Blockchain Tabanlı Noter Uygulaması Teknik Rapor

Hazırlayan: Mehmet Nasim Yılmaz
Teslim Tarihi: 22 Mayıs 2025
GitHub Proje Adresi: [BlockChain_NoteryProject]
Github.com/menasy
linkedin.com/in/menasy

---

1. Amaç

Bu projenin amacı, blockchain teknolojisinin temel avantajlarından biri olan veri bütünlüğü, değişmezlik, zaman damgalama ve doğrulama mekanizmalarını kullanarak gerçek hayatta uygulanabilir bir noter sistemi geliştirmektir. Belgeler, bayt olarak okunup, elde edilen veriler üzerinden SHA256 algoritması ile hash değeri oluşturularak Ethereum tabanlı test ağına kaydedilmektedir. Böylece, belgelerin değişip değişmediği doğrulanabilmekte; aynı zamanda kaydedilen belgelerin eklenme tarihleri ve zaman damgaları da kontrol edilebilmektedir.

---

2. Yaklaşım

Projede üç temel bileşen yer almaktadır:

- **Hash Oluşturma:**
  Kullanıcının girdiği belge, bayt değerleri üzerinden SHA256 algoritması kullanılarak benzersiz hash değeri üretilir.
- **Blockchain'e Kaydetme:**
  Hesaplanan hash değeri, akıllı sözleşme aracılığıyla Ethereum blockchain ağına kaydedilir.
- **Doğrulama:**
  Kullanıcı, daha sonra aynı belgeyi tekrar yükleyerek, belgenin önceden kaydedilip kaydedilmediğini; aynı zamanda ne zaman kaydedildiğini görebilir.

Backend tarafında ilk olarak akıllı kontratları hangi dille yazabileceğimi araştırdım. Daha önce Rust deneyimim olsa da, Ethereum tabanlı bir yapı kurmak için Solidity dilini tercih ettim. Solidity, Ethereum için özel olarak geliştirilmiş olması, Remix IDE üzerinden kolayca yazılıp deploy edilebilmesi, dokümantasyonunun ve kaynakların fazlalığı gibi etkenlerle seçildi. Frontend tarafında ise, daha önce edindiğim JavaScript ve HTML bilgim doğrultusunda, gerekli arayüzü oluşturarak backend ile entegre ettim.

---

3. Kullanılan Teknolojiler ve Araçlar

- **Solidity**
    - Ethereum üzerinde çalışan akıllı kontratları yazmak için kullanılan programlama dilidir. Ethereum Sanal Makinesi (EVM) üzerinde çalıştığı için Ethereum ile tam uyumludur.
    - Daha önce C ve C++ gibi dillerde çalıştığım için Solidity'nin yapısını anlamak ve uygulamak benim için kolay oldu.
    - Projede akıllı kontratlar Solidity ile yazıldı, Remix IDE üzerinden test edilip deploy edildi.
- **Ethereum Sepolia Test Ağı**
    - Gerçek Ethereum ağı yerine, testler için Sepolia test ağı kullanıldı. Sepolia, Ethereum'un resmi test ağlarından biridir ve geliştiricilerin gerçek para harcamadan akıllı kontratlarını test etmelerini sağlar.
    - Sepolia üzerinde çalışan node'lar işlemleri onaylayarak blok zincirine ekler. Test blokları belirli aralıklarla sıfırlanır, bu da ağı testler için uygun hale getirir.
    - Sepolia faucet'lerinden ücretsiz test Ether (Sepolia ETH) alınarak, akıllı kontratlar deploy edildi ve fonksiyonları test edildi.
    - Ağa ait işlemler, Proof-of-Authority (PoA) veya Proof-of-Stake (PoS) gibi yöntemlerle gerçekleştirilmektedir.
- **Ethers.js**
    - JavaScript ortamında Ethereum blockchain'i ile etkileşim kurmak için kullanılan bir kütüphanedir.
    - Akıllı kontratlar ile iletişim sağlamak, kontrat fonksiyonlarını çağırmak, blockchain'den veri almak ve işlemleri göndermek için kullanıldı.
    - Backend ve frontend arasındaki bağlantının kurulmasında önemli bir rol oynadı.
- **MetaMask**
    - Tarayıcı üzerinde çalışan bir Ethereum cüzdanıdır.
    - Kullanıcıların blockchain üzerinde işlem yapmasını, işlemleri onaylamasını ve cüzdanlarını yönetmesini sağladı.
    - Uygulama MetaMask ile entegre edildi.
- **CryptoJS**
    - JavaScript ortamında kullanılan bir kriptografi kütüphanesidir.
    - SHA256 hash fonksiyonlarını kullanarak verilerin güvenliğini sağlamak ve veri bütünlüğünü kontrol etmek için uygulandı.
    - Bu yöntem, verinin değiştirilmediğini doğrulamak amacıyla kullanıldı.
- **HTML + JavaScript**
    - Kullanıcı arayüzü ve işlem kontrolü için kullanıldı.
    - HTML ile arayüz oluşturulurken, JavaScript ile dinamik işlemler ve veri akışı sağlandı.
    - Temel seviyede çalışan bir arayüz geliştirildi.

