Push Provisioning - iOS implementation guide

Prerequisites

This documentation assumes that you already have a configured integration layer for the TMP API. All subsequent information assumes that the connection has been properly configured. The provided code samples are for reference purposes only and should not be used in production environments without appropriate modifications.

It is also assumed that the application has already retrieved the list of cards from DataCore and has access to their identifiers in the following form:

```
let cardIds: [String]
```

Apple Push Provisioning Configuration

To enable push provisioning for Apple Wallet, your application must be properly configured and approved by Apple.

Requirements

Push provisioning requires a dedicated entitlement that is not enabled by default for standard iOS applications. Before implementation, ensure that:

- Your organization is enrolled in the Apple Developer Program.
- You have an active issuer agreement and the necessary payment network approvals.
- Your application has been approved for push provisioning capabilities.

Required Entitlement

The application must include the appropriate Wallet provisioning entitlement in the app’s entitlements file. This entitlement is granted individually by Apple and is required to access provisioning APIs related to payment cards and passes.

Without this entitlement:

- Push provisioning APIs will not be accessible.
- Wallet provisioning flows will fail at runtime.
- The application may be rejected during App Review.

Requesting Access

To obtain the required entitlement:

1. Contact your Apple representative or provisioning support contact.
2. Provide details about:
   - Your application
   - Issuer information
   - Payment network participation
   - Intended provisioning flow
3. Complete any requested compliance or certification processes.

Apple may require additional validation before enabling the entitlement for your App ID.

Apple Developer Configuration

After approval:

1. Enable the provisioning capability for the selected App ID in the Apple Developer portal.
2. Regenerate provisioning profiles associated with the application.
3. Download and install updated provisioning profiles.
4. Verify that the entitlement is present in the signed application build.

Additional Notes

- Entitlements are environment-specific and may differ between development and production environments.
- Distribution builds should always be signed using provisioning profiles containing the approved entitlement.
- Access to push provisioning functionality may also depend on issuer and payment network configuration performed outside of the iOS application itself.

Retrieving Existing Passes from PassKit

Before starting the provisioning flow, the application should retrieve existing passes from PassKit.

The `deviceAccountIdentifier` values are required to query the TMP API for the current token status and determine whether a card has already been provisioned on the device or on paired devices such as Apple Watch.

Example

```
import PassKit

let passLibrary = PKPassLibrary()

let localPasses = passLibrary.passes().compactMap {
    $0.secureElementPass?.deviceAccountIdentifier
}

let remotePasses = passLibrary.remoteSecureElementPasses.compactMap {
    $0.deviceAccountIdentifier
}

let deviceAccountIdentifiers = localPasses + remotePasses
```

Local passes represent cards provisioned directly on the iPhone, while remote passes represent cards provisioned on external or paired devices, including **Apple Watch**.

Fetching Current Tokens from TMP API

After collecting `deviceAccountIdentifiers` from PassKit, the application should send them to the TMP API to retrieve the current token status.

Use the endpoint:

```
POST /issuer/push-provisioning/tokens/searches
```

For Apple Pay, the request must include `walletType: APPLE_PAY` and `tokenUniqueReferences`, where `tokenUniqueReferences` should contain the previously collected `deviceAccountIdentifiers`. The response contains token data such as `externalCardId`, `tokenStatus`, and `processStatus`.

Example Request Body

```
{
  "walletType": "APPLE_PAY",
  "tokenUniqueReferences": [
    "8YUZErg1CWsPG5uVa",
    "9XVAfh2DXWtQH6wWb"
  ]
}
```

Swift Example

```
import Foundation

struct GetTokensRequest: Encodable {
    let walletType: String
    let tokenUniqueReferences: [String]
}

struct TokenResponse: Decodable {
    let tokenUniqueReference: String?
    let panUniqueReference: String?
    let externalCardId: String?
    let tokenStatus: String?
```

```
    let authorizationPath: String?
    let processStatus: String?
}

let requestBody = GetTokensRequest(
    walletType: "APPLE_PAY",
    tokenUniqueReferences: deviceAccountIdentifiers
)

let url = URL(string: "https://your-tmp-api-base-url/issuer/push-
provisioning/tokens/searches")!
var request = URLRequest(url: url)
request.httpMethod = "POST"
request.setValue("application/json", forHTTPHeaderField: "Content-Type")
request.httpBody = try JSONEncoder().encode(requestBody)

let (data, response) = try await URLSession.shared.data(for: request)

guard let httpResponse = response as? HTTPURLResponse,
      httpResponse.statusCode == 200 else {
    throw URLError(.badServerResponse)
}

let tokens = try JSONDecoder().decode([TokenResponse].self, from: data)
```

Mapping TMP Tokens to DataCore Cards

After receiving the token list from TMP API, the application should match the response with the previously retrieved DataCore card identifiers.

