10

Set Up the Interactive Brokers Python API

In *Chapters 1* through *9*, we learned the foundational tools and techniques of algorithmic trading. Now, we'll put this into practice using Python with the **Interactive Brokers** (**IB**) API and **Trader Workstation** (**TWS**). TWS is an advanced trading platform that is used by a wide range of traders, both professional and retail. You may consider alternatives to IB that offer API access. Some other brokers include Alpaca, Think Or Swim, Tasty Trade, and Tradier. When opening an account, make sure to consider market access, account type, commissions, and other factors when selecting a broker. Different brokers and account types may have different costs and restrictions.

TWS is known for its suite of trading tools and features that really enhance the trading experience. TWS is equipped with robust risk management tools so traders can effectively manage and mitigate potential trading risks. IB has unparalleled global market access with the ability to trade across a vast array of financial instruments in 135 markets spanning 33 countries. This feature is useful for traders who want to diversify their portfolios on a global scale.

Further, the paper trading functionality of TWS is an invaluable tool for both novice and experienced traders. It provides a risk-free environment for testing and refining trading strategies using real-time market conditions. This feature is instrumental in helping algorithmic traders develop and hone their algorithms without putting their money at risk.

Finally, TWS's API integration is a significant advantage since it exposes all the features of TWS through an API that is accessible with Python. The IB API is especially useful for algorithmic traders since it lets them automate their trading strategies. We will take advantage of the paper trading account and IB API in the next three chapters.

In this chapter, we present the following recipes:

- Building an algorithmic trading app
- Creating a **`Contract`** object with the IB API
- Creating an **`Order`** object with the IB API
- Fetching historical market data
- Getting a market data snapshot
- Streaming live tick data
- Storing live tick data in a local SQL database

Building an algorithmic trading app

When using the IB API, there's a lot of code that can be reused across different trading apps. Connecting to TWS, generating orders, and downloading data are done the same way despite the trading strategy. That's why it's a best practice to get the reusable code out of the way first. But before we can start building our algorithmic trading app, we need to install TWS and the IB API.

We'll begin with three important topics when using the IB API:

- The first is the architecture of the IB API, which operates on an asynchronous model. In this model, operations are not executed in a linear, blocking manner. Instead, actions are initiated by requests, and responses to these requests are handled via callbacks.
- The second concept is inheritance, which is common across all computer programming languages. Inheritance is where a new child

class acquires the properties and methods of another, parent, class. Our trading app will inherit functionality from two parent classes provided by the IB API (`EClient` and `EWrapper`).
- The third concept is overriding where a child class provides the specific implementation for a method that is defined in a parent class.

We'll learn more about how these three concepts function in the *How it works* section of this recipe.

As we learned in the introduction of this chapter, TWS is IB's trading platform. While running, TWS acts as a server and exposes ports to which the IB API connects. It's through this connection that the IB API operates. In this chapter, we'll install TWS, the IB API, and start building the reusable parts of our algorithmic trading app.

Getting ready...

Before we install the IB API, we need TWS installed on our local computer. You can download TWS for your computer from the IB website (**https://www.interactivebrokers.com/en/trading/tws.php#tws-software**). It's available for Windows, Mac, and Linux.

Once it is installed, you can start TWS.

If you have an account, log in. If not, then you can use the Demo account option on the login screen.

We need to change some settings to allow our Python app to connect to TWS. Navigate to **Trader Workstation Configuration** under **Edit** –> **Global Configuration** –> **API** –> **Settings**. You should a screen that looks like this:

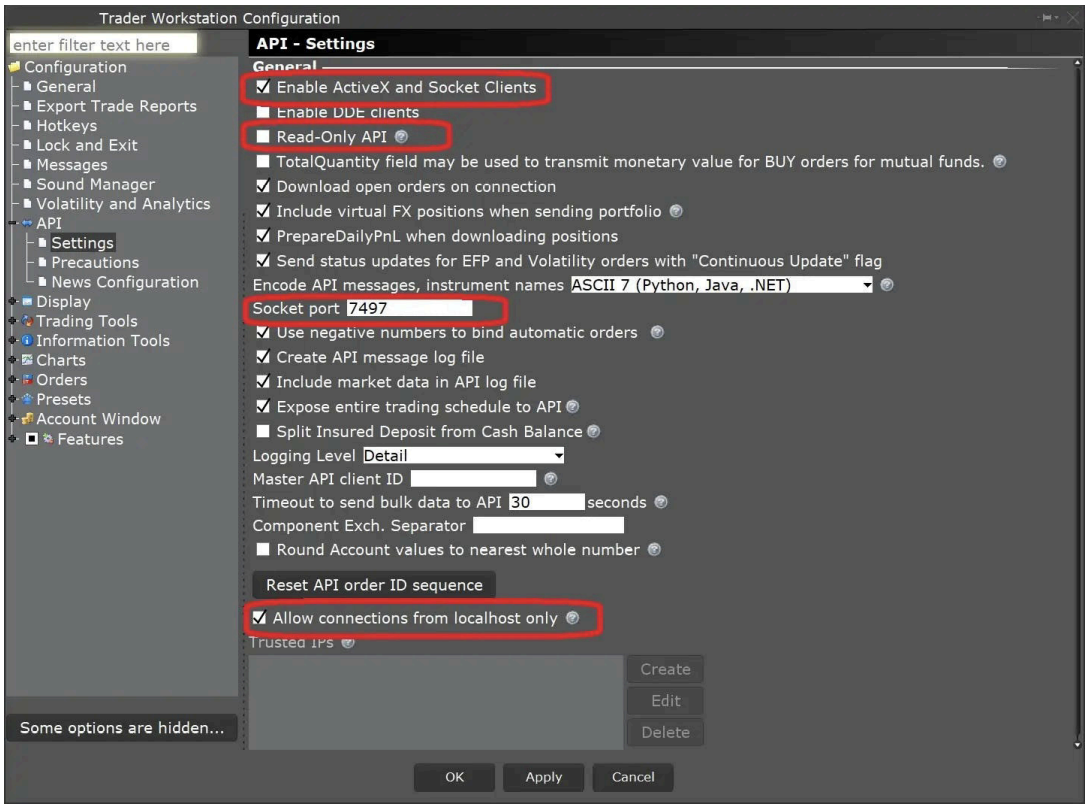

Figure 10.1: Global configuration settings

Make sure to check **Enable ActiveX and Socket Clients**. Check **Read-Only API** if you want extra protection against sending orders to IB. Lastly, check **Allow connections from localhost only** for security.

Make note of the **Socket port**, which you'll need to connect through Python. Depending on whether you logged in with a live or paper trading account, it's either **7497** or **7496**.

> *IMPORTANT NOTE*

> *To work with the IB API live, you need an IBKR Pro account (not IBRK Lite). If you don't want (or can't) set up an IB account, you can use the free demo account. In the free demo account, you will be unable to download or stream market data.*

Near the top-right corner of TWS, you should see a green button that says **DATA**. By clicking on it, you will see the data farms you're connected to. Note the empty section near the bottom titled **API**

Connections (listening on *:7497). Having this section means you've correctly enabled API connections.

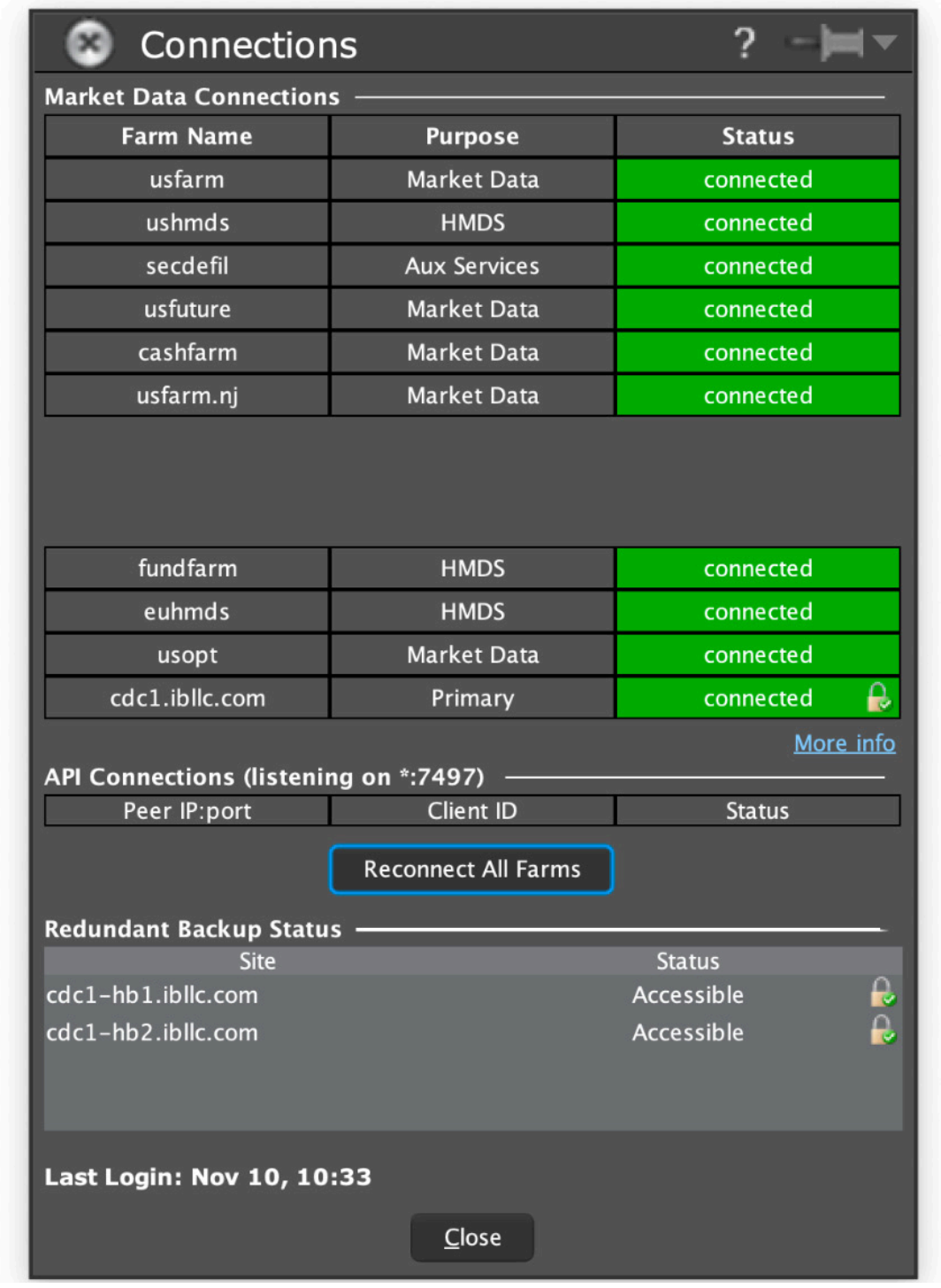

Figure 10.2: Data connections screen showing empty API connections

Once we have TWS set up, you can install the IB API. Download the Python API from the Interactive Brokers GitHub page here: http://interactivebrokers.github.io/. Follow the instructions on the

download page to download and install the latest stable version of the API for your operating system.