4. Akıllı Sözleşme Kodu, Açıklamaları

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract DocumentNotary
{
    struct Document
    {
        bytes32 hash;
        uint256 timestamp;
    }
    mapping(bytes32 => Document) private documentsMap;
    mapping(bytes32 => bool) private documentExists;
   function uploadDocument(bytes32 docHash) external
   {
        require(!documentExists[docHash], "Document is already exists !");
        documentsMap[docHash] = Document(docHash, block.timestamp);
        documentExists[docHash] = true;
    }

    function verifyDocument(bytes32 docHash) external view returns (uint256
timestamp)
    {
        require(documentExists[docHash], "Document not found !");
        return documentsMap[docHash].timestamp;
    }
}
```

uploadDocument Fonksiyonu

```
function uploadDocument(bytes32 docHash) external {
    require(!documentExists[docHash], "Document is already exists !");
        documentsMap[docHash] = Document(docHash, block.timestamp);
        documentExists[docHash] = true; }
```

- **Fonksiyon Tanımı ve Parametre Kullanımı:**
  `uploadDocument` fonksiyonu dışarıdan erişilebilir (external) olarak tanımlandı. Bu fonksiyon, SHA256 ile oluşturulmuş `bytes32` türünde bir belge hash'i alıyor. `bytes32` kullanmamın nedeni, hem sabit uzunlukta olması hem de Ethereum'da düşük gas tüketimi sağlamasıdır. Akıllı sözleşmelerde performans yalnızca çalışma hızı değil, aynı zamanda maliyet anlamında da önem taşır. Bu nedenle, fonksiyonun hem mantıksal kontrol yapısı net tutulmuş hem de veri yapısı düşük maliyetli olacak şekilde seçilmiştir.
- **Kontrol Mekanizması:**
  Fonksiyon içinde ilk olarak `require(!documentExists[docHash], "Document is already exists !");` ifadesi kullanılarak, aynı belgenin

birden fazla kez eklenmesi engelleniyor. Eğer gelecek hash daha önce eklenmişse, işlem geri çevriliyor ve kullanıcıya hata mesajı veriliyor.

- **Veri Kaydı:**
  Kayıt yoksa, gelen hash ile yeni bir `Document` oluşturulup `documentsMap` içerisine kaydediliyor. Ardından, `documentExists[docHash] = true;` ifadesi ile sistemde bu hash'in var olduğu belirtiliyor.

verifyDocument Fonksiyonu

