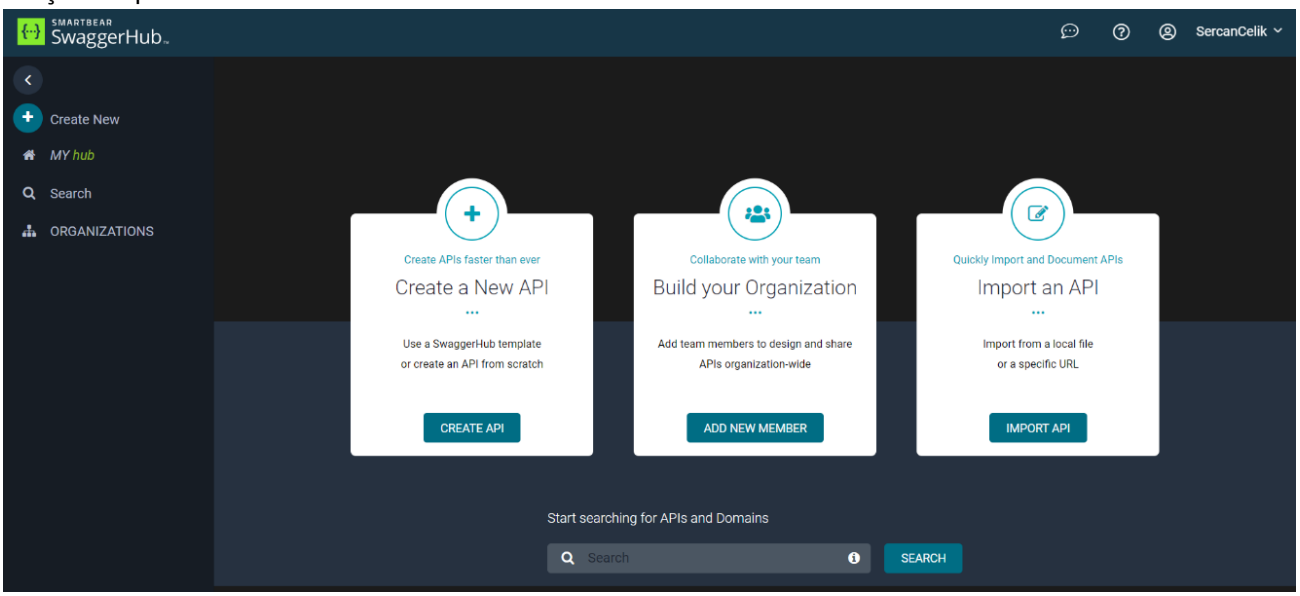
SWAGGER NEDİR?

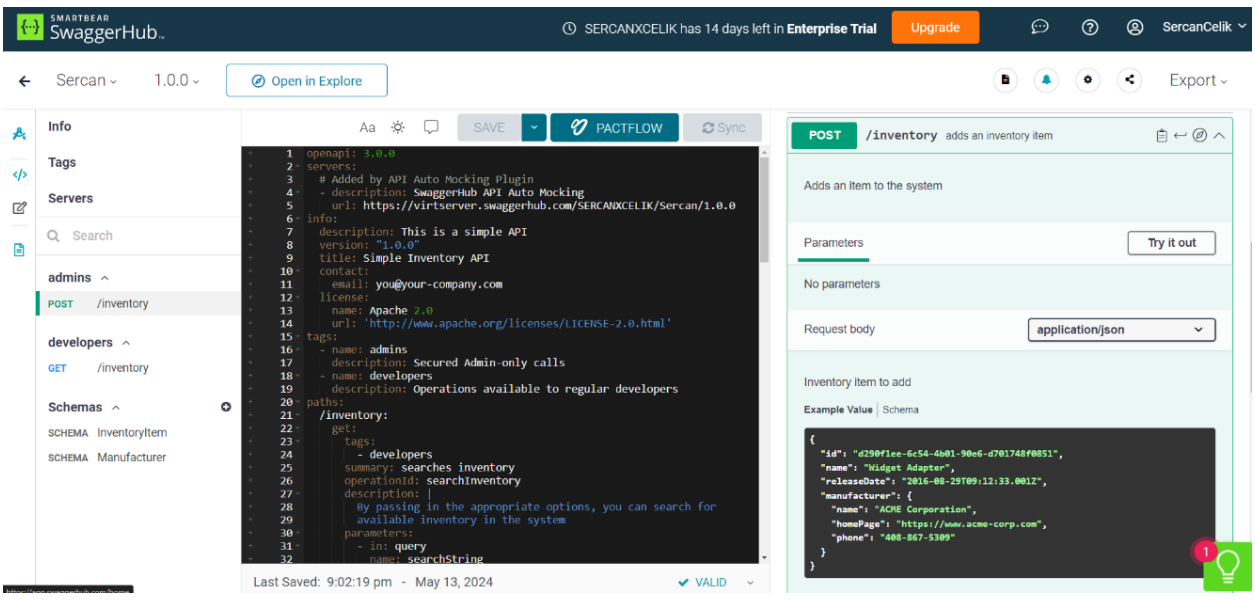
Swagger, yazılmış olan API'nin açık ve erişilebilir hale getirmek için kullanılan bir araçtır, RESTful API'lerin yapısını, veri modellerini, istek ve yanıtlarını ve yetkilendirme gereksinimlerini belgeleler ve bu belgelendirme Swagger sayesinde otomatikte yapılabilir. Swagger, açıklayıcı bir şekilde API'nin belgelenmesine olanak tanır ve API'lerin başka geliştiricilere yol göstermesi ve bu API'leri test etmeyi sağlar.

