CFPAYMENT API Overview

CFPAYMENT is a payment processing abstraction library for credit card and EFT gateways. It is based roughly on the Ruby ActiveMerchant library. We have organized the project into several layers that can be used depending on your needs:

1. Core (basic processing, error handling, responses)
2. Transaction (wrap core with pre/post database storage for reliability, reporting)
3. High-availability (wrap transaction with failover capability)

The configuration of each gateway may be different but all should respond with a single, normalized response object:

```
.purchase(money, account, options) => response
.authorize(money, account, options) => response
.capture(money, authorization, options) => response
.void(transactionid, options) => response
```

Each gateway maps its unique implementation details into a common response object.

The Core and Transaction APIs are designed for most developers who want to do the most common thing: process payments for a single merchant account against a single gateway.

LICENSING

CFPAYMENT is licensed under the **Apache Software License 2.0**, the full text of which is at http://www.apache.org/licenses/LICENSE-2.0. Generally speaking, the ASL allows you to use this code in any way you see fit, including in closed-source or commercial software, so long as the copyright notice stays intact and you clearly mark any changes you make. While you are not required, please consider contributing enhancements back to the project so that everyone benefits. That's how open source works!

CORE API

core.init(config)
.createCreditCard() - for credit card transactions with .validate()
.createEFT() - for e-check transactions with .validate()
.createResponse() - normalized response object
.createMoney() - money object handles amounts and currencies, will do conversion in future
.getGateway() - returns the actual gateway

The core is effectively a factory for generating objects and instantiating gateways. Actual gateway implementations extend cfpayment.api.gateway.base which provides a boilerplate interface. The core service is initialized with a config object (struct) having a path to gateway object, MID, username, password, etc:

```
cfg_cc = {path: 'itransact.itransact_cc'
      ,mid: 123456
      ,username: production
      ,password: production}
```

```
cfg_eft = {path: 'itransact.itransact_eft'
      ,mid: 223422
      ,username: test
      ,password: test
      ,TestMode: true}        // offer way of toggling on a per-gateway basis

cfg_bt = {path: 'braintree.braintree'
        ,mid: 654321
        ,username: btree
        ,password: btree
        ,failOnAVS: true    // additional config options on a per-gateway basis could
support custom capabilities
        ,failOnCVV: true}
```

Gateways are designed to be in test mode by default! That is, it requires an explicit "TestMode: false" configuration to enable live processing of transactions.

Each gateway implementation extends cfpayment.api.gateway.base. The base component provides the network transmission and error handling for all gateway implementations in process(). This centralizes the actual network component in a single location where we can focus on the most robust error handling known to man kind. With payment processing, it is critically important to be able to recover when things go wrong to prevent double charges and keep records accurate. My experience with a flaky gateway company in the past has given me great insight into how to manage these exceptions.

This could be overridden for a gateway that used a protocol other than HTTP or had some other unusual requirements. It could also be extended and executed via super.process() depending on requirements. Most developers will simply call it normally:

```
basegw.init()
       .process() - package access, handles all network transport and error handling
```

Individual gateways implement the following:

```
gateway.purchase(money, account, options) - authorize+capture or a specific method
like some gateways offer
        .authorize(money, account, options)
        .capture(money, authorization, options)
        .void(id, options)
        .credit(money, id, options)
        .status(options)
        .recurring(money, account, options)
        .settle(options)
```

others could be:

```
        .store(account, options) - vault
        .unstore(account, options) - delete from vault
        .get(id, options) - vault
        .status(transactionid, options) - return transaction status
```

Gateway methods return response objects which normalize the results with an API like:

```
response.getSuccess() - true/false, did the transaction succeed
       .hasError() - if !getSuccess(), is there an error or was it just declined?
```

.set/getStatus() - get the status code, -1 to 5, defined in core.cfc. This is more valuable than just success/fail because you probably want to handle connection timeout differently than declined.
.set/getMessage() - the result in plain text
.set/getResult() - get the raw result from the gateway
.set/getParsedResult() - get the parsed result (after some processing by the gateway)
.set/getTest() - is this a test transaction? *** not sure about this, or how I want to handle test transactions in general... each gateway is so different
.isValidAVS(allowblank, allowpostalonlymatch, allowstreetonlymatch)
.isValidCVV(allowblank)
.get/setCVVCode() - get or set the result character (single character)
.get/setCVVMessage()
.get/setAVSCode() - get or set the result character (single character)
.get/setAVSMessage()
.get/setAVSPostalMatch()
.get/setAVSStreetMatch()