```
function verifyDocument(bytes32 docHash) external view returns (uint256 timestamp)
{
        require(documentExists[docHash], "Document not found !");
        return documentsMap[docHash].timestamp;}
```

- **Fonksiyon Tanımı ve Kullanımı:**
  - `verifyDocument` fonksiyonu da dışarıdan erişilebilir (external) olarak tanımlandı ve yine SHA256 ile oluşturulmuş `bytes32` türünde bir belge hash'i alıyor.
- **Kontrol ve Geri Dönüş:**
  - İlk olarak, bu hash'in sistemde kayıtlı olup olmadığı kontrol ediliyor. Eğer belge bulunamazsa, `require(documentExists[docHash], "Document not found !");` ifadesi ile işlem iptal ediliyor.
  - Belge mevcutsa, ilgili hash'e ait zaman bilgisi `documentsMap[docHash].timestamp` üzerinden geri döndürülüyor. Frontend tarafında bu fonksiyon çağrılarak kullanıcılara "bu belge şu tarihte kaydedilmiş" gibi net bilgiler sunuluyor.
  - Tüm exception durumları JavaScript tarafında yakalanarak, kullanıcıya anlamlı geri dönüşler sağlanıyor. Ayrıca, fonksiyon `view` olarak tanımlandığından, blockchain üzerinde değişiklik yapmadığı için gas tüketimi gerçekleşmiyor.

5. Web Arayüzü Özellikleri

Crypto İşlemleri ve Ortak Fonksiyonlar

- **Kütüphane Entegrasyonu:**
  - Crypto işlemlerini ele almak için HTML sayfasına **crypto-js.js** ve **ethers.umd.min.js** kütüphaneleri import edilmiştir.
  - JavaScript kodlaması, tüm işlevleri barındıran **functions.js** dosyasında ele alınmıştır.
- **Ortak Fonksiyonlar:**
  - İki ana işlevde (belge kaydetme ve belge doğrulama) kullanılacak ortak işlemleri, kod tekrarı olmaması amacıyla ayrı metodlar olarak yazdım:

getFileHash Fonksiyonu

```
async function getFileHash(fileInputId)
{
	const fileInput = document.getElementById(fileInputId);
	const file = fileInput.files[0];

	if (!file) {
		alert("Lütfen önce bir dosya seçin!");
		throw new Error("Dosya seçilmedi!");
	}else if (file.size == 0){
		alert("Dosya boş olamaz!");
		throw new Error("Dosya boş!");
	}

	return new Promise((resolve, reject) =>
	{
		const reader = new FileReader();
		reader.onload = (e) =>
		{
			const wordArray = CryptoJS.lib.WordArray.create(new
Uint8Array(e.target.result)); // DÖNÜŞÜM ÖNEMLİ!
			const hashHex =
CryptoJS.SHA256(wordArray).toString(CryptoJS.enc.Hex);
			resolve("0x" + hashHex);
		};
		reader.onerror = (error) => reject(error);
		reader.readAsArrayBuffer(file);
	});
}
```

- **İşlevi:** HTML'den alınan dosyayı kontrol edip okumak, dosya varsa içeriğini SHA256 algoritması ile hashleyerek döndürmek.
- **Detaylar:**

- Dosya seçilmediğinde kullanıcı uyarılır ve işlem hata fırlatarak durdurulur.
- Seçilen dosya, `FileReader` yardımıyla ArrayBuffer biçiminde okunur.
- Elde edilen veri, CryptoJS kullanılarak SHA256 ile hashlenir ve hash değeri “0x” öneki eklenerek ethereum adres formatına uygun olarak döndürülür.

createContract Fonksiyonu