SWAGGER NASIL KULLANILIR?

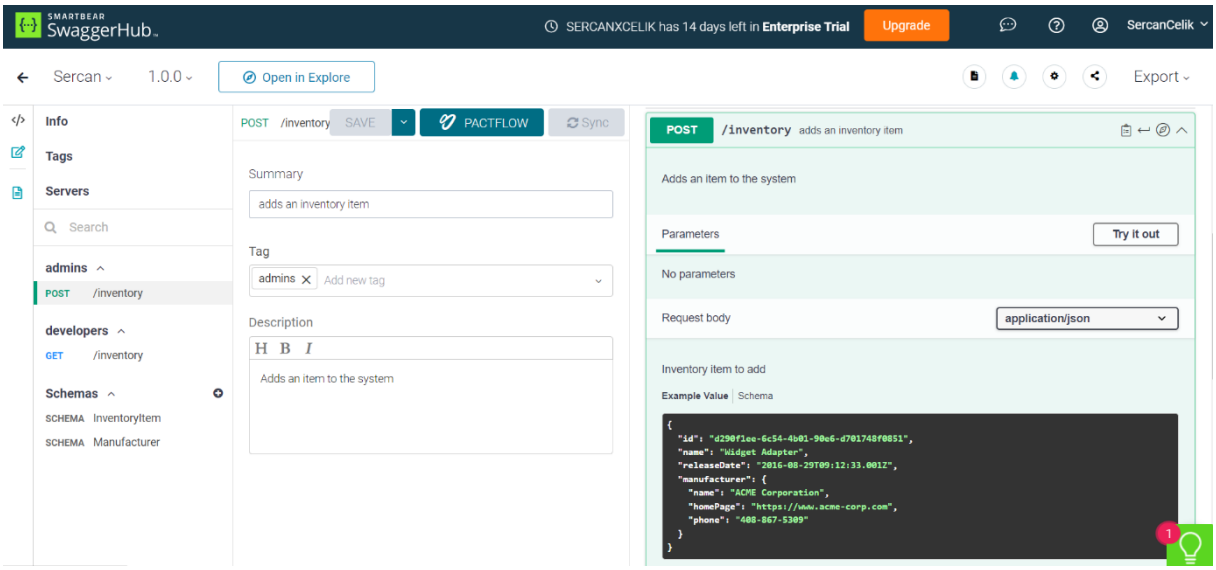
Swagger' kullanıcı girişi yapıldıktan sonra aşağıdaki ekran bizleri karşılar, burada yeni bir API oluşturulup kullanılabilir.



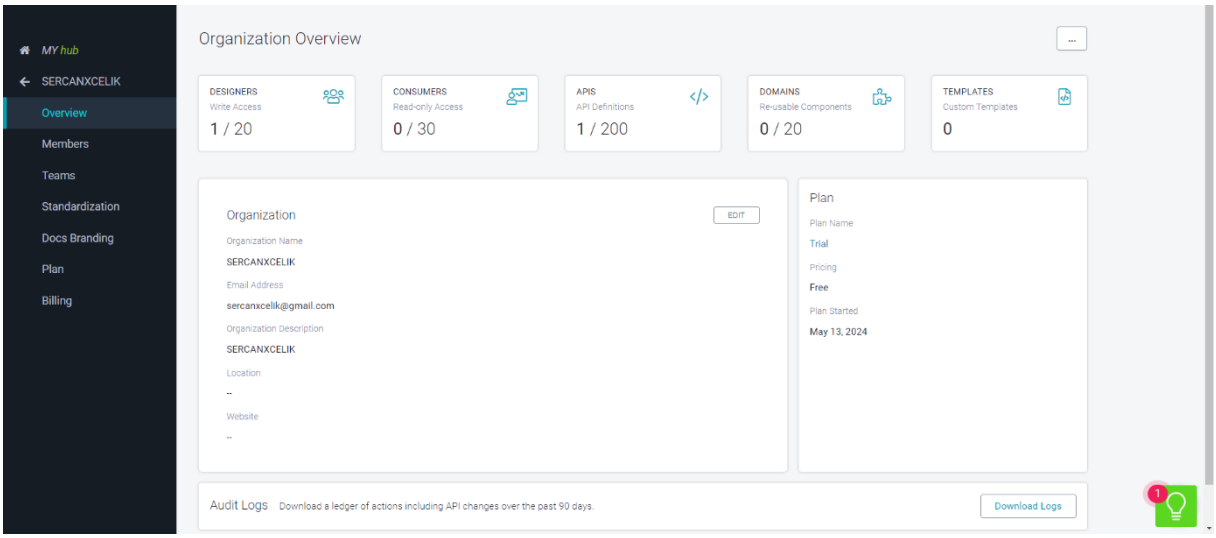
API oluşturuldaktan sonra alttaki ekranda ise, hazır olarak bir API örneği bizi karşılar. Burada API üzerinde hangi olayların gerçekleşmesi isteniyorsa ona göre API'ni özelliklerine göre Yaml kodları yazılır.