Once you have the IB API set up, create a new directory called **`trading-app`**. Inside, create the following Python script files:

- **`__init__.py`**
- **`app.py`**
- **`client.py`**
- **`wrapper.py`**
- **`contract.py`**
- **`order.py`**
- **`utils.py`**

How to do it...

We'll start by setting up the code we need to connect to TWS through the IB API.

1. Add the following code to the **`client.py`** file, which imports a base class from the IB API and implements a custom class we'll use to build our trading app:

```
from ibapi.client import EClient
class IBClient(EClient):
    def __init__(self, wrapper):
        EClient.__init__(self, wrapper)
```

2. Add the following code to the **`wrapper.py`** file, which imports a base class from the IB API and implements a custom class we'll use to build our trading app:

```
from ibapi.wrapper import EWrapper
class IBWrapper(EWrapper):
    def __init__(self):
        EWrapper.__init__(self)
```

3. Add the following code to the **`app.py`** file, which implements a custom class that we'll develop into our trading app:

```
import threading
import time
from wrapper import IBWrapper
from client import IBClient
class IBApp(IBWrapper, IBClient):
    def __init__(self, ip, port, client_id):
        IBWrapper.__init__(self)
        IBClient.__init__(self, wrapper=self)
        self.connect(ip, port, client_id)
        thread = threading.Thread(target=self.run, daemon=True)
        thread.start()
        time.sleep(2)
if __name__ == "__main__":
    app = IBApp("127.0.0.1", 7497, client_id=10)
    time.sleep(30)
    app.disconnect()
```

How it works...

The architecture of the IB API operates on an asynchronous model, which is fundamental to understanding its interaction and data flow mechanisms. In this model, operations are not executed in a linear, blocking manner. Instead, actions are initiated by requests, and responses to these requests are handled via callbacks, making the system highly efficient and responsive.

Request-callback pattern

When we interact with the IB API, we typically start by sending a request for an action, such as retrieving market data or placing an order. This request is made through an instance of **`EClient`**, a component of the API responsible for initiating communication with the IB servers. The **`EClient`** sends the request and then continues with other tasks, rather than waiting for a response. This non-blocking pattern gets around the **Global Interpreter Lock** (**GIL**) of Python and exemplifies the asynchronous nature of the architecture.

The responses to these requests are not received directly by the **`EClient`**. Instead, they are handled by another component called **`EWrapper`**. **`EWrapper`** is essentially a set of callback functions that are triggered when responses or data from the IB server are received. For instance, when market data is requested, **`EWrapper`** will have a specific method that gets called when that market data arrives. The user implements these methods in **`EWrapper`** to define how the data or responses should be processed.

This separation of concerns, where **`EClient`** handles sending requests and **`EWrapper`** handles receiving responses, allows for a more organized and manageable code structure. It also enables the handling of multiple simultaneous data streams and requests efficiently. The asynchronous model ensures that the application remains responsive and can process incoming data or events as they occur, which is crucial for real-time trading applications where timely response to market changes is critical.

Inheritance and overriding

`EWrapper` is an interface class that defines a set of callback methods, which are intended to be overridden by our custom class that inherits from **`EWrapper`**. This inheritance allows us to define specific behaviors for each callback method, tailoring the response to various events, such as receiving market data or order updates. Similarly, **`EClient`** can be extended, or its methods overridden to customize the way requests to the IB server are made, providing flexibility and control over the interaction with the trading system.

Our trading app class

Our main trading app is a custom class that inherits our custom **`IBWrapper`** and **`IBClient`** classes (which in turn inherits the IB API's **`EWrapper`** and **`EClient`** classes). This class is designed to initialize and manage our connection to the IB API.

In the **`__init__`** method of **`IBApp`**, we initialize **`IBWrapper`** and **`IBClient`**, using **`IBWrapper.__init__(self)`** and **`IBClient.__init__(self, wrapper=self)`**. This sets up the environment and properties inherited from these parent classes. The **`IBClient`** initialization specifically requires a reference to an **`IBWrapper`** instance, which is provided by passing **`self`**.

Next, we call the **`connect`** method with the parameters **`ip`**, **`port`**, and **`client_id`**. This method is inherited from **`IBClient`** and is used to establish a connection to TWS. After we connect, a new thread is created and started using Python's **`threading`** module. This thread runs the **`run`** method in a daemon mode, which means it runs in the background and will automatically terminate when the main program exits. The **`run`** method is responsible for processing incoming messages and events from the IB API. Running this in a separate thread allows the **`IBApp`** to handle API messages asynchronously without blocking the main execution flow of the application.

There's more...

Now that we have the base code established, we can test our connection.

Using your terminal, run the **`app.py`** script:

```
python app.py
```

You will see a series of error messages printed on the screen. Don't worry, these are not actually errors. They are just messages indicating you've successfully connected to the IB data farms.

```
ERROR -1 2104 Market data farm connection is OK:usfarm.nj
ERROR -1 2104 Market data farm connection is OK:usfuture
ERROR -1 2104 Market data farm connection is OK:cashfarm
ERROR -1 2104 Market data farm connection is OK:usopt
ERROR -1 2104 Market data farm connection is OK:usfarm
ERROR -1 2106 HMDS data farm connection is OK:euhmds
ERROR -1 2106 HMDS data farm connection is OK:fundfarm
ERROR -1 2106 HMDS data farm connection is OK:ushmds
ERROR -1 2158 Sec-def data farm connection is OK:secdefil
```

Figure 10.3: Output from the IB API showing we successfully connected

> *IMPORTANT NOTE*
>
> *Error code* **-1** *is not actually an error. These are messages indicating successful connections to the IB data farms.*

Also note in the TWS data connection screen, we now have a peer IP connected to TWS. That's the Python trading app we built!

