vindicia
Build Subscription Revenue

CashBox®
Programming Guide

CashBox 5.0
February, 2014

March 1, 2014

Table of Contents

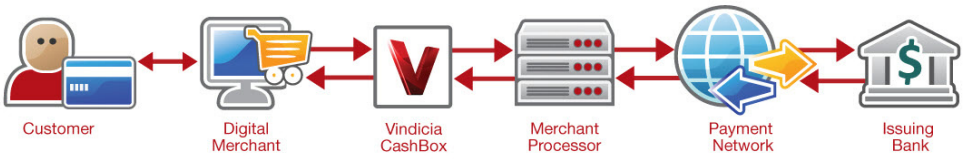
CashBox® Programming Guide Preface

CashBox is an on-demand solution for recurring and one-time billing, available for integration with your application through an object-oriented application programming interface (API), based on the Simple Object Application Protocol (SOAP). This manual, the *CashBox Programming Guide*, leads you through the process of integrating your application with the CashBox and ChargeGuard services offered by Vindicia.

Vindicia's CashBox API allows you to both integrate with CashBox, and take advantage of Vindicia's ChargeGuard protection services.

About CashBox

CashBox delivers an on-demand billing solution for both subscription and one-time payment offerings that maximizes online revenue through managed customer retention and extended customer life cycles. The CashBox API enables you to seamlessly integrate Vindicia's services into your online applications.



CashBox Workflow

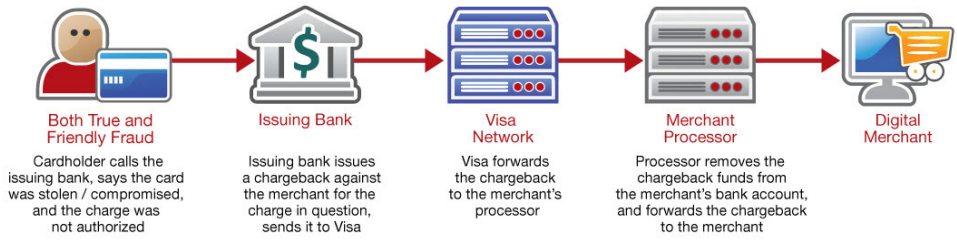
Using the CashBox API, you can create sophisticated and automatic billing services for your products, and pass credit-card, electronic-check, PayPal, and other payment method transactions to CashBox for processing. CashBox also ensures security and compliance with rules and regulations, and supports token or virtual-currency purchases, such as frequent flyer miles, usage minutes, and incentive points.

Relationship between the CashBox API and Merchant Website

API calls are automatically converted to Secure Sockets Layer (SSL)-based Simple Object Access Protocol (SOAP) calls to Vindicia's servers for processing and record-keeping. CashBox returns the transaction results as SOAP calls, and translates the results into return values for processing in your application.

About ChargeGuard

Fraudulent chargebacks present a unique and different problem because they distract you from your core business, limit marketing initiatives, and drain your customer-service resources. For online merchants, systematically challenging friendly-fraud chargebacks (see www.vindicia.com/products/fraud_management/chargeback_management.html) is an important component for building online revenue. Often, credit-card networks end up protecting fraudsters as you lose revenue and incur higher transaction and operational costs.

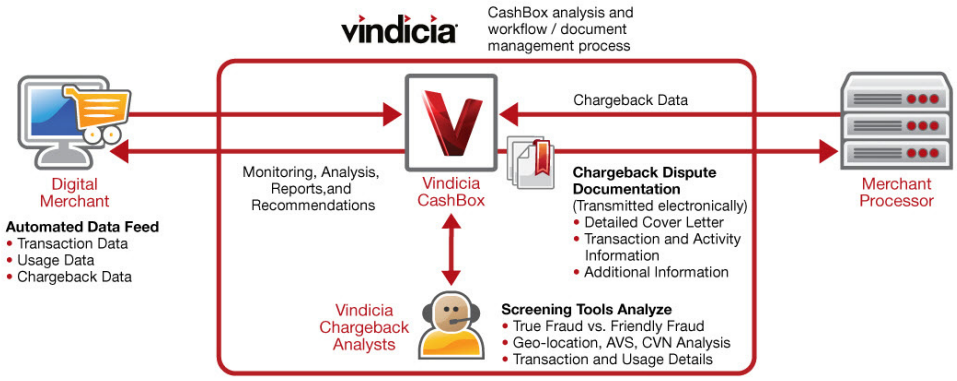


True and Friendly Fraud Process

As a subset of CashBox, ChargeGuard delivers an overall approach to managing chargebacks by becoming your dispute agent in gaining recognition of legitimate transactions. In essence, ChargeGuard minimizes chargeback-related revenue loss by securely leveraging fraud risk screening capabilities to prevent chargebacks of completed transactions. If chargebacks occur, ChargeGuard help you to successfully dispute them.

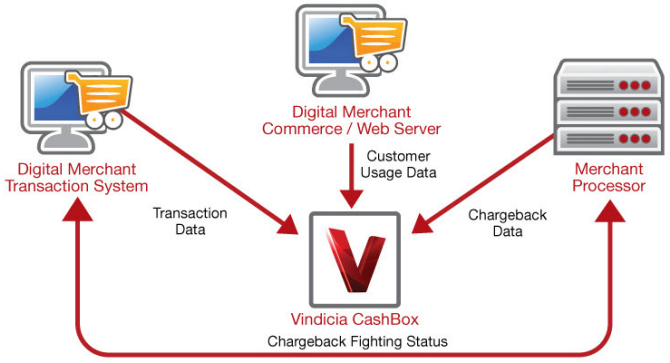
Chargeback Dispute Process

Vindicia first assesses the information reported by your customer and by you. If the chargeback is legitimate, Vindicia leaves it as is. However, if it appears to be fraudulent, Vindicia disputes it on your behalf.



Chargeback Dispute Process

Gathering as much information as possible about the original transaction, the customer, and the customer's previous and current use activity history, can make a significant difference in the success of chargeback challenges. ChargeGuard classifies, collects, and assembles four types of data: transaction data, customer usage data, chargeback data, and chargeback fighting status.



ChargeGuard Data Types

CashBox automatically sends transaction and customer use data to Vindicia. The CashBox API also allows you to retrieve chargebacks and their latest status from the Vindicia servers.

Fraud Risk Screening

The Vindicia fraud risk screening features are integrated into ChargeGuard and CashBox. Fraud risk screening analyzes and scores risk factors for online transactions in real time, so that you can process customer orders more efficiently with little or no manual intervention. Risk screening helps reduce fraud risk with minimal intervention from your organization, and minimizes false positives (incorrectly rejected valid transactions).

Vindicia's risk screening system has the added benefit of cross-referencing data across a worldwide network of merchants. When a user issues a chargeback or commits fraud at one of Vindicia's merchant sites, that information feeds back into the risk screening system for all merchants. Because Vindicia's risk screening solution is informed by a database of chargebacks processed across the Vindicia merchant network, the system becomes more predictive everyday in identifying potentially fraudulent transactions.

Fraud risk screening offers the following key features:

- **IP Geolocation control**, which enables you to pinpoint your customer's physical location, and removes some of the anonymity offered by the Internet. The locator can highlight, for example, the distance between the billing address and the location where the order was placed to help determine transaction risk, especially as it applies to specific countries.
- **Proxy detection**, which determines whether an IP address is an anonymous or open proxy. Both types of proxies are commonly used to mask the original IP address, thus bypassing IP geolocation controls.
- **Validation of Bank Identification Number (BIN),** which checks the first six digits of the credit-card number for information on the issuing bank. This feature verifies whether the country of the billing address matches that of the issuing bank.
- **Analysis of email addresses,** which determines whether the customer's email address is from a free email provider, and if that email address has been associated with high-risk or fraudulent transactions.
- **Creation of true-probability profiles**, which validates the risk screen against the body of chargebacks that Vindicia processes across its database to help you determine whether to accept or reject the transaction.
- **A broad database that houses data on businesses that subscribe to the Vindicia service,** which enables them to indirectly collaborate and share non-personally identifiable information through Vindicia for mutual protection. This network effect significantly contributes to the detection of fraudulent orders and probable chargebacks.

1 CashBox Client Library Setup

The CashBox API is composed of objects accessed through a SOAP interface. Integrate with the Vindicia service by making SOAP calls supported by the objects. The CashBox client library allows you to send SOAP requests to Vindicia without writing code at the SOAP level, because the library wraps around the SOAP objects. Typically, each SOAP object directly translates into a corresponding language-specific object.

This chapter introduces the CashBox API objects and describes the procedure to configure the CashBox client library.

1.1 CashBox API

The following table lists and summarizes the CashBox API objects.

Table 1-1 Summary of CashBox API Objects

CashBox API Object	Description
`Account`	Encapsulates a customer account.
`Activity`	Records a nontransaction (nonpurchase) activity on your site.
`Address`	Records a customer's address.
`AutoBill`	Describes the terms of a customer's relationship to a product or service and a Billing Plan.
`BillingPlan`	Describes a billing plan which defines how charges are made over time.
`Campaign`	Describes the parameters of a sales Campaign.
`Chargeback`	Details the chargeback information for a customer account. (Works in conjunction with ChargeGuard.)
`Entitlement`	Details the status of a customer's current entitlement to your product or service.
`GiftCard`	Encapsulates details of a gift card redeemed or to be redeemed through CashBox.
`NameValuePair`	Referenced by several CashBox objects, the `NameValuePair` object is used to hold attributes not otherwise supported in the object.
`PaymentMethod`	Details a customer's payment method, such as Credit Card, PayPal, or Direct Debit.
`PaymentProvider`	Se rv es as a wrapper to contain static information required by a payment provider for payment processing.
`Product`	Describes a product or service that you offer.
`RatePlan`	Defines the logic by which the pricing structure for Rated Products will be determined.
`Refund`	Describes a refund on a transaction or account.
`SeasonSet`	Defines a group of time intervals, which may be used with Billing Plans to define both Billing Cycles, and Entitlement grants.
`Token`	Describes a customer account's non-currency balance, such as virtual currency, frequent flier miles, downloads, or storage space.
`Transaction`	Handles the transactions, generated through CashBox, that relate to a customer account. (With ChargeGuard integration, be certain to report transactions to Vindicia.)
`WebSession`	Tracks your Web order form's submission activity in the context of the Hosted Order Automation (HOA) capacity.

The CashBox API Guide describes the objects in detail. These objects include data members, and methods that operate on the data members. Write to the CashBox API with PHP, Perl, Java, .NET with C#, or C++ by using the CashBox client libraries. You may also implement a native library for other environments with the CashBox Web Services Description Language (WSDL) files.

Note	Because Vindicia supports multiple programming languages, the descriptions and examples of the CashBox objects, data members, and methods are in generic pseudo-code. Translate this pseudo-code to your programming language of choice.

1.2 Support for Development

Before using CashBox, collect your customer requests, package the data, send it to Vindicia through the CashBox API, and receive, store, and report the return data from CashBox for your needs, as appropriate. The API, a library or package for PHP, Visual Basic (VB), C++, Active Server Pages (ASP), Perl, Java, and .NET with C#, makes it simple and secure for you to hand off data to CashBox for processing.

Be sure to also read the ***CashBox Portal User's Guide***. Because most of the data created and processed by the CashBox API is displayed on the CashBox Portal, the Portal may be useful for debugging during your development process. The CashBox Portal also allows you to create most CashBox objects, and may be used in conjunction with the API throughout your deployment and maintenance cycle.

1.2.1 Installing and Configuring the CashBox Library

Vindicia provides libraries specific to several development environments. To build applications that employ CashBox, first set up the library files, as described in the following subsections.

PHP

The CashBox client library for PHP contains a ZIP file with several PHP files in the directory `Vindicia/Soap`. Two of those files, `Vindicia.php` and `Const.php`, are included in your source code.

Install the PHP client library:

1. Extract the zip archive, and install it into the proper location for operating system and PHP engine and distribution.
2. Type:
```
require_once('Vindicia/Soap/Vindicia.php');
require_once('Vindicia/Soap/const.php');
```

Note: You must add these `require_once` statements to every PHP source file which references the CashBox API in any way.

3. Double check that you have downloaded the correct version for your site, and that the extraction of the archive was completed successfully.

VB, C++, and ASP

Install VB, C++, and ASP:

1. Install the client library.

   This client library is shipped as a Windows Microsoft Installer (MSI) file. When you run this file, it installs the dynamic link library (DLL) in the folder specified. The default location is

   `C:\Program Files\Vindicia\CashBoxAPI`***version***.

2. After you have installed the client library, register the object on the server.

   In a command window, go to the library's installation location and type:

   ```
   regsvr32 VindiciaCOM.dll
   ```

   `regsvr32` then displays a message that the registration is successful.

3. Once you have installed the files, access the objects by referencing `VindiciaCOM` when creating instances of CashBox objects.

   For example, to create a CashBox `Transaction` object in VB:

   ```
   Dim transaction As New VindiciaCOM.CTransaction()
   ```

   To create the same object in VB to program ASP:

   ```
   Set transaction = CreateObject("VindiciaCOM.Transaction")
   ```

Perl

Install the Perl client library on Mac OS X:

1. Install the modules required by the Perl API client. Type:

   ```
   sudo perl -MCPAN -e 'install Crypt::SSLeay'
   sudo perl -MCPAN -e 'install SOAP::Lite'
   ```

2. Install the API client.

   Obtain a current copy and place it in a directory.

   Navigate to that directory in a terminal, and type:

   ```
   sudo perl Makefile.PL
   sudo make
   sudo make install
   ```

The Perl client is now installed on your Mac. The default location is `/Library/Perl/perl-version`, with `Vindicia.pm` and all the other modules in the Vindicia directory.

3. During development, import the Perl module into your source files. Type:

   ```
   use Vindicia;
   use Vindicia::Soap::Vindicia;
   ```

Configuring the Perl API Client

The installation process above creates the file `/etc/vindicia_conf.xml`, a configuration file in XML format. Note that the path name assumes a Mac setup. Path names on Windows vary according to the configuration.

1. Edit the following fields in `vindicia_conf.xml`:
   - `VIN_SOAP_Server`: The server's host name. For example, the name of the `Prodtest` development server is `soap.prodtest.sj.vindicia.com`.
   - `VIN_SOAP_Version`: The version of CashBox you will call, for example, `4.2`.
   - `VIN_Server`: This value must be `0`, which means that this is a Vindicia Perl client, not a server.
   - `VIN_Client_Timeout_Usec`: The client timeout in milliseconds. For example, for 250 seconds, set the value to `250000`.
2. Create a temporary directory on your computer for the client, and then specify the directory's full path as the value for the following fields:

`VIN_Base_Dir`	`VIN_Soap_Cache_Dir`
`VIN_Var_Dir`	`VIN_Client_Var_Dir`
`VIN_Tmp_Dir`	`VIN_Log_Cache`
`VIN_Lock_Dir`	

Specifying the First Parameter in Perl

Certain methods in Perl take a first parameter that is not specified in other languages, because the first parameter is the invoking object, such as in the `fetchCreditHistory` method. For example:

In PHP (and similarly in Java and C#):

```
ab = AutoBillFactory::getObject();
ab->fetchCreditHistory($timestamp, $end_timestamp, $page,
page_size);
```

In Perl:

```
ab = Vindicia::Soap::AutoBill->new(...);
ab->fetchCreditHistory($ab, $timestamp, $end_timestamp, $page,
page_size);
```

In Perl, the first parameter is the `AutoBill` object, but in other languages it is provided using the invoking parameter. This is true with any method where the first parameter is an object of the class in question.

Java

The CashBox client library for Java is in the Java archive file `vindicia.jar`, which bundles the CashBox API and the underlying Apache Axis library for sending and receiving SOAP calls. The release contains two files: vindicia_java_client_***version***.zip and vindicia_java_client_***version***.docs.zip.

To set up the Java client library, unzip vindicia_java_client_***version***.zip and add `vindicia.jar` to your project's `classpath`. Then, import the Vindicia classes to your source file. For example, for CashBox 4.2:

```
import com.vindicia.client.ClassName;
import com.vindicia.soap.v4.2.Vindicia.ClassName;
```

If you will be running the Vindicia Java client library on a machine which can access a URL (such as a SOAP end point) outside of your company's firewall ***only*** through a proxy server, please configure the Java client library to identify the proxy server as follows

```
System.setProperty("https.proxyHost", "web-proxy.mycompany.com");
   // Use the address of your company's proxy server
System.setProperty("https.proxyPort", "8080");
   // Use the HTTPS port supported by your proxy server
```

Caution	Some class names in the packages are identical. The class in the `com.vindicia.client` package is usually a child of a class with an identical class name in the `com.vindicia.soap.Vindicia` package. Vindicia recommends that you import classes with fully qualified package names, and identify the types similarly in your Java code.

.NET With C#

The CashBox client library for .NET is in the DLL file `Vindicia.dll`, which bundles the CashBox API and the underlying SOAP stubs.

To set up the .NET client library, add a reference to the `Vindicia.dll` file in your project. Then, import the Vindicia namespace to your source file:

```
using Vindicia;
```

1.2.2 Setting Up Authentication Parameters

The API calls that you make to the Vindicia servers use SOAP over SSL, which encrypts the data that travels between your servers and Vindicia's, and renders the data tamper-proof en route. To ensure that the calls originate from a legitimate source, CashBox requires that each call include authentication. The `Authentication` object contains a SOAP user name and password, which are provided to you by Vindicia Client Services, and which vary between the production and test environments.

If you generate API calls directly through the CashBox WSDL files without using any client libraries, you must include the `Authentication` object in your calls to Vindicia. By using a Vindicia client library, however, you need not do so. The client library enables you to globally configure the authentication parameters (the SOAP user name and password), automatically construct the `Authentication` object, and pass it in with your calls. Some libraries, such as Java and PHP, also enable you to specify authentication parameters during the process of constructing a target object on which to make calls.

The following subsections describe how authentication parameters operate in the client libraries. The concepts are illustrated through the construction of the commonly used `Transaction` object.

In Perl

Create a Transaction in a Perl program:

Set your Vindicia user name and password variables:

```
$login = "MyVindiciaLogin";
$password = "MyVindiciaPassword";
```

Create a new Transaction:

```
my $tx = Vindicia::Soap::Transaction->
new(auth_login => $login, auth_password => $password);
```

where `MyVindiciaLogin` and `MyVindiciaPassword` are the SOAP user name and password, respectively, assigned to you by Vindicia.

In PHP

In the PHP library, edit the Const.php file in the Vindicia/Soap directory to change the values of the global constants that contain the Vindicia SOAP login and password. Change the values of the following constants:

```
define("VIN_SOAP_CLIENT_USER", "your username here");
define("VIN_SOAP_CLIENT_PASSWORD", "your password here");
```

Then, you can instantiate an object, such as `Transaction`, as follows:

```
$txn = new Transaction();
```

In Java

Create a Transaction in Java:

```
// Create a new transaction
String username = "MyVindiciaLogin";
String password = "MyVindiciaPassword";
com.vindicia.client.Transaction transaction =
   new com.vindicia.client.Transaction(username, password);
```

where `MyVindiciaLogin` and `MyVindiciaPassword` are the SOAP user name and password, respectively, assigned to you by Vindicia.

You may also globally set the SOAP user name and password for making CashBox API calls in the Java library. Define the constants in the `com.vindicia.client.ClientConstants` class as follows:

```
com.vindicia.client.ClientConstants.SOAP_LOGIN = "my_soap_user_name";

com.vindicia.client.ClientConstants.SOAP_PASSWORD = "my_soap_password";
```

If you set the SOAP user name and password globally, use object constructors that do not take those values as their parameters.

In VB, C++, and ASP

In VB, you must create an object and allocate space for it before sending the authentication information.

1. To create a new `Transaction` object, type:

```
Create a new transaction
Dim transaction As New VindiciaCOM.CTransaction()
```

2. After creating the `Transaction`, initialize your authentication data and call the `SetAuthenticationInfo` method for the `Transaction` object. Type:

```
Dim username = "MyVindiciaLogin"
Dim password = "MyVindiciaPassword"
transaction.SetAuthenticationInfo(username, password)
```

where `MyVindiciaLogin` and `MyVindiciaPassword` are the SOAP user name and password, respectively, assigned to you by Vindicia.

In C#

In C#, the Vindicia namespace includes an object called `Environment`, which enables you to set the SOAP endpoint server and authentication. For example:

```
string login = "MyVindiciaLogin";
string password = "MyVindiciaPassword";

// This method allows you to change the server you connect to
Vindicia.Environment.SetEndpoint("soap.prodtest.sj.vindicia.com");

// This method allows you to set your auth info once
Vindicia.Environment.SetAuth(login, password);
```

where `MyVindiciaLogin` and `MyVindiciaPassword` are the SOAP user name and SOAP password, respectively, assigned to you by Vindicia.

1.2.3 Configuring the SOAP Timeout for Client Libraries

The method calls in your client library result in SOAP calls to the CashBox Web service hosted on Vindicia servers. You can configure the method calls to wait for a response from the server for no longer than a fixed, maximum amount of time. Each client library contains a global setting for the timeout value.

The optimal value of the timeout depends on the amount of data you are fetching from CashBox in a single call, and other factors, such as server load and network latencies. Calls that are not designed well often result in timeouts. For example, a single `Transaction.fetchDeltaSince()` call might fetch many results. To avoid data overload, set the paging parameters to page through the set when making the call. If you are experiencing frequent timeouts, examine the nature of the call you are making and determine if you can reduce the size of the result set returned by the call. If not, raise the timeout value for your client library.

In PHP

In the PHP library, edit the `Const.php` file in the `Vindicia/Soap` directory to change the value of the global constant `VIN_SOAP_TIMEOUT`. This value is in seconds and is set to 60 by default. To change the value, type:

```
define("VIN_SOAP_TIMEOUT", "30");
```

In Java

In Java, set the global SOAP timeout for CashBox API calls in the Java library. The setting is in the `com.vindicia.client.ClientConstants` class with a default value of 2,500 milliseconds. To change this value, type:

```
com.vindicia.client.ClientConstants.DEFAULT_TIMEOUT = 6000;
// in millisecs
```

In ASP, VB, and C++

The `VindiciaCOM.dll` file for ASP, VB, and C++ contains the `VindiciaCOM` object. That object supports the method `SetTimeout()`, for which you can specify the desired global SOAP timeout in milliseconds. The default is 2,500 milliseconds.

In C#

in C#, set the SOAP timeout globally (in milliseconds) using the `SetTimeout` method to the `Vindicia.Environment` class. Local timeouts may be set individually, when you instantiate the object in the C# client.

```
Vindicia.Environment.SetTimeOut(30000);
// in millisecs
```

1.2.4 Checking an Object Method's Return Value

All CashBox object methods include a `Return` data structure which indicates the success or failure of the method call. `Return` can contain three data members: `returnCode`, `returnString`, and `soapid`. (For details, see The Return Object in the ***CashBox API Guide***.)

Note	The examples in this guide do not contain error-handling. Your production applications, especially those that involve financial transactions, should ***always*** include robust error-checking and handling.

For Java, the SOAP call's `Return` object is represented by the `com.vindicia.client.VindiciaReturn` object in the Java CashBox client API.

For Java calls that are not expected to return a meaningful value (that is, if the call is expected to return a void value), the method usually returns a `VindiciaReturn` object instead of void. Examine the `VindiciaReturn` object for the `returnCode`, `returnString`, and `soapid` values.

For calls that are expected to return a meaningful value, the `VindiciaReturn` object can be a static member `LAST_RETURN` of the class of the object on which you made the call. For example, `com.vindicia.client.AutoBill.fetchByEmail()` returns an array of `com.vindicia.client.AutoBill` objects, that match the specified email address. If `com.vindicia.client.AutoBill.fetchByEmail()` does not return the expected values, check the `VindiciaReturn` object associated with the call. You can find the object in `AutoBill.LAST_RETURN`.

(The `com.vindicia.client.AutoBill.cancel()` call forms an exception to this rule, in that it returns a `VindiciaReturn` object.)

When a SOAP call returns multiple meaningful values (for example, `AutoBill.update()` returns the next billing amount, its currency, and the date in addition to the standard `VindiciaReturn` object), those values and the `VindiciaReturn` object are lumped into a single object that is returned by the corresponding Java method.

For instance, the `com.vindicia.client.AutoBill.update()` method returns a `com.vindicia.client.AutoBillingReturn` object, which lumps together all the return values of the underlying SOAP call.

1.2.5 Setting UNIX Timestamps in VB

Many CashBox objects and methods require a timestamp. By default, VB uses a built-in Microsoft date technology that produces different timestamp data than the timestamp that CashBox expects. To generate a timestamp from a VB date, add a VB function to your CashBox application, as follows:

```
Function UnixDate (ByVal theDate)
   UnixDate = DateDiff("s", "01/01/1970 00:00:00", theDate)
End Function
```

To set a timestamp for a `TransactionDetail` object named `detail`:

```
Dim timestamp = UnixDate(Now())
detail.SetTimestamp(timestamp)
```

1.2.6 Date and Timestamp Format

When a parameter to a method is a date or a timestamp, Vindicia expects them in the standard ISO8601 format:

Data Type	Format	Example
dates	`YYYY-MM-DD`	`2010-10-23`
timestamp	`YYYY-MM-DDTHH:MM:SS(+|-)HH:MM`	`2010-10-23T14:23:12-07:00`

(Note the **T** between the date and the time.)

Note: If you omit the timezone offset, it will default to Pacific Time (-7:00 or -8:00).

1.2.7 Assigning Unique Identifiers for Objects

Most top-level objects must have unique identifiers. You can directly manipulate objects, such as `AutoBill` or `Account` objects. You can load the object, usually with a `fetchBy` method; and save it, usually with an `update` method. Most objects of this kind have two unique identifiers: one assigned by you and the other by Vindicia.The naming convention for the IDs that you assign is in the form of `merchant`***`Class`***`Id`, for example, `merchantAccountId` correspond to the unique identifiers for the objects in your database, such as database IDs, email addresses, or invoice numbers, with which you track the items in question.

When you create an object in the CashBox database, Vindicia assigns the object a globally unique Vindicia identifier, called a VID, which you can access through the object's `VID` attribute. If you do not specify a VID when you call the `update()` method for an object, Vindicia generates a new VID or loads the existing one and returns it.

Caution	Do *not* specify your own VID values when creating objects.

Vindicia best practices recommend that, when creating objects with `update()` calls, clients generate their own set of object identifiers to populate the corresponding `merchant`***`Class`***`Id` values.

1.2.8 Ensuring the Correct Character Encoding

CashBox supports the UTF-8 encoded character set. When you construct CashBox objects, especially with data input by your customers (such as the data on a Web form), ensure that the strings are UTF-8 encoded.

Note:	Most programming languages contain a built-in function that enables you to convert strings into UTF-8 encoded strings. In PHP, for example, that function is `utf8_encode()`.

1.3 Working with CashBox WSDL Files

If a CashBox API client library is not available for your programming language, you can integrate with CashBox by making SOAP calls directly to Vindicia's Web services.

Because of the prevalence of Web services with SOAP as a protocol of choice for integration of disparate systems, most programming languages have built-in support for developing SOAP client-server code. A third-party plug-in or library might also be available for your language of choice. For example, Python programmers can build SOAP client-server code with the SOAPpy library. Programming a SOAP client in the language of your choice usually requires access to a Web Services Description Language (WSDL) file that describes the Web service for which you wish to create a client.

Vindicia Web services consist of a set of objects and the SOAP calls that they support (CashBox SOAP API), with the calls described in a set of WSDL files. These WSDL files support document-literal SOAP calls, as defined in the World Wide Web Consortium (W3C)

standards. Each WSDL file corresponds to a logical object commonly used in billing solutions. Objects (complex types) shared across all WSDL files are defined in the `Vindicia.xsd` file that every WSDL file includes in its definition.

Each WSDL file describes a set of calls supported by the logical object. For example, the `Account.wsdl` file describes the calls with which you can perform activities on a customer account (an `Account` object) in CashBox. Make an `update` call to create or update an `Account` object; or a `fetchByMerchantAccountId()` call to retrieve an `Account` object by the unique ID assigned by you when you created the object.

Each WSDL file defines only one SOAP port bound to the object-specific interface (a set of methods). For example, `Transaction.wsdl` defines a port called `TransactionPort`, which supports only the SOAP calls that relate to the `Transaction` object.

The ports defined in each of the WSDL files are associated with the same SOAP address (endpoint). That address is a script on Vindicia servers that receives all SOAP calls and routes them to object-specific code for further processing, depending on which objects the calls are for. For example, for CashBox 4.2:

```
<service name="Transaction">
    <port binding="tns:TransactionBinding" name="TransactionPort">
    soap:address
location="https://soap.vindicia.com/v4.2/soap.pl" />
    </port>
</service>
```

Each WSDL file imports the `Vindicia.xsd schema` file, which defines the data structure of all top-level and helper objects. Your client code must be able to access this schema file.

1.3.1 Specifying the SOAP Address

By default, the SOAP address points to Vindicia's production servers. Before going live with CashBox, test your integration code in a Vindicia sandbox server. If your language-specific implementation of a SOAP client does not override the SOAP address specified by the WSDL file, you can download the WSDL files from Vindicia, save them locally, and update the SOAP address to reflect the sandbox and CashBox SOAP API version you will use.

For example, if you are working with CashBox SOAP API version 4.2 and want to call in to CashBox on Vindicia's `Prodtest` sandbox, update the `service` attribute in your local WSDL file as follows:

```
<service name="Transaction">
    <port binding="tns:TransactionBinding" name="TransactionPort">
    <soap:address
location="https://soap.prodtest.sj.vindicia.com/v4.2/soap.pl" />
    </port>
</service>
```

1.3.2 Performing the Prerequisite Steps

Before developing client code with the CashBox WSDL files:

1. Download the WSDL files and the schema file from the Vindicia servers.

Note	Depending on your integration needs, you might not need all the WSDL files. Feel free to consult with Vindicia Client Services to decide which ones you need. See the ***CashBox API Guide*** for the objects and the methods supported by the WSDL files.

   For CashBox API production releases, download from the following sites:

   - **WSDL file:** `https://soap.vindicia.com/`***version/object***`.wsdl`, for example, `https://soap.vindicia.com/4.2/PaymentMethod.wsdl`
   - **Schema file:** `https://soap.vindicia.com/`***version***`/Vindicia.xsd`

   For CashBox API nonproduction releases that are in Vindicia's staging servers for testing prerelease client regression, download from the following sites:

   - **WSDL file:** `https://soap.staging.sj.vindicia.com/`***version***`/`***object***`.wsdl`
   - **Schema file:** `https://soap.staging.sj.vindicia.com/`***version***`/Vindicia.xsd`

2. **Optional.** Update the SOAP endpoint address to reflect the server to which to send your SOAP calls, assuming that you cannot programmatically do so in your client code.

3. Generate local stub or proxy objects with language-specific tools. For example:
   - If you program in Java and are using the Apache Axis library to work with SOAP, generate local stub objects with the utility `WSDL2Java`.
   - If you program in .Net with C#, import the WSDL files into your project to automatically generate local stub objects.
   - If you program in another language, such as Python, for which no appropriate tools are available, consult the language- or package-specific SOAP documentation on how client code can make the SOAP calls as described in the WSDL files.

4. Ensure that your SOAP library supports making SOAP calls to external servers through HTTPS.

   For security, Vindicia does not support HTTP-based SOAP calls. You might need to install additional SSL-specific libraries on your system.

1.4 Tips for Developing SOAP Clients

Remember:

- In most SOAP libraries, you can set a timeout value for the SOAP calls that you make from the client to the server. Ensure that the value is globally configurable. You may wish to change the value to fine-tune it, depending on the amount of data you will be sending or receiving from the Vindicia servers.
- Every SOAP call you make to CashBox requires that you pass an `Authentication` object, which contains the SOAP login and password assigned to you by Vindicia. Make those credentials globally configurable. When you switch to production, you must log in with credentials that differ from those used in testing against Vindicia's sandboxes.
- You might also want to make the Vindicia server, to which your client code points globally, configurable. This can simplify the process of switching from testing to production.
- Log all calls made with the CashBox client library. At a minimum, log the following:
  - Timestamps
  - Classes and methods
  - The Vindicia `Return` data structure (all three fields)
  - SOAP envelopes that are sent to and received from Vindicia servers may be used to debug data-related errors. Most SOAP libraries allow you to add an option to log these envelopes.

2 Working with Accounts

The ***Account*** in CashBox represents your customer, and contains all the data necessary to provide them services, communicate with them, and charge them for one-time or recurring purchases.

The `Account` object encapsulates the customer's account data, including billing address, shipping address, preferred payment method, and contact preferences. Create an `Account` when a customer visits your online store and registers with you.

This chapter describes creating customer Accounts; creating Account Payment Methods; accessing existing Customer Accounts; and creating Account hierarchies.

2.1 Creating Customer Accounts

When a new customer visits your site to purchase a product or service, establish an account for that customer and store the related information in a database in your system. You can also create a record of customer information in CashBox to facilitate the financial transactions you later pass to CashBox for processing.

To establish a new customer account, create an `Account` object, populate it with data, and store that data in the CashBox database using the `Account` object's `update` method. Note that the authentication information resides in the `Account` object, and that you must add the `Account` object information as a first step in a new call.

```
// Create a new Account object
$account = new Account();

// Provide basic account information: a Customer name, and
// a unique Customer ID
$account->setName('Somebody Q. Customer');
$account->setMerchantAccountId('IN9430-8421');

// To create address information, create an address object
$address = new Address();
$address->setAddr1('123 Main Street');
$address->setAddr2('Apt. 4');
$address->setCity('San Carlos');
$address->setDistrict('CA');
$address->setPostalCode('94070');
$address->setCountry('US');
$address->setPhone('123-456-7890');

// Associate the Address object with the account

$account->setShippingAddress($address);

// To set information about the customer's email contact info
$account->setEmailAddress('John.Doe@gmail.com');
$account->setEmailTypePreference('html');
$account->setWarnBeforeAutoBilling(true);

// Okay, basic information is entered, so save the account
$response = $account->update();

// Check to see that the account was created
if($response['returnCode'] == 200) {
    // You can save the VID (Vindicia ID) for later use
    $returnedAccount = $response['data']->account;
    $accountVid = $returnedAccount->getVID();
}
```

When you create an `Account` object, CashBox assigns it a globally unique Vindicia identifier (VID). As the preceding example illustrates, your application can capture the VID for later use.

The `Account` object also includes a data member called `merchantAccountId`, which enables you to assign your own identifier for the account, such as a database login, user name, or email address. Be certain to specify a unique value for this field when creating an Account.

For more information, see Section 1: The Account Object in the ***CashBox API Guide***.

2.2 Setting Up Account Payment Methods

The `Account` object can store multiple payment methods, defined by `PaymentMethod` objects. CashBox supports the following payment method types:

- **Merchant Recorded:** may be used to manually enter payments made by cash, check, or other payment method, and accepted from your customers outside the CashBox system.
- **Credit cards:** Credit cards may be used for standard purchases, including subscriptions.
- **Electronic Check Payment (ECP) through Automated Clearing House (ACH):** ECP may be used for recurring or onetime transactions. You can also use ECP to pay your vendors or affiliates in real-time transactions.
- **European Direct Debit (EDD):** This payment method is similar to ECP in the United States, in that it is a direct debit from a banking account. (Einzugsermachtigungsverfahren (ELV), the German version of EDD, requires that a customer complete a mandate to authorize the merchant to debit the customer's bank account, and is the primary payment method in use for online transactions in Germany.) CashBox supports EDD in Germany, Austria, and the Netherlands through Chase Paymentech as the payment processor.
- **Boleto Bancário:** This payment method, primarily used in Brazil, requires that the customer first provide you with a fiscal number. The payment processor then creates a payment slip with that number and other customer details. Finally, the customer presents the payment slip to the bank to complete the transaction.

For real-time transactions, the payment processor sends the merchant a URL to a website, which contains instructions for payment processing, which the merchant must present to the customer. (The steps usually involve the customer printing the instructions and a payment slip, which they then present to their bank.) Your customer must complete the steps listed on the website, before the transaction can proceed. When complete, the bank transfers the money to the payment processor, who captures the transaction, then notifies CashBox. On the CashBox side, the transaction remains pending until the payment is captured or the transaction expires.

For recurring billing (that is, an `AutoBill` object), the customer receives an email with the URL to the website with payment processing instructions every billing period payment is due. The corresponding transaction is generated and processed by CashBox in coordination with the payment processor.

- **PayPal:** CashBox supports real-time transactions through PayPal Express Checkout. CashBox also supports PayPal reference transactions for AutoBill-based recurring billing through e-wallet, whereby PayPal maintains the customer's payment information, such as the credit card number, checking account number, and bank routing number.

When making a purchase on your site, the customer is redirected to PayPal's login page to complete the payment information then, after success or failure, redirected back to your site to continue the process. For recurring bills with reference transactions, the customer needs to visit the PayPal site only once at startup. The subsequent rebilling transactions require no action on the customer's part.

- **Token:** This is a Vindicia-specific payment method that measures usage or metering, and enables you to support complex billing models that involve tracking token units in addition to fixed-price billing cycles. You define the units, such as minutes, downloads, incentive points, virtual currency, and storage. You also define token types, and manage your customers' balances by granting or decrementing tokens with CashBox objects (`Product`, `BillingPlan`, and `AutoBill`) on the CashBox Portal or with the CashBox API.

The tokens that you grant a customer are associated with a customer account, each token type having a separate balance. For example, a customer account can have separate balances for downloads, storage, and logins, which are displayed as payment methods on the customer's account page.

Note:	CashBox support for payment methods varies from payment processor to payment processor. Contact Vindicia Client Services for more information.