Bu ekranda ise (editör ekranı) API'nin dokümantsayonu, API'yi ziyaret edecek kişilere göre tasarlanır (örneğin admin, developer gibi). Buna göre gelecek kullanıcının hangi istekleri(GET, POST gibi.)yapabileceği açık açık yazılır.



Kullanıcı Overview ekranında ise şirket veya kuruluşta kaç çalışanın aktif olarak iş aldığı görülebilir.

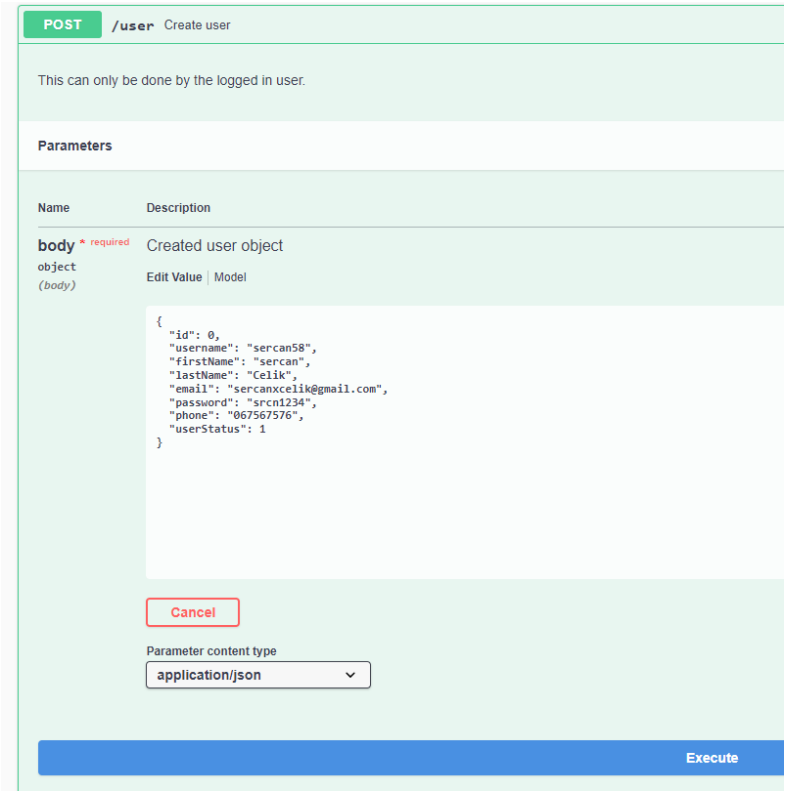


SWAGGER NASIL TEST EDİLİR

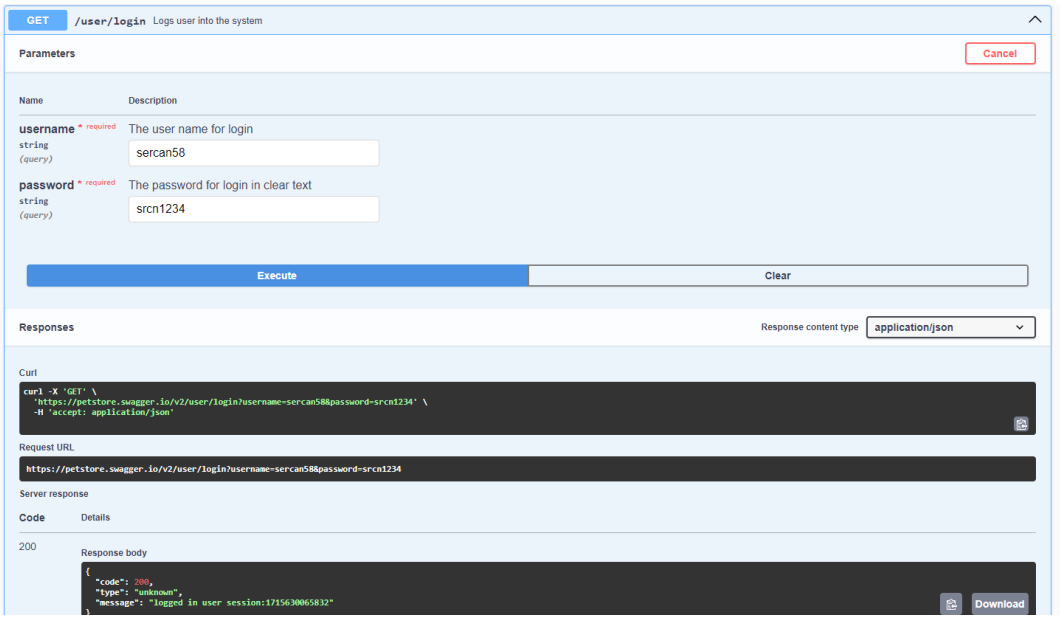
Swagger UI Kullanarak Test:

İlk olarak kullanıcı oluşturma örneği ile başlanacak olursa username, first name gibi doldurulması gereken yerler doldurulur execute edilir ve kullanıcı kaydı oluşup oluşmadığı kontrol edilmesi gereklidir.

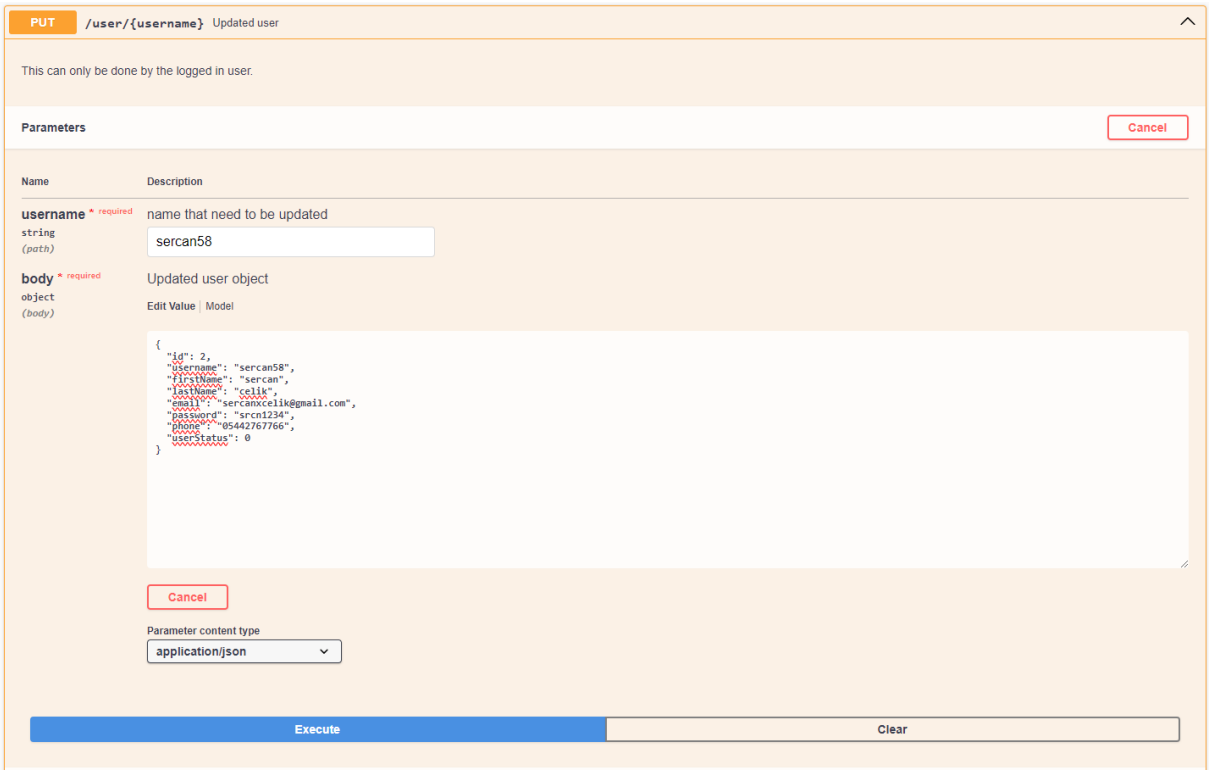
Post /user -> Create User



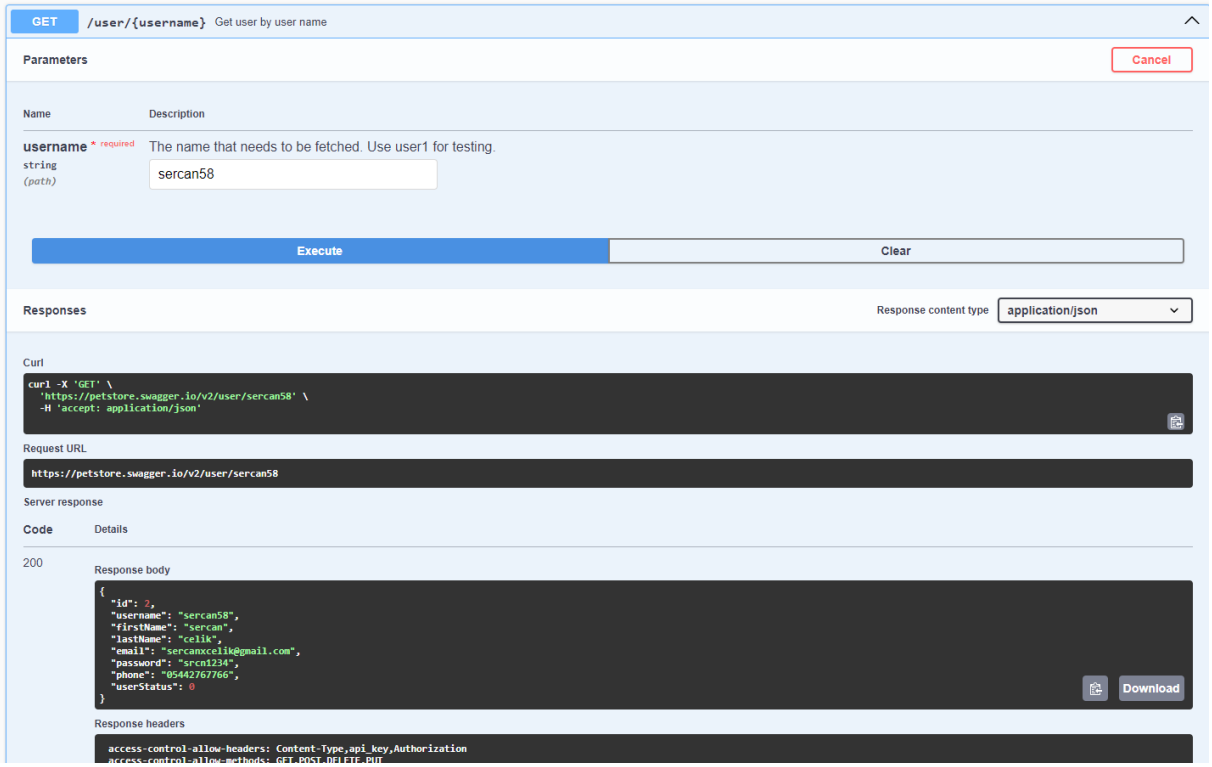
Get /user/login path' inde ise username ve password girilip execute edilip kullanıcının login olması beklenir ve 200 Ok kodu geri dönerek logged in user komutu görülür.



Put /user/{username} path' inde ise username girilip logged in olan kullanıcının username, password, phone vb. kayıtların değiştirilmesine olanak sağlar.



Daha sonra get isteğiyle kullanıcın bilgileri getirildiğinde yaptığımız put isteğinden dolayı kullanıcı bilgilerinin değiştiği görülmektedir.



SWAGGER VE JWT

JWT Nedir?

