PyAvaTax Documentation

Release 0.1

John Obelenus, Active Frequency

May 01, 2015

Contents

CHAPTER 1

What is PyAvaTax?

As of Sept 2012 US internet retailers are required to pay sales tax in all the states they do business. Avalara offers a fully featured web-based service to report your transactions, return your sales tax, and store all the information until you need to report it.

Avalara is a US-only service, and thus all amounts passing through their system, and this api, are assumed to be US Dollars (USD)

We developed PyAvaTax as a Python client library for easily integrating with Avalara's RESTful AvaTax API Service to report your transactions.

PyAvaTax **does not require Django**, though if you are using a Django system we have some admin-based goodies for you to check out! If you're running this on a system with Django installed (e.g. we can find Django in the import path) we will attempt to integrate with it. If you don't want this default behavior, please see the Django section on how to prevent it.

AvaTax expects a JSON (or XML) POST to their tax/get/ URI, like this:

```
{
    "DocDate": "2012-10-24",
    "CompanyCode": "FooBar",
    "CustomerCode": "email@example.com",
    "DocCode": "1001",
    "DocType": "SalesOrder",
    "Addresses":
    [
        {
            "AddressCode": "1",
            "Line1": "435 Ericksen Avenue Northeast",
            "Line2": "#250",
            "PostalCode": "98110"
        },
        {
            "AddressCode": "2",
            "Line1": "7562 Kearney St.",
            "PostalCode": "80022-1336"
        }
    ],
    "Lines":
    [
        {
            "LineNo": "1",
            "DestinationCode": "2",
            "OriginCode": "1",
            "Qty": 1,
```

```
            "Amount": "100"
        }
    ]
}
```

Our library, accepts your data in a variety of ways. You instantiate the API like so

```
api = API(AVALARA_ACCOUNT_NUMBER, AVALARA_LICENSE_KEY, AVALARA_COMPANY_CODE)
```

Then, you can perform an action (e.g. "Post Tax"), by passing in a data dictionary. We will parse it, validate it, handle the HTTP layer for you, and return a response object to you.

```
tax_response = api.post_tax(dictionary_data)
print tax_response.TotalTax  # this is unicode
>>> 0.86
```

That returned object will have all the response data from AvaTax easily accessible by dot-notation.

Or, you can use the library to construct objects from kwargs

```
api = API(AVALARA_ACCOUNT_NUMBER, AVALARA_LICENSE_KEY, AVALARA_COMPANY_CODE)
doc = Document.new_sales_order(DocCode='1001', DocDate=datetime.date.today(), CustomerCode='email@exa
doc.add_from_address(Line1="435 Ericksen Avenue Northeast", Line2="#250", PostalCode="98110")
doc.add_to_address(Line1="7562 Kearney St.", PostalCode="80022-1336")
doc.add_line(Amount=10.00)
response = api.post_tax(doc)
```

We have a full-fledged introduction, from installation, logging, making requests, and handling responses, with a full example in the next topic: Basics

If you have any issues, improvements, requests, or bugs please use Github

Contents:

1.1 The Basics

You can rely on our integration to validate what information you're providing. We handle the simple case of shipping and line numbers, so you don't have to think about AvaTax's abstractions and data structures. If you don't add line numbers to your items, we'll add them for you. If you use `add_to_address` and `add_from_address` you can ignore the `AddressCode`, ``DestinationCode`, and `OriginCode` attributes as well. See the section below about creating a document manually for steps on how to do this.

Of course, for more complicated interactions all the AvaTax flexibility is at your disposal.

1.1.1 Installing the Project

If you are using pip (we *highly* recommend using it for managing your Python packages), this is the installation command:

```
pip install pyavatax
```

If you are using this project via its source files you will find the dependencies of the project in the provided requirements.txt file. We use *py.test* for testing, but you don't need to install that to use the library.

```
pip install -r requirements.txt
```

If you are unfamiliar with pip/pypi you should check out the short wiki entry page, and then pypi.org

