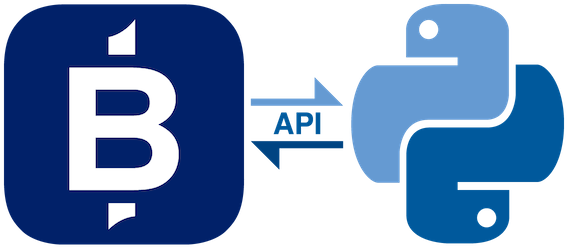
BidFX API Python

Release 1.1.0

Mar 09, 2022

Contents



The BidFX Python API is a price API that connects to the BidFX trading platform to subscribe to realtime pricing. The API supports the following features:

- FX streaming executable prices (RFS)
- FX request for quote (RFQ)

CHAPTER 1

Contents

1.1 About BidFX

BidFX is the market-leading provider of electronic trading solutions for the global foreign exchange marketplace. BidFX has addressed the challenges of the FX market by introducing a complete suite of negotiation protocols – which include: auto-routing, streaming, request-for-quote, voice, algo-trading and best execution – via a cloud-based SaaS trading platform. BidFX offer clients access to a cutting edge, broker-neutral, Execution Management System (EMS) backed by a hub to all major bank's algo suites. You can read about all BidFX products on the main BidFX Website.

1.1.1 BidFX APIs

BidFX clients access the trading platform via a dedicated User Interface (UI) either on their desktop PC, web browser or mobile device.

Public APIs provide a secondary means of accessing the trading platform that can either supplement the UI or replace it entirely in some use cases, including: systematic trading, OMS integration and market intelligence. BidFX place significant emphasis on API support and therefore provide a suite of APIs for different high-level programming languages and common protocols.

You can read about the complete BidFX API range, and their different capabilities, at BidFX API Overview.

1.1.2 Python API

This document describes the *BidFX Public API for Python*. The Python API is written pure Python and is compatible with `Python 3.6` and above. All of the code examples below are presented in Python.

We use the nomenclature *Public* to indicated that this API is designed and maintained for public use by BidFX clients. Being Public implies a degree of support, API stability and future compatibility appropriate to client usage.

1.2 FX liquidity

BidFX are connected to all the major tier-1 banks and FX liquidity providers (LP). LPs publish tradable FX price/quotes into the BidFX platform using the FIX protocol. The quotes from LPs are firm prices with an associated *price-ID* that needs to be attached to any order placed against the quote. The BidFX platform consumes billions for FIX messages per day. We provision high-bandwidth, cross-connect circuits in the main global data centres for this purpose. BidFX connect to banks where they host their price engines, in particular in:

- London (LD4),
- New York (NY4) and
- Tokyo (TY3).

1.2.1 Last look

FX quotes are short-lived and LPs reserve the right of **last look**. A quote usually is good for no more than a few hundred milliseconds. Network latency between the client application and the LP is therefore a significant consideration if we want to avoid order rejections.

If clients intend to trade directly against price-IDs, then it is recommended that they run their application very close to the source of liquidity and cross-connect within the same data centre. Alternatively, clients may route their orders to the BidFX *Strategy Server* which is located close to LPs to minimise both rejections and slippage.

1.2.2 Binary protocols

The BidFX Python API implements two binary protocols that are optimised to deliver realtime quotes from LPs directly to into a client's application with minimal latency. The binary delivery mechanism is more efficient than the FIX protocol used by most banks to publish prices. Furthermore, using the *publish and subscribe* paradigm, BidFX servers publish only those quotes that are subscribed to, thus saving significantly in network traffic.

1.3 User Guide

1.3.1 Download and installation

Installation

The library is setup to use the `pip` package manager. It can be installed running `pip` as follows.

```
pip install bidfx-api
```

Python version

BidFX Python API works with Python 3.6 and greater. To check that your have the right version of Python installed use the command:

```
python --version
```

You should get some output like `Python 3.7.5`.

If you do not have Python, please install the latest 3.x version from python.org or refer to the Installing Python section of the Hitchhiker's Guide to Python.

Git repository

As part of the *BidFX Open Source initiative*, BidFX intend to published the source code for the Python API on the BidFX Github Page. This source will be released soon. When available, you will be able to clone the API with the following command.

```
git clone https://github.com/bidfx/bidfx-api-py.git
```

1.3.2 Session configuration

The `bidfx` package contains all of the classes, methods and event handlers that are necessary to subscribe to pricing from multiple pricing services.

To work with the API, the first thing you need to do is create a `Session`. *`Session`* is the core class that allows you to subscribe to prices and trade via BidFX. To connect to the BidFX platform the Session must first be configured. There are three ways to configure the Session:

1. by using a default config file located in your home directory
2. specifying a named config file from any location
3. creating a config in code.

Details of how to configure the API can be found at *API Configuration*. In our examples we will just use the default method to create and configure a Session as follows.

```
from bidfx import Session
session = Session.create_from_ini_file()
```

1.3.3 Pricing API

The Pricing API interface is obtained as a property of the *`Session`*. Pricing makes use of a publish-subscribe paradigm in which clients register for price updates by subscribing on subjects. A *`Subject`* identifies a view of an individual instruments for which realtime pricing may be obtained.

LPs publish streams of realtime prices against large numbers of subjects. The BidFX price service matches the client's subscribed subjects against the total universe of published subjects and forwards on to each client only those price updates that match their subscriptions.