JWT, yani JSON Web Token, bir kimlik doğrulama ve bilgi paylaşımı yöntemidir. JWT'ler, verileri bir JSON nesnesinde taşıyan bir veri formatıdır ve dijital imza veya şifreleme kullanılarak güvenli bir şekilde iletilirler. Bir JWT, üç bölümden oluşur: başlık (header), yük (payload) ve imza (signature) Başlık (Header): JWT'nin türünü ve kullanılan algoritmayı içerir. Yük (Payload): İstekte bulunan tarafın talep ettiği bilgileri içerir. Bu bilgiler kullanıcı kimliği, yetkiler, ve/veya ekstra veriler olabilir. İmza (Signature): Başlık ve yükün birleşimi, belirli bir anahtar kullanılarak şifrelenir. Bu imza, JWT'nin doğruluğunu ve bütünlüğünü sağlar.

JWT'ler, özellikle web uygulamalarında kimlik doğrulama ve yetkilendirme işlemlerinde sıklıkla kullanılır. Token'lar, sunucu ve istemci arasında güvenli bir şekilde bilgi alışverişi yapmak için kullanılır. Örneğin, bir kullanıcının oturumunun süresini belirlemek veya belirli bir kullanıcının belirli bir kaynağa erişim yetkisini doğrulamak için JWT'ler kullanılabilir.