The mapping should be done using:

```
externalCardId
```

Mapping Rules

- If a card ID is not present in the TMP response, the card has not been added to Apple Wallet.
- If a card ID is present in the TMP response, the card has already been added to Apple Wallet:
  - If `tokenStatus == "ACTIVE"`, the card has already been added and activated.
  - If `tokenStatus == "INACTIVE"`, the card has been added but requires activation.

Swift Example

```
enum WalletCardStatus {
    case notAdded
    case active
    case requiresActivation
}

let tokensByCardId = Dictionary(
    uniqueKeysWithValues: tokens.compactMap { token in
        token.externalCardId.map { ($0, token) }
    }
)

let walletStatuses: [String: WalletCardStatus] = Dictionary(
    uniqueKeysWithValues: cardIds.map { cardId in
        guard let token = tokensByCardId[cardId] else {
            return (cardId, .notAdded)
        }
```

```
        switch token.tokenStatus {
        case "ACTIVE":
            return (cardId, .active)
        case "INACTIVE":
            return (cardId, .requiresActivation)
        default:
            return (cardId, .notAdded)
        }
    }
)
```

UI Handling

Based on the resolved wallet status, the application should display the appropriate user action:

- `notAdded` → display the “Add to Apple Wallet” button
- `requiresActivation` → display the “Activate Card” button
- `active` → display information that the card has already been added to Apple Wallet

Push provisioning

To start Apple Wallet push provisioning, use `PKAddPaymentPassViewController` with `PKAddPaymentPassRequestConfiguration`.

The configuration should use the `.ECC_V2` encryption scheme.

```
import PassKit

let configuration = PKAddPaymentPassRequestConfiguration(encryptionScheme: .ECC_V2)!

configuration.cardholderName = "John Doe"
configuration.primaryAccountSuffix = "1234" // Last 4 digits of the card number

// Required when provisioning to Apple Watch.
// Use deviceAccountIdentifier from the phone PassItem.
configuration.primaryAccountIdentifier = phonePassItem.deviceAccountIdentifier

let viewController = PKAddPaymentPassViewController(
    requestConfiguration: configuration,
    delegate: delegate
)
```

If the card is being added to Apple Watch, `primaryAccountIdentifier` should be set using `deviceAccountIdentifier` from the phone `PassItem`.

Delegate Implementation

The application must implement `PKAddPaymentPassViewControllerDelegate`.

In `generateRequestWithCertificateChain`, call the TMP API endpoint:

```
POST /issuer/push-provisioning/signed-cards
```

For Apple Pay, the request requires `certificate`, `nonce`, and `nonceSignature`. The TMP response returns `activationData`, `ephemeralPublicKey`, and `encryptedData`, which are required to create `PKAddPaymentPassRequest`.

Swift Example

```
import PassKit

final class AddPaymentPassDelegate: NSObject, PKAddPaymentPassViewControllerDelegate {

    func addPaymentPassViewController(
        _ controller: PKAddPaymentPassViewController,
        generateRequestWithCertificateChain certificates: [Data],
        nonce: Data,
        nonceSignature: Data,
        completionHandler handler: @escaping (PKAddPaymentPassRequest) -> Void
    ) {
        Task {
            do {
                let signedCard = try await signCard(
                    certificates: certificates,
                    nonce: nonce,
                    nonceSignature: nonceSignature
                )

                let request = PKAddPaymentPassRequest()
                request.activationData = Data(base64Encoded: signedCard.activationData)
                request.ephemeralPublicKey = Data(base64Encoded:
signedCard.ephemeralPublicKey)
                request.encryptedPassData = Data(base64Encoded: signedCard.encryptedData)

                handler(request)
            } catch {
                controller.dismiss(animated: true)
            }
        }
    }

    func addPaymentPassViewController(
        _ controller: PKAddPaymentPassViewController,
        didFinishAdding pass: PKPaymentPass?,
        error: Error?
    ) {
        controller.dismiss(animated: true)

        if let error {
```

```
            // Handle provisioning error
            print("Apple Wallet provisioning failed: \\(error)")
            return
        }

        // Provisioning finished successfully
    }
}
```

Notes

- The code above is simplified and should be adapted to the existing TMP integration layer.
- Sensitive card data should not be handled directly in the mobile application unless explicitly required by the approved architecture.

