Orange API Documentation

Linking Wallet API V2.0.0

Abstract

These APIs are designed to enable Linking Wallet functionalities that support customer subscriptions for merchants and businesses offering dynamic plans, facilitating seamless and automated financial interactions across platforms.

Orange Money Technical Team
orangemoneytechnical.ojo@orange.com
11 November 2015

Orange Restricted

Orange API Documentation

This document contains a collection of APIs that serve Linking Wallet functionalities that serve customer subscriptions on merchants' platforms and business entities with dynamic plans.

1. Introduction

These APIs are intended for partners and developers integrating wallet services into their platforms for enhanced financial interactions between individual users and businesses.

Base URL (Production): https://orangemoney.orange.jo:1594/

Base URL (Sandbox): https://om-dev.orange.jo:1445/

2. Test requirement

To ensure a smooth integration and testing processes. The following points need to be considered for testing and production:

1. **Two phone numbers are required for testing purposes:**
   - **One associated with the Personal Wallet.**
   - **One associated with the Business/Merchant Wallet.**
2. **Credentials will be sent by Orange Money through text message on the number related to business wallet.**
3. **Credentials for testing (staging) are different from the ones for live (production).**
4. **The agent needs to provide a static IP address that needs to be whitelisted by Orange Money during testing (staging) and another one for live (production).**
5. **The agent can review all transactions from the Agent Portal.**

3. Authentication

All API endpoints require JWT token–based authentication to be included in the Authorization header:

- **Authorization: Bearer <AccessToken>.**

📌 ***Notes:***

- The external client must first authenticate using the Authorization API, which returns a JWT token.

Authorization

Path	api/ExternalAPI/Authorization
HTTP Method	POST
JSON format	Content-Type: application/json

Request

Parameter	Type	Required	Description	Encryption
UserName	string	Yes	The UserName	Yes
Password	string	Yes	The Password	Yes

Response

Parameter	Type	Description
AccessToken	String	Access Token
token_type	String	Toke type always Bearer
expires_in	int	ExpiresIn
errorCode	String	Error code
errorDescription	String	Error message text
isSuccess	bool	Indicates if the transaction was successful
IsOTPRequired	bool	Specifies if an OTP is required for the transaction
errors	List [{ "description": "string", "descriptionAr": "string", }]	List of Errors

4. API Structure & Design

Recurring Payment

Path	**api/Subscriptions/V2/LinkingWallet**
HTTP Method	**POST**
JSON format	**Content-Type: application/json**

5. Error Handling

The API's use standard HTTP status codes:

Code	Meaning
200	**OK**
500	**Internal Server Error**

The API's also return the following Application-Level errors codes:

Note: these error codes are returned in the response with 200 OK HTTP status code.

Code	Description	Description Arabic
236	The amount entered is invalid	المبلغ المدخل غير صحيح
311	**The minimum allowed amount is 0.1 JD**	الحد الأدنى للمبلغ المسموح به هو **0.1** دينار
53	Transaction is not allowed	المعاملة غير مسموح بها
230	Invalid Signature	Invalid Signature
110	Agent wallet not found	محفظة الوكيل غير موجودة
334	The user is not authorized to perform this transaction	المستخدم غير مخول بإجراء هذه المعاملة
335	Duplicate Merchant Reference	رقم المرجع مكرر
46	Transaction amount exceeds the allowed limit	قيمة هذه الحركة أعلى من الحد المسموح

112	Sorry, we are facing an issue at the moment, please try again later	نأسف، نواجه مشكلة في الوقت الحالي، يرجى المحاولة لاحقاً
187	Invalid Transaction	معاملة غير صحيحة
407	Invalid Reference Number	رقم مرجعي غير صالح
411	Duplicate Merchant Transaction Reference	مرجع معاملة التاجر المكرر
406	Invalid Merchant Code	رمز التاجر غير صالح

6. Endpoints

6.1 Linking Wallet

The transaction process is carried out in two steps:

1. **Validation Step: Send the request with IsConfirmed = false. This is used to validate the transaction details before execution. This also includes sending OTP to the customer.**
2. **Execution Step: Send the same request again with IsConfirmed = true and including the OTP to finalize and perform the transaction.**

Method	**POST**
URL	**api/Subscriptions/V2/LinkingWallet**
Header	**Key: Signature** **Value: "string"**
Request	**{** **"MerchantCode": "U2FsdGVkXDSEFOVF9DT0RFXzEyMw==",** **"CustomerUID": "U2FsdGUT01FUl9VSURfMTIzNDU2Nzg5MA==",** **"CustomerPhoneNumber": "U2FsdGV19QSE9DkxMjM0NTY3OA==",** **"IsConfirmed": false,** **"ReferenceNumber ": "U2FsdGVkfMjAyNTExMTBfMDAx==",**

Response	"OTP": "U2FsdGVkX19PVFBfMTIzNDU2==" } { "MerchantCode": null, "CustomerUID": null, "ReferenceNumber": null, "OTP": null, "isSuccess": true, "errors": [] }

7. Data Models

7.1 Linking Wallet

7.1.1 Request

Field	Type	Required	Encrypted	Description
MerchantCode	**string**	**Yes**	**Yes**	Code of the user (will be provided soon from orange once the wallet created)
ReferenceNumber	**string**	**Yes**	**Yes**	reference to Identify the Customer (generated from **OM** side)
CustomerUID	**string**	**Yes**	**Yes**	reference to Identify the Customer (generated from **Merchant** side)
CustomerPhoneNumber	**string**	**Yes**	**Yes**	Customer phone number
IsConfirmed	**bool**	**Yes**	**No**	to identify the step
OTP	**string**	**No**	**Yes**	Entered OTP by the customer

