Loyalty app Merchant API documentation

Version 1.1.0

Revision 2

Contents

Introduction

The Loyalty app Merchant API provides a native .NET interface to the Loyalty app system. It requires the .NET Framework 3.5 or above, and JSON.NET.

Staging Environment

The staging environment is a sandboxed testbed, where software development may take place, without worrying about causing any charges to merchants or users.

Developers may interact with the staging environment freely, using it to test and explore the Loyalty app system. Transactions can be created freely, as the staging environment will never charge the developer's account. Points may be added to a staging account using the Stripe test card numbers.

Gaining Access

Access to the staging environment requires your account to be verified.

Sign up to the staging environment by visiting this link:
https://staging.loyaltyapp.com.au/manager/sign_up

You will receive a verification e-mail. Click the verification link to validate your account.

Once your account is validated, Loyalty app must approve the merchant account. Send a support request to https://www.loyaltyapp.com.au/support/, providing your username and e-mail address, and request verification of your staging account. You must specify that this is for the staging environment in your e-mail.

Once your account is approved, log in to the staging website:
https://staging.loyaltyapp.com.au/manager/login

Once you are logged in, you will need to enter banking details. Go to My Business -> Banking Details and add a new banking detail. The staging environment doesn't need valid banking details. You can use the Stripe test card numbers for the credit card. Details here:
https://stripe.com/docs/testing#cards

Once your banking details are complete, you can create your store. Go to My Business -> Store and click the "New Store" button. Enter your store details and click "Create Store".

After these steps are complete, you can add points to your store, and begin using the staging API.

Production Environment

The production environment is a live environment, with live transactions taking place. This is the primary interface that the Loyalty app ecosystem uses.

Do not use the production environment for testing, as any transactions created will move live currency in the Loyalty app system. Charges can be incurred.

The same login details used on the website/mobile app are used with the API.

Loyalty app Ecosystem

The Loyalty app system has three main components: The Loyalty app servers, the merchant API, and the customer mobile app.

The Loyalty app servers provide connectivity to the Loyalty app API, and the customer mobile app. It is responsible for storing, managing, and processing transactions between the merchant API and the customer app.

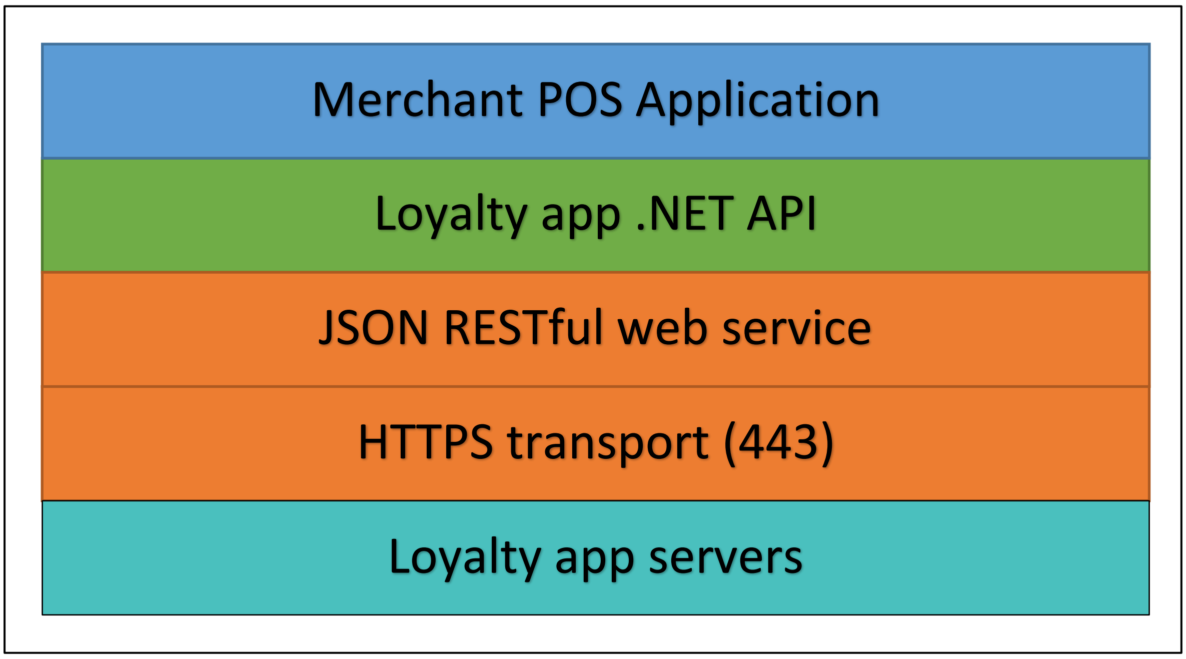
The Loyalty app API provides a strongly typed interface to the Loyalty app servers. It allows the merchant to create, modify and retrieve transactions between the merchant and the customer.

The customer app provides an interface to the Loyalty app servers, for the purpose of making and completing transactions.

Loyalty app Merchant API

The Loyalty app merchant API uses HTTP web methods to communicate with the Loyalty app servers. An active internet connection is required to communicate with the Loyalty app servers.

The network stack for the API is as follows:



Technical Support

Technical support can be obtained through the Loyalty app support portal.

https://www.loyaltyapp.com.au/support/

Example Code

Example .NET code is provided in the API package. Look in the “Samples” folder for code examples.

API Objects Overview

These are the object classes available in the Loyalty app API.

`LoyaltyappApi`: API class.

This class provides access to the Loyalty app API through calls on an instance of this class. Both synchronous and asynchronous calls are provided. This class connects to either the staging environment, or the production environment, depending on a property.

`Transaction`: Transaction entry.

This class may not be directly instantiated. It contains transaction details of any created or queried transactions.

`Filter`: Transaction query filter parameters.

This class allows the user to specify search parameters for transaction queries. By setting the properties of this class, and passing it to one of the `GetTransactions()` functions, the returned transaction list will be limited to records that match these filters.

`LoyaltyappException`: Loyalty app API exception.

This exception type contains information on any errors returned from the Loyalty app system. By examining the properties of this exception, the developer can determine exactly what failed, and why.

API Interface

The API procedures available in the Loyalty app API objects are listed below. These procedures are available from an instance of the **`LoyaltyappApi`** class.

Constructors

A small selection of constructor calls are provided for instantiating the Loyalty app API, and optionally resuming an already-authenticated session.

`New(gDeviceUid)`

> **`gDeviceUid`** – Guid, required. Identifies the current device.

This constructor creates a new, unauthenticated Loyalty app API object. **`gDeviceUid`** is the current device identifier. This identifier should remain static throughout the life of an installation on a single device.

`New(gDeviceUid, strAuthenticationToken, blnUseStagingServer)`

> **`gDeviceUid`** – Guid, required. Identifies the current device.
>
> **`strAuthenticationToken`** – String, required. Provides the authentication token of a previous session to resume.
>
> **`blnUseStagingServer`** – Boolean, required. Indicates which environment the resumed session belongs to.

This constructor creates a new, authenticated Loyalty app API object. This call resumes a previously authenticated session, without needing to open a new session. **`gDeviceUid`** is the current device identifier. This identifier should remain static throughout the life of an installation on a single device. **`strAuthenticationToken`** is the saved authentication token from a previous session, saved from the **`AuthenticationToken`** property. **`blnUseStagingServer`** is a Boolean value indicating which environment the given authentication token is valid for.

Properties

Some properties provide information about the connection and the merchant, while others are used to set information required for logging in.

`UseStagingServer`

> Boolean, read/write

Gets or sets a value indicating whether to use the Loyalty app API development staging server, or the live production server. Use the staging server for development.

`IsSessionOpen`

Boolean, read-only

Returns true if the API class is holding a Loyalty app authentication token. Returns false if no merchant session has been opened.

`DeviceUid`

Guid, read/write

Unique identifying GUID that identifies this device to the Loyalty app system. Every device should have a different **`DeviceUid`** and should remain static throughout the installation on that device.

The **`DeviceUid`** is used during authentication to catch new devices logging in to the system.

`StoreMultiplier`

Decimal, read-only.

The **`StoreMultiplier`** property returns a number used for converting between cents and points.

`AuthenticationToken`

String, read-only.

The **`AuthenticationToken`** property returns the currently held API session token. This value may be saved and used when instantiating the class, in order to resume an existing session.

Procedures

These procedures provide access to the Loyalty app functions.

`int OpenSession(strEmail, strPassword, [strPinCode])`

`strEmail` – String, required. The e-mail address of the merchant account to log in to.

`strPassword` – String, required. The password for the merchant account.

`strPinCode` – String, optional. The PIN code for the merchant account.

