder'a kaydedilmesi gerekir:
   Rust
   ```
   // src-tauri/src/lib.rs
   use tauri_plugin_log::LogTarget;

   #[cfg_attr(mobile, tauri::mobile_entry_point)]
   pub fn run() {
      tauri::Builder::default()
        .plugin(
            tauri_plugin_log::Builder::default()
              .targets()
              .build()
        )
        //... diğer eklentiler ve invoke_handler
        .run(tauri::generate_context!())
        .expect("error while running tauri application");
   }
   ```

3. **İzinleri Yapılandırma:** Eklentinin çalışabilmesi için src-tauri/capabilities/default.json dosyasında gerekli iznin verilmesi zorunludur. permissions dizisine "log:default" eklenmelidir [26]:
   JSON
   ```
   {
     "permissions": [
       "core:default",
       "log:default"
       //... diğer izinler
     ]
   }
   ```

Bu temel yapılandırma ile loglama altyapısı kullanıma hazırdır.

3.2. Log Hedefleri: Konsol, Dosya ve WebView

tauri-plugin-log eklentisi, logların birden fazla hedefe aynı anda gönderilmesine olanak tanır. Bu hedefler, geliştirme ve üretim senaryoları için farklı ihtiyaçları karşılar.[27]

- **LogTarget::Stdout:** Logları, uygulamanın başlatıldığı standart terminal konsoluna

yazdırır. Geliştirme sırasında anlık geri bildirim almak için vazgeçilmezdir.

- **LogTarget::Webview:** Rust tarafından üretilen logları, tarayıcının geliştirici konsoluna (WebView inspector) iletir. Bu, ön yüz ve arka plan loglarını aynı arayüzde görerek entegre hata ayıklama yapmayı son derece kolaylaştırır.
- **LogTarget::LogDir:** Logları, işletim sistemine özgü standart uygulama log dizinindeki bir dosyaya yazar. Bu, üretim ortamındaki bir uygulamada oluşan hataları analiz etmek için en önemli hedeftir. Kullanıcılar, bir sorunla karşılaştıklarında bu log dosyasını geliştirici ekibine göndererek sorunun teşhis edilmesine yardımcı olabilirler.

Log dosyalarının konumu işletim sistemine göre değişir [13]:

- **macOS:** $HOME/Library/Logs/{bundleIdentifier}
- **Windows:** %APPDATA%\{bundleIdentifier}\logs
- **Linux:** $HOME/.config/{bundleIdentifier}/logs

Buradaki {bundleIdentifier}, tauri.conf.json dosyasında tanımlanan benzersiz uygulama kimliğidir.

Üretim Odaklı Hedef Yapılandırması:

Rust

```
// src-tauri/src/lib.rs
use tauri_plugin_log::{LogTarget, RotationStrategy};

//...
.plugin(
   tauri_plugin_log::Builder::default()
     .targets([
        // Geliştirme sırasında konsol logları için
        #[cfg(debug_assertions)]
        LogTarget::Stdout,
        #[cfg(debug_assertions)]
        LogTarget::Webview,
        // Üretim ortamında dosya logları için
        LogTarget::LogDir,
      ])
     .rotation_strategy(RotationStrategy::KeepAll) // Log rotasyonunu etkinleştir
     .max_file_size(2_000_000) // Maksimum dosya boyutu (byte cinsinden)
     .build()
)
//...
```

3.3. İleri Düzey Yapılandırma: Filtreleme, Formatlama ve Rotasyon

Eklenti, loglama davranışını daha hassas bir şekilde kontrol etmek için çeşitli ileri düzey yapılandırma seçenekleri sunar.[27]

- **Filtreleme (level):** Üretim ortamında log gürültüsünü azaltmak için belirli bir seviyenin altındaki logların (örneğin, Debug ve Trace) yazdırılmasını engelleyebilirsiniz. .level(log::LevelFilter::Info) metodu ile global bir log seviyesi belirlenebilir. Ayrıca, belirli modüller için farklı seviyeler tanımlamak da mümkündür. Bu mekanizma, standart Rust log crate'inin filtreleme yeteneklerini kullanır.[28]
- **Rotasyon (rotation_strategy):** Log dosyalarının zamanla kontrolsüz bir şekilde büyümesini önlemek için log rotasyonu kullanılır. rotation_strategy(RotationStrategy::KeepAll) ayarı, max_file_size ile belirlenen boyuta ulaşıldığında eski log dosyasını yeniden adlandırarak arşivler ve yeni bir log dosyası oluşturur.
- **Formatlama (format):** Log mesajlarının varsayılan formatını (DATE[LEVEL] MESSAGE) değiştirmek için .format() metoduna bir closure (isimsiz fonksiyon) geçirilebilir. Bu, log çıktısını projenin ihtiyaçlarına göre özelleştirme imkanı tanır.

Rust

```
// src-tauri/src/lib.rs
.plugin(
   tauri_plugin_log::Builder::default()
     .targets()
     .level(log::LevelFilter::Info) // Sadece Info ve üstü seviyeleri logla
     .format(|out, message, record| {
         out.finish(format_args!(
            "{} [{}] - {}",
            chrono::Local::now().format("%Y-%m-%d %H:%M:%S"),
            record.level(),
            message
         ))
      })
     .build()
)
```

3.4. Rust Arka Planından ve React Ön Yüzünden Etkili Loglama

Loglama altyapısı kurulduktan sonra, uygulamanın her iki tarafından da log göndermek oldukça basittir.