1.1.2 Copy & Paste

If you're looking for something to copy and paste into your python code base and play with, try this block of code. However, I do ask that you continue to read this basics section (at least) to get a better idea of exactly what is going on.

```
import pyavatax
api = pyavatax.API(YOUR_AVALARA_ACCOUNT_NUMBER, YOUR_AVALARA_LICENSE_KEY, YOUR_AVALARA_COMPANY_CODE,
data = {
    "DocDate": "2012-06-13",
    "CompanyCode": YOUR_AVALARA_COMPANY_CODE,
    "CustomerCode": "YourClientsCustomerCode",
    "DocCode": "20120613-1",
    "DocType": "SalesOrder",
    "Addresses":
    [
        {
            "AddressCode": "1",
            "Line1": "435 Ericksen Avenue Northeast",
            "Line2": "#250",
            "City": "Bainbridge Island",
            "Region": "WA",
            "PostalCode": "98110",
            "Country": "US",
        },
        {
            "AddressCode": "2",
            "Line1": "7562 Kearney St.",
            "City": "Commerce City",
            "Region": "CO",
            "PostalCode": "80022-1336",
            "Country": "US",
        },
    ],
    "Lines":
    [
        {
            "LineNo": "1",
            "DestinationCode": "2",
            "OriginCode": "1",
            "ItemCode": "AvaDocs",
            "Description": "Box of Avalara Documentation",
            "Qty": 1,
            "Amount": "100",
        },
    ],
}
try:
    tax = api.post_tax(data)
except pyavatax.AvalaraServerNotReachableException:
    raise Exception('Avalara is currently down')
else:  # try else runs whenever there is no exception
    if tax.is_success is True:
        tax.total_tax  # has your total amount of tax for this transaction
    else:
        raise Exception(tax.error)  # Avalara found a problem with your data
```

1.1.3 Instantiating the API

Looks like:

```
import pyavatax
api = pyavatax.API(YOUR_ACCOUNT_NUMBER, YOUR_LICENSE_NUMBER, YOUR_COMPANY_CODE, live=True/False)
```

Once you have an account with AvaTax their dashboard page contains the account number and license number. You can choose a meaningful company code. When live is *False*, the request will be sent to Avalara's test environment. When it is is *True* it will be sent to the production environment.

1.1.4 Creating a Document From Data

Looks like:

```
import pyavatax
doc = Document.from_data(dictionary_data)
```

The `dictionary_data` will be validated against the formatting expected by AvaTax. An `AvalaraException` will be raised in the cases it does not validate.

For all the API calls you can pass a dictionary, or an object:

```
doc = Document.from_data(dictionary_data)
tax = api.post_tax(doc)
# this line performs the same operation as the above two
tax = api.post_tax(data_dictionary)
```

1.1.5 Making an API call

Here are a few example calls. You can find Avalara's documentation on each of these calls and the parameteres they expect here: Validate Address, Get Tax, Post Tax, Cancel Tax

```
response = api.validate_address(address)
lat = 47.627935
lng = -122.51702
response = api.get_tax(lat, lng, doc)
# in lieu of making a whole document, you can alternatively pass the amount to be taxed
response = api.get_tax(lat, lng, None, sale_amount=100.00)
response = api.post_tax(doc)
response = api.post_tax(doc, commit=True)
response = api.cancel_tax(doc)
```

Using the `commit=True` on the post_tax call is a shortcut, it is the equivalent of doing this:

```
doc.update({'Commit': True})
api.post_tax(doc)
```

However, it will also perform an additional check. Submitting a `SalesOrder` (any `XXXXXOrder`) to AvaTax with `Commit=True` won't result in a saved and committed document. It is the wrong type. It needs to be `SalesInvoice` (or `XXXXXXInvoice`). So if we find an `XXXXXOrder` and you pass `commit=True` we will automatically update the type for you.

So far you have noticed we are always using `SalesOrder` and `SalesInvoice` in our examples. This is for when you are selling products to customers, the most basic example. Other document types are `ReturnOrder`, `ReturnInvoice`, `PurchaseOrder`, `PurchaseInvoice`, `InventoryTransferOrder`,

and `InventoryTransferInvoice`. They are used when a customer is returning an item, when you're purchasing items, and when you're transfering inventory.

As an added convenience the response objects from `post_tax` and `get_tax` have a `total_tax` property:

```
response = api.get_tax(lat=47.627935, lng=-122.51702, doc)
response.Tax  # is the attribute AvaTax returns
response.total_tax  # maps to Tax
response = api.post_tax(doc)
response.TotalTax  # is the attribute AvaTax returns, note it is not consistent with the other name
response.total_tax  # maps to TotalTax
```

1.1.6 Creating a Document Manually

Looks like:

```
import pyavatax
doc = pyavatax.Document(**kwargs)
address = pyavatax.Address(**kwargs)
line_item = pyavatax.Line(**kwargs)
```

