Worldpay Total Python SDK Demo (SNAP)

This package allows a Python developer to access all of the Worldpay Total REST APIs using native data structures. This allows her to focus her energy on creating the value added business logic and not get hung up in the details of communicating with the remote server.

Note that this package is not supported by Worldpay and is only provided as a proof of concept that you may keep and reuse in your own applications if desired.

Package Layout

There are three types of files in this package. Those starting with **wp** are part of the core services used to develop your own application. They provide all of the data structures, marshaling, and communication necessary to interact with the Worldpay Total API. These are the files you need to develop your application.

Files that begin with **x** are provided as examples of how to use the **wp** files. They provide a series of function calls, all starting with **do** that can be invoked to encapsulate a particular API. These 'do' functions are designed to help teach as well as provide a test harness to exercise the **wp** library. You would likely replace these files with your own application.

The third type of files are for general support and are only necessary for the testing of this application. Currently this includes wptestcard.py, code that uses testdata.csv for benign data sets that can be used in creating API transactions.

Functional Categories

There are seven main categories of operations. They follow the same layout presented on the API DOCS doRecurringPaymentPlan.

Auth - Card present/Card not present transactions
Chargebacks - Reversing or crediting a previous transaction
Settlement - Operations that affect the closing and status of a batch
Recurring - Creating/modifying installment, recurring, and variable payment plans
Tokens - Operations that engage tokenization
Vault - Adding/modifying customer records and transactions stored in the vault
Transaction Reporting - Searching, retrieving, and modifying existing transactions.

With one excpetion, all **x** files share the same base name as the equivalent **wp**. These base names are named after the category of functions they provide - wpauth provides the Auth functionality noted above and is used by xauth. The only excpetion is xchargebacks which uses wpauth as well.

Invoking the Demo App

The demo app is called snap.py and has the following invocation:

> python snap.py -lDEBUGLEVEL -p
>
> where:
> -l = log debugging information to logging.txt. DEBUGLEVEL can be one of: DEBUG, INFO, WARNING, ERROR, CRITICAL
> -p = use proxy. The proxy settings are contained in wptotal.py in httpProxy and httpsProxy.
> -s = doublesecretprobation mode. This is a specialized switch used purely for purpose of isolating the core data structures to ensure

> that they are in lign with any future Worldpay Total API changes. You should never need to use this switch, but I know you are going to go see what it does ;)

As it is currently designed, the demo app will simply go through each of the transactions listed in the API Docs. The app pulls test data from testdata.csv and generates API calls to complete the operation.

snap.py

This file provides the script for invoking the test calls. It makes calls to the doXXX functions contained within the X Files. Every available Worldpay Total function is called at least once and some are repeated to setup more complex interactions. Feel free to experiment by changing the calls and order in this file. Like the X Files that follow, it is meant to be replaced by your application.

X Files

The filenames beginning with the letter x are meant as teaching guides on how to use the core system. They should be replaced with your application files as needed. These files are all named accoring to their equivalent pages in API Docs.

The callable functions all begin with the prefix **do**. All other functions without this prefix are designed to support the doXXX calls and should not be called from your application,

> **Callable Functions:**
>
> **doAuth()** - xauth.py
> **doCharge()** - xauth.py
> **doVerify()** - xauth.py
> **doPriorAuthCapture()** - xauth.py
> **doVoid()** - xchargeback.py
> **doRefund()** - xchargeback.py
> **doCredit()** - xchargeback.py
> **doGetBatch()** - xsettlement.py
> **doGetBatchById()** - xsettlement.py
> **doCloseBatch()** - xsettlement.py
> **doCreateToken()** - xtokens.py
> **doCreateCustomer()** - xvault.py
> **doUpdateCustomer()** - xvault.py
> **doGetCustomer()** - xvault.py
> **doGetVaultAccount()** - xvault.py
> **doUpdatePaymentAccount()** - xvault.py
> **doDeletePaymentAccount()** - xvault.py
> **doCreateCustomerAndPayment** - xvault.py
> **doUpdateCustomerAndPayment()** - xvault.py
> **doRecurringPaymentPlan** - xrecurring.py
> **doInstallmentPaymentPlan** - xrecurring.py
> **doVariablePaymentPlan** - xrecurring.py
> **doGetPaymentPlan** - xrecurring.py
> **doSearchTransactions** - xtransactionreporting.py
> **doGetTransaction** - xtransactionreporting.py
> **doUpdateTransaction** - xtransactionreporting.py

WP Files

These files are the core system and designed to be incorporated into your application. They contain all of the classes used to communicate with the Worldpay Total REST API.