We use a money object to track the amount to be charged and the currency in which to charge it. Currently this is more or less a placeholder until we get proper currency conversion and other features into place but it can be used to pass a different currency to a gateway in its current implementation.

money.init(cents, currency)
.set/getCents() - we store amount as an integer in "cents"
.getAmount() - getCents() / 100 as a convenience function.
.set/getCurrency() - defaults to USD but can be changed; no auto conversion currently; uses three-letter ISO codes

Methods like authorize(), purchase(), etc take an options structure for additional parameters to send to the gateway as either URL or FORM variables depending on GET/ POST. Some examples might include:

- External ID
- Currency type
- IP address
- Tax Rate / Tax Amount
- Country code

Exceptions and Validation

In general, the API throws errors that can be caught with CFTRY/CFCATCH for unrecoverable errors introduced *by the developer*. Our model objects like creditcard and eft come with a validate() routine which returns an array of errors and helper function getIsValid() to determine if the *user-supplied* data is valid. The idea here is that we throw errors for things that should be corrected during development and validate for things that can be corrected in production.

The core API throws the following exceptions:

- cfpayment.InvalidGateway - the gateway specified by the config object does not exist or could not be instantiated. This is probably because your path to the gateway CFC is wrong. It should be relative to the "gateways" folder so cfpayment/api/gateways/bogus/gateway.cfc would be specified as "bogus.gateway".

- cfpayment.InvalidAccount - the account type passed is not supported (e.g., used a creditcard for a check operation)
  cfpayment.MethodNotImplemented - the method has not been written or is not supported (e.g., calling authorize() for e-checks (which only have purchase typically) cfpayment.MissingParameter
    - cfpayment.MissingParameter.Argument - a required argument was missing
    - cfpayment.MissingParameter.Option - a required attribute in the Options structure was missing. These are checked in gateway implementations using the verifyRequiredOptions() method

  cfpayment.InvalidResponse
    - cfpayment.InvalidResponse.AVS - the returned AVS code (a single character) was not understood - this may mean a new response type has been introduced that needs to be added to the response object
    - cfpayment.InvalidResponse.CVV - the returned CVV code (a single character) was not understood - same result as AVS.

If you're not familiar with custom exception type handling in ColdFusion, you can catch them like so:

```
<cftry>
   <cfset bogusGateway.credit(money = myMoney, account = myAccount, options =
myOptions) />

   <cfcatch type="cfpayment.MissingParameter.Argument">
      // do something when an argument is missing
   </cfcatch>
   <cfcatch type="cfpayment.MissingParameter">
      // do something if any kind of missing parameter error is throw, .Argument,
.Option, etc
   </cfcatch>
   <cfcatch type="cfpayment.MethodNotImplemented">
      // do something if this method is not implemented
   </cfcatch>
   <cfcatch type="cfpayment">
      // catch any other kind of cfpayment.* error type not specifically caught above
from cfpayment.InvalidResponse.CVV to cfpayment.InvalidGateway
   </cfcatch>

</cftry>
```

TRANSACTION API

The Core API illustrates the simple, building-block approach to payment processing. The Transaction API was born out of five years of production experience and understanding the full range of things that can go wrong with any given transaction. Eventually something will go bump in the night and a transaction will fail. Being able to reconcile these transactions either automatically or manually is a critical component of ensuring that your records are accurate and your customers were not charged more than once.

Generally speaking, the Transaction API is simply functionality that executes before and after the Core API. It requires our database tables to be present. It first inserts the payment attempt in the database, then attempts to process the payment using the Core API, then updates the database with the results and returns them. Probably extends the response object to look more like our current "payment" transfer object which knows

more about its ID, etc. May add methods like:

cfpayment-transaction.init(config, encryptionService) extends core
.set/getRequest() - get the original request

Just wraps the API of the core API, works with a single gateway at a time. Our implementation could either:

a) .purchase(), .void(), .capture() but that means transaction interface has to have every method possible in every gateway. OR, use onMissingMethod() to support any method but requires CF8. Could AOP be another solution here?
b) .transaction("method", params) which runs "method" on the underlying gateway. Cleaner implementation wise but won't allow swapping from non-trans to trans cleanly.