7.1.2 Response

Field	Type	Description

MerchantCode	String	Merchant code
CustomerUID	String	From the request
ReferenceNumber	String	OM generate it, and it will be sent at the second step
OTP	String	One time password
isSuccess	bool	Indicates if the transaction was successful
IsOTPRequired	bool	Specifies if an OTP is required for the transaction
Errors	List [{ "description": "string", "descriptionAr": "string", }]	List of Errors

8. Security

To ensure secure communication and data integrity between client systems and the API's, the followings must be followed:

8.1 Encryption:

The encryption algorithm used is Advanced Encryption Standard (AES) with the flowing AES Key & HMAC Key:

Linking Wallet

	Staging	Production
AES Key	**yC/o3bNNivbrN40v9Vc+CV499jIY0U3O7kV02+EPwbI=**	**9U55TUNQU8j698A9LNukVwQQn6rCSUXdAYzcCIbT7t8=**
HMAC Key	**1rO7BTr4AnKSmm1UPosyXkcEp5VJQfBmFoPWeooaX7M=**	**Q0ZpcDT9W5xlYZPP5m9/GK2kut1rzeXKKYjRk+wCNQ4=**

Here's a step-by-step breakdown of how the method Encrypt AES works:

1. Decoding Input Keys (AES and HMAC):

- **AES Key:**
  The provided base64AesKey string is decoded into a byte array to be used for the AES encryption process.

- **HMAC Key:**
  The base64HmacKey string is also decoded into a byte array, which will be used to compute the HMAC for integrity verification.

  C# Example:
  byte[] key = Convert.FromBase64String(AesKey);
  byte[] hmacKey =
  Convert.FromBase64String(HmacKey);

2. Setting Up AES for Encryption:

- **An Aes object is created and configured with the provided AES key.**
- **Mode: Set to CBC (Cipher Block Chaining), which requires an Initialization Vector (IV) for encryption.**
- **Padding: Set to PKCS7 (to handle cases where the plaintext size isn't a multiple of the AES block size).**
- **IV Generation: A random IV is generated for this encryption session to ensure that even if the same plaintext is encrypted multiple times, the ciphertext will be different each time.**

  C# Example:
  using var aes = Aes.Create();
  aes.Key = key;
  aes.Mode = CipherMode.CBC;
  aes.Padding = PaddingMode.PKCS7;
  aes.GenerateIV(); // Generates a new random IV each time

3. Encrypting the Plaintext:

- **Encryptor Creation: A CryptoStream is used to perform the actual encryption. It applies the AES encryption to the plaintext and stores the result in the cipherBytes array.**
- **The plaintext is written to a StreamWriter, which is wrapped in a CryptoStream. The CryptoStream applies the encryption and writes the encrypted data to a MemoryStream.**
- **Ciphertext: The result is the encrypted data (ciphertext) of the plaintext.**

  C# Example:

```
byte[] cipherBytes;
using (var encryptor = aes.CreateEncryptor())
using (var ms = new MemoryStream())
using (var cs = new CryptoStream(ms, encryptor,
CryptoStreamMode.Write))
using (var sw = new StreamWriter(cs, Encoding.UTF8))
{
    sw.Write(plainText);
    sw.Close();
    cipherBytes = ms.ToArray();
}
```

4. Combining IV and Ciphertext:

- The IV (Initialization Vector) and the ciphertext are concatenated together into a single byte array (ivAndCipher). This step ensures that the IV is included in the final encrypted data.
- IV: The IV is placed at the beginning, followed by the ciphertext.

```
C# Example:
byte[] ivAndCipher = new byte[aes.IV.Length + cipherBytes.Length];
Buffer.BlockCopy(aes.IV, 0, ivAndCipher, 0, aes.IV.Length);
Buffer.BlockCopy(cipherBytes, 0, ivAndCipher, aes.IV.Length,
cipherBytes.Length);
```

5. Computing the HMAC for Integrity:

- HMAC Calculation: A HMAC-SHA256 hash is calculated over the concatenated IV and ciphertext (ivAndCipher). This HMAC ensures that the encrypted data has not been tampered with.
- HMAC Key: The HMAC is computed using the provided HMAC key.

```
C# Example:
byte[] hmac;
using (var hmacSha = new HMACSHA256(hmacKey))
hmac = hmacSha.ComputeHash(ivAndCipher);
```

6. Combining HMAC, IV, and Ciphertext:

- The HMAC, IV, and Ciphertext are concatenated together into a final byte array (finalData).

- This final byte array represents the complete encrypted data that includes both the ciphertext and the HMAC for integrity checking.

```
C# Example:
byte[] finalData = new byte[hmac.Length + ivAndCipher.Length];
Buffer.BlockCopy(hmac, 0, finalData, 0, hmac.Length);
Buffer.BlockCopy(ivAndCipher, 0, finalData, hmac.Length,
ivAndCipher.Length);
```

7. Base64 Encoding:

- The final concatenated data (HMAC | IV | Ciphertext) is base64-encoded to make it suitable for transmission or storage. This ensures that the resulting string is in a safe, readable format.
- The base64-encoded result is returned as a string.

```
C# Example:
return Convert.ToBase64String(finalData);
```

8.2 Signature From Header :

The signature field is used to validate the authenticity of the request. The signature is generated using the SHA-256 hashing algorithm.

📌 *Note: use this link https://emn178.github.io/online-tools/sha256.html*

To generate the signature, use the following formulas per each API:

	Signature Formula
Recurring Payment	(API KEY + MerchantCode + CustomerUID + IsConfirmed (True/False) + OTP + API KEY)

API KEY: ABC123

📌 *Note: All the above signature values must be generated before hashing and/or encryption.*

9. Contact Information

For support or questions, please contact:

Email: orangemoneytechnical.ojo@orange.com

Phone: