amazon payment services

.NET SDK
Developers Guide

Table of Contents

Copyright Statement

Trademark

Contact Us

integration-ps@amazon.com

https://paymentservices.amazon.com

1. About this document

This document describes how to integrate the .NET SDK into your solution.

2. Integrations steps

Step 1: Install SDK NuGet Package

Install the .NET SDK NuGet Package from your solution or download the .dll file from the GitHub repository.

Step2: Add Serilog configuration

After the .NET SDK installation, make sure the following packages are installed. If not, please install them.

```
<PackageReference Include="Serilog" Version="2.10.0" />
<PackageReference Include="Serilog.Enrichers.Environment" Version="2.2.0" />
<PackageReference Include="Serilog.Extensions.Logging" Version="3.1.0" />
<PackageReference Include="Serilog.Formatting.Compact" Version="1.1.0" />
<PackageReference Include="Serilog.Settings.Configuration" Version="3.3.0" />
<PackageReference Include="Serilog.Sinks.Async" Version="1.5.0" />
<PackageReference Include="Serilog.Sinks.Console" Version="4.0.1" />
<PackageReference Include="Serilog.Sinks.File" Version="5.0.0" />
```

Table 1.1. Serilog Packages

The next part for configuration of Serilog is to create a JSON file which contains the configuration for Serilog. In this configuration, you can change the path for the file. Also, here you can change the minimum level of logging. In the JSON file, we have set the Default parameter to “Debug”.

```
{
  "Serilog": {
    "Using": [
      "Serilog.Sinks.Async"
    ],
    "MinimumLevel": {
      "Default": "Debug",
      "Override": {
        "Microsoft": "Warning",
        "System": "Warning"
      }
    },
    "WriteTo": [
      {
        "Name": "Async",
        "Args": {
          "blockWhenFull": true,
          "configure": [
            {
              "Name": "Console"
            },
            {
              "Name": "File",
              "Args": {
                "formatter": "Serilog.Formatting.Json.JsonFormatter, Serilog",
                "path": "./logs/$APPLICATIONNAME.$HOSTNAME.json",
                "rollOnFileSizeLimit": true,
                "rollingInterval": "Day",
                "fileSizeLimitBytes": "1053696",
                "shared": true,
                "retainedFileCountLimit": 7
              }
            }
          ]
        }
      }
    ],
    "Properties": {
      "ApplicationName": "$APPLICATIONNAME"
    }
  }
}
```

Table 1.2. JSON Configuration for Serilog

You, also, need to create an instance of LoggingConfiguration class which is provided by the SDK. In your program class, you must initiate the constructor of the class. For example, see the following table (Table 1.3.).

```
LoggingConfiguration loggingConfiguration = new LoggingConfiguration(
    YourServiceCollection,
    @"YourSerilogConfigJsonPath",
    "YourApplicationName");
```

Table 1.3. Usage of Logging Configuration method

Here, we have created an instance of the LoggingConfiguration class. The parameters of the constructor, in order, are the following:

- serviceCollection – the application service collection
- jsonLoggingPath – the path to above created JSON file
- applicationName – the name of the application

STEP 3: Add SDK Configuration

As a merchant you need to send to the gateway some mandatory properties like access code, merchant identifier, request sha phrase, response sha phrase and sha type. We have created a method which helps you to globalize those properties.

To use that method, first step is to create a JSON file which has the structure from the following table (Table 1.4.).

```
{
  "SdkConfiguration": {
    "Environment": "Test",
    "AccessCode": "YourAccessCode",
    "MerchantIdentifier": "YourMerchantIdentifier",
    "RequestShaPhrase": "YourRequestShaPhrase",
    "ResponseShaPhrase": "YourResponseShaPhrase",
    "ShaType": "YourShaType"
  }
}
```