Pricing uses threads to manage asynchronous communication with the price servers. The threads need to be started explicitly. Realtime data is passed back to the user-code via event-handling callback functions. The normal pattern for using the pricing API is:

1. configure the Session
2. fetch the pricing API from the Session
3. register the pricing callback functions
4. start the pricing threads
5. subscribe to Subjects

Minimal example

Here is a small but complete example of a price consuming application:

```
from bidfx import Session


def on_price_event(event):
    print(f"Price update to {event}")


def main():
    session = Session.create_from_ini_file()
    pricing = session.pricing
    pricing.callbacks.price_event_fn = on_price_event
    pricing.subscribe(
        pricing.build.fx.stream.spot.liquidity_provider("CSFX")
        .currency_pair("EURUSD")
        .currency("EUR")
        .quantity(1000000)
        .create_subject()
    )
    pricing.start()


if __name__ == "__main__":
    main()
```

After subscribing to a Subject, you will start receiving related *PriceEvent* notifications via the registered callback function: `pricing.callbacks.price_event_fn`.

In addition, if required, whenever the status of a subscription changes a *SubscriptionEvent* notification is published via the registered subscription status callback `pricing.callbacks.subscription_event_fn`.

FX streaming example

Example of streaming (RFS) firm spot rates direct from LPs

```
import logging

from bidfx import Session, Subject


def on_price_event(event):
    if event.price:
        print(
            "{} {} {} {} {} -> {}".format(
                event.subject[Subject.CURRENCY_PAIR],
                event.subject[Subject.LIQUIDITY_PROVIDER],
                event.subject[Subject.DEAL_TYPE],
                event.subject[Subject.CURRENCY],
                event.subject[Subject.QUANTITY],
                event.price,
            )
        )


def on_subscription_event(event):
    print(f"Subscription to {event}")
```

(continues on next page)

(continued from previous page)

```
def on_provider_event(event):
    print(f"Provider {event}")


def main():
    logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s %(levelname)-7s %(threadName)-12s %(message)s",
    )
    session = Session.create_from_ini_file()
    pricing = session.pricing
    pricing.callbacks.price_event_fn = on_price_event
    pricing.callbacks.subscription_event_fn = on_subscription_event
    pricing.callbacks.provider_event_fn = on_provider_event
    pricing.start()

    pricing.subscribe(
        pricing.build.fx.stream.spot.liquidity_provider("DBFX")
        .currency_pair("EURUSD")
        .currency("EUR")
        .quantity(1000000)
        .create_subject()
    )
    pricing.subscribe(
        pricing.build.fx.stream.spot.liquidity_provider("DBFX")
        .currency_pair("USDJPY")
        .currency("USD")
        .quantity(5000000)
        .create_subject()
    )


if __name__ == "__main__":
    main()
```

Callbacks

The *Price API* notifies user-code of various pricing events via a set of callback functions. Separate callback are provided for:

- Price update events (*price_event_fn* for *PriceEvent*)
- Subscription status events (*subscription_event_fn* for *SubscriptionEvent*)
- Provider status events (*provider_event_fn* for *ProviderEvent*)

Events are dispatched through in instance of the class *Callbacks* which is a property of the *PricingAPI*.

Price field names

Price Fields as just strings key-pairs. The field names are simple works or terms such as “Bid”, “Ask” or “AskSize”. A list of the most common price field names is provided by the class *Field*.

Building subjects

Because BidFX connects to many different liquidity providers our instrument symbology is necessarily complex. Each instrument that can be subscribed on is defined by a unique *Subject*. A Subject is an immutable object that looks and behaves similar to a read-only `dict`. It contains many key-value string pairs called *Subject components*. FX price Subjects can be particularly large, especially when it comes to swaps and NDS which are described by many components. Here are a few example Subjects parsed from strings (not recommended):

```
from bidfx import Subject
indi_spot = Subject.parse_string("AssetClass=Fx,Exchange=OTC,Level=1,Source=Indi,
↪Symbol=USDCAD")
rfs_spot  = Subject.parse_string("AssetClass=Fx,BuySideAccount=GIVE_UP_ACCT,
↪Currency=EUR,DealType=Spot,Level=1,LiquidityProvider=CSFX,Quantity=5000000.00,
↪RequestFor=Stream,Symbol=EURUSD,Tenor=Spot,User=smartcorp_api")
rfq_ndf   = Subject.parse_string("AssetClass=Fx,BuySideAccount=GIVE_UP_ACCT,
↪Currency=USD,DealType=NDF,Level=1,LiquidityProvider=DBFX,Quantity=1000000.00,
↪RequestFor=Quote,Symbol=USDKRW,Tenor=1M,User=smartcorp_api")
```

Subjects are case sensitive. Their components are ordered alphabetically by key. It is important to get the Subject syntax and component spellings right, otherwise the subscription will fail. This is non-trivial for newcomers as Subject formats vary by both asset class and deal type.

To build Subjects correctly, its is best to use a *Subject builder* which provides method-chaining to aid syntax discovery and validation to check the result. The API provides a Subject builder as a property of the *PricingAPI* interface. This allows you to construct to the following types of Subject:

- Indicative FX
- FX Request for Stream (RFS/ESP) - Spot, Forward, NDF
- FX Request for Quote (RFQ) - Spot, Forward, NDF, Swap and NDS
- Future
- Equity

Below are some Subject building examples that produce the same Subjects as the parsed strings above.

```
from bidfx import Session
pricing = Session.create_from_ini_file().pricing
indi_spot = pricing.build.fx.indicative.spot.currency_pair("USDCAD").create_subject()

rfs_spot  = pricing.build.fx.stream.spot.liquidity_provider("CSFX").currency_pair(
    "EURUSD").currency("EUR").quantity(5000000).create_subject()

rfq_ndf   = pricing.build.fx.stream.spot.liquidity_provider("DBFX").currency_pair(
    "USDKRW").currency("USD").quantity(1000000).create_subject()

# To subscribing to pricing
pricing.subscribe(indicative_spot)

# To un-subscribing from pricing
pricing.unsubscribe(indicative_spot)
```

1.4 Example Programs

The BidFX API comes bundled with a number of example programs to demonstrate its usage. These can be found under the `examples` directory, immediately under the top-level directory. Two sets of examples are provided to

demonstrate the:

- Pricing API

These are located in separate sub-directories called `examples/pricing`.

1.4.1 Configuration

All of the examples share a common means of configuration using a Window-style INI file. Out of the box the examples will attempt to locate the API configuration in the file `.bidfx/api/config.ini` located in the users home directory i.e. `$HOME/.bidfx/api/config.ini`. An example config file is provided at `examples/config_example.ini` to get you started. Tailor the configuration as follows before attempting to run the examples.

```
cp examples/config_example.ini $HOME/.bidfx/api/config.ini
vi $HOME/.bidfx/api/config.ini
chmod 600 $HOME/.bidfx/api/config.ini
```

Edit the INI file to add the host name and user credentials provided by your BidFX account manager. It is best to make the file read-only to protect the credentials. See *API Configuration* for documentation of the supported parameters.

1.4.2 Running the examples

Running in an IDE

For running the example programs and for general Python development, we recommend an integrated development environment (IDE) designed for programming in Python. We like the PyCharm from JetBrains. With **PyCharm** the examples can be run directly by right-clicking on the example program in the `Project` tab and selecting `Run`.

Running from the command line

You may also run the examples directly from the command line. The examples are all executable Python scripts.

The scripts will select the first version of `python` from your `$PATH` environment variable. The scripts can be run directly on UNIX as follows.

```
./examples/pricing/example_indicative_fx.py
```

Alternatively, if you want to use a specific version of `python` then pass the example program file as the argument to the version of `python` that you prefer.

1.5 API Configuration

The API is best configured using a Window's style INI file. It is common to configure the API by supplying the name of the INI on a call *`Session.create_from_ini_file()`*. If no filename is provided then the *`Session`* will look for a file called `.bidfx/api/config.ini` in the user's home directory.

1.5.1 Configuration methods

There are three ways to provide a configuration for the API:

1. by using a default config file located in your home directory

2. specifying a named config file from any location
3. creating a config in code.

Default configuration

For convenience, all of configuration items may be provided through a single configuration object. The API uses a ConfigParser to read its configuration from a Windows-style INI file.

The config file is commonly located at `$HOME/.bidfx/api/config.ini`. When using the default INI file location, the *Session* can be created and configured as follows.

```
from bidfx import Session
session = Session.create_from_ini_file()
```

Non-standard configuration file

The config file location may be changed by passing a in file name to *Session.create_from_ini_file()*.

```
from bidfx import Session
session = Session.create_from_ini_file("./my_config.ini")
```

Configuration in code

The `ConfigParser` can be built directly in code if preferred, then passed into the *Session* constructor as follows.

```
from bidfx import Session
from configparser import ConfigParser
config_parser = ConfigParser()
# configure config manually
session = Session(config_parser)
```

1.5.2 INI file sections

There is four sections in the configuration INI file:

- `[DEFAULT]` - defines shared properties such as *host* and *port*.
- `[Exclusive Pricing]` - is for overrides and properties particular to *Exclusive Pricing* (Pixie protocol).
- `[Shared Pricing]` - is for overrides and properties particular to *Shared Pricing* (Puffin protocol).

Default section

The `[DEFAULT]` section provides default properties that are shared by the other sections. At present all four protocols can be accessed by tunnelling via a single host on secure port 443. The required user credentials are also the same for all usages of the API. These data can therefore be defined once in the `[DEFAULT]` section of the configuration. Should this situation change then it is possible to override default settings in the each specific sections.

Shared pricing section

Shared Pricing, as the name suggests, is pricing which is shared between many users. An example of shared pricing is Exchange Listed Futures; all exchange members receive the same shared price stream. The BixFX API currently implements the Puffin protocol for subscribing to shared pricing via a Puffin server.

The configuration of the Price Provider for *shared pricing* requires the following properties:

- host
- port
- username
- password

Exclusive pricing section

Exclusive pricing by contrast is not shared across users. It is exclusive to one particular user or group of users. Tradable FX OTC prices direct from liquidity providers are exclusive to the subscribing user. The BixFX API currently implements the Pixie protocol for subscribing to exclusive pricing from BidFX price servers.

