CFPAYMENT 1.0

Project founder: Brian Ghidinelli / http://www.ghidinelli.com
Core contributor: Mark Mazelin / http://www.mkville.com

CFPAYMENT is a library designed to make implementing e-commerce transactions easy in ColdFusion. Rather than roll your own gateway with error, currency and money handling, either leverage one of our existing gateways or extend our base gateway and write only the code necessary to create requests and parse responses for your gateway. This eliminates writing lots of boilerplate code and handling esoteric CFHTTP errors that only seem to happen in production.

You may notice copyright notices in some files going as far back as 2008. While we are just now releasing a 1.0, this code has processed tens of millions of dollars in mission-critical e-commerce systems. Use it with confidence!

Overview

CFPAYMENT is a payment processing abstraction library for credit card, ACH and third-party payment gateways (like Paypal). It is inspired by the Ruby ActiveMerchant library and is designed to work with any payment gateway including name-value pairs, XML, RESTful or any other API. The project is organized into several layers that can be used depending on your needs:

Library	**Description**	**Status**
Core	Basic processing, error handling, currency support, gateway responses	AVAILABLE (1.0 as of 3/21/2012)
Transaction	Wrap core with pre/post database storage for reliability, reporting	Pending
High-Availability	Wrap transaction with failover and multi-gateway support	Pending

One of the core goals of CFPAYMENT is to abstract away the specifics of individual payment gateways to make it easier to change gateways when the need arises. This provides more flexibility for the business without sacrificing anything at the code level.

As a result, each gateway maps its unique implementation details into a common response object:

```
.purchase(money, account, options) => response
.authorize(money, account, options) => response
```

.capture(money, authorization, options) => response
.void(transactionid, options) => response

The Core and Transaction APIs are designed for most developers who want to satisfy the most common need: process payments for a single merchant account against a single gateway. The high-availability API is designed for more advanced e-commerce operations who have multiple merchant accounts and gateways for failover or load balancing.

Stupid Simple Code Sample

Here's what it takes to get started with CFPAYMENT in just a few lines of code:

```
var cfg = { path = "braintree.braintree", Username = "test", Password
= "password", SecurityKey = "1234567890", SecurityKeyID = "1234", TestMode
= true }


cfpayment = createObject("component", "cfpayment.api.core").init(cfg);
gateway = cfpayment.getGateway();


money = cfpayment.createMoney(5000); // $50.00 in cents
account = cfpayment.createCreditCard().setAccount("4111111111111111")
  .setMonth(10)
  .setYear(2015)
  .setVerificationValue(999)
  .setFirstName("John").setLastName("Smith")
  .setAddress("888 Main Road").setPostalCode("77777");


response = gateway.purchase(money = money, account = account);


if (response.getSuccess())
  // charge went thru!  you're rich!
else
  // charge failed, why?
  writeOutput(response.getMessage());
```

Easy, eh?

System Requirements

- ColdFusion MX 7+ (Note: examples use CF8 inline structure creation syntax)
- CreateObject() must be enabled on the server
- Ability to map "cfpayment" OR put the files in a folder off of the webroot
- Optional: Java object access will enable smart timeout handling

Supported Gateways

Authorize.net	
Braintree	Orange platform currently (including transparent redirect and vault support), Blue platform coming soon
GoEMerchant	
iTransact	XML backend gateway
Paypal	Website Payments Pro
SkipJack	
Stripe	
VirtualMerchant	Under development
Not listed?	If you don't see your gateway listed here, adding support is super easy. Check out our Notes for Creating a Gateway

CORE API

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 by passing in a configuration argument (ColdFusion structure) having a path to the gateway object and various other parameters such as MID, username, password, etc:

Method	**Description**
.init(config)	Initialize the core API with a struct and configure a gateway implementation. Each gateway has its own config parameters that typically include usernames, passwords, test mode flag, merchant account ID, etc. See Appendix A for examples.
.getVersion()	Return the CFPAYMENT version you are using.
.createCreditCard()	Return a credit card object that holds name, address, cardholder data and has verifications for valid card numbers and card type (Visa, M/C, AmEx, Discover). Can return a masked credit card number in compliance with PCI DSS.
.createEFT()	Return an EFT object that holds name, address, and bank account information.
.createMoney()	Return a currency object to hold an amount. Supports currencies that do not divide units into 100 parts.

.createToken()	For tokenized gateways such as Braintree, Authorize.net and TrustCommerce which return identifiers that represent cardholder data without requiring you to hold on to the original numbers.
.getGateway()	Return the gateway specified by the configuration passed to init()
.getStatusUnprocessed()	Denotes a transaction is not yet processed
.getStatusSuccessful()	Denotes transaction was processed
.getStatusPending()	Denotes transaction has been submitted to gateway but has not yet returned with a result.
.getStatusDeclined()	Transaction was declined
.getStatusFailure()	Indicates something went wrong like the gateway threw an error but we believe the transaction was not processed
.getStatusTimeout()	The request to the gateway timed out leaving the transaction in an unknown state.
.getStatusUnknown()	An exception that we don't know how to handle (yet) has occurred.
.getStatusErrors()	Returns a list of statuses that are considered an error condition (Failure, Timeout, Unknown)

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. See the Notes for Gateway Developers at the end of this document for more information on gateway creation.