Swagger ve JWT Testi

Swagger'ın doğrudan JWT doğrulaması için bir desteği yoktur(Postman gibi). Ancak, JWT doğrulamasını API belgelerine eklenebilir ve ardından Swagger ile bu belgeleri kullanarak test yapılabilir.

```yaml
swagger: '2.0'
info:
  version: 1.0.0
  title: Sample API
securityDefinitions:
  JWT:
    type: apiKey
    name: Authorization
    in: header
    description: Format: Bearer {token}
paths:
  /endpoint:
    get:
      security:
        - JWT: []
      responses:
        200:
          description: OK
```

Jwt Türleri

HS256, HS384, HS512 (HMAC-SHA): Bu JWT türleri, simetrik anahtarlı (şifreleme ve doğrulama için aynı anahtarın kullanılması) algoritmalar kullanarak imzalanır ve doğrulanır. HS256, HS384 ve HS512, sırasıyla HMAC-SHA256, HMAC-SHA384 ve HMAC-SHA512 algoritmalarını temsil eder.

RS256, RS384, RS512 (RSA): RSA algoritmasını kullanan JWT türleridir. Bu türler, genel/özel anahtar çifti kullanarak imzalanır ve doğrulanır. RS256, RS384 ve RS512, sırasıyla RSA-SHA256, RSA-SHA384 ve RSA-SHA512 algoritmalarını ifade eder.

ES256, ES384, ES512 (Elliptic Curve): Bu JWT türleri, eliptik eğri algoritmalarını kullanarak imzalanır ve doğrulanır. ES256, ES384 ve ES512, sırasıyla Elliptic Curve Digital Signature Algorithm (ECDSA) kullanılarak SHA-256, SHA-384 ve SHA-512 ile imzalanmış JWT'leri ifade eder.

Bazı JWT'nin bir imzası olmadığını belirtmek için kullanılır(none). Bu tür JWT'lerin güvenliği yoktur ve sadece güvenli olmayan ortamlarda veya test amaçlı kullanılmalıdır.

Bearear Token

Bir HTTP yetkilendirme başlığıdır ve çoğunlukla OAuth 2.0 ve diğer yetkilendirme mekanizmalarında kullanılır. Bearer token, bir kullanıcının yetkilendirme ve kimlik doğrulama bilgilerini sunmak için kullanılır ve bu bilgilerin güvenliğini sağlamak için HTTPS üzerinden iletilmelidir. Bearer token, bir isteğin "Authorization" başlığı altında gönderilir ve genellikle "Bearer" kelimesiyle başlar.

KEYSTORE

Keystore, kriptografik anahtarları ve sertifikaları saklamak için kullanılan bir tür veri deposudur. Genellikle bir dosya, veritabanı veya donanım güvenlik modülü gibi farklı biçimlerde olabilir.

Keystore'nin Temel İşlevleri:

Anahtar Yönetimi: Keystore, genellikle asimetrik anahtarlar, simetrik anahtarlar ve diğer kriptografik anahtarlar gibi çeşitli anahtar türlerini saklar. Bu anahtarlar, şifreleme, imza ve dijital kimlik doğrulama gibi işlemlerde kullanılır.

Sertifika Saklama: Keystore, dijital sertifikaları saklamak için de kullanılır (örneğin JKS doyası). Sertifikalar, genellikle anahtarların sahipliğini doğrulamak, şifreleme anahtarlarını değiştirmek veya SSL/TLS ile güvenli bağlantılar kurmak gibi işlemlerde kullanılır.

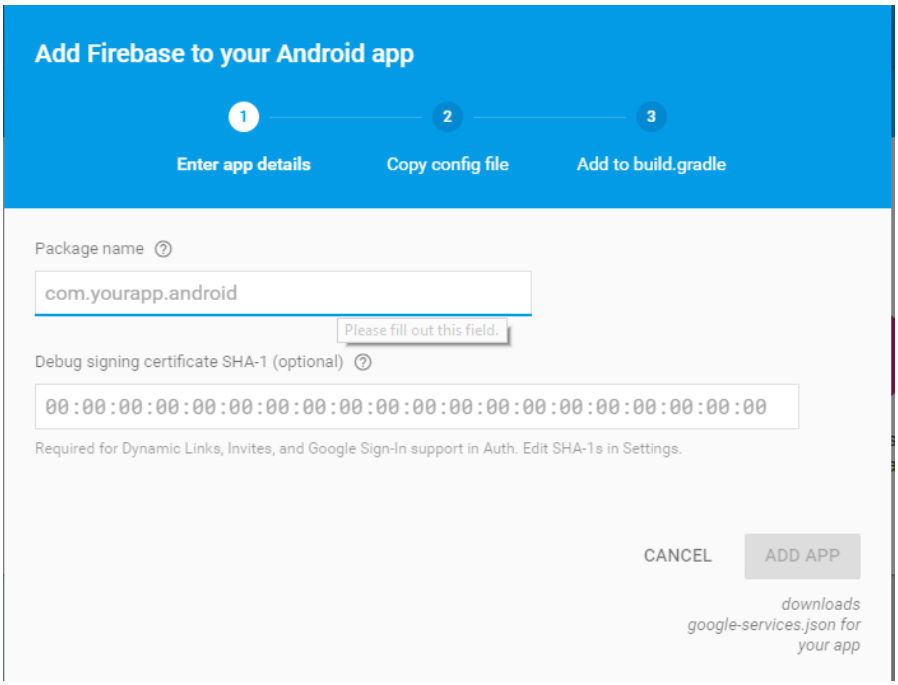
Not: Java-Kotlin uygulamaları genellikle Java KeyStore(JKS) kullanırken OpenSSL uygulamaları ise PKCS#12 formatındaki keystore'leri tercih edebiliyor çünkü daha esnektir, Flutter'da Apache Hive kullanılması gibi.