All of the WP objects communicate through discrete class attributes and methods. There is no need for dictionary or JSON marshaling as that is all handled by the core system.

Files that end in **object** are the *Request Objects*. These are the classes that define the data to be sent to the API. Classes contained in wpresponseobjects.py are the *Response Objects*. These are the classes that contain the data returned from the API calls.

Executing a Transaction

There are three steps to executing a transaction within SNAP.

1. Create and fill in a Request Object
2. Execute wpTransact()
3. Access results in a Response Object

Every transaction type has a Resquest and Response Object pair that it expects.

Request Objects

Request Objects are the main data structure you will use to send data in the transaction. Each transaction uses one of the following Request Object data structures to complete its operation.

Al Request Objects are defined in their equivalent *wpYYYobjects* file, where YYY is one of the functional categories listed earlier.

> **AuthorizationRequest**
> (used in doAuth, doCharge, doVerify, doVoid, doRefund, doCredit)
>
> **PriorAuthCaptureRequest**
> (used in doPriorAuthCapture)
>
> **BatchRequest**
> (used in doGetbatch, doGetBatchById, doCloseBatch)
>
> **TokenRequest**
> (used in doCreateToken)
>
> **CustomerRequest**
> (used in doCreateCustomer, doUpdateCustomer, doGetCustomer)
>
> **VaultAccountRequest**
> (used in doCreateVaultAccount, doGetVaultAccount, doUpdatePaymentAccount, doDeletePaymentAccount)
>
> **CustomerAndPaymentRequest**
> (used in doCreateCustomerAndPayment, doUpdateCustomerAndPayment)
>
> **RecurringPaymentPlanRequest**
> (used in doRecurringPaymentPlan)

> **InstallmentPaymentPlanRequest**
> (used in doInstallmentPaymentPlan)
>
> **VariablePaymentPlanRequest**
> (used in doVariablePaymentPlan)
>
> **GetPaymentPlanRequest**
> (used in doGetPaymentPlan)
>
> **SearchTransactionsRequest**
> (used in doSearchTransaction)
>
> **UpdateTransactionRequest**
> (used in doUpdateTransaction)

Many times you will only need to fill in the variables in the Request Object. Other times you may have to fill in other data structures and attach them to the Request Object. If there are classes that require additional classes to be attached, they will support a method called attachXXX, where xxx is the object to attach.

For instance, the AuthorizationRequest object contains another object called Card. You would then create a Card object, fill it in, and then attach it to the AuthorizationRequest object.

```
card = Card()
card.number = 1111 2222 3333 4444
card.cvv = 999
card.expirationDate = 12/20
ar = AuthorizationRequest()
ar.attachCard(card)
```

As per convention, all class names follow the Upper Camel Case format while all attributes and methods follow Camel Case notation. Also note that any fields not filled in will not be sent in the transaction.

Response Objects

Response Objects are classes that contain the response from an API call. They are only created after a dictionary-based response is received from wpTransact(). This is because the constructor of a Response Object automatically deserializes the response from wpTransact() and places data into the proper discrete data attributes. No parsing of dictionary formats is necessary. More on this later.

All Response Object definitions are contained in *responseparameters.py*.