Once the core object has been created using createObject(), you can create a gateway object by calling core.getGateway(). Individual gateways implement one or more of the following methods:

Method	**Description**
.purchase(money, account, options)	Authorize + capture in a single transaction
.authorize(money, account, options)	Authorize a transaction (usually returning an authorization number for later capture)
.capture(money, authorization, options)	Use a previous authorization to capture a transaction
.void(id, options)	Void a previous transaction that has not yet settled

.status(transactionid, options)	Retrieve details about a transaction
.recurring(mode, money, account, options)	Set up or modify a recurring transaction
.settle(options)	Some gateways require an explicit settlement call to close a batch.
.store(account, options)	Put an account into a vault like with Braintree, usually returns a token
.unstore(account, options)	Remove an existing account from the vault
.get(id, options)	Retrieve an account from the vault

RESPONSE OBJECT

Method	**Description**
.getSuccess()	true/false if the transaction succeeded
.hasError()	If NOT getSuccess(), is there a defined error or was the transaction just declined?
.set/getStatus()	Get the status code as 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 (e.g., name-value pairs, XML, JSON, etc)
.set/getParsedResult()	Get the parsed result after some processing by the gateway into a ColdFusion data structure like struct or array
.set/getTest()	Is this a test transaction?
.isValidAVS(allowBlank, allowPostalOnlyMatch, allowStreetOnlyMatch)	Did the address verification system (AVS) pass? To what extent?
.isValidCVV(allowBlank)	Did the security code check pass?
.get/setCVVCode()	Get or set the result character
.get/setCVVMessage()	Get the text description of the result which can be displayed to the user.
.get/setAVSCode()	Get or set the result character

.get/setAVSMessage()	Get the text description of the result which can be displayed to the user.
.get/setAVSPostalMatch()	Did the Postal/Zip code match?
.get/setAVSStreetMatch()	Did the street address match?
.get/setTokenID()	Get the vault token ID associated with this transaction.

MONEY OBJECT

Gateway methods use a money object to track the amount to be charged and the currency in which to process. Currently this is more or less a placeholder for more robust currency conversion but it can be used to pass a different currencies to a single gateway (like USD and CAD):

Method	**Description**
.init(cents, currency)	Not all currencies divide up their units into fractions of 100. We also want to be careful about rounding errors caused by floating point math. As a result, we store amounts as an integer in cents. $1.00 USD would be 100 cents. $50.50 USD would be 5050 cents. Currency defaults to USD.
.get/setCents()	Store amount as an integer in "cents"
.getAmount()	Return getCents()/100 as a convenience function. For currencies that don't use 100, this method would be overridden to divide by the right number.
.get/setCurrency()	Defaults to USD but can be changed. No auto foriegn exchange conversion performed at this time. Uses three-letter ISO codes.

Because gateway implementations vary so much, methods like authorize() and purchase() take an options structure for additional parameters to send to the gateway. The options structure gives cfpayment the flexibility to support multiple gateways, but also becomes the part that will vary as your application uses different gateways. Some examples might include:

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

The key to cfpayment is that we have tried to normalize these option names. While Braintree expects transaction_id and iTransact wants ExternalID, cfpayment defines it as transactionId. Each gateway developer translates the common transactionId into the required parameter for their gateway.

Exceptions and Validation

In general, the API throws errors that can be caught with CFTRY/CFCATCH for unrecoverable errors introduced *by the developer*. 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 is that CFPAYMENT throws exceptions for things that should be corrected during development and returns an array of errors for things that can be corrected in production.

The core API throws the following exceptions:

Exception	**Description**
cfpayment.InvalidGateway	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/gateway/**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	Method has not been written or is not supported. An example would be calling authorize() for e-checks which only have purchase() typically.
cfpayment.MissingParameter.Argument	Required argument was missing
cfpayment.MissingParameter.Option	Required parameter in the options structure was missing. These are checked in individual gateways using the verifyRequiredOptions() method.
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 (under development)

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 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. The API will appear to be a lot like our response object but with "stateful" information such as its unique ID, status, processor ID and so forth. These attributes allow a developer to relate a single payment attempt to the rest of their system.

HIGH-AVAILABILITY API (under development)

The High-Availability API takes the Transaction API and extends it work across multiple payment gateways. If a transaction fails and meets certain criteria, it is automatically retried against another gateway.