```
async function createContract(needsSigner = false) {
       if (!window.ethereum) throw new Error("MetaMask yüklü değil!");
       const provider = new ethers.providers.Web3Provider(window.ethereum);
       const network = await provider.getNetwork();
       if (network.chainId !== 11155111) {
              await window.ethereum.request({method:
"wallet_switchEthereumChain", params: [{ chainId: "0xAA36A7" }]});
       }
       if (needsSigner) {
              await provider.send("eth_requestAccounts", []);
              const signer = provider.getSigner();
              return new ethers.Contract(contractAdress,contractABI,signer);
       }
       return new ethers.Contract(contractAdress,contractABI,provider);}
```

- **İşlevi:** Hem upload hem de doğrulama işlemlerinde kullanılacak şekilde akıllı sözleşme (contract) nesnesi oluşturmak.
- **Detaylar:**
  - Tarayıcıda MetaMask’ın kurulu ve aktif olup olmadığı kontrol edilir; yoksa hata fırlatılır.
  - `ethers.providers.Web3Provider` kullanılarak bir provider oluşturulur ve Ethereum ağına erişim sağlanır.
  - Kullanıcının bağlı olduğu ağ, Sepolia test ağı (chainId: 11155111) olup olmadığı kontrol edilir; eğer test ağında değilse, otomatik ağ değiştirme isteği gönderilir.
  - Upload işleminde imzalı (signer) işlem gerektiğinde, `eth_requestAccounts` ile kullanıcı hesabı alınır ve `getSigner()` ile signer elde edilir.
  - Contract oluşturma sırasında, Remix IDE üzerinden deploy edilen contract’ın blockchain’de yer alan adresi (contract adresi) ve derlenmiş contract ABI bilgileri kullanılır.
  - Gerekli bilgiler (contract adres, contract ABI, signer veya provider) sağlanarak, oluşturulan contract nesnesi ana metodlarda uploadHash ve verifyHash işlemlerinde kullanılacaktır.

Upload İşlemi (uploadHash) ve İş Akışı

```
async function uploadHash() {
	try
	{
		const hash = await getFileHash("uploadInput");
		const contract = await createContract(true);
		const tx = await contract.uploadDocument(hash);
		await tx.wait();
		document.getElementById("result").innerHTML = `Belge başarıyla
kaydedildi`;
	}
	catch (error)
	{
		let errorMessage = error.message;
		if (error.code === 'ACTION_REJECTED') {
			errorMessage = "İşlem iptal edildi !";
		}
		else if (error.code === 'UNPREDICTABLE_GAS_LIMIT')
		{
			errorMessage = "Belge zaten kayıtlı !";
		}
		document.getElementById("result").innerHTML = `Hata: ${errorMessage}`;}}
```

- **Hash Hesaplama:**
  - Kullanıcı tarafından seçilen dosyadan, getFileHash fonksiyonu yardımıyla hash değeri elde edilir.
- **Contract Nesnesi Oluşturma:**
  - İmzalı işlemler gerektiren upload işlemi için createContract(true) çağrılarak imzalı contract nesnesi oluşturulur.
- **Blockchain İşlemi:**
  - Elde edilen hash değeri, akıllı sözleşmede tanımlı olan `uploadDocument` metoduna parametre olarak gönderilir: `javascript const tx = await contract.uploadDocument(hash);`
  - Bu satır, işlemin blockchain ağına gönderilmesini sağlar (transaction nesnesi oluşturulur).
  - Ancak, işlem henüz onaylanmamış olabilir; bu nedenle, `await tx.wait();` ifadesi kullanılarak işlemin bloklarda onaylanması (mining/confirmation) beklenir.
- **Sonuç ve Hata Yönetimi:**
  - İşlem başarılı olursa, ekrana “Belge başarıyla kaydedildi” mesajı yazdırılır.
  - Eğer işlem sırasında hata meydana gelirse, örneğin işlem iptal edilmiş veya belge zaten kayıtlı ise, ilgili hata mesajları yakalanarak kullanıcıya uygun yanıtlar verilir.

Doğrulama İşlemi (verifyHash) ve İş Akışı