This procedure opens a new merchant session on the Loyalty app API. If a session is already open, an **`InvalidOperationException`** is thrown.

This procedure returns the current store balance, in points.

On devices that have not logged in before, a PIN code is required. In this instance, if a PIN code is not supplied, a **`LoyaltyException`** is thrown with the **`ErrorCode`** property set to "`new_device`".

Other **`LoyaltyException.ErrorCode`** values are as follows:

"`not_confirmed`": The merchant has not activated their account.

"`store_unpublished`": The merchant has not published their store on the Loyalty app website.

"`base`": The authentication attempt failed due to incorrect login information supplied.

`void CloseSession()`

This procedure will close any active merchant session, invalidating the privately held session token.

This call will throw an `InvalidOperationException` if no open merchant session exists.

`bool VerifyPinCode(strPinCode)`

`strPinCode`: String, required. The PIN code for the merchant account.

This procedure validates the merchant PIN code. This procedure may be used for extra security during transactions. For instance, the POS integrator may require staff enter the merchant PIN code before performing transactions. This procedure will allow the integrator to do that.

This call will throw an **`InvalidOperationException`** if no open merchant session exists.

`int GetStoreBalance()`

This procedure queries Loyalty app for the current store balance, returning the current store balance to the caller.

This call will throw an **`InvalidOperationException`** if no open merchant session exists.

`Transaction PayWithCash(decAmount)`

`decAmount`: Decimal, required. The dollar amount of the transaction.

This procedure executes a "pay with cash" transaction. This transaction type is used when the customer pays with cash and earns points. The sale dollar total must be passed in the **`decAmount`** parameter.

A Transaction object is returned, describing the transaction generated.

The POS integrator is responsible for polling the Loyalty app API for the status of the transaction, to determine if/when the transaction completes.

This call will throw an **`InvalidOperationException`** if no open merchant session exists.

`Transaction PayWithPoints(decAmount)`

`decAmount`: Decimal, required. The dollar amount of the sale.

This procedure executes a "pay with points" transaction. This transaction type is used when the customer wishes to pay with loyalty points. The sale dollar total must be passed in the **`decAmount`** parameter.

A Transaction object is returned, describing the transaction generated.

The POS integrator is responsible for checking the status of the transaction, if desired. Please see the section on Transaction Status Updates for more information.

This call will throw an **`InvalidOperationException`** if no open merchant session exists.

`Transaction Refund(decAmount, intTransactionID)`

`decAmount`: Decimal, required. The negative dollar amount to refund to the customer.

`intTransactionID`: Integer, required. The TransactionID value of the transaction being refunded.

This procedure executes a "refund" transaction. This transaction type is used to refund loyalty points to the customer when a sale is cancelled or refunded. The sale dollar total must be passed in the **`decAmount`** parameter, and the transaction ID of the original transaction must be passed in the **`intTransactionID`** parameter.

Note that the **`decAmount`** parameter must contain a negative value for the refund.

A Transaction object is returned, describing the transaction generated.

The POS integrator is responsible for polling the Loyalty app API for the status of the transaction, to determine if/when the transaction completes.

This call will throw an **`InvalidOperationException`** if no open merchant session exists.

`void CancelTransaction(intTransactionID)`

`intTransactionID`: Integer, required. The TransactionID value of the transaction being cancelled.

This procedure cancels a transaction created with one of the **`PayWithCash()`**, **`PayWithPoints()`**, or **`Refund()`** procedures.

The transaction is marked as cancelled, and further processing on that transaction is no longer possible.

This call will throw an **InvalidOperationException** if no open merchant session exists.

List<Transaction> GetTransactions()

This procedure queries Loyalty app for a list of all transactions on the merchant account.

This procedure should be used sparingly, as a great many number of **Transaction** objects may be downloaded and returned. Developers are suggested to use the **GetTransactions(fFilter)** overload that allows filtering of results.

This call will throw an **InvalidOperationException** if no open merchant session exists.

List<Transaction> GetTransactions(fFilter)

fFilter: Filter object, required. The search filters to apply to the transactions before downloading.

This procedure is similar to the unfiltered **GetTransactions()** procedure. However, a **Filter** object may be passed in to filter the transactions before being retrieved from the Loyalty app server.

This call will throw an **InvalidOperationException** if no open merchant session exists.

TransactionStatusEnum GetTransactionStatus(tTransaction)

tTransaction: Transaction object, required. The transaction to query for an updated status.