Keystore'nin Kullanımı:

Android Studio'da Key Store uygulamayı imzalamak için özel bir JKS oluşturmamızı ister ve bu JKS dosyası SHA certificate fingerprints(SHA sertifikası parmak izleri) içerir özellikle Play Console' da uygulama güvenliğini sağlamak için (App güncelleme, kimlik doğrulama ,vs.) kullanılır. Aynı zamanda çoğu Google hizmetleri bu SHA üzerinden haberleşir. Örneğin, Firebase ve Play Consoledeki App'i bağlamak gibi.

```
android {
    signingConfigs {
        release {
            // JKS dosyanızın konumu
            storeFile file('/path/to/your/keystore.jks')
            // JKS dosyanızın şifresi
            storePassword 'your_store_password'
            // Alias ismi
            keyAlias 'your_alias'
        }
    }
}
```

JKS gradle'a eklenmesi



SHA bilgileri Firebase İletişim Örneği

Keystore'nin Kotlin'de Kullanımı:

Kotlinde, Javada olan kütüphaneleri kullanabildiği için JKS dosyalarını oluşturmak, KeyStore sınıfından instance olmak kolay bir davranış oluyor.

```
fun getKey(): SecretKey {
    val keystore = KeyStore.getInstance( type: "AndroidKeyStore")
    keystore.load( param: null)

    val secretKeyEntry = keystore.getEntry( alias: "MyKeyAlias",  protParam: null) as KeyStore.SecretKeyEntry
    return secretKeyEntry.secretKey
}
```

Keystore özellikle Token veya özel anahtarları şifreleyerek Shered Preference' de saklanmasını sağlar, bu da uygulama içerisinde kullanılan API, Token, vs. özel şifrelerin başkaları tarafından kullanılmasının önüne geçer ve cihaz hafızasında bu şifrelenen veri güvenli bir şekilde yeri geldiğinde encrypt ve decrypt yöntemleriyle kullanılır.

```kotlin
fun encryptData(data: String): Pair<ByteArray, ByteArray> {
    val cipher = Cipher.getInstance( transformation: "AES/CBC/NoPadding")

    var temp = data
    while (temp.toByteArray().size % 16 != 0)
        temp += "\u0020"

    cipher.init(Cipher.ENCRYPT_MODE, getKey())

    val ivBytes = cipher.iv
    val encryptedBytes = cipher.doFinal(temp.toByteArray(Charsets.UTF_8))

    return Pair(ivBytes, encryptedBytes)
}
```

```kotlin
fun decryptData(ivBytes: ByteArray, data: ByteArray): String{
    val cipher = Cipher.getInstance( transformation: "AES/CBC/NoPadding")
    val spec = IvParameterSpec(ivBytes)

    cipher.init(Cipher.DECRYPT_MODE, getKey(), spec)
    return cipher.doFinal(data).toString(Charsets.UTF_8).trim()
}
```

Bu örneklerde Chiper sınıfında instance alarak, veriler şifrelenir ve çözülmek isteniyorsa burada çözülür.

Örnek Şifreleme:

```kotlin
val sharedPreferences = getSharedPreferences(PREF_NAME, Context.MODE_PRIVATE)

val pair = encryptData( data: "p1r2i3v4a5t6e.7k8e9y")

val encryptedBase64 = Base64.encodeToString(pair.second, Base64.DEFAULT)
val ivBase64 = Base64.encodeToString(pair.first, Base64.DEFAULT)


val editor = sharedPreferences.edit()
editor.putString(IV_KEY, ivBase64)
editor.putString(DATA_KEY, encryptedBase64)
editor.apply()


val ivFromPrefs = Base64.decode(sharedPreferences.getString(IV_KEY,  defValue: ""), Base64.DEFAULT)
val dataFromPrefs = Base64.decode(sharedPreferences.getString(DATA_KEY,  defValue: ""), Base64.DEFAULT)

// Veriyi çöz
val decryptedData = decryptData(ivFromPrefs, dataFromPrefs)
println("Encrypted data: $encryptedBase64")
println("Decrypted data: $decryptedData")
```

Burada ise, veri şifrelenir Base64 ile encode edilir ve yeni bir Shared Preference oluşturulur, şifrelenen veri buraya kaydedilir.

```
I  Encrypted data: 4wOuPvQ7NHi0D8V1QGPz7VZoS8BrY06902Xzh6YJLeU=
I  Decrypted data: p1r2i3v4a5t6e.7k8e9y
```

Şifrelenen veri asıl verinin Logcat çıktısı

```
<map>
    <string name="data">4wOuPvQ7NHi0D8V1QGPz7VZoS8BrY06902Xzh6YJLeU=&#10;    </string>
    <string name="iv">Aa4FZ6OgYgUw6jxAhyVijg==&#10;    </string>
</map>
```

Şifrelenen verinin Shared Prefernces'ta ki son hali (buradaki iv(başlatma vektörü) chiper.iv ile oluşturulan rasgele bir değerdir ve şifrelemenin güvenliğini arttırmak için kullanılır.)