The config object is an array of gateway configs 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}
  ];
```

The High-Availability API can also be used for load balancing between multiple gateways if desired.

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 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

The easiest way to create your own gateway is to open up an existing gateway and make changes. Although it may look intimidating at first, most of your actual code will simply be converting the normalized CFPAYMENT parameter names into gateway-specific parameter names, building up the data for a request and passing it to process(). Then parse the response back into normalized CFPAYMENT parameters and use the common response object.

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 robust error handling. 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 gateway developers will simply call it without any additional processing.

Method	**Description**
.init(config)	Initializes the gateway. Automatically runs set*() when it finds an argument with a corresponding set method.
getGatewayName()	Get the gateway name as configured
getGatewayVersion()	Get gateway version as configured
get/setTimeout()	Control the network timeout (for CFHTTP)
.get/setTestMode()	Control test mode. Gateways *must* default to test mode. This requires the user to put it into production mode.
.getGatewayURL()	Get the URL to the gateway end point. Supports live and test URLs.
.getService()	Private method that can be used to access the core API for access to status methods or generating model objects.
.get/setMerchantAccount()	Control the merchant ID or account ID for the gateway
.get/setUsername()	Login credentials for gateway
.get/setPassword()	Login credentials for gateway
.get/setGatewayID()	Used by Transaction/HA APIs to differentiate which gateway processed a given transaction.
.getCurrentRequestTimeout()	Uses underlying Java to figure out how long the current request time out is set to. We use this to intelligently modify the page request timeout to make sure error handling can run successfully.
.process(method, payload, headers)	Transport and network/connection error handling. Expects a payload (either XML/JSON for a body request or a structure of name-value pairs) and an optional structure of headers. Normalizes the HTTP response code and response into a structure.
.doHTTPCall(url, method, timeout, headers, payload)	Private method that wraps CFHTTP to improve testing. Do not call this directly.

.createResponse()	Generate the response object with status set to unprocessed. On very rare occasion, a gateway may need to override this with a custom implementation.
.getOption(), .verifyRequiredOptions()	Use to verify that the options struct is properly populated for a given request.
.isValidPeriodicity(), .getPeriodicityValue()	Normalized recurring transaction values to make it more consistent how to pass in "weekly", "monthly", etc.
.getIsCCEnabled(), .getIsEFTEnabled()	Reports back if the gateway supports these two types of transactions.

Options Naming

While the base API is well defined in CFPAYMENT, each gateway may accept any number of optional or required parameters via the options structure. To simplify end-user consumption of the gateways, we have standardized the name of the keys for commonly used options. It is important that you translate these option keys to whatever your gateway requires to help keep the API homogenous.

Option Key	Value	Description
email	E-mail Address	
transactionId	Transaction ID	Id provided by gateway for referencing the processed transaction
ipAddress	IP Address	Address of the payee, not your server (typically CGI.remote_addr)
orderId	Internal Order or Payment ID	Some gateways allow you to record your ID in their system for reporting, email receipt or redirect use. We would recommend passing in whatever value you will record this payment as in your system. If you use an integer or UUID primary key, for example, pass the value here. Not all gateways support all data types or lengths here, so check the docs.
customerId	Internal customer ID	Some gateways allow you to record this in their system for reporting, email receipt or redirect use. If orderId refers to a single transaction, customerId would refer to "John Smith".

shippingAddress	Structure containing the following keys: name, company, address1, address2, city, region, country, postalcode, phone	While the billing address is stored in the *account* you pass to a method, you can also supply alternative shipping address in a structure.
orderDescription	Description of the purchase	
startDate	Start date	Used for status and reporting queries
endDate	End date	""
tokenId	Vault/Lockbox token identifier	For services that support remote data storage (Braintree, Authorize.net, TrustCommerce...), this is the normalized ID that references the stored token. For example, Braintree calls this customer_vault_id, so we map options.tokenId to customer_vault_id behind the scenes.
tokenize	Vault/Lockbox token trigger	Vault/lockbox services typically support two mechanisms for storing data: either explicitly as a "store" method, or as part of a regular transaction like an authorize or purchase to save the account details for later use. This boolean may be necessary when you want to trigger the creation of a token for a given account.

Testing

All gateways are required to have unit tests. If you are not familiar with MXUnit, copy the tests from the Braintree template as a starting point. You can get MXUnit at mxunit.org.

Support

If you need help developing a gateway, we are happy to help! Just hit up the Google Group at http://groups.google.com/group/cfpayment

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!

Appendix A: Sample Gateway Configs

When instantiating the core API, a configuration object must be passed in order to initialize the gateway. Each gateway will have its own requirements (read the docs). Here are some samples with common parameters such as account numbers, usernames and passwords:

```
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}

cfg_sj = {path: 'skipjack.skipjack_cc'
                  ,MerchantAcount="123456"
                  ,UserName="devuser"
                  ,Password="devpass"
                  ,DeveloperSerialNumber="11223344"  // additional config options on a per-
gateway basis support custom capabilities
                  ,LoginSerialNumber="55667788"
                  ,TestMode=true }
```