This procedure queries the Loyalty app server for an updated transaction status.

If the transaction is not found, a **LoyaltyappExeption** exception is thrown with the **ErrorCode** property set to "no_transaction".

API Enumerations

A number of enumerations are available in the Loyalty app API. These enumerations describe things like transaction types, transaction statuses, and filter properties.

`TransactionTypeEnum`

This enumeration describes a transaction type.

`PayWithCash`: This transaction is a "pay with cast" transaction, where the customer has paid with cash, and is earning points.

`PayWithPoints`: This transaction is a "pay with points" transaction, where the customer has paid for their purchase using loyalty points.

`Refund`: This transaction is a "refund" transaction, where the merchant has refunded points to the customer.

`TransactionStatusEnum`

This enumeration describes the status of a transaction.

`Pending`: Transaction is currently pending and has not yet completed.

`Complete:` Transaction has completed, and points have been transferred.

`Cancelled`: Transaction has been cancelled and has not been processed.

`TransactionFilterTypeEnum`

This enumeration is used with the **`Filter`** object and specifies the transaction type to request. Only one value may be selected – transaction types cannot be combined in a query.

`Any`: All transactions should be returned. I.e. filter is disabled.

`PayWithCash`: Only transactions of the "pay with cash" type will be returned.

`PayWithPoints`: Only transactions of the "pay with points" type will be returned.

`Refund`: Only transactions of the "refund" type will be returned.

`TransactionFilterStatusEnum`

This enumeration is used with the `Filter` object and specifies the transaction status to request. Only one value may be selected = transaction statuses cannot be combined in a query.

`Any`: All transactions should be returned. I.e. filter is disabled.

`Pending`: Only pending transactions should be returned.

`Complete`: Only completed transactions should be returned.

`Cancelled`: Only cancelled transactions should be returned.

API Objects

These objects are used to hold information about transactions or provide filter parameters when querying transactions.

`Filter`

The **`Filter`** object provides properties for filtering records returned from Loyalty app when querying the server for transactions.

By default, all properties are set to allow all transactions to be queried. By setting any property to anything other than their default value will enable that filter.

`TransactionType`: This property specifies the transaction types to return.

`TransactionStatus`: This property specifies what status of transactions to return.

`StartTimestampUtc`: This property sets the lower UTC time and date to query for. Only transactions on or after this timestamp are returned.

`EndTimestampUtc`: This property sets the upper UTC time and date to query for. Only transactions before this timestamp are returned.

As well as the above properties, some static functions are defined to return useful filters.

`Pending()`: Crafts a **`Filter`** object that returns only pending transactions.

`Complete()`: Crafts a **`Filter`** object that returns only completed transactions.

`Today()`: Crafts a **`Filter`** object that returns all transactions for today's local date.

`ThisWeek()`: Crafts a **`Filter`** object that returns all transactions for the current local week. This function uses the current system culture's "first day of week" configuration to determine the start date.

`ThisWeek(dowFirstDayOfWeek)`: Crafts a **`Filter`** object that returns all transactions for the current local week. Uses the **`dowFirstDayOfWeek`** parameter to determine the start date.

`ThisMonth()`: Crafts a **`Filter`** object that returns all transactions for the current local month.

`Transaction`

The **`Transaction`** object describes a transaction in the Loyalty app system.

`int TransactionID`: Contains the transaction identifier for this transaction.

`TransactionTypeEnum TransactionType`: Describes what kind of transaction this is.

`TransactionStatusEnum TransactionStatus`: Describes the current status of this transaction.

`DateTime TransactionTimestampUtc`: Describes the UTC date and time of when the transaction was created.

`Decimal Amount`: Describes the dollar amount of this transaction. Note that refund transactions have a negative **`Amount`** value.

`string Token`: Nullable. This property contains the QR code that should be shown to the customer to complete the transaction.

`int RefundOfTransactionID`: Default -1. For refund transactions, this holds the **`TransactionID`** of the original transaction that is being/has been refunded.

`int PointsEarned`: Describes the number of points earned in the transaction.

`int PointsSpent`: Describes the number of points spent in the transaction.

`int TransactionFee`: Describes the transaction fee applied for the transaction. This value is only available if the **`Transaction`** was retrieved from one of the **`GetTransactions(...)`** functions.