Token Activation in the Mobile Application

The application should support a deeplink configured in MDES. The deeplink should contain a `serialNumber` parameter in the query string.

Based on this value, the application should find the matching PassKit item using the `serialNumber` property. Then, it should call the TMP API endpoint:

```
POST /issuer/push-provisioning/tokens/activations
```

The request should contain the token identifier. If the response field `issuerMobileAppAuthResponse` is `APPROVED`, the card has been successfully activated. For `DECLINED` or `FAILED`, the activation was not completed successfully.

Process Overview

1. Receive the deeplink from MDES.
2. Extract `serialNumber` from the query string.
3. Find the matching PassKit item by `serialNumber`.
4. Read its `deviceAccountIdentifier`.
5. Send it to TMP API as `tokenUniqueReference`.
6. Handle the activation response.

Swift Example

```
import Foundation
import PassKit

enum ActivationStatus: String, Decodable {
    case approved = "APPROVED"
    case declined = "DECLINED"
    case failed = "FAILED"
}

struct TokenActivationRequest: Encodable {
    let tokenUniqueReference: String
}
```

```
struct TokenActivationResponse: Decodable {
    let tokenUniqueReference: String?
    let cardLast4Digits: String?
    let issuerMobileAppAuthResponse: ActivationStatus
    let comment: String?
}

func handleActivationDeeplink(_ url: URL) async throws {
    guard
        let components = URLComponents(url: url, resolvingAgainstBaseURL: false),
        let serialNumber = components.queryItems?.first(where: { $0.name == "serialNumber"
})?.value
    else {
        throw URLError(.badURL)
    }

    let passLibrary = PKPassLibrary()

    guard let pass = passLibrary.passes().first(where: {
        $0.serialNumber == serialNumber
    }),
    let tokenUniqueReference = pass.secureElementPass?.deviceAccountIdentifier
    else {
        throw URLError(.cannotFindHost)
    }

    let requestBody = TokenActivationRequest(
        tokenUniqueReference: tokenUniqueReference
    )

    let url = URL(string: "<https://your-tmp-api-base-url/issuer/push-
provisioning/tokens/activations>")!
    var request = URLRequest(url: url)
    request.httpMethod = "POST"
    request.setValue("application/json", forHTTPHeaderField: "Content-Type")
    request.httpBody = try JSONEncoder().encode(requestBody)

    let (data, response) = try await URLSession.shared.data(for: request)

    guard let httpResponse = response as? HTTPURLResponse,
```

```
          httpResponse.statusCode == 200 else {
        throw URLError(.badServerResponse)
    }

    let activationResponse = try JSONDecoder().decode(
        TokenActivationResponse.self,
        from: data
    )

    switch activationResponse.issuerMobileAppAuthResponse {
    case .approved:
        // Card has been successfully activated.
        break

    case .declined, .failed:
        // Card activation failed.
        break
    }
}
```

Notes

- The deeplink format must match the configuration provided in MDES.
- The `serialNumber` value is used only to find the correct PassKit item.
- The activation request uses the token identifier obtained from the matching pass item.

Wallet Extension

Official Apple reference: https://applepaydemo.apple.com/in-app-provisioning-extensions

Wallet Extensions allow users to start card provisioning directly from Apple Wallet, without opening the issuer application first. Apple Wallet displays cards that are available for provisioning from the installed issuer app.

This feature requires two extensions:

- **Non-UI Extension** — provides status, available pass entries, and generates provisioning requests.
- **UI Extension** — handles user authentication when required.

Apple Wallet invokes the extension methods directly. The application should provide only cards that are eligible for provisioning and are not already added to Apple Wallet.

Process Overview

1. User opens Apple Wallet and starts adding a card.
2. Apple Wallet checks the issuer app extension status.
3. The extension reports whether pass entries are available.
4. If required, the UI extension authenticates the user.
5. The non-UI extension provides available cards using `passEntries`.
6. User selects a card.
7. The extension generates `PKAddPaymentPassRequest` using TMP API data.

In `passEntries`, provide data analogous to the in-app provisioning flow described earlier, including card metadata, suffix, cardholder name, and identifier required to start provisioning.