```
async function verifyHash() {
       try
       {const hash = await getFileHash("verifyInput");
              const contract = await createContract();
              const timestamp = await contract.verifyDocument(hash);
              const date = new Date(timestamp * 1000).toLocaleString();
              document.getElementById("result").innerHTML = ` Belge doğrulandı
!<br>Zaman Damgası: ${date}`;
       } catch (error)
       {if (error.message.includes("Document not found !")) {
                     document.getElementById("result").innerHTML = "Belge
blockchain'de mevcut değil !";}else {
                     document.getElementById("result").innerHTML = `Hata:
${error.message}`;}}}
```

- **Dosya Hash Hesaplama:**
  - Kullanıcı tarafından yüklenen dosyadan getFileHash fonksiyonu ile hash değeri hesaplanır (bu sefer giriş alanı "verifyInput" olarak kullanılır).
- **Contract Nesnesi Oluşturma:**
  - Salt okunur işlemler yapılacağından createContract() çağrısı ile imzasız contract nesnesi oluşturulur.
- **Doğrulama Çağrısı:**
  - Oluşturulan contract üzerinden `verifyDocument` metodu çağrılarak hash değeri gönderilir: `javascript const timestamp = await contract.verifyDocument(hash);`
  - Eğer hash, blockchain'de mevcutsa, ilgili zaman damgası (timestamp) elde edilir.
  - Eğer hash bulunamazsa, exception fırlatılır ve catch bloğunda hata yakalanarak uygun mesaj ekrana yazılır.
- **Zaman Formatlama:**
  - Dönen timestamp değeri, smart contract'tan gelen Unix zamanı (Epoch time) formatındadır (1 Ocak 1970'ten itibaren geçen saniye sayısı).
  - JavaScript'te tarihin milisaniye cinsinden alınması nedeniyle, timestamp değeri 1000 ile çarpılarak Date nesnesi aracılığıyla okunabilir tarih formatına dönüştürülür ve kullanıcıya sunulur.

Contract Tarafı Bilgileri

- **Contract Adres:**
  - Remix IDE üzerinden projenin derlendikten sonra, Sepolia test ağı üzerinde deploy edilen contract'ın blockchain'de bulunan adresini kullanıyorum.

- **Contract ABI:**
    - Contract'ın fonksiyonlarını ve yapısını tanımlayan gerekli ABI bilgileri girilerek, sözleşmenin işlevlerine erişim sağlanıyor.
- **Signer:**
    - Upload gibi imzalı işlemler için, MetaMask üzerinden elde edilen kullanıcı hesabı (signer) kullanılıyor.
- **Provider:**