Use the `kwargs` parameter to send all the relevant AvaTax fields into the document. Any keys that are not AvaTax fields will throw an `AvalaraException`. All the keys **do use AvaTax's camel-case notation**.

```
doc.add_to_address(address)
doc.add_from_address(another_address)
doc.add_line(line_item)
```

For simple shipping cases you can use the helper functions `add_to_address` and `add_from_address`. These will manually add the AvaTax `OriginCode` and `DestinationCode` to the corresponding `AddressCode`. If your shipping scenario isn't simple, we cannot assume what you're doing - so you will have to input that data onto the objects yourself. Here is an exaggerated example to make this use case as clear as possible:

```
address.update({'AddressCode': 3})  # updating address dictionary with address code
another_address.update({'AddressCode': 2})
a_third_address.update({'AddressCode': 1})
line.update({'OriginCode': 1, 'DestinationCode': 3})
another_line.update({'OriginCode': 2, 'DestinationCode': 3})
doc.add_address(address)
doc.add_address(another_address)
doc.add_address(a_third_address)
doc.add_line(line)
doc.add_line(another_line)
```

Alternatively, if you don't have to have address objects running around for you to modify at a future point before adding to them to a document, you can do it all in one step (like you saw on the documentation index page)

```
doc.add_from_address(Line1="435 Ericksen Avenue Northeast", Line2="#250", PostalCode="98110")
doc.add_to_address(**kwargs)
```

1.1.7 Handling a response

Looks like:

```
try:
    response = api.get_tax(lat=47.627935, lng=-122.51702, doc)
except AvalaraServerNotReachableException:
```

```
    raise ApplicationException('Avalara is currently down')
else:
    if response.is_success is True:
        return response.Tax
    else:
        raise ApplicationException(response.error)
```

The JSON response from AvaTax is automatically parsed onto the response object. In the case of a "GetTax" call the attribute 'Tax' is the total taxable amount for your transaction.

If the response is not successful, the `error` attribute is a list of tuples. The first item is either the offending field (if there is one) or the AvaTax class which threw the error. The second item is a human readable description of the error provided by AvaTax.

Should you need access to the actual response or request, the `response` attribute has the `Request` object which has `headers`, `full_url`, `body`, and other parameters. The `response` attribute also has a `request` attribute which contains information about the raw request. If you need more details check out the AvaTax documentation.

You should use a `try: except:` block to catch `AvalaraServerNotReachableException` in the case your network, or Avalara's network has connectivity problems.

Since the `Request` library sits on top of urllib you may not get the **exact data/headers being transmitted**. To account for this you can pass a `proxies` dictionary to the `API` constructor. You can use this setting to setup Charles Proxy, an excellent and free GUI application for sniffing the exact data being sent over the wire. You can see more detail about Request and proxies here:

1.1.8 Logging

PyAvaTax uses standard Python logging, with a logger called `pyavatax.api`. All HTTP requests are logged at the `INFO` level. All changes that our API makes to your Document objects are logged at the `DEBUG` level. All 500 errors, or HTTP Errors (timeouts, unreachable, etc.) are logged to the `ERROR` level.

You can pass your own logger, should you so choose, like so:

```
import pyavatax.base.AvalaraLogging
AvalaraLogging.set_logger(my_custom_logger)
# subsequent api calls will use the custom logger
response = api.get_tax(lat=47.627935, lng=-122.51702, doc)
```

1.2 API Object

class `pyavatax.api.`**API**(*account_number*, *license_key*, *company_code*, *live=False*, *logger=None*, *recorder=None*, ***kwargs*)

DEVELOPMENT_HOST = 'development.avalara.net'

PRODUCTION_HOST = 'rest.avalara.net'

VERSION = '1.0'

cancel_tax(*doc*, *reason=None*, *doc_id=None*)
Performs a HTTP POST to tax/cancel/

get_tax(*lat*, *lng*, *doc*, *sale_amount=None*)
Performs a HTTP GET to tax/get/

post_tax(*doc*, *commit=False*)
Performs a HTTP POST to tax/get/ If commit=True we will update the document's Commit flag to True, and we will check the document type to make sure it is capable of being Commited. XXXXXOrder is not capable of being commited. We will change it to XXXXXXXInvoice, which is capable of being committed

validate_address(*address*)
Performs a HTTP GET to address/validate/

1.3 Avalara Objects

1.3.1 Avalara Document

class pyavatax.base.**Document**(*logger=None*, **args*, ***kwargs*)
Represents the Avalara Document

CANCEL_ADJUSTMENT_CANCELED = 'AdjustmentCanceled'