Define a Payment Method for an existing Account:

```
$paymentMethod1 = new PaymentMethod();
$paymentMethod1->setAccountHolderName("John Doe");

// To create billing address information, create an address object
$address = new Address();
$address->setAddr1('123 Main Street');
$address->setAddr2('Apt. 4');
$address->setCity('San Carlos');
$address->setDistrict('CA');
$address->setPostalCode('94070');
$address->setCountry('US');
$address->setPhone('123-456-7890');

$paymentMethod1->setBillingAddress($address);
// depending on the type specified below, you must populate the
// PaymentMethod object with correct sub-object (e.g. CreditCard)
// containing details of the payment method
$paymentMethod1->setType('CreditCard');

$card = new CreditCard();
$card->setAccount('4444222211113333');
$card->setExpirationDate('xxxxxx');'); // Use YYYYMM format

$paymentMethod1->setCreditCard($card);
```

```
// Sort order specifies the position (preference)
// this payment method will occupy (have) in the list of
// payment methods associated with an account. When a payment
// method is not explicitly specified in an AutoBill object,
// the first payment method in the array that is active
// will be used to schedule a recurring billing transaction
$paymentMethod1->setSortOrder(1);
$paymentMethod1->setActive('true');

// Second payment method
$paymentMethod2 = new PaymentMethod();
$paymentMethod2->setAccountHolderName("John P Doe");
$paymentMethod2->setBillingAddress($address);
$paymentMethod2->setType('CreditCard');

$card2 = new CreditCard();
$card2->setAccount('5555444411112222');
$card2->setExpirationDate('xxxxxx'); // Use YYYYMM format for date

$paymentMethod2->setSortOrder(0);
$paymentMethod2->setActive('true');

// create a payment method array
$paymentMethods = array($paymentMethod1, $paymentMethod2);

// set the payment methods in the account and create the
// account using the update call
$account->setPaymentMethods($paymentMethods);
…
```

2.3 Accessing Existing Customer Accounts

After creating a customer account, it must be accessed each time the customer is involved in a transaction.

The following table lists the methods available to access the `Account` object.

Table 2-1 Access Methods for `Account` Object

Method	Description
`fetchByEmail`	Returns the `Account` object with the specified email address.
`fetchByMerchantAccountId`	Returns the `Account` object with the specified `merchantAccountId`.
`fetchByPaymentMethod`	Returns all the `Account` objects with the specified payment method.
`fetchByVid`	Returns the `Account` object with the specified VID.
`fetchByWebSessionVid`	Returns the `Account` object with the specified `WebSession` VID.

The following example demonstrates all three methods. In production code, call only one method at a time.

```
$customerID = '1234-5678-9000';
$accountVid = 'MyCustomerVindiciaId'; // for illustration purposes only!

// Create an account object
$account = new Account();

// now load a customer account into the account object
// this example illustrates all three methods back-to-back
// but in your code you'll use only one of these methods at a time

$response = $account->fetchByMerchantAccountId($customerID);
$response = $account->fetchByVID($accountVid);
// fetchByEmail returns an array of accounts with matching email
// So in that case $response will contain an array
$response = $account->fetchByEmail('somebody@yahoo.com');

if($response['returnCode'] == 200) {
    print "ok\n"; # 200 is HTTP status code for success
}
```

The preceding example checks the return array's `returnCode`, which corresponds to a standard HTTP status code, to determine if the fetch is successful. A value of `200` indicates success, that is, the `Account` object that you created earlier now contains the customer record you would like to access.

For more information, see Section 1: The Account Object in the ***CashBox API Guide***.

2.4 Creating Account Hierarchies

CashBox supports two-level account hierarchies for payment and reporting. You may define parent and children accounts. A parent can have multiple children, but a child may have only one parent, and a child may not be a parent to another account.

A parent can pay for its own AutoBills or one-time transactions, or for any of its children's AutoBills or one-time transactions, by including the parent's Payment Method on an AutoBill with the child's Account.

Children may have Payment Methods that differ from their Parent's, and can use either to pay for their AutoBills.

When two accounts are linked or unlinked (as parent and child), an email will be sent to both.

Create an Account hierarchy, by adding credit to the parent's Account and transferring the credit to the child's Account:

```
// Create a new Account object for parent
$parent = new Account();

// Provide basic account information
$parent->setName('Somebody Q. Customer'); // Customer name
$parent->setMerchantAccountId('IN9430-8421');
// Unique customer id

// Create a new Account object for child
$child = new Account();
$child->setName('Somebody Q. Customer Jr.'); // Customer name
$child->setMerchantAccountId('IN9430-8421JR');
// Unique customer id

// Establish a parent->child relationship between
// $parent and $child
$childrenAdded = $parent->addChildren($parent, array($child))

//Grant credit to the parent
$curAmt = new CurrencyAmount ;
$curAmt->setCurrency('USD');
$curAmt->setAmount(100.00);

$cr = new Credit();
$cr->setCurrencyAmounts(array($curAmt));

// Now make the SOAP API call to grant credit to the parent
$response = $parent->grantCredit($cr);
if ($response['returnCode'] == 200) {
    // Credit successfully granted to the account
    $updatedAcct = $response->['account'];
}
else {
    // Error while granting credit to the account
    print $response['returnString'] . "\n";
}
```

```
//Define credits to be transferred from parent to child
$curTranAmt = new CurrencyAmount ;
$curTranAmt->setCurrency('USD');
$curTranAmt->setAmount(12.34);

$crTran = new Credit();
$crTran->setCurrencyAmounts(array($curTranAmt));

//Transfer specified credits from parent to child account
$response = $parent->transferCredit($child, $crTran);
if ($response['returnCode'] == 200) {
    // Credit successfully granted to the account
}
else {
// Error while transferring credit between accounts
print $response['returnString'] . "\n";
```

For more information on methods related to Account hierarchy, see Section 1: The Account Object in the ***CashBox API Guide***.

3 Working with Products

`Product` objects encapsulate information on your products or services. The `Product` object contains product information, including the product's name, description, and price. It may be a specific piece of merchandise, a one-time event, a service, or a subscription.

`Product` objects may be pre-defined, for standard merchandise or subscription plans, or created on the fly, for specialty items, such as event tickets, or limited availability objects.

3.1 Creating Products

Use `Product.update` to create a Product and populate it with data.

```
// Create a new product
$product = new Product();

// Identify the product by your unique identifier, etc.
$product->setmerchantProductId('12345');  //SKU, Database ID, etc

// This becomes transaction line item description if the
// product is used for an autobill, so this must be set
// to some meaningful string
$product->setDescription('Online subscription with video access');

$product->setPreNotifyDays(7);
$product->setStatus('Active');
$product->setDefaultBillingPlan($billing_plan);
    //From a previous update or fetch

$product->merchantEntitlementIds[0] =
    (new MerchantEntitlementId
        id => 'Video Only',
        description => 'Video access'));

$response = $product->update(DuplicateBehavior::SucceedIgnore);

if($response['returnCode'] == 200) {
    print "ok\n";
    $prodVid = $product->getVID();
        // capture the product VID for later use
}
```

When you create a `Product` object, CashBox assigns it a globally unique Vindicia identifier (VID). As the preceding example illustrates, your application can capture the VID for later use.

The `Product` object also includes a data member called `merchantProductId`, which allows you to assign your own identifier, such as the stock-keeping unit (SKU), for the product. The `merchantProductId` ***must*** be unique for each `Product` object you define.

Like `BillingPlan` objects, `Product` objects are stable objects that are usually created at the start of a paid service by business analysts or people in business roles. The CashBox Portal allows you to create and manipulate `Product` objects through a user interface. See the ***CashBox User Guide*** for more information.

Note	Do not make changes to an active `Product` object once subscriptions are activated in the system (that is, once the related `Account`, `AutoBill`, and `BillingPlan` objects that reference a particular `Product` object are active). Create a new `Product` object instead.

The `Product` object supports several information-only attributes, such as `endOfLifeTimestamp` and `status`. These attributes may be used to sort or categorize your products.

`Product` objects can grant any number of token types. When a real-time or recurring transaction is made for a product that grants tokens, CashBox grants those tokens, and associates them with the `Account` object in question.

For more information, see Section 13: The Product Object in the ***CashBox API Guide***. For information on how the `Product` object affects entitlements, see Section 7.1: Creating Entitlements.

3.2 Creating Bundled Products

Bundling Products allows you to offer special packages, in which multiple Products are included as a single item on the AutoBill. The price for a bundled Product is defined by the top-level Product, but this price may be overridden by the Billing Plan or AutoBill, if desired.

Create a bundled Product.

```
$top_product = new Product();
$top_product->setMerchantProductId('top-1');
$top_product->update();

$bundled1 = new Product();
$bundled1->setMerchantProductId('sub-1');
$bundled1->update();

$bundled2 = new Product();
$bundled2->setMerchantProductId('sub-2');
$bundled2->update();

// of course, other attributes can be set on the above products

$top_product->bundledProducts(array($bundled1, $bundled2));

$response = $top_product->update();

if ($response['returnCode'] == 200) {

    $ret_prod = $response['data']->product;
    print "got product ", $ret_prod->merchantProductId(), "\n";
    $bundledProds = $ret_prod->bundledProducts();

    if ($bundledProds != null) {
        foreach ($bundledProds as $bp) {
            print $bp->merchantProductId(), " is bundled\n";
        }
    }
}
```

Note: The price for a Bundled Product is defined by the top-level Product. This allows you to create groups of Products that may be purchased for one set price.

3.3 Accessing Existing Products

After creating a `Product` object, access it each time a customer purchases or subscribes to it.

The following table shows the methods of access to the `Product` object.

Table 3-1 Access Methods for the `Product` Object

Name	Description
`fetchAll`	Returns all the `Product` objects.
`fetchByAccount`	Returns one or more `Product` objects whose `Account` object matches the input.
`fetchByMerchantEntitlementId`	Returns all the `Product` objects whose entitlement ID assigned by you (`merchantEntitlementId`) matches the input.
`fetchByMerchantProductId`	Returns an `Account` object with the specified `merchantAccountId`.
`fetchByVid`	Returns an `Account` object with the specified VID.

The following example calls two methods. In production, call only one method at a time.

```
$sku = '12345';
$productVid = 'MyProductVindiciaId'; // for illustration purposes only!

// Create a product object
$product = new Product();

// now load an existing product into the Product object
// this example illustrates both methods back-to-back
// but in your code you'll use only one of these methods at a time
$response = $product->fetchByMerchantProductId($sku);
$response = $product->fetchByVid($productVid);

if($response['returnCode'] == 200) {
print "ok\n"; # 200 is HTTP status code for success
}
```

The preceding example checks the return array's `returnCode` to determine if the `fetch` succeeded. A value of `200` indicates success; that is, the `Product` object created earlier now contains the product record you would like to access.

For more information, see Section 13: The Product Object in the ***CashBox API Guide***.

4 Working with Billing Plans

Billing Plans define the rate and frequency of subscription charges. Billing Plans include Billing Periods, which include the Billing Plan's subset cycles, of varying frequency, duration, and price. For example, a Billing Plan may be made up of an initial Billing Period, which bills your customer $29.95 on the 5th of each month for three months, and a second Billing Period, which bills your customer $199.95 on the 5th day of every 6th month (bills twice per year), indefinitely.

The `BillingPlan` object encapsulates Billing Plan information, including the current status of the Billing Plan, the number of Billing Periods contained within it, and any Entitlements that may be associated directly with the Billing Plan.

4.1 Creating Billing Plans

`BillingPlan` objects describe the amount and schedule for recurring charges. `Product` objects describe the products or services that you sell. Both objects may include a Price..

> **Note:** An `AutoBill` object includes a `BillingPlan`, and an array of `AutoBillItems`. An `AutoBillItem` includes a `Product`, the number of cycles the `Product` is to be included on the `AutoBill`.
>
> For more information, see the `AutoBillItem` Subobject in the ***CashBox API Guide***.

For example, a business that sells access to online magazines, which cover different topics, and are supported by different websites, but offer a standard subscription plan, might want to create a single `BillingPlan` which offers a one-month free trial, followed by a recurring bill of US$19.95 or C$22.40 per year.

> **Note:** Although multiple Billing Plans may be created dynamically, as shown in the following example, best practice recommendations are to create stable `BillingPlan` objects to be used when individual customers subscribe.
>
> The CashBox Portal allows you to create and manipulate `BillingPlan` objects through a user interface. See the ***CashBox User Guide*** for details.

Create a Billing Plan with a one-month free trial, followed by a recurring bill of US$19.95 or C$22.40 per year:

```
// Create a new BillingPlan
$bp = new BillingPlan();

$bp->setMerchantBillingPlanId('1MF1995Y');
$bp->setDescription('1 Free Month then followed by $19.95(USD),
          $22.40(CAD) per year');
$bp->periods[0]=(new BillingPlanPeriod(
       type => 'Month',
       quantity => 1,
       cycles => 1, //Just once
       prices => [new BillingPlanPrices('amount' => 0,
              'currency' => 'USD'),
          new BillingPlanPrices('amount' => 0,
              'currency' => 'CAD')]));
$bp->periods[1]=(new BillingPlanPeriod(
       type => 'Year',
       quantity => 1,
       cycles => 0, //Repeat infinitely
       prices => [new BillingPlanPrices('amount' => 19.95,
              'currency' => 'USD'),
          new BillingPlanPrices('amount' => 22.40,
              'currency' => 'CAD')]));
```

```
$bp->merchantEntitlementIds[0] = (new MerchantEntitlementId (
       id => 'Standard',
       description => 'Standard subscription access'));

    $response = $bp->update();

if($response['returnCode'] == 200) {
    print "ok\n";
    $bpVid = $bp->getVID();
       //capture the billing plan VID for later use
}
```

Note:	Once subscriptions have been activated in the system (that is, an `AutoBill` object that uses a particular `BillingPlan` object is active), do *not* make changes to the underlying `BillingPlan` object, other than to change the `BillingPlan` Cycle amounts. For all other changes, create a new `BillingPlan` object.

The `BillingPlan` object supports several information-only attributes, such as `endOfLifeTimestamp` and `status`. These attributes may be used to sort or categorize your customers. For example, you can fetch all `BillingPlan` objects, and display only those objects for active customers, or for customers whose end-of-life timestamp has not yet passed.

Note:	Billing Plans may be processed in currency or in tokens, but not in both. For example, you may set up your environment to support several token types (such as downloads and storage) and charge 50 downloads per month for access to the system.

For details on how the `BillingPlan` object affects entitlements, see Section 7.1: Creating Entitlements.

Billing Plans may also be used to grant Seasonal Entitlements, using the SeasonSet object. For example, Create a 4-installment seasonal Billing Plan for the next 3 summers. The initial purchase must result in 2 free weeks, but subsequent years should not include a free period. (This is controlled using "`skipInitialFreeWhenRepeating`.")

Create a Billing Plan with an attached Season Set:

```
// To create this BillingPlan, you must first create the SeasonSet.

$ss = new SeasonSet;
$ss->merchantSeasonSetId("Summer");

$s2014 = new Season;
$s2014->description("Summer 2014");
$s2014->startDate("2014-05-15");
$s2014->endDate("2014-10-10");

$s2015 = new Season;
$s2015->description("Summer 2015");
$s2015->startDate("2015-05-12");
$s2015->endDate("2015-10-11");

$s2016 = new Season;
$s2016->description("Summer 2016");
$s2016->startDate("2016-05-19");
$s2016->endDate("2016-10-08");

$ss->seasons( [$s2014, $s2015, $s2016] );
$ss_factory->update($ss);

// Check the return code from update.

$bp = new BillingPlan;
$bp->merchantBillingPlanId("summer4installment");
$bp->description("Summer Installment Plan");
$bp->seasonSet($ss);
$bp->repeatEvery("1 Season");
$bp->timesToRun("unlimited");
$bp->skipInitialFreeWhenRepeating(1);

// Note that, although the code says to run this every 1 Season
// for an unlimited number of times, there are only 3 seasons
// in the SeasonSet. This is OK as long as you (later) add more seasons.
// (They must be added before the time we would bill for them;
// in this example they would have to be added by spring 2017.)

$bp_free = new BillingPlanPeriod;
$bp_free->free(1);
$bp_free->cycles(1);
$bp_free->quantity(2);
$bp_free->type("Week");

$bp_monthly = new BillingPlanPeriod;
$bp_monthly->free(0);
$bp_monthly->cycles(4);
$bp_monthly->quantity(1);
$bp_monthly->type("Month");

$bp->periods( [ $bp_free, $bp_monthly ] );

$bp_factory->update($bp);

// Check the return code from update.
```

5 Working with AutoBills

An AutoBill is used to manage a customer's subscription to a Product. The AutoBill combines a customer Account, a Product, and a Billing Plan to describe the subscription. An AutoBill automates billing notifications and recurring billing over the life of the subscription, by generating and submitting Transactions to payment processors.

The `AutoBill` object defines the relationship of a customer `Account` to a `Product` and a `BillingPlan`. It includes information on the currency to be used for payment processing, the day on which billing is to occur, and whether or not automated billing notifications should be issued.

5.1 Creating AutoBills

The CashBox API makes it easy to set up automatically recurring billing. Construct an `AutoBill` object and set its attributes to define the behavior of the recurring bill. AutoBills contain three main components: the `Account`, `Product`, and `BillingPlan`.

CashBox automatically generates recurring transactions according to the definition in the `AutoBill` object, and processes them with your payment processor.

Create a Product:

```
$sku = '12345';

// Create a Product object
$product = new Product();

$product->setMerchantProductId($sku);

//Describe (name) the product. This description will
//appear with the transaction item for each recurring bill

$product->setDescription("Virtual world game");

// You can define entitlements on products and billing plans.
// Entitlements provided by a product will be available to all
// AutoBills using this product, no matter which
// Billing Plan they use

$product->merchantEntitlementIds[0] =
    (new MerchantEntitlementId (
       id => 'VW Game',
       description => 'Game subscription'));

// Now create the product
$response = $product->update(DuplicateBehavior::SucceedIgnore);

if($response['returnCode'] == 200) {
   print "ok\n";
}
```

Establish recurring billing with an AutoBill by specifying the initial commitment period:

```
$autobill = new AutoBill();

// Subscribe an existing customer to an existing product with a
// default billing plan specified

$autobill->setAccount($account);
$item = new AutoBillItem();
$item->setIndex(0);
$item->setProduct($product); // set the Product in the AutoBillItem
$autobill->setItems(array($item));

// Create a new BillingPlan
$bp = new BillingPlan();

$bp->setMerchantBillingPlanId('1MF1995Y');
$bp->setDescription('Basic monthly billed subscription plan');

// The billing plan offers gold level access
$bp->merchantEntitlementIds[0] = (new MerchantEntitlementId (
       id => 'Gold',
       description => 'Games with gold level access'));

// … Other billing plan definition code would go here

// Create a BillingPeriod object
$period = new BillingPlanPeriod();
// zero specifies an infinite number of rebilling periods
$period->count = 0;
// number of period types that comprise a single unit
$period->quantity = 1; // want to bill once per month
$period->type = 'Month';

// Associate the Billing Plan Period with the Billing Plan

$bp->periods[0]= $period;

//Now create the BillingPlan

$response = $bp->update();
if($response['return']->returnCode == 200) {
   print "ok\n";
}

// Set the BillingPlan for the AutoBill

$autobill->setBillingPlan($bp)

// Set initial minimum commitment period
$autobill->setMinimumCommitment(2);
$autobill->setSourceIp('56.34.189.212');
```

```
// If subscribed to Vindicia's risk screening service, you can
// specify the minimum chargeback probability (0-100)
// that you will tolerate. At creation time of the AutoBill,
// CashBox will generate and score a transaction with the
// payment method specified for the AutoBill and billing address
// on the account, and IP address on the
// AutoBill. If the risk score returned is less than the minimum
// chargeback probability specified here, the AutoBill will be
// created. Otherwise, it will fail.
// The evaluated score and messages explaining the score
// are available in the return parameters 'score' and 'scoreCodes'

$minChargebackProbability = 50;

// The duplicate behavior parameter is not in use.
// Its value does not currently affect the behavior of the
// AutoBill->update() call. In general, remember that
// the update() call will update an existing object if an
// AutoBill object with VID or merchantAutoBillId in the
// input object already exists. If such an object does not
// already exist in the CashBox database, a new one will be created.

$duplicateBehavior = 'Fail';

// When creating an AutoBill, specify whether or not the
// payment method associated with the AutoBill should be validated.
// If set to true, this flag causes CashBox to pre-auth
// the payment method. Type of validation and whether validation
// is performed depends on the payment method used.

$validatePaymentMethod = 'true';

// Call update to update or create the AutoBill in CashBox

$response = $autobill->update($duplicateBehavior,
       $validatePaymentMethod, $minChargebackProbability);

if($response['returnCode'] == 200) {
   print "ok\n";
}
```

5.1.1 Creating an AutoBill with Multiple Products

CashBox allows you to include multiple line items on an AutoBill, which may include multiple, recurring Products, and one-time, non-recurring Charges.

> **Note:** Products and Charges may be added to or removed from the AutoBill at any time, using the `AutoBill.modify` call. If a product or charge is added or removed in the middle of a billing cycle, you may set a flag to allow CashBox to determine the prorated amount to charge or refund, based on the point in the billing cycle and the billing date of the AutoBill.

Add Products to an AutoBill using the `AutoBillItem` object. An `AutoBillItem` includes a `Product`, the number of cycles the `Product` is to be included on the `AutoBill`, and an amount (price).

For more information, see the `AutoBillItem` Subobject in the ***CashBox API Guide***.

Create an AutoBill with two products:

```
$autobill = new AutoBill();
$autobill->setMerchantAutoBillId('ab-1');

$product1 = new Product();
$product1->setMerchantProductId('prod-AAA');

$product2 = new Product();
$product2->setMerchantProductId('prod-BBB');

$item1 = new AutoBillItem();
$item1->setIndex(0);
$item1->setAmount(4.50);
$item1->setCurrency('USD');
$item1->setCycles(null);
$item1->setProduct($product1);

$item2 = new AutoBillItem();
$item2->setIndex(1);
$item2->setAmount(7.95);
$item2->setCurrency('USD');
$item2->setCycles(null);
$item2->setProduct($product1);

$autobill->setItems( [ $item1, $item2 ] );
```

```
$autobill_factory = new AutoBill();
$response = $autobill_factory->update(
       $autobill, // $autobill is of type Vindicia::Soap::AutoBill
       'Fail',    // $duplicateBehavior is of type
            // Vindicia::Soap::DuplicateBehavior
       true,      //$validatePaymentMethod is of type xsd:boolean
       100,       // $minChargebackProbability is of type xsd:int
       false,     // $ignoreAvsPolicy is of type xsd:boolean
       false,     // $ignoreCvnPolicy is of type xsd:boolean
       null,      // $campaignCode is of type xsd:string
       false,     // $dryrun is of type xsd:boolean
       );

// check $response

$response = $autobill->addCharge(
       'prod-bac', // product Id for this charge 'fee for swapping tiles',
       null,      // will get tax class from Product
       1.50,
       'USD',
       null,      // not a token
       1          // just once
       );

// check $response
```

5.1.2 Updating and Validating AutoBill Objects

To ensure that a customer has entered a valid Payment Method, before accepting it for use to pay AutoBills, turn on payment method validation by setting the ***validatePaymentMethod*** or validate flag to true in either the AutoBill or PaymentMethod.update() method. However, If you have already validated the payment method (for example, if you are importing and creating AutoBill objects from previously successful transactions, or if you have successfully performed a real-time transaction before creating the AutoBill object), you might choose to turn validation off and reduce the number of calls that you make.

Note: (CVN is Vindicia's generic term for credit-card security code, usually located on the back of the card. Some credit-card companies call it CVV, CVV2, CVC2, CCID, or other acronyms.)

Validation generates and sends to the payment processor one of three types of transactions:

- **$0 Authorization:** Assuming that the payment processor supports $0 authorization, if the creation of an AutoBill object does not result in an immediate billing, as in the case of a free trial period, or if the AutoBill does not start immediately, CashBox creates a real-time validation with a zero amount, and sends it to the payment processor.
- **$1 Authorization:** If the payment processor does not support $0 authorization, and if the creation of an AutoBill object does not result in an immediate billing, or if the AutoBill object does not start immediately, CashBox creates a real-time transaction with a 1.00 amount and sets the currency to the customer's currency (for example, US$1.00 if USD, or €1.00 if EUR is the currency associated with the Account object) and sends it to the payment processor. CashBox authorizes this transaction instantly with your payment processor, but does not mark the transaction for capture, so the customer is not charged for it.
- **Authorization for the full amount of the first billing cycle:** Using a configuration setting, you can enable CashBox to perform a real-time AuthCapture operation for the full amount of the initial billing cycle. This operation occurs only if the AutoBill object is set to start immediately, that is, it does not offer any initial free trial period or is not scheduled to start today. Instead of performing a validation (for $0 or $1) and then an AuthCapture operation for the full amount due later, CashBox simply performs AuthCapture, saving you the cost of the validation.

If payment authorization fails on the attempted validation method, CashBox does not create the AutoBill object, and you may request a different payment from the customer.

CashBox also validates other payment methods, such as ECP, if they are supported by the payment processor. For example, with Chase Paymentech, CashBox validates ECP-based payment methods by running initial checks on the related bank routing numbers and account numbers. Part of this check verifies whether the routing number belongs to a known bank, and if the account number is blacklisted in the Chase Paymentech database.

For more information, see the TransactionStatus Subobject in the ***CashBox API Guide***, and Section 4.1: AutoBill Data Members in the ***CashBox API Guide***.

5.1.3 Verifying AVS and CVN for Recurring Billing

Using AVS (Address Verification System) verification requires that the `PaymentMethod` object that represents the credit card contain a billing address. Using CVN (Card Verification Number) verification requires that the `PaymentMethod` object contain the security code from the back of the card. Pass the security code from the credit-card owner to CashBox using a name–value pair on the `PaymentMethod` object. In this pair, set the name to `CVN`, the CashBox generic name for all security codes; and the value to the ***security code***.

Note	PCI regulations prohibit storing a credit card's CVN security code beyond the limited time frame for verification. Do not store the CVN security code in your system. Vindicia retains it only for as long as it is needed for authorization, and then discards it.

Authorization data from your payment processor for the transaction is located in the `transactionStatus` field (see Section 18: The Transaction Object in the ***CashBox API Guide***, for more information) returned by the `AutoBill.update()` call. Ensure that your code examines this field.

- If the credit card is approved, the `TransactionStatus` object's `status` attribute is set to `Authorized`; if not, it is set to `Cancelled`.
- CashBox interprets the AVS response code from your payment processor, which can vary from processor to processor, and sets `vinAVS` to one of the values specified in the `AVSMatchType` enumeration (see the `AVSMatchType` Subobject in the ***CashBox API Guide***).

  The values of AVS response codes interpreted by CashBox from your payment processor are in simple English: `Match, Partial Match, No Match, No Opinion, Not Supported, and Issuer Error`. To retrieve the actual response code, look up the `TransactionStatusCreditCard` object inside the `TransactionStatus` object (`creditCardStatus` attribute). The `CreditCardStatus` object contains an attribute called `avsCode`, which contains the return code received from your payment processor.
- The values of CVN response codes interpreted by CashBox from your payment processor are in simple English: `Match`, `Not Processed`, `Not Supported`, `Not Present`, `Invalid`, and `No Response`. To examine the CVN response code, use the `cvnCode` attribute of the `TransactionStatusCreditCard` object inside the `TransactionStatus` object (`creditCardStatus` attribute). `cvnCode` contains the response from your payment processor for the security code you sent with the payment method.

Refer to the documentation from your payment processor for the translation of the return codes; they vary from processor to processor.

```
$paymentMethod1 = new PaymentMethod();

$paymentMethod1->setAccountHolderName("John Doe");

// To create billing address information, create an address object
$address = new Address();
$address->setAddr1('123 Main Street');
$address->setAddr2('Apt. 4');
$address->setCity('San Carlos');

// populate other address attributes here

// Billing address on payment method is required to
// implement address verification
$paymentMethod1->setBillingAddress($address);

$paymentMethod1->setType('CreditCard');

$card = new CreditCard();
$card->setAccount('4444222211113333');
$card->setExpirationDate('xxxxxx'); // Use YYYYMM format for date

$paymentMethod1->setCreditCard($card);

$nv = new NameValuePair();
$nv->setName("CVN");
$nv->setValue("123");
   // this is the card security code provided by customer

// set the card security code inside the payment method
$paymentMethod1->setNameValues(array($nv));

$account = new Account();
// populate other account attributes here

$account->setPaymentMethods(array($paymentMethod1));

$abill = new AutoBill();
$abill->setAccount($account);

$minChargebackProbability = 50;
$duplicateBehavior = 'Fail';
$validatePaymentMethod = true;

// Now call update to create the autobill on CashBox servers

$response = $autobill->
       update($duplicateBehavior,
       $validatePaymentMethod,
       $minChargebackProbability);

if($response['returnCode'] == 200) {
   print "Credit card was approved and subscription created. \n";
}
```

```
else if ($response['returnCode'] == 402) {
    print "Payment method was not approved "
    $txStatus = $response['authStatus'];
    $authCode = $txStatus->getCreditCardStatus()->getAuthCode();
    print "Auth code received from processor " . $authCode . "\n";
}
```

Note	Use the results from the of credit-card security code verifications to determine whether to create an `AutoBill` object, or update one with a new payment method. Once you have created an `AutoBill` object and started generating transactions, these verification mechanisms may no longer be used. PCI regulations forbid the storage of credit-card security codes. Because CashBox does not store these numbers, CashBox is unable to perform verification for subsequent transactions using the payment method for the `AutoBill` object when it was first created.

5.2 Modifying AutoBills

`AutoBill.modify` may be used to add, remove, or exchange a Product on the existing AutoBill, or to replace the existing Billing Plan with a new Billing Plan.

`AutoBill.modify` is designed to:

- Work with AutoBills that have any number of AutoBillItems.
- Allow the addition, removal or replacement of any AutoBillItems, in a single call.
- Work with Campaigns.
- Retain historic AutoBill data (such as payment history and Product choice evolution) while making changes to existing AutoBills.
- Generate a pro-rated net charge or refund for the combined modification activity. This charge or refund will appear through the API and GUI with other Transactions from this AutoBill.
- Keep the AutoBill in its original state if any aspect of the call, including the modification charge or refund, fails.

Note:	AutoBills containing Rated Products may not be modified.

The `AutoBill.modify` call allows you to change the existing Products or Billing Plan for an AutoBill, without losing the history of the subscription.

Do not use the `AutoBill.modify` call to change anything other than the Products or Billing Plan for an AutoBill.

5.2.1 Prorating Modification-Based Price Changes

When making the `modify` call, you must specify an effective date, which may be the next billing date, or the current date. If you elect to set the `billProratedPeriod` input variable to `true`, CashBox will calculate any prorated charges or refunds as of the effective date. (The net charge or refund is calculated based on the difference between your customer's current AutoBill charge, and the modified AutoBill's charge, as of the effective date.)

If you wish the change to take effect immediately, specify an effective date of `TODAY`, and issue a pro-rated charge for the partial Billing Period that your customer will enjoy the new Products.

For example, if customer is billed on the first of every month, and on the 7th day of a 30-day month, they change from a $10/month product to a $15/month product, the month's charges will be calculated as follows:

(new plan cost - old plan cost) * (number of days on new plan / number of days in period) =

($15 - $10) * (24 / 30) = $4

Note: Please note that CashBox calculates prorated charges by the day.

$4 is the cost of the change today, and will be charged to the customer immediately. (Set the ***billProratedPeriod*** flag of the `AutoBill.modify` method to `false` to skip this charge, if desired). If you are performing a downgrade, the cost will be negative, and will be immediately refunded.

Their next bill would then be on the first of the next month for $15.

Using the `AutoBill.modify` call to change the customer's Billing Plan will always reset the billing date to ***today***. Any surcharges or refunds will be calculated appropriately.

Note: Use the ***dryrun*** input parameter for the `AutoBill.modify` call to return any charges or credits that would be levied as a result of the change, without modifying the AutoBill. This allows you to show your customers a preview of changes to their subscription, without saving the change.

5.2.2 Changing Products for an AutoBill

The following example demonstrates how to use the `AutoBill.modify` call to change Products on an AutoBill, replacing an existing Product with a new Product, to which a Campaign discount has been applied.

```
$abill = new AutoBill();
$abill->setMerchantAutoBillId('vin_test_abill1391561675069');
   // This is ID of the AutoBill you want to modify.

$productRemoved = new Product();  // Product we want to remove.
$productRemoved->setMerchantProductId('regular-avataxed-product');
   // Identify the Product by its ID. There is no need to fetch it.
```

```
$itemRemoved = new AutoBillItem();
$itemRemoved->setProduct($productRemoved);

$productAdded = new Product(); // Product we want to add.
$productAdded->setMerchantProductId('avataxed-ott-product');
   // Identify the Product by its ID. There is no need to fetch it.

$itemAdded = new AutoBillItem();
$itemAdded->setProduct($productAdded);
$itemAdded->setMerchantAutoBillItemId('item-987654');  // Unique item ID.
$itemAdded->setCampaignCode('OTT');
   // To apply a promo to the Product being added.

$modification = new AutoBillItemModification();
$modification->setRemoveAutoBillItem($itemRemoved);
$modification->setAddAutoBillItem($itemAdded);

$modifications = [ $modification ];

$abmr = abill.modify(
          true,    // bill prorated period
          'today', // make the change effective today
          null,    // not changing the billing plan
          $modifications,
          false    // no dry run
          );

if ($abmr->getReturnCode == 200) {
   // We got a 200 response code back
   // The modification may result a net refund or net charge
   // to the customer.
   $refunds = $abmr->getRefunds();
   $tx = $abmr->getTransaction();
   if ($refunds != null && $refunds->length > 0 ) {
      // Modification resulted into net refund
      // In most cases there should be only one refund
      $the_refund = $refunds[0];
      print('Total refund amount due to modification: '
          . $refunds[0]->getAmount()
          . ' . See break down below: ');

      // To give the customer a break down of how we reached
      // the refund amount, we must parse the $0 transaction
      // that accompanies this.
      if ($tx != null ) {
         printTxDetails($tx);
      }
   }
   else  {
      // Refund is null, so there must be net cost to the customer
      if ($tx != null ) {
         print('Total charge processed during modification: $'
              . $tx->getAmount()
              . ' - see break down below:');
         printTxDetails($tx);
      }
   }
```

```
}
```

5.2.3 Changing the Billing Plan for an AutoBill

The following example demonstrates how to use the `AutoBill.modify` call to change the Billing Plan on an AutoBill, replacing the existing with a new Billing Plan.

```
$abill = new AutoBill();
$abill->setMerchantAutoBillId('TEST-AB-2');
   // this is ID of the AutoBill you want to modify

$newBp = new BillingPlan();
$newBp->setMerchantBillingPlanId('monthly');
   // The ID of the Billing Plan you want change the AutoBill to.

$abmr = $abill->modify(
          true,    // bill prorated period
          'today', // make the change effective today
          $newBp,  // changing the billing plan
          null,    // no product modifications
          false    // no dry run
          );

if ($abmr->getReturnCode() == 200) {
   // We got a 200 response code back
   // The modification may result a net refund or net charge to the customer
   $refunds = $abmr->getRefunds();
   $tx = $abmr->getTransaction();
   if ($refunds != null and $refunds->length > 0 ) {
      // Modification resulted in a net refund.
      // In most cases there should be only one refund.
      $the_refund = $refunds[0];
      print('Total refund amount due to modification: '
          . $the_refund->getAmount() . '
          . See break down below: ');

      // To give customer break down of how we reached refund amount,
      // we must parse the $0 transaction that accompanies this.
      if ($tx != null ) {
         printTxDetails($tx);
      }
   }
   else  {
      // Refund is null, so there must be net cost to the customer
      if ($tx != null ) {
         print('Total charge processed during modification: $'
             . $tx->getAmount()
             . ' - see break down below:');
         printTxDetails(tx);
      }
   }
}
```

5.2.4 Changing both Products and Billing Plan in a Single Call

The following example demonstrates how to use the `AutoBill.modify` call to change both the Products and the Billing Plan for an AutoBill simultaneously.

```
$abill = new AutoBill();
$abill->setMerchantAutoBillId('vin_test_abill1391560836975');
   // This is ID of the AutoBill you want to modify.

$productRemoved = new Product();  // Product we want to remove.
$productRemoved->setMerchantProductId('monthlySub');
   // Simply identify the Product by its ID. No need to fetch it.

$itemRemoved = new AutoBillItem();
$itemRemoved->setProduct($productRemoved);

$productAdded = new Product(); // Product we want to add
$productAdded->setMerchantProductId('AnnualSubProduct');
   // Simply identify the Product by its ID. No need to fetch it.

$itemAdded = new AutoBillItem();
$itemAdded->setProduct($productAdded);
$itemAdded->setMerchantAutoBillItemId('item-425304');  // Unique item ID.
$itemAdded->setCampaignCode('ANNUALPROMO');
   // To apply a Promo to the Product being added.

$modification = new AutoBillItemModification();
$modification->setRemoveAutoBillItem($itemRemoved);
$modification->setAddAutoBillItem($itemAdded);

// We want to change to the annual plan.

$newPlan = new BillingPlan();
$newPlan->setMerchantBillingPlanId('annual-plan-2');

$modifications = [ $modification ];

$abmr = abill.modify(
          true,  // Bill prorated period.
          'today', // Make the change effective today.
          $newPlan, // Not changing the Billing Plan.
          $modifications,
          false  // No dry run.
          );
```

```
if ($abmr->getReturnCode() == 200) {
   // We got a 200 response code back.
   // The modification may result a net refund or
   // net charge to the customer.
   $refunds = $abmr->getRefunds();
   $tx = $abmr->getTransaction();
   if ($refunds != null && $refunds->length > 0 ) {
      // Modification resulted into net refund
      // In most cases there should be only one refund
      $the_refund = $refunds[0];
      print('Total refund amount due to modification: '
          . refunds[0]->getAmount()
          . ' . See break down below: ');

      if ($tx != null ) {
         printTxDetails($tx);
      }
   }
   else {
      // Refund is null, so there must be net cost to the customer.
      if ($tx != null ) {
         print('Total charge processed during modification: $'
              . $tx->getAmount()
              . ' - see break down below:');
         printTxDetails($tx);
      }
   }
}
```

5.3 Cancelling AutoBills

To stop recurring billing, cancel the corresponding `AutoBill` object. Either:

- Retrieve the object and call `cancel()` on it. For more information, see the `AutoBill.cancel` method in the ***CashBox API Guide***.
- Call `stopAutoBilling()` on the `Account` object that represents the customer. (See the `Account.stopAutoBilling` method in the ***CashBox API Guide***.)

(Both methods allow you to select whether to disentitle the customer immediately, or allow the entitlements to continue until the end of the current Billing Period.)

When you make these calls, CashBox notifies the customer of the cancellation. For details, see Section 9.2: Working with Billing Events.

CashBox allows you to automatically cancel an `AutoBill` if a customer charges back an `AutoBill` transaction. To take advantage of this feature, contact Vindicia Client Services.

5.3.1 Cancelling `AutoBills` on Billing Day

If you cancel an `AutoBill` object on billing day, and the CashBox process that generates the related transactions has already begun, CashBox still bills your customer. If you have set up cancellation and success notifications, your customer could also, for a short period of time, receive a cancellation notice followed by a success notification.

The result of the `AutoBill.cancel()` call returns a success code of `200` if the call succeeded. Vindicia recommends that you also call `Transaction.fetchByAutoBill()` to check the transaction status.

5.4 Importing AutoBills from other Billing Systems to CashBox

The CashBox `AutoBill.migrate`, `Transaction.migrate`, and `Refund.report` calls allow you to import existing subscription and Transaction information from your billing system to CashBox. The `AutoBill.migrate` call will create new AutoBills which reflect the imported information.

`AutoBill` and `Transaction.migrate` allow you to:

- Bring a subscriber into the CashBox system,, while preserving their pre-CashBox history.
- Refund transactions using CashBox, even if the transaction was not originally from CashBox, eliminating the need for you to maintain multiple billing systems.
- Perform all customer service functionality (including modify) on existing subscribers before CashBox has billed them, eliminating the need to maintain two customer service flows.

After migration, these Transactions and AutoBills will be processed and treated as if they had originated with CashBox, allowing you to use this method to:

- import existing customers in good standing.
- import customers who were in good standing, but whose most recent billing cycle was unsuccessful
- import historic billing information for your customers, offering you a continuous record of their subscriptions.

Use `AutoBill.migrate` to migrate existing active subscriptions from your billing system to CashBox. The `AutoBill` and `MigrationTransactions` provided in this call will be used to replicate your system's billing history, and future subscription behavior (to the maximum extent possible) in CashBox. Once migrated to CashBox, you may perform operations on the AutoBill (such as modify, cancel, and update) and expect behaviors that replicate that of an AutoBill created in CashBox. `MigrationTransactions` included in the `AutoBill.migrate` request will result in the creation of Transactions that can be operated on as if they had originated in CashBox (including `refund` and `fetch` calls).

CashBox supports the following transaction status types on `MigrationTransactions` included in an `AutoBill.migrate` call:

- `Captured`
- `Cancelled`
- `Refunded`
- `Settled`
- `Void`

5.4.1 Key Migrate Parameters

Be certain to include a `MigrationTransaction` reflecting the most-recent subscription billing attempt in your first `migrate` call for a given AutoBill. This `MigrationTransaction`, and the `nextPeriodStartDate` will be used to reconstruct the billing schedule for the AutoBill. Supplemental calls to `AutoBill.migrate` may be made to back-fill historical data for a given AutoBill.

Note:	Subsequent calls to `AutoBill.modify` will not result in modifications to the AutoBill (only the `MigrationTransactions` will be processed to create historic Transaction information). Also note that Vindicia allows only one recurring Transaction to be provided for each billing period in a subscription's billing history.

5.4.2 Migrating an AutoBill During a Billing Cycle

This example demonstrates how to migrate an AutoBill that has completed two monthly billing cycles on a pre-existing billing system. The AutoBill is migrated to CashBox before its third billing date.

```
// Construct the AutoBill to be migrated

$migrBill = new AutoBill();

// The subscription was originally started on the
// existing system on Dec. 27, 2013.
$migrBill->setStartTimestamp('2013-12-27');

// Set a unique subscription ID. If the subscription
// existing in your current system has an ID.
migrBill.setMerchantAutoBillId('SampleMigratedAutoBill-950681');


// Specify the Billing Plan the migrated AutoBill will be on.
// Make sure the Billing Plan used below preexists in CashBox.
$billPlanId = 'migrMonthly';
$bp = new BillingPlan();
$bp->setMerchantBillingPlanId($billPlanId);
$migrBill->setBillingPlan($bp);

// Specify the Product the migrated AutoBill will be using.
// Make sure the Product used below is created in CashBox in advance

// Now, fetch the Product to get more info about the Product.
// That info can be used to fill in the migration transactions.
// Exception handling code for the fetch call is omitted for brevity.
// This step is optional. The necessary product
// information can also be retrieved from your local store.
```

```
$product_factory = new Product();
$vrc = $product_factory->fetchByMerchantProductId('monthlySub');
$prod = $vrc->product();  // assuming here that the return is 200

$item = new AutoBillItem();
$item->setProduct($prod);
// The item was added the same day the AutoBill started,
// so you must specify the start date explicitly.

$migrBill->setItems( [ $item ] );

// This sample creates a new customer Account
// and a new PaymentMethod for every run.
// For actual migration, the Account used here, and its associated
// PaymentMethod, is expected to be already present in CashBox.
$account_factory = new Account();
$vrc = $account_factory->fetchByMerchantAccountId('user_207408');
$acct = $vrc->account();  // assuming that the return is 200

$migrBill->setAccount($acct);

$paymentProcessor = 'Litle';
   // Your merchant account must already have an active
   // routing set to process transactions at Litle.
$paymentProcessorMerchantId = '9104658';

// ************************************************

// Construct the Transaction for the first Billing Cycle to migrate.

$mt0 = new MigrationTransaction();
$mt0->setMerchantTransactionId('MIGR-SAMPL-0-196114'); // each transaction
should have unique ID
$mt0->setAutoBillCycle(0);
$mt0->setType('Recurring');
$mt0->setBillingPlanCycle(0);
$mt0->setMerchantBillingPlanId(billPlanId);
$mt0->setRetryNumber(0);
$mt0->setPaymentMethod(acct.getPaymentMethods()[0]);
$mt0->setAccount(acct);
$mt0->setSalesTaxAddress(acct.getPaymentMethods()[0].getBillingAddress());
$mt0->setPaymentProcessor(paymentProcessor);
$mt0->setPaymentProcessorTransactionId('Litle-' . curTime);
$mt0->setDivisionNumber(paymentProcessorMerchantId);

$mt0->setBillingDate('2013-12-27');

$mt0->setCurrency('USD');

$txTotal = 0.0;  // This must match the total of the items,
          // so we will add to this total as we construct line items.
```

```
$mTxItem00 = new MigrationTransactionItem();
$mTxItem00->
   setItemType(com.vindicia.soap.v5_0.Vindicia.MigrationTransactionItemType
   .RecurringCharge);
$mTxItem00->setSku(prod.getMerchantProductId());
$mTxItem00->setName(prod.getDescriptions()[0].getDescription()
   . ' - includes discounts (if any)');
$discount = 0.15 ; // Customer got 15% discount for this transaction.
$price_array = $prod->getPrices();
$the_price = $price_array[0];
$productPrice = $prod->getAmount();
$discountedProductPrice  = $productPrice - $productPrice * $discount;
$mTxItem00->setPrice($discountedProductPrice);
$txTotal = $txTotal + $discountedProductPrice;
$mTxItem00->setTaxClassification('DC010500');
   // This should be the Avalara tax code associated with this product

$mTxItem00->setServicePeriodStartDate('2013-12-27');
$mTxItem00->setServicePeriodEndDate('2014-01-26');

$mTxTaxItem000 = new MigrationTaxItem();
$mTxTaxItem000->setAmount(7.50);
$mTxTaxItem000->setJurisdiction('STATE_06');
$mTxTaxItem000->setName('CALIFORNIA STATE SALES TAX');

// Let's add the tax to our transaction total.
$txTotal = $txTotal + 7.50;

$mTxTaxItem001 = new MigrationTaxItem();
$mTxTaxItem001->setAmount(1.00);
$mTxTaxItem001->setJurisdiction('COUNTY_085');
$mTxTaxItem001->setName('SANTA CLARA COUNTY SALES TAX');

// Let's add the tax to our transaction total.
$txTotal = $txTotal + 1.00;

$mTxItem00->setMigrationTaxItems( [ $mTxTaxItem000, $mTxTaxItem001 ] );

$mt0->setMigrationTransactionItems( [ $mTxItem00 ] );

// Let's set the transaction total.
$mt0->setAmount($txTotal);  // Total tx amt should be same as.

// Report this transaction in 'captured' status so it can be refunded.
$status00 = new TransactionStatus();
$status00->setStatus('Captured');
$status00->setPaymentMethodType('CreditCard');
$status00->setTimestamp('2013-12-27 01:30:40');
$ccStatus00 = new TransactionStatusCreditCard();
$ccStatus00->setAuthCode('000');
$status00->setCreditCardStatus($ccStatus00);

$mt0->setStatusLog( [ $status00 ] );

// ************************************************
```

```
// Construct the Transaction for the second billing cycle to migrate.

$mt1 = new MigrationTransaction();
$mt1->setMerchantTransactionId('MIGR-SAMPL-1-' . curTime);
$mt1->setAutoBillCycle(1);
$mt1->
   setType(com.vindicia.soap.v5_0.Vindicia.MigrationTransactionType
   .Recurring);
$mt1->setBillingPlanCycle(1);
$mt1->setMerchantBillingPlanId(billPlanId);
$mt1->setRetryNumber(0);
$mt1->setPaymentMethod(acct.getPaymentMethods()[0]);
$mt1->setAccount(acct);
$mt1->setSalesTaxAddress(acct.getPaymentMethods()[0].getBillingAddress());
$mt1->setPaymentProcessor(paymentProcessor);
$mt1->setPaymentProcessorTransactionId('Litle-' . curTime);
$mt1->setDivisionNumber(paymentProcessorMerchantId);
$mt1->setCurrency('USD');
$mt1->setBillingDate('2014-01-27');

$txTotal = 0.0;  // This must match the total of the items.

$mTxItem10 = new MigrationTransactionItem();
$mTxItem10->setItemType('RecurringCharge');
$mTxItem10->setSku($prod->getMerchantProductId());
$discount = 0.1 ;
   // This product got a 10% discount.
   // Specify a price that includes a discount.
$description_array = $prod->getDescriptions();
$the_description = $description_array[0];
$mTxItem10->setName($the_description->description()
   . ' - includes discounts (if any)');
$price_array = $prod->getPrices();
$the_price = $price_array[0];
$productPrice = $the_price->getAmount();
$discountedProductPrice  = $productPrice - $productPrice * $discount;
$mTxItem10->setPrice($discountedProductPrice);
$txTotal = $txTotal + $discountedProductPrice;

$mTxItem10->setTaxClassification('DC010500');
$mTxItem10->setServicePeriodStartDate('2014-01-27');
$mTxItem10->setServicePeriodEndDate('2014-02-26');
// Let's use same period as the Billing Period end.
// Construct tax line items applicable to the above product line item.
$mTxTaxItem100 = new MigrationTaxItem();
$mTxTaxItem100->setAmount(7.50);
$mTxTaxItem100->setJurisdiction('STATE_06');
$mTxTaxItem100->setName('CALIFORNIA STATE SALES TAX');
$txTotal = $txTotal + 7.50;

$mTxTaxItem101 = new MigrationTaxItem();
$mTxTaxItem101->setAmount(1.00);
$mTxTaxItem101->setJurisdiction('COUNTY_085');
$mTxTaxItem101->setName('SANTA CLARA COUNTY SALES TAX');
$txTotal = $txTotal + 1.00;
```

```
$mTxItem10->setMigrationTaxItems( [ $mTxTaxItem100, $mTxTaxItem101 ] );

$mt1->setMigrationTransactionItems( [ $mTxItem10 ] );

$mt1->setAmount($txTotal);

// Report this transaction in 'captured' status so it can be refunded.
$status10 = new TransactionStatus();
$status10->setStatus('Captured');
$status10->setPaymentMethodType('CreditCard');
$status10->setTimestamp('2014-01-27 02:36:57');
$ccstatus10 = new TransactionStatusCreditCard();
$ccstatus10->setAuthCode('000');
$status10->setCreditCardStatus($ccstatus10);

$mt1->setStatusLog( [ $status10 ] );

// Now construct an array of the Transactions to be migrated.
$migrTxs = [ $mt0, $mt1 ];


// Next billing for the AutoBill should happen a
// month from the last billing (2nd transaction in the
// migrated transaction array above) i.e. on Dec. 27, 2013.

// Now make the CashBox SOAP API call to migrate the
// AutoBill along with the Transactions constructed above.

$vr = $migrBill->migrate('2014-01-27', $migrTxs);
if ($vr->getReturnCode() != 200) {
    print('Migrate call failed - return code '
         . vr->getReturnCode()
         . ' and return string '
         . vr->getReturnString());
    print('Soap id ' . $vr->getSoapId());
}
else {
   print('Migration successful. Created autobill id '
. $migrBill->getMerchantAutoBillId()
. ' in CashBox. Its VID: '
. $migrBill->getVID()
. ' . This autobill is for customer Account ID '
       . $migrBill->getAccount()->getMerchantAccountId());
}
```

5.4.3 Migrating an AutoBill During a Free Trial Period

This example demonstrates how to migrate an AutoBill that is currently within a one-month free trial period. Once migrated, CashBox will begin billing when this trial period has completed.

```
$migrBill = new AutoBill();

// The subscription was originally started,
// in the existing system, on Jan. 25, 2014.

$migrBill->setStartTimestamp('2014-01-25');

// Set unique subscription ID.
$migrBill->setMerchantAutoBillId('SampleMigratedAutoBill-ABC123');

// Specify the Billing Plan the migrated AutoBill will be on.
// Make sure the Billing Plan used below is created in CashBox in advance.
// This is a monthly Billing Plan with first cycle marked FREE.
$bp = new BillingPlan();
$bp->setMerchantBillingPlanId('first-free-monthly');
$migrBill->setBillingPlan($bp);

// Specify the Product the migrated AutoBill will be using.
// Make sure the Product used below is created in CashBox in advance.

// First, fetch the Product to get more info about the Product,
// which can be used to fill into the migration transactions.

$product_factory = new Product();
$vr = $product_factory->fetchByMerchantProductId('monthlySub');

// assuming here that the return is 200
$prod = $vr->getProduct();  // The product that the fetch returned.

$item = new AutoBillItem();
$item->setProduct($prod);

$migrBill->items[0] = $item;

// For actual migration the Account used here with its associated
// PaymentMethod is expected to preexist in CashBox.
$account_factory = new Account();
$vrc = $account_factory->fetchByMerchantAccountId('user_xyz123');
$acct = $vrc->account();  // Assuming that the return is 200.

$migrBill->setAccount($acct);

// ************************************************
```

```
// Construct the "Free" Transaction for the first
// billing cycle to migrate.

$mt0 = new MigrationTransaction();
$mt0->setMerchantTransactionId('MIGR-SAMPL-0-987654');
   // Each transaction should have a unique ID.

$mt0->setAutoBillCycle(0);
$mt0->setType('Recurring');
$mt0->setBillingPlanCycle(0);
$mt0->setMerchantBillingPlanId($bp->getMerchantBillingPlanId);
$mt0->setRetryNumber(0);
$pm_array = $acct->getPaymentMethods();
$target_pm = $pm_array[0];
$mt0->setPaymentMethod($target_pm);
$mt0->setAccount($acct);
$mt0->setSalesTaxAddress($target_pm->getBillingAddress());

$mt0->setBillingDate('2014-01-25');
$mt0->setCurrency('USD');
$mt0->setAmount(0.0);
   // This amount must be 0 to make this transaction FREE.

$mTxItem00 = new MigrationTransactionItem();
$mTxItem00->setItemType('RecurringCharge');
$mTxItem00->setSku('FREE');
$mTxItem00->setName('Free Cycle');
$mTxItem00->setPrice(0.0);

$mTxItem00->setServicePeriodStartDate('2014-01-25');
$mTxItem00->setServicePeriodEndDate('2014-02-24');

$mt0->setMigrationTransactionItems([ $mTxItem00 ]);

// Report this transaction in "captured" status.
$status00 = new TransactionStatus();
$status00->setStatus('Captured');
$status00->setPaymentMethodType('CreditCard');
$status00.setTimestamp('2014-01-25 01:30:40');

$mt0->setStatusLog( [ $status00 ] );

// Now construct the array of Transactions to be migrated.
$migrTxs = [ $mt0 ];

// The next billing for the AutoBill should happen a month
// from the last billing (2nd transaction in the migrated
// transaction array above) i.e. on Dec. 27, 2013.
// Now, make the CashBox SOAP API call to migrate the AutoBill
// along with the transactions constructed above.

$vr = $migrBill->migrate('2014-01-25', $migrTxs);
```

```
if ($vr->getReturnCode() != 200) {
   print('Migrate call failed - return code '
          . $vr->getReturnCode()
          . ' and return string '
          . $vr->getReturnString());
   print('Soap id ' . $vr->getSoapId());
}
else {
   print('Migration successful. Created autobill id '
          . $migrBill->getMerchantAutoBillId()
          . ' in CashBox. Its VID: '
          . $migrBill->getVID()
          . ' . This autobill is for customer Account ID '
          . $migrBill->getAccount()->getMerchantAccountId());
   print('Next billing will be on '
          . $migrBill->getNextBilling()
          . ' for $'
          . $migrBill->getNextBilling()->getAmount());

   $transaction_factory = new Transaction();
   $tx_array = $transaction_factory->fetchByAccount($acct, false);

   // Because we migrated only the $0 transaction,
   // there should be only one transaction.
   $the_tx = $tx_array[0];
   $status_log = $the_tx->getStatusLog();
   $the_status = $status_log[0];

   print('Last transaction for this account was for $'
          . $the_tx->getAmount()
          . ' processed on '
          . $the_status->getTimestamp);
}
```

5.5 Using EDD for Recurring Billing

To implement AutoBill-based recurring billing for EDD, construct and populate a `PaymentMethod` object and set it on the `paymentMethod` attribute of the `AutoBill` object.

Note: If you do not specify a `paymentMethod` attribute, CashBox uses the first payment method in the sort order on the associated Account object.

The following example constructs an EDD-based `PaymentMethod` object, and sets it as the payment method for an `AutoBill` object. The example then creates the `AutoBill` object on the Vindicia server by calling `update()` on `AutoBill` with payment method validation turned on. (Note that this example shows a recurring transaction with a full rebill amount.)

```
// Create a payment method object to make the call
$edd_pm = new PaymentMethod();

$edd_pm->setType('DirectDebit');

$dd = new DirectDebit();
$dd->setAccount('8888888888');
$dd->setBankSortCode('12345678');
$dd->setCountryCode('DE');

$edd_pm->setDirectDebit($dd);

$bill_addr = new Address();
$bill_addr->setName('Lutz Haff');
$bill_addr->setAddr1('Leonrodstrasse 57');
$bill_addr->setCity('Munchen');
$bill_addr->setPostalCode('D-80636');
$bill_addr->setCountry('DE');

$edd_pm->setBillingAddress($bill_addr);
$edd_pm->setAccountHolderName('Lutz Haff');
$edd_pm->setCustomerSpecifiedType('EDD');
$edd_pm->setMerchantPaymentMethodId('pmid-edd-342123');

// Create a payment method object to make the call
$autobill = new AutoBill();

$autobill->setPaymentMethod($edd_pm);

// Populate other AutoBill attributes here
…

// Create the autobill while validating the payment method

$autobill->setCurrency('EUR');
// If you are enabling mandate storage for this AutoBill,
// as discussed later in this document, include the
// IP address from which the customer purchased this subscription.
// In some countries, the IP address is part of the
// signature on the mandate.

$autobill->setSourceIp('198.209.56.17');

$validate = true;
$fraudScore = 100 ; // do not want to do risk screening

$response =
    $autobill->update('SucceedIgnore', $validate, $fraudScore);

if($['response['data']->refunds['returnCode'] == 200
        && $response['created'] ){
    print "AutoBill created with VID " .
        $response['data']->autobill->getVID() . "\n";
}
```

Once the `AutoBill` object is in place, the recurring EDD-based transactions it generates determine its status and the entitlements it grants to the associated `Account` object, as follows:

1. CashBox validates (the payment method for) a recurring transaction before submitting it to the payment processor for capture. (For more information, see the `AutoBill.validate` method in the ***CashBox API Guide***.)
   - If the validation fails, CashBox sets the `AutoBill` status to `Hard Error`. This rarely happens if you validate the EDD payment method when creating an `AutoBill` object or when updating its payment method.
   - If the transaction passes the validation, CashBox submits it to the payment processor. The processor checks the account number and bank routing number for the EDD payment method against a negative file. If the transaction passes, CashBox considers the transaction `Authorized`, and the fund withdrawal process begins. CashBox sets the `AutoBill` status to `Good Standing` (or retains that status), and `AutoBill` grants (or continues to grant) the customer entitlements.
2. During the fund deposit process:
   - If the processor receives a ***decline*** return code (a soft error due to insufficient funds, for example), the retry process starts. The payment processor triggers this process according to the retry schedule defined by the merchant. A retry does not affect the `AutoBill` status, which remains `Good Standing`. The rebill `Transaction` moves to the `DepositRetryPending` status, which is internal to CashBox. The customer continues to have the entitlements granted by the `AutoBill` object.
   - If the processor receives a ***reject*** return code, (a hard error because, for example, the customer's bank account has been closed), and you have not provided a retry schedule to the processor, the rebill transaction fails. CashBox sets the transaction status to `Cancelled` and the corresponding `AutoBill` status to `Hard Error`.
   - If CashBox receives ***no response*** from the processor for four banking days, CashBox sets the `Transaction` status to `Captured`. The `AutoBill` object retains its `Good Standing` status until the beginning of the next billing period, and the customer's entitlements remain valid until the next billing date.

5.5.1 Understanding Mandates for Recurring Billing with EDD

Mandates are written agreements from your customers that authorize you to withdraw funds from their bank accounts. Although you can create `AutoBills` without mandates in CashBox, Vindicia recommends that you store mandates, because it is required by law in most countries that support EDD.

To enable mandate storage for an AutoBill paid through EDD:

1. Upload the HTML template of the mandate's general text to Vindicia. Each template must specify the country to which the template applies (see the valid values for `countryCode` in the `DirectDebit` Subobject in the ***CashBox API Guide***), the language the template is in (specify the ISO code for the language), and the template's version number. Contact Vindicia Client Services to upload your mandate template.

   The HTML mandate template contains placeholder tuples (tags), listed in the document ***European Direct Debit for CashBox in The Netherlands, Germany, and Austria: An Overview.*** When creating an instance of the mandate from the template for a subscription, CashBox replaces those tuples with the values from the corresponding `AutoBill` object, then stores the mandate instance with that `AutoBill` object in the Vindicia database.

2. Inform CashBox that you intend to store the mandate for a subscription by including special flags in the form of name–value pairs in the `AutoBill` object before calling `update()` to create the object. Table 5-1 lists the flags.

Table 5-1 `AutoBill` Object Flags for Mandates

Name of Flag	Value
`vin:MandateFlag`	A value of 1 indicates that you would like to store the mandate for the associated `AutoBill` object.
`vin:MandateVersion`	Specifies the mandate instance to be used for the associated `AutoBill` object. **Note:** If you do not specify this field, CashBox uses the most recently added template for the customer's preferred language and country.
`vin:MandateBankName`	Specifies the name of the customer's bank used for the mandate. (Required only in the Netherlands.)

3. Include the source IP address from which the customer made this purchase in the `AutoBill` object. Some countries consider the IP address, coupled with the timestamp for the `AutoBill`'s creation, to be a valid replacement for the customer's signature on the mandate.

   CashBox accepts transactions without IP addresses, to allow for cases in which a paper mandate is kept on file, precluding the requirement for an IP address.

4. Add the name–value pairs to the code.

Enable mandate storage for an existing AutoBill:

(This example expands on the previous.)

```
…
$autobill->setPaymentMethod($edd_pm);

// Set flags in the autobill to enable mandate storage
// You must have uploaded a mandate template of version 1.0
// to Vindicia servers prior to this.

$nv1 = new NameValuePair();
$nv1->setName('vin:MandateFlag');
$nv1->setValue('1');

$nv2 = new NameValuePair();
$nv1->setName('vin:MandateVersion');
$nv1->setValue('1.0');

$nv3 = new NameValuePair();
$nv1->setName('vin:MandateBankName');
$nv1->setValue('Deutsche Bank');

// nameValues attribute is available in Vindicia's AutoBill
// object from API version 3.4 onwards

$autobill->setNameValues(array($nv1, $nv2, $nv3));

// update the AutoBill as shown in the previous example
```

Multiple mandate templates may be uploaded for different languages. When creating a mandate instance to store with an `AutoBill`, CashBox uses the template that matches the `preferredLanguage` attribute of the `AutoBill` object associated with the transaction.

Note:	If you do not specify a value for `preferredLanguage`, CashBox defines the language for the mandate using the country specified in the EDD payment method.

Use the CashBox Portal to view and retrieve mandates. If you have enabled an AutoBill for mandate storage, a link to view the mandate is displayed on the **AutoBill Detail** page on the Portal. Click that link to display the PDF document for the mandate.

5.6 Using PayPal for Recurring Billing

To set up AutoBill-based recurring billing with PayPal, you must be preapproved by PayPal to conduct *reference transactions*. Contact Vindicia Client Services for more information.

Note: Not all merchants will be authorized to obtain the `referenceId` from PayPal. Work with your PayPal representative to obtain the right to use this process.

Set up recurring billing in an AutoBill:

1. If the `PaymentMethod` exists and contains a `ReferenceID` for PayPal, turn off the ***validatePaymentMethod*** flag in the `update` call. (This condition may exist if the `PaymentMethod` was previously validated and approved.)
2. Set the following for the PayPal payment method (`PaymentMethod` object and its `payPal` attribute) on `AutoBill`, or on the `Account` object if the settings are not specified on `AutoBill`:
   a. Set `emailAddress` to the subscriber's email address.
   b. Set `returnUrl` to an existing URL on your site to which the customer will be redirected after a successful authentication with PayPal.
   c. Set `cancelUrl` to an existing URL on your site to which the customer will be redirected after a failed authentication, that is, if PayPal does not authorize the payment information.
3. Call `update()` to create the `AutoBill` object, and return a `TransactionStatus` object. The `TransactionStatus` object must contain a URL in the `redirectUrl` field, to which the customer will be redirected to complete the PayPal payment process. Without completion of this step, recurring billing will not begin.

Retrieve the redirectUrl:

```
$minChargebackProbability = 90;

$duplicateBehavior = 'Fail';

$validatePaymentMethod = 'true';

// Call update to update or create the autobill on Vindicia servers

$response = $autobill->update($duplicateBehavior,
       $validatePaymentMethod, $minChargebackProbability);

if($response['returnCode'] == 200) {
    printLog "AutoBill created \n";

    $txnStatus = $response['data']->refundsauthStatus;
    $redirectUrl = $txnStatus->payPalStatus->redirectUrl
    print "Visit " . $redirectUrl .
        " to complete payment at PayPal site"
}
```

While waiting for the customer to complete payment at PayPal, the `AutoBill` object will have status: `PendingCustomerAction`. The object is dormant and will not perform any billing.

At the PayPal site, the customer logs in and agrees to the contract for recurring billing. When the process is complete, the customer clicks a button that takes them to your success page (`returnUrl`) or failure page (`cancelUrl`). From the success page, make the `finalizePayPalAuth()` call to inform CashBox of successful authorization of the PayPal-based payment method. After receiving the successful authorization, the status of the `AutoBill` object changes to `New`, indicating that the subscription has started and that the `AutoBill` will start billing the customer according to the specified dates.

Finalize the PayPal payment method authorization:

```
$soap_caller = new AutoBill();

// obtain id of the PayPal validation transaction
// from the redirect URL. It is the value associated with name
// 'vindicia_vid'

$payPalTxId = … ;

// if calling from return URL which is reached when the PayPal
// transaction is successfully authorized you should set the
// success input parameter to true

$success = true;
$response =
       $soap_caller->finalizePayPalAuth($payPalTxId, $success);

if($response['data']->refunds['returnCode'] == 200) {
   printLog "PayPal validation transaction successful";
   printLog "- subscription started";
}
```

For every subsequent transaction performed by the `AutoBill` object, no action will be required from the customer. PayPal will notify the customer when the transaction is complete. That notification is in addition to any communication issued by your organization through the CashBox email notification system.

For more information, see the ***Using CashBox with PayPal*** white paper, available from Vindicia Client Services.

6 Working with One-Time Transactions

CashBox Billing Events occur when an AutoBill generated event passes through your payment processor. A Billing Event might be a recurring or one-time transaction, a manually accepted and entered payment, a credit check (for credit cards), a bank withdrawal (EDD), or a customer refund.

One-Time Transactions may be generated for a single item purchase, or for the first in a series of recurring billings. Note that some payment methods, supported by CashBox, do not allow recurring transactions, and allow only one-time transactions.

While AutoBill payments are also processed as CashBox Transactions, this chapter deals specifically with the `Transaction` object, with an emphasis on one-time purchases. For more information on recurring billing, see Chapter 5: Working with AutoBills.

CashBox allows you to issue email notifications when these events occur. For more information, see Chapter 9: Working with Customer Notifications.

6.1 Setting Up Real-Time Billing for One-Time Purchases

CashBox supports real-time billing for one-time purchases. In this case, you must explicitly create a `Transaction` and send it to CashBox for processing. CashBox performs part of this processing synchronously and returns the results to you. For example, for a credit card-based transaction, the payment processor immediately authorizes it and CashBox returns the results.

The `Transaction` object supports multiple API calls for real-time billing. You may set `Transaction` attributes to specify details including line items, prices, total amount, and payment method.

Note: `PaymentMethod: MerchantAcceptedPayment` may not be used for one-time payments.

6.1.1 Monitoring Transaction Status

For all payment methods, the status of the transactions initiated through an `authCapture()` call changes at several points after the call returns. The immediate status returned by the call is simply the initial status that indicates the success or failure of the transaction.

The `Transaction` object supports multiple ID fetch calls to enable you to retrieve transactions from CashBox. To monitor the latest status of a `Transaction` object after the `authCapture()` call is complete, call one of the object's fetch methods, such as `fetchByMerchantTransactionId`. Other calls retrieve transactions in batches. If you are maintaining a local database of transactions, use a batch call to stay in sync with the transaction statuses in CashBox.

6.2 Using Credit Cards for One-Time Transactions

Create, populate, authorize, and capture a Transaction with payment method type: CreditCard:

```
$tx = new Transaction();
$tx->setAmount('9.90');
$tx->setCurrency('USD');

// Merchant transaction id must be unique for each new
// transaction you wish Vindicia to process. If you use an id
// that has been used before, the authCapture() call will
// simply update the corresponding
// existing transaction with new data

$tx->setMerchantTransactionId('txid-123456');

$tx->setTimestamp('2006-09-11T22:34:32.265Z');
$tx->setSourceIp('34.67.89.234');
$tx->setSourcePhoneNumber('650-874-6784');

// Reference an existing account
$account = new Account();
$account->setMerchantAccountId('9876-5432');
$tx->setAccount($account);

// Different shipping address from Account?
$shippingAddress = new Address();
$shippingAddress->setName('Jane Doe');
$shippingAddress->setAddr1('44 Elm St.');
$shippingAddress->setAddr2('Apt 55');
$shippingAddress->setAddr3('');
$shippingAddress->setCity('San Mateo');
$shippingAddress->setDistrict('CA');
$shippingAddress->setPostalCode('94403');
$shippingAddress->setCountry('US');
$shippingAddress->setPhone('650-555-3444');
$shippingAddress->setFax('650-555-3445');

$tx->setShippingAddress($shippingAddress);

// The line items of the transaction
$tx_item = new TransactionItem();
$tx_item->setSku('sku-1234');
$tx_item->setName('Widget');
$tx_item->setPrice('3.30');
$tx_item->setQuantity('3');
$tx->setTransactionItems(array($tx_item));

$paymentMethod = new PaymentMethod();
$paymentMethod->setBillingAddress($address);
$paymentMethod->setType('CreditCard');
```

```
$card = new CreditCard();
$card->setAccount('4444222211113333');
$card->setExpirationDate('xxxxxx'); // Use YYYYMM format for date
$paymentMethod->setCreditCard($card);

$tx->setSourcePaymentMethod($paymentMethod);

// CashBox can send an email notification to the customer
// associated with this transaction, if an email template
// for this is uploaded to the CashBox database

$sendEmailNotification=false;

$response = $tx->authCapture($sendEmailNotification);

if($response['returnCode']==200) {
    // The transaction statuses can be found in statusLog attribute
    // of the Transaction object. This is an array of
    // TransactionStatus objects. The first entry in this array
    // is the latest status of the transaction
    if($tx->statusLog[0]->status=='Authorized') {
        print "Captured\n";
    }
    else if($tx->statusLog[0]->status=='Cancelled') {
        // The transaction did not go through
        print "Declined.
             Reason code received from payment processor: ";
        print $tx->statusLog[0]->status->creditCardStatus->authCode
             . "\n";
    }
}
```

With the `CreditCard` payment method, the `Transaction` status after an `authCapture()` call is `Authorized`. The payment processor has authorized the transaction, and CashBox has marked it for capture with the payment processor. After capture, the status changes to `Captured`. Although this change usually takes less than 24 hours, in most cases you can assume that if `Authorized` is returned as the status, CashBox will capture the transaction.

6.2.1 Verifying AVS and CVN for One-Time Transactions

To verify security codes for real-time billing, set the name–value pair for the `PaymentMethod` object. See Section 5.1.3: Verifying AVS and CVN for Recurring Billing for details.

The `authCapture()` method authorizes a transaction and marks it for capture by a CashBox back-end batch process. `auth()` also authorizes a transaction and returns the authorization results, including the results from address and security-code verifications, but does not mark the transaction for automatic capture. To capture the transaction using `auth()`, you must call `capture()` before the authorization period expires. (The authorization period varies by payment processor, but is typically three days for real-time transactions, and seven days for recurring transactions.)

If you have shippable merchandise or want an additional risk review process before deciding whether to capture a transaction, make two separate SOAP calls. (When calling `authCapture()` to immediately mark a transaction for capture, the `capture` operation might occur before you have a chance to call `cancel()`.)

Payment processors recommend authorizing a transaction when the order is placed, and capturing it only after you have shipped the merchandise. To do this, make a live `auth()` call when a customer places an order. After the product is authorized or shipped, use a `capture()` call in batch mode to process an array of transactions at the end of a day, or more frequently, depending on the volume of your transactions.

The `authCapture()` call may save time if your one-time transactions are for relatively small amounts and do not require physical shipments. However, Vindicia recommends that you screen transactions for fraud risk with the `score` call before calling `authCapture`.

To verify address and security codes, retrieve AVS and CVN code responses, then examine the `TransactionStatus` object in the `Transaction`. Look at the most recent entry in the `statusLog` array of the `Transaction` object, located in the first position in the array.

If you specify a `minChargebackProbability` of less than 100 when calling `auth()`, CashBox evaluates the risk score for the transaction, and includes the results in the return parameters `score` and `scoreCodes` (codes and corresponding messages that explain the score). If the score evaluates higher than `minChargebackProbability`, no authorization with the payment processor is performed, saving you the cost of authorizing a transaction with the payment processor if it is determined to be potentially fraudulent.

Note:	To ignore the fraud score, set `minChargebackProbability` to 100.

Verify for real-time billing:

```
$tx = new Transaction();
$tx->setAmount('9.90');
$tx->setCurrency('USD');

$tx->setMerchantTransactionId('txid-123456');

$paymentMethod = new PaymentMethod();
$paymentMethod->setBillingAddress($address);
$paymentMethod->setType('CreditCard');

$card = new CreditCard();
$card->setAccount('4444222211113333');
$card->setExpirationDate('xxxxxx'); // Use YYYYMM format for date
$paymentMethod->setCreditCard($card);

$nv = new NameValuePair();
$nv->setName("CVN");
$nv->setValue("123"); // card security code provided by customer

// set the card security code inside the payment method
$paymentMethod->setNameValues(array($nv));
$tx->setSourcePaymentMethod($paymentMethod);

// set other transaction attributes here
$sendEmailNotification=false;
$minChargebackProbability = 100; // not doing risk screening

$response = $tx->auth($minChargebackProbability, $sendEmailNotification);

if($response['returnCode']==200) {

   if($tx->statusLog[0]->status=='Authorized') {
      print "Card approved.
           Checking address and security code responses \n";
      $txnStatus = $tx->statusLog[0];
           // latest transaction status

      if ($txnStatus->vinAVS == "FullMatch"
           || $txnStatus->vinAVS == "PartialMatch")
      {
      print " Address verified \n";
      }
   else {
      $avs = $txnStatus->creditCardStatus->avsCode;

      // work with the AVS response code here. If there is a
      // certain value of the AVS code which is not acceptable,
      // then cancel the transaction

        if ($avs == "xyz") {
           $soapCallerTxn = new Transaction();
           $soapCallerTxn->cancel(array($tx));
           exit();
        }
   }
```

```
    // AVS is OK, now check the response to security code
    // verification

    $cvn = $txnStatus->creditCardStatus->cvnCode;

    // work with the AVS response code here. If a certain
    // value of the CVN response code is not acceptable, then
    // cancel the transaction

    if ($cvn == "abc") {
        $soapCallerTxn = new Transaction();
        $soapCallerTxn->cancel(array($tx));
        exit();
    }
    else if($tx->statusLog[0]->status=='Cancelled') {
        // The transaction did not go through
        print "Declined. Reason code received from
             payment processor: ";
        print $tx->statusLog[0]->status->creditCardStatus->
             authCode . "\n";
    }
}
```

Capture multiple authorized Transactions in a batch:

```
$tx_soap = new Transaction();

$txn1 = new Transaction();

$txn1->setMerchantTransactionId('id1');
       // this is a previously authed transaction

$txn2 = new Transaction();

$txn2->setMerchantTransactionId('id2');
       // this is a previously authed transaction

$response = $tx_soap->capture(array($txn1, $txn2));

if($response['returnCode']==200) {
    // capture success
}
```

6.2.2 Calling the auth and capture Methods Separately

If a Transaction object's payment method is credit card, you may also set up real-time billing by calling the auth() and capture() methods separately.

To specify a risk score threshold (also known as chargeback probability), call auth(). If the score evaluates to a value higher than the threshold, CashBox does not authorize the transaction. auth() also allows you to examine the responses to the AVS and CVN verifications returned by the payment processor. If the risk score exceeds your allowable value, or if the AVS or CVN return score is not acceptable, do not capture the transaction, and set its status to Cancelled by calling Transaction.cancel().

For more detail on AVS and CVN Return Codes, please work with your Vindicia Client Services representative.

To capture an authorized transaction before the authorization period expires, call capture(). (The authorization period varies by card issuer, but is typically seven days.)

Capture an authorized Transaction:

```
$tx = new Transaction();

// populate the Transaction object as illustrated above
// for credit card based authCapture call

$sendEmailNotification = false;
$minChargebackProbability = 100; // not doing risk score based rejection
$response = $tx->auth($minChargebackProbability,
       $sendEmailNotification);

if($response['returnCode']==200) {

    if($tx->statusLog[0]->status=='Authorized') {
       // Check AVS match. vinAVS attribute provides an abstraction
       // for AVS response codes from various payment processors based
       // on CashBox's interpretation. If vinAVS is set to NoOpinion,
       // check $tx->statusLog[0]->status->creditCardStatus->avsCode
       // for the response received from your payment processor

       if ($txnStatus->vinAVS == "FullMatch"
            || $txnStatus->vinAVS == "PartialMatch") {
           print " Address verified \n";
       }
       else {
            $avs = $txnStatus->creditCardStatus->avsCode;
            // work with the AVS response code here. If a return value for
            // the ACS response code is not acceptable,
            // cancel the autobill
            if ($avs == "xyz") {
                 $autobill->cancel(true, true)
            // immediate disentitle and forced cancellation
                 exit();
            }
       }
```

```
            // AVS is OK, now check the response to security code verification

            $cvn = $txnStatus->creditCardStatus->cvnCode;
            // work with the CVN response code here. If a return value for
            // the CVN response code is not acceptable, cancel the autobill
            if ($cvn == "abc") {
                 $autobill->cancel(true, true)
                      // immediate disentitle and forced cancellation
                 exit();
            }
            else {
                 $soapTxn = new Transaction();
                 $txnBatchToCancel = array($tx);
                 $soapTxn.cancel($txnBatchToCancel);
            }
        }
        else if($tx->statusLog[0]->status=='Cancelled') {
            // The transaction did not go through
            print "Declined. Reason code received from payment
                 processor: ";
            print $tx->statusLog[0]->status->creditCardStatus->authCode . "\n";
        }
    }
```

For more information, see Section 18: The Transaction Object in the ***CashBox API Guide***.

6.3 Using Carrier Billing for One-Time Transactions

CashBox supports BOKU as a payment processor for Carrier Billing. Work with your BOKU representative to define your pricing structures, and to configure your account as a sub-merchant to Vindicia.

Before processing BOKU Transactions, you must create one or more services on the BOKU website which define your pricing schedule(s). When constructing these services, the "Forward To After Purchase" and "Forward To After Failed Purchase" attributes for each service must point to URIs at your site. The "Callback Url" must point to Vindicia. (Work with your Vindicia Client Services representative to define this URL.)

BOKU-based real-time transactions use the following payment flow:

1. When a customer clicks the BOKU button on your site, create a `Transaction` object that specifies `CarrierBilling` as the payment method, and make a `Transaction.authCapture` call.
2. When that call returns, examine the status of the returned `Transaction` object. If the status is not a failure (`Cancelled`), it will be `AuthorizedPending`, which means that the Transaction is in the CashBox and BOKU systems, and that it requires further action from the customer for completion.
3. The returned `Transaction` object will also include a `TransactionStatusCarrierBilling` subobject, which will include a `buyUrl` which the customer should be presented (or redirected to) in order to complete the transaction.

> **Note:** This `Transaction` includes the BOKU Transaction ID in the `nameValues` array (`name` = `provider_trx_id`), which will prove useful when correlating Transaction data between your site and BOKU/Vindicia.

4. Once the customer has followed the `buyUrl`, and completes the BOKU payment, they will be redirected to the "Forward To Url After Purchase" URL that you included in your BOKU Service Configuration (or the "Forward To Url After Failed Purchase" URL if the payment attempt fails).
5. You may verify the status of the `Transaction` by re-retrieving the CashBox `Transaction` object, and verifying that the status is now `Captured`. (Note that there may be a delay between the time the payment process completes, and the time that BOKU notifies Vindicia about the status of the payment attempt.) You may also perform a `verify-trx-id` `dataRequest` call, which will retrieve the Transaction status information directly from BOKU (see below for details).

6.3.1 BOKU Static Pricing Transactions

To use BOKU's Static Pricing, create a Service ID on the BOKU website before creating a Transaction. use the CashBox `setMerchantServiceIdentifier` call to pass in the corresponding Service ID.

Create a Transaction using BOKU Static Pricing:

```
$txn = new Transaction;
//Populate this Transaction as shown in previous examples.

$criteria = new Criteria();
$criteria->setCurrency('USD');
$criteria->setCountryCode('US');
$criteria->setStaticPriceIncSalesTax(1.00);
$criteria->setMerchantServiceIdentifier('14246ab82c24e44bf9862406');

$paymentProvider = new PaymentProvider();
$paymentProvider->setName('BOKU');
$criteria->setPaymentProvider($paymentProvider);

$carrierBilling = new CarrierBilling();
$carrierBilling->priceCriteria($criteria);

$paymentMethod = new PaymentMethod();
$paymentMethod->setType('CarrierBilling');
$paymentMethod->setCarrierBilling($carrierBilling);

$txn->setSourcePaymentMethod($paymentMethod);

$response = $txn->authCapture();
if($response['returnCode'] ==200)
{
    if($tx->statusLog[0]->status=='AuthorizedPending')
    {
        print "Successful\n";
        display(print txn->statusLog[0]->status->
             carrierBillingStatus->buyUrl);
    }
}
else if($tx->statusLog[0]->status=='Cancelled')
{
    // The transaction did not go through
}
```

6.3.2 BOKU Dynamic Pricing Transactions

BOKU also allows Dynamic Pricing, which allows you to define a target price, and an allowable deviation from that price.

The following example specifies a target price of USD 10.00, and a desired country of BG (with an allowed dynamic deviation of 50%). Note that the price (10.00) is shown in a floating point format, indicating dollars and cents, rather than the BOKU "fractional amount" format (1000).

Create a Transaction using BOKU Dynamic Pricing:

```
$criteria = new Criteria();
$criteria->setCurrency('USD');
$criteria->setCountryCode('BG');
$criteria->setMerchantServiceIdentifier('14246ab82c24e44bf9862406');
$criteria->setDynamicDeviation(50);
$criteria->setPricePointDeviationPolicy('HiPreferred');
$criteria->setDynamicMatch(11);
$criteria->setDynamicTargetPrice(10.00);
$criteria->setPaymentProvider($paymentProvider);
```

BOKU also allows you to define static service tables, from which you may reference a defined dynamic price using its "row ref."

Create a Transaction using a BOKU service table:

```
$criteria = new Criteria();
$criteria->setMerchantServiceIdentifier('140ba94f2c24e44b5cb85730');
$criteria->setStaticSelectionRowRef(1);
$criteria->setCountryCode('NZ');
$criteria->setPaymentProvider($paymentProvider);
```

6.3.3 Using CashBox to query BOKU

CashBox also provides a “data pipe” which allows users to perform the following queries directly against the BOKU web site:

price

service-prices

lookup

verify-trx-id

Create a simple BOKU price query:

```
$provider = new paymentProvider();
$rc = $provider->dataRequest('price',
    [
    NameValuePair->new(name  => 'reference-currency',
        value => 'USD'),
    NameValuePair->new(name  => 'service-id',
        value => '140ba94f2c24e44b5cb85730')
    ]
);

// The return from this call (as well as all data request calls)
// includes two components:
$rc->request  //Contains the Vindicia formatted request sent to BOKU.
$rc->response //Contains BOKU's response
```

Create a BOKU price request using dynamic pricing criteria:

```
$provider = new paymentProvider();
$rc = $provider->dataRequest('price',
    [
    NameValuePair->new(name  => 'reference-currency',
        value => 'USD'),
    NameValuePair->new(name  => 'service-id',
        value => '140ba94f2c24e44b5cb85730'),
    NameValuePair->new(name  => 'dynamic-price-mode',
        value => 'price'),
    NameValuePair->new(name  => 'dynamic-deviation',
        value => 20),
    NameValuePair->new(name  => 'dynamic-deviation-policy',
        value => 'hi-preferred'),
    NameValuePair->new(name  => 'dynamic-match',
        value => 0),
    NameValuePair->new(name  => 'currency',
        value => 'USD'),
    NameValuePair->new(name  => 'target',
        value => 1000)
    ]
);
```

This request would produce a BOKU query similar to the following:

```
[BOKU URL]?action=price&service-id=14246ab82c24e44bf9862406
       &dynamic-price-mode=price&dynamic-deviation=20
       &dynamic-deviation-policy=hi-preferred
       &dynamic-match=0&currency=USD&target=1000
```

Create a BOKU service-prices request:

Service-prices requests allow you to restrict output by (service table) row and country. The following example restricts output to row: 2, and country: New Zealand.

```
$provider = new paymentProvider();
$rc = $provider->dataRequest('service-prices',
    [
    NameValuePair->new(name  => 'service-id',
        value => '140ba94f2c24e44b5cb85730'),
        NameValuePair->new(name  => 'country',
        value => 'NZ'),
        NameValuePair->new(name  => 'row-ref',
    ]
);
```

Create a "lookup" data request to determine the country associated with an IP address:

```
$provider = new paymentProvider();
$rc = $provider->dataRequest('lookup',
    [
    NameValuePair->new(name  => 'ip-address',
        value => '23.11.248.110')
    ]
);
```

Check the status of a BOKU Transaction:

```
$provider = new (paymentProvider();
$rc = $provider->dataRequest('verify-trx-id',
    [
    NameValuePair->new(name  => 'trx-id',
        value => [THE ID ASSIGNED TO THIS TRANSACTION BY BOKU])
    ]
);
```

6.4 Using Boleto Bancario for One-Time Transactions

For the payment method Boleto Bancário, the `Transaction` status after an `authCapture()` call is `Authorized`. That means that CashBox has validated the *fiscal number* and will prepare the transaction for the payment processor. After the payment processor has accepted the fiscal number in the payment method, the transaction status changes to `AuthorizedPending`. In response, the payment processor returns a URL in the `TransactionStatus` object.

Send the customer this URL, which points to further instructions from the payment processor for completing the transaction. When the transaction is complete, the payment processor notifies CashBox, which updates the status to `Captured` or `Cancelled`, depending on the success or failure of the transaction.

Create a Transaction with Boleto Bancário as the payment method, and set the fiscal number:

```
$txn = new Transaction();

// populate the transaction as shown in the previous example
// When associating a customer account with this transaction,
// ensure that the account has language preference indicated.
// This will set the language used in the payment instructions
// displayed to the customer
$tx->setAccount($account);

$paymentMethod = new PaymentMethod();

// For Boleto payment make sure country is specified in the address

$paymentMethod->setBillingAddress($address);

$paymentMethod->setType('Boleto');
$blt = new Boleto();
$blt->setFiscalNumber('123456789');
$paymentMethod->setBoleto($blt);

$tx->setSourcePaymentMethod($paymentMethod);
$sendEmailNotification=false;

$response = $tx->authCapture($sendEmailNotification);

if($response['returnCode'] ==200) {
    if($tx->statusLog[0]->status=='AuthorizedPending') {
        print "Successful\n";
        display(print $tx->statusLog[0]->status->boletoStatus->uri);
    }
    else if($tx->statusLog[0]->status=='Cancelled') {
        // The transaction did not go through
    }
}
```

Note	For Boleto Bancário, be sure to specify the country in the billing address, and the language preference in the customer account. Those two attributes determine the language for CashBox customer notifications (for payment instructions, for example).

6.5 Using ECP for One-Time Transactions

For the ECP payment method, the status of a transaction after an `authCapture()` call is `Authorized`. The payment processor has performed a real-time validation of the payment information to ensure, for example, that the bank routing number is not blacklisted. CashBox then submits the transaction to the payment processor for further processing (deposit or withdrawal from the specified bank), and changes the status to `AuthorizedPending`, to indicate that processing of the transaction has begun.

Six banking days must elapse before CashBox sets the status to `Captured`. If, during that time, CashBox receives notice (by a reason code) from the payment processor that the transaction failed, CashBox changes the transaction status to `Cancelled`.

If the reason code from the payment processor indicates that there will be an internal retry of the transaction, CashBox changes the transaction status to `RetryPending`. The retry date depends on the retry schedule that the payment processor has previously defined with you according to your division ID. (Be sure to provide Vindicia with your division ID's retry schedule.)

If CashBox does not receive any decline codes during the six banking days after the retry, CashBox sets the transaction status to `Captured`.

Create a Transaction with ECP as the payment method:

```
$txn = new Transaction();

// populate the transaction as shown in the previous example
$paymentMethod = new PaymentMethod();
$paymentMethod->setBillingAddress($address);
$paymentMethod->setType('ECP');

$ecp = new ECP();

// specify account number where funds will be with withdrawn from
$ecp->setAccount('123456789');

// specify bank routing number
$ecp->setRoutingNumber('3409284043');
$ecp->setAccountType('ConsumerChecking');
$paymentMethod->setECP($ecp);

// If this is an inbound payment (a withdrawal from a
// specified bank account and deposit into the merchant's
// account), set the source payment method in the transaction.
// For paying out (a deposit into a specified bank account and
// withdrawal from the merchant's bank account), set the
// destinationPaymentMethod attribute of the transaction

$tx->setSourcePaymentMethod($paymentMethod);
$tx->setEcpTransactionType('Inbound');

$sendEmailNotification = false;
$response = $tx->authCapture($sendEmailNotification);
```

```
if($response['returnCode'] ==200) {
    if($tx->statusLog[0]->status=='Authorized') {
        print "Successful\n";
    }
    else if($tx->statusLog[0]->status=='Cancelled') {
        // The transaction did not go through
        print "Declined. Reason code from payment processor: ";
        print $tx->statusLog[0]->status->ecpStatus->authCode . "\n";
    }
}
```

6.5.1 Creating Outbound Payment Transactions with ECP

The majority of transactions, real-time or recurring, processed by CashBox are inbound, as they originate with your customers, and are inbound to your bank. For the ECP payment method, CashBox also supports real-time outbound transactions, where money is withdrawn from your bank account and deposited into someone else's account. This may result from payments to customers, or payments to business partners and vendors.

Note: Outbound ECP support is available *only* to clients currently using Chase Paymentech as their processor.

When creating an outbound ECP `Transaction` object:

1. Set the `destPaymentMethod` attribute with an ECP-based `PaymentMethod` object that contains information on the payee's bank account. Leave the `sourcePaymentMethod` attribute unspecified.
2. Set the `ecpTransactionType` attribute to the value `Outbound`.
3. Point the `account` attribute to an `Account` object that contains the payee's information.

```
$txn = new Transaction();

// populate the transaction as shown in the previous example

$paymentMethod = new PaymentMethod();
$paymentMethod->setBillingAddress($address);
$paymentMethod->setType('ECP');

$ecp = new ECP();

// specify the account number where funds will be deposited
// (payee's account)
$ecp->setAccount('123456799');

// specify bank routing number (payee's bank)
$ecp->setRoutingNumber('3409284044');
$ecp->setAccountType('ConsumerChecking');
$paymentMethod->setECP($ecp);
```

```
// Since this is an outbound payment i.e. a deposit into
// payee's bank account and withdrawal from merchant's bank
// account, set the destPaymentMethod attribute of the transaction

$tx->setDestPaymentMethod($paymentMethod);
$tx->setEcpTransactionType('Outbound');
$sendEmailNotification = false;
$response = $tx->authCapture($sendEmailNotification);

if($response['returnCode']==200) {
    if($tx->statusLog[0]->status=='Authorized') {
        print "Successful\n";
    }
    else if($tx->statusLog[0]->status=='Cancelled') {
        // The transaction did not go through
        print "Declined. Reason code received from payment processor: ";
        print $tx->statusLog[0]->status->ecpStatus->authCode . "\n";
    }
}
```

The status changes on outbound transactions are similar to those for ECP inbound transactions. When the status changes to `Captured`, assume that the outbound payment is complete.

6.6 Using EDD for One-Time Transactions

For real-time EDD-based billing, construct a one-time `Transaction` object and call either the `auth()` or `authCapture()` method on it. If you call `auth()`, capture all the authorized transactions by making the batch `capture()` call later.

If a transaction's source payment method is of type Direct Debit (DD), that is, if the `sourcePaymentMethod` attribute of the `Transaction` object is set to a `PaymentMethod` object that uses EDD, the transaction will process through the following status cycle:

1. CashBox assigns the transaction an immediate status of `Authorized`, indicating that both CashBox and the payment processor have performed a real-time validation of the payment information, and verified that the bank sort code and account number are not blacklisted. The processor has accepted the transaction, and the deposit process can begin. At this time, no funds are transferred.

2. The payment processor submits the transaction to the payment network for continued processing (withdrawal from the specified bank). The transaction status in CashBox changes to `AuthorizedPending`, indicating that the deposit process has started.

3. After four European banking days have elapsed, CashBox sets the transaction status to `Captured`. Note that the number of banking days varies between payment processors.

   During the four European banking days:

   - If CashBox is notified by the payment processor that the transaction failed, that is, received a hard-error reason code from the processor, CashBox changes the transaction status to `Cancelled`. (For example, if the account specified in the payment method does not exist at the bank in question.)
   - If the payment processor sends CashBox a soft-error reason code that indicates that there will be an internal retry of the transaction (for example, due to insufficient funds), CashBox changes the transaction's status to `DepositRetryPending`. The processor determines when to retry the transaction according to the retry schedule defined by you for your division ID registered with the processor. If the processor does not have such a schedule on file, it hard-fails the transaction, and returns a hard-error reason code. In response, CashBox changes the transaction status to `Cancelled`.

   If during the four European banking days subsequent to the retry attempt, CashBox does not receive any hard or soft error codes from the processor, CashBox sets the transaction status to `Captured`.

Note	Be sure to send Vindicia your retry schedule by division ID.

The following example constructs a one-time `Transaction` object and calls `authCapture()` on it to process the transaction. This example also verifies that the processor has accepted the transaction for deposit, by ensuring that the immediate status of the transaction after the call is `Authorized`.

Use EDD for a one-time transaction:

```
$txn = new Transaction();

// Populate the transaction with other attributes such
// as account, transaction items, and amount.

// Assume that there is an existing Account with
// the merchantAccountId specified. If mandate storage
// is required for this transaction, this
// account must have a preferred language setting on it.

$acct = new Account();
$acct->setMerchantAccountId('lhaff1');

$txn->setAccount($acct);
$txn->setAmount(34.99);
$txn->setCurrency('EUR');
$txn->setMerchantTransactionId('MRCH-3402284');

// If you are enabling mandate storage for this transaction,
// include the IP address from which the purchase was made.
// In some countries the IP address is part
// of the signature on the mandate.

$txn->setSourceIp('198.209.56.17');

$txItem = new TransactionItem();
$txItem->setSku('5492');
$txItem->setName('Online video access');
$txItem->setPrice(34.99);
$txItem->setQuantity(1);
$txn->setItems(array($txItem));

// set EDD as source payment method

$paymentMethod = new PaymentMethod();

$bill_addr = new Address();
$bill_addr->setName('Lutz Haff');
$bill_addr->setAddr1('Leonrodstrasse 57');
$bill_addr->setCity('Munchen');
$bill_addr->setPostalCode('D-80636');
$bill_addr->setCountry('DE');

$paymentMethod->setBillingAddress($bill_addr);
$paymentMethod->setType('DirectDebit');

$dd = new DirectDebit();

// specify the account number from which funds will be withdrawn
$dd->setAccount('8888888888');

// specify bank sort code of the bank from which funds will
// be withdrawn
$dd->setBankSortCode('12345678');

$dd->setCountry('DE');
   // needed for Vindicia's internal validation

$paymentMethod->setDirectDebit($dd);

// If this is an inbound payment, i.e. a withdrawal from a
```

```
// specified bank account, and a deposit into the
// merchant's account, set the source payment method in the
// transaction.
// Outbound payments (from the merchant's bank account to the
// customer's), defined by setting the destination
// payment method in the transaction, are not supported.

$tx->setSourcePaymentMethod($paymentMethod);

$sendEmailNotification = false;

$response = $tx->authCapture($sendEmailNotification);

if($response['returnCode']==200) {
    if($tx->statusLog[0]->status=='Authorized') {
        print "Successful\n";
    }
    else if($tx->statusLog[0]->status=='Cancelled') {
        // The transaction did not go through
        print "Declined. Reason code received from payment processor: ";
        print $tx->statusLog[0]->status->directDebitStatus
             ->authCode . "\n";
    }
}
```

After initial authorization of the real-time transaction, monitor the transaction's subsequent status changes by either looking it up on the CashBox Portal, or by calling one of the `Transaction` object's `fetch` methods. For details, see Section 18: The Transaction Object in the ***CashBox API Guide***.

Note	Outbound payment transactions are those in which you have set the `destPaymentMethod` attribute to pay your customers. EDD does not support outbound transactions.

6.6.1 Understanding Mandates for Real-Time Billing with EDD

Mandates may be used for real-time billing with EDD. While recurring billing associates the mandate with an `AutoBill` object, real-time billing associates a mandate with a `Transaction` object.

For more information on working with mandates, see Section 5.5.1: Understanding Mandates for Recurring Billing with EDD.

To enable mandate storage for a real-time transaction:

1. Upload the HTML template of the mandate's general text to the Vindicia servers.
2. Inform CashBox that you intend to store the mandate for a transaction by including special flags in the form of name–value pairs in the `Transaction` object before calling `auth()` or `authCapture()`.

The following table lists the flags.

Table 6-1 `Transaction` Object Flags

Name of Flag	Value
`vin:MandateFlag`	A value of 1 indicates that you would like to store the mandate for the associated `Transaction` object.
`vin:MandateVersion`	Specifies the mandate instance to be used for the associated `Transaction` object. If you do not specify this field, CashBox uses the most recently added template for the customer's preferred language and country.
`vin:MandateBankName`	Specifies the name of the customer's bank used for the mandate. (Required only in the Netherlands.)

3. Include the source IP address from which the customer made this purchase in the `Transaction` object.
4. Add the name–value pairs to the `Transaction` object before calling `authCapture()` on it.

Enable mandate storage for a transaction:

```
$tx->setSourcePaymentMethod($paymentMethod);

// Set flags in the transaction to enable mandate storage
// You must have uploaded a mandate template of version 1.0
// to Vindicia servers prior to this.

$nv1 = new NameValuePair();
$nv1->setName('vin:MandateFlag');
$nv1->setValue('1');

$nv2 = new NameValuePair();
$nv1->setName('vin:MandateVersion');
$nv1->setValue('1.0');

$nv3 = new NameValuePair();
$nv1->setName('vin:MandateBankName');
$nv1->setValue('Deutsche Bank');

$tx->setNameValues(array($nv1, $nv2, $nv3));

// authCapture the transaction as shown in the previous example.
```

When creating a mandate instance to store with a transaction, CashBox uses the template that matches the `preferredLanguage` attribute of the `Account` object associated with the transaction.

Note: If you do not specify a value for `preferredLanguage`, CashBox defines the language for the mandate using the country specified in the EDD payment method.

Use the CashBox Portal to view and retrieve mandates. If you have enabled a transaction for mandate storage, a link to view the mandate is displayed on the **Transaction Detail** page. Click that link to display the PDF of the mandate.

6.7 Using PayPal for One-Time Transactions

For the PayPal payment method, the status of a transaction after an `authCapture()` call is `AuthorizationPending`. The payment flow for PayPal-based real-time transactions proceeds as follows:

1. When a customer clicks the PayPal button on your site, create a `Transaction` object that specifies PayPal as the payment method, and make a `Transaction.authCapture()` call.
2. When that call returns, examine the status of the returned `Transaction` object. If the status is not a failure (`Cancelled`), it is `AuthorizationPending`, which means that the transaction is in the CashBox and PayPal systems, and that it requires further action from the customer for completion.
3. PayPal notifies CashBox of the successful creation of the Transaction by issuing a PayPal token, which keeps the transaction valid for the next few hours.
4. The returned `Transaction` object contains a PayPal-specific status along with a URL, which contains the token information. Redirect the customer to that URL to complete PayPal's payment process.
5. Depending on the customer's success or failure in completing the payment process, PayPal redirects the customer to a CashBox landing page, along with an indication of whether the payment succeeded or failed. That landing page in turn redirects the customer to a success or failure URL on your site. (Provide CashBox the success and failure URLs as attributes `returnUrl` and `cancelUrl` of the PayPal payment method for the transaction.) From this page, make a call to CashBox to finalize the PayPal authorization so that CashBox can update the status of the transaction. This call requires you to pass in the ID of the transaction, which you can find in the redirected URL. It is value-associated with name: `vindicia_vid` in the redirect URL.

Use PayPal for a one-time transaction:

```
$tx = new Transaction();

// populate the transaction as shown in earlier examples

$paymentMethod = new PaymentMethod();
$paymentMethod->setType('PayPal');

$payPal = new PayPal();

// request a ReferenceId from PayPal
$payPal->setRequestReferenceId('true');

// Set the URL to which the customer will be redirected after
// completing the payment process at PayPal's site, and
// returning to Vindicia's landing page.

$payPal->setReturnUrl('http://myshoppingcart.merchant.com');

// specify the bank routing number
$payPal->setCancelUrl('http://tryagain.merchant.com');

$paymentMethod->setPayPal($payPal);

$tx->setSourcePaymentMethod($paymentMethod);
```

```
$sendEmailNotification = false;
$response = $tx->authCapture($sendEmailNotification);

if($response['returnCode']==200) {

    if($tx->statusLog[0]->status=='AuthorizationPending') {
        $payPalUrl = $tx->statusLog[0]->payPalStatus->redirectUrl;

        // send customer to this URL for completion of payment
        // formalities at PayPal's site
    }
    else if($tx->statusLog[0]->status=='Cancelled') {
        // The transaction was not accepted by PayPal
    }
}
```

After successfully completing the payment process, the customer is redirected to the return URL in the PayPal-based `PaymentMethod` object. From this page, finalize the Transaction to update its status in CashBox.

```
$soap_caller = new Transaction();

// obtain id of the PayPal transaction
// from the redirect URL. It is the value associated with name
// 'vindicia_vid'

$payPalTxId = … ;

// if calling from return URL which is reached when the PayPal
// transaction is successfully authorized you should set the
// success input parameter to true, from the cancelUrl it should
// be set to false. Let's assume success here:

$success = true;
$response =
    $soap_caller->finalizePayPalAuth($payPalTxId, $success);

if($response['returnCode'] == 200) {
       printLog "Transaction authorized";
}
```

CashBox updates the Transaction status to `Authorized`, which changes to `Captured` after CashBox has finished processing this and other PayPal transactions in a batch.

Note: If you request that PayPal return a `referenceId` for the Transaction, its `paymentMethod` may be used for recurring payments. For more information, see Section 5.6: Using PayPal for Recurring Billing, and the ***Using CashBox with PayPal*** white paper, available from Vindicia Client Services.

6.8 Recording a Payment Manually

Entering a payment manually allows a merchant to enter a transaction that occurs outside the CashBox automated process. This may be used to enter cash or check payments made in person to the merchant, goods or services accepted in trade for an outstanding invoice, or any other payment method the merchant allows.

CashBox offers two ways to enter a payment manually though the API: using `Account.makePayment`, and using `AutoBill.makePayment`. (You may also enter a payment through the CashBox Portal.)

A merchant-entered payment is applied to outstanding AutoBills with a `PaymentMethod` of "Merchant Recorded Payment." Unless otherwise specified by the Merchant, CashBox credits the Account's AutoBills as follows:

- A manually entered payment is applied first to the oldest outstanding AutoBill.
- Any remaining monies are applied to the next oldest AutoBill, until the payment is exhausted, or all AutoBills have been paid.
- Any monies left after all AutoBills have been paid appear as a credit to the account.

Use the `makePayment` method on the `AutoBill` object to apply a payment directly to an outstanding AutoBill. To make a payment to the oldest open invoice, use `Account.makePayment` instead.

If a payment fails at the financial institution, or if you wish to reverse a payment for other reasons, use `reversePayment` on the corresponding object to reverse a payment entered using `makePayment`.

Record a payment manually:

```
$autobill = new AutoBill();
$autobill->setMerchantAutoBillId($abID);      // for some $abID

$pm = new PaymentMethod();
$pm->setType('MerchantAcceptedPayment');
$pm->setMerchantPaymentMethodId('macc cash '. $time);

$macc = new MerchantAcceptedPayment();
$macc->setAmount(4.50);
$macc->setCurrency('USD');
$macc->setTimestamp($now);
$macc->setPaymentId('macc cash ' . $time);
$macc->setNote('cold hard cash');

$pm->setMerchantAcceptedPayment($macc);

$pm->update(
    true,       // validate
    0,          // chargeback probability
    false,      // replace on all AutoBills
    null,       // ip
    null,       // AVS
    null        // CVN
);
```

```
$response = $autobill->makePayment(
    $pm,
    null,       // amount - see $macc
    null,       // currency - see $macc
    'inv-bac',  // invoice id
    null,
    null,
    '$4.50 in cold hard cash'
);

// check $response
```

6.9 Importing Transactions from other Billing Systems to CashBox

Use `Transaction.migrate` to migrate historic Transaction information into CashBox. Each `MigrationTransaction` included in the `migrate` call will result in the creation of a `Transaction` that can be operated on (`fetch`, `refund`, and etc.) as if it had originated in CashBox. Note, however, that some operations (`refund`) require that the `Transaction` be in one of the following states:

- `Captured`
- `Refunded`
- `Settled`

After you send data to CashBox, CashBox will issue a Return object with ***returnCode*** and ***returnString*** to inform you if the call completed successfully. (Codes for the `Return` object are modeled after standard HTTP return codes). If the call succeeds, you receive a code of 200 and the string OK. ***returnCode*** and ***returnString*** may be used to interpret errors. If one or more of the `MigrationTransactions` included in the `migrate` call failed to be migrated, the return will also include an array of `TransactionValidationResponse` objects describing how/why the migration attempt failed. Be certain to act upon the `TransactionValidationResponses` returned.

For more information, see Transaction.migrate in the ***CashBox API Guide***.

6.10 Refunding Customers

Use the `Refund` object to refund customers. For compliance reasons, CashBox allows refunds for no more than the amount of the original transaction, but supports both full and partial refunds of the original charge.

CashBox automatically creates transactions for recurring billing from `AutoBill` objects. Fetch these transactions from CashBox by calling `Transaction->fetchDeltaSince()`, or search for them on the CashBox Portal. Use transaction data as a basis for your customer refund. Search by `merchantTransactionId` for an AutoBill-related transaction and for the refund amount.

Refund a previously completed Transaction:

```
// Create a new Refund object
$refund = new Refund();

$txn = new Transaction();

// specify a known transaction by its merchant ID. This transaction
// should be in the 'Captured' state so it can be refunded
$txn->setMerchantTransactionId('WID-CUS-9302871');

// associate the account and refund objects
$refund->setTransaction($txn);

// set the amount of the refund
$refund->setAmount(10.00);
$refund->setTimestamp('2009-02-11T22:34:32.265Z');
$refund->setReferenceString('myRefundId101');

// object created so we can call perform() method on it
$refundFactory = new Refund();

// refund the transaction using the SOAP call
$response = $refundFactory ->perform(array($refund));
```

For more information, see the Section 15: The Refund Object in the ***CashBox API Guide***.

7 Working with Entitlements

An ***Entitlement*** defines the goods or services to which a customer is entitled, as obtained through a subscription. An `Entitlement` object associated with an `Account` object specifies whether a customer has access to a service or product on the date the `Entitlement` object is retrieved from CashBox.

Entitlements may be associated with Accounts, Billing Plans, or Products.

The `Entitlement` object encapsulates the Entitlement's description, status, start and end timestamp, and the Account to which the Entitlement applies.

Note:	If you are upgrading from CashBox 4.1 or previous, you must contact Vindicia Client Services to enable a merchant configuration setting which will allow Entitlements to work properly for CashBox 4.2 and greater.

7.1 Creating Entitlements

Entitlement IDs are optional, and are simple strings that are merchant-defined. The example in Section 4.1: Creating Billing Plans, defines the `merchantEntitlementId`: "***Standard,"*** and associates it with the `BillingPlan` object. The example in Section 3.1: Creating Products, defines the `merchantEntitlementId`: "***Video Access,"*** and associates it with the `Product` object. Billing plans and products may contain an unlimited number of entitlement IDs.

Use the online magazine site as an example in defining a customer's access using entitlement IDs. Define one `merchantEntitlementId` for general access, and name it ***Standard***. Create another `merchantEntitlementId` for more extensive access to the site, and name it ***Premium***. Then, design your site so that *premium* customers can access a special multimedia content area in addition to the magazines available to ***standard*** subscribers.

To grant an entitlement to an Account, create an AutoBill that includes a Product with the entitlement, or assign the entitlement directly to the Account.

Entitlements granted directly to an Account must be granted and revoked manually; they will not be automatically generated by CashBox.

Entitlements granted to an Account through an AutoBill (with a Product or Billing Plan holding Entitlements), remain on the Account as long as the AutoBill is in *Good Standing* status.

The `entitlement.fetch*` methods will return entitlements granted both ways.

For example, if an `AutoBill` for an `Account` includes one `Product` that offers *Video access* entitlement, and a second `Product` that offers *Premium* entitlement, the effective entitlements available to the `Account` while the `AutoBill` object is active, are *Video access* and *Premium*.

7.2 Entitlement Status

CashBox manages Entitlements with the assumption that your customers will continue to pay their bills. Working under this premise, an Entitlement is deemed to be active until the end of Billing Plan associated with the AutoBill. If the Billing Plan defines an unlimited series of payments, the `endTimestamp` for the `Entitlement` will be infinite, and the `Entitlement` will be considered active until the Billing Plan ends, or the AutoBill is stopped due to customer request, or failure to pay.

In creating Entitlement object, CashBox assumes that the Entitlement will be active as long as the Billing Plan is in effect.

An `Entitlement` object becomes inactive:

- when the AutoBill ends.
- when the AutoBill is cancelled with immediate disentitlement.
- if your customer has failed to pay a scheduled payment, and their grace period has expired. (The grace period allows your customers continued access after the Billing date, to allow for attempted retries, if necessary. Note that the grace period does not extend the end-date if the `AutoBill` object has been cancelled.)

7.3 Caching Entitlements

Do not make a SOAP call to CashBox to check a customer's account's entitlements every time the customer logs in or attempts to access a certain resource. Instead, cache entitlements locally, and query their status at periodic intervals (for example, by making a `$entitlement->fetchDeltaSince()` call once a day).

While caching an `Entitlement` object for an `Account` object, remember that the active status of the entitlement is valid only until the `endTimestamp` value specified on the `Entitlement` object. If you make periodic `Entitlement->fetchDeltaSince()` calls, you may receive an updated `Entitlement` object. If you do not receive such an update, assume that the `Entitlement` object is still valid.

If you are not making periodic `Entitlement->fetchDeltaSince()` calls to automatically receive updated `Entitlement` objects, the cached `active` status of an entitlement is valid until `endTimestamp`. To proactively obtain an update to the `Entitlement` object, make an `entitlement->fetchByEntitlementIdAndAccount()` or `Entitlement->fetchByAccount()` call. If the `Entitlement` object is revoked prematurely due to an `AutoBill` cancellation, your cached records will not be up-to-date. Therefore, make a periodic `Entitlement->fetchDeltaSince()` call that returns all `Entitlement` objects that might have changed since the specified timestamp, to maintain current entitlement status.

Tip: Create a descriptive and consistent entitlement ID naming system to prevent confusion due to multiple entitlements linked to a single `AutoBill` or `Account` object.

For example, instead of creating entitlements for one `AutoBill` object called ***Standard*** and ***Video Only,*** and for another `AutoBill` called ***Blog Access*** and ***Video Only***, name the entitlements for the first AutoBill ***Standard Access*** and ***Video Only,*** and the second ***Blog Access*** and ***Blog Video Only***.

This allows you to reuse previously defined Entitlement objects in new combinations, without ambiguity. For those customers who wish to access video across your site, you could create an AutoBill which grants ***Video Only*** and ***Blog Video Only*** entitlements. For your customers who don't wish to read the Blogs, but do want access to their video content, you could offer an AutoBill that combines ***Standard Access*** with ***Blog Video Only*** entitlements.

7.4 Monitoring Entitlement Status

When a customer logs into your site and tries to access a resource, examine the (cached) Account's Entitlements and their expiration dates, to determine whether to allow the customer entry. Do **not** retrieve the `AutoBill` object for the Account and examine ***its*** status, as Entitlements may remain active after an AutoBill has been stopped. For example, if a subscription is cancelled midway through a Billing Period, the `AutoBill` status will be `Stopped`, but the customer may be granted access until the paid period expires.

The `Entitlement` object provides two methods for retrieving entitlements by the `Account` object, `entitlementID`, or both:

```
$entitlement->fetchByEntitlementIdAndAccount()
```

and

```
$entitlement->fetchByAccount().
```

Call these methods to determine if a customer can access a specific resource on your site. Both methods return `Entitlement` objects, each specifying an `Account` object, an `entitlementID`, and an `endTimestamp` value. To show if the `Entitlement` object is active on the date fetched, CashBox sets the value of the active flag on the `Entitlement` objects returned by the calls according to the status of the associated `AutoBill` object and its end-date (the time until which the customer is expected to pay).

8 Working with Rate Plans

CashBox Rate Plans allow you to create tiered pricing structures for your Products. These may be License or Usage based, and the units by which they are measured may be defined to fit your needs.