Table 1.4. SDK Configuration without Apple Pay

If you want integration with Apple Pay, you should use the structure from the following picture (Table 1.5.).

```
{
  "SdkConfiguration": {
    "Environment": "Test",
    "AccessCode": "YourAccessCode",
    "MerchantIdentifier": "YourMerchantIdentifier",
    "RequestShaPhrase": "YourRequestShaPhrase",
    "ResponseShaPhrase": "YourResponseShaPhrase",
    "ShaType": "YourShaType",
    "ApplePay": {
      "AccessCode": "YourAccessCode",
      "MerchantIdentifier": "YourMerchantIdentifier",
      "RequestShaPhrase": "YourRequestShaPhrase",
      "ResponseShaPhrase": "YourResponseShaPhrase",
      "ShaType": "YourShaType",
      "DisplayName": "YourDisplayName",
      "MerchantUid": "YourMerchantUid",
      "DomainName": "YourDomainName"
    }
  }
}
```

Table 1.5. SDK Configuration with Apple Pay

After the JSON file is created, you must call the method *SdkConfiguration*. Configure the following parameters:

- filePath – the path to the above created JSON file
- loggingConfiguration – the logging configuration created at Step2
- applePayConfiguration – this parameter is optional; in case you have integration with Apple Pay you must send the certificate (see Table 1.6)

```
LoggingConfiguration loggingConfiguration = new LoggingConfiguration(
   YourServiceCollection,
   @"YourSerilogConfigJsonPath",
   "YourApplicationName");

var certificate = new X509Certificate2("YourCertificateFilePath", "YourCertificatePassword");

var serviceProvider = SdkConfiguration
   .Configure(@"YourSdkConfigurationJsonFilePath",
          loggingConfiguration,
          new ApplePayConfiguration(certificate));
```

Table 1.6. Usage of Configure method with Apple Pay

```
LoggingConfiguration loggingConfiguration = new LoggingConfiguration(
   YourServiceCollection,
   @"YourSerilogConfigJsonPath",
   "YourApplicationName");

var serviceProvider = SdkConfiguration
   .Configure(@"YourSdkConfigurationJsonFilePath",
          loggingConfiguration
```

Table 1.7. Usage of Configure method without Apple Pay

Step 4: Apple Pay configuration

For Apple Pay Configuration, you need to create a folder called *Configuration* which contains a .txt file. The file should be called "*ApplePayJavascriptTemplate*" and contains the JavaScript Template to retrieve the merchant session. This file is used in the method *GetJavaScriptForApplePayIntegration* provided by the SDK.