CANCEL_CODES = ('PostFailed', 'DocDeleted', 'DocVoided', 'AdjustmentCanceled')

CANCEL_DOC_DELETED = 'DocDeleted'

CANCEL_DOC_VOIDED = 'DocVoided'

CANCEL_POST_FAILED = 'PostFailed'

DOC_TYPES = ('SalesOrder', 'SalesInvoice', 'ReturnOrder', 'ReturnInvoice', 'PurchaseOrder', 'PurchaseInvoice', 'Inver

DOC_TYPE_INVENTORY_INVOICE = 'InventoryTransferInvoice'

DOC_TYPE_INVENTORY_ORDER = 'InventoryTransferOrder'

DOC_TYPE_PURCHASE_INVOICE = 'PurchaseInvoice'

DOC_TYPE_PURCHASE_ORDER = 'PurchaseOrder'

DOC_TYPE_RETURN_INVOICE = 'ReturnInvoice'

DOC_TYPE_RETURN_ORDER = 'ReturnOrder'

DOC_TYPE_SALE_INVOICE = 'SalesInvoice'

DOC_TYPE_SALE_ORDER = 'SalesOrder'

static _clean_date(*date*)

static _clean_float(*f*)

static _clean_int(*i*)

_contains = ['Lines', 'Addresses']

_fields = ['DocType', 'DocId', 'DocCode', 'DocDate', 'CompanyCode', 'CustomerCode', 'Discount', 'Commit', 'Custo

_has = ['DetailLevel', 'TaxOverride']

add_address(*address=None*, ***kwargs*)
Adds an Address instance to this document. Nothing about the address will be changed, you are entirely responsible for it

add_from_address(*address=None*, ***kwargs*)
Only use this function when performing a simple shipping operation. The default from address code will be used for this address

add_line(*line=None*, ***kwargs*)
Adds a Line instance to this document. Will provide a LineNo if you do not

add_override(*override=None*, ***kwargs*)
Adds a tax override instance to this document

add_to_address(*address=None*, ***kwargs*)
Only use this function when performing a simple shipping operation. The default to address code will be used for this address

clean_Commit()

clean_Discount()

clean_DocDate()

clean_DocType()

clean_PaymentDate()

static from_data(*data*)

static new_inventory_invoice(**args*, ***kwargs*)

static new_inventory_order(**args*, ***kwargs*)

static new_purchase_invoice(**args*, ***kwargs*)

static new_purchase_order(**args*, ***kwargs*)

static new_return_invoice(**args*, ***kwargs*)

static new_return_order(**args*, ***kwargs*)

static new_sales_invoice(**args*, ***kwargs*)

static new_sales_order(**args*, ***kwargs*)

set_detail_level(*detail_level=None*, ***kwargs*)
Add a DetailLevel instance to this Avalara document

total
Helper representing the line items total amount for tax. Used in GetTax call

update_doc_code_from_response(*post_tax_response*)
Sets the DocCode on the Document based on the response if Document does not have a DocCode

validate()
Ensures we have addresses and line items. Then calls validate_codes

validate_codes()
Look through line items making sure that origin and destination codes are set set defaults if they exist, raise exception if we are missing something

1.3.2 Document static factory methods

The `new_xxxxx_order` and `new_xxxxx_invoice` calls are static factory functions on the Document class to create a corresponding Document with the intended DocType

1.3.3 Avalara Line

class `pyavatax.base.`**Line**(**args*, ***kwargs*)
Represents an Avalara Line

_fields = ['LineNo', 'DestinationCode', 'OriginCode', 'Qty', 'Amount', 'ItemCode', 'TaxCode', 'CustomerUsageType'

clean_Amount()

clean_ItemCode()

clean_Qty()

static **from_data**(*data*)

1.3.4 Avalara Address

class pyavatax.base.**Address**(*allow_new_fields=False*, **args*, ***kwargs*)

Represents an Avalara Address

DEFAULT_FROM_ADDRESS_CODE = '1'

DEFAULT_TO_ADDRESS_CODE = '2'

_fields = ['AddressCode', 'Line1', 'Line2', 'Line3', 'PostalCode', 'Region', 'City', 'TaxRegionId', 'Country', 'Address

describe_address_type
Returns human-readable description

describe_carrier_route
Returns human-readable description

describe_fips_code
Returns human-readable description

describe_post_net
Returns human-readable description

static **from_data**(*data*)

1.3.5 Avalara TaxOverride

class pyavatax.base.**TaxOverride**(*allow_new_fields=False*, **args*, ***kwargs*)