The configuration of the Price Provider for *exclusive pricing* uses the following properties:

- host
- port
- username
- password
- product_serial
- default_account
- min_interval

1.5.3 Example INI config file

```
[DEFAULT]
# The host and port number of the BidFX service to connect to.
host = api.ld.bidfx.biz
port = 443

# Provide the API login credentials provided by your BidFX account manager.
username = smartcorp_api
password = 4EL77HqPC2W8hQut

# If you have an API serial key then set it below otherwise leave it blank.
# product_serial = aad33247deffe2aa2832001f

[Exclusive Pricing]
# Use this section to override DEFAULT settings for user-exclusive pricing.

# When subscribing to user-exclusive quotes, the prices consumed may vary by account.
# A default account is defined here for use no explicit account has been provided.
default_account = GIVE_UP_ACCT
```

(continues on next page)

(continued from previous page)

```
# The minimum price publication interval is given below in milliseconds.
min_interval = 500

[Shared Pricing]
# Use this section to override DEFAULT settings for use with shared pricing.
```

1.6 Global Points of Presence

BidFX provides several network Points-Of-Presence (POPs) around the world where our clients may connect to access our pricing and trading service. TLS secures all connections over a single port which is sufficient for both pricing and order flow.

1.6.1 Production

The table below lists the available Production POPs.

City	Environment	DNS name	IP	Port
London	PROD	api.ld.bidfx.com	64.52.164.90	443
New York	PROD	api.ny.bidfx.com	64.52.165.90	443
Tokyo	PROD	api.ty.bidfx.com	64.52.166.90	443
Singapore	PROD	api.sg.bidfx.com	64.52.167.90	443

Clients are encouraged to connect to their **geographically closest** access point for minimum latency.

1.6.2 User acceptance testing

BidFX provide dedicated User Acceptance Test (UAT) environments which clients can uses to test and certify their software before going into production. The table below lists the available UAT POPs.

City	Environment	DNS name	IP	Port
London	UAT	api.ld.bidfx.biz	64.52.164.80	443
Tokyo	UAT	api.ty.bidfx.biz	64.52.166.80	443

UAT pricing subscriptions are routed to liquidity provider **test sessions**. These will quote prices that are *similar* to current market rates but will not compare well with production data. Prices on bank test session are good for heavily traded instruments but less so for less-liquid instruments such as NDF.

In UAT orders are routed to a BidFX market simulator running in New York. Clients located far from the simulator may experience latency on their orders as they route to New York for handling.

1.7 Potential issues

1.7.1 On-boarding issues

Credentials

Applications need valid credentials to connect to services using the BidFX API. Please don't use personal credentials for programmatic access via the API. BidFX on-boarding staff provide dedicated credentials for API usage. Credentials will differ between UAT and Production, so be sure not to mix the two.

If you create a `Session` using an incorrect username or password, or you fail to configure credentials, then the API will issue a suitable error soon after starting.

API product

A valid credentials are insufficient by themselves to access resources using the API. The user also requires a backend *product assignment* for API access. Your BidFX account manager will set this up for you. There may be a usage charge associated with API access.

Booking account

FX subscriptions require a booking account. Prices, session routing and entitlements can differ from one account to another. If required, a separate account can be set for each subscription but commonly a single default account is used for all API subscriptions. The *default account* is configured on the API `Session` config. Please verify with your account manager to ensure you are providing the correct account to the API.

Relationships and entitlements

You might find that you can connect to the price service, receive some pricing but not get data for all subscriptions; perhaps some LPs or currency pairs are absent. This situation is most likely due to the assigned customer relationships or entitlements for the user account. Talk to you account manager to ensure that all of your LPs have been fully on-boarded.

1.7.2 Connectivity issues

First time users of the API may experience connectivity issues due to firewalls, content filtering or network accelerator devices managed by their corporate network team.

Firewalls

A firewall is in place at most organisations to provide basic network security. It is often necessary for API users to request their local network team to white-list an IP address or open a port in the firewall to the required BidFX POP. See *Global Points of Presence* for a list of IP addresses.

If you encounter network issues then it is possible to test the connectivity to BidFX using the `ping` and `telnet` commands as follows:

```
$ ping api.ld.bidfx.biz
PING api.ld.bidfx.biz (64.52.164.80): 56 data bytes
64 bytes from 64.52.164.80: icmp_seq=0 ttl=58 time=1.175 ms
64 bytes from 64.52.164.80: icmp_seq=1 ttl=58 time=1.076 ms

$ telnet api.ld.bidfx.biz 443
Trying 64.52.164.80...
Connected to api.ld.bidfx.biz.
Escape character is '^]'.
```

Consult your network team if either command returns an error or hangs.

Network devices

Security-conscious organisations deploy sophisticated network devices to inspect packets, shape traffic and optimise bandwidth. Such devices can corrupt the message flow to BidFX and cause a protocol marshalling error, quickly followed by a session disconnect.

Network security devices may allow the initial connection through to BidFX but interfere subsequently with either the price protocol or the TLS security layer. Packet inspection might, for example, appear to the API as a man-in-the-middle attack. Some packet inspection devices prevent WebSocket upgrades by filtering HTTP headers, either intentionally or unintentionally.

The observed symptom in these cases is a read timeout or an SSL handshake error, received soon after establishing the connection. The solution to all of these issues is to request your company's network team to bypass the offending device(s) for the BidFX IP addresses and ports. See *Global Points of Presence* for a list of IP addresses and ports used by BidFX.

1.7.3 Subscription limits

The BidFX Price Service limits the number of simultaneous price subscriptions that an API may make. We restrict subscriptions, both to protect the Price Service from excessive use and to guarantee a high quality of service for all users. The default subscription limits allow hundreds of price subscriptions and are sufficient and appropriate for most applications.