Also, for Apple Pay make sure you use ssl protocol ***Tls12*** on the machine on which SDK is running.

```
(function($) {
    'use strict';

    /**
     * All of the code for your checkout functionality placed here.
     * should reside in this file.
     */
    var debug = false;
    $(document).ready(function() {
        if (window.ApplePaySession) {
            if (ApplePaySession.canMakePayments) {
                setTimeout(function() {
                    $('.apple_pay_option').removeClass('hide-me')
                }, 2000);
            }
        }
    });
```

```
    function initApplePayment(evt) {
        var runningAmount = 5;
        var runningPP = parseFloat(0);
        var runningTotal = function() {
            return parseFloat(runningAmount + runningPP).toFixed(2);
        }
        var shippingOption = "";

        var cart_array = [];
        var x = 0;
        var subtotal = 5;
        var tax_total = 1;
        var shipping_total = 2;
        var discount_total = 3;
        var supported_networks = [SupportedNetworks];

        cart_array[x++] = {
            type: 'final',
            label: 'Subtotal',
            amount: parseFloat(subtotal).toFixed(2)
        };
        cart_array[x++] = {
            type: 'final',
            label: 'Shipping fees',
            amount: parseFloat(shipping_total).toFixed(2)
        };
        if (parseFloat(discount_total) >= 1) {
            cart_array[x++] = {
                type: 'final',
                label: 'Discount',
                amount: parseFloat(discount_total).toFixed(2)
            };
        }
        cart_array[x++] = {
            type: 'final',
            label: 'Tax',
            amount: parseFloat(tax_total).toFixed(2)
        };

        shippingOption = [{
            label: 'Standard Shipping',
            amount: getShippingCosts('domestic_std', true),
            detail: '3-5 days',
            identifier: 'domestic_std'
        }, {
            label: 'Expedited Shipping',
            amount: getShippingCosts('domestic_exp', false),
            detail: '1-3 days',
            identifier: 'domestic_exp'
        }];

        function getShippingCosts(shippingIdentifier, updateRunningPP) {
            var shippingCost = 0;

            switch (shippingIdentifier) {
                case 'domestic_std':
                    shippingCost = 0;
                    break;
                case 'domestic_exp':
```

```
                shippingCost = 0;
                break;
            case 'international':
                shippingCost = 0;
                break;
            default:
                shippingCost = 0;
        }

        if (updateRunningPP == true) {
            runningPP = shippingCost;
        }

        return shippingCost;

    }
    var paymentRequest = {
        currencyCode: '[CurrencyCode]',
        countryCode: '[CountryCode]',
        //requiredShippingContactFields: ['postalAddress'],
        lineItems: cart_array,
        total: {
            label: '[DisplayName]',
            amount: runningTotal()
        },
        supportedNetworks: supported_networks,
        [SupportedCountries]
        merchantCapabilities: ['supports3DS']
    };

    var session = new ApplePaySession(5, paymentRequest);

    // Merchant Validation
    session.onvalidatemerchant = function(event) {
        var promise = performValidation(event.validationURL);
        promise.then(
            function(merchantSession) {
                session.completeMerchantValidation(merchantSession);
            }
        );
    }

    function performValidation(apple_url) {
        return new Promise(
            function(resolve, reject) {
                $.ajax({
                    url: '[AjaxSessionValidationUrl]',
                    type: 'POST',
                    data: {
                        url: apple_url
                    },
                    success: function(data) {
                        if (!data) {
                            reject;
                        } else {
                            resolve(data);
                        }
                    },
                    error: function() {
```

```
                            reject;
                        }
                    })
                }
            );
        }

        session.onpaymentauthorized = function(event) {
            var promise = sendPaymentToken(event.payment.token);
            promise.then(
                function(success) {
                    var status;
                    if (success) {
                        document.getElementById("applePay").style.display =
"none";
                        status = ApplePaySession.STATUS_SUCCESS;
                        sendPaymentToAps(event.payment.token);
                    } else {
                        status = ApplePaySession.STATUS_FAILURE;
                    }

                    session.completePayment(status);
                }
            );
        }

        function sendPaymentToken(paymentToken) {
            return new Promise(
                function(resolve, reject) {
                    resolve(true);
                }
            );
        }

        function sendPaymentToAps(inputData) {
            $.ajax({
                url: '[AjaxCommandUrl]',
                type: 'POST',
                data: {"data":inputData},
                success: function(data) {
                    if (!data) {
                        reject;
                    } else {
                        data = JSON.parse(data);
                        resolve(data);
                    }
                },
                error: function() {
                    reject;
                }
            })
        }

        session.oncancel = function(event) {
            //collect the data if you want to track the ApplePay events
        }

        session.begin();
    }
```

```
    $(document.body).on(
        'click',
        '#applePay',
        function(evt) {
            initApplePayment(evt);
        }
    );

})(jQuery);
```

3. Payment flow

We can see a high-level diagram with the systems that are involved in the payment flow:

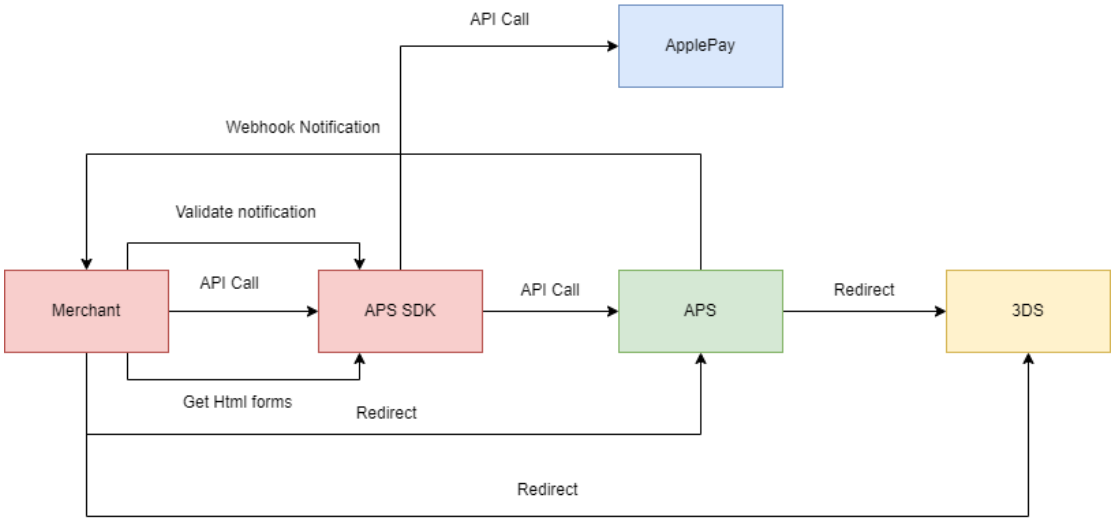

Figure 1.8. Payment flow diagram

In the diagram (Figure 1.8.), the Merchant will call the SDK in the following situation:

- Make an API call to the APS through the APS SDK for completing the payment process.
- Get the HTML Forms necessary to initiate one payment integration (redirect, standard iframe checkout or custom checkout).
- Validate the webhook notification and signature sent by the payment gateway.

4. Maintenance Operations

All maintenance operations provided by the .NET SDK can be found in IMaintenanceOperations interface.

4.1. Capture

An operation that allows the Merchant to capture the authorized amount of a payment. For more details, regarding parameters refer Capture Operation

```
var captureRequestCommand = new CaptureRequestCommand()
{
   MerchantReference = "YourMerchantReference",
   Amount = YourAmount,
   Currency = "YourCurrency",
   Language = "YourLanguage"
};

MaintenanceOperations operations = new();
var response = await operations.CaptureAsync(captureRequestCommand);
```

Table 4.1. Capture operation

4.2 Refund

An operation that returns the entire amount of a transaction, or returns part of it, after a successful capture operation is done. For more details regarding parameters refer: Refund Operation

```
var refundRequestCommand= new RefundRequestCommand()
{
   MerchantReference = "YourMerchantReference",
   Amount = YourAmount,
   Currency = "YourCurrency",
   Language = "YourLanguage"
};

MaintenanceOperations operations = new();
var response = await operations.RefundAsync(refundRequestCommand);
```

Table 4.2. Refund operation

4.3. Void

An operation that allows you to cancel the authorized amount after you have sent a successful authorize request. For more details, Void Operation.

```
var voidRequestCommand = new VoidRequestCommand()
{
   MerchantReference = "YourMerchantReference",
   Language = "YourLanguage"
};

MaintenanceOperations operations = new();
var response = await operations.VoidAsync(voidRequestCommand);
```

Table 4.3. Void operation

4.4. Check Status

In case you need to verify the status of a transaction in progress you can do by using the Check Status method. For more details, Check Status Operation

```
var checkStatusRequestCommand = new CheckStatusRequestCommand()
{
    MerchantReference = "YourMerchantReference",
    Language = "YourLanguage"
};

MaintenanceOperations operations = new();
var response = await operations.CheckStatusAsync(checkStatusRequestCommand);
```

Table 4.4. Check Status operation

5. Integration Channels

5.1. Redirect

The method which returns the form post which needs to be added in the html page, is called ***GetHtmlForRedirectIntegration.*** This method is found in the IHtmlProvider interface. This method can be used for Authorization or for Purchase command. For example, see the code (Table 5.1.).

```
HtmlProvider provider = new();
AuthorizeRequestCommand command = new()
{
   MerchantReference = "YourMerchantReference",
   Language = "YourLanguage",
   Amount = YourAmount,
   Currency = "YourCurrency",
   CustomerEmail = "YourCustomerEmail",
   ReturnUrl = @"YourReturnUrl",
   Description = "YourOrderDescription",
   CustomerName = "YourCustomerName"
};

var redirectFormPost = provider.GetHtmlForRedirectIntegration(command);
```

Table 5.1. Get Html form post for redirect

In this example, we have created an instance of HtmlProvider and AuthorizeRequestCommand object. We call the method with the command, and we get a string which contains the HTML for form post.

If the return URL is not set in the back office, you need to send it in the request command.