- **Rust Tarafından Loglama:** Eklenti, Rust ekosisteminin standart loglama arayüzü olan log crate'i ile entegre çalışır. Cargo.toml dosyanıza log = "0.4" bağımlılığını ekledikten sonra, kodunuzun herhangi bir yerinde standart log makrolarını (log::info!, log::warn!, log::error!, log::debug!) kullanabilirsiniz. Bu makrolar aracılığıyla gönderilen tüm loglar, eklenti yapılandırmasında belirtilen hedeflere otomatik olarak yönlendirilecektir.[26]

Rust

```
// Herhangi bir Rust dosyasında
fn process_data() {
    log::info!("Veri işleme süreci başladı.");
    //...
    log::warn!("Beklenmedik bir değerle karşılaşıldı.");
    //...
    log::error!("Kritik hata: Veri işlenemedi.");
}
```

- **React Tarafından Loglama:** @tauri-apps/plugin-log paketi, JavaScript tarafında kullanılmak üzere info, warn, error gibi fonksiyonlar sunar. Bu fonksiyonlar, React bileşenleri içinden çağrılarak Rust tarafındaki loglama sistemine mesaj gönderilmesini sağlar.[27]

TypeScript

```
// React bileşeni içinde
import { info, error } from '@tauri-apps/plugin-log';

function MyComponent() {
  const handleClick = () => {
    info('Butona tıklandı.');
    try {
      //... bir işlem...
    } catch (e) {
      error(`Bir hata oluştu: ${e}`);
    }
  };

  return <button onClick={handleClick}>İşlemi Başlat</button>;
}
```

- **En İyi Pratik: attachConsole Kullanımı:** Eklentinin en güçlü özelliklerinden biri attachConsole fonksiyonudur. Bu fonksiyon çağrıldığında, tarayıcının yerel console.log, console.warn gibi tüm metodlarını üzerine alır ve bu metodlara gönderilen tüm mesajları otomatik olarak Rust loglama altyapısına yönlendirir.[26] Bu, hem kendi yazdığınız console.log'ların hem de kullandığınız üçüncü parti kütüphanelerin ürettiği konsol çıktılarının merkezi log sistemine dahil edilmesini sağlar. Bu sayede, uygulamanın tamamından gelen loglar tek bir birleşik akışta toplanır.

TypeScript

```
// Uygulamanın giriş noktasında (örn: main.tsx)
import { attachConsole } from '@tauri-apps/plugin-log';

// Konsol loglarını Rust tarafına yönlendirmek için
const unlisten = await attachConsole();

// Bileşen unmount olduğunda veya uygulama kapandığında dinleyiciyi kaldırmak önemlidir
// return () => { unlisten(); };
```

Bu birleşik loglama mimarisi, özellikle üretim ortamında karşılaşılan karmaşık hataların teşhisinde paha biçilmezdir. Kullanıcıdan alınan tek bir log dosyası, hem ön yüz hem de arka plan olaylarının kronolojik ve birbiriyle ilişkili bir kaydını içerir. Bu durum, IPC sınırı boyunca meydana gelen sorunların kök neden analizini önemli ölçüde hızlandırır ve loglamayı basit bir geliştirme yardımcısından güçlü bir üretim teşhis aracına dönüştürür.[30]

Bölüm 4: Otomatik Güncelleme Mekanizması

Modern masaüstü uygulamaları için otomatik güncelleme yeteneği, kullanıcıların her zaman en son özelliklere ve güvenlik yamalarına sahip olmasını sağlamak adına kritik bir özelliktir. Tauri, bu süreci güvenli ve esnek bir şekilde yönetmek için yerleşik bir güncelleme mekanizması sunar. Bu bölüm, bu mekanizmanın nasıl yapılandırılacağını ve kullanılacağını detaylı bir şekilde ele almaktadır.

4.1. Güncelleme Stratejisi ve Güvenlik Esasları

Tauri'nin güncelleme sistemi, güvenliği en ön planda tutar. Bu felsefenin temel taşı, tüm güncelleme paketlerinin kriptografik olarak imzalanmasının zorunlu olmasıdır. Tauri, bu işlem için Ed25519 imza şemasını kullanan Minisign aracını temel alır. Bu imzalama süreci isteğe bağlı değildir ve atlanamaz. Bu zorunluluk, güncelleme paketlerinin geliştiriciden kullanıcıya ulaşana

kadar değiştirilmediğini garanti altına alır ve "ortadaki adam" (man-in-the-middle) gibi saldırıları engeller.[32]

4.2. İmzalama Anahtarlarının Oluşturulması ve Yönetimi

Güncelleme sürecini başlatmak için öncelikle bir anahtar çifti oluşturulmalıdır. Bu işlem, Tauri CLI kullanılarak kolayca gerçekleştirilebilir [33]:

Bash

```
pnpm tauri signer generate -w ~/.tauri/myapp.key
```

Bu komut, iki adet dosya oluşturur ve bu dosyaların yönetimi büyük bir sorumluluk gerektirir:

- **Özel Anahtar (myapp.key):** Bu anahtar, güncelleme paketlerini imzalamak için kullanılır ve **kesinlikle gizli tutulmalıdır**. Güvenli bir yerde (örneğin bir parola yöneticisi veya donanım anahtarı) saklanmalı ve yedeklenmelidir. Bu anahtarın kaybedilmesi, mevcut kullanıcılara yeni güncellemelerin dağıtılamayacağı anlamına gelir, çünkü eski sürümler yeni anahtarla imzalanmış güncellemeleri kabul etmeyecektir.
- **Genel Anahtar (myapp.key.pub):** Bu anahtar, imzaların doğrulanması için kullanılır. Uygulamanın içine gömülür ve herkesle paylaşılması güvenlidir.