> **AuthResponseParameters**
> (used in doAuth, doCharge, doVerify, doPriorAuthCapture, doVoid, doRefund, doCredit)
>
> **BatchResponseParameters**
> (used in doGetBatch, doGetBatchById, doCloseBatch)
>
> **TokenResponseParameters**
> (used in doCreateToken)
>
> **CustomerResponseParameters**
> (used in doCreateCustomer, doUpdateCustomer, doGetCustomer)
>
> **PaymentAccountResponseParameters**

(used in doCreateVaultAccount, doGetVaultAccount, doUpdatePaymentAccount, doDeletePaymentAccount)

CustomerAndPaymentResponseParameters
(used in doCreateCustomerAndPayment, doUpdateCustomerAndPayment)

RecurringPaymentPlanResponseParameters
(used in doRecurringPaymentPlan)

InstallmentPaymentPlanResponseParameters
(used in doInstallmentPaymentPlan)

VariablePaymentPlanResponseParameters
(used in doVariablePaymentPlan)

GetPaymentPlanResponseParameters
(used in doGetPaymentPlan)

TransactionReportingResponseParameters
(used in doSearchTransactions, doGetTransaction, doUpdateTransaction)

ResponseParameters
(this is the base class of all of the above)

Note that all of the Response Objects are a subclass of ResponseParameters. Therefore make sure you look at the base class to see all data attributes and methods.

wpTransact()

Once you have chosen the correct Request and Response Objects for the type of transaction you want to execute, it's time to make it happen by calling wpTransact().

```
response = wpTransact(operation, payload, p1, p2)

where:
    operation - string of the operation name as defined in wptotal.py (see below)
    payload - the serialized Request Object
    p1 - only needed for a handful of API calls.
    p2 - needed for even fewer API calls.
```

The *operation* parameter tells wpTransact which API call to make. You will pass it a string that must match the text string defined below. The operation name is case-insensitive. Here is a list of all operation types:

Operations
Authorize
PriorAuthCapture
Charge
Credit
Verify
CloseBatch
GetBatchById
GetBatch
Refund

> Void
> CreateToken
> CreateCustomer
> GetCustomer
> UpdateCustomer
> CreateVaultAccount
> GetVaultAccount
> UpdatePaymentAccount
> DeletePaymentAccount
> CreateCustomerAndPayment
> UpdateCustomerAndPayment
> CreateRecurringPaymentPlan
> UpdateRecurringPaymentPlan
> CreateInstallmentPaymentPlan
> UpdateInstallmentPaymentPlan
> CreateVariablePaymentPlan
> UpdateVariablePaymentPlan
> GetPaymentPlan
> SearchTransactions
> GetTransaction
> UpdateTransaction

Payload is a dictionary based representation of the Request Object. Technically, the wpTransact() function takes input and returns results using dictionary objects. However, SNAP does the heavy lifting of converting the dictionary structures into discrete Request and Response class objects with associated attributes and methods. You never have to parse a dictionary - simply access all data in dot notation. This serialization and de-serialization process is built into the Request and Response Objects.

A Request Object is serialized in the wpTransact call through the 'serialize' method. It is recommended this is done within the wpTransact call as follows:

```
    wpTransact(operation, requestObject.serialize())
```

A Response Object, on the other hand, is automatically derserialized in its constructor. The result of wpTransact is a dictionary object. By then creating a new Response Object and assigning it the dictionary response from wpTransact, the dat is automatically de-serialized.

```
    ar = AuthorizationRequest  # Request Object
    # (fill in attributes of ar)
    responseDictionary = wpTransact('Charge', ar.serialize())
    rp = AuthResponseParameters(responseDictionary)
    # extract results from rp attributes
```

p1 is required in some types of operations. You might pass a customer or batch id, for example.

> **Operations That Require p1**
> GetBatchById
> GetCustomer
> UpdateCustomer
> CreateVaultAccount
> UpdateCustomerAndPayment
> CreateRecurringPaymentPlan

> CreateVariablePaymentPlan
> GetTransaction
> Updatetransaction

Some operations require two variables. In this case, pass p1 and p2.