Optionally pass in an encryption service. If present, we encrypt account details and store in database otherwise we just store what we store today (last four, cvv2 response, avs response, etc). In our case, we want the encryption service to also encrypt details as soon as they are received and decrypt them only when going to pay. Encryption service
should be passed into model objects (CC and EFT) to allow them to encrypt and decrypt their details (based upon available keys, for us, only public key is available on web and private key is available on pay) but other people could have a single symmetric key available on a single box).

Encryption service needs to support just three methods:

.init(...)
.encryptData(...)
.decryptData(...)

However it gets this done, whether it uses built-in encrypt/decrypt functions, uses an asymmetric Java library or ties in with a hardware encryption device, the implementation is hidden behind the simple interface which CFPAYMENT understands.

HIGH-AVAILABILITY API

cfpayment-ha.init(array_of_configs, encryptionService) extends transaction

where config is an array of gateway config objects like:

```
cfg = [{path: 'itransact.itransact_cc'
      ,mid: 123456
      ,username: test
      ,password: test
      ,priority: 1
      ,weight: 100}
  ,{path: 'braintree.braintree'
        ,mid: 654321
        ,username: btree
        ,password: btree
        ,priority: 2
        ,weight: 100}
  ];
```

Exposes more or less same API but internally uses the transaction API to see if charges

fail. If they fail due to a gateway timeout or error condition, it may automatically try the next gateway. Could be configured to have a "threshold" setting like 0 = no failover, 1 = for gateway failures, 2 = for gateway failures and any declines. Still under development.

Builds an array of transaction objects internally with a priority or ID to control failover (including threshold for retrying)

Can also be used for load balancing between multiple gateways which can be used for various reasons.

Supported Gateways

Name	Purchase	Authorize	Capture	Void	Credit	Recurring	Status	Account Types
Braintree	Y	Y	Y	Y	Y	N	Y	CC, CHK, STORAGE
iTransact	Y	Y	Y	Y	Y	N	Y	CC, CHK
SkipJack	Y	Y	Y	S	S	Y	N	CC

Y - Full support
N - No support
S - Support coming soon

Planned Support

- Paypal (seems like a must-have for adoption, could convert ASL-licensed CFPaypal @ http://www.indiankey.com/cfPaypal/; Arjun has tentatively agreed to help convert his CFpaypal into cfpayment, may also get help from http://www.coldfusionguy.com/ColdFusion/blog/index.cfm/2007/10/28/PayPalcfc-That-Returns-a-Structure-Instead-of-a-Java-Object or Jared Rypka Hauer has a kind-of commercial package at http://www.web-relevant.com/web-relevant/lib/paypalservice/docs/index.cfm but he may be willing to help with the low-level stuff?)
- Google Checkout would be nice, another "integration", same with Amazon payments
- Authorize.net, direct, ARB and CIM

System Requirements

- ColdFusion MX 7
- CreateObject()
- Ability to create a mapping OR put the files in a folder off of the webroot.

Examples

We have included two examples that show how to use the gateways. They both default to using the bogus gateway, but can be easily changed with one line of code. To run them, make sure you can access the cfpayment folder from your webroot, then visit the examples home page (changing **localhost** to whatever your workstation/server name is):

http://localhost/cfpayment/docs/examples/

The Simple Checkout has a somewhat generic credit card form that submits to the bogus gateway. It shows any errors in both your form fields and gateway results at the top of the form. The _process.cfm file can be used as an example of how to process form submissions, but stops short of doing anything with the gateway processing results.

The ColdSpring example shows how you can instantiate the cfpayment service core and gateway using a coldspring xml file. It also implements a simple AOP logging system that tracks calls to the gateway and stores them in the request scope. This example requires that you have coldspring installed and accessible:
http://www.coldspringframework.org/

Notes for Creating a Gateway