`Transaction SampleTransaction`: Static. This static property holds a sample "pay with cash" transaction the developer may use for testing, without needing to generate a real transaction.

Templating System

The Loyalty app templating system allows generation of **`Bitmap`** objects representing a Loyalty app **`Transaction`**. These images can be printed out as a physical voucher or shown on-screen to the customer. This templating system is available from the **`LoyaltyappReceiptLayout`** class.

Receipt layouts use a "box view" model layout engine to position visual elements on the resulting image.

The developer may also embed images in the receipt layout. When an image file is needed, the **`LoyaltyappReceiptLayout`** will fire an event, asking the POS application to retrieve the file bytes. This allows the embedded images to be stored on the file system, in a database, or in embedded application resources.

Events

Only a single event is emitted from the templating system.

`RequestFile(sender, e)`

`object sender`: The object that raised this event.

`RequestFileEventArgs e`: An EventArgs object used to tell the host application what file to return and allows the host application to return the file's bytes.

This event is fired when the templating system needs the contents of a file that was embedded in the receipt layout. The **`FileName`** property of this event args will contain the name of the file requested. The host application should populate the **`Bytes`** property of the event args with the file's contents before returning.

Constructors

The templating system has no public constructors and must be instantiated by one of the static functions provided.

Static Functions

The **`LoyaltyappReceiptLayout`** class provides several static functions for loading a receipt layout template.

`LoyaltyappReceiptLayout Load(strFileName)`

`string strFileName`: The path of the XML file to load.

This static function loads a receipt layout from the local file system.

`LoyaltyappReceiptLayout Load(sStream)`

`Stream sStream`: A Stream object containing the XML layout data to load.

This static function loads a receipt layout from the passed stream.

`LoyaltyappReceiptLayout Load(abytLayout[])`

`Byte[] abytLayout`: A byte array containing the XML layout data to load.

This static function loads a receipt layout from the passed byte array.

`LoyaltyappReceiptLayout Load(xdDocument)`

`XDocument xdDocument`: A XDocument object containing the XML layout data to load.

This static function loads a receipt layout from the passed XDocument object.

`LoyaltyappReceiptLayout LoadFromString(strLayout)`

`string strLayout`: A string containing the XML layout data to load.

This static function loads a receipt layout from the passed XML string.

`LoyaltyappReceiptLayout Load(xeElement)`

`XElement xeElement`: An XElement object containing the XML layout data to load.

This static function loads a receipt layout from the contents of the passed XElement object.

`Dictionary<string, object> GetDataFromTransaction(tTransaction, strStoreName)`

`Transaction tTransaction`: The Transaction object to use to build the data dictionary.

`string strStoreName`: The name of the store to add to the data dictionary.

This static function builds a dictionary data source from the passed **`Transaction`** and store name. This dictionary is suitable for passing into the **`Render(sngWidth, dicSource)`** member function.

Properties

There is only a single property exposed by the **`LoyaltyappReceiptLayout`** class.

`bool DebugView`

Boolean, read/write

This property enables and disables visual debug output. When enabled, the templating system will colour the background of visual elements to show the area they occupy.

Procedures

The templating system exposes a single procedure for the host application to call.

`Bitmap Render(sngWidth, dicSource)`

`float sngWidth`: The width of the generated receipt image.

`Dictionary<string, object> dicSource`: The data to display on the receipt.

This procedure generates a receipt image of the defined with, using the data from the passed dictionary. This procedure will produce a 24-bpp image of fixed width, and variable length.

The **`sngWidth`** value will be rounded up to create the output bitmap. However, this value will be respected as-is for layout functions, allowing layout to sub-pixel accuracy.

Any additional data needed by your receipt layout may be added to the **`dicSource`** dictionary before being passed into this function.

Transaction Procedure

Transactions occur over several steps. A standard transaction process is described as:

1. The merchant begins a transaction, obtaining a **`Transaction`** object.
2. The merchant generates a QR code from the **`Transaction`** object's **`Token`** property, presenting this code to the customer for scanning.
3. Optionally, the merchant waits for a status update request from the POS operator, calling **`GetTransactionStatus(tTransaction)`** to obtain an updated transaction status. When the transaction completes, the merchant continues with the tender process.
4. Meanwhile, the customer scans the generated QR code with their Loyalty app mobile app and accepts the transaction. This marks the transaction as complete, and the next poll from the merchant will reflect this.

In detail, a transaction should follow this procedure.

Step 1 – Merchant begins transaction.