Derleme sırasında Tauri'nin özel anahtarı kullanabilmesi için, bu anahtarın yolunun veya içeriğinin bir ortam değişkeni aracılığıyla sağlanması gerekir. .env dosyaları bu işlem için **kullanılamaz**.

- **macOS/Linux:**
  Bash
  ```
  export TAURI_SIGNING_PRIVATE_KEY="/yol/dosya/myapp.key"
  export TAURI_SIGNING_PRIVATE_KEY_PASSWORD="anahtar_sifreniz"
  ```

- **Windows (PowerShell):**
  PowerShell
  ```
  $env:TAURI_SIGNING_PRIVATE_KEY="C:\yol\dosya\myapp.key"
  $env:TAURI_SIGNING_PRIVATE_KEY_PASSWORD="anahtar_sifreniz"
  ```

Bu ortam değişkenleri ayarlandıktan sonra pnpm tauri build komutu çalıştırıldığında, oluşturulan yükleyici dosyaları (.msi, .AppImage, vb.) ve güncelleme paketleri otomatik olarak imzalanacaktır.[33]

4.3. tauri.conf.json İçinde Updater'ın Detaylı Yapılandırılması

Otomatik güncelleyicinin davranışını kontrol etmek için tauri.conf.json dosyasında updater nesnesi yapılandırılmalıdır. Aşağıdaki tablo, bu yapılandırmanın temel parametrelerini açıklamaktadır.[32]

Tablo 4.1: Updater Yapılandırma Parametreleri

Parametre (Parameter)	Tip (Type)	Gerekli mi? (Required?)	Açıklama (Description)
active	boolean	Evet	Güncelleyiciyi etkinleştirir veya devre dışı bırakır. true olarak ayarlanmalıdır.
endpoints	string	Evet	Güncelleme bilgilerini sorgulamak için kullanılacak sunucu URL'lerinin bir dizisi.
pubkey	string	Evet	İmza doğrulaması için kullanılacak genel anahtarın **içeriği**. Dosya yolu değil.
dialog	boolean	Hayır (varsayılan: true)	true ise, bir güncelleme bulunduğunda Tauri yerleşik bir diyalog kutusu gösterir. false ise, güncelleme süreci JavaScript API'si ile manuel olarak yönetilmelidir.
windows.installMode	string	Hayır	Yalnızca Windows için. Güncellemenin nasıl kurulacağını belirler ("passive", "basicUi", "quiet"). "passive" genellikle en iyi kullanıcı deneyimini sunar.

Örnek Yapılandırma:

JSON

```
{
  "tauri": {
    "updater": {
```

```
    "active": true,
    "dialog": false, // Özel bir UI için false yapıldı
    "pubkey": "UPDATE_KEY_PUB_ICERIGI_BURAYA_GELECEK",
    "endpoints": [
      "https://benim-guncelleme-sunucum.com/updates/{{target}}/{{arch}}/{{current_version}}"
    ],
    "windows": {
      "installMode": "passive"
    }
   }
  }
}
```

endpoints dizisindeki URL'lerde kullanılan dinamik değişkenler, sunucunun doğru güncellemeyi sunmasını sağlar:

- {{target}}: İşletim sistemi (windows, linux, darwin).
- {{arch}}: İşlemci mimarisi (x86_64, aarch64, vb.).
- {{current_version}}: Güncellemeyi talep eden uygulamanın mevcut sürümü.

Bu yapı, tek bir sunucu uç noktasının tüm platformlar ve sürümler için güncelleme sunabilmesini sağlar ve aşamalı sürüm dağıtımı gibi gelişmiş senaryolara olanak tanır.[35]

4.4. Güncelleme Sunucusu Yanıt Formatları (Statik ve Dinamik)

Tauri'nin güncelleyicisi, güncelleme bilgilerini almak için iki tür sunucu yapısını destekler:

- **Statik JSON Dosyası:** Güncelleme bilgilerini içeren bir JSON dosyasını GitHub Releases, Amazon S3 gibi statik bir barındırma hizmetinde tutabilirsiniz. Bu, en basit yaklaşımdır.
- **Dinamik Sunucu:** Gelen isteğe göre (örneğin, kullanıcının sürümüne veya lisans durumuna göre) dinamik olarak bir JSON yanıtı üreten bir sunucu uç noktasıdır.