Most Rate Plan operations should be performed using the CashBox GUI, including creating and applying Rate Plans to existing Products. Use the CashBox API to create an AutoBill that includes a `AutoBillItem` with a pre-defined Rate Plan, to upload collected Rated Units, or to fetch a history of reported Events.

8.1 Recording Rated Units

"Rated Units" are the means by which CashBox measures the number of Licenses or units of a Rated Product for which your customers should be billed. CashBox does not automatically record or track Rate Plan use by your customers, nor does it tally the number of Rated Units applied to any AutoBill. To enter customer use for billing calculations, you must report Rated Units to CashBox through the CashBox API.

Rated Units are grouped into `Events` for reporting purposes. Each `Event` must be associated with a single `AutoBillItem`; but one `AutoBillItem` may include several Events.

When Events are reported, CashBox ***replaces*** the previous total for License based plans, and ***augments*** the previous total for Usage based plans. For example, in a License based Plan, if the previous number of recorded Units was 15, and you call the `recordEvent` method to report 12, your customer's current number of unbilled Rated Units will be 12. With a Usage based Plan, reporting 12 new units of use will increase the number of Unbilled Units from 15 to 27.

Use the `RatePlan.recordEvent` method to pass an array of `Events` to your CashBox system. If you wish to pass more than 50 Events, you must break your call into several separate Record calls.

Note: CashBox allows 50 Events to be reported through each `Record` call. Vindicia best practices recommendation is to use the `recordEvent` method to report batches of Events, at designated moments throughout the day.

Each `Event` refers to exactly one `AutoBillItem`, and the `Event` object must identify the item that is intended. When reporting Events, be certain to identify a unique `AutoBillItem` for each Event. The `AutoBillItem` may be identified using any combination of the following objects' identifiers: `Account`, `AutoBill`, `AutoBillItem`, or `Product`. CashBox requires that at least one of the following three data members be specified: `Account`, `AutoBill`, or `AutoBillItem`.

When reporting `Events`, CashBox will issue an error if two `AutoBillItems` exist which fulfill the reported Event parameters. Be certain to pass in enough information to uniquely identify the `AutoBillItem` to which the `Event` should refer.

The following example identifies the `AutoBillItem` directly.

Record a Rated Unit Event:

```
$rateplan = new RatePlan;
$event = new Event;

$event->setMerchantEventId('rating_123');
$event->setMerchantAutoBillItemId('abitem_321');
$event->setAmount(42);

$response = $rateplan->recordEvent(array($event));
```

While the `merchantEventId` is optional, it may be used to guarantee that a single Event is not reported multiple times. If you report the same Event twice with the same identifier, CashBox will reject the second reported Event. It is also useful in customer support, when searching for a questioned billing item.

In the previous example, `rating_123` is used as an identifier for the Reported Event.

Record an Event for a specific AutoBill:

The following example identifies the rated `AutoBillItem` by its `MerchantAutoBillId`. If there is only one `AutoBillItem` on the listed `AutoBill`, this method will uniquely identify the `AutoBillItem`. If there is more than one `AutoBillItem` on the `AutoBill`, CashBox will return an error string saying the input `AutoBillItem` is not unique.

Note:	When working with Rate Plans, assign a name to `AutoBillItems`, to facilitate working with `Events`.

```
$rateplan = new RatePlan;
$event = new Event;

$event->setMerchantEventId('rating_124');
$event->setMerchantAutoBillId('ab_715');
$event->setAmount(42);

$response = $rateplan->recordEvent(array($event));
```

Record an Event for a specific AutoBill and Product:

```
$rateplan = new RatePlan;
$event = new Event;

$event->setMerchantEventId('rating_125');
$event->setMerchantAutoBillId('ab_715');
$event->setMerchantProductId('pr_29');
$event->setAmount(42);

$response = $rateplan->recordEvent(array($event));
```

If there are multiple rated Products on an AutoBill, identifying both the AutoBill and the Product ID may uniquely identify the `AutoBillItem` associated with the `Event`.

8.2 Deducting Rated Units

CashBox also allows you to deduct unbilled Rated Units from an AutoBill.

Use the `deductEvent` method to reduce the outstanding, unbilled balance of `Events` for an AutoBill, by creating a new Event, with a negative value. Use the `reverseEvent` call to remove a specific, previously recorded Event.

> **Note:** Calling `deductEvent` will fail if it results in a negative Rated Unit balance on the `AutoBillItem`. CashBox does not support negative balances on Rated Units.

Use `reverseEvent` to credit an Account for previously billed Events.

Deduct Events:

```
$rateplan = new RatePlan;
$event = new Event;

$event->setMerchantEventId('rating_129');
$event->setMerchantAutoBillId('ab_715');
$event->setAmount(2);

$response = $rateplan->deductEvent(array($event));
```

CashBox tracks this deduction as a distinct Event, available in any audit trail of Events.

8.3 Reversing (Billed) Rated Unit Events

`reverseEvent` is similar to `deductEvent`, but differs in that it must refer to an Event that has been previously recorded. `reverseEvent` may not be used to add a negative number of Rated Units to an Account; it may only be used to reverse a previously recorded Event.

`reverseEvent` may only be applied to Events for which the customer has not yet been billed. Therefore, CashBox will not construct a Credit or process a refund for the item.

Reverse Rated Unit Events:

```
$rateplan = new RatePlan;
$event = new Event;

$event->setMerchantEventId('rating_124');
$event->setMerchantAutoBillId('ab_715');

$response = $rateplan->reverseEvent(array($event));
```

8.4 Fetching and Reporting Rated Units

In working with Rated Products, your customers' access will be measured (and billed) in Rated Unit Events. CashBox offers an API interface to work with this reporting system.

8.4.1 Fetching a Summary (Total) of Unbilled Rated Unit Events

Calling `fetchUnbilledRatedUnitsTotal` calculates the number of Units and the currency amount billable for a given AutoBill (or AutoBills) at the moment of the call. This call returns a `RatedUnitSummary` object, which contains the totals for a single `AutoBillItem`.

> **Note:** This call returns the number of unbilled Units at the moment of the call. This call may not reflect the amount for which your customer will be billed, if more Events are reported before the end of the current Billing Cycle.

This example requests a report for all the unbilled Rated Units for a specified AutoBill, and returns an array of one Summary for each rated `AutoBillItem` included on the AutoBill. For example, if an `AutoBill` has two rated `AutoBillItems`, the call will return an array of two `RatedUnitSummary` objects.

Fetch a summary of unbilled Rated Unit Events:

```
$rateplan = new RatePlan;
$response = $rateplan->fetchUnbilledRatedUnitsTotal(
    null,      # $account
    $myAutoBill,# $myAutoBill
    null,      # $product
    null,      # $ratePlan
    null,      # $startTimestamp
    null,      # $endTimestamp
    0,         # $page
    50,        # $pageSize
);
$summaries = $response->['data']->ratedUnitSummary;
foreach ($summaries as $sum) {
    print $sum->ratedUnitTotal;
    print $sum->currentTotalRatedUnitsBill;
}
```

Fetch a summary of unbilled Events for a specified AutoBill and Product:

```
$rateplan = new RatePlan;
$response = $rateplan->fetchUnbilledRatedUnitsTotal(
    null,      # $account
    $myAutoBill,# $myAutoBill
    $myProduct,# $myProduct
    null,      # $ratePlan
    null,      # $startTimestamp
    null,      # $endTimestamp
    0,         # $page
    50,        # $pageSize
);
$summaries = $response->['data']->ratedUnitSummary;
foreach ($summaries as $sum) {
    print $sum->ratedUnitTotal;
    print $sum->currentTotalRatedUnitsBill;
}
```

This call returns two Summary objects, as shown below.

1st object	
accountVid	0b69d0...
autoBillItemVid	ae3992...
autoBillVid	60263a...
productVid	8479ce9...
ratePlanVid	b4130b...
merchantAccountId	account_13345386734
merchantAutoBillId	autobill_13345386734
merchantAutoBillItemId	autobillitem_13345386734
merchantProductId	product_13345386734
merchantRatePlanId	rateplan_13345386734
currentTier	basic
currentTotalRatedUnits-Bill	4.27
eventCount	1
ratedUnit	'namePlural' => 'minutes' 'nameSingular' => 'minute'
ratedUnitTotal	37

2nd object	
accountVid	0b69d0...
autoBillItemVid	370de7...
autoBillVid	60263a...
productVid	8479ce...
ratePlanVid	496c89...
merchantAccountId	account_13345386734
merchantAutoBillId	autobill_13345386734
merchantAutoBillItemId	autobillitem_23345386734
merchantProductId	product_13345386734
merchantRatePlanId	rateplan_13345386734
currentTier	basic
currentTotalRatedUnits-Bill	3.15
eventCount	1
ratedUnit	namePlural' => 'hour' nameSingular' => 'hours’
ratedUnitTotal	21

This example returns two `RatedUnitSummary` objects, because there are two different `AutoBillItems` that match the query. The first bills for 37, and the second for 21 minutes of use on different Rate Plans. There is one event for each. If the customer were to be billed now, they would be charged for one minute at $4.27, and one hour at $3.15.

8.4.2 Fetching Billed or Unbilled Rated Unit Events

CashBox allows you to fetch both billed and unbilled Rated Unit Events, allowing you to compare your customer's current with previous use patterns.

Calling `fetchUnbilledEvents` returns the (unbilled) Events themselves, rather than a Summary Report. Use this call to display your customer's (not yet billed) use for the current Billing Cycle.

`fetchEvents` differs from this call only in that it will return ***ALL*** Events for the given input parameters, billed or unbilled. Use this call to compare your customer's previous use with their current use, or to determine applicable upgrade plans to offer.

Fetch unbilled Rated Unit Events:

```
$rateplan = new RatePlan;
$response = $rateplan->fetchUnbilledEvents(
    null,      # $account
    $myAutoBill,# $myAutoBill
    $myProduct,# $myProduct
    null,      # $ratePlan
    null,      # $startTimestamp
    null,      # $endTimestamp
    0,         # $page
    50,        # $pageSize
);
$events = $response->['data']->event;
foreach ($events as $ev) {
    print $ev->amount;
    print $ev->description;
    print $ev->eventDate;
    print $ev->billedStatus;
    print $ev->VID;
}
```

For this call, the arguments are the same as those for `fetchUnbilledRatedUnitsTotal`, but the objects returned are different; `fetchUnbilledRatedUnitsTotal` returns all Events corresponding to the `AutoBillItems` for the given AutoBill and Product, rather than simply a summary.

The returned Events include two fields in addition to those passed in using `recordEvent`:

- `billedStatus` lists whether the event has been billed (in items returned from `fetchUnbilledEvents`, this will always be false; if you call `fetchEvents`, it could be true or false).
- `VID` is Vindicia's unique identifier for the Event.

For more information on returned data members for the Event, see the `Event` Subobject in the ***CashBox API Guide***.

Note: All optional data members identifying the event (such as the `autoBillVid`) will be entered by CashBox, if available.

9 Working with Customer Notifications

CashBox can automatically issue customer notifications at predefined moments in the billing cycle, using customized templates. Templates may be defined to notify your customer of billing events (imminent, in-process, successful, or failed), to submit invoices, to warn of a pending subscription expiration, to inform of an overdue balance, or to simply keep them informed of changes in their subscription plan. Work with Vindicia Client Services to create templates to keep your customers informed and engaged in their relationship with your company and your products.

Note:	Creating CashBox templates differs for Billing notifications and Invoicing events, in that Billing templates use a different tag format than Invoicing templates. Be certain to use the appropriate system when creating your Templates.

CashBox requires that you submit an email template as well, if you wish Billing and Invoicing notifications to be emailed to your customers. The email template contains the email *headers*; the Billing and Invoicing templates include the email *contents*.

Note:

- If you do not submit an email template, no notifications will be emailed to your customers.
- If you do not define any templates, no notifications will occur. For example, if no Soft Fail notification is in the database, CashBox will not notify the customer on a soft fail.
- If you do not set a preferred language for any template and an English template exists in the database, CashBox notifies the customer using the English template.

If the `prenotifyDays` setting in a `BillingPlan` object is `0` or is not set, CashBox sends no prebilling notifications.

9.1 Setting the Preferred Language

CashBox allows you to offer templates in multiple languages. For example, you may define billing notification templates in English, German, French, and Chinese. The notification template used will be based on the customer's preferred language setting in the `Account` object.

To define the preferred language, use the W3C IANA Language Subtag Registry standard. (CashBox supports the ISO-639.2 standard, but recommends the IANA Language Subtag Registry, which is more recent and complete than the ISO-639.2.)

If no active template in the customer's preferred language exists for a billing event, CashBox sends the English version. If the English version does not exist, CashBox sends no notifications.

9.2 Working with Billing Events

CashBox may be configured to automatically send email notifications to your customers for various billing events, including impending transactions, AutoBill expiration or renewal, or payment processing. To enable these events, you must both set the corresponding flags through the Cashbox API, and supply Vindicia with the corresponding email templates. If no template is supplied, no email can be sent.

Contact your Client Services representative for more information.

9.2.1 CashBox Billing Events

CashBox may be configured to issue email notification of the following billing events. To generate the email, the appropriate flag must be set to `true`, and the corresponding email template must be available to CashBox.

The following table lists the CashBox billing events and suggested email content.

Table 9-1 Billing Events

Notice / Event	Suggested Content
Prebilling The billing event is pending.	Product information, AutoBill expiration date, the date on which billing will occur, the amount of the bill, the opt-out procedure, and contact information for your support team. Specify when to send this notification in the `prenotifyDays` data member of the `BillingPlan` object.
Initial Success The first billing event for a new AutoBill has been successful.	Use this email to welcome new customers to the business. The date of the next billing attempt, with product information, billing address, and transaction details (transaction number, billing date, and payment information), and the total amount for the transaction. (If the payment method is Tokens, replace the currency with the Token type.)
Success A successful billing event has occurred.	The date of the next billing attempt, with product information, billing address, and transaction details (transaction number, billing date, and payment information), and the total amount for the transaction. (If the payment method is Tokens, replace the currency with the Token type.)
Soft Fail The billing attempt has failed. The payment processor's return code indicates that if the card is resubmitted, the billing might succeed. CashBox will retry.	The date of the next billing attempt, with product information, billing address, and transaction details (transaction number, billing date, and payment information), and the total amount for the transaction. Include instructions for your customer to update their billing information.
Hard Fail The billing attempt has failed. The payment processor's return code indicates that no more transactions will be accepted.	Product information, billing address, and transaction details (transaction number, billing date, and payment information), and the total amount for the transaction. Include instructions for your customer to update their account information to remain in good standing.
Cancellation The customer has opted out of the AutoBill Subscription, and an upcoming bill has been cancelled. (Sent only if a prebilling notification has already been issued.)	Notification that the transaction has been cancelled, and that your customer will not be billed on the next billing date (if a recurring bill.) Include contact information for re-subscribing to your service.
Billing Delay Issued upon a billing delay (extension of entitlements, without captured payment), only if the customer has not been pre-notified of the billing event.	The date of the next billing attempt, with product information, billing address, and transaction details (transaction number, billing date, and payment information), and the total amount for the transaction.
Prenotification: No Payment Method Issued as a Prebilling Notification, when no Payment Method is listed for the AutoBill.	Product information, AutoBill expiration date, the date on which billing will occur, the amount of the bill, the opt-out procedure, and contact information for your support team. Include instructions for your customer to update their billing information. Specify when to send this notification in the prenotifyDays data member of the BillingPlan object.

Table 9-1 Billing Events *(Continued)*

Notice / Event	Suggested Content
Failure: No Payment Method The billing attempt has failed due to lack of a defined Payment Method.	The date of the next billing attempt, with product information, billing address, and transaction details (transaction number, billing date, and payment information), and the total amount for the transaction. Include instructions for your customer to update their billing information.
Expiration The billing plan is about to expire with no more periods defined.	Product information, an expiration date for the subscription, and information on how to extend the subscription.
End of Trial The customer's free trial period is about to expire.	Product information, the duration and expiration date of the free trial, subscription amount, and other billing details for the paying subscription that is about to begin. Include opt-out instructions.
Real-Time Inbound Failure A real-time transaction initiated by a customer for you has failed.	Product information, billing address, and transaction details, including the transaction number, billing date, account number, amount, currency and sales tax (if applicable). Include instructions on how to update the account and remedy any issues.
Real-Time Inbound Success A real-time transaction initiated by a customer for you has succeeded.	Product information, billing address, and transaction details, including the transaction number, billing date, account number, amount, currency and sales tax (if applicable).
Real-Time Outbound Failure A real-time transaction initiated by you for a customer has failed.	Line-item details on the transaction and on the account to which the payment applies.
Real-Time Outbound Success A real-time transaction initiated by you for a customer has succeeded.	Line-item details on the transaction and on the account to which the payment applies.
Refund A refund to the customer has succeeded.	Product information, billing address, and refund details: transaction number, refund date, payment method, and amount. (If the Payment Method is Tokens, provide a Token balance in the message.)
Push Initiation A push payment has begun.	Product information, and instructions on how to complete the transaction, including the URL for the payment slip form.
Push Reminder The customer has not completed a push payment transaction within the allotted time.	Product information, and instructions on how to complete the transaction, including the URL for the payment slip form.
Failure: Insufficient Tokens Not enough tokens are available to pay for a recurring Transaction.	Token information, and instructions on how to obtain additional tokens.
Failure: Insufficient Tokens Not enough tokens are available to pay for a one-time Transaction.	Token information, and instructions on how to obtain additional tokens.

9.2.2 Billing Event Settings

The following table lists the customizations available to your notification timing through the CashBox API.

Table 9-2 Settings for Notification Templates

Notification	Settings
Prebilling	`BillingPlan.prenotifyDays`: Sets the number of days before billing will occur to send the notification. If the `prenotifyDays` setting in a `BillingPlan` object is `0` or not set, CashBox sends no prebilling notifications. `Account.warnBeforeAutobilling`: Specifies whether or not to send prebilling notifications to the customer. `doNotNotifyOnFirstBill`: Cancels the first prebilling notification for a `BillingPlan` period, to prevent the customer from receiving a prebilling notification on the same day that they sign up.
Initial Success	None.
Success	None.
Hard Fail	Work with Vindicia to map the payment processor reason codes to hard fails, to define the number of days after the billing attempt fails CashBox should retry, and to define the number of retries. By default, CashBox retries hard fails only once, and sends notifications if the first retry fails. (CashBox may also be set to send hard-fail notifications for push payment methods after the transaction has expired, for example, if a customer does not perform the tasks necessary to complete the transaction.)
Soft Fail	Work with Vindicia to map the payment processor reason codes to soft fails, to define the number of days after the billing attempt fails CashBox should retry, and to define the number of retries. CashBox retries soft fails until it exhausts the number of retries specified. If the final retry fails, CashBox sets the Soft Fail to Hard Fail.
Cancellation	None.
End of Trial	`BillingPlan.expireWarningDays`: Sets the number of days before a free trial ends to send a warning email. Whether to send this notification may also be defined at the AutoBill level.
Expiration	`BillingPlan.expireWarningDays`: Sets the number of days before a subscription ends to send a warning email. Whether to send this notification may also be defined at the AutoBill level.
Refund	None.
Real-Time Inbound Success	None.
Real-Time Outbound Success	None.
Real-Time Inbound Failure	None.
Push Initiation	None.

Table 9-2 Settings for Notification Templates *(Continued)*

Notification	Settings
Push Reminder	Contact Vindicia Client Services to set the number of days after initiation to send this reminder, and to specify the number of reminders (retry times).
Insufficient Tokens	None.
Tokens Granted	None.

9.2.3 Parent-Child Account Billing Notifications

CashBox automatically determines which Accounts will receive Billing Notifications in a Parent-Child relationship, depending on which Account is the holder of the Payment Method attached to the AutoBill.

(In all cases, email generation is dependent upon your notification settings.)

For AutoBills held by a Child Account, but paid by the Parent's Payment Method, the following notifications are sent to both the Parent and the Child Account:

- Billing Delay, No Payment Method?
- Cancellation
- Expiration
- Failure (Insufficient Tokens or No Payment Method)
- Pre-billing, No Payment Method

The following notifications are sent only to the Parent Account (the holder of the Payment Method associated with the AutoBill):

- Billing Delay
- End of Trial
- Hard Fail
- Pre-billing
- Real-Time Tx Outbound Fail
- Real-Time Tx Outbound Success
- Real-Time Tx, Insufficient Tokens
- Refund Success
- Soft Fail
- Success

Note: If a Child Account is the holder of the Payment Method, only the Child Account will receive these notifications.

9.2.4 Creating Billing Notification Templates

To create a notification template, use variables, or “tags,” that pull information from CashBox to provide customer- and Account-specific information. Work with Vindicia Client Services to submit your templates for inclusion in your CashBox system.

Create billing notification templates in HTML. Vindicia recommends that you use an industry-standard HTML editor (**not** Word HTML). Host graphics, if any, from your site and point to them in the templates using HTML `href` tags.

Billing Event Template Tags

Tags are used to provide a place holder in the template where CashBox data will be inserted when the billing event notification is rendered. CashBox notification templates support looping, to allow for multiple charges, credits, or payments.

The following tags may be used in a Billing Event template:

Table 9-3 Billing Event Template Tags (Variables)

CashBox Tag	Description
<tpl name="address"/>	The billing address (multiple lines). **Note:** This tuple will print the customer’s name, as well as all lines included in the billing address.
<tpl name="amount"/>	The total transaction amount. Note: This is a deprecated tag. Specify `grand_total` instead.
<tpl name="billing_plan_desc"/>	The Billing Plan’s description.
<tpl name="billing_plan_id"/>	The `merchantBillingPlanId`.
<tpl name="ccnum"/>	The credit-card number (the last four digits only).
<tpl name="cctype"/>	The credit-card type, such as Visa or MasterCard.
<tpl name="currency"/>	The currency code, for example, USD, or the description of the token type that serves as payment, for example, Award Points, Downloads, or Transférer.
<tpl name="currency_symbol"/>	The Billing Plan’s currency, as indicated by the ISO 4217 currency code entered for the `BillingPlanPrice` object’s `currency` data member.
<tpl name="date"/>	The billing date (MM-DD-YYYY).

<tpl name="desc[***n***]"/>	An array of line-item descriptions that accompany real-time transaction notifications. Specify each item of the array and provide the maximum number of line items expected. For example:
	`<tpl name="desc[0]"/>`
	`<tpl name="desc[1]"/>`
	`<tpl name="desc[2]"/>`

<tpl name="expirationdate"/>	The subscription’s expiration date (MM-DD-YYYY).
<tpl name="formaction"/>	The URL for a push payment method slip from the payment processor.

Table 9-3 Billing Event Template Tags (Variables) *(Continued)*

CashBox Tag	Description
<tpl name="grand_total"/>	The grand total of the transaction. For prebilling and postbilling notifications, this is the total cost (subtotal plus tax). For tokens, this value is the same as `amount`.
<tpl name="interval"/>	The length of the billing period, such as 12 months.
<tpl name="invoiceno"/>	The transaction ID, available only after you have submitted a transaction.
<tpl name="ISOdate"/>	The billing date (YYYY-MM-DD).
<tpl name="ISOexpirationdate"/>	The subscription's expiration date (YYYY-MM-DD).
<tpl name="ISOnextdate"/>	The next billing date, that is, the next retry after failure or the next scheduled billing (YYYY-MM-DD).
<tpl name="length"/>	The length of the current billing period, for example, 2-week or 1-month. CashBox supports this tag only in the End of Trial and Expiration templates.
<tpl name="merchant"/>	Merchant's name.
<tpl name="merchant_affiliate"/>	The partner or affiliate associated with the Billing event.
<tpl name="name"/>	The customer's first and last names.
<tpl name="name_on_entitlement_account"/>	The Account name (`merchantAccountId`) referenced in the email. (The Account holding the entitlements referenced in the email.) When a parent Account is notified about a child Account's entitlements, this field will display the child Account's name.
<tpl name="name_on_payer_account"/>	The name of the payer account. When a child account is sent an email notification, this field will display the parent Account's name.
<tpl name="nextdate"/>	The next billing date, that is, the next retry after failure or the next scheduled billing (MM-DD-YYYY).
<tpl name="payment_provider"/>	The Payment Provider who handled the Billing event.
<tpl name="payment_token_balance"/>	Valid for tokens only, this tag, which supports English only, returns the sentence "Your ***token-account*** balance is *X*," where *token-account* is the description of the token type and *X* is the balance. For example: "Your Award Points balance is 50.00." Note that the balance is the token balance after the transaction. If multiple token types apply to the customer account, this tag returns multiple lines, with each line corresponding to a different token type. Note: If included in an email notification template but tokens are not in use, this tag returns a blank.

Table 9-3 Billing Event Template Tags (Variables) *(Continued)*

CashBox Tag	Description
<tpl name="grand_total"/>	The grand total of the transaction. For prebilling and postbilling notifications, this is the total cost (subtotal plus tax). For tokens, this value is the same as `amount`.
<tpl name="interval"/>	The length of the billing period, such as 12 months.
<tpl name="invoiceno"/>	The transaction ID, available only after you have submitted a transaction.
<tpl name="ISOdate"/>	The billing date (YYYY-MM-DD).
<tpl name="ISOexpirationdate"/>	The subscription's expiration date (YYYY-MM-DD).
<tpl name="ISOnextdate"/>	The next billing date, that is, the next retry after failure or the next scheduled billing (YYYY-MM-DD).
<tpl name="length"/>	The length of the current billing period, for example, 2-week or 1-month. CashBox supports this tag only in the End of Trial and Expiration templates.
<tpl name="merchant"/>	Merchant's name.
<tpl name="merchant_affiliate"/>	The partner or affiliate associated with the Billing event.
<tpl name="name"/>	The customer's first and last names.
<tpl name="name_on_entitlement_account"/>	The Account name (`merchantAccountId`) referenced in the email. (The Account holding the entitlements referenced in the email.) When a parent Account is notified about a child Account's entitlements, this field will display the child Account's name.
<tpl name="name_on_payer_account"/>	The name of the payer account. When a child account is sent an email notification, this field will display the parent Account's name.
<tpl name="nextdate"/>	The next billing date, that is, the next retry after failure or the next scheduled billing (MM-DD-YYYY).
<tpl name="payment_provider"/>	The Payment Provider who handled the Billing event.
<tpl name="payment_token_balance"/>	Valid for tokens only, this tag, which supports English only, returns the sentence "Your ***token-account*** balance is *X*," where *token-account* is the description of the token type and *X* is the balance. For example: "Your Award Points balance is 50.00." Note that the balance is the token balance after the transaction. If multiple token types apply to the customer account, this tag returns multiple lines, with each line corresponding to a different token type. Note: If included in an email notification template but tokens are not in use, this tag returns a blank.

Table 9-3 Billing Event Template Tags (Variables) *(Continued)*

CashBox Tag	Description
<tpl name="payment_type"/>	A descriptor of the payment method, as follows: • For credit cards, the descriptor is “credit card.” • For ECP, the descriptor is "bank account.” • For PayPal, the descriptor is “PayPal account.” • For direct debits, the descriptor is “direct debit accounts.” • For Boleto Bancário, the descriptor is “conta bancária” (Portuguese only). • For tokens, the descriptor is the token type for payment, followed by account. For example, if the token type is Award Points, the descriptor is “Award Points account.”

<tpl name="price"/>	An array of product prices that accompanies recurring or real-time transaction notifications. If used in conjunction with multiple line items and the desc tag, specify the maximum number of line items for each item, for example:
	`<tpl name="price[0]"/>`
	`<tpl name="price[1]"/>`
	`<tpl name="price[2]"/>`

<tpl name="product"/>	The product name. **Note:** This tuple will be populated ***only*** for AutoBills, and not for One-Time Transactions.
<tpl name="refid"/>	The Refund ID.
<tpl name="refund_amount"/>	The amount of the refund. In the case of tokens, this is the number of units of the token type that serves as payment.
<tpl name="refund_approval_code"/>	The Payment Processor’s refund approval code.
<tpl name="refund_capture_timestamp"/>	The `Refund` object’s `timestamp`.
<tpl name="refund_currency"/>	The currency of the refund. In the case of tokens, this is the description of the token type that serves as payment, for example, Award Points, Downloads, or Transférer.
<tpl name="refund_currency_symbol"/>	The refund’s currency symbol, as indicated by the ISO 4217 currency code of the `Refund` object’s `currency` data member.
<tpl name="refund_merchant_identifier"/>	The `Refund` object’s `merchantRefundId`.
<tpl name="refund_note"/>	A note on the refund.
<tpl name="refund_payment_provider"/>	The Payment Processor for the refund.
<tpl name="refund_pp_order_number"/>	The Payment Provider’s order number for the refund.
<tpl name="refund_status"/>	The refund’s status.
<tpl name="refund_timestamp"/>	The timestamp in the format “2008-09-19 15:20:04.”
<tpl name="send_to_email"/>	The email address to which the Billing Event email should be sent.
<tpl name="serialnum"/>	Your identifier for the `AutoBill` object.

Table 9-3 Billing Event Template Tags (Variables) *(Continued)*

CashBox Tag	Description
<tpl name="sku"/>	Your product identifier.
<tpl name="statementno"/>	The statement number for the Billing event.
<tpl name="subtotal"/>	The pretax amount. For tokens, this value is the same as `amount`.
<tpl name="tax"/>	The applicable tax. If the payment method is tokens, the value is 0.
<tpl name="token_balance"/>	Available remaining Tokens for the Account.
<tpl name="token_change"/>	The change in number of Tokens available to the Account as a result of this Billing event.
<tpl name="token_id"/>	The `merchantTokenId` for the Tokens transferred in this Billing event.
<tpl name="uri"/>	The URL the Payment Processor returned in response to the presentment of a fiscal number (for Boleto Bancario.)
<tpl name="vid"/>	CashBox's unique identifier for the `AutoBill` object, which is useful if embedded in a URL to which your site is redirected when you call `AutoBill.fetchByVid()`.

Note: Not all Billing Event tags are available to all Billing events. For example, refund tuples may not be available to pre-billing notifications. Please work with your Vindicia Client Services representative for more information.

9.3 Working with Invoices

An invoice is used to notify customers of their closing balance for the current billing cycle. You may define the number of days before a Billing Cycle closes for an Invoice to be issued.

An Invoice typically lists an opening balance, previously submitted invoices, payments, credit memos, debit memos, and an ending balance for a given billing cycle.

CashBox automatically generates email messages in parallel with Invoice processing, to which a rendered Invoice template may be attached. You may create your own, custom template, or use one of the default templates provided. In either case, CashBox saves an image of each Invoice sent to a customer, as a PDF, which may be retrieved at a later date for bookkeeping or record keeping purposes.

To generate Invoice emails from CashBox:

- Supply an email template to Vindicia Client Services. This template will be used to generate email notifications to your customers, and must be supplied or no invoice or dunning notifications will be mailed to your customers.
- Supply an invoice template to Vindicia Client Services. Vindicia provides a Default Invoice Template, which may be used in place of your own, customized template.
- Supply a dunning template to Vindicia. If no template is supplied, dunning notices will not be sent to your customers.

Note: If you supply a dunning template you must also provide an invoice template.

Work with Vindicia Client Services to enable Invoice and Dunning Notice generation, and to define the timing for these notifications.

Dunning Notices

Dunning is an extension of Invoicing, and is the process of systematically reminding a customer that payment is overdue, and informing them of the payment process. Dunning notices typically progress from reminders of overdue payment, to notice of imminent account closure. The last dunning notice usually informs the customer of account closure, and entitlement revocation.

CashBox allows you to define the timing of each step in this process. By default, the first dunning/overdue notice is issued a week after the initial due date, to allow time for payments arriving on the due date to be processed and entered into the system. The sequence may be customized to change the timing, content, or inclusion of dunning notices.

(For all steps in this process, CashBox recommends that you include appropriate links and phone numbers to contact your Customer Service department to arrange payment.)

As with all CashBox customer notifications, you must supply CashBox with an email template and a Dunning Notice template for these notices to be rendered and mailed.

Note:	Dunning notice timing is set by Vindicia Client Services. Please speak with your Vindicia representative to define these intervals for your company.

9.3.1 CashBox Invoicing Events

CashBox provides several pre-defined invoice events, which will automatically generate an email notification to your customers. Emails will be sent to the email address specified on the `Account` on the `AutoBill`, with the subject line "Invoice ***<invoice number>*** dated ***<date>,***" and your email address (if supplied) as the **From:** field.

> **Note:** If you do not supply CashBox with an email template, no invoices will be issued. An Invoice will be rendered, but it will not be mailed to your customer.

Work with Vindicia Client Services if you wish to alter the invoicing schedule or if you wish to send more than two dunning notices.

CashBox can be configured to render and issue the following emails in relation to the Invoicing cycle:

Table 9-4 Invoicing Events

Notice / Event	Suggested Content
Open Invoice An *open* invoice may be generated a merchant-specified number of days before the end of every billing cycle.	An email message to the customer, with the invoice attached as a PDF, or inline as plain text or HTML (depending on the customer's email preferences).
Payment Due (1st dunning Notice) An invoice becomes *Due* a merchant-defined number of days after the billing cycle closes.	A reminder that payment is now due. This may include notice of any interest charges or penalties will be added to the account.
Overdue (2nd Dunning Notice) An account becomes *Overdue* a merchant-defined number of days after the Due date.	A notice that payment is overdue, which may include any interest charges or penalties that have been added to the account. This notice often includes information on future action, including account closure and collection agency involvement.

9.3.2 Creating Invoice Templates

Invoice templates allow you to create custom Invoices. They may include custom text, like customer support information, marketing information, or custom graphics. CashBox also provides a generic Invoice Template, if you do not wish to create your own.

If you create a customized template, create both a plain text, and an HTML version, to provide for customers' email preferences settings. You may use text and standard HTML markup in the template.

The tags used in Invoice Templates map directly to your CashBox database, as described in Table 9-5: Invoice Template Tags (Variables).

Default Invoice Template

CashBox supplies a default Invoice template in both plain text and HTML. This Template includes the most common Invoice components.

[% format_amount=format(%10.2f);-%]	
[% merchant %] [% cust_name %] [%cust_address %]	Invoice #: [% invoice_num %] Date: [% invoice_date %]
Previous Balance	[% format_amount(balance_forward) %]
Payments: [% FOR x IN payments %] [% x.amount %] [% x.date %] [% END %]	[% format_amount %]
Total Payments:	[% format_amount(total_payments %]
Balance:	[% format_amount(opening_balance) %]
Current Charges: [% FOR x IN charges %] [% x.product %] [% x.desc %] [% x.quantity %] [% x.total %] [% END %]	[% format_amount %]
Total Current Charges:	[% format_amount(total_charges) %]
Credits: [% FOR x IN credits %] [% x.date %] [% END %]	[% format_amount %]
Total Credits:	[% format_amount(total_credits) %]
Tax:	[% format_amount(tax) %]
Total Amount Due:	[% format_amount(balance_due) %]
Pay By:	[% payment_due_date %]

Figure 9-1 Default Invoice Template (HTML)

Invoice Template Tags

Tags are used to provide a place holder in the template where CashBox data will be inserted when the Invoice is rendered. CashBox Invoice templates support looping, to allow for multiple charges, credits, or payments.

The following tags may be used in an Invoice template:

> **Note:** Because CashBox allows for complex charge calculations, the charges tags are called out separately in the table.

Note: Arrays are shown in Vindicia Red in the following table.

Table 9-5 Invoice Template Tags (Variables)

CashBox Tag	Description
Invoice Information	
[% date %]	Today's date.
[% merchant %]	Your merchant name as it is currently defined in CashBox. **Note:** If this value is not as you expect, please work with your Vindicia Client Support contact to change it.
[% invoice_num %]	Your Invoice number.
[% opening_balance %]	The amount after payments have been subtracted from the balance forward.
[% payment_type %]	The payment type (MAP, CC, EDD, etc.) used to make payment of the balance due.
[% balance_forward %]	The balance carried forward from previous invoices.
[% balance_due %]	The balance due after all payments, credits, charges and taxes have been applied to the balance forward.
[% transaction_id %]	Unique identifier, generated by CashBox for the Invoice. **Note:** This field will be populated only after a payment is made against the Invoice.
[% invoice_date %]	The date the invoice was created and sent.
[% currency %]	The currency used for the transaction.
[% payment_due_date %]	The date the invoice becomes due.
Customer Information	
[% cust_name %]	The customer's name.
[% cust_email_address %]	The customer's email address.
[% cust_address %]	The customer's address. A concatenation of all available customer address information. For example: 1820 Gateway St${sep}Apt. A${sep}Foster City, California 94403${sep}US

Table 9-5 Invoice Template Tags (Variables) *(Continued)*

CashBox Tag	Description
[% cust_address_street1 %]	The customer's street address (1st line).
[% cust_address_street2 %]	The customer's street address (2nd line).
[% cust_address_street3 %]	The customer's street address (3rd line).
[% cust_address_city %]	The customer's city.
[% cust_address_district %]	The customer's state.
[% cust_address_postal_code %]	The customer's zip code.
[% cust_address_city_dist %]	The customer's city, state, and zip code. For example: Foster City, California 94403
Payments	
[% total_payments %]	Currency value of all the payments.
[% payments %]	An array of payments applied to the AutoBill. [% FOR x IN payments %] [% x.pay_type %] [% x.pay_date %] [% x.pay_amount %] [% END %]
type	Used with the [% payments %] tag to retrieve the type of payment used.
date	Used with the [% payments %] tag to retrieve the date the payment was applied to the account.
amount	Used with the [% payments %] tag to retrieve the payment amount.
Credits	
[% total_credits %]	Currency value total of all credits.
[% credits %]	An array of credits applied to the AutoBill. [% FOR y IN credits %] [% y.credit_date %] [% y.credit_amount %] [% END %]
date	Used with the [% credits %] tag to retrieve the date the credit was made to the account.
amount	Used with the [% credits %] tag to retrieve the credit amount.

Table 9-5 Invoice Template Tags (Variables) *(Continued)*

CashBox Tag	Description
Charges	
[% total_charges %]	Currency value total of all charges.
[% charges %]	An array of charges applied to the AutoBill. [% FOR c IN charges %] [% c.product %] [% c.desc %] [% c.price %] [% c.quantity %] [% c.total %] [% c.rate_plan_id %] [% c.rate_plan_description %] [% c.included_units %] [% c.min_fee %] [% c.max_fee %] [% c.unit_name_singular %] [% c.unit_name_plural %] [% c.unit_total_amount %] [% c.unit_amount_and_name %] [% c.rate_plan_tiers %] [% c.rated_unit_events %] [% END %]
product	Used with the [% charges %] tag to retrieve the Product's ID.
desc	Used with the [% charges %] tag to retrieve the Product's description.
price	Used with the [% charges %] tag to retrieve the Product's price.
quantity	Used with the [% charges %] tag to retrieve the number of Products.
total	Used with the [% charges %] tag to retrieve the total charge.
rate_plan_id	Your Rate Plan ID.
rate_plan_description	Your description for the Rate Plan.
included_units	The number of Units included with the Rate Plan.
min_fee	The minimum fee for the Rate Plan.
max_fee	The maximum fee for the Rate Plan.
unit_name_singular	The singular version of the Unit name.
unit_name_plural	The plural version of the Unit name.
unit_total_amount	The total number of reported Units.

Table 9-5 Invoice Template Tags (Variables) *(Continued)*

CashBox Tag	Description
unit_amount_and_name	The number and name of the reported Units. (CashBox will automatically return the singular or plural name, dependent upon the reported number of Units.)
[% rate_plan_tiers %]	An array of all the Tiers in the Rate Plan. [% FOR t IN c.rate_plan %] [% t.tier_name %] [% t.tier_begins_at_level %] [% t.tier_ends_at_level %] [% t.tier_rate_price %] [% t.tier_rated_units_tier_total %] [% t.tier_cost_total %] [% END %]
tier_name	Used with the [% rate_plan_tiers %] tag to retrieve the name of the Tier.
tier_begins_at_level	Used with the [% rate_plan_tiers %] tag to retrieve the `beginsAtLevel` for the Tier.
tier_ends_at_level	Used with the [% rate_plan_tiers %] tag to retrieve the `endsAtLevel` for the Tier.
tier_rate_price	Used with the [% rate_plan_tiers %] tag to retrieve the `ratePrice` for the Tier.
tier_rated_units_tier_total	Used with the [% rate_plan_tiers %] tag to retrieve the number of Rated Units reported for the Tier.
tier_cost_total	Used with the [% rate_plan_tiers %] tag to retrieve the total charge for the Tier.
[% rated_unit_events %]	An array of all (unbilled) Rated Unit Events applied to the AutoBill. [% FOR e IN c.rated_unit_events %] [% e.event_date %] [% e.received_date %] [% e.amount %] [% e.unit_name %] // singular or plural, based on # of units [% e.description %] [% e.rate_plan_id %] [% e.rate_plan_description %] [% e.product_id %] [% e.product_description %] [% END %]
event_date	Used with the [% rated_unit_events %] tag to retrieve the date the specified Event occurred, or the date the Event was reported to CashBox.

Table 9-5 Invoice Template Tags (Variables) *(Continued)*

CashBox Tag	Description
received_date	Used with the [% rated_unit_events %] tag to retrieve the date the event was received.
amount	Used with the [% rated_unit_events %] tag to retrieve the number of Rated Units included in the Event.
unit_name	Used with the [% rated_unit_events %] tag to retrieve the name for the Unit associated with the Event. CashBox will automatically return the singular or plural version of the name, dependent upon the number of Units reported for the Event.
description	Used with the [% rated_unit_events %] tag to retrieve the description for the Event.
rate_plan_id	Used with the [% rated_unit_events %] tag to retrieve the Rate Plan ID for the Event.
rate_plan_description	Used with the [% rated_unit_events %] tag to retrieve the description for the Rate Plan associated with the Event.
product_id	Used with the [% rated_unit_events %] tag to retrieve the Product ID associated with the Event.
product_description	Used with the [% rated_unit_events %] tag to retrieve the description for the Product associated with the Event.
Taxes	
[% tax %]	The total of all taxes applied to the Invoice.
[% taxes %]	An array of taxes applied to the AutoBill. [% FOR t IN taxes %] [% t.taxes_description %] [% t.taxes_amount %] [% END %]
description	Used with the [% taxes %] tag to retrieve the description of the tax.
amount	Used with the [% taxes %] tag to retrieve the amount of the tax.

10 Working with Tokens

A `Token` object represents a metering or virtual-currency unit of a specific type, and may be used to support billing models that use arbitrary tracking units with tokens and related objects in the CashBox API. The units are of your own choosing and may be minutes, downloads, incentive points, virtual currency, storage space, number of users, or any other imaginable unit. In the CashBox API, those units are represented by a generic `Token` object. Tokens allow you to define token *types*, and associate them with a customer Account. Tokens may be purchased, granted, decremented, or refunded, and you may retrieve Token balances for customer Accounts.

The following table describes the three token-related objects.

Table 10-1 Token-Related Objects

CashBox Object	Description
`Token`	Defines a token of a certain type. It must have a unique ID.
`TokenAmount`	The number of tokens, of a certain type, used to define a Billing Plan or product price in terms of tokens.
`TokenTransaction`	A purchase made with tokens. `TokenTransaction` contains attributes that specify the customer account that made the purchase, the amount, and the token type. `TokenTransaction` also includes a timestamp that shows when the transaction occurred. The purchase is a lightweight transaction that can be conducted in lieu of standard CashBox transaction.

You may define any number of Tokens, and can manage each customer's associated balance of units by granting (incrementing), and decrementing Token counts through Products, Billing Plans, or AutoBills.

Token Transactions pass through the CashBox Token Processor, and appear as Transactions in all applicable CashBox Portal pages, including generated Reports.

10.1 Understanding CashBox Token Objects

Any number of token types may be defined that include token IDs and descriptions. The `description` attribute provides a user-friendly token name, which can be included in email notifications. To create Tokens with multiple language-descriptions, create multiple language-specific token types.

For example, if you have customers in the United States and France, define the token types as shown below.

Table 10-2 Multiple Language-Specific Token Types

Token ID	Description
`01_EN_Downloads`	Downloads
`01_FR_Downloads`	Transférer
`04_EN_Storage`	Storage
`04_FR_Storage`	Entreposage

The following CashBox objects are related to tokens:

- `BillingPlan`: You can create billing plans for recurring billing in tokens or in currency. Use tokens or currency, but do ***not*** mix them in billing. Specify the quantity and token type for token billing plans. At billing time, if the customer does not have enough tokens in the account, have CashBox notify the customer with instructions for obtaining additional tokens. For billing plans with multiple token types, CashBox uses the type specified in the payment method for the related `AutoBill` object to determine the amount for the bill.
- `Account`: An `Account` object can possess multiple token types. The `Account` object's `tokenBalances` data member houses the current amounts of different token types that are available to the `Account`.
- `AutoBill`: An `AutoBill` object may have a billing plan that is priced in terms of tokens of a certain type. If the payment method associated with the `AutoBill` is token-based, then the `AutoBill`'s recurring billings will be transacted in tokens.
- `Transaction`**:** A `Transaction` object may be for `Product` objects (used as transaction line items) that grant tokens. Tokens granted by each `Product` are defined by the `Product`'s `creditGranted` attribute. A transaction may also be performed in tokens, if the transaction uses a token-based payment method. The line items include the tokens that will be consumed when the transaction is captured.

  In refunding a transaction which originally granted tokens, you may leave the token balance unchanged (default), zero out the balance, or specify a negative balance for the token type. In refunding a transaction in tokens, CashBox adds the number of tokens to the customer's applicable token balance.

10.2 Understanding Token Activities

View token activities on the CashBox Portal by selecting **Search > Token Activity** from the menu bar, then specifying your company name. The four token activity types are:

- **Decrement**: When a customer accesses a service (downloads music or uses storage), make an API call to deduct the appropriate number of tokens.
- **Grant**: Grant tokens to a customer by making an API call or by performing a customer-service action of Grant Tokens on the CashBox Portal.
- **Purchase**: A one-time or recurring transaction that references a product with tokens granted, or a transaction whose payment method is tokens.
- **Refund**: Refund a purchase that was transacted in tokens or that was for a product that had tokens granted.

Table 10-3 Token Activity Types

Activity Type	Example	Description	Results	Token Balance
Decrement	The customer accesses a service, and the merchant reduces the customer's token balance.	Call the API's `decrementTokens` method on the customer account.	• Reduced token balance. • Updated token activity on customer's **Account Details** page. • Updated token balance for the related payment method. • Details on the **Token Activity Results** page.	Decremented
Grant	Add tokens to a customer's account for filling out a survey.	Call the API's `incrementTokens` method on the customer account.	• Increased token balance. • Updated token activity on customer's **Account Details** page. • New token payment methods, if they do not already exist. • Details on the **Token Activity Results** page.	Incremented
Grant	Add tokens to a customer's account to compensate for degraded service.	Grant tokens on the CashBox Portal.	• Notification email to the customer on the tokens granted. • New token payment methods, if they do not already exist. • Increased token balance. • Updated token activity on the customer's **Account Details** page. • Updated token balance for the related payment method. • Details on the **Token Activity Results** page.	Incremented

Table 10-3 Token Activity Types *(Continued)*

Activity Type	Example	Description	Results	Token Balance
Purchase	A customer makes a purchase that adds token types to the customer's account balance. The transaction is successfully captured.	Grant tokens to a one-time purchase.	• Notification email to the customer on the token activity and account balance. • New token payment methods, if they do not already exist. • Increased token-type balance. • Display of line items associated with each token type in the **Token Activity** table on the **Transaction Details** page. • Updated token activity on the customer's **Account Details** page. • Updated token balance for the related payment method. • Details on the **Token Activity Results** page. • Reference of the transaction ID by the activity.	Incremented
Purchase	A customer has an AutoBill subscription for a product that adds multiple token types to the customer's account balance. The transaction is successfully captured.	Grant tokens to an AutoBill subscription with a product that has associated tokens granted.	Same as above.	Decremented
Purchase	A customer makes a purchase for which you accept tokens as the payment method. Afterwards, you submit a transaction to CashBox and reference the token type.	Transact a one-time purchase in tokens.	• Notification to the customer: Real-Time Statement of Success or Real-Time Statement of Failure. • Reduced token-type balance. • Inclusion of the line items associated with the token type in the transaction. • Updated token activity on the customer's **Account Details** page. • Updated token balance for the related payment method. • Details on the **Token Activity Results** page.	Decremented

Table 10-3 Token Activity Types *(Continued)*

Activity Type	Example	Description	Results	Token Balance
Purchase	A customer has an AutoBill subscription for a product associated with a billing plan that is transacted in tokens. There are enough tokens in the related account to cover the cycle cost.	Transact an AutoBill subscription with a billing-period cycle in tokens.	• Prebilling notification to the customer. • A success notification to the customer if there are enough tokens in the account. Otherwise, the notification states that not enough tokens are available. • Reduced token-type balance. • Inclusion of the line items associated with the token type in the transaction. • Updated token balance for the related payment method. • Details on the **Token Activity Results** page. • Updated token activity on the customer's **Account Details** page.	Decremented
Refund	A customer makes a purchase with tokens and then requests a refund. (CashBox only supports *full* refunds for token transactions.)	Refund a real-time transaction that was paid for in tokens.	• Refund notification. • Updated transaction in question. • Increased token-type balances. • Updated token balance for the related payment method. • Details on the **Token Activity Results** page. • Updated token activity on the customer's **Account Details** page.	Incremented

Table 10-3 Token Activity Types *(Continued)*

Activity Type	Example	Description	Results	Token Balance
Refund	A customer who has an AutoBill subscription that references a billing plan transacted in tokens, requests a refund.	Refund an AutoBill transaction that was paid for in tokens.	• Refund notification. • Updated transaction in question. • Increased token-type balances. • Updated token balance for the related payment method. • Details on the **Token Activity Results** page. • Updated token activity on the customer's **Account Details** page.	Incremented
Refund	A customer purchases a product that has tokens granted and then requests a refund.	Refund a one-time purchase transacted in currency for a product for which tokens were granted.	• Refund notification. • Reduced token-type balances. • Updated token balance for the related payment method. • Details on the **Token Activity Results** page. • Updated token activity on the customer's Account Details page.	Decremented
Refund	A customer has an AutoBill subscription for a product that has tokens granted and requests a refund.	Refund of an AutoBill subscription transacted in currency for a product for which tokens were granted.	• Refund notification. • Reduced token-type balances. • Updated token balance for the related payment method. • Details on the **Token Activity Results** page. • Updated token activity on the customer's Account Details page.	Decremented

10.3 Defining New Token Types

Define new token types by instantiating `Token` objects and making API calls to specify the new types in the CashBox database.

Define a new Token type:

```
$tok = new Token();

// Use a unique id when defining a new token type
$tok->setMerchantTokenId("US_FREQ_BOOK_BUYER_PT");

$tok->setDescription("A frequent book buyer point for US customers");

// Make SOAP call to create token in CashBox database
$response = $tok->update();

if($response['returnCode']==200)
{
    print "Token created successfully";
}
```

10.4 Incrementing Token Balances

You can increment a customer account's token balance in several ways: through a successfully captured transaction (purchase), through a refund of a transaction that was paid for in tokens, with a grant-tokens call through a customer-service interaction, or with the `increment` method.

If a token balance is incremented but does not exist in the related `Account` object, CashBox creates a new `PaymentMethod` object of type `Token`.

10.4.1 Purchasing Tokens

Because token balances are associated with an `Account` object, to allow customers to purchase tokens, you must define `Product` objects and specify how many tokens of a certain type to grant to a customer (`Account` object) when that customer purchases the related products. After each purchase, CashBox adds the appropriate number of tokens to the balance attached to the `Account` object.

The following example creates a `Product` object that grants 10 tokens, of the type defined in the previous example, to an `Account` object that purchases the `Product` object.

Use a Product purchase to grant Tokens:

```
$prod = new Product();
$prod->setMerchantProductId("sku101");

// Populate various product attributes here
$tok = new Token();

// we created token with this id already - we want to refer to the
// same token type

$tok->setMerchantTokenId("US_FREQ_BOOK_BUYER_PT");

// create a TokenAmount object and populate it with token type
// and quantity
$tokAmt = new TokenAmount();
$tokAmt->setToken($tok);
$tokAmt->setAmount(10);

// we need an array of token amounts since a product can
// grant multiple token types. However, in our example the
// product grants only one token type
$tokAmounts = array($tokAmt);

$cr = new Credit();
$cr->setTokenAmounts($tokAmounts)

// Specify token amounted granted by purchase of this product
$prod->setCreditGranted($cr);

// Make SOAP call to create the product in CashBox database
$response = $prod->update();

if($response['returnCode']==200) {
   print "Product created successfully";
}
```

The `Product` object created can be used in both one-time and recurring billing:

- **Recurring billing**: Construct an `AutoBill` object with the `Product` object. With successful capture of each recurring transaction generated by the `AutoBill`, CashBox adds the tokens granted by the `Product` to the `Account` object associated with the `AutoBill`.
- **Real-time billing**: Construct a one-time transaction that refers to the `Product` object as one of its transaction items. Specify the SKU of `TransactionItem` as `merchantProductId` for `Product`. With a successful capture of a one-time transaction, CashBox adds the tokens granted by `Product` to the `Account` object associated with the transaction.

Use a Product that grants Tokens in a one-time Transaction:

```
$tx = new Transaction();

// Reference an existing account to which tokens are to be granted
$account = new Account();
$account->setMerchantAccountId('9876-5432');
$tx->setAccount($account);

// One of the line items of the transaction should be the product
// that grants tokens
$tx_item = new TransactionItem();
$tx_item->setSku('sku101'); // the id of the product that grants tokens
$tx_item->setName('Token granter product');
$tx_item->setPrice(75.00);
$tx_item->setQuantity(1);
$tx->setTransactionItems(array($tx_item));

// set other transaction attributes here

$sendEmailNotification=false;
$response = $tx->authCapture($sendEmailNotification);
    // SOAP call
if($response['returnCode']==200) {
    if($tx->statusLog[0]->status=='Authorized') {
        print "Purchase complete. Tokens granted";
    }
}
```

10.4.2 Granting Tokens to Accounts

To grant a customer tokens outside the framework of recurring or real-time billing transactions (for example, to compensate a customer for an issue, or to reward a customer for completing a survey) the `Account` object supports calls to directly increment or decrement token balances.

Increment an Account's Token balance:

```
// Reference an existing account to which tokens are to be granted
$acct = new Account();
$acct->setMerchantAccountId('9876-5432');

// Refer to an existing token type using its id
$tok = new Token();
$tok->setMerchantTokenId("US_FREQ_BOOK_BUYER_PT");

// create a TokenAmount object and populate it with token type and
// quantity
$tokAmt = new TokenAmount();
$tokAmt->setToken($tok);
$tokAmt->setAmount(5);
// want to award the Account with 5 tokens of this type

// Refer to another existing token type using its id
$tok2 = new Token();
    $tok2->setMerchantTokenId("US_FREQ_DVD_BUYER_PT");

// create a TokenAmount object and populate it with token type and
// quantity
$tokAmt2 = new TokenAmount();
$tokAmt2->setToken($tok2);
$tokAmt2->setAmount(2);
// want to award the Account with 2 tokens of this type

    $tokAmounts = array($tokAmt, $tokAmt2);

// make the SOAP call to increment tokens
$response = $acct->incrementTokens($tokAmounts);

if($response['returnCode']==200) {
    // the call returns new token balances on the account
    // print those out
    $newTokBalances = $response['tokenAmounts'];
    foreach ($newTokBalances as $newTokBal) {
        print "Token type"
             . $newTokenBal->token->merchantTokenId; . "\n";
        print "Token amount available" . $newTokenBal->amount;
             . "\n";
    }
}
```

The drawback of incrementing tokens on an `Account` object is that the action is not registered in CashBox's transaction framework. It is, however, tracked and available on the CashBox Portal (choose **Search > Token Activity**). You may also obtain all token activities with SOAP 3.5 API calls.

10.5 Decrementing Token Balances

Your customers can spend tokens by purchasing a product that is priced in tokens, or by signing up for a subscription (represented by an `AutoBill` object) with a billing plan priced in tokens.

Reduce a customer's token balance, as the result of a billing event, by making a `decrement` call. Because `decrement` calls are made directly on `Account` objects, the action lies outside of Vindicia's transaction framework, with no accounting of the spent tokens unless you develop separate client-side mechanisms to maintain an audit trail. However, Vindicia tracks the action and makes it available on the CashBox Portal (choose **Search > Token Activity**). You may also obtain all token activities with SOAP 3.2 API calls.

Decrement an Account's Token balance:

```
// Reference an existing account from which the tokens
// are to be decremented
$acct = new Account();
$acct->setMerchantAccountId('9876-5432');

// Refer to an existing token type using its id
$tok = new Token();
$tok->setMerchantTokenId("US_FREQ_BOOK_BUYER_PT");

// create a TokenAmount object and populate it with token type and
// quantity
$tokAmt = new TokenAmount();
$tokAmt->setToken($tok);
$tokAmt->setAmount(20); // remove 20 tokens from the account's balance

// Refer to another existing token type using its id
$tok2 = new Token();
$tok2->setMerchantTokenId("US_FREQ_DVD_BUYER_PT");

// create a TokenAmount object and populate it with token
// type and quantity
$tokAmt2 = new TokenAmount();
$tokAmt2->setToken($tok2);
$tokAmt2->setAmount(40);
// want to decrement 40 tokens from the account's balance
$tokAmounts = array($tokAmt, $tokAmt2);

// make the SOAP call to decrement tokens
$response = $acct->decrementTokens($tokAmounts);

if($response['returnCode']==200) {
    // the call returns new token balances on the account
    // print those out
    $newTokBalances = $response['tokenAmounts'];
    foreach ($newTokBalances as $newTokBal) {
        print "Token type"
             . $newTokenBal->token->merchantTokenId; . "\n";
        print "Token amount available" . $newTokenBal->amount; .  "\n";
    }
}
```

10.5.1 Transacting Purchases in Tokens

You may also decrement a customer's token balance by conducting a transaction with the `TokenTransaction` object. Unlike the `Transaction` object, which works with both tokens and money (currency), `TokenTransaction` deals only with tokens. The advantage of using `TokenTransaction` over `Transaction` is that you need not define a token-based `BillingPlan` or `Product` object. Use `TokenTransaction` if you wish to take advantage of CashBox's token ledger for token accounting and record-keeping, but do not have `BillingPlan` objects explicitly priced in tokens, or do not want to browse transaction reports and make refunds on the CashBox Portal.

Conduct multiple Token transactions, for multiple Token types, in a single call:

```
//Create a new TokenTransaction object
$tokTxn1 = new TokenTransaction();

// Reference an existing account to which this transaction is to be
// applied
$acct = new Account();
$acct->setMerchantAccountId('9876-5432');

$tokTxn1->setAccount($acct);

// Specify information about the token type that will be used
// for this transaction
$tok1 = new Token();
$tok1->setMerchantTokenId("US_FREQ_BOOK_BUYER_PT");
   $tokAmt1 = new TokenAmount();
$tokAmt1->setToken($tok1);
$tokAmt1->setAmount(4); // number of tokens used with this transaction

$tokTxn1->setTokenAmount($tokAmt1);

$tokTxn1->setDescription("Purchase: Infinite Jest-Paperback");

$tokTxn2 = new TokenTransaction();

$tokTxn2->setAccount($acct);
// Specify information about the tokens that will be used for this
// transaction
$tok2 = new Token();
$tok2->setMerchantTokenId("US_FREQ_BOOK_BUYER_PT");
   $tokAmt2 = new TokenAmount();
$tokAmt2->setToken($tok2);
$tokAmt2->setAmount(3); // Number of tokens used with this transaction

$tokTxn2->setTokenAmount($tokAmt2);
$tokTxn2->setDescription("Purchase: Blink-Paperback");
$tokTxns = array($tokTxn1, $tokTxn2);

// make the SOAP call to perform the token transaction
// Ensure that the Account set in each TokenTransaction object is
// the same object on which you make the following SOAP call

$response = $acct->tokenTransaction($tokTxns);
```

```
if($response['returnCode']==200) {
    // the call returns new token balances on the account.
    // print the new balances.
    $newTokBalances = $response['tokenAmounts'];

    print "New token balances for account with id "
        . $acct->merchantAccountId . "\n";
    foreach ($newTokBalances as $newTokBal) {
        print "Token type"
             . $newTokenBal->token->merchantTokenId; . "\n";
        print "Token amount available" . $newTokenBal->amount; . "\n";
    }
}
```

10.5.2 Token Transactions in Real Time

To track your customers' token transactions in CashBox, execute standard `Transaction` objects. For example, if your customer makes a one-time purchase for which the price is a token type instead of currency, construct a real-time `Transaction` object.

Note:

- The `Transaction` object's `sourcePaymentMethod` attribute must refer to a `PaymentMethod` object called `Token`.
- The transaction items in the `Transaction` object must all be for amounts that are in tokens.
- Token transactions must have a `_VT` setting for `currency`, which is Vindicia's internal code for token-based payment.
- You ***cannot*** mix items priced in currency and tokens. Apply only a token type to the `Transaction` object.

Create and process a real-time Transaction:

```
//Create a new Transaction object
$tx = new Transaction();

// Reference an existing account to which tokens are to be granted
$account = new Account();
$account->setMerchantAccountId('9876-5432');
$tx->setAccount($account);

// One of the line items of the transaction must be the
// product that grants tokens
$tx_item = new TransactionItem();
$tx_item->setSku('sku112');
$tx_item->setName('Product priced in token);
$tx_item->setPrice(2); // this is the number of tokens
$tx_item->setQuantity(1);
$tx->setTransactionItems(array($tx_item));

// The source payment method used for the transaction must be
// a token payment method
$srcPm = new PaymentMethod ();
$srcPm->setType('Token');
```

```
// because the product we want to purchase uses tokens of a
// certain type, create Token objects of that type to specify
// in the payment method
$tok = new Token();
$tok->setMerchantTokenId("US_FREQ_BOOK_BUYER_PT");

$srcPm->setToken($tok);

// set the payment method in the transaction

$tx->setSrcPaymentMethod($srcPm);

// set currency of the Transaction to be tokens

$tx->setCurrency('_VT');

// set other transaction attributes here

…

$sendEmailNotification=false;
$response = $tx->authCapture($sendEmailNotification);
// SOAP call

if($response['returnCode']==200) {
    if($tx->statusLog[0]->status=='Authorized') {
        print "Purchase complete";
    }
}
```

10.6 Handling Recurring Billing with Tokens

To enable customers to pay for recurring billing with tokens, construct an `AutoBill` object with a `BillingPlan` object that is transacted in tokens, and a token-based `PaymentMethod` object.

Create a Billing Plan priced in Tokens:

```
//Create a new BillingPlan object
$bp = new BillingPlan();

// first, create a TokenAmount object that will be used as price of
// the billing plan
$tok = new Token();
$tok->setMerchantTokenId("US_FREQ_BOOK_BUYER_PT");
   $tokAmt = new TokenAmount();
$tokAmt->setToken($tok2);
$tokAmt->setAmount(2);

// Price of the billing plan in terms of tokens
$price = new BillingPlanPrice();
$price->setTokenAmount($tokAmt);

// create a billing plan period that uses the token-based price
$bperiod = new BillingPlanPeriod();
$bperiod->setType('Month');
$bperiod->setQuantity(1); // a billing period of 1 month
$bperiod->setCycles(0);  // infinite
$bperiod->setPrices(array($price));
// token-based price as defined above

// now create a billing plan that uses the period that uses
// token-based price
$bp->setMerchantBillingPlanId("bpid-111");
$bp->setDescription("Token priced billing plan");
$bp->setPeriods(array($bperiod)); // use the period defined above

// set other billing plan attributes below

…

// Next, create an AutoBill object for the BillingPlan object:
$abill = new AutoBill();

// Reference an existing account. The monthly transaction will
// deduct tokens from this account
$acct = new Account();
$acct->setMerchantAccountId('9876-5432');
$abill->setAccount($acct);

// Use the token-based billing plan created above
$abill->setBillingPlan($bp);

// you may also reference a Product that grants Tokens for
// an AutoBill (use this to exchange one type of tokens for
// another)
```

```
// The payment method used for the autobill must be a token-based
// payment method
$pm = new PaymentMethod ();
$pm->setType('Token');

// since the billing plan uses tokens of certain type, create
// Token object of that type to specify in the payment method
$tok = new Token();
$tok->setMerchantTokenId("US_FREQ_BOOK_BUYER_PT");
$pm->setToken($tok);

// set the payment method in the autobill
$abill->setPaymentMethod($pm);

// Populate other autobill attributes here
…
$dupBehave = 'Duplicate';
$minChargebackProb = 100; // don't check for risk screen
$validatePm = false;
       // do not validate payment method since it is tokens

// Make SOAP call to create the autobill
$response = $abill->update($dupBehave, $validateP, $minChargebackProb);

if($response['returnCode']==200) {
   print 'AutoBill created!';
}
```

The `AutoBill` object you created generates monthly transactions that decrement tokens from the related `Account` object. If you fetch the transactions with the CashBox API, note the following:

- The `PaymentMethod` object in the `Transaction` object's `sourcePaymentMethod` attribute has a type setting of `Token`. The `token` attribute of the `PaymentMethod` object contains the token type for the transaction.
- The `currency` attribute of the `Transaction` object is `_VT`, CashBox's internal currency code for tokens.
- The `Transaction` object's `amount` attribute refers to the total number of tokens used (decremented from the `Account` object in question) for the transaction.

Process a Token-based Transaction fetched from CashBox:

```
// Create a transaction object so we can make the fetch call
$soapTx = new Transaction();
// now load the most recently changed 100 transaction records,
// using a date prior to Jan 1 1970
$paymentMethod = null;
       // do not filter returned transactions by payment method
$since = '1969-01-01T00:00:00Z';
$today = '2009-05-19 T00:00:00Z'; // until today
$pageNumber = 0; // want to get the first page
$pageSize = 100; // want 100 records in a page
$response = $tx->fetchDeltaSince($since, $today, $paymentMethod,
       $pageNumber, $pageSize);
if($response['returnCode'] == 200) {
    $fetchedTxs = $response['transactions'];
    foreach ($fetchedTxs as $fetchedTx) {
       $srcPm = $fetchedTx->sourcePaymentMethod;
       if (srcPm->type == 'Token'&& $fetchedTx->currency == '_VT')
       {
       // This is a token-based transaction
       print "This transaction used tokens of type "
            . $srcPm->token->merchantTokenId . "\n";
       print "Amount of tokens decremented by this transaction "
            . $tx->amount;
       }
    }
}
```

10.7 Refunding Transactions in Tokens

The CashBox API allows you to refund transactions that granted or decremented tokens. Refunding token transactions returns the tokens to the token balance in the customer's account.

If a refund is for a `Product` that granted tokens, be sure to define the action to take for the tokens. When creating the `Refund` object, specify the `tokenAction` attribute, which is an enumeration of the following three values:

- `None`: Makes no changes to the token balance. All previous token balances will stand as if the transaction had not been refunded.
- `CancelZeroBalance`: CashBox will cancel all previous token transactions. If the cancellation causes the token balance to drop below zero, CashBox will reset it to zero.
- `CancelNegativeBalance`: CashBox will cancel all previous token transactions. If the cancellation causes the token balance to drop below zero, the negative balance will remain, and subsequent token-based transactions will fail until the balance rises above zero.

Refund a Transaction that granted Tokens to an Account:

```
//Create a new refund object
$refund = new Refund();

$txn = new Transaction();

// specify a known transaction by its merchant ID. This transaction
// should be in the 'Captured' state so it can be refunded
$txn->setMerchantTransactionId('WID-CUS-9302871');

// associate the account and refund objects
$refund->setTransaction($txn);

// set the amount of the refund
$refund->setAmount(10.00);
$refund->setTimestamp('2009-02-11T22:34:32.265Z');
$refund->setReferenceString('myRefundId101');
$refund->setTokenAction('CancelNegativeBalance');

$tempSoapRef = new Refund();

// refund the transaction
$response = $tempSoapRef ->perform(array($refund));
```

10.8 The CashBox Token Processor

Token transactions are processed using Vindicia's Token Processor. Like other payment processors, the Token Processor returns reason codes for transaction requests, as shown below.

Table 10-4 Token Processor Reason Codes

Reason Code	Description
000	Approved.
001	Fractional funds.
002	No customer tokens.
003	Insufficient funds.
004	Authorization failed.

The reason code describes whether a token transaction succeeded or failed, and suggests appropriate action. For example, in the case of an insufficient token balance, direct the customer to a site to purchase a product that grants the token type in question.

11 Working with Campaigns

CashBox Campaigns allow you to offer discounts on existing Products, or time grants to existing AutoBills. Discounts may be currency or percentage based, and may be single purchase, or period based offers. Time grants serve to grant customers free time extensions to their existing AutoBills.

Use the CashBox Portal to define Campaign parameters, then generate and retrieve Campaign Codes. Use the CashBox API to apply a (Coupon or Promotion) Campaign to an AutoBill.

CashBox offers several ways in which a Campaign discount may be applied to an AutoBill.

- Use `AutoBill.update` to create an `AutoBill` and apply a Campaign discount Code simultaneously.
- Use `AutoBill.addCampaign` to apply a Campaign Code to an existing `AutoBill`.
- Use `AutoBill.modify` to add a new Product, and its corresponding Campaign discount, to an existing `AutoBill`.

11.1 Creating an AutoBill with a Campaign discount

CashBox allows you to create an `AutoBill` and apply a Campaign discount Code in a single operation, using `AutoBill.update`.

Create an AutoBill with an attached Campaign Code:

```
$autobill = new AutoBill();
$response = $autobill->update(
    'SucceedIgnore',// Duplicate Behavior
    false,        // validate PaymentMethod?
    100,          // min Chargeback Probability
    true,         // ignore Avs Policy?
    true,         // ignore Cvn Policy?
    'promoABC'    // promoCode
);

// check $response
```

Note: This example neither validates the Payment Method, nor checks the chargeback probability or AVS and CVN returns. These parameters must be populated so that the `promoCode` is the 6th parameter.

11.2 Adding a Campaign Code to an AutoBill

Both `AutoBill.update` and `AutoBill.addCampaign` may be used to apply a Campaign discount to an existing AutoBill. With these methods, CashBox will automatically apply the Campaign discount to all eligible Products that do not yet have a discount applied.

For more information on discount calculation, see the Chapter 10: Campaigns in the ***CashBox User Guide***.

If there are any eligible Products on an existing AutoBill, a Campaign discount may be applied toward it, regardless of the history of the AutoBill, or the length of its duration.

11.2.1 Applying a Campaign Code to an existing AutoBill

Use `AutoBill.addCampaign` to add a Campaign Code to an existing AutoBill.

Add a Campaign Code to an existing AutoBill:

First, create the AutoBill:

```
$autobill = new AutoBill();
$autobill->update(
    'SucceedIgnore',// Duplicate Behavior
    false,        // validate PaymentMethod?
    100,          // min Chargeback Probability
    true,         // ignore Avs Policy?
    true,         // ignore Cvn Policy?
    null          // NO promoCode
);
```

> **Note:** This example neither validates the Payment Method, nor checks the chargeback probability or AVS and CVN returns. The default behavior for ***duplicateBehavior*** is used. These parameters must be populated, simply to set a variable for the last in the sequence (***ignoreAvsPolicy***).

To attach the Campaign Code, use `AutoBill.update`, or `AutoBill.addCampaign`. (`AutoBill.addCampaign` works similarly to `AutoBill.update` when adding a Campaign Code to an existing AutoBill.)

```
$response = $autobill->addCampaign(
    'promoABC'      // promoCode
    );

// check $response
```

If there is an error with the discount, CashBox will not process the `AutoBill.update`, and will return an error code of 400, with a text string explanation. The error may be that no Product on the AutoBill is eligible for the Promotion, or that the Coupon Code has already been used against its defined number of AutoBills, or that the Promotion Code is out of date.

11.2.2 Applying a Campaign Code to a Specific Product on an AutoBill

Using `AutoBill.addCampaign` will automatically calculate the Product(s) to which the Campaign should apply, based on the eligible products, as defined in the Campaign.

To *specify* the Product to which a Campaign Code should be applied, use `AutoBill.modify` to add a Product and Campaign Code simultaneously.

Add a Product, with an attached Campaign Code, to an AutoBill:

Using `modify` with a Campaign Code as a parameter forces the discount to be applied to the added Product. It will not be applied to any other Product on the AutoBill.

```
$response = $autobill->modify(
    $ab_item// AutoBillItem
    false, // addNextPeriod
    false, // proRate
    'promoABC'// promoCode
);

    // check $response
```

Using `modify` without specifying the ***campaignCode*** parameter, then using `addCampaign` against the same AutoBill, will apply the Campaign discount to all Products in the AutoBill eligible for the discount.

12 Credit Grants and Gift Cards

CashBox supports multiple payment methods, such as credit cards, PayPal, and electronic checks, that you may attach to an `AutoBill` object. In a recurring billing model, CashBox uses the `PaymentMethod` object as the source Payment Method for the periodic billing transactions it processes with the payment processor, charging the customer for every periodic bill.

CashBox also allows you to place credit on an `AutoBill`, or on a customer's `Account`. Any transaction generated by an `AutoBill` will first attempt to use credits on that `AutoBill`, or on the customer's `Account`, and bill the `AutoBill PaymentMethod` for the remaining balance after available credit is redeemed. CashBox may also automatically adjust *one-time* transactions by the amount of credit available on a customer's `Account`, redeem the appropriate credits, then adjust the transaction for the remaining balance.

Credits are stored in the form of *tokens* (see Section 17: The Token Object in the ***CashBox API Guide***), *time* or *currency*. Token and currency credits are automatically deducted from available balances when a transaction is processed using the same token type or currency. Time credits serve to delay an `AutoBill`'s billing process by the amount of time granted.

CashBox redeems Credit automatically when Transactions are processed, in an order determined by the Credit's Type, sort value, and timestamp. Time and Currency credits may also be assigned a sort value upon grant, which may be used to customize the order in which they are redeemed.

All credit activity is marked with a timestamp, which allows you to analyze your Credit process.

Add credits to an `Account` or `AutoBill` directly using the CashBox Portal or an API call, or indirectly by making a call to redeem a gift card. (CashBox currently supports gift cards issued by InComm.)

12.1 Working with Credit

The `Credit` object encapsulates the different types of credit that can be stored on a `Product`, `Account` or `AutoBill`, and applied to one-time or recurring transactions.

- **Tokens:** Tokens may be used as both payment methods, and credit options. The `merchantTokenId` attribute of a `Token` object identifies its type. Use this data member to define your `Token` types.

  For example, create an `AutoBill` with a monthly `BillingPlan` whose price is defined in terms of `Tokens` of a certain type. Create a `Product` object which grants credit for a number of Tokens of that type. Allow your customer to purchase the Product, and acquire the allotted Tokens. Then, each time the `AutoBill` is due , CashBox will deduct the specified number of `Tokens` from the available `Token` credit.
- **Credits (time):** Grant time credit to an `AutoBill` by including an array of `TimeInterval` objects in the `Credit` object associated with the `AutoBill`. The `TimeInterval` object allows you to define a time extension to an `AutoBill` in terms of years, months, weeks, or days. When you grant time credit to an `AutoBill`, CashBox delays the next billing for the `AutoBill` by the specified amount of time, similar to calling `delayBillingByDays()` on the `AutoBill` object.
- **Credits (currency):** CashBox enables you to grant or revoke currency credits currency to `Accounts` and `AutoBills`. All ISO 42171 currencies are supported. For more information, see the `Credit` Subobject in the ***CashBox API Guide***.
- **Gift Cards:** Add Token credits to an `AutoBill` or an `Account` by redeeming a gift card. Your customers may redeem gift cards, through CashBox, to add credit to their `Account`. For more information, see Section 12.2: Working with Gift Cards.

12.1.1 Redeeming Credit

CashBox automatically processes all credit redemption. Neither the CashBox API, nor the CashBox Portal allow you to manually redeem Credit. CashBox does, however, provide you with rules which you may use to determine the order in which Credits are redeemed.