    - Salt okunur işlemler (örneğin doğrulama) için Ethereum sağlayıcısı (provider) oluşturularak, blockchain'den veri çekme işlemi gerçekleştiriliyor.
- **Kullanım:**
    - Tüm bu bilgiler kontrat oluşturma aşamasında kullanılarak contract nesnesi üretiliyor; ardından uploadHash ve verifyHash metodlarında bu nesne üzerinden işlemler gerçekleştiriliyor.

Bu yapı sayesinde, hem dosyanın hash hesaplaması ve doğrulama işlemi hem de contract işlemleri (upload ve verify) asgari hata ile güvenli, optimize ve tekrarsız şekilde gerçekleştirilebiliyor. Özellikle, test ağı (Sepolia) kontrolü, kullanıcı hesabı (signer) ve provider ayrımının yapılması, işlem maliyetlerinin azaltılması ve hataların en aza indirilmesi açısından önemli aksiyonlar alınmıştır.

6. Karşılaşılan Sorunlar ve Çözümler

- **Tekrarlı Veri Kaydı:** Proje sürecinde, aynı belgenin birden fazla kez blockchain'e eklenme riskiyle karşılaşıldı. Bu durum, gereksiz transaction'lara ve artan gas maliyetlerine yol açabilirdi. Bunun çözümü olarak, akıllı sözleşmede her hash değeri için ayrı bir kontrol mekanizması (örneğin, `documentExists` mapping'i) oluşturuldu. Bu kontrol sayesinde, aynı belge tekrar eklenmeye çalışıldığında işlem iptal edilip hata mesajı veriliyor.
- **Veri Tipi Seçimi ve Gas Maliyeti:** Ethereum'da işlem maliyetlerini minimize etmek amacıyla, sabit uzunluklu olan `bytes32` veri tipi tercih edildi. Bu seçim, hem veri boyutunun sabit tutulması hem de gas tüketiminin düşürülmesi açısından önemli bir avantaj sağladı.
- **Hata Yönetimi (Exception Handling):** Akıllı sözleşmedeki işlemlerde, hash kontrolü yapılmadan erişim sağlanması durumunda revert (exception) meydana geliyordu. Bu durumu önlemek için, her fonksiyonun başında `require` ifadeleri ile dolaylı hata kontrolü yapılarak, front-end tarafında anlaşılır hata mesajları yakalandı ve kullanıcıya iletildi.
- **Frontend Entegrasyonu ve Test Ağı Kontrolü:** Kullanıcıların, MetaMask üzerinden doğru ağa (Sepolia) bağlı olup olmadığının kontrolü önemli bir zorluktu. Bu amaçla, `createContract` metodunda ağ kontrolü eklenerek, yanlış ağa bağlı kullanıcılara otomatik ağ değiştirme isteği gönderildi. Böylece, backend işlemlerinde hata almadan sürecin devam etmesi sağlandı.

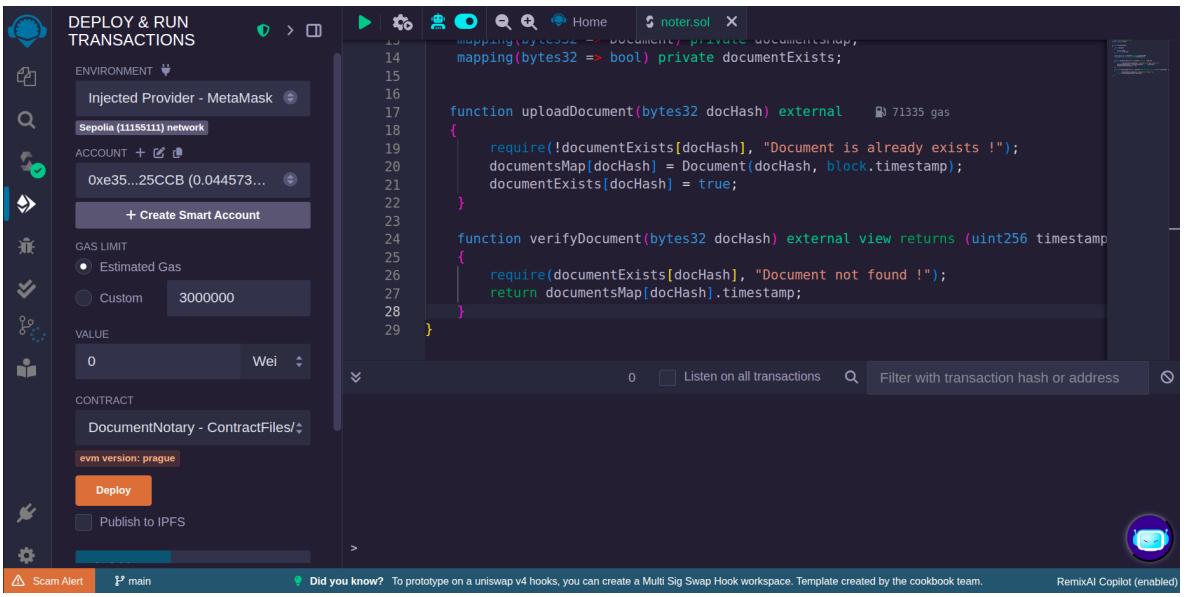
7. Test Süreci

- Uygulamanın doğrulama sürecinde, dosyalardan elde edilen hash değerleri, akıllı sözleşmedeki kayıtlarla karşılaştırılarak belgenin değişip değişmediği test edilmiştir.