Increasing limits

BidFX may provide a licence key that grants a larger subscription limit for demanding applications. The licence key is provided to the API via the `Session` config. To qualify for a higher subscription limit, the client will need to:

- have sufficient CPU capacity to consume the increased price update load,
- run their application physically close to the source of liquidity,
- have a good quality, high bandwidth network connection (ideally a data centre cross-connect to BidFX).

There may be an additional service charge for an increased subscription limit. Ask your BidFX sales representative for price details.

1.7.4 Latency

FX pricing has the potential to update very rapidly, especially around times of major news announcements. Too many price subscriptions can generate substantial amounts of network traffic, causing bandwidth saturation and heavy CPU load on the application host. If an overworked application becomes a *slow consumer* then it will experience latency.

We recommend all API users to close subscriptions that are no longer required to minimise network load.

If you experience latency then there are a few remedial actions you can take:

- Reduce the number of open subscriptions.
- Change the configuration to increase the price publication throttle.
- Move your application close to your main source of liquidity.
- Install a dedicated network link with high capacity and QoS.
- Ideally cross-connect at the same data center as BidFX.

CHAPTER 2

API docs

2.1 BidFX package

The BidFX API provides a *Session* via which access is given to all of the API features. The Session represents a applications's working session with the API for accessing either real-time pricing or the trading capabilities. Sessions open and maintain network connections to services running within the BidFX platform. They create threads to manage these connections asynchronously. These threads must be started explicitly after the Session has been configured.

Typical Usage

```
# Create the Session from the INI file at .bidfx/api/config.ini
session = Session.create_from_ini_file()

# Set the callback for receiving price update events
session.pricing.callbacks.price_event_fn = on_price_event

# Set the pricing threads
session.pricing.start()

# Subscribe to streaming FX prices for €1m EURUSD at spot from DBFX
subject = session.pricing.build.fx.stream.spot.liquidity_provider("DBFX")
    .currency_pair("EURUSD").currency("EUR").quantity(1000000).create_subject()
session.pricing.subscribe(subject)
```

2.1.1 Session

class bidfx.**Session**(*config_parser*)

A Session is the top-level API class which gives access to all of the features of the API. It represents a client's working session with the API.

Parameters **config_parser** (*configparser.ConfigParser*) – The API configuration settings.

static create_from_ini_file(*config_file='~/.bidfx/api/config.ini'*)
Creates a new Session using configuration data parsed from an INI file. The default behaviour is to search for the file `.bidfx/api/config.ini` in the user's home directory.

Parameters **config_file** – is the name of the INI file.

Returns the *Session* configured from the configuration file.

pricing
Gets the Pricing API session used for subscribing to realtime _prices.

Returns A configured interface to the *PricingAPI*.

Return type *PricingAPI*

static version() → str
Gets the API version number.

Return type str

2.1.2 BidFXError

exception bidfx.**BidFXError**
Base class for all errors raised by the BidFX API. Extends `Exception`.

2.1.3 PricingError

exception bidfx.**PricingError**
Base class for all errors raised by the BidFX Pricing API. Extends *BidFXError*.

2.1.4 InvalidSubjectError

exception bidfx.**InvalidSubjectError**
Error indicating the a price *Subject* is invalid. Extends *PricingError*.

2.2 Pricing

The Pricing API is used to provide users with access to real-time pricing from the BidFX platform. Although the API can be used independently of trading, it is recommended that pricing be accessed via the top-level *Session* class of the BidFX API.

2.2.1 PricingAPI

class bidfx.**PricingAPI**(*config_parser*)
Pricing is the top-level API interface for accessing the real-time pricing services of BidFX. It implements two *PriceProvider* implementations: one for *exclusive pricing* that uses the Pixie protocol, and one for *shared pricing* that uses the Puffin protocol.

Parameters **config_parser** (*configparser.ConfigParser*) – The API configuration.

start()
Starts the pricing threads which connect to and manage real-time price services asynchronously.

stop()
Stops the pricing threads.

subscribe(*subject*)
Subscribes to real-time price publications on a given *Subject* representing an instrument.

Parameters **subject** (Subject) – The price subject to subscribe to.

unsubscribe(*subject*)
Un-subscribes from a previously subscribed price *Subject*.

Parameters **subject** (Subject) – The price subject to unsubscribe from.

build
Provides a handle to a the subject builder interface that provides a convenient way to construct a well-formed and validated price *Subject* by using a blend of method-chaining and the builder pattern. The method-chains guide the user to find a correct subject for common classes of instrument, and the builder then validates the resulting subject. For example:

```
# Create an indicative FX spot subject
pricing.build.fx.indicative.spot.currency_pair("GBPAUD").create_subject()

# Create a tradable FX OTC spot subject
pricing.build.fx.stream.spot.liquidity_provider("DBFX").currency_pair("USDJPY
↪").currency("USD").quantity(5000000).create_subject()
```

Returns A method-chain that should lead to the creation a valid *Subject*.

callbacks
Accessor for setting callbacks for pricing related events.

Returns The set of *Callbacks* that determine which user-functions get called for each type of event.

Return type *Callbacks*

static create_price_provider(*config_section*, *callbacks*, *protocol*)
Creates a price provider for a given protocol. Allowed values are 'Pixie' or 'Puffin'. Most applications will not use this method directly as the *PricingAPI* will create the required price providers.