The merchant will call **`PayWithCash()`**, **`PayWithPoints()`**, or **`Refund()`**. These procedures will return a **`Transaction`** object.

The merchant keeps the **`Transaction`** object in memory. This object will be needed later to poll for completion of the transaction.

Step 2 – Merchant generates a QR code.

The merchant will generate a QR code from the **`Token`** property of the **`Transaction`** object returned in step 1.

This QR code will be given to the customer, either displayed on-screen, or printed out, for the customer to scan using their Loyalty app mobile app.

Step 3 – Optional: Merchant waits for transaction completion.

The POS application should display an interface to the POS operator, allowing the operator to request a transaction status update.

Once the operator observes the customer completing the transaction, they use the POS interface to request a transaction status update.

The POS calls **`GetTransactionStatus(tTransaction)`**, passing in the **`Transaction`** object generated in step 1. This procedure returns an updated status for the transaction.

If the transaction status is **`Complete`**, the merchant may continue with a successful tender. If the transaction is still **`Pending`**, then the POS interface should inform the operator that the transaction is not yet complete and should wait for another status update request from the operator.

If the transaction status is **`Cancelled`**, then the transaction should be considered in error, and an error returned to the POS.

If the operator cancels the tender, the transaction must be marked as cancelled, using the **`CancelTransaction(intTransactionID)`** procedure.

This step is entirely optional and may be ignored if the merchant does not need to monitor the transaction status. For “pay with points” and “refund” transactions, it is recommended that the merchant include this step.

Step 4 – Customer scans QR code.

The customer must scan the QR code generated by the POS system, to complete the transaction. The customer must then accept the transaction.

Once the QR code is scanned, and the transaction accepted, the transaction will be marked as complete.

On the next polling request, the POS will find the transaction marked as complete, and may proceed with the tender process.

Obtaining Transaction Status Updates

There are three transaction status scenarios supported by the Loyalty app API. “Fire-and-forget”, where status updates are not required, “triggered poll”, where the POS operator triggers a status update check, and “looping poll”, where the POS application periodically polls for status updates.

It is recommended to use the “fire-and-forget” scenario, or the “triggered poll” scenario, depending on the type of transaction being performed.

To keep server load low, using the “looping poll” scenario is not recommended.

Fire-and-Forget Scenario

In this scenario, a transaction is started, and a QR code is presented to the customer. The POS system can then continue with regular operations, without monitoring the transaction status.

It is the merchant’s responsibility to make sure the QR code is scanned, and the transaction completes.

This scenario can be used with the “pay with cash” transaction type, or the “refund” transaction type, although it is not the recommended method to use for refunds.

Triggered Poll Scenario

In this scenario, a transaction is started, the QR code is presented to the customer, and the POS application waits for confirmation from the POS operator that the transaction has occurred. The POS then queries the Loyalty app API for the status of the transaction, making sure the transaction is in the completed phase.

This is the recommended method for “pay with points” transactions, and “refund” transactions.

Looping Poll Scenario

In this scenario, a transaction is started, the QR code is presented to the customer, and the POS enters a polling loop, periodically checking the status of the transaction.

It is important to not poll the transaction too quickly, as each poll request generates a new call to the Loyalty app servers. Polling too quickly will cause excessive load on the servers and will degrade performance for all users of the system. A good rule of thumb is to delay at least 5 seconds between requests, and only poll while waiting for a transaction to complete.

Using this scenario is supported, but not recommended.

Generating Vouchers

Vouchers can be generated from a Loyalty app API **`Transaction`** object. Once a transaction has been created, the host app may create a data dictionary from the transaction. The developer may then create a L**`oyaltyappReceiptLayout`** object, and call the **`Render(sngWidth, dicSource)`** function to generate a receipt bitmap for printing or display.

The usual procedure is as follows:

1. The host app creates a new Loyalty app transaction.
2. The host app passes the new transaction to the **`LoyaltyAppReceiptLayout.GetDataFromTransaction(tTransaction, strStoreName)`** static procedure, which will return a Dictionary<string, object> object containing the transaction data.
3. The host app loads a receipt layout, using one of the static loader functions on the **`LoyaltyappReceiptLayout`** class.
4. The host app calls the **`Render(sngWidth, dicSource)`** instance function, and receives a **`Bitmap`** object back.
5. The host app either displays this bitmap to the operator/customer or prints this bitmap to a receipt/document printer.