- Test sırasında, her işlem için gerçek zamanlı blockchain simülasyonu yapılarak (Sepolia üzerinde) transaction onayları kontrol edilmiş ve hata senaryoları (ör. aynı belgenin tekrar eklenmesi) simüle edilmiştir.

8. Kullanım Senaryosu

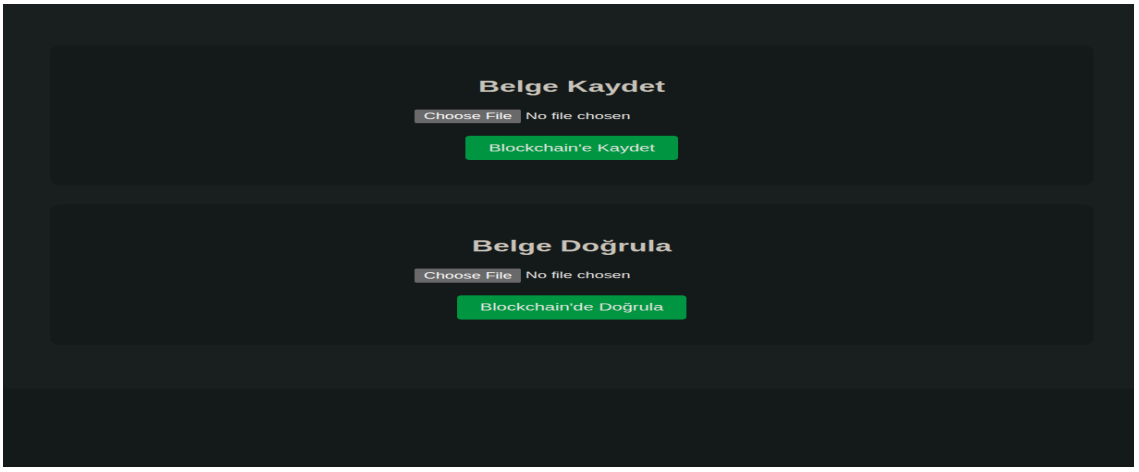
1. Projenin Hazır Hale Getirilmesi

- **Akıllı Sözleşmenin Derlenmesi ve Deploy Edilmesi:**



  - **notary.sol** dosyası, Remix IDE kullanılarak derlenir.
  - **Deploy & Run** kısmında, **injected provider** seçeneği ile MetaMask entegrasyonu yapılır ve **Sepolia test ağı** üzerinden deploy edilir.
- **Contract Bilgilerinin JavaScript'e Tanımlanması:**
  - Deploy işlemi tamamlandıktan sonra **contract adresi** alınarak, **JavaScript dosyasında** `contractAdress` **değişkenine** atanır.
  - **Akıllı sözleşmenin ABI bilgisi** de alınarak, **JavaScript içindeki** `contractABI` **değişkenine** tanımlanır.

Web Arayüzünün Başlatılması:



- **index.html** dosyası açılarak kullanıcı arayüzü başlatılır.
- Kullanıcı, arayüz üzerinden belge yükleme ve doğrulama işlemlerini gerçekleştirmeye başlayabilir.

2. Uygulamanın Kullanımı

- **Belge Yükleme:** Kullanıcı, web arayüzündeki ilgili dosya yükleme alanı üzerinden bir dosya (belge) seçer.
- **Hash Hesaplama:** Seçilen dosya, `getFileHash` fonksiyonu aracılığıyla kontrol edilir, okunur ve CryptoJS kütüphanesinin SHA256 algoritması kullanılarak hash değeri oluşturulur.
- **Contract Oluşturma ve İşlem:** Oluşturulan hash değeri, `createContract` metoduyla elde edilen contract nesnesine gönderilerek akıllı sözleşmedeki `uploadDocument` fonksiyonu ile blockchain'e kaydedilir.
- **İşlem Onayı:** Gönderilen işlem, `await tx.wait();` ifadesiyle blok onayları (mining/confirmation) beklenir.
- **Belge Doğrulama:** Kullanıcı daha sonra, doğrulama alanındaki dosya yükleme işlemi sayesinde aynı dosyanın hash'ini yeniden hesaplayarak, blockchain'de kayıtlı hash ile karşılaştırır. Eğer hash eşleşiyorsa, dosyanın değiştirilmediği ve orijinal olduğu tespit edilir.

9. Sonuç

Bu proje, blockchain teknolojisiyle veri bütünlüğünü sağlamak ve belgeleri güvenli şekilde doğrulamak için geliştirilmiştir. Akıllı sözleşmelerle kayıt altına alınan hash değerleri sayesinde belgelerin değişmediği kanıtlanabilir. Kullanılan araçlar ve testler, sistemin güvenli ve verimli çalışmasını desteklemiştir.