Her iki durumda da, sunucunun döndürmesi gereken JSON formatı standartlaştırılmıştır.
Güncelleme Mevcut Olduğunda Dönen JSON Yanıtı:
Sunucu, bir güncelleme mevcutsa 200 OK durum kodu ile aşağıdaki yapıda bir JSON nesnesi döndürmelidir 32:

JSON

```
{
  "version": "v1.2.0",
  "notes": "Bu sürümdeki yenilikler:\n- Yeni özellik eklendi.\n- Performans iyileştirmeleri yapıldı.",
```

```
  "pub_date": "2023-10-27T14:00:00Z",
  "platforms": {
    "windows-x86_64": {
      "signature": "IMZA_ICERIGI_BURAYA",
      "url": "https://.../my-app-1.2.0-x64.msi.zip"
    },
    "darwin-aarch64": {
      "signature": "IMZA_ICERIGI_BURAYA",
      "url": "https://.../my-app-1.2.0-aarch64.app.tar.gz"
    }
    // Diğer platformlar...
  }
}
```

Burada signature, ilgili güncelleme paketinin .sig dosyasının içeriğidir.
Güncelleme Mevcut Olmadığında Dönen Yanıt:
Dinamik bir sunucu kullanılıyorsa ve bir güncelleme mevcut değilse, sunucu 204 No Content durum kodu ile boş bir yanıt veya 200 OK durum kodu ile boş bir JSON nesnesi ({}) döndürmelidir.32

4.5. React Arayüzünde Özel Güncelleme Deneyimi Oluşturma

"dialog": false olarak ayarlandığında, güncelleme sürecini tamamen kontrol edebilir ve markanıza uygun bir kullanıcı deneyimi oluşturabilirsiniz. Bu, @tauri-apps/api/updater ve @tauri-apps/api/process paketleri kullanılarak yapılır.[32]
Aşağıda, bu süreci yöneten bir React useEffect kancası örneği bulunmaktadır [37]:

TypeScript

```
import { useEffect } from 'react';
import { checkUpdate, installUpdate } from '@tauri-apps/api/updater';
import { relaunch } from '@tauri-apps/api/process';
import { ask } from '@tauri-apps/api/dialog'; // Veya kendi modal bileşeninizi kullanın

function useAppUpdater() {
  useEffect(() => {
    const checkForUpdates = async () => {
      try {
        const { shouldUpdate, manifest } = await checkUpdate();
```

```
      if (shouldUpdate) {
        const userWantsToUpdate = await ask(
          `Yeni bir sürüm (${manifest?.version}) mevcut. Şimdi güncellemek ister
misiniz?\n\nSürüm Notları:\n${manifest?.body}`,
          { title: 'Güncelleme Mevcut', type: 'info' }
        );

        if (userWantsToUpdate) {
          // Güncellemeyi arka planda indir ve kur
          await installUpdate();

          // Kurulum tamamlandıktan sonra uygulamayı yeniden başlat
          await relaunch();
        }
      }
    } catch (error) {
      console.error('Güncelleme kontrolü sırasında hata:', error);
    }
  };

  // Uygulama başladığında güncellemeleri kontrol et
  checkForUpdates();
 },);
}

// Bu kancayı ana App bileşeninizde çağırabilirsiniz:
function App() {
 useAppUpdater();
 //... uygulamanızın geri kalanı
 return <div>...</div>;
}
```

Ayrıca, indirme ilerlemesini kullanıcıya göstermek için "tauri://update-status" olayını dinleyebilirsiniz. Bu, özellikle büyük güncellemeler için daha iyi bir kullanıcı deneyimi sağlar.[32] Bu adımlar, Tauri uygulamanıza güvenli, güvenilir ve kullanıcı dostu bir otomatik güncelleme özelliği eklemek için eksiksiz bir yol haritası sunar.

Bölüm 5: JetBrains IDE'leri ile Entegre Debug Ortamı

Tauri uygulamaları için profesyonel bir hata ayıklama (debugging) ortamı kurmak, özellikle yeni başlayanlar için karmaşık görünebilir. Bunun temel nedeni, geliştirme sırasında çalışan iki ayrı

sürecin (Rust arka planı ve Node.js ön yüz sunucusu) varlığıdır. Bu bölüm, JetBrains IDE'leri olan RustRover (Rust için) ve WebStorm (React için) kullanarak bu iki süreci de kapsayan entegre ve verimli bir hata ayıklama ortamının nasıl kurulacağını adım adım açıklamaktadır.

5.1. Tauri'nin İkili Hata Ayıklama Yapısına Genel Bakış

pnpm tauri dev komutu, geliştirme sürecini basitleştirmek için harika bir soyutlama sunar. Ancak bu soyutlama, arka planda çalışan iki süreci tek bir komut arkasına gizler:

1. **Vite Geliştirme Sunucusu:** React kodunu sunan ve HMR sağlayan bir Node.js süreci.
2. **Tauri Çekirdeği (Core):** Arka plan mantığını yürüten derlenmiş bir Rust süreci.

Standart bir hata ayıklayıcı, bu iki farklı dildeki ve ortamdaki sürece aynı anda kolayca bağlanamaz. Etkili bir hata ayıklama yapabilmek için bu soyutlamayı bilinçli olarak kırmamız, her iki süreci de manuel olarak başlatmamız ve her birine kendi özel hata ayıklayıcısını bağlamamız gerekir. Bu, profesyonel bir Tauri hata ayıklama iş akışının temel mantığıdır.[38]

5.2. WebStorm'da React/Vite Arayüzünü Debug Etme

Ön yüzü debug etmek, standart bir web geliştirme sürecidir ve oldukça basittir. WebStorm, bu işlem için mükemmel araçlar sunar.