5.1.2. Installments

To work with installments, you must send the installments property in the request command, besides the other properties (Table 5.2.). If the return URL is not set in the back office, you need to send it in the request command. Other installment types other than standalone.

```
HtmlProvider provider = new();
PurchaseRequestCommand command = new()
{
   MerchantReference = "YourMerchantReference",
   Language = "YourLanguage",
   Amount = YourAmount,
   Currency = "YourCurrency",
   CustomerEmail = "YourCustomerEmail",
   ReturnUrl = @"YourReturnUrl",
   Description = "YourOrderDescription",
   CustomerName = "YourCustomerName",
   Installments = "STANDALONE",
};

var redirectFormPost = provider.GetHtmlForRedirectIntegration(command);
```

Table 5.2. Get Html form post for redirect with installments

5.2. Standard Checkout

5.2.1. Tokenization and payment

For standard checkout, do the following:

1. Get the tokenization form post with iframe. The .NET SDK provides a method called ***GetHtmlTokenizationForStandardIframeIntegration.*** This method returns a string which contains the form post with an iframe. This method is found in the IHtmlProvider interface.

```
TokenizationRequestCommand command = new()
{
    MerchantReference = "YourMerchantReference",
    Language = "YourLanguage",
    ReturnUrl = @"YourReturnUrl",
};

var redirectFormPost = _htmlProvider
    .GetHtmlTokenizationForStandardIframeIntegration(command);
```

Table 5.3. Get Html form post for standard checkout with iframe

2. After the tokenization step, when you receive the response, you need to validate it, to see if everything is as expected. To do that, you can use the method ***ValidateRequest*** which is part of INotificationValidator interface.

3. If the result is valid, you can continue and make an operation. From the response you need to get the created token and send it when you make the request. Below is an example of the operation purchase (Table 5.4.).

```
var request = ValidateRequest(out var result);

if (result.IsValid)
{
    var purchaseCommand = new PurchaseRequestCommand()
    {
        MerchantReference = "YourMerchantReference",
        Amount = YourAmount,
        Currency = "YourCurrency",
        TokenName = result.RequestData["tokenName"],
        CustomerEmail = "YourCustomerEmail",
        Language = "YourLanguage",
        ReturnUrl = @"YourReturnUrl",
        PhoneNumber = YourCustomerPhoneNumber",
        Description = "YourDescription",
        CustomerName = "YourCustomerName"
    };

    MaintenanceOperations operations = new();
    var response = await operations.PurchaseAsync(purchaseCommand);
}
```

Table 5.4. Operation Purchase on standard checkout

4. To be able to close the opened iframe from standard checkout, check chapter 8, 3DS Modal. There is a method which returns different codes which helps you to close the iframe.

5.2.2. Installments

In tokenization part for installments, you should send mandatory properties like currency, amount, and installments, besides the other properties needed for tokenization. See the following example for tokenization with installments.

```
TokenizationRequestCommand command = new()
{
    MerchantReference = "YourMerchantReference",
    Language = "YourLanguage",
    ReturnUrl = @"YourReturnUrl",
    Installments = "STANDALONE",
    Amount = YourAmount,
    Currency = "YourCurrency",
};

var redirectFormPost = _htmlProvider
    .GetHtmlTokenizationForStandardIframeIntegration(command);
```

Table 5.3. Get Html form post for standard checkout installments with iframe

For installments you need to send the following properties: Installments, PlanCode, IssuerCode. The last two properties you can get them from the tokenization response. Installments property should be set to "YES" in this case (Table 5.5.). For installments, also, you need the validation of the request using the method "***ValidateRequest***" from INotificationValidator interface.

```
var request = ValidateRequest(out var result);

if (result.IsValid)
{
    var purchaseCommand = new PurchaseRequestCommand()
    {
        MerchantReference = "YourMerchantReference",
        Amount = YourAmount,
        Currency = "YourCurrency",
        TokenName = result.RequestData["tokenName"],
        CustomerEmail = "YourCustomerEmail",
        Language = "YourLanguage",
        ReturnUrl = @"YourReturnUrl",
        Installments = "YES",
        PlanCode = result.RequestData["plan_code"],
        IssuerCode = result.RequestData["issuer_code"],
    };

    MaintenanceOperations operations = new();
    var response = await operations.PurchaseAsync(purchaseCommand);
}
```