Parameters

- **config_section** (*configparser.ConfigParser[section]*) – Provider section of the API configuration.
- **callbacks** (Callbacks) – The callback functions to handle events.
- **protocol** (*str*) – The protocol implementation for the provider. Defaults to 'Pixie'.

Returns A new price provider instance.

Return type *PriceProvider*

Raises ***PricingError*** – if the protocol is not supported.

2.2.2 PriceProvider

class bidfx.**PriceProvider**
A PriceProvider is an interface that encapsulates the operations of an underlying price provider implementation.

start()
Starts the pricing threads which connect to and manage real-time price services asynchronously.

stop()
Stops the pricing threads.

subscribe(*subject*)
Subscribes to real-time price publications on a given *Subject* representing an instrument.

Parameters **subject** (Subject) – The price subject to subscribe to.

unsubscribe(*subject*)
Un-subscribes from a previously subscribed price *Subject*.

Parameters **subject** (Subject) – The price subject to unsubscribe from.

2.2.3 Callbacks

class bidfx.**Callbacks**
This class provides a set of callback functions that can be overridden by the API user to handle the different types of event that are published by the Pricing API.

price_event_fn
The callback function to be used for handling price events.

Type def function(event: *PriceEvent*)

subscription_event_fn
The callback function to be used for handling subscription events.

Type def function(event: *SubscriptionEvent*)

provider_event_fn
The callback function to be used for handling provider events.

Type def function(event: *ProviderEvent*)

2.2.4 Subject

class bidfx.**Subject**(*components*)
A subject is an immutable, multi-component identifier used to identify instruments that may be subscribed to via the pricing API. Subjects are represented as tuples of many nested tuple pairs, where each pair provides a component key and value. Subject components are alphabetically ordered by key.

Example subjects are:

- `AssetClass=Fx,BuySideAccount=ABC,Currency=EUR,DealType=Spot,Level=1, LiquidityProvider=DBFX, Quantity=100000.00,RequestFor=Stream, Symbol=EURUSD,Tenor=Spot,User=smartcorp_api`
- `AssetClass=Fx,Exchange=OTC,Level=1,Source=Indi,Symbol=USDJPY`

Subject are safe to compare for equality and to use as the keys of a dictionary. Subjects can be converted to string using `str(subject)` for display purposes. Instances of the subject class can be used much like a `dict` to pull out the individual component parts. For example:

```
>>> subject = Subject(...)
>>> ccy_pair = subject[Subject.CURRENCY]
```

A number of common subject component keys are provided as constants of the Subject class for this purpose.

Parameters **components** (*tuple*) – A tuple of subject components, key-value pairs as tuples.

flatten()
Flattens the subject into a simple list of the subject's key and value pairings.

Returns A flattened list of strings.

Return type list

static parse_string(*s*)
Creates a new Subject by parsing the string form of subject.

Parameters **s** (*str*) – The string to be parsed.

Returns A new Subject

Return type *Subject*

static from_dict(*d*)
Creates a new Subject from a dictionary.

Parameters **d** (*dict*) – The dictionary to convert.

Returns A new Subject

Return type *Subject*

get(*key*, *default*)
Gets the value of a Subject component. The component value is returned if the component is present in the subject, otherwise the default value is returned.

Parameters

- **key** (*str*) – The key of the subject component to get.
- **default** (*str*) – The default value to be returned if the key is not present in the subject.

Returns A the value of the component mapped from the *key*, or the *default* value.

Return type str

__contains__(*key*)
Checks if this Subject contains a component with the given key.

__len__()
Gets the length of the Subject in terms of components.

__str__()
Gets a string representation of the Subject.

__eq__(*other*)
Tests the subject for equality with another Subject.

__hash__()
Provides a hash code of the Subject.

ASSET_CLASS = 'AssetClass'

BUY_SIDE_ACCOUNT = 'BuySideAccount'

CURRENCY = 'Currency'

CURRENCY_PAIR = 'Symbol'

DEAL_TYPE = 'DealType'

EXCHANGE = 'Exchange'

EXPIRY_DATE = 'ExpiryDate'

FAR_CURRENCY = 'FarCurrency'

FAR_FIXING_DATE = 'FarFixingDate'

FAR_QUANTITY = 'FarQuantity'

FAR_SETTLEMENT_DATE = 'FarSettlementDate'

FAR_TENOR = 'FarTenor'

FIXING_CCY = 'FixingCcy'

FIXING_DATE = 'FixingDate'

LEVEL = 'Level'

LIQUIDITY_PROVIDER = 'LiquidityProvider'

ON_BEHALF_OF = 'OnBehalfOf'

PUT_CALL = 'PutCall'

QUANTITY = 'Quantity'

REQUEST_TYPE = 'RequestFor'

ROUTE = 'Route'

ROWS = 'Rows'

SETTLEMENT_DATE = 'SettlementDate'

SOURCE = 'Source'

STRIKE = 'Strike'

SYMBOL = 'Symbol'

TENOR = 'Tenor'

USER = 'User'

2.2.5 Tenor

class bidfx.**Tenor**

Tenor values for defining the settlement period in a *Subject* for FX futures and swaps.

BROKEN_DATE = 'BD'

Broken data tenor implied that an explicit settlement date is provided.

TODAY = 'TOD'

Today or same day settlement.

TOMORROW = 'TOM'

Tomorrow or next day settlement. Next good business day after today.