1. **JavaScript Debug Yapılandırması Oluşturma:**
    - WebStorm'da Run > Edit Configurations... menüsüne gidin.
    - Sol üst köşedeki + simgesine tıklayın ve JavaScript Debug seçeneğini seçin.[41]
    - Yapılandırmaya "Debug Frontend (Vite)" gibi bir isim verin.
    - **URL** alanına Vite geliştirme sunucusunun adresini girin: http://localhost:5173.
    - Tarayıcı olarak Chrome veya Chromium tabanlı bir tarayıcı seçin.
    - Ayarları kaydedin.
2. **Hata Ayıklama Oturumunu Başlatma:**
    - Öncelikle, terminalde pnpm dev komutunu çalıştırarak Vite sunucusunu başlatın.
    - Sunucu çalıştıktan sonra, WebStorm'da oluşturduğunuz "Debug Frontend (Vite)" yapılandırmasını seçin ve Debug (böcek) simgesine tıklayın.
    - WebStorm, hata ayıklayıcıyı tarayıcıya bağlayacaktır. Artık .tsx dosyalarınızın içine breakpoint'ler koyabilir, bileşenlerin state ve prop'larını inceleyebilir, kodda adım adım ilerleyebilir ve konsolda ifadeleri değerlendirebilirsiniz. Vite'nin ürettiği kaynak haritaları (source maps) sayesinde, yazdığınız orijinal TypeScript/React kodunda hata ayıklama yapabilirsiniz.[42]

5.3. RustRover'da Rust Arka Planını Debug Etme: Adım Adım Kurulum

Rust arka planını debug etmek, Tauri'nin çalışma mantığını anlamayı gerektiren daha incelikli

bir süreçtir. Bu işlem, pnpm tauri dev komutunu tamamen atlayıp, süreci iki ayrı "Run/Debug Configuration" ile manuel olarak yönetmeyi içerir.[39]

5.3.1. Vite Geliştirme Sunucusu için npm Run Configuration Oluşturma

RustRover'ın içinden Vite sunucusunu rahatça başlatabilmek için bir npm yapılandırması oluşturulur. (Bu adım WebStorm'da da yapılabilir ve sunucu oradan başlatılabilir.)

- RustRover'da Run > Edit Configurations... menüsüne gidin.
- + simgesine tıklayıp npm seçeneğini seçin.
- **Name:** Run Vite Dev Server
- **package.json:** Projenizin kök dizinindeki package.json dosyasını seçin.
- **Command:** run
- **Scripts:** dev (Bu, package.json dosyanızdaki dev script'i ile eşleşmelidir).
- **Package manager:** pnpm seçeneğini belirtin.
- Ayarları kaydedin. Bu yapılandırma, sadece Vite sunucusunu çalıştıracaktır, Tauri uygulamasını başlatmayacaktır.

5.3.2. Tauri Çekirdeği için Cargo Run/Debug Configuration Oluşturma

Bu, hata ayıklama kurulumunun en kritik adımıdır. Burada, RustRover'a Tauri'nin Rust sürecini nasıl derleyip hata ayıklayıcı ile başlatacağını öğreteceğiz.[39]

- RustRover'da Run > Edit Configurations... menüsüne gidin.
- + simgesine tıklayıp Cargo seçeneğini seçin.
- **Name:** Debug Tauri Core (Rust)
- **Command:** run
- **Program arguments:** --no-default-features

Bu Argüman Neden Kritik?
--no-default-features bayrağı, Tauri'ye üretim için derlenmiş ön yüz varlıklarını uygulama paketine gömmemesini söyler. Bunun yerine, tauri.conf.json dosyasındaki build.devPath (http://localhost:5173) adresinden içeriği yüklemesi gerektiğini belirtir. Normalde tauri dev komutu bu bayrağı arka planda otomatik olarak geçer. Biz bu komutu atladığımız için, bu davranışı manuel olarak tetiklememiz zorunludur. Aksi takdirde, Tauri uygulaması boş bir pencere ile başlar veya hata verir çünkü Vite sunucusuna bağlanmayı bilmez.39
Aşağıdaki tablo, RustRover'da oluşturulması gereken iki yapılandırmayı özetlemektedir.

Tablo 5.1: RustRover Debug Yapılandırmaları

Yapılandırma (Configuration)	Tip (Type)	Ayar (Setting)	Değer (Value) & Gerekçe (Rationale)
Run Vite Dev Server	npm	Command	run
Run Vite Dev Server	npm	Scripts	dev (Uygulamanın beforeDevCommand

			komutuyla eşleşir)
Debug Tauri Core (Rust)	Cargo	Command	run (Uygulamayı çalıştırır)
Debug Tauri Core (Rust)	Cargo	Program arguments	--no-default-features (Tauri'ye geliştirme sunucusunu kullanmasını söyler)

5.3.3. Hata Ayıklama Oturumunu Başlatma ve Etkili Kullanma

Her iki yapılandırma da hazır olduğunda, tam entegre bir hata ayıklama oturumu başlatmak için aşağıdaki adımlar izlenir:

1. **Adım 1: Ön Yüz Sunucusunu Başlatın**
    - RustRover'da (veya WebStorm'da), oluşturduğunuz "Run Vite Dev Server" yapılandırmasını seçin ve Run (oynat) simgesine tıklayın.
    - Terminalde Vite sunucusunun başarıyla başladığını ve http://localhost:5173 adresinde dinlediğini doğrulayın. Bu terminali oturum boyunca açık bırakın.
2. **Adım 2: Rust Koduna Breakpoint'ler Ekleyin**
    - RustRover'da, src-tauri/src/lib.rs veya diğer .rs dosyalarınızda, yürütmenin durmasını istediğiniz satırlara breakpoint'ler (kırmızı noktalar) ekleyin.
3. **Adım 3: Rust Arka Planını Debug Modunda Başlatın**
    - RustRover'da, "Debug Tauri Core (Rust)" yapılandırmasını seçin.
    - Run simgesinin yanındaki Debug (böcek) simgesine tıklayın.
4. **Adım 4: Hata Ayıklama**
    - RustRover, Rust kodunu derleyecek ve ardından yerel uygulama penceresini hata ayıklayıcı bağlı olarak başlatacaktır.
    - Uygulama penceresi açıldığında, içeriğini çalışan Vite sunucusundan yükleyecektir.
    - Ön yüzde, bir Rust komutunu tetikleyen bir butona tıkladığınızda, Rust kodundaki yürütme ilgili breakpoint'e ulaştığında RustRover'da duracaktır.
    - Bu noktada, RustRover'ın Debug panelinde değişkenleri inceleyebilir, bellek durumunu kontrol edebilir ve kodda adım adım ilerleyebilirsiniz (Step Over, Step Into, vb.).[46]
    - Aynı anda, WebStorm'daki JavaScript hata ayıklayıcısını veya Ctrl+Shift+I ile açılan WebView geliştirici araçlarını kullanarak ön yüzdeki hataları ayıklayabilirsiniz.[38]

Bu iki parçalı yaklaşım, Tauri'nin esnek mimarisinin bir sonucudur ve bu sürece hakim olmak, geliştiricilere hem ön yüz hem de arka plan üzerinde tam kontrol ve derinlemesine hata ayıklama yeteneği kazandırır. Bu, bir Tauri uygulamasını, bileşenleri ayrı ayrı test edilebilen ve debug edilebilen bir istemci-sunucu uygulaması gibi ele almayı sağlar.

Bölüm 6: Sonuç ve İleri Adımlar

Bu kılavuz, Tauri, Vite, React ve Rust teknolojilerini kullanarak modern, güvenli ve üretim odaklı bir masaüstü uygulaması geliştirmek için kapsamlı bir yol haritası sunmuştur. Projenin temel yapılandırmasından başlayarak, güvenli iletişim, kurumsal düzeyde loglama, otomatik güncellemeler ve profesyonel hata ayıklama ortamlarına kadar tüm kritik aşamalar detaylı bir şekilde ele alınmıştır.

6.1. Yapılandırmanın Özeti ve En İyi Pratikler

Geliştirme süreci boyunca benimsenen temel mimari prensipler ve en iyi pratikler şunlardır:

- **Net Sorumluluk Ayrımı:** Tauri'nin ön yüz (Vite+React) ve arka plan (Rust) projelerini ayırması, modüler, bakımı kolay ve ölçeklenebilir bir uygulama yapısını teşvik eder. Bu ayrım, farklı uzmanlıklara sahip geliştiricilerin paralel olarak verimli bir şekilde çalışmasına olanak tanır.
- **Güvenlik Odaklı İletişim:** invoke ve events üzerine kurulu IPC mimarisi, capabilities sistemi ile birleştiğinde, varsayılan olarak güvenli bir iletişim kanalı sağlar. Ön yüzün arka plana erişimi "en az ayrıcalık ilkesi" ile kısıtlanarak saldırı yüzeyi en aza indirilir. Harici API isteklerinin tauri-plugin-http ve scope kısıtlamaları ile Rust üzerinden yapılması, bu güvenlik duruşunu daha da pekiştirir.
- **Birleşik ve Yapılandırılabilir Loglama:** tauri-plugin-log kullanarak hem ön yüz hem de arka plan loglarını tek bir yapılandırılabilir akışta birleştirmek, özellikle üretim ortamında hata teşhisini büyük ölçüde kolaylaştırır. Dosyaya loglama, rotasyon ve seviye filtreleme gibi özellikler, onu kurumsal bir çözüm haline getirir.
- **Zorunlu Kriptografik Güncellemeler:** Otomatik güncelleme mekanizmasının imza doğrulamayı zorunlu kılması, kullanıcıların her zaman güvenli ve doğrulanmış yazılım sürümlerini almasını garanti eder. Bu, Tauri'nin güvenlik konusundaki tavizsiz yaklaşımının bir başka göstergesidir.
- **Bilinçli Hata Ayıklama:** Geliştirme sırasında çalışan iki ayrı süreci (Node.js ve Rust) anlamak ve JetBrains IDE'lerinde bu süreçler için ayrı hata ayıklama yapılandırmaları oluşturmak, en karmaşık sorunların bile kök nedenine inilmesini sağlayan profesyonel bir iş akışı sunar.