Represents an Avalara TaxOverride

OVERRIDE_AMOUNT = 'TaxAmount'

OVERRIDE_DATE = 'TaxDate'

OVERRIDE_EXEMPT = 'Exemption'

OVERRIDE_NONE = 'None'

OVERRIDE_TYPES = ('None', 'TaxAmount', 'TaxDate', 'Exemption')

_fields = ['TaxOverrideType', 'TaxAmount', 'TaxDate', 'Reason']

clean_Reason()

clean_TaxAmount()

clean_TaxDate()

clean_TaxOverrideType()

clean_me()

static **from_data**(*data*)

1.4 Avalara Response Representations

class pyavatax.base.**BaseResponse**(*response*, **args*, ***kwargs*)
Common functionality for handling Avalara server responses

error
Returns a list of tuples. The first position in the tuple is either the offending field that threw an error, or the class in the Avalara system that threw it. The second position is a human-readable message from Avalara

is_success
Returns whether or not the response was successful

class pyavatax.base.**ErrorResponse**(*response*, **args*, ***kwargs*)
Common error case functionality from a 500 error

error
Returns a list of tuples. The first position in the tuple is either the offending field that threw an error, or the class in the Avalara system that threw it. The second position is a human-readable message from Avalara

is_success
Returns whether or not the response was successful

1.4.1 GetTax Response

class pyavatax.api.**GetTaxResponse**(*response*, **args*, ***kwargs*)

_contains = ['TaxDetails']

_fields = ['Rate', 'Tax', 'ResultCode']

total_tax

class pyavatax.base.**TaxDetails**(*allow_new_fields=False*, **args*, ***kwargs*)
Represents TaxDetails response from Avalara

_fields = ['Country', 'Region', 'JurisType', 'JurisCode', 'Taxable', 'Rate', 'Tax', 'JurisName', 'TaxName']

1.4.2 PostTax Response

class pyavatax.api.**PostTaxResponse**(*response*, **args*, ***kwargs*)

_contains = ['TaxLines', 'TaxDetails', 'TaxAddresses']

_fields = ['DocCode', 'DocId', 'DocDate', 'Timestamp', 'TotalAmount', 'TotalDiscount', 'TotalExemption', 'TotalTaxa

total_tax

class pyavatax.base.**TaxLines**(*allow_new_fields=False*, **args*, ***kwargs*)
Represents TaxLines response from Avalara

_contains = ['TaxDetails']

_fields = ['LineNo', 'TaxCode', 'BoundaryLevel', 'Taxability', 'Taxable', 'Rate', 'Tax', 'Discount', 'TaxCalculated', 'E

class pyavatax.base.**TaxDetails**(*allow_new_fields=False*, **args*, ***kwargs*)
Represents TaxDetails response from Avalara

_fields = ['Country', 'Region', 'JurisType', 'JurisCode', 'Taxable', 'Rate', 'Tax', 'JurisName', 'TaxName']

class pyavatax.base.**TaxAddresses**(*allow_new_fields=False*, **args*, ***kwargs*)
Represents TaxAddress response from Avalara

_contains = ['TaxDetails']

_fields = ['Address', 'AddressCode', 'Latitude', 'Longitude', 'City', 'Country', 'PostalCode', 'Region', 'TaxRegionId',

1.4.3 CancelTax Response

class pyavatax.api.**CancelTaxResponse**(*response*, **args*, ***kwargs*)

_details

_has = ['CancelTaxResult']

error
Returns a list of tuples. The first position in the tuple is either the offending field that threw an error, or the class in the Avalara system that threw it. The second position is a human-readable message from Avalara. Avalara bungled this response, it is formatted differently than every other response

is_success
Returns whether or not the response was successful. Avalara bungled this response, it is formatted differently than every other response

class pyavatax.base.**CancelTaxResult**(*allow_new_fields=False*, **args*, ***kwargs*)
Represents CancelTaxResult response from Avalara

_contains = ['Messages']

_fields = ['DocId', 'TransactionId', 'ResultCode']

1.4.4 ValidateAddress Response

class pyavatax.api.**ValidateAddressResponse**(*response*, **args*, ***kwargs*)

_fields = ['ResultCode']

_has = ['Address']

class pyavatax.base.**Address**(*allow_new_fields=False*, **args*, ***kwargs*)
Represents an Avalara Address

DEFAULT_FROM_ADDRESS_CODE = '1'

DEFAULT_TO_ADDRESS_CODE = '2'