SPOT = 'Spot'

Spot date settlement. Spot is T+1 or T+2 depending of the currency pair.

SPOT_NEXT = 'S/N'

Spot/next settlement. The next good business day after spot.

IN_1_WEEK = '1W'

Settlement in one week.

IN_2_WEEKS = '2W'
Settlement in two weeks.

IN_3_WEEKS = '3W'
Settlement in three weeks.

IN_1_MONTH = '1M'
Settlement in one month.

IN_2_MONTHS = '2M'
Settlement in two months.

IN_3_MONTHS = '3M'
Settlement in three months.

IN_4_MONTHS = '4M'
Settlement in four months.

IN_5_MONTHS = '5M'
Settlement in five months.

IN_6_MONTHS = '6M'
Settlement in six months.

IN_7_MONTHS = '7M'
Settlement in seven months.

IN_8_MONTHS = '8M'
Settlement in eight months.

IN_9_MONTHS = '9M'
Settlement in nine months.

IN_10_MONTHS = '10M'
Settlement in ten months.

IN_11_MONTHS = '11M'
Settlement in eleven months.

IN_18_MONTHS = '18M'
Settlement in eighteen months.

IN_30_MONTHS = '30M'
Settlement in thirty months.

IN_1_YEAR = '1Y'
Settlement in one year.

IN_2_YEARS = '2Y'
Settlement in two years.

IN_3_YEARS = '3Y'
Settlement in three years.

IN_4_YEARS = '4Y'
Settlement in four years.

IN_5_YEARS = '5Y'
Settlement in five years.

IMM_MARCH = 'IMMH'
Settlement coinciding with the IMM cash futures contract for March.

IMM_JUNE = 'IMMM'
Settlement coinciding with the IMM cash futures contract for June.

IMM_SEPTEMBER = 'IMMU'
Settlement coinciding with the IMM cash futures contract for September.

IMM_DECEMBER = 'IMMZ'
Settlement coinciding with the IMM cash futures contract for December.

classmethod of_week(*week: int*)
Gets the weekly tenor of the given number of weeks. :param week: number of weeks :return: the tenor value

classmethod of_month(*month: int*)
Gets the monthly tenor of the given number of months. :param month: number of months :return: the tenor value

classmethod of_year(*year: int*)
Gets the yearly tenor of the given number of years. :param year: number of months :return: the tenor value

classmethod of_imm_month(*month*)
Gets the IMM monthly contract tenor of the given month. :param month: months number in year 1..12 :return: the tenor value

2.2.6 PriceEvent

class bidfx.**PriceEvent**(*subject*, *price*, *full*)
This class defines a Price Event that gets published for each price tick received on a subscription. Price events should be handled by setting a callback function via *PricingAPI.callbacks*. The callback function could be implemented and used as follows.

```
def on_price_event(event):
    if event.price:
        print("price update {} {} / {}".format(
            event.subject[Subject.CURRENCY_PAIR],
            event.price.get(Field.BID, ""),
            event.price.get(Field.ASK, "")))

def main():
    session = Session.create_from_ini_file()
    session.pricing.callbacks.price_event_fn = on_price_event
```

Notice from above that you can use the constants provided by:

- *Subject* to access the components of the subject
- *Field* to access the fields of the price update.

Parameters

- **subject** (*Subject*) – The unique subject of the price subscription.
- **price** (*dict*) – The price as a map for fields.
- **full** (*bool*) – Flag indicating if this is a full or partial price update.

subject
The *Subject* of the price event.

Type *Subject*

price
The map of updated price field.

Type dict

full
A boolean flag indicating if the update represents a full or partial update. The value is set to `True` for a full price image and `False` for a partial update.

Type bool

2.2.7 SubscriptionEvent

class bidfx.**SubscriptionEvent**(*subject*, *status*, *explanation*)
This class defines a Subscription Event that gets published whenever the status of subscription changes. Subscription events should be handled by setting a callback function via *PricingAPI.callbacks*. The callback function could be implemented and used as follows.

```
def on_subscription_event(event):
    print(f"Subscription to {event.subject} is {event.status.name}")

def main():
    session = Session.create_from_ini_file()
    pricing.callbacks.subscription_event_fn = on_subscription_event
```

Parameters

- **subject** (Subject) – The unique subject of the price subscription.
- **status** (SubscriptionStatus) – The subscription status.
- **explanation** (*str*) – An explanation of the status reason.

subject
The *Subject* of the price event.

Type *Subject*

status
The *SubscriptionStatus* associated with the event.

Type *SubscriptionStatus*

explanation
An optional explanation message for the status event.

Type str

2.2.8 ProviderEvent

class bidfx.**ProviderEvent**(*provider*, *status*, *explanation*)
This class defines Provider Event that gets published whenever the status of price provider changes. Provider events should be handled by setting a callback function via *PricingAPI.callbacks*. The callback function could be implemented and used as follows.

```
def on_provider_event(event):
    print(f"Provider {event.provider} is {event.status.name}")

def main():
    session = Session.create_from_ini_file()
    pricing.callbacks.provider_event_fn = on_provider_event
```

Parameters

- **provider** (*str*) – The unique name of the price provider.
- **status** (ProviderStatus) – The provider status.
- **explanation** (*str*) – An explanation of the status reason.

provider
The name of the price provider that issued the event.

Type str

status
The *ProviderStatus* associated with of the event.

Type *ProviderStatus*

explanation
An optional explanation message for the status event.