6.2. Üretim Derlemesi ve Dağıtım için İpuçları

Geliştirme ve hata ayıklama süreçleri tamamlandığında, uygulama son kullanıcıya dağıtılmaya hazırdır.

- **Üretim Derlemesi:** Uygulamanın dağıtılabilir paketlerini oluşturmak için projenin kök dizininde aşağıdaki komut çalıştırılır:

Bash
pnpm tauri build

Bu komut, aşağıdaki adımları otomatik olarak gerçekleştirir:

1. tauri.conf.json'daki beforeBuildCommand (pnpm build) komutunu çalıştırarak ön yüzü optimize edilmiş statik dosyalara derler.
2. Rust arka planını release modunda (performans optimizasyonları etkinleştirilmiş olarak) derler.
3. Ön yüz varlıklarını derlenmiş Rust ikili dosyasına gömer.
4. Son olarak, hedef platform için yerel yükleyiciler ve paketler oluşturur.

- **Derleme Çıktıları:** Oluşturulan dağıtım paketleri src-tauri/target/release/bundle/ dizininde bulunur. Tauri, platforma bağlı olarak aşağıdaki gibi çeşitli formatlarda çıktılar üretir [3]:
  - **Windows:** .msi (WiX ile) ve .exe (NSIS ile) yükleyicileri.
  - **macOS:** .app uygulaması ve .dmg disk imajı.
  - **Linux:** .deb (Debian/Ubuntu), .rpm (Fedora/CentOS) ve .AppImage (dağıtımdan bağımsız).
- **Sürekli Entegrasyon ve Dağıtım (CI/CD):** Farklı işletim sistemleri için derleme yapmak, her platformda ayrı bir makine gerektirir. Bu süreci otomatikleştirmek için GitHub Actions gibi CI/CD platformları kullanılabilir. Tauri, platformlar arası derlemeyi kolaylaştıran resmi bir GitHub Action sunmaktadır. Bu Action, her kod değişikliğinde veya yeni bir sürüm etiketlendiğinde otomatik olarak Windows, macOS ve Linux için derlemeler yapabilir, bunları imzalayabilir ve bir GitHub Release'i olarak yayınlayabilir. Bu, dağıtım sürecini önemli ölçüde basitleştirir ve hızlandırır.[14]

Bu kılavuzda sunulan yapılandırmalar ve pratikler, sadece bir "Merhaba Dünya" uygulamasının ötesine geçerek, gerçek dünya ihtiyaçlarını karşılayacak, güvenli, sürdürülebilir ve kullanıcı dostu masaüstü uygulamaları oluşturmak için sağlam bir temel oluşturmaktadır.

Alıntılanan çalışmalar

1. How to Create Modern Desktop Apps with React and TypeScript using Tauri - YouTube, erişim tarihi Ekim 1, 2025, https://www.youtube.com/watch?v=XmTdvx4xM6I
2. Building a pomodoro timer with Tauri using React and Vite - LogRocket Blog, erişim tarihi Ekim 1, 2025, https://blog.logrocket.com/build-pomodoro-timer-tauri-using-react-and-vite/
3. tauri-apps/tauri: Build smaller, faster, and more secure desktop and mobile applications with a web frontend. - GitHub, erişim tarihi Ekim 1, 2025, https://github.com/tauri-apps/tauri
4. Create a Project - Tauri, erişim tarihi Ekim 1, 2025, https://v2.tauri.app/start/create-project/
5. Quick Start | Tauri v1, erişim tarihi Ekim 1, 2025, https://tauri.app/v1/guides/getting-started/setup/
6. Vite | Tauri v1, erişim tarihi Ekim 1, 2025,

https://tauri.app/v1/guides/getting-started/setup/vite/
7. Tauri framework: Building lightweight desktop applications with Rust - Medium, erişim tarihi Ekim 1, 2025, https://medium.com/codex/tauri-framework-building-lightweight-desktop-applications-with-rust-3b3923c72e75
8. Project Structure - Tauri, erişim tarihi Ekim 1, 2025, https://v2.tauri.app/start/project-structure/
9. Getting Started - Vite, erişim tarihi Ekim 1, 2025, https://vite.dev/guide/
10. Tauri Architecture | Tauri v1, erişim tarihi Ekim 1, 2025, https://tauri.app/v1/references/architecture/
11. My opinion on the Tauri framework - A Java geek, erişim tarihi Ekim 1, 2025, https://blog.frankel.ch/opinion-tauri/
12. Creating Your First Tauri App with React: A Beginner's Guide - DEV Community, erişim tarihi Ekim 1, 2025, https://dev.to/dubisdev/creating-your-first-tauri-app-with-react-a-beginners-guide-3eb2
13. Where to find log files on Tauri Apps - Aptabase, erişim tarihi Ekim 1, 2025, https://aptabase.com/blog/where-to-find-tauri-logs
14. Create a Simple Desktop App Using Vite, React, and Tauri.js - Agallio Samai, erişim tarihi Ekim 1, 2025, https://www.agallio.xyz/post/simple-desktop-app-tauri
15. Inter-Process Communication - Tauri, erişim tarihi Ekim 1, 2025, https://v2.tauri.app/concept/inter-process-communication/
16. Inter-Process Communication - The Tauri Documentation WIP, erişim tarihi Ekim 1, 2025, https://jonaskruckenberg.github.io/tauri-docs-wip/development/inter-process-communication.html
17. Calling Rust from the Frontend - Tauri, erişim tarihi Ekim 1, 2025, https://v2.tauri.app/develop/calling-rust/
18. Calling the Frontend from Rust - Tauri, erişim tarihi Ekim 1, 2025, https://v2.tauri.app/develop/calling-frontend/
19. Events | Tauri v1, erişim tarihi Ekim 1, 2025, https://tauri.app/v1/guides/features/events/
20. Handling events in Tauri - Tauri Tutorials Home, erişim tarihi Ekim 1, 2025, https://tauritutorials.com/blog/tauri-events-basics
21. tauri-plugin-http - crates.io: Rust Package Registry, erişim tarihi Ekim 1, 2025, https://crates.io/crates/tauri-plugin-http
22. HTTP Client | Tauri, erişim tarihi Ekim 1, 2025, https://v2.tauri.app/plugin/http-client/
23. http | Tauri v1, erişim tarihi Ekim 1, 2025, https://tauri.app/v1/api/js/http/
24. reqwest - Rust - Docs.rs, erişim tarihi Ekim 1, 2025, https://docs.rs/reqwest/
25. What does it take to make an HTTP request - The Rust Programming Language Forum, erişim tarihi Ekim 1, 2025, https://users.rust-lang.org/t/what-does-it-take-to-make-an-http-request/125980
26. tauri-plugin-log - crates.io: Rust Package Registry, erişim tarihi Ekim 1, 2025, https://crates.io/crates/tauri-plugin-log
27. Logging | Tauri, erişim tarihi Ekim 1, 2025, https://v2.tauri.app/plugin/logging/