_fields = ['AddressCode', 'Line1', 'Line2', 'Line3', 'PostalCode', 'Region', 'City', 'TaxRegionId', 'Country', 'Address

describe_address_type
Returns human-readable description

describe_carrier_route
Returns human-readable description

describe_fips_code
Returns human-readable description

describe_post_net
Returns human-readable description

static from_data(*data*)

1.5 Exceptions

exception pyavatax.base.**AvalaraException**(**args*, ***kwargs*)
Raised when operating unsuccessfully with document, address, line, etc objects

CODE_BAD_ADDRESS = 201

CODE_BAD_ARGS = 100

CODE_BAD_BOOL = 305

CODE_BAD_CANCEL = 103

CODE_BAD_DATE = 303

CODE_BAD_DEST = 307

CODE_BAD_DETAIL = 202

CODE_BAD_DOC = 101

CODE_BAD_DOCTYPE = 302

CODE_BAD_FLOAT = 304

CODE_BAD_LATLNG = 102

CODE_BAD_LINE = 203

CODE_BAD_ORIGIN = 306

CODE_BAD_OTYPE = 309

CODE_BAD_OVERRIDE = 204

CODE_HAS_FROM = 104

CODE_HAS_TO = 105

CODE_INVALID_FIELD = 301

CODE_REQD = 50

CODE_TOO_LONG = 308

exception pyavatax.base.**AvalaraTypeException**(**args*, ***kwargs*)
Raised when passed wrongly typed data, or a non-Avalara object when one is expected

exception pyavatax.base.**AvalaraValidationException**(**args*, ***kwargs*)
Raised when object data does not pass validation

exception pyavatax.base.**AvalaraServerException**(*response*, **args*, ***kwargs*)
Used internally to handle 500 and other server error responses

errors
Will return an ErrorResponse details property, or the raw text server response

full_request_as_string
Returns all the info we have about the request and response

exception pyavatax.base.**AvalaraServerDetailException**(*response*, **args*, ***kwargs*)
Useful for seeing more detail through the tester and logs We always throw this exception, though you may catch AvalaraServerException if you don't care to see the details in the __str__

exception pyavatax.base.**AvalaraServerNotReachableException**(*request_exception*, **args*, ***kwargs*)

Raised when the AvaTax service is unreachable for any reason and no response is received

1.6 PyAvaTax features for Django

If you are integrating PyAvaTax into a Django environment you are in luck. In addition to the standard Python logging I have implemented an AvaTaxRecord model in this project. If you put *pyavatax* into your installed apps and run *syncdb*, you'll find a new Admin entry.

Note: We are currently supporting Django version >= 1.6. The only change is the deprecation of *get_query_set*, replaced with *get_queryset*.

This way your clients can see which records failed to make it into the Avalara system, since they don't usually have access to, or care to access, the logs.

If you really don't want any integration with Django you can turn it off by setting *NO_PYAVATAX_INTEGRATION = True* in your settings file.

You can also get at these records:

```
import pyavatax.models AvaTaxRecord
AvaTaxRecord.failures.all()
```

After a Document which has failed runs successfully you'll see it leave that list. And you'll see it pop up over here:

```
AvaTaxRecord.successes.all()
```

Note: if a Document never failed it is never put into either of these lists.

1.6.1 Your Own Recorder

If you want to create your own recorder to perform some special action when a success or failure occurs, the interface looks like this:

```
class YourSpecialRecorder(object):

    @staticmethod
    def failure(doc, response):
        pass

    @staticmethod
    def success(doc):
        pass
```

To have the API use this recorder just pass the class (or instance, there is no need for them to actually be static methods) as the `recorder` keyword-arg to the API instantiation.

Note: `success` will always be called in the event of a success, even if a prior failure never occurred.

1.7 Advanced

1.7.1 Running the Tests

If you're working with the source code and want to run our tests, you can run the test suite (we are using pytest).

There are some tests specifically for the Django features. If you're not running in a Django environment, those specific tests will return with an expected failure (they will show as `passed` because they were expected to fail)

The test script uses a `settings_local.py` secrets file that isn't included in this package. We've included a `settings_local.py.example` file that you can copy into `settings_local.py` and update with your credentials.

If you have a Django environment you can run `manage.py shell` locally and then this:

```
>>> import pytest
>>> pytest.main('path/to/pyavatax/test_avalara.py')
```

Alternatively, you can just do:

```
$ py.test
```

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*
- *search*

p

Symbols

A

B

C

D

E

F

G

I

L

N

O

P

S

T

U

V