Time and Currency credits may be assigned a Sort Value when they are granted. They also carry a VID and timestamp. Use these values to both track your credit grants and revocations, and to define the order in which granted Credits should be redeemed when AutoBill Transactions are processed.

CashBox processes Credit redemption in the following order:

1. Time Interval credits are redeemed before currency credits.
2. Credits automatically granted by CashBox are redeemed before those granted through the CashBox Portal or API.
3. Credits are redeemed based on Sort Value. Use this value to define the order in which you would like Credits to be redeemed.
4. After Sort Value has been considered, Credits are redeemed based on Grant time, from oldest to most recent.

Credits may not be revoked after they've been redeemed, and they may not be redeemed after they've been revoked.

12.1.2 Using Credits with an Account

The read-only credit attribute of the Account object holds the current credits available to the Account. Use the grantCredit and revokeCredit calls to manipulate credit. These methods generate appropriate audit trails of credit changes. Do not set the value of this attribute directly. (For example, when creating a new Account object by calling update().)

Note: Use token-related methods such as Account.incrementTokens() or decrementTokens() to manipulate tokens available to the Account. CashBox can handle tokens both as payment methods outside the credit framework, and as part of the credit system.

Although you may use Tokens interchangeably across the two systems, Vindicia recommends that in your implementation, you choose one system, rather than mixing the two. If you have not previously implemented a token-as-payment-method system, choose credit-based token management, because the Credit object allows you to abstract the type of credit, and therefore offers more flexibility for future replacement.

Granting Credit to an Account

If a customer performs a specific activity at your site, you may choose to give that customer credit. For example, if a customer redeems a phone card, a cell phone company may add a number of cell phone minutes to the customer's account. You may also apply currency credits to the account. Use the Credit's name-value pairs to define the reason for the Grant, and to provide another field by which your Credit activity may be analyzed.

Use `Account.grantCredit` to add credit to an Account:

```
$acct = new Account();

// account id for an existing customer

$acct->setMerchantAccountId('jdoe101');
$tok = new Token();

// specify the id of an existing token type.
// (the assumption here is that you have already created
// a Token object with this id.)

$tok->setMerchantTokenId('ANYTIME_PHONE_MINUTES_2010');

$tokAmt = new TokenAmount();
$tokAmt->setToken($tok);
$tokAmt->setAmount(100);

$cr = new Credit();
$cr->setTokenAmounts(array($tokAmt));

// make the SOAP API call to grant credit to the acct
$response = $acct->grantCredit($cr);

if ($response['returnCode'] == 200) {

    // Credit successfully granted to the account

    $updatedAcct = $response->['account'];
    $availableCredits = $updatedAcct->getCredit();
    $availableTokens = $availableCredits->getTokenAmounts();

    print "Available token credits: \n";
    foreach($availableTokens as $tkAmt) {
        print "Token type: " . $tkAmt->getMerchantTokenId() . " ";
        print "Amount: " . $tkAmt->getAmount() . "\n";
    }
}
else {

    // Error while granting credit to the account

    print $response['returnString'] . "\n";
}
```

> **Note:** CashBox does not allow you to grant time-based credit to an `Account` object.

Revoking Credit from an Account

For some customer activities, you might revoke credit from the customer's account. For example, if a customer uses airline frequent flier miles to book a flight, an airline company would deduct frequent flier miles from the customer's account. If you accept currency credits, you may also deduct currency from the account.

Use `Account.revokeCredit` to deduct credit from an Account:

```
$acct = new Account();

// account id for an existing customer

$acct->setMerchantAccountId('ff_flier_101');

$tok = new Token();

// specify the id of an existing token type.
// (the assumption here is that you have already created
// a Token object with this id.)

$tok->setMerchantTokenId('UA_FF_MILES');

$tokAmt = new TokenAmount();
$tokAmt->setToken($tok);
$tokAmt->setAmount(25000);

$cr = new Credit();
$cr->setTokenAmounts(array($tokAmt));

// make the SOAP API call to deduct miles
$response = $acct->revokeCredit($cr);

if ($response['returnCode'] == 200) {

    // Credit successfully revoked from the account

    $updatedAcct = $response->['account'];
    $availableCredits = $updatedAcct->getCredit();
    $availableTokens = $availableCredits->getTokenAmounts();

    print "Available token credits: \n";
    foreach($availableTokens as $tkAmt) {
        print "Token type: " . $tkAmt->getMerchantTokenId() . " ";
        print "Amount: " . $tkAmt->getAmount() . "\n";
    }
}
else {

    // Error while revoking credit from the account
    print $response['returnString'] . "\n";
}
```

> **Note:** If the amount of token-based credit to be revoked results in a negative balance, CashBox sets the balance to 0.

Using Credits for a One-Time Transaction

One-time transactions may use credit available to an Account. To use Credits for one-time transactions, make certain that the Transaction object includes a PaymentMethod that matches the currency or token type you wish to use to conduct the transaction. If the customer Account has enough credit to support the transaction, CashBox authorizes the transaction, and adds a new transaction item to the Transaction object returned to you in response to your authCapture() call. This transaction item has its sku attribute set to VIN_Credit, and its price set to a negative value (equivalent to the amount of the transaction), signifying that CashBox deducted the credit from the customer's account.

Create a one-time credit transaction:

```
$acct = new Account();

// account id for an existing customer

$acct->setMerchantAccountId('jdoe101');

$tok = new Token();

// specify the id of an existing token type.
// (the assumption here is that the customer with
// account id 'jdoe101' has credit available in this
// type of token.)

$tok->setMerchantTokenId('ANYTIME_PHONE_MINUTES_2010');

// source payment method for the transaction should be
// a token-based payment method
$srcPm = new PaymentMethod();
$srcPm->setType('Token');
$srcPm->setToken($tok);
$srcPm->setMerchantPaymentMethodId('5933054820');

$txn = new Transaction();
$txn->setAccount($acct);
$txn->setSourcePaymentMethod($srcPm);

// This transaction will deduct 50 credit units from the account
$txn->setAmount(50);

$txn->merchantTransactionId('TK00234918'); // must be unique

// Add a transaction item describing the transaction detail

$txItem = new TransactionItem();
$txItem->setSku('MONTH50');
$txItem->setName('Monthly Anytime Minutes');
$txItem->setPrice(50);
$txItem->setQuantity(1);

// set the transaction item into the transaction
$txn->setTransactionItems(array($txItem);

$sendEmail = false;
```

```
// Make the SOAP call to authorize and capture the transaction
$response = $txn->authCapture($sendEmail);

if ($response['returnCode'] == 200) {

    // the SOAP call succeeded. Now check if the
    // transaction was authorized
    $retTxn = $response->['transaction'];
    if($retTxn->statusLog[0]->status=='Authorized') {
        print "Transaction approved";
    }
    else if($retTxn->statusLog[0]->status=='Cancelled') {
        print "Transaction not approved \n";
    }
    else {
        print "Error: Unexpected transaction status\n";
    }
}
else {
    // Error while conducting transaction
    print $response['returnString'] . "\n";
}
```

Fetching Account Credit History

CashBox maintains a log of credit-related events for each Account. This log keeps track of various credit-related events such as credit granted, revoked, consumed, or earned from a gift card redemption. Retrieve the audit log by calling the Account object's fetchCreditHistory() method. The method includes paging and a time range, so you can limit the number of records returned. It returns an array of CreditEventLog objects. Each CreditEventLog object holds the Credit object, a timestamp, the type of activity performed with the Credit object, and a note text field.

Call Account.fetchCreditHistory:

```
$acct = new Account();

// account id for an existing customer whose
// credit history you want to retrieve

$acct->setMerchantAccountId('jdoe101');

$page = 0; // paging begins at 0
$pageSize = 5; // five records
$startTime = '2010-01-01T22:34:32.265Z';
$endTime = '2010-01-30T22:34:32.265Z';

do {
    $ret =
        $acct->fetchCreditHistory($startTime, $endTime $page, $pageSize);
    $count = 0;
    if ($ret['returnCode']== 200) {
        $fetchedLogs = $ret['creditEventLogs'];
        $count = sizeof($fetchedLogs);
        foreach ($fetchedLogs as $log) {
              $credit = $log->getCredit();
             $ts = $log->getTimeStamp();
             $eventType = $log->getType();
             // process retrieved credit event log
             // details here.
        }
        $page++;
    }
} while ($count > 0);
```

12.1.3 Using Credits with an AutoBill

The read-only credit attribute of the AutoBill object holds the array of Credit amount objects available to the AutoBill. Do not set this field directly by calling AutoBill.update. Instead, use AutoBill methods such as grantCredit() and revokeCredit() to alter credits available to the AutoBill, and provide an audit trail in the credit log. Currency-, time-, and token-based credits may be used with AutoBill objects. CashBox manages credits available to an AutoBill object as follows:

- CashBox draws currency- and token-based credits from the AutoBill for each periodic transaction it generates for the AutoBill.
- If the Account associated with the AutoBill also has currency- or token-based credits available to it, when the AutoBill is billed, AutoBill credits are redeemed before Account credits.
- If an AutoBill has an associated time credit, CashBox uses this credit before redeeming any token-based or currency credits.
- CashBox uses currency and token-based credit to process an AutoBill's periodic transaction only if *both* of the following are true:
  - The payment method specified on the AutoBill is a currency or token-based payment method that specifies a type of applicable token,

  and

  - The billing plan associated with the AutoBill has a price listed in terms of the same token type or currency.
- When CashBox applies time-based credit to an AutoBill, it adjusts the next billing date of the AutoBill accordingly. For example, a 15-day credit will postpone an AutoBill object's next billing date by 15 days. Application of such a credit does not generate a transaction.

Granting Credit to an AutoBill

Some situations may require you to extend a customer's subscription to your site. For example, if a technical problem at your site prevented a customer from accessing the site for two days, a customer service representative might decide to extend the customer's subscription by two days, to guarantee customer satisfaction. The grantCredit() method of the AutoBill object allows you to add time credit to an AutoBill object to implement such an extension.

Use grantCredit to add time credit to an AutoBill:

```
$abill = new AutoBill();

// autobill id for an existing subscription

$abill->setMerchantAutoBillId('SBCR312345');

// We want to grant 2 days of credit
$time = new TimeInterval();
$time->setType('Day');
$time->setAmount(2);

$cr = new Credit();
$cr->setTimeIntervals(array($time));

// Now make the SOAP API call to grant credit to the autobill
$response = $abill->grantCredit($cr);

if ($response['returnCode'] == 200) {

    // Credit successfully granted to the autobill

    $updatedABill = $response['data']->autobill;

    print "Current entitlements are valid till: ";
    print $updatedABill->getEndDate() . "\n";
}
else {
    // Error while granting credit to the account
    print $response['returnString'] . "\n";
}
```

Revoking Credit from an AutoBill

Some activities a customer performs at your site might revoke credit from an AutoBill object. For example, an online game company might offer subscriptions that are paid for by points a customer earns in the game. If the customer loses some points in the game, the company might deduct the same number of points from the customer's subscription. (Note: Time-based credit granted to an AutoBill cannot be revoked.)

Use revokeCredit to revoke currency or Token-based credits from an AutoBill:

```
$abill = new AutoBill();

// autobill id for customer's existing subscription to a game

$abill->setMerchantAutoBillId('STARWARS-239181');

$tok = new Token();

// specify the id of an existing token type.
// the autobill has a payment method defined in terms of this
// token type. Also the billing plan used by the autobill
// specifies a price in terms of this token type.

$tok->setMerchantTokenId('STARWARS_POINTS');

$tokAmt = new TokenAmount();
$tokAmt->setToken($tok);
$tokAmt->setAmount(100); // customer lost 100 points in the game

$cr = new Credit();
$cr->setTokenAmounts(array($tokAmt));

// Now make the SOAP API call to deduct points from customer's
// subscription

$response = $abill->revokeCredit($cr);

if ($response['returnCode'] == 200) {

    // Credit successfully revoked from the autobill

    $updatedAbill = $response->['data']->autobill;
    $availableCredits = $updatedAbill->getCredit();
    $availableTokens = $availableCredits->getTokenAmounts();

    print "Available points to subscription: \n";
    foreach($availableTokens as $tkAmt) {
        print "Token type: " . $tkAmt->getMerchantTokenId() . " ";
        print "Amount: " . $tkAmt->getAmount() . "\n";
    }
}
else {
    // Error while revoking credit from the autobill
    print $response['returnString'] . "\n";
}
```

If the amount of currency or token-based credit to be revoked results in a negative balance, CashBox sets the balance to 0. Disentitlement (loss of customer access to the product) will occur when CashBox attempts to process the *next* billing transaction for the AutoBill, and the transaction fails due to insufficient credit.

Fetching AutoBill Credit Transactions

CashBox stores token- and currency-based periodic transactions in the same way that it stores AutoBills paid for with currency payment methods. Fetch token-based transactions with API calls such as the Transaction object's fetchByAutoBill() method. These transactions contain a transaction item with sku attribute set to VIN_Credit, and price attribute set to a negative value equivalent to the amount of the transaction (signifying that CashBox deducted the credit).

Fetching an AutoBill's Credit History

CashBox maintains a log of credit-related events for each AutoBill object. This log tracks credit-related events such as credit granted, revoked, consumed, and earned due to a gift card redemption. Retrieve the audit log with the AutoBill object's fetchCreditHistory() method. (Because the method includes paging and a time range, you may limit the number of records returned.)

fetchCreditHistory() returns an array of CreditEventLog objects. Each CreditEventLog object holds the Credit object used for that specific event, a timestamp, the type of activity performed with the Credit object, and a text note field.

Use fetchCreditHistory to return an array of CreditEventLog objects:

```
$abill = new AutoBill();

// autobill id for an existing customer whose
// credit history you wish to retrieve

$abill->setMerchantAccountId('jdoe101');

$page = 0; // paging begins at 0
$pageSize = 5; // five records
$startTime = '2010-01-01T22:34:32.265Z';
$endTime = '2010-01-30T22:34:32.265Z';

do {
    $ret =
        $abill->fetchCreditHistory($startTime, $endTime $page, $pageSize);
    $count = 0;
    if ($ret['returnCode'] == 200) {
        $fetchedLogs = $ret['creditEventLogs'];
        $count = sizeof($fetchedLogs);
        foreach ($fetchedLogs as $log) {
             $credit = $log->getCredit();
             $ts = $log->getTimeStamp();
             $eventType = $log->getType();
             // process retrieved credit event log
             // details here.
        }
        $page++;
    }
} while ($count > 0);
```

12.2 Working with Gift Cards

You may also add credit to an `AutoBill` or `Account` by redeeming a gift card through CashBox. Gift Cards are processed much like currency, in that CashBox contacts a gift card processor company, which notifies CashBox if a gift card is redeemable, and handles the transaction. (CashBox currently supports gift cards issued by InComm.)

Gift Cards may be used to add Token Credits to an AutoBill, in that a Gift Card may be used to purchase a Product which grants Token Credits.

Note:	Products associated with Gift Cards are a special case, in that they may hold only Token Credits.

Store Products created for Gift Card redemption in CashBox in order to track the SKU provided by the gift card processor. Do not create AutoBills which contain these Products.

12.2.1 Understanding the Attributes of the `GiftCard` Object

The `GiftCard` object encapsulates gift card details such as the card's unique identification number (`pin`) and the processor that CashBox should contact for redemption of the gift card (this value defaults to `InComm` if left unspecified). When you make the API call to redeem a gift card, CashBox also assigns a unique `VID` to the corresponding `GiftCard` object, and stores the latest status of the gift card (such as whether it was redeemed or is still pending) in the `status` attribute of the `GiftCard` object. See Section 9: The GiftCard Object in the ***CashBox API Guide*** for details.

12.2.2 Determining Redemption Credit Amount

CashBox redeems Gift Cards for Token credit associated with a Product object.

When a customer offers a Gift Card PIN for redemption, CashBox contacts your gift card processor to redeem the card, and the processor returns a SKU or a UPC number if the card is valid. This SKU or UPC must match the `merchantProductId` of a `Product` object that you have previously created in CashBox. When you create the `Product`, use the `creditGranted` data member to associate a number of Token credits with it, which will be granted to the Account or AutoBill upon redemption of the associated Gift Card.

To redeem a Gift Card:

1. CashBox sends the Gift Card number (as defined by you and your gift card processor, and presented by your customer) to your gift card processor.
2. Your card processor validates that the Gift Card associated with the number is still active.
3. Upon validation, your Gift Card processor returns a SKU or UPC number to CashBox.
4. CashBox matches this SKU to a Product's `merchantProductId` (or SKU).
5. If a match is found, the Token credits associated with the Product are granted to the Account or AutoBill.

Before accepting gift cards, work with your gift card processor to define the SKU or UPC your processor will return for each type of gift card you accept, and create `Product` objects with the corresponding `merchantProductId` in CashBox.

The following example creates a new `Product` object that grants token-based credit (5000 tokens of type `'STARWARS_POINTS'`). The `merchantProductId` of this `Product` matches the SKU/UPC the processor returns when a $10 gift card is redeemed.

```
$product = new Product();

// Identify the product by your unique identifier
// This should be the SKU/UPC returned by gift card processor

$product->setMerchantProductId('49238434023383');

$product->setStatus('Active');
$product->setDescription('Redeem product for ten dollar gift');

// define the credit that this product will grant
$tok = new Token();

$tok->setMerchantTokenId('STARWARS_POINTS');

$tokAmt = new TokenAmount();
$tokAmt->setToken($tok);
$tokAmt->setAmount(5000);

$cr = new Credit();
$cr->setTokenAmounts(array($tokAmt));

// Set the credit into the product

$product->setCreditGranted($cr);

// Now make API call to create the product

$response = $product->update(DuplicateBehavior::SucceedIgnore);
if($response['return Code'] == 200 && $response['created']) {
    $createdProduct = $response['data']->product;
    print "Created product with VID " . $createdProduct->getVID();
}
```

12.2.3 Redeeming a Gift Card

Redeeming a gift card is a two-step process. First, check whether the gift card is redeemable by calling the `GiftCard` object's `statusInquiry()` method. If the call shows that the status of the gift card is `Active`, you may redeem the gift card.

Determine the status of a Gift Card:

```
$gc = new GiftCard();

// set the PIN provided by the customer
$gc->setPin('683092298403');
$gc->setPaymentProvider('InComm');

// Now make API call to inquire about the status of the gift card

$response = $gc->statusInquiry();
if($response['return Code'] == 200) {
    // The API call is successful. Now check the
    // status in the updated GiftCard object returned by
    // this call

    $updatedGc = $response['data']->giftcard;
    $status = $updatedGc->getStatus();

    // the status thus obtained is an object of type GiftCardStatus
    // Now check if it says the gift card is redeemable

    if ($status->getStatus() == 'Active') {
        // The gift card is redeemable, retrieve its VID
        // so we can reference it just by VID when we redeem it

        $gcVID = $updatedGc->getVID();
    }
    else {

        // Gift card is not redeemable. Inform the customer here
        // You may want to include the response received from the
        // gift card processor

        $responseCode = $status->getProviderResponseCode();
        $responseMsg = $status->getProviderResponseMessage();
    }
}
```

After determining the status of the gift card, you may redeem the card. Both `AutoBill` and `Account` objects support `redeemGiftCard` calls. When `redeemGiftCard` is successful, the credit granted by the `Product` (with `merchantProductId` returned by the gift card processor in response to this call) is added to the `Account` or `AutoBill` object against which the call was made.

Redeem a Gift Card, and add credit to an AutoBill:

```
$abill = new AutoBill();

// autobill id for a customer's existing subscription
// the customer wants to redeem a gift card and add credit
// to this autobill

$abill->setMerchantAutoBillId('SBCR312345');

$gc = new GiftCard();

// set the VID of the gift card, obtained when the status
// status of the gift card was checked, and determined to be active

$gc->setVID($gcVID);

// Now make the SOAP API call to redeem the gift card

$response = $abill->redeemGiftCard($gc);

if ($response['returnCode'] == 200) {

   // Redemption successful. Check if credit was added
   // to the autobill

   $updatedABill = $response['data']->autobill;

   $availableCredits = $updatedABill->getCredit();
   $availableTokens = $availableCredits->getTokenAmounts();

   print "Available token credits: \n";
   foreach($availableTokens as $tkAmt) {
      print "Token type: " . $tkAmt->getMerchantTokenId() . " ";
      print "Amount: " . $tkAmt->getAmount() . "\n";
   }

   // Also make sure status of the gift card is 'Redeemed'

   $updatedGc = $response['data']->giftcard;

   print "Status of the gift card: ";
   print $updatedGc->getStatus()->getStatus() . "\n";
}
else {
   // Error while granting credit to the account
   print $response['returnString'] . "\n";
}
```

Note: CashBox allows only full redemption of a gift card. You may not partially redeem a gift card.

12.2.4 Reversing a Gift Card Redemption

A gift card must be `Active` before a successful `redeemGiftCard()` call can be made. It is possible that during your redemption attempt, due to a technical problem such as a connection drop out, the operation does not complete and the underlying `Account` or `AutoBill` does not receive any credit. If this happens, make the `redeemGiftCard()` call again.

To reverse a gift card redemption, first ensure that the status of the gift card is `Active`, in case a previous operation has (erroneously) changed the status. If the gift card is no longer `Active`, you may reverse the last operation upon it using the `GiftCard.reverse()` method.

Note:	The purpose of this call is to correct an unwanted situation, as described above. Do not use this call to reverse a successful redemption call. It does not automatically revoke credits granted in the previous redemption call.

Reverse a Gift Card redemption:

```
// set the VID of the gift card. We obtained this when we
// inquired status of the gift card

$gc->setVID($gcVID);

// Now make the SOAP API call to reverse the redemption

$response = $gc->reverse();

if ($response['returnCode'] == 200) {

    // Reversal successful.
    // fetch the autobill against which we originally
    // redeemed the gift card here.

    // Also make sure status of the gift card is 'Active'

    $updatedGc = $response['data']->giftcard;

    if($updatedGc->getStatus()->getStatus() == 'Active') {

        // try redemption again here
    }
}
else {
    // Error while reversing the card
}
```

13 Hosted Order Automation

CashBox Hosted Order Automation allows you to accept and process customer payment method information, without exposing your own servers, by submitting it directly from your order form to Vindicia over Secure Sockets Layer (SSL). This gives you a means by which you may create a storefront which accepts and stores credit card information, without ever loading it into your own database. Because HOA bypasses your server at form submission, thus limiting your system's exposure to client Credit Card information, your need for PCI compliance is alleviated. Use Vindicia's Hosted Order Automation (HOA) to create CashBox objects that contain sensitive payment information, such as credit-card account numbers. Use a specially designed Web order form, accessed from your server, to store customer credit card information directly on Vindicia's servers from your submission page. With HOA, Vindicia makes the SOAP calls, and populates the client data in the CashBox database.

HOA works by creating a `WebSession` object, with a WebSession ID, which is used to track the interaction between your web pages and the CashBox API, and to temporarily store sensitive customer information while the session is in progress. Create the WebSession ID when your customer requests a data-sensitive page from your server, by calling `WebSession.initialize` on the Vindicia servers. From the moment the Initialize call is made, the WebSession ID is used to both track and validate the session.

The `WebSession` object is used to temporarily store payment method information in the CashBox database, which is then permanently loaded when the `WebSession.finalize` method is called.

For more information on the WebSession object, see Section 19: The WebSession Object in the ***CashBox API Guide***.

Use HOA as an adjunct to the SOAP-based CashBox API to integrate your applications with CashBox. HOA allows you to generate certain CashBox objects directly, without making SOAP calls.

Note: HOA's primary function is to mitigate your need for PCI compliance when accepting payment information from your customers. For that reason, the WebSession object is limited in scope and properties, and is meant only to create or authorize `AutoBills`, `PaymentMethods`, and `Transactions`. HOA is not designed to create other CashBox objects.

Caution	Even though you need not permanently store payment data, heed the related Payment Card Industry (PCI) regulations. While creating CashBox API objects and transmitting data to Vindicia, your customers' payment data is stored for a short period of time in RAM on your server, where it could potentially be swapped to the hard drive. This temporary storage might mean that you must comply with PCI requirements. If you desire no involvement with such compliance, take advantage of CashBox's Hosted Order Automation (HOA) feature, which eliminates the need for you to store sensitive payment data, even transiently. Using HOA, payment data is never collected on your server.

13.1 HOA Features

HOA guarantees security because:

- Vindicia receives data submitted by the order form over Secure Sockets Layer (SSL).
- HOA tracks an order form submission through a `WebSession` object, created on the Vindicia server before the order form is presented to your customer. This allows you to pre-load information, including the IDs of CashBox objects to which the order refers (such as an existing `BillingPlan` or `Product` object), ***without*** displaying the information on the form.
- A `WebSession` object has a 40-character unique ID, which is included as the session ID in the form. This ID expires 1) after a form is submitted using that ID, 2) after 1 hour (3600 seconds, the default expiration time), or 3) after the merchant-configured expiration time, whichever occurs first. This prevents illegal or repeated use of session IDs by hackers. HOA stores the results of the API call in the CashBox `WebSession` object. Fetch this object and, based on the results, determine the next steps the customer should take on your site.
- While HOA allows you to create CashBox objects that contain sensitive payment information, you may also manipulate and retrieve those objects, and take advantage of other CashBox tools, using the CashBox API, or the CashBox Portal.

Vindicia does not host any HTML pages in the process. Your customers may notice **Loading https://www.vindicia.com …** , in the status bar of their browser, but only momentarily, after they have submitted the Web order form, and before HOA loads your redirection page. Your order form and success or failure pages, with your branding style, are the only pages visible to your customers.

HOA does not make the API call to create objects containing sensitive payment information, such as an `AutoBill`, or a `PaymentMethod` object, until you "finalize" the `WebSession` object from the success page that you host. Therefore, if the connection is severed as HOA redirects the customer's browser from Vindicia's server to your server, your success page will never be reached, the `WebSession` will not finalize, and HOA will not make the API call. This approach preserves data integrity in cases of connections dropped during the process.

Note:	Vindicia never returns or displays sensitive payment information, such as credit-card account numbers, in full. When returning information through an API call, or displaying the data in the Portal, CashBox always partially masks account numbers.

13.2 HOA Process Flow

The HOA process differs from standard CashBox process in that, using HOA, you do not pass sensitive Customer information directly to the CashBox servers. Instead, use HOA to handle the data transfer.

13.2.1 HOA Work Flow Overview

This section describes the page flow for the HOA process.

1. Customer navigates to your Offer page.

   Use this page to request non-sensitive information, such as name, address, and email address, with which the customer's Account may be created.

   When the customer clicks **Submit** or **Save**:

   a. Use `Account.update` to create a new `Account` object, and load the acquired customer information.

   b. Then, call `WebSession.initialize` to create a `WebSession` object, with which this session will be tracked and managed

   c. Redirect the customer to your Payment Information page.

2. Customer arrives at your Payment Information page.

   Use this page, and HOA, to collect sensitive payment information, such as the credit card number, its expiration date, and the CVN code on the reverse.

   When the customer clicks **Submit**:

   a. Data is temporarily stored within the CashBox database, awaiting the session to terminate by timeout or by finalizing the HOA calls (below).

   b. Redirect the customer to your Confirmation page.

3. Customer arrives at your Confirmation page.

   a. The confirmation page is loaded with the WebSession ID (in a hidden form), to confirm the validity of the transaction

   When the customer clicks **Submit**:

4. Call `WebSession.finalize` to confirm the submission, finalize the session, and instruct CashBox HOA to make the SOAP calls necessary to load the temporary data from the `WebSession` object into the appropriate CashBox objects.

13.2.2 HOA Server Work Flow

This section describes the data flow between your website's servers, the CashBox WebSession servers, and the CashBox database.

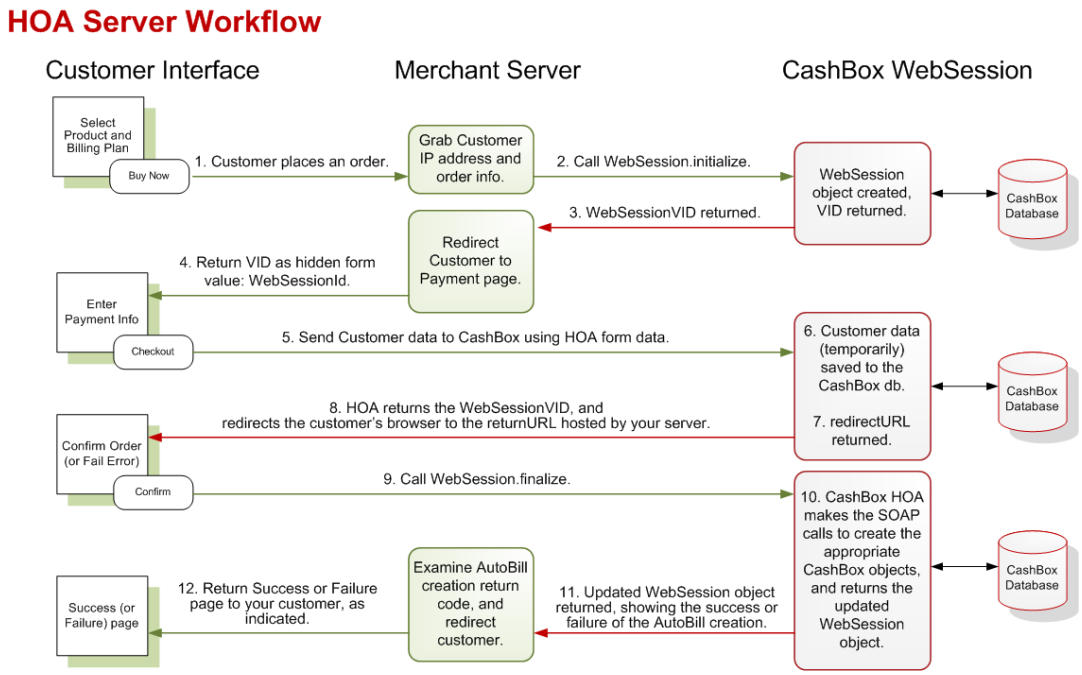


Figure 13-1 HOA Server Work Flow

1. Your customer requests a page from your Web application (makes an http call to your website), such as your Order page, which requires that they submit payment information.

2. Call `WebSession.initialize()` in the CashBox API to initialize a `WebSession` object on Vindicia's server. In the `WebSession` object, specify two key attributes:
   - The IP address from which the customer requested the order form. For example:

     ```
     billing_CODE_ONLY.php:$ws->setIpAddress($ip_address);
     ```

   - The API call HOA should make when the customer submits the form. For example, calling `PaymentMethod.update()` allows you to create a new payment method. For example:

     ```
     <input type="hidden" name="vin_WebSession_method"
     value="PaymentMethod_Update" />
     ```

The initialize call creates a `WebSession` object, which may include a `method` data member. This data member may be used to specify the SOAP call for which this WebSession is acting as proxy. Valid input includes:

- `Account_Update`
- `Account_UpdatePaymentMethod`
- `AutoBill_Update`
- `PaymentMethod_Update`
- `PaymentMethod_Validate`
- `Transaction_Auth`
- `Transaction_AuthCapture`

(The `initialize` call takes several parameters to mitigate the risk of an un-authenticated call.)

Note: Capture your customers' IP addresses, to better assess fraud risk, and to allow Vindicia's ChargeGuard team to pursue chargeback investigations.

---

3. After the `initialize` call is made, the `WebSession` object returns a `VID`.

   This `VID` is used to track the HOA session, and serves as its means of control and authentication. Only one `VID` is generated per session, and it has a time limit for validity.

CashBox returns the `WebSession` `VID` to your application in response to the `initialize` call. Other form elements, hidden or to be provided by the customer, must be consistent with the data required to complete a `PaymentMethod.update()` call (or other API call, listed above). The form's action URL points to the HOA address on the Vindicia server.

4. Return the `VID` to the customer's webpage as a hidden form value: `WebSessionId`. (Be certain to retain the `VID` for later use.)

Note: Steps 1 through 4 are parts of a single synchronous call and response chain initiated by the customer's request for the order form.

---

5. Use the `VID` to post the customer's payment information directly to the Vindicia servers. The customer fills out your form, and clicks **Save** or **Submit**. Because the form's action URL points to the HOA address on the Vindicia server, HOA receives the content, including payment information, directly, bypassing your server altogether.

Note: The customer must complete the form within one hour (the default expiration time), or within your specified expiration time, after which CashBox marks the `WebSession` object expired.

---

6. Vindicia saves the form data to CashBox.

HOA loads the `WebSession` object with `vin_WebSession_VID`, which accompanied the form, and stores the data submitted by the order form. Note that HOA does not make the API call to create the `PaymentMethod` object at this point in the process.

7. CashBox returns the `redirectURL` (specified in the merchant's submitted page / form value), and the merchant redirects the customer to their confirmation page.

8. HOA redirects the customer's browser, which is awaiting a response to the form submission, to the success page (`returnURL`) hosted by your server. Form data is passed to the CashBox servers; but the customer experience is that they never leave your website. HOA includes the `WebSession` VID in the redirection URL, so that it is available to your success page.
   a. The redirect carries the `VID` for the `WebSession` back to the merchant site, thus confirming the session, and allowing the merchant to access information contained in the `WebSession` object.
   b. The `VID` acts as both transport and handshake, in that it carries through the `WebSession`.

9. Merchant makes a `WebSession.finalize` SOAP call to Vindicia.
   a. `WebSession.finalize` returns both the WebSession results, and the result for the underlying API calls. (For more information, see Section 19: The WebSession Object in the ***CashBox API Guide***.)

The success page makes a `WebSession.finalize()` call, instructing HOA to make the API call needed to create the CashBox objects described.

10. CashBox finalizes the session, saves the customer data to the database (previous save was only temporary), and allows the CashBox process to begin (updating `AutoBill`, `PaymentMethod`, `Product`, and `Account` data.)

Using data in the `WebSession` object, and data stored after the form submission, HOA internally makes the API call (in this case, the `PaymentMethod.update()` call) to create a CashBox object (in this case, the `PaymentMethod` object) and updates the `WebSession` object with the results of the API call. HOA returns this updated `WebSession` object to your success page.

11. The `WebSession` object returned by the `finalize()` call contains the results of the `PaymentMethod.update()` call. Using these results, the success page determines its dynamic content to be sent to the customer's web browser.

12. Your application returns the success page to the customer's browser.

(Note that steps 5 through 11 are parts of a single synchronous call and response chain initiated by the customer's submission of the order form.)

13.3 Working with HOA

13.3.1 CashBox objects affected by HOA

Using the CashBox API, you may access objects generated during an HOA process using their `WebSessionVid`, and the `fetchByWebSessionVid` method. The four CashBox objects which may be fetched using their `WebSessionVid` are:

- `Account`
- `AutoBill`
- `PaymentMethod`
- `Transaction`

13.3.2 HOA Naming Schema

Web forms are a flat name space. To accommodate this, CashBox flattens its standard SOAP calls by concatenating them using underscores. HOA uses the following naming rules:

- Every name begins with `vin_`
- Followed by the object type you with to create: `vin_Transaction_`
- Followed by the name of the data member you wish to set: `vin_Transaction_amount`
- Followed by the value for the data member: `vin_Transaction_amount => 100`

> **Note:** HOA naming rules are case-sensitive.

Follow the same pattern to set the value for an object:

```
vin_PaymentMethod_billingAddress_addr1 => "1639 Harrison Ave."
```

In creating forms, use the format:

```
<input type="text" name="vin_Transaction_transactionItems_0_sku"/>'
```

For example:

```
vin_PaymentMethod_billingAddress_addr1 => "1639 Harrison Ave."
```

In the form, becomes:

```
<input type="text" name=" vin_PaymentMethod_billingAddress_addr1"/>'
```

Note:	Calling `WebSession.finalize` instructs HOA to take ALL form information gathered through the HOA http POST, and create and populate the appropriate CashBox objects.

Naming schema for parameter values

Setting parameter values follows the same structure. For example, to set ***minChargebackProbability*** to 100:

```
vin_Transaction_auth_minChargebackProbability => 100
```

Naming scheme for an object with an array

Comma delimited values are used for private form values, therefore you cannot use them elsewhere, and you cannot use them to define an array.

To set an array of values, use the pattern described above, but add the index value for the item after the name of the attribute, then the item itself.

```
"Transaction_taxExemptions_0_ taxExemption"
"Transaction_taxExemptions _1_ taxExemption2"
```

Naming scheme for name-value arrays

To set a name-value array, simply use the flattened object name, method name, and parameter name, concatenated with an underscore:

```
Transaction_nameValues_name => value
    $mpnv1->setName('AutoBill_Update_minChargebackProbability');
    $mpnv1->setValue('95');
```

13.3.3 HOA Form Post Parameters

To create a Form Post parameter, simply create an HTML form that contains elements, forms, and widgets whose names follow the examples shown here. The values associated with these names will be dependent upon your customer's actions or selections in the HTML form.

The following lists examples of HOA Form Post parameters:

```
vin_WebSession_version='3.9'
vin_WebSession_method='Account_updatePaymentMethod'
vin_WebSession_vid
vin_PaymentMethod_type='CreditCard'
vin_PaymentMethod_accountHolderName
vin_PaymentMethod_creditCard_account
// Upon form submission, HOA will validate the value entered in
// this field by running a Luhn check. If the check fails,
// the customer's browser will be redirected to your error URL.

vin_PaymentMethod_creditCard_expirationDate='YYYYMM'
// Upon form submission, HOA will validate the value entered
// in this field by making certain the year begins with "20," and
// the month is between 01 and 12. If this check fails,
// your customer's browser will be redirected to your error URL.
//(full expirationDate takes precedence)

vin_PaymentMethod_creditCard_expirationDate_Month='MM'
// Upon form submission, HOA will validate the value entered
// in this field by making certain that the month is between
// 01 and 12. If this check fails, your customer's browser will
// be redirected to your error URL.

vin_PaymentMethod_creditCard_expirationDate_Year='YYYY'
// Upon form submission, HOA will validate the value entered
// in this field by making certain that the year begins with "20."
//If this check fails, your customer's browser will
// be redirected to your error URL.

vin_PaymentMethod_nameValues_cvn
vin_PaymentMethod_billingAddress_name
vin_PaymentMethod_billingAddress_addr1
vin_PaymentMethod_billingAddress_addr2
vin_PaymentMethod_billingAddress_city
vin_PaymentMethod_billingAddress_district
vin_PaymentMethod_billingAddress_postalCode
vin_PaymentMethod_billingAddress_country
vin_PaymentMethod_billingAddress_phone
```

If the customer is redirected to your error URL by one of these error checks, be certain to inform them why form submission failed. Then, initiate a new WebSession, and re-present the order form to your customer.