Table 5.5. Operation Purchase on standard checkout with installments

5.3. Custom Checkout

5.3.1. Tokenization

For custom checkout the first step is to get the tokenization form post. The .NET SDK provides a method called ***GetHtmlTokenizationForCustomIntegration.*** This method is found in the IHtmlProvider interface. This method returns a string which contains the form post with fields for credit card details displayed. This fields can be customized after receiving the form post.

```
TokenizationRequestCommand command = new()
{
    MerchantReference = "YourMerchantReference",
    Language = "YourLanguage",
    ReturnUrl = @"YourReturnUrl",
};

var redirectFormPost = _htmlProvider
```

```
    .GetHtmlTokenizationForCustomIntegration(command);
```

Table 5.6. Get Html form post for custom checkout

After the tokenization step, when you receive the response, you need to validate it, to see if everything is as expected. To do that, you can use the method ValidateRequest which is part of INotificationValidator interface. If the result is valid, you can continue and make an operation. From the response you need to get the created token and send it when you make the request. In the following image is an example of the operation purchase (Table 5.7.).

```
var request = ValidateRequest(out var result);

if (result.IsValid)
{
   var authorizeCommand = new AuthorizeRequestCommand()
   {
      MerchantReference = "YourMerchantReference",
      Amount = YourAmount,
      Currency = "YourCurrency",
      TokenName = result.RequestData["tokenName"],
      CustomerEmail = "YourCustomerEmail",
      Language = "YourLanguage",
      ReturnUrl = @"YourReturnUrl",
      PhoneNumber = YourCustomerPhoneNumber",
      Description = "YourDescription",
      CustomerIp = "172.254.12.5",
      CustomerName = "YourCustomerName"
   };

   MaintenanceOperations operations = new();
   var response = await operations.AuthorizeAsync(authorizeCommand);
}
```

Table 5.7. Operation Authorize on custom checkout

5.3.2. Installments

First step, to work with installments on custom checkout is to call the method ***GetInstallmentsPlansAsync*** which is defined in ***IInstallmentsProviderInterface***. You need to create an instance of ***GetInstallmentsRequestCommand*** with the properties you want to (Table 5.8.).

```
InstallmentsProvider installmentsProvider = new();
GetInstallmentsRequestCommand installmentsRequestCommand = new()
{
   Amount = 243000,
   Currency = "AED",
   Language = "en",
};
var installmentsPlans = await installmentsProvider
    .GetInstallmentsPlansAsync(installmentsRequestCommand);

TokenizationRequestCommand command = new()
{
   MerchantReference = "YourMerchantReference",
   Language = "YourLanguage",
   ReturnUrl = @"YourReturnUrl"
};

var redirectFormPost = _htmlProvider
```

```
    .GetHtmlTokenizationForCustomIntegration(command);
```

Table 5.8. GetInstallmentsPlans method and tokenization

You can do this step, in tokenization step. The next step is to create a request command instance and send the following parameters for installments: Installments, IssuerCode, PlanCode. The value of Installments property should be "HOSTED", and the other two parameters you can take them from the installment's plans explained above.

For installments, also, create the validation of the request using the method "ValidateRequest" from INotificationValidator interface.

```
var request = ValidateRequest(out var result);

if (result.IsValid)
{
   var purchaseCommand = new PurchaseRequestCommand()
   {
      MerchantReference = "YourMerchantReference",
      Amount = YourAmount,
      Currency = "YourCurrency",
      TokenName = result.RequestData["tokenName"],
      CustomerEmail = "YourCustomerEmail",
      Language = "YourLanguage",
      ReturnUrl = @"YourReturnUrl",

      Installments = "HOSTED",
      PlanCode = "FromGetInstallmentsPlans",
      IssuerCode = "FromGetInstallmentsPlans"
   };

   MaintenanceOperations operations = new();
   var response = await operations.PurchaseAsync(purchaseCommand);
}
```

Table 5.9. Operation Purchase custom checkout with installments

6 Apple Pay Integration

6.1. JavaScript

To retrieve the merchant session, you need a JavaScript to inject into your html page. The .NET SDK provides you that JavaScript by using the method ***GetJavaScriptForApplePayIntegration*** which is found in IJavaScriptProvider interface. The method returns a string which contains the needed JavaScript to be able to retrieve the merchant session. An example on how to use that method in Table 6.1.

```
IJavascriptProvider provider = new JavascriptProvider();
var javascriptContent = provider.GetJavaScriptForApplePayIntegration(
    "YourAjaxSessionValidationUrl",
    "YourAjaxCommandUrl",
    "YourCountryCode",
    "YourCurrencyCode",
    "YourSupportedNetworkList"
);
```