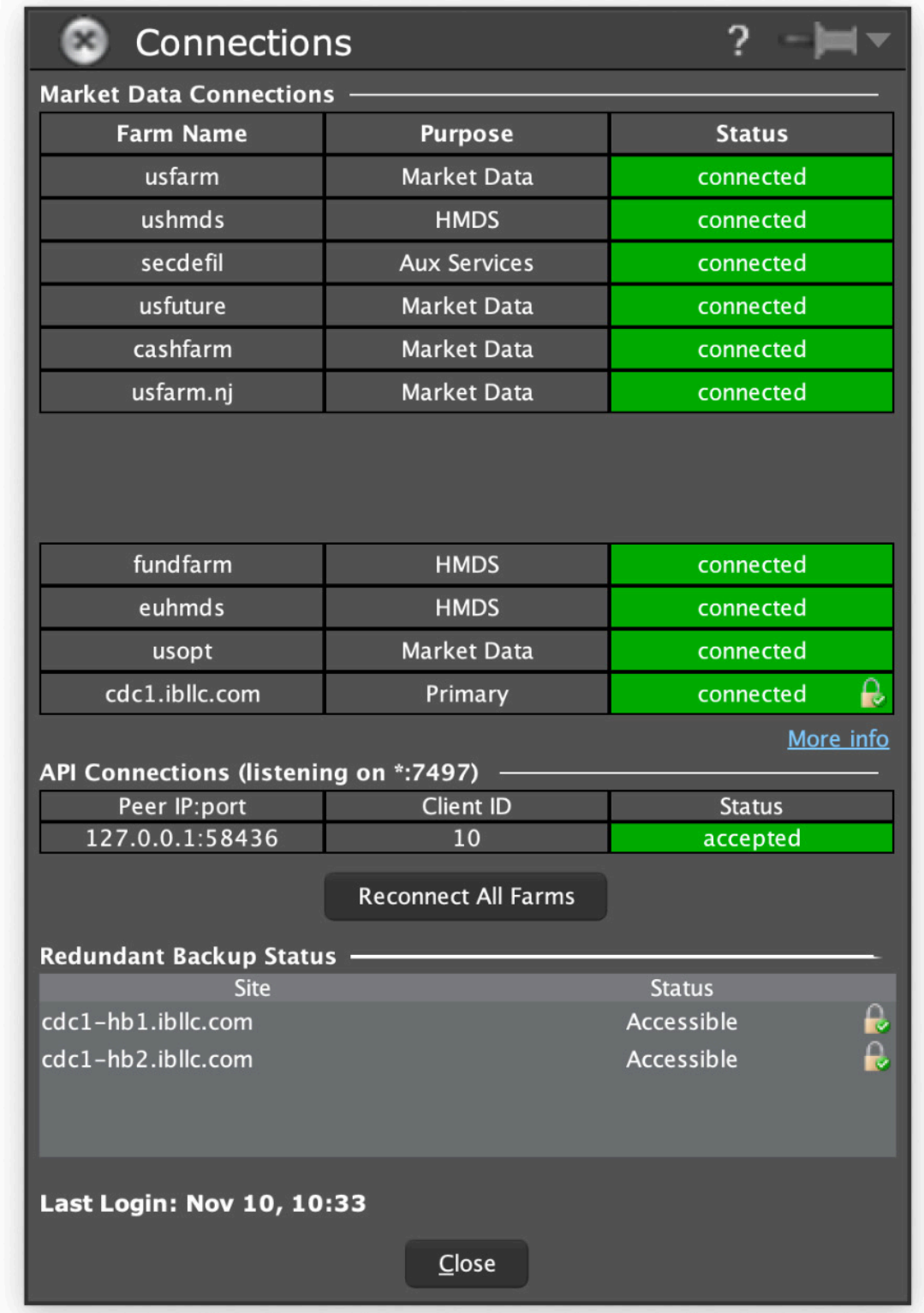

Figure 10.4: TWS data connection screen showing our Python trading app connected to TWS

See also

The IB API has extensive documentation:

- Documentation for the initial setup: **https://interactivebrokers.github.io/tws-api/initial_setup.html**
- Documentation about the **`EClient`** and **`EWrapper`** classes: **https://interactivebrokers.github.io/tws-api/client_wrapper.html**
- Documentation about connectivity: **https://interactivebrokers.github.io/tws-api/connection.html**

Creating a Contract object with the IB API

When requesting market data or generating orders, we do it using the IB **`Contract`** object. An IB **`Contract`** contains all the information required for IB to correctly identify the instrument in question. Using one class, we can represent a broad spectrum of financial instruments, including stocks, options, futures, and more. In this recipe, we'll create an IB **`Contract`**.

The **`Contract`** class is used to define the specifications of a financial instrument that we might want to trade or query. The class has all the necessary details that uniquely identify a financial instrument across various asset classes, such as stocks, options, futures, forex, bonds, and more.