Non-UI Extension Example

```
import PassKit

final class IssuerProvisioningExtensionHandler: PKIssuerProvisioningExtensionHandler {

    override func status(
        completion: @escaping (PKIssuerProvisioningExtensionStatus) -> Void
    ) {
        let status = PKIssuerProvisioningExtensionStatus()
        status.passEntriesAvailable = true
        status.remotePassEntriesAvailable = true
```

```
        status.requiresAuthentication = true

        completion(status)
    }

    override func passEntries(
        completion: @escaping ([PKIssuerProvisioningExtensionPaymentPassEntry]) -> Void
    ) {
        let configuration = PKAddPaymentPassRequestConfiguration(encryptionScheme: .ECC_V2)!
        configuration.cardholderName = "John Doe"
        configuration.primaryAccountSuffix = "1234"

        let entry = PKIssuerProvisioningExtensionPaymentPassEntry(
            identifier: "card-id-123",
            title: "Example Card",
            art: UIImage(named: "card-art")!,
            addRequestConfiguration: configuration
        )

        completion([entry])
    }

    override func remotePassEntries(
        completion: @escaping ([PKIssuerProvisioningExtensionPaymentPassEntry]) -> Void
    ) {
        let configuration = PKAddPaymentPassRequestConfiguration(encryptionScheme: .ECC_V2)!
        configuration.cardholderName = "John Doe"
        configuration.primaryAccountSuffix = "1234"

        let entry = PKIssuerProvisioningExtensionPaymentPassEntry(
            identifier: "card-id-123",
            title: "Example Card",
            art: UIImage(named: "card-art")!,
            addRequestConfiguration: configuration
        )

        completion([entry])
    }

    override func generateAddPaymentPassRequestForPassEntryWithIdentifier(
```

```
        _ identifier: String,
        configuration: PKAddPaymentPassRequestConfiguration,
        certificateChain certificates: [Data],
        nonce: Data,
        nonceSignature: Data,
        completionHandler handler: @escaping (PKAddPaymentPassRequest?) -> Void
    ) {
        Task {
            do {
                let signedCard = try await signCard(
                    cardId: identifier,
                    certificates: certificates,
                    nonce: nonce,
                    nonceSignature: nonceSignature
                )

                let request = PKAddPaymentPassRequest()
                request.activationData = Data(base64Encoded: signedCard.activationData)
                request.ephemeralPublicKey = Data(base64Encoded:
signedCard.ephemeralPublicKey)
                request.encryptedPassData = Data(base64Encoded: signedCard.encryptedData)

                handler(request)
            } catch {
                handler(nil)
            }
        }
    }
}
```

Signing Card with TMP API

Inside `generateAddPaymentPassRequestForPassEntryWithIdentifier`, call:

```
POST /issuer/push-provisioning/signed-cards
```

Use the certificate chain, nonce, and nonce signature provided by Apple Wallet. TMP returns `activationData`, `ephemeralPublicKey`, and `encryptedData`, which are used to create `PKAddPaymentPassRequest`.

```
struct SignCardResponse: Decodable {
    let activationData: String
    let ephemeralPublicKey: String
    let encryptedData: String
}

func signCard(
    cardId: String,
    certificates: [Data],
    nonce: Data,
    nonceSignature: Data
) async throws -> SignCardResponse {
    // Use your existing TMP API integration layer here.
    // The payload is analogous to the in-app provisioning flow.

    fatalError("Example only")
}
```

Sharing Data Between the App and Extensions

Wallet Extensions run in a separate process from the main application. The main app may not be running when Apple Wallet invokes the extension. Because of that, shared data should be stored in a location available to both the app and its extensions.

App Groups

Configure an App Group for:

- the main application target,
- the non-UI Wallet Extension,
- the UI Wallet Extension.

Use the same App Group identifier for all targets, for example:

```
group.com.example.issuerapp
```

Shared non-sensitive data can be stored using:

```
let sharedDefaults = UserDefaults(
    suiteName: "group.com.example.issuerapp"
```

```
)


sharedDefaults?.set(cardIds, forKey: "cardIds")
let cardIds = sharedDefaults?.stringArray(forKey: "cardIds")
```

Shared Keychain

For sensitive data, such as authentication tokens or user session data, configure **Keychain Sharing** for the main app and both extensions.

All targets must use the same Keychain Access Group, for example:

```
$(AppIdentifierPrefix)com.example.issuerapp.shared
```

Use the shared keychain to store data required by the extensions to authenticate requests and communicate with the backend/TMP integration layer.

Notes

- Wallet Extensions require the appropriate Apple entitlement and allow listing.
- The user must open and log in to the issuer app at least once before Apple Wallet can detect the extensions.
- `passEntries` should return only cards that are eligible for provisioning.
- Existing cards should be filtered out using the same PassKit and TMP token status logic described in previous sections.