10. Kaynaklar

1. **Sepolia Etherscan**
   https://sepolia.etherscan.io/
   Açıklama: Sepolia test ağı üzerinde gerçekleştirilen işlemlerin, blok, transfer ve akıllı sözleşme kayıtlarının incelenebildiği bir block explorer hizmetidir.
2. **Blockchain Uygulaması Video**
   https://www.youtube.com/watch?v=u0vhvBHd4Ro
   Açıklama: Bu video, blockchain tabanlı uygulama geliştirme sürecinden ve entegrasyon işlemlerinden örnekler sunarak, projenin temel kavramlarına görsel destek sağlamaktadır.
3. **Merkle Ağacı – Investopedia**
   https://www.investopedia.com/terms/m/merkle-tree.asp
   Açıklama: Investopedia'da yer alan bu makale, Merkle ağacının temel prensiplerini, veri bütünlüğünün sağlanmasında nasıl kullanıldığını ve yapısının önemini ayrıntılı şekilde açıklar.
4. **Merkle Ağacı – Medium Makalesi**
   https://medium.com/@rajeevprasanna/understanding-merkle-trees-the-backbone-of-data-verification-13b39af26fff
   Açıklama: Bu makale, Merkle ağaçlarının veri doğrulama sürecindeki rolünü ve yapısal bileşenlerini anlatır; veri güvenliği ve doğrulama açısından Merkle ağacının önemine değinir.
5. **Mutabakat Mekanizması (Consensus Mechanism)**
   https://www.investopedia.com/terms/c/consensus-mechanism-cryptocurrency.asp
   Açıklama: Investopedia'nın bu sayfası, kripto para dünyasında kullanılan mutabakat (consensus) mekanizmalarını, özelliklerini ve işleyişlerini detaylandırmaktadır.
6. **Hashing Algoritmaları – Sadi Evren Şeker**
   https://www.youtube.com/watch?v=2AmKrvTdH-g
   Açıklama: Bu YouTube videosunda, Sadi Evren Şeker tarafından hash algoritmalarının temel prensipleri ve uygulama örnekleri açıklanmakta; hash hesaplamaları ve veri bütünlüğü konularına değinilmektedir.
7. **Solidity Temel Bilgiler – Medium Yazısı**
   https://enginunal.medium.com/solidity-1-temel-bilgiler-40ab2ac36878
   Açıklama: Bu Medium yazısı, Solidity programlama dilinin temel kavramlarını, sözdizimini ve kullanım örneklerini ele alarak, yeni başlayanlar için rehber niteliğinde bilgiler sunmaktadır.
8. **Solidity Basic Syntax – GeeksforGeeks**
   https://www.geeksforgeeks.org/solidity-basic-syntax/
   Açıklama: GeeksforGeeks üzerinde yer alan bu kaynak, Solidity dilinin temel sözdizimini ve temel kullanım örneklerini adım adım açıklayarak, pratik uygulama örnekleri sunmaktadır.