A key attribute of the **`Contract`** class is **`conId`** (contract ID), which is a unique identifier assigned by IB to each financial instrument. However, in many cases, we do not need to specify this ID directly. Instead, we typically provide other descriptive attributes that the IB system uses to uniquely identify the contract. These attributes typically include the following:

- **`symbol`**: The ticker symbol of the asset
- **`secType`**: Specifying the security type (e.g., **`STK`** for stock, **`OPT`** for option, or **`FUT`** for future)
- **`expiry`**: The expiration date for derivative instruments

- **`strike`**: The strike price for options
- **`right`**: Indicating the option type (call or put)
- **`multiplier`**: Defining the leverage or contract size
- **`exchange`**: The primary exchange where the asset is traded

For more complex instruments, additional attributes may be specified, such as **`currency`** for assets that trade in multiple currencies or on international markets. The **`lastTradeDateOrContractMonth`** attribute is used for futures and options to specify the contract month. For options and futures, **`includeExpired`** can be set to indicate whether expired contracts should be considered. The **`localSymbol`** and **`primaryExchange`** attributes can provide more specific contract details, especially for instruments where the standard symbol may not be unique.

In this recipe, we'll create custom functions for a future, stock, and option contract.

> *IMPORTANT NOTE*
>
> *All financial instruments are considered contracts by IB. This might come as a surprise when considering stocks or FX.*

Getting ready…

We assume you've created the **`contract.py`** file in the **`trading-app`** directory. If not, do it now.

How to do it…

We'll create custom functions that create instances of different types of contracts, set the properties as relevant to the type of financial instrument, and return it:

Add the following code to the **`contract.py`** file:

```
from ibapi.contract import Contract
def future(symbol, exchange, contract_month):
    contract = Contract()
    contract.symbol = symbol
    contract.exchange = exchange
    contract.lastTradeDateOrContractMonth = contract_month
    contract.secType = "FUT"
    return contract
def stock(symbol, exchange, currency):
    contract = Contract()
    contract.symbol = symbol
    contract.exchange = exchange
    contract.currency = currency
    contract.secType = "STK"
    return contract
def option(symbol,exchange,contract_month,strike,right):
    contract = Contract()
    contract.symbol = symbol
    contract.exchange = exchange
    contract.lastTradeDateOrContractMonth = contract_month
    contract.strike = strike
    contract.right = right
    contract.secType = "OPT"
    return contract
```

How it works…

The functions are designed to encapsulate the common patterns when creating contracts. The **`future`** function creates a **`Contract`** object representing a futures contract. The function takes three parameters: **`symbol`**, **`exchange`**, and **`contract_month`**, which are required to set up the contract. When called, the function creates an instance of the **`Contract`** class, which is a core component of the IB API for defining financial instruments. This instance, stored in the variable **`contract`**, is then configured with specific properties: the **`symbol`** of the futures contract, the **`exchange`** where it is traded, and the **`lastTradeDateOrContractMonth`**, which defines the expiration date. Additionally, the security type is set

to **FUT**, explicitly marking the contract as a futures contract. The **stock** and **option** functions follow the same pattern.

There's more...

Now that we've defined our contract functions, let's use them. Open **app.py** and add a new import directly under the contract import:

```
from contract import stock, future, option
```

Then right under the instantiation of the app, add the following:

```
aapl = stock("AAPL", "SMART", "USD")
gbl = future("GBL", "EUREX", "202403")
pltr = option("PLTR", "BOX", "20240315", 20, "C")
```

The result of the changes is the following code in the **app.py** file:

```
import threading
import time
from wrapper import IBWrapper
from client import IBClient
from contract import stock, future, option
class IBApp(IBWrapper, IBClient):
    def __init__(self, ip, port, client_id):
        IBWrapper.__init__(self)
        IBClient.__init__(self, wrapper=self)
        self.connect(ip, port, client_id)
        thread = threading.Thread(target=self.run,
            daemon=True)
        thread.start()
        time.sleep(2)
if __name__ == "__main__":
    app = IBApp("127.0.0.1", 7497, client_id=10)
    aapl = stock("AAPL", "SMART", "USD")
    gbl = future("GBL", "EUREX", "202403")
    pltr = option("PLTR", "BOX", "20240315", 20, "C")
```

```
    time.sleep(30)
    app.disconnect()
```

Running this code will create a stock, future, and option contract, wait for 30 seconds, and then disconnect.

See also

All financial instruments are considered contracts by the IB API. This includes stocks, ETFs, bonds, options, and futures. To read more about the Contract objects, see this URL: https://interactivebrokers.github.io/tws-api/contracts.html

Creating an Order object with the IB API

Similar to how a **`Contract`** object encapsulates all the information IB needs for financial instruments, **`Order`** objects contain all the information required for IB to correctly place orders in the market. IB is famous for having dozens of order types, from simple market orders to advanced algorithms that slowly execute trades based on time or volume conditions. Key attributes of the **`Order`** class include the following:

- **`orderId`**: A unique identifier for the order, typically assigned by the client
- **`action`**: Specifies the action type of the order, such as **`BUY`** or **`SELL`**
- **`totalQuantity`**: The amount of the asset to be bought or sold
- **`orderType`**: Defines the type of order, such as **`LMT`** for limit orders or **`MKT`** for market orders
- **`lmtPrice`**: The limit price for limit orders
- **`auxPrice`**: Used for stop or stop-limit orders to specify the stop price
- **`tif`**: *Time in force*, determining how long the order remains active
- **`outsideRTH`**: A boolean indicating whether the order can be executed outside regular trading hours

- **`account`**: Specifies the account the order should be executed from

Additional attributes can be set to further customize the order. These include attributes for advanced order types and conditions, such as **`stopPrice`** for stop orders, **`trailStopPrice`** for trailing stop orders, and various attributes for conditional orders. The **`Order`** class also supports attributes for specifying commission, margin, and other trade-related parameters.