Type str

2.2.9 SubscriptionStatus

class bidfx.**SubscriptionStatus**
This enum defines the number of different statuses that can be applied to a pricing subscription.

OK = 1
The subscription is OK. This state is not normally published, it is implied by any price update.

PENDING = 2
The subscription is pending an update from an upstream service or provider.

STALE = 3
The subscription is stale, possibly due to a connection issue.

CANCELLED = 4
The subscription has been cancelled.

DISCONTINUED = 5
The subscription has been discontinued by the provider (common on RFQ subscriptions).

PROHIBITED = 6
The subscription is prohibited by entitlements.

UNAVAILABLE = 7
The subscription is unavailable perhaps due to routing issues or setup.

REJECTED = 8
The subscription has been rejected by the provider.

TIMEOUT = 9
The subscription has timed out.

INACTIVE = 10
The subscription has been detected as being inactive.

EXHAUSTED = 11
The subscription is has exhausted a usage limit or resource.

CLOSED = 12
The subscription has been closed (normally by the client API). This is a terminal state.

2.2.10 ProviderStatus

class bidfx.**ProviderStatus**
This enum defines the number of different statuses that can be applied to a price provider.

READY = 1
The price provider is ready for use.

DISABLED = 2
The price provider is has been disabled.

DOWN = 3
The price provider is down and attempting to reconnect.

UNAVAILABLE = 4
The price provider is unavailable.

INVALID = 5
The price provider is invalid most likely due to misconfiguration.

CLOSED = 6
The price provider has been closed. This is the terminal state.

2.2.11 Field

class bidfx.**Field**
Fields provides constants for the most commonly used price field names.

ASK = 'Ask'
Price field containing the ask price.

ASK_END_SIZE = 'AskEndSize'
Price field containing the ask size of the end leg of a swap or NDS.

ASK_EXCHANGE = 'AskExchange'
Price field containing the exchange code from where the ask price has originated.

ASK_FORWARD_POINTS = 'AskForwardPoints'
Price field containing the ask forward points of an FX forward.

ASK_ID = 'AskID'
Price field containing the price ID of a quote of the ask side of an quote book.

ASK_LEVELS = 'AskLevels'
Price field containing the number of market-depth levels of the ask side of an order book.

ASK_FIRM = 'AskFirm'
Price field containing the firm (company) offering on the ask side of an order book.

ASK_SIZE = 'AskSize'
Price field containing the ask size.

ASK_SPOT = 'AskSpot'
Price field containing the ask spot rate associated with an FX forward.

ASK_TICK = 'AskTick'
Price field containing the tick direction for the ask price relative to the previous price.

ASK_TIME = 'AskTime'
Price field containing the time of the last change in the ask price.

BID = 'Bid'
Price field containing the bid price.

BID_END_SIZE = 'BidEndSize'
Price field containing the bid size of the end leg of a swap or NDS.

BID_EXCHANGE = 'BidExchange'
Price field containing the exchange code from where the bid price has originated.

BID_FORWARD_POINTS = 'BidForwardPoints'
Price field containing the bid forward points of an FX forward.

BID_ID = 'BidID'
Price field containing the price ID of a quote of the bid side of an quote book.

BID_LEVELS = 'BidLevels'
Price field containing the number of market-depth levels of the bid side of an order book.

BID_FIRM = 'BidFirm'
Price field containing the firm (company) offering on the bid side of an order book.

BID_SIZE = 'BidSize'
Price field containing the bid size.

BID_SPOT = 'BidSpot'
Price field containing the bid spot rate associated with an FX forward.

BID_TICK = 'BidTick'
Price field containing the tick direction for the bid price relative to the previous price.

BID_TIME = 'BidTime'
Price field containing the time of the last change in the bid price.

BROKER = 'Broker'
Price field containing the name of the broker quoting the price.

CLOSE = 'Close'
Price field containing the previous market close price.

HIGH = 'High'
Price field containing the market high price for the current day or session.

LAST = 'Last'
Price field containing the last traded price.

LAST_SIZE = 'LastSize'
Price field containing the size of the last trade.

LAST_TICK = 'LastTick'
Price field containing the tick direction of the last trade relative to the previous trade.

LOW = 'Low'
Price field containing the market low price for the current day or session.

NET_CHANGE = 'NetChange'
Price field containing the net change in price between the last price and the open price.

NUM_ASKS = 'NumAsks'
Price field containing the number of participants offering at the ask price.

NUM_BIDS = 'NumBids'
Price field containing the the number of participants bidding at the bid price.

OPEN = 'Open'
Price field containing the market price at the open.

OPEN_INTEREST = 'OpenInterest'
Price field containing the open interest in a future.

ORIGIN_TIME = 'OriginTime'
Price field containing the time of a price tick as measured at the originating source.

PERCENT_CHANGE = 'PercentChange'
Price field containing the percentage change between the last price and the market open.

PRICE_ID = 'PriceID'
Price field containing the ID used by an LP to identify a tradable quote.

STRIKE = 'Strike'
Price field containing the strike price of an option.

VOLUME = 'Volume'
Price field containing the volume.

VWAP = 'VWAP'
Price field containing the volume weighted average price.

CHAPTER 3

Indices and tables

- genindex
- search

Python Module Index

b

Index

Symbols

A

B

C

D

E

F

G

H

I

L

N

O

P

Q

R

S

T

U

V