Table 6.1. GetJavaScriptForApplePayIntegration usage

The above method has the following parameters:

- ajaxSessionValidationUrl – the ajax session validation URL
- ajaxCommandUrl – the ajax command URL
- countryCode – the country code
- currencyCode – the currency code
- IEnumerable<string> supportedNetworks – the supported networks
- IEnumerable<string> supportedCountries – this parameter is optional; if you don't send it, it will not be added in the JavaScript

6.2. Retrieve merchant session and payment

To be able to make a Purchase or Authorize request, you need to retrieve the merchant session. The .NET SDK provides a method called ***RetrieveMerchantSessionAsync*** in IApplePayClient interface. An example, on how to use the method is in Table 6.2.

```
var client = new ApplePayClient();
var result = await client.RetrieveMerchantSessionAsync(url);
```

Table 6.2. RetrieveMerchantSessionAsync usage

After the merchant session is retrieved you can make an authorize or purchase request.

```
public async Task<IActionResult> AuthorizeOrPurchase(ApplePayRequestCommand inputData)
{
  string req = JsonSerializer.Serialize(inputData,
      new JsonSerializerOptions
      {
         DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull
      });

  var purchaseRequestCommand = new PurchaseRequestCommand()
  {
     MerchantReference = "YourMerchantReference",
     Amount = YourAmount,
     Currency = "YourCurrency",
     CustomerEmail = "YourCustomerEmail",
     Language = "YourLanguage",
     ReturnUrl = "YourReturnUrl"
  };

  var maintenanceOperation = new MaintenanceOperations();
```

```
    var response = await maintenanceOperation.PurchaseAsync(purchaseRequestCommand, inputData);

    return this.Json(JsonSerializer.Serialize(response,
        new JsonSerializerOptions
        {
            DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull
        }));
}
```

Table 6.3. Purchase request on Apple Pay

7.Trusted Channels

7.1. MOTO

MOTO channel enables you to process a range of transactions that do not follow the standard online shopping workflow. For example, your customer may want to pay you offline. By sending an order in the post, by calling you, or indeed in a face-to-face transaction. You can process offline transactions using the MOTO channel.

Note that the MOTO (Mobile Order/ Telephone Order) channel allows you to process MOTO transactions through the Amazon Payment Services API only if you have already established a token for your customer's payment card.

In the table below, Table 7.1. we have created an example of a request with Eci property set to MOTO when creating a AuthorizeRequestCommand object. You need to know the TokenName of the transaction.

```
var authorizeCommand = new AuthorizeRequestCommand()
   {
      MerchantReference = "YourMerchantReference",
      Language = "YourLanguage",
      Amount = YourAmount,
      Currency = "YourCurrency",
      CustomerEmail = "YourCustomerEmail",
      ReturnUrl = @"YourReturnUrl",
      TokenName = result.RequestData["tokenName"],
      Eci = "MOTO"
   };

   MaintenanceOperations operations = new();
   var response = await operations.AuthorizeAsync(authorizeCommand);
```

Table 7.1. MOTO

7.2. Recurring

You can effortlessly configure secure, recurring payments for any defined billing cycle – whether daily, weekly, monthly, or annual. You do so through a workflow that is not much different from the normal checkout process.

In the table below, Table 7.2. we have created an example of a request with Eci property set to RECURRING when creating a PurchaseRequestCommand object. You need to know the TokenName of the transaction.

```
var purchaseCommand = new PurchaseRequestCommand()
   {
      MerchantReference = "YourMerchantReference",
      Language = "YourLanguage",
      Amount = YourAmount,
      Currency = "YourCurrency",
      CustomerEmail = "YourCustomerEmail",
      ReturnUrl = @"YourReturnUrl",
      TokenName = result.RequestData["tokenName"],
      Eci = "RECURRING"
   };

   MaintenanceOperations operations = new();
   var response = await operations.PurchaseAsync(purchaseCommand);
```

Table 7.2. Recurring

7.3. Trusted

If you are a PCI-certified merchant, you can collect your customers' credit card details on your checkout page and store the sensitive payment card data on your server. Read more about PCI compliance here.