Getting ready…

We assume you've created the **`order.py`** file in the **`trading-app`** directory. If not, do it now.

How to do it…

We'll create custom functions that create instances of different types of contracts, set the properties relevant to the type of financial instrument, and return it.

Add the following code to the **`order.py`** file, which implements custom functions to create and return **`Order`** objects using the IB API:

```
from ibapi.order import Order
BUY = "BUY"
SELL = "SELL"
def market(action, quantity):
    order = Order()
    order.action = action
    order.orderType = "MKT"
    order.totalQuantity = quantity
    return order
def limit(action, quantity, limit_price):
    order = Order()
    order.action = action
    order.orderType = "LMT"
    order.totalQuantity = quantity
```

```
        order.lmtPrice = limit_price
        return order
    def stop(action, quantity, stop_price):
        order = Order()
        order.action = action
        order.orderType = "STP"
        order.auxPrice = stop_price
        order.totalQuantity = quantity
        return order
```

How it works…

A market order is executed immediately at the current market price. It prioritizes speed of execution over price, meaning it will be filled quickly but not necessarily at a specific price point. Market orders are commonly used in situations where the certainty of execution is more important than the exact price of the trade. We can define a market order object using the IB API by first instantiating an **`Order`** object, then assigning either **`BUY`** or **`SELL`** to the **`action`** property, the quantity desired to the **`totalQuantity`** property, and **`MKT`** to the **`orderType`** property.

In similar fashion, we can create a limit order by specifying **`LMT`** as the **`orderType`** and a limit price as the **`lmtPrice`**. A limit order allows traders to specify the maximum price they are willing to pay when buying or the minimum price they are willing to accept when selling. Unlike market orders, limit orders provide price control but do not guarantee execution, as they will only be filled if the market price meets or is better than the limit price.

A stop loss order limits the loss on a position by automatically executing an order when the price reaches a specified stop price. We create a stop loss order by setting **`orderType`** to **`STP`** and setting **`auxPrice`** to the desired stop price.

There’s more…

Now that we've defined our order functions, let's use them. Open **app.py** and add a new import directly under the contract import:

```
from order import limit, BUY
```

Then right under the creation of our contracts, add the following:

```
limit_order = limit(BUY, 100, 190.00)
```

The result of the changes is the following code in the **app.py** file:

```
import threading
import time
from wrapper import IBWrapper
from client import IBClient
from contract import stock, future, option
from order import limit, BUY
class IBApp(IBWrapper, IBClient):
    def __init__(self, ip, port, client_id):
        IBWrapper.__init__(self)
        IBClient.__init__(self, wrapper=self)
        self.connect(ip, port, client_id)
        thread = threading.Thread(target=self.run,
            daemon=True)
        thread.start()
        setattr(self, "thread", thread)
if __name__ == "__main__":
    app = IBApp("127.0.0.1", 7497, client_id=10)
    aapl = stock("AAPL", "SMART", "USD")
    gbl = future("GBL", "EUREX", "202403")
    pltr = option("PLTR", "BOX", "20240315", 20, "C")
    limit_order = limit(BUY, 100, 190.00)
    time.sleep(30)
    app.disconnect()
```

Running this code will create a limit order object, wait for 30 seconds, and then disconnect. We'll look at how to submit orders in ***Chapter 11***, *Manage Orders, Positions, and Portfolios with the IB API*.

See also

To read more about the `Order` object and the available order types, check the documentation here: https://interactivebrokers.github.io/tws-api/available_orders.html

Fetching historical market data

Requesting historical market data using the IB API is an asynchronous process, emphasizing non-blocking, event-driven data retrieval. To kick off the process, we send a request for historical data by invoking the `reqHistoricalData` method, specifying parameters such as the financial instrument's identifier, the duration for which data is needed, the bar size, and the type of data required. Once the request is made, the IB API processes it and begins sending back the data. However, instead of waiting for all data to be received before continuing with other tasks, the API employs a callback mechanism, specifically the `historicalData` method. This method is called asynchronously for each piece of data received from IB. Each invocation of `historicalData` provides a snapshot of market data for a specific time interval, which we can then process or store. In this recipe, we'll set up the code to request and receive historical market data.

Getting ready...

We assume you've created the `app.py`, `client.py`, and `wrapper.py` files in the `trading-app` directory. If not, do it now.

How to do it...

We'll update `app.py`, `client.py`, and `wrapper.py` to request historic market data.