> **Operations That Require p2**
> GetVaultAccount
> UpdatePaymentAccount
> DeletePaymentAccount
> UpdateRecurringPaymentPlan
> UpdateInstallmentPlan
> UpdatevariablePaymentPlan
> GetPaymentPlan

Putting It All Together

That's all there is. Create a Request Object, fill it in, call wpTrasact(), and get your results in a Response Object. Use the following table as a reference in constructing your functions.

Operation	Request Object	Response Object	P1	P2
Authorize	AuthorizationRequest	AuthResponseParameters		
PriorAuthCapture	PriorAuthorCaptureRequest	PriorAuthResponseParameters		
Charge	AuthorizationRequest	AuthResponseParameters		
Credit	AuthorizationRequest	AuthResponseParameters		
Verify	AuthorizationRequest	AuthResponseParameters		
CloseBatch	Batch Request	BatchresponseParameters		
GetBatchById	Batch Request	BatchresponseParameters	batchId	
GetBatch	Batch Request	BatchresponseParameters		
Refund	AuthorizationRequest	AuthResponseParameters		
Void	AuthorizationRequest	AuthResponseParameters		
CreateToken	TokenRequest	TokenResponseParameters		
CreateCustomer	CustomerRequest	CustomerResponseParameters		
GetCustomer	CustomerRequest	CustomerResponseParameters	custId	
UpdateCustomer	CustomerRequest	CustomerResponseParameters	custId	
CreateVaultAccount	VaultAccountRequest	PaymentAccountResponseParameters	custId	
GetVaultAccount	VaultAccountRequest	PaymentAccountResponseParameters	custIdd	payId
UpdatePaymentAccount	VaultAccountRequest	PaymentAccountResponseParameters	custId	payId

DeletePaymentAccount	VaultAccountRequest	PaymentAccountResponseParameters	custId	payId
CreateCustomerAndPayment	CustomerAndPaymentRequest	CustomerAndPaymentResponseParameters		
UpdateCustomerAndPayment	CustomerAndPaymentRequest	CustomerAndPaymentResponseParameters	custId	
CreateRecurringPaymentPlan	RecurringPaymentPlanRequest	RecurringPaymentPlanResponseParameters	custId	
UpdateRecurringPaymentPlan	RecurringPaymentPlanRequest	RecurringPaymentPlanResponseParameters	custId	payId
CreateInstallmentPaymentPlan	InstallmentPaymentPlanRequest	InstallmentPaymentPlanResponseParameters		
UpdateInstallmentPaymentPlan	InstallmentPaymentPlanRequest	InstallmentPaymentPlanResponseParameters	custId	payId
CreateVariablePaymentPlan	VariablePaymentPlanRequest	VariablePaymentPlanResponseParameters	custId	
UpdateVariablePaymentPlan	VariablePaymentPlanRequest	VariablePaymentPlanResponseParameters	custId	payId
GetPaymentPlan	GetPaymentPlanRequest	GetPaymentPlanResponseParameters		
SearchTransactions	SearchTransactionsRequest	TransactionReportingResponseParameters		
GetTransaction	(none)	TransactionReportingResponseParameters	transId	
UpdateTransaction	UpdateTransactionRequest	TransactionReportingResponseParameters	transId	

Error Management

This package utilizes exceptions to handle all error conditions. All of the exceptions are listed in wpexceptions.py. Each of the exceptions will contain an error message embedded further clarifying why the exception was raised.

Settings

All pre-defined settings are contained in the class WPTotal in wptotal.py. This is where you will need to place your merchantId, metchantKey, publicKey (if using tokenization) appVersion, proxy info and other sundry information.

Debugging

A log file can be generated at run-time by including the -l switch and then specifying how much information you want to include. Each layer includes the previous higher levels messages as well. For instance, if you specify -lINFO, it will include WARNING, ERROR, and CRITICAL messages as well, but not DEBUG messages.

- DEBUG - very verbose
- INFO - information that demonstrates state flows and major data structures
- WARNING - warning messages
- ERROR - general error states that must be handled
- CRITICAL - program stopping events.

```
python snap.py -LINFO
```

Documentation

The code has been designed so that data attribute names match the names provided in API Docs.