Jetpack Library Kullanarak EncryptedPreferenceManager Kullanımı:

Önceki örnekte verilen kod örneği temel ve uzun olmakla birlikte kullanım zorluğu getirmektedir bunun yerine Jetpack security kütüphanesini kullanmak hem kod yazılmasında kolaylık sağlar hem de yeni çıkacak olan sürümlerle kütüphaneyi güncel tutarak veri güvenliği üst seviyeye çıkarılabilir.

```
implementation "androidx.security:security-crypto:1.1.0-alpha03"
```

```
implementation (libs.androidx.security.crypto)
```

```
package com.sercancelik.libraryencryptedsharedpreferences

import android.content.Context
import androidx.security.crypto.EncryptedSharedPreferences
import androidx.security.crypto.MasterKeys

class EncryptSharedPrefenceManager(context: Context) {
    private val mainKeyAlias = MasterKeys.getOrCreate(MasterKeys.AES256_GCM_SPEC)
    private val encryptedSharedPreferences = EncryptedSharedPreferences.create(
        fileName: "Shared_prefs",
        mainKeyAlias, context,
        EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
        EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
    )
    private val encryptedEditor = encryptedSharedPreferences.edit()
```

Bu kod örneğinde ise MasterKeys sınıfından eğer yoksa yeni bir anahtar oluşturulur, daha sonra oluşturulan anahtar Shered Prefences' ın içine yazılır ve bu SheredPfrefencesManager sınıfı hangi Context'te kullanılması isteniyorsa orada çağırılır. Dosya adı (Sehered Preferencesin adı)

ve hangi anahtarın çitinden hangisi şifrelenecekse (Key-Value) onu şifreler her iki değerin aynı anda şifrelenmesi güvenlik açısından daha iyi olacaktır.

```
private val keyName = "name"
private val keyAge = "age"

var name
    get() = encryptedSharedPreferences.getString(keyName, defValue: "").toString()
    set(value) {
        encryptedEditor.putString(keyName, value)
        encryptedEditor.commit()

    }
var age
    get() = encryptedSharedPreferences.getInt(keyAge, defValue: 0)
    set(value) {
        encryptedEditor.putInt(keyAge, value)
        encryptedEditor.commit()

    }
```

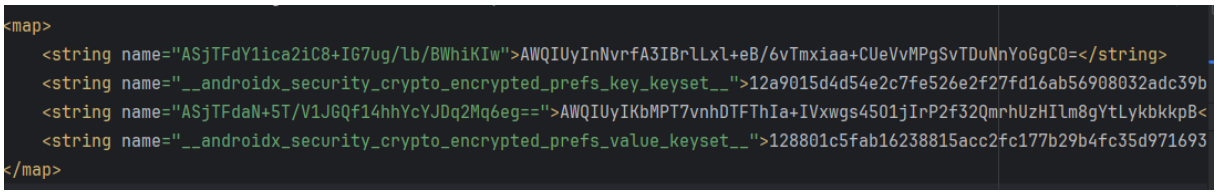
Burada ise Fieldların getter - settter metodları yazılıyor. Bu veriler şifreli SheredPreferences' a kaydediliyor ve Field'lerden çağırılması isteniyor.

```
val encryptSharedPrefenceManager = EncryptSharedPrefenceManager( context: this)
btnSet.setOnClickListener { it: View!
    if (etName.text.isEmpty()) {
        Toast.makeText( context: this, text: "Lütfen Bir İsim Girin", Toast.LENGTH_SHORT).show()
    } else {
        encryptSharedPrefenceManager.name = etName.text.toString()
        encryptSharedPrefenceManager.age = etAge.text.toString().toInt()
    }

}
btnGet.setOnClickListener { it: View!

    etName.setText(encryptSharedPrefenceManager.name)
    etAge.setText(encryptSharedPrefenceManager.age.toString())
}
btnClear.setOnClickListener { it: View!
    etName.setText("")
    etAge.setText("")
}
```

Main sınıfındaki kodlar: UI konrtolü ve EncrypteSheredPrefenceManager' in çağırılması



SherePrefences dosyası burada hem Key hem de Value'lerin şifrelendiği görülüyor

Sercan ÇELİK