28. Debugging - The Tauri Documentation WIP, erişim tarihi Ekim 1, 2025, https://jonaskruckenberg.github.io/tauri-docs-wip/development/debugging.html
29. Your approach to logging : r/rust - Reddit, erişim tarihi Ekim 1, 2025, https://www.reddit.com/r/rust/comments/ye0a5j/your_approach_to_logging/
30. Best Practices for Client-Side Logging and Error Handling in React - Loggly, erişim tarihi Ekim 1, 2025, https://www.loggly.com/blog/best-practices-for-client-side-logging-and-error-handling-in-react/
31. Where can I find the log info? · tauri-apps tauri · Discussion #5160 - GitHub, erişim tarihi Ekim 1, 2025, https://github.com/tauri-apps/tauri/discussions/5160
32. Updater - The Tauri Documentation WIP, erişim tarihi Ekim 1, 2025, https://jonaskruckenberg.github.io/tauri-docs-wip/distributing/updater.html
33. Updater | Tauri, erişim tarihi Ekim 1, 2025, https://v2.tauri.app/plugin/updater/
34. Updater | Tauri v1, erişim tarihi Ekim 1, 2025, https://v1.tauri.app/v1/guides/distribution/updater/
35. Tauri auto-updater for private applications - Anystack, erişim tarihi Ekim 1, 2025, https://anystack.sh/docs/integrations/tauri
36. updater | Tauri v1, erişim tarihi Ekim 1, 2025, https://tauri.app/v1/api/js/updater/
37. Tauri v2 with Auto-Updater - CrabNebula Docs, erişim tarihi Ekim 1, 2025, https://docs.crabnebula.dev/cloud/guides/auto-updates-tauri/
38. Debug - Tauri, erişim tarihi Ekim 1, 2025, https://v2.tauri.app/develop/debug/
39. Debugging in RustRover | Tauri v1, erişim tarihi Ekim 1, 2025, https://v1.tauri.app/v1/guides/debugging/rustrover/
40. How to debug tauri rust code with Rustrover? #8253 - GitHub, erişim tarihi Ekim 1, 2025, https://github.com/tauri-apps/tauri/discussions/8253
41. React Native | WebStorm Documentation - JetBrains, erişim tarihi Ekim 1, 2025, https://www.jetbrains.com/help/webstorm/react-native.html
42. Debugging React Apps Created With Create React App in WebStorm - The JetBrains Blog, erişim tarihi Ekim 1, 2025, https://blog.jetbrains.com/webstorm/2017/01/debugging-react-apps/
43. What is the best way to debug a React application in WebStorm? : r/Jetbrains - Reddit, erişim tarihi Ekim 1, 2025, https://www.reddit.com/r/Jetbrains/comments/1d81976/what_is_the_best_way_to_debug_a_react_application/
44. Cargo run/debug configuration | RustRover Documentation - JetBrains, erişim tarihi Ekim 1, 2025, https://www.jetbrains.com/help/rust/cargo-run-debug-configuration.html
45. Debugging in VS Code - Tauri v1, erişim tarihi Ekim 1, 2025, https://v1.tauri.app/v1/guides/debugging/vs-code/
46. Debug code | RustRover Documentation - JetBrains, erişim tarihi Ekim 1, 2025, https://www.jetbrains.com/help/rust/debugging-code.html
47. Step through the program | RustRover Documentation - JetBrains, erişim tarihi Ekim 1, 2025, https://www.jetbrains.com/help/rust/stepping-through-the-program.html
48. Application Debugging | Tauri v1, erişim tarihi Ekim 1, 2025,

https://v1.tauri.app/v1/guides/debugging/application/