1. Open `client.py` and add the following imports to the top of the file:

```
import time
import pandas as pd
```

2. Create a **`list`** as a constant after the imports. The strings in this list represent the columns of the DataFrame that we will populate with historical market data:

```
TRADE_BAR_PROPERTIES = ["time", "open", "high", "low",
    "close", "volume"]
```

3. Inside the **`IBClient`** class in the **`client.py`** file, add the following method:

```
def get_historical_data(self, request_id, contract, duration, bar_size):
    self.reqHistoricalData(
        reqId=request_id,
        contract=contract,
        endDateTime="",
        durationStr=duration,
        barSizeSetting=bar_size,
        whatToShow="MIDPOINT",
        useRTH=1,
        formatDate=1,
        keepUpToDate=False,
        chartOptions=[],
    )
    time.sleep(5)
    bar_sizes = ["day", "D", "week", "W", "month"]
    if any(x in bar_size for x in bar_sizes):
        fmt = "%Y%m%d"
    else:
        fmt = "%Y%m%d %H:%M:%S %Z"
    data = self.historical_data[request_id]
    df = pd.DataFrame(data, columns=TRADE_BAR_PROPERTIES)
    df.set_index(pd.to_datetime(df.time, format=fmt),
        inplace=True)
    df.drop("time", axis=1, inplace=True)
    df["symbol"] = contract.symbol
    df.request_id = request_id
    return df
```

4. Create a method that allows users to request data for more than one contract:

```
def get_historical_data_for_many(self, request_id,
    contracts, duration, bar_size,
    col_to_use="close"):
        dfs = []
        for contract in contracts:
            data = self.get_historical_data(
                request_id, contract, duration,
                bar_size)
            dfs.append(data)
            request_id += 1
        return (
            pd.concat(dfs)
            .reset_index()
            .pivot(
                index="time",
                 columns="symbol",
                 values=col_to_use
            )
        )
```

The result of the changes is the following code in **`client.py`**:

```
import time
import pandas as pd
from ibapi.client import EClient
TRADE_BAR_PROPERTIES = ["time", "open", "high", "low",
    "close", "volume"]
class IBClient(EClient):
    def __init__(self, wrapper):
        EClient.__init__(self, wrapper)
    def get_historical_data(self, request_id,
        contract, duration, bar_size):
            self.reqHistoricalData(
                reqId=request_id,
                contract=contract,
                endDateTime="",
                durationStr=duration,
                barSizeSetting=bar_size,
                whatToShow="MIDPOINT",
```

```
                useRTH=1,
                formatDate=1,
                keepUpToDate=False,
                chartOptions=[],
            )
            time.sleep(5)
            bar_sizes = ["day", "D", "week", "W",
                "month"]
            if any(x in bar_size for x in bar_sizes):
                fmt = "%Y%m%d"
            else:
                fmt = "%Y%m%d %H:%M:%S %Z"
            data = self.historical_data[request_id]
            df = pd.DataFrame(data,
                columns=TRADE_BAR_PROPERTIES)
            df.set_index(pd.to_datetime(df.time,
                format=fmt), inplace=True)
            df.drop("time", axis=1, inplace=True)
            df["symbol"] = contract.symbol
            df.request_id = request_id
            return df
        def get_historical_data_for_many(self,
            request_id, contracts, duration, bar_size,
            col_to_use="close"):
                dfs = []
                for contract in contracts:
                data = self.get_historical_data(
                    request_id, contract, duration,
                     bar_size)
                dfs.append(data)
                request_id += 1
                return (pd.concat(dfs)
            .reset_index()
            .pivot(
                index="time",
                columns="symbol",
                values=col_to_use
            )
        )
```

5. Next, open the **`wrapper.py`** file and add the following line of code to the **`__init__`** method in the **`IBWrapper`** class:

```
self.historical_data = {}
```

6. Then add the following method to the **`IBWrapper`** class:

```
def historicalData(self, request_id, bar):
    bar_data = (
        bar.date,
        bar.open,
        bar.high,
        bar.low,
        bar.close,
        bar.volume,
    )
    if request_id not in self.historical_data.keys():
        self.historical_data[request_id] = []
    self.historical_data[request_id].append(bar_data)
```

The result of the changes is the following code in **`wrapper.py`**:

```
from ibapi.wrapper import EWrapper
class IBWrapper(EWrapper):
    def __init__(self):
        EWrapper.__init__(self)
        self.historical_data = {}
    def historicalData(self, request_id, bar):
        bar_data = (
            bar.date,
            bar.open,
            bar.high,
            bar.low,
            bar.close,
            bar.volume,
        )
        if request_id not in self.historical_data.keys():
            self.historical_data[request_id] = []
        self.historical_data[request_id].append(
            bar_data)
```

How it works...

Getting historical data through the IB API clearly demonstrates the request-callback architecture. We get historical data by first requesting it, then processing it through a callback.

Requesting historic data

The `get_historical_data` method accepts four parameters: `request_id`, `contract`, `duration`, and `bar_size`. `request_id` is an arbitrary, but unique identifier for the data request, `contract` specifies the financial instrument for which historical data is being requested, `duration` indicates the time span for the historical data, and `bar_size` defines the granularity of the data.

The valid duration strings are as follows:

Unit	Description
S	Seconds
D	Day
W	Week
M	Month
Y	Year

Figure 10.5: Valid duration strings for requesting historical data

The valid bar size strings are as follows:

Size							
1 secs	5 secs	10 secs	15 secs	30 secs			
1 min	2 mins	3 mins	5 mins	10 mins	15 mins	20 mins	30 mins
1 hour	2 hours	3 hours	4 hours	8 hours			
1 day							
1 week							
1 month							

Figure 10.6: Valid bar sizes for requesting historical data

The method begins by calling **`reqHistoricalData`**, which is a method provided by the IB API to request historical data. The parameters passed to this function include **`request_id`**, the **`contract`** object, an empty string for **`endDateTime`** (indicating the request is for the most recent data up to the current date), the **`duration`** of historical data, **`bar_size`**, and other settings such as **`whatToShow`** (indicating the type of data required), **`useRTH`** (data within regular trading hours), **`formatDate`** (the format of the returned date), and **`keepUpToDate`** (indicating no need for live updates). These are hardcoded in our implementation, but you can modify them to allow the user to pass their own options.