PCI-compliant merchants can use the Amazon Payment Services trusted channel to submit payment card details so that Amazon Payment Services can execute transactions using the payment card data. Trusted Channel URLs.

In the table below, Table 7.3. we have created an example of a request with Eci property set to ECOMMERCE when creating a PurchaseRequestCommand object.

```
var purchaseCommand = new PurchaseRequestCommand()
   {
      MerchantReference = "YourMerchantReference",
      Language = "YourLanguage",
      Amount = YourAmount,
      Currency = "YourCurrency",
      CustomerEmail = "YourCustomerEmail",
      CardNumber = "CustomerCardNumber",
      SecurityCode = "CustomerCardSecurityCode",
      ExpiryDate = "CustomerCardExpiryDate",
      CardHolderName = "CustomerCardHolderName",
      ReturnUrl = @"YourReturnUrl",
      Eci = "ECOMMERCE"
   };

   MaintenanceOperations operations = new();
   var response = await operations.PurchaseAsync(purchaseCommand);
```

Table 7.3. Trusted channel

8. 3DS Modal

The .NET SDK provides you code for handle the 3ds secure part. After calling AuthorizeAsync or PurchaseAsync from IMaintenanceOperations interface, you get the response from the gateway. In the response, you can get the 3DS Secure URL.

You are able by use the modal provided by the SDK, by calling the method Handle3dsSecure found in IHtmlProvider interface. The method has 3 parameters:

- secure3dsUrl – if not sent, you will only get the JavaScript to close the iframe used in Standard checkout; if sent, you will get the JavaScript to be redirected to the 3ds without using the modal.
- useModal – if set to true, you will get the code for the modal; if set to false you will get the JavaScript to be redirected to the 3ds without using the modal
- standardCheckout – if set to true, you will get the code to close also the open iframe used in Standard checkout; if set to false, you can do it for custom checkout.

The following pictures provides an example how to use the 3ds modal on standard checkout (Table 8.1.) and custom checkout (Table 8.2.).

```
MaintenanceOperations operations = new();
var response = await operations.PurchaseAsync(purchaseCommand);
url = ResponseHelper.GetRedirectUrl(response);

if (!string.IsNullOrEmpty(response.Secure3dsUrl))
{
   ViewBag.CloseIframe = _htmlProvider.Handle3dsSecure(response.Secure3dsUrl, true, true);

   return View("StandardCheckout", true);
}
else
{
   response.HtmlDecodeProperties().UrlEncodeProperties();
   url = ResponseHelper.GetRedirectUrl(response);
}
```

Table 8.1. Handle 3ds modal on Standard Checkout

```
MaintenanceOperations operations = new();
var response = await operations.PurchaseAsync(purchaseCommand);
url = ResponseHelper.GetRedirectUrl(response);

if (!string.IsNullOrEmpty(response.Secure3dsUrl))
{
   ViewBag.CloseIframe = _htmlProvider.Handle3dsSecure(response.Secure3dsUrl, true);

   return View("CustomCheckout", true);
}
else
{
   response.HtmlDecodeProperties().UrlEncodeProperties();
   url = ResponseHelper.GetRedirectUrl(response);
}
```

Table 8.2. Handle 3ds modal on Custom Checkout

In case when secure 3ds is not present and you have to build the redirect URL, please be sure you have special characters in some property, please make sure to encode them to URL encode. Because in some cases, some special characters are not read as they are. In this case, you may have problems with the method **ValidateRequest**. You can check if the request is correctly by using this method, which validates the parameters, creating the signature and verify if the created signature is equal to the signature which is sent.

9. Webhook notification

To validate the request received from the payment gateway you have a method called ValidateRequest which you can find in INotificationValidator interface. Above on the example we have explained how to use that method.

```
NotificationValidator notificationValidator = new();
NotificationValidationResponse result = notificationValidator.Validate(request);
```

Table 9.1. Validate usage

In case the client does not have any internet connection during the payment transaction to see the payment gateway response you can use the method ValidateAsyncNotification which is also in the INotificationValidator interface.

```
NotificationValidator notificationValidator = new();
NotificationValidationResponse result = notificationValidator.ValidateAsyncNotification(request);
```

Table 9.2. ValidateAsyncNotification usage

The methods return a bool IsValid and a dictionary which contains the request data.