Private Form Values

To set the `merchantTransactionId` of a `Transaction`, pass a FORM name-value pair to a `WebSession` `NameValuePair` object. For example:

```
{name => 'vin_Transaction_merchantTransactionId', value => 'tx_' . $time}
```

(Notice that the `Transaction` object's `merchantTransactionId` data member is flattened out and prefixed with `vin_`.)

The following lists examples of HOA Private Form values:

```
vin_Account_merchantAccountId
vin_PaymentMethod_merchantPaymentMethodId
```

13.3.4 HOA Method Parameters

To pass a variable to a CashBox method, use a name-value pair.

For example, to pass the ***minChargebackProbability*** attribute to the `Transaction.auth` method:

```
{name => 'Transaction_Auth_minChargebackProbability', value => 1},
```

The following lists examples of HOA Method parameters:

```
Account_updatePaymentMethod_replaceOnAllAutoBills=true
Account_updatePaymentMethod_updateBehavior=Validate
Account_updatePaymentMethod_ignoreAvsPolicy=false
Account_updatePaymentMethod_ignoreCvnPolicy=false
```

13.3.5 HOA Error Checking

It is possible for the `WebSession.finalize` call to be successful while the underlying SOAP API call was not.

Therefore, your code should check that both the `WebSession.finalize` call **and** the API return within it returned a 200 for success.

As different API calls will use the same return codes to mean different things, please reference API name, return code, and return strings to get a complete error view and reference lists for each method in the ***CashBox API Guide***.

13.4 WebSession Object

The `WebSession` object must be created before serving the order form to the customer for submission to the Vindicia server.

The `WebSession` object:

- Tracks the submission of an order form containing sensitive payment data. Use the `WebSession`'s `VID` as a session ID, to track the session from your customer's first order form request, to your final success or failure page for the session.
- Establishes a time limit for the entire sequence.
- Ensures that a single form submission from a customer results in only one API call, specified when the session is initiated (see the `WebSession` object's `method` data member). That way, if the customer submits the same form repeatedly, HOA does not make multiple API calls, resulting in the creation of multiple CashBox objects. The `WebSession` object supports the following CashBox API calls (methods):

  ```
  Account_Update
  Account_UpdatePaymentMethod
  AutoBill_Update
  PaymentMethod_Update
  PaymentMethod_Validate
  Transaction_Auth
  Transaction_AuthCapture
  ```

- Stores the result of the CashBox API calls made by HOA after finalization of the `WebSession` object. Your success / failure page then examines this information to determine its content, and directs the customer to the appropriate next steps. (See the `WebSession` object's `apiReturn` attribute in Section 19.1: WebSession Data Members in the ***CashBox API Guide***.)
- Holds some of the data required to complete the CashBox API call that HOA makes after the customer has submitted the form. (See the `WebSession` object's `privateFormValues` attribute in Section 19.1: WebSession Data Members in the ***CashBox API Guide***.)

For details on the `WebSession` object's attributes, see Section 19.1: WebSession Data Members in the ***CashBox API Guide***.

The following example shows a typical initialization of a `WebSession` object. This example creates an `AutoBill` object for a new subscription, without passing sensitive Payment Method information through your servers. When a customer requests a page to start a subscription, initiate a `WebSession` object.

To populate the data in the `WebSession` object:

- HOA calls `AutoBill.update` when you finalize the `WebSession` object.
- `AutoBill.update()` may use an existing `Account` object for the customer.

  Note: Do not allow this object's VID to appear in the form sent to the customer's browser, not even as a hidden element.

- `AutoBill.update()` uses an existing `Product` object.

Note: Do not allow this object's VID to appear in the form sent to the customer's browser. Hiding this VID prevents hackers from specifying a random Product ID, that may correspond to a Product ID that you do not wish to make available for subscriptions in this context.

- `AutoBill.update()` uses one of two existing `BillingPlan` objects. (Specify existing Billing Plan IDs to prevent other, possibly inactive or invalid, IDs from being submitted. This helps to regulate and control the AutoBill creation process.)

Because all POSTed data is submitted through an HTTP Post, it must be submitted as name-value pairs.

(Note: These are HTTP name-value pairs; not CashBox `NameValuePair` objects.)

```
$ws = new WebSession();

// HOA must use CashBox API version 3.4 or later to make the
// AutoBill.update call
$ws->setVersion('3.4');

// HOA should make an AutoBill.update call when the form
// is submitted

$ws->setMethod('AutoBill_Update');

// Capture the customer's IP address. When the customer submits the form,
// it should come from the same IP address

$ws->setIpAddress("124.23.210.175");

// Page to which HOA will redirect customer's browser after
// successfully storing the data received when the customer
// submits the form

$ws->setReturnURL("https://merchant.com/subscribe/success.php");

// Page to which HOA will redirect customer's browser if HOA fails to
// store the data received when the customer submits the form

$ws->setErrorURL("https://merchant.com/subscribe/failed.php");

// Private name-value pairs. These are needed to create the AutoBill
// object, but are NOT included in the customer form

$pnv1 = new NameValuePair();

// The name is flattened object name, concatenated
// with attribute names with an underscore.

// The CashBox Account object for which HOA should create the
// AutoBill object

$pnv1->setName('vin_Account_merchantAccountId');
$pnv1->setValue('df943');
```

```
// The CashBox Product object HOA should use to construct
// the AutoBill object

$pnv2 = new NameValuePair();
$pnv2->setName('vin_Product_merchantProductId');
$pnv2->setValue('BlorgWars II');

$pnv3 = new NameValuePair();
$pnv3->setName('vin_BillingPlan_merchantBillingPlanId');

// When customer submits the form, the Billing Plan
// must be one of these two comma separated values

$pnv3->setValue('GoldAccess2010, PlatinumAccess2010');

$ws->setPrivateFormValues(array($pnv1, $pnv2, $pnv3));

// Create method parameter name-value pairs. These are needed to make the
// AutoBill.update call which takes parameters in addition to the
// AutoBill object itself. Do not allow these to come from the form
// submission, because that makes them susceptible to hacking

$mpnv1 = new NameValuePair();

// The name is flattened object name, method name, and
// parameter name, concatenated with underscores.

$mpnv1->setName('AutoBill_Update_minChargebackProbability');
$mpnv1->setValue('80');

// Leave other parameter values to their default values

$ws->setMethodParamValues (array($mpnv1));

// Now create the WebSession object on Vindicia servers
// by making the SOAP call to initialize the object

$response = $ws->initialize();

if ($response['returnCode'] == 200) {

    $ret_ws = $response['data']->session;

    // The VID of the WebSession object serves as session id

    $sessionId = $ret_ws->getVID();

    // Embed the sessionId as a hidden field named
    // vin_WebSession_VID in the order web form

    // Compose and present the order web form to the customer here
}
else {
    // Return error to the customer who requested the web order form
}
```

Note that this example stores the merchant IDs of the `Account`, `BillingPlan`, and other objects as private values in the `WebSession` object, and assumes that objects with these IDs already exist in CashBox. Always specify merchant IDs of objects as private values stored in the `WebSession` object, even if the objects do not yet exist. These are your internal object IDs; for security reasons do not pass them in through the form submission, or store them as hidden fields in the form. If an object ID is present in the `WebSession` object's private values, when HOA completes the desired API call, HOA first looks for an object with that ID in the CashBox database. If no such object is available, HOA creates a new object with that ID, and populates it with related data, either submitted through the form, or previously stored in the `WebSession` object.

For example, if an `Account` with `merchantAccountId: df943` does not yet exist in CashBox, when the `WebSession` is finalized, HOA will create a new `Account` object with `merchantAccountId: df943`. In this case, the `Account` attributes, such as customer's name, email, and shipping address, must be passed in through the form; which means that your form must include the fields necessary for your customer to submit this information.

The `WebSession` object created remains valid for one hour (by default), within which time the customer must complete and submit the form.

13.4.1 Integrating HOA with CashBox

To integrate with CashBox using HOA:

1. Create and host a Web order form on your website (called an **order form** or just **form** in the rest of this section). The action URL to process the form data is an address on Vindicia's server.

When your customer enters payment data and clicks **Submit** on your order form, their browser passes the data to an HOA component hosted on Vindicia's server.

2. Create and retrieve a `WebSession` object on the Vindicia server with the CashBox API. HOA tracks the order form's submission activity through a `WebSession` object instance.

When your customer requests the order form from your site, create a `WebSession` object on the Vindicia server. Upon receiving the data submitted by the form, HOA stores the data on Vindicia's secure server, and redirects the customer's browser to a success page hosted by you on your server.

3. Create and host a success and failure page to which HOA will redirect your customer's browser after storing the data from the submitted form.

After storing the data, HOA immediately redirects the customer's browser to the dynamically generated success or failure page on your site. On this page, make a SOAP API call to finalize the `WebSession`. In response to the call, HOA makes a CashBox API call internally on Vindicia's server, using the data in your `WebSession` object, and the data stored from the order form submission. For example, depending on the form's content and the corresponding `WebSession` object instance, HOA might call `update()` on a `PaymentMethod` object it constructs with the form data and other data in the `WebSession` object. `WebSession` finalization returns the results of the API call HOA made; you can then process the results on that page and display an appropriate message to your customer.

13.5 Creating Order Forms for HOA

To provide payment information and other data required by the `WebSession` object, your customer order form must contain the following attributes:

- The VID of the corresponding `WebSession` object as a hidden element named `vin_WebSession_VID` in the form, for example:

```
<input type="hidden" name="vin_WebSession_VID" value="$sessionId" />
```

- The action URL of the form pointing to the HOA page on the Vindicia server (**https://secure.vindicia.com**). The method for submitting the form is POST.

> **Note:** URLs vary by working environment. Please contact Vindicia client services for your current location.
>
> **Prodtest:** https://secure.prodtest.sj.vindicia.com/vws
> **Staging:** https://secure.staging.sj.vindicia.com/vws
> **Production:** https://secure.vindicia.com/vws

- The attributes required for the CashBox objects HOA will create with finalization of the `WebSession`. These attributes must be present as form elements, unless stored as private values in the `WebSession` object. The element name must begin with the prefix `vin_`, followed by a flattened attribute name that contains the object name, sub-object name, and attribute name, concatenated by underscores.

In the following example, the form collects data for a `BillingPlan` object and a `PaymentMethod` object to construct an `AutoBill` object. The corresponding `WebSession` object specifies `AutoBill_Update` in its method attribute.

```
Select Billing Plan: <p>
<input type="radio" name="vin_BillingPlan_merchantBillingPlanId"
   value="GoldAccess2010" />
<input type="radio" name="vin_BillingPlan_merchantBillingPlanId"
   value="PlatinumAccess2010" />

<input type="hidden" name="vin_AutoBill_currency" value="USD" />
<input type="hidden" name="vin_PaymentMethod_Type"
value="CreditCard" />

Enter credit card details: <p>
Account Holder Name: <input type="text"
name="vin_PaymentMethod_accountHolderName" /> <br>
Credit card number: <input type="text"
name="vin_PaymentMethod_creditCard_account" /> <br>
Expiration Date: <input type="text"
   name="vin_PaymentMethod_creditCard_expirationDate" /> <br>
CVV Number: <input type="text" name="vin_PaymentMethod_nameValues_cvn"
   " size="4" maxlength="4" /> <br>
```

Note	You need not embed all the data in the form; feel free to preload some data in the corresponding `WebSession` object. For example, the above form, which results in the creation of an `AutoBill` object, does not include the customer account information, because the customer's `Account` object's `merchantAccountId` attribute is already in the `WebSession` object's `privateFormValues` attribute.

If an attribute is an array, concatenate the element with a number that specifies the array index.

For example, to specify the line items to construct the `Transaction` object's `items` attribute (for a `WebSession` object with the `Transaction.auth` method):

```
<input type="text" name="vin_Transaction_transactionItems_0_sku"/>'
<input type="text" name="vin_Transaction_transactionItems_0_name"/>
<input type="hidden" name="vin_Transaction_transactionItems_0_price"
   value="9.95"/>
<input type="text" name="vin_Transaction_transactionItems_0_quantity"/>

<input type="text" name="vin_Transaction_transactionItems_2_sku"/>
<input type="text" name="vin_Transaction_transactionItems_2_name"/>
<input type="hidden" name="vin_Transaction_transactionItems_2_price"
   value="8.88" />
<input type="text" name="vin_Transaction_transactionItems_2_quantity"/>
```

If you include elements in the form that are relevant to you, but not required by a Vindicia object (such as form elements whose names do not contain the `vin_` prefix), HOA stores the corresponding HTTP POST data in the `postValues` attribute of the corresponding `WebSession` object. After posting, the data is available to you when you retrieve the `WebSession` object on your success or failure page.

When your customer submits the form, HOA stores the form data with the `WebSession` object. It does not create any CashBox objects (by making the API call you specified in the `method` attribute of the `WebSession` object) until you call `finalize()` on the `WebSession` object. Finalize the `WebSession` from the success page to which HOA redirects the customer's browser after storing the form data.

13.6 Creating Success or Failure Pages for HOA

When a customer submits an order form, HOA receives and stores the data on Vindicia servers, and redirects the customer's browser to your success page, hosted on your server. Specify the URL of your success page in the `returnURL` attribute of the `WebSession` object. (This is generally a dynamically generated page that intersperses CashBox API calls with HTML to be sent to the browser.) If a failure occurs when HOA stores the form data, HOA redirects the customer's browser to the page specified in the `WebSession` object's `errorURL` attribute. (That page is also a dynamically generated "failure page," hosted on your server to notify your customer about the failure and its causes.) If you do not specify the `errorURL` attribute, HOA will use the `returnURL` for redirection upon failure.

To work with the CashBox API and your success and failure pages:

- When HOA redirects to the success and failure pages, it passes the `WebSession` object's `VID`. Use this `WebSessionVid` to make a `finalize()` call on the `WebSession` object. Upon finalization, HOA internally makes the call specified in the `method` data member of the corresponding `WebSession` object, creates the desired object or objects on the Vindicia server, and updates the `WebSession` object with the call's results. This updated `WebSession` object is available to you in the response to your `finalize()` call.
- When creating a success page, extract the non-Vindicia form data submitted by the customer by examining the `WebSession` object's `postValues` data member.
- On the success page, fetch the object created by the API call made by HOA. For example, if your `WebSession` object's `method` data member is set to `AutoBill.update`, after finalization of the `WebSession`, HOA creates an `AutoBill` object. Fetch that object with the `WebSession` object's `VID` by calling `AutoBill.fetchByWebSessionVid()`. Similar methods are available for the `Account`, `Transaction`, and `PaymentMethod` objects. You may then include information from these objects on your success page.

The following illustrates some of the activities performed on a success page:

```
$sessionId = …; //passed in by redirected page

$ws = new WebSession($soapLogin, $soapPwd);
$ws->setVID($sessionId);

// finalize the WebSession so HOA can make the API call to
// create CashBox object/s containing sensitive payment
// information

$response = $ws->finalize();

if ($response['returnCode'] == 200) {

    $updatedWs = $response['data']->session;

    // Check if the API call HOA made to create the
    // CashBox object containing payment
    // information was successful

    if ($updatedWs->apiReturn->getReturnCode == 200) {
```

```
            // Extract non-Vindicia values submitted by the web
            // order form, and process them to prepare the HTML to
            // be returned to the customer

            $postVals = $updatedWs->getPostValues()

            // Assuming HOA created an AutoBill object, let's fetch it
            $soapAbill = new AutoBill($soapLogin, $soapPwd);

            $resp = $soapAbill->fetchByWebSessionVid($sessionId);

            if ($resp['returnCode'] == 200) {
                 $createdAutoBill = $resp['data']->autobill;

            // Get AutoBill contents here to be included in
            // HTML returned to the customer.
            }
        }
        else {
            // The API call HOA made to create or manipulate object
            // containing sensitive payment data did not succeed.
            // Return error message to customer

            $errorString = $updatedWs->getApiReturn()->getReturnString();

            …

        }
    }
    else {

        // Finalization failed
        // Return error message to the customer
    }
```

14 Common ChargeGuard Programming Tasks

Data must be integrated between Vindicia's ChargeGuard and your information system, as follows:

- **Integration of your data into ChargeGuard**. This is the collection and integration of your *transaction* data into ChargeGuard, which enables Vindicia to dispute chargebacks on your behalf.
- **Integration of chargeback data into your system**. This is the collection and integration of the relevant *chargeback* information back into your system, after which you can update and turn off accounts, as appropriate.

This chapter describes the related processes.

14.1 Integrating Data into ChargeGuard

To analyze and processes chargeback rebuttals or requests through ChargeGuard, Vindicia requires three types of data:

- **Chargeback data**. Vindicia's technical and operations groups can handle this integration directly with your payment processor and receive the information from the processor. To pass the data to Vindicia yourself, see Section 14.3: Data Reporting to Vindicia.
- **Transaction data.** Examples include the transaction ID and date, stock-keeping unit (SKU) ID, price, quantity, shipping and billing addresses, utility, and credit-card information.
- **Activity data**. Examples include user ID and user activity, such as logins to your site, pages visited, phone or email contacts, and fulfillment information.

Transaction and activity data is usually stored in your system. Extract it and map it to the format accepted by ChargeGuard using the CashBox API, then process the data, and automatically send to Vindicia at regular intervals.

Note	Be certain to send Vindicia your transaction and activity data regularly, to guarantee that the ChargeGuard team has the most recent information for chargeback disputes. Most merchants upload batches daily or weekly.

14.2 Integration of Chargeback Data Back into Your System

Chargeback dispute resolution usually takes a minimum of 90 days. When a chargeback occurs, most merchants:

1. Immediately limit any other potential risks associated with the chargeback. Turn off the current, future, and associated account information along with the related credit-card information, email address, or customer ID.
2. Update transaction and activity systems to reflect the latest chargeback status while Vindicia disputes the chargeback.

You have three options by which to accomplish those tasks.

- Receive updates from your payment processor, and manually alter account status in your system.
- Use the CashBox Chargeback Spreadsheet to determine a course of action, then manually update customer Account status.
- Use the CashBox API to automatically manipulate your customer `Account` objects, based on the CashBox Chargeback Spreadsheet.

14.2.1 Use Payment Processor Data to Manually Alter Account Status

With this option, you receive updates directly from your payment processors, and turn off accounts in your transaction systems, if appropriate.

14.2.2 Use CashBox Data to Manually Alter Account Status

With this option, manually turn off accounts in your transaction systems based on the data in the CashBox Chargeback Spreadsheet, which is a daily extract that shows all changes to your chargebacks.

To download the spreadsheet:

1. Log into the CashBox Portal at www.vindicia.com.
2. Select **Manage > Chargebacks > Spreadsheet Download**.
3. Specify the date range and download format.
4. Click **Download File**.

This file contains the latest status of the chargebacks in the CashBox system. Use this information to determine the action to take on customer accounts in question, such as closing them, placing a hold on them, or updating the status of the transactions and activities to which the chargebacks pertain.

14.2.3 Use the CashBox API to Automatically Update Account Status.

Use the CashBox API to automatically extract and map the Chargeback Spreadsheet data into your CashBox system.

1. Automatically download the CashBox Chargeback Spreadsheet by creating a script that simulates an HTTP POST operation to pull the spreadsheet from ChargeGuard with the CashBox API.

   Vindicia recommends *cURL* for this process.

2. Extract and map the data in the spreadsheet using a CashBox API integration.

3. Build logic at your end to add a chargeback to the customer ID for which Vindicia received new chargebacks.

4. Determine and perform the action you want to take, for example, such as cancelling or suspending the account in question.

14.3 Data Reporting to Vindicia

You may choose to report transaction and activity data to Vindicia either in real-time, or in batches. To report in real-time, send data to and receive data back from Vindicia one item at a time. To send batch reports, gather the information on multiple items and send it all at the same time. In both cases, you must collect data from your system, and assign it to the appropriate data structures, as defined later in this chapter.

Note: To leverage ChargeGuard's risk-screening feature, send the data to Vindicia in real time.

14.3.1 Initial Load of Historic Data

Before Vindicia can begin processing your chargebacks, you must send an initial set of transactions to be loaded into ChargeGuard. Because chargebacks are typically issued by the customer within 30 to 90 days after the transaction, ChargeGuard must have at least 90 days' worth of data to begin.

Note:	This data requirement applies to first-time ChargeGuard subscribers only. If you already subscribe to CashBox, Vindicia has your transaction data and you need not resend it. However, we strongly recommend that you send us the historical activity data, which often serves as valuable background information for chargeback disputes

The CashBox API allows you to report a vast array of information on your transactions and customer activity. Although many data items are optional, providing Vindicia with more information increases the success rate of winning chargeback disputes.

14.3.2 Key ChargeGuard Objects

The data that Vindicia and you exchange for ChargeGuard is represented by objects in the CashBox API. The most important objects for ChargeGuard are:

1. **Transaction:** The `Transaction` object encapsulates all transaction information, including the amount, payment method, associated customer account, and transaction status. (Most of this information is also provided to your payment processor.)

   The following Transaction information must be sent to Vindicia to provide evidence for disputes:

   - Transaction data fields
   - Item information
   - Rebill information
   - Payment method, including information on the credit card and the bank, if applicable
   - Customer information, including the customer name, billing address, and shipping address

   If post-transaction activity occurs, ChargeGuard can capture it for any subsequent chargeback disputes. Use the `Activity` object to notify Vindicia of your decision to complete, authorize to capture, or cancel the transaction.

2. **Activity:** The `Activity` object contains information about your interaction with a customer outside of the monetary transaction, such as a phone or email contact or a login to your site.

3. **Chargeback:** The `Chargeback` object contains information about a chargeback against a specific transaction, including the chargeback status provided by Vindicia during the dispute process.

4. **Refund:** The `Refund` object contains information about a refund from you to a customer for a specific Transaction. ChargeGuard applies partial and full refund data to transactions and to chargeback processing through the `Refund` object by associating the refund with the original transaction ID.

14.3.3 Reporting Transaction Data to Vindicia

The Transaction and MigrationTransaction objects support methods to migrate Transaction information into Vindicia, and to receive a chargeback risk percentage. These classes require that you instantiate your merchant user name and password assigned by Vindicia.

To send transaction information to Vindicia:

1. Collect the information about the transactions.
2. Create either a `MigrationTransaction` object (for use with `Transaction.migrate`), or a `Transaction` object (for a `Transaction.score` request).

To migrate historic data to CashBox, please see Section 5.4: Importing AutoBills from other Billing Systems to CashBox, or Transaction.migrate in the ***CashBox API Guide***,

To report data in real time (for `Transaction.score` requests), see Reporting Real-Time Transaction Information for Fraud Screening below.

To report the data in real time, see Reporting Real-Time Transaction Information for Fraud Screening in the next section.

After you send data to Vindicia, Vindicia returns a `Return` object with `returnCode` and `returnString` to inform you if the communication with the Vindicia server completed successfully. (Codes for the `Return` object are modeled after the standard HTTP return codes.) If the call succeeds, you receive a code of `200` and the string `OK`. `returnCode` and `returnString` may be used to interpret errors. See The Return Object in the ***CashBox API Guide***, for more information.

Be certain to act upon the `Return` value.

Reporting Real-Time Transaction Information for Fraud Screening

To report the transaction information in real time and receive a chargeback probability score, call the `score()` method on the `Transaction` object. (Be certain to pass only a single `Transaction` object to the `score()` call.) This call not only reports your transaction details to Vindicia but also returns to you a chargeback probability score (also called a risk score).

For the `score()` call to succeed, your transaction must contain at least the following attributes. If you do not specify any one of them, the call returns a score of `-1`, which means “no opinion.”

- IP address
- Payment method: Billing address: City
- Payment method: Billing address: State (“district”)

  State is a required attribute: Do not leave it unspecified and do not specify a value of `null`. If the state is not known or does not exist, set this attribute to `Unknown`.
- Payment method: Billing address: Country
- Payment method: Billing address: Zip code

  For countries with no zip or postal codes, set this attribute to `None`.
- Credit card BIN (the first six digits of the credit-card number)

Report real-time Transaction data to Vindicia:

```
$tx = new Transaction();
$tx->setAmount('29.90');
$tx->setCurrency('USD');
$tx->setMerchantTransactionId('txid-123456');

// IP is one of required attributes for scoring a transaction
$tx->setSourceIp('35.45.123.158');

$account = new Account();
$account->setMerchantAccountId('9876-5432');
$account->setEmailAddress('jdoe@mail.com');
$account->setName('J Doe');
$tx->setAccount($account);

$shippingAddress = new Address();
$shippingAddress->setName('Jane Doe');
$shippingAddress->setAddr1('44 Elm St.');
$shippingAddress->setCity('San Mateo');
$shippingAddress->setDistrict('CA');
$shippingAddress->setPostalCode('94403');
$shippingAddress->setCountry('US');

$tx->setShippingAddress($shippingAddress);

// The line items of the transaction
$tx_item = new TransactionItem();
$tx_item->setSku('sku-1234');
$tx_item->setName('Widget');
$tx_item->setPrice('3.30');
$tx_item->setQuantity('3');
$tx->setTransactionItems(array($tx_item));

$paymentMethod = new PaymentMethod();
$ccCard = new CreditCard();
$ccCard->setAccount('4111111111111111');
$ccCard->setExpirationDate('201109');
$paymentMethod->setType('CreditCard');
$paymentMethod->setCreditCard($ccCard);

// Billing address city, district, country are required for score
// call to work
$paymentMethod->setBillingAddress($shippingAddress);

$tx->setSourcePaymentMethod($paymentMethod);

$response = $tx->score();
```

```
if ($response['returnCode'] == 200) {
    if($response['score']->score <= 50) {
        print "Acceptable score, processing transaction";
        // process the transaction further here
    }
    else {
        print "High risk of chargeback. Reasons are: \n";
        $scoreCodes = $response['scoreCodes'];
        foreach ($scoreCodes as $scoreCode) {
             print("Score code ". $scoreCode['id'] . " : " .
                  $scoreCode['description'] . "\n");
        }
    }
}
else {
    // the score call did not succeed, check return code
    // and return string and try to re-submit
}
```

The `score()` call returns a `Return` object that describes the success or failure of the call, the chargeback probability (`score`), and an array of reason codes (`scoreCodes`) that explain the risk score.

Based on the transaction information provided, the chargeback probability ranges from 0 to 100. A probability of 100 indicates that CashBox is 100 percent certain that this transaction is fraudulent and will result in a chargeback. The score can also be `-1`, indicating no opinion from Vindicia; or `-2`, indicating an error condition. Based on the score, you can decide to either proceed with the transaction (by capturing it) or cancel it.

For more information on `scoreCodes` and the `score()` call, see Section 18: The Transaction Object in the ***CashBox API Guide***.

Reporting Activity Information

To combat fraudulent chargebacks, Vindicia uses your records to build an evidentiary record of your customer's activities, and challenge a chargeback case on your behalf.

Usage data, which usually resides in the commerce server on your site, is not mandatory, but can be key in helping win chargeback disputes. Examples of usage data are logins and logouts, visits to certain Web pages, email or phone communications, and shipping confirmations. This data can help counter customer claims that they did not make a specific purchase. Vindicia strongly suggests that you maintain a record of usage data. If you sell digital goods, most of this information data is required by the issuer in case of chargeback disputes. Be sure to report all the events that are significant or relevant, including email or phone communications, and access to or downloads of for-pay content.

The `Activity` object contains data structures through which you can report the following types of customer activities:

- Logins to a site
- Logouts from a site
- Views, visits, or downloads of a Web resource (URI views)
- Phone contacts with a customer

- Email contacts with a customer
- Fulfillment of an order for physical goods
- Use of quantifiable objects that are meaningful to your business
- Cancellation of a service
- An arbitrary note that contains a maximum of 1,024 characters

Report these activities to Vindicia in batch mode, when your transaction processing system is relatively quiet.

Record a phone contact with a customer as an Activity:

```
$soap_act = new Activity();

// Create an account object
$account = new Account();

// Specify account by the customer id
$account->setMerchantAccountId('9876-5432');

// Now create Activity to report a customer's phone call
// and corresponding ActivityTypeArgs objects

$activity = new Activity();
$typeArgs = new ActivityTypeArgs();

// fill in the relevant info for this activity record
$activity->setAccount($account); //associate the activity and account
$activity->setActivityType('Phone');
$activity->setTimestamp(getdate());

$phoneArgs = new ActivityPhoneContact();
$phoneArgs->setCidPhoneNumber('1234567890');
$phoneArgs->setDurationSeconds(367)
$phoneArgs->setType('FromCustomerToMerchant');
$phoneArgs=>setNote('Customer agreed to be rebilled for services');

$typeArgs->setPhoneArgs($phoneArgs);

// associate typeArgs to the Activity object
$activity->setActivityArgs($typeArgs);

// now record the data
$response = $soap_act->record(array($activity));

if($response['returnCode'] == 200) {
   print "ok\n"; # 200 is HTTP status code for success
}
```

For more information, see Section 2: The Activity Object in the ***CashBox API Guide***.

Reporting Refund Information

If you process transactions and refunds through CashBox, you need not report refunds separately to Vindicia.

Report refunds to Vindicia for transactions processed outside CashBox:

```
$refundVid = 'MyVindiciaRefundVID';

// Create a refund object
$refund1 = new Refund();
$refund1->setMerchantRefundId('REF101');

$transaction1 = new Transaction();
// merchant ID of a previously reported transaction
$transaction1->setMerchantTransactionId('TX101');

$refund1->setTransaction($transaction1);
$refund1->setAmount(5.99);
$refund1->setNote('Refunded due to service outage');
// Payment Processor's refund id when you processed
// this refund with it directly - if available
$refund1->setReferenceString('2033992');

// Create another refund object
$refund2 = new Refund();
$refund2->setMerchantRefundId('REF102');

$transaction2 = new Transaction();
// merchant ID of a previously reported transaction
$transaction1->setMerchantTransactionId('TX102');

$refund2->setTransaction($transaction2);
$refund2->setAmount(10.99);
$refund2->setNote('Customer did not receive delivery);

$soap_refund = new Refund();
$response = $soap_refund->report(array($refund1, $refund2));
if($response['returnCode'] == 200) {
   print ("All refunds submitted successfully");
}
```

If you refund customers outside of your CashBox system, you must report the refunds to Vindicia so that ChargeGuard can use them when disputing chargebacks. To report refunds in batch mode, first construct a batch of `Refund` objects, as shown above. For more information, see Section 15: The Refund Object in the ***CashBox API Guide***.

14.4 Retrieving Chargeback Updates

The `Chargeback` object supports a `fetchDeltaSince()` call with which you can retrieve chargebacks that have changed in status or that have been newly added since the timestamp you specify as an argument for the call.

Fetch chargebacks that have changed in a specific time frame:

```
$cb = new Chargeback();
$page = 0;
$pageSize = 50;

// Here we want to fetch chargebacks that have changed in status or
// have been added since the last time we ran this call.
// Assume we have a function available to us that gives us
// the timestamp for the last time we ran this call

$since = getLastCallTime();
do {
    $ret = $cb->fetchDeltaSince($since, null, $page, $pageSize);
    $count = 0;
    if ($ret['returnCode'] == 200) {
        $fetchedChargebacks = $ret['chargebacks'];
        if ($fetchedChargebacks != null) {
             $count = sizeof($fetchedChargebacks);
             foreach ($fetchedChargebacks as $chargeback) {

               // process a fetched chargeback here …
               $status = $chargeback->getStatus();
               $transactionId =
                      $chargeback->getMerchantTransactionId();
               $amount = $chargeback->getAmount();
             }
             $page++;
        }
    }
} while ($count > 0);
```

To retrieve all the chargebacks that match the search criteria, construct the sample above in a loop by incrementing the page number for each iteration until the returned number of chargebacks in a page is less than the specified page size.

The `Chargeback` object also supports several other `fetch` calls to retrieve chargebacks through a variety of search criteria. For details, see Section 7: The Chargeback Object in the ***CashBox API Guide***.

Make this call to Vindicia at periodic intervals with a UNIX daemon, a Microsoft Windows scheduler, or a similar technology. Use this process to automate the tasks of downloading chargebacks, learning their status, and enabling or disabling customer accounts.

Appendix A Custom Billing Statement Identifier Requirements

The Billing Statement Identifier field enables merchants to define the line of text that will appear on their customers' credit card statements, in association with the related charge. To enable this field, payment processors require merchants to provide certain information, in a specific format.

(If you work with several payment processors, please be certain to consider their individual requirements when deciding how to populate data on your CashBox objects.)

This appendix describes the data and configuration requirements for creating Payment Processor specific Billing Statements for Chase Paymentech, GlobalCollect, Litle & Co., and Merchant e-Solutions (MeS). While the requirements for these three processors are virtually identical, specific differences are called out, where appropriate.

To comply with Visa transmission rules, the following information must be included with the Billing Statement Identifier field:

- a Visa-issued Merchant Category Code (MCC),
- an associated Merchant Name (if applicable),
- a Customer Service Phone Number, and
- a Description of the product or purchase.

A.1 Billing Statement Identifier

Customize a Billing Statement Identifier in CashBox using either:

- the Billing Statement ID on a Billing Plan for an AutoBill in the CashBox UI; or
- the `billingStatementIdentifier` attribute in the SOAP API.

For credit card-based recurring billing, set the attribute for the `BillingPlan`, `Product`, or `AutoBill` object. CashBox will then insert the identifier into every Transaction generated for the `AutoBill`, and the identifier will appear on your customer's next billing statement. If you set the attribute on all three objects, the order of precedence is `BillingPlan`, `Product`, `AutoBill`.

For real-time Transactions, set the `billingStatementIdentifier` attribute for the `Transaction` object.

Billing Statement Identifier example: `website.com|3101231234`.

Note	The asterisk is a reserved character that is not allowed in the Billing Statement Identifier.

A.2 MCC-Associated Merchant Name

Work with your payment processor to provide Vindicia Client Services with your MCC (Merchant Category Code) and associated Merchant Name.

- Obtain your MCC and the associated Merchant Name (set by Visa) from your payment processor account representative. The Merchant Name may be 3, 7, or 12 characters.
- Send the information to Vindicia Client Services.

Vindicia will add your Merchant Name to your CashBox configuration for the Transactions Vindicia submits on your behalf. Take note of the length of the Merchant Name; it will affect the allowable length of description text (see Section A.4: Billing Description).

Note:	The length of the Merchant Name associated with your MCC will directly affect the maximum allowable length of your Description. For more information, see Section A.4: Billing Description.

Do not pass the Merchant Name in the Billing Statement Identifier. Pass only the MCC.

A.3 Default Customer Service Phone Number

Provide Vindicia Client Services with a default customer-service phone number for each Chase Paymentech Division ID, GlobalCollect Billing Descriptor, Litle & Co. Merchant ID, or MeS Profile ID, including the default ID.

For Chase or MeS: Provide a 10-digit number separated by dashes (NNN-NNN-NNNN or NNN-NNNNNNN) or a three-digit number followed by a 7-digit alphanumeric code (for example, 800-CALLNOW).

For Litle: Provide a 10 digit phone number for US billing addresses, and up to 13 digits for non-US addresses.

For all listed payment processors, Vindicia will add the information to your CashBox configuration, and enable the default phone number for your Division ID, Merchant ID, or Profile ID.

Note: The Billing Statement ID will not work without this phone number.

Overriding the Default Customer Service Phone Number

To override the default phone number, use the Billing Statement ID field to append a product description with the pipe symbol (|), followed by the desired number. (Non-numeric characters will be stripped from the phone number.)

For example: `Product XYZ | 877-555-1212`.

In the CashBox UI, enter the Description and override phone number in the **Billing Statement ID** field of the Billing Plan.

With the CashBox API, provide the Description and override phone number on the `billingStatementIdentifier` data member of the `BillingPlan`, `Product`, `AutoBill` or `Transaction` object. Use the symbol "`->`" to signify "is overridden by, if present." If the override number is provided for several object, the order of precedence is `BillingPlan > Product > AutoBill > Transaction`.

For Chase or MeS: Chase and MeS will reject the Transaction if the override number exceeds 10 digits. To prevent this, CashBox will automatically use the default number for your Division or Profile ID for any override numbers exceeding 10 digits.

For Litle: Litle will reject the Transaction if your override number exceeds 13 digits. To prevent this, CashBox will automatically use the default number for your Merchant ID for any override numbers exceeding 13 digits.

A.4 Billing Description

The Description is a string, up to 22 characters long, which includes the length of the MCC-associated Merchant Name, and an asterisk (*), leaving 9 to 18 characters for the descriptive text. (CashBox will automatically truncate any Description string exceeding 22 characters.)

The Description should be recognizable to the account holder. It should consist of the company name and/or trade name (Merchant Name), combined with a description of the product or service purchased.

There are three possible formats:

- 3-character Merchant Name, an asterisk (*), and an 18-character description
- 7-character Merchant Name, an asterisk (*), and a 14-character description
- 12-character Merchant Name, an asterisk (*), and a 9-character description

Each description must be validated by your payment processor's Risk Department before being put in use.Vindicia cannot verify that this step has occurred; you must secure the validation yourself.

Valid characters in the Description string include:

- Numbers
- Letters
- Special characters: ampersand (&), comma (,), dash (-), period (.), pound sign (#)

The asterisk is a reserved character for marking the end of the MCC only. Do not include an asterisk in your descriptor.

Note: Chase Paymentech will reject Transactions which use the following symbols, with Response Reason Code 225 (invalid field data): caret (^), backslash (\), open square bracket ([), closed square bracket (]), tilde (~), or accent (`).

For example, if Vindicia's MCC-associated Merchant Name were `VIN`, and our Billing Statement Identifier for CashBox were `VIN* CashBox Software`, the identifier would appear on the credit-card holder's statement as

```
VIN* CashBox Software 650-264-4700.
```