The available kinds of historic data type strings (**`whatToShow`**) are as follows:

Type	Open	High	Low	Close	Volume
TRADES	First traded price	Highest traded price	Lowest traded price	Last traded price	Total traded volume
MIDPOINT	Starting midpoint price	Highest midpoint price	Lowest midpoint price	Last midpoint price	N/A
BID	Starting bid price	Highest bid price	Lowest bid price	Last bid price	N/A
ASK	Starting ask price	Highest ask price	Lowest ask price	Last ask price	N/A
BID_ASK	Time average bid	Max Ask	Min Bid	Time average ask	N/A
ADJUSTED_LAST	Dividend-adjusted first traded price	Dividend-adjusted high trade	Dividend-adjusted low trade	Dividend-adjusted last trade	Total traded volume
HISTORICAL_VOLATILITY	Starting volatility	Highest volatility	Lowest volatility	Last volatility	N/A
OPTION_IMPLIED_VOLATILITY	Starting implied volatility	Highest implied volatility	Lowest implied volatility	Last implied volatility	N/A
FEE_RATE	Starting fee rate	Highest fee rate	Lowest fee rate	Last fee rate	N/A
YIELD_BID	Starting bid yield	Highest bid yield	Lowest bid yield	Last bid yield	N/A
YIELD_ASK	Starting ask yield	Highest ask yield	Lowest ask yield	Last ask yield	N/A
YIELD_BID_ASK	Time average bid yield	Highest ask yield	Lowest bid yield	Time average ask yield	N/A
YIELD_LAST	Starting last yield	Highest last yield	Lowest last yield	Last last yield	N/A
SCHEDULE	N/A	N/A	N/A	N/A	N/A
AGGTRADES	First traded price	Highest traded price	Lowest traded price	Last traded price	Total traded volume

Figure 10.7: Valid data types to return for historical data

After sending the request, the method pauses execution for five seconds to allow time for the data to be retrieved and stored in the **`historical_data`** dictionary, keyed by **`request_id`**.

> *IMPORTANT NOTE*
>
> *The five-second delay is arbitrary and does not guarantee that all data has been received. You can extend this delay or implement more sophisticated methods to check whether data exists in the* **`historicData`** *method in the* **`IBWrapper`** *class.*

The method then determines how to parse the incoming time string depending on whether the requested data is intraday.

Next, the method retrieves the historical data and creates a pandas DataFrame from it. The DataFrame is structured with columns defined by **`TRADE_BAR_PROPERTIES`**. The index of the DataFrame is set to the time column, which is formatted, and the original time column is dropped. Additionally, a new **`symbol`** column is added to the DataFrame, containing the symbol of the contract, and **`request_id`** is stored as an attribute of the DataFrame.

Receiving historic data

The **`historicalData`** method is a callback and is automatically invoked for each bar of data received from IB following a call to **`reqHistoricalData`**. When **`reqHistoricalData`** is called to request historical market data, the IB API responds by sending back this data in individual bar units, with each bar triggering the **`historicalData`** method. This method then processes and stores each piece of data.

`historicalData` receives two parameters: **`request_id`** and **`bar`**. The **`request_id`** parameter is the unique identifier associated with the specific request for historical data, which allows the method to differentiate between data from different requests. The **`bar`** parameter is an object that contains the historical bar data for the instrument. Within the method, the bar data is structured into a tuple that contains the date (**`bar.date`**), the opening price (**`bar.open`**), the highest price (**`bar.high`**), the lowest price (**`bar.low`**), the closing price (**`bar.close`**), and the trading volume (**`bar.volume`**) for that specific time period.

The method then checks whether the **`request_id`** already exists as a key in the **`historical_data`** dictionary. If it does not exist the method initializes an empty list under this key. This step ensures that there is a structure in place to store the historical data for each unique request.

Finally, the method appends the **`bar_data`** tuple to the list associated with the **`request_id`** in the **`historical_data`** dictionary. This action effectively stores the historical bar data, allowing the method to accumulate the historical data points for each request over time.

The **`get_historical_data_for_many`** method loops through each provided contract, requests the historical data, and concatenates their resulting DataFrames together. The method returns a pivoted DataFrame, which moves the data in **`col_to_use`** into the rows and each requested contract into the columns.

There's more...

We now have the code built to request and receive historical market data. Open the **`app.py`** file and add the following code block under the definitions of the contracts:

```
data = app.get_historical_data(
    request_id=99,
    contract=aapl,
    duration='2 D',
    bar_size='30 secs'
)
```

The result of the changes is the following code in **`app.py`**:

```
import threading
import time
from wrapper import IBWrapper
from client import IBClient
from contract import stock, future, option
class IBApp(IBWrapper, IBClient):
    def __init__(self, ip, port, client_id):
        IBWrapper.__init__(self)
        IBClient.__init__(self, wrapper=self)
        self.connect(ip, port, client_id)
        thread = threading.Thread(target=self.run,
            daemon=True)
```

```
            thread.start()
            time.sleep(2)
    if __name__ == "__main__":
        app = IBApp("127.0.0.1", 7497, client_id=10)
        aapl = stock("AAPL", "SMART", "USD")
        gbl = future("GBL", "EUREX", "202403")
        pltr = option("PLTR", "BOX", "20240315", 20, "C")
        data = app.get_historical_data(
            request_id=99,
            contract=aapl,
            duration='2 D',
            bar_size='30 secs'
        )
        time.sleep(30)
        app.disconnect()
```

After running this code, **`data`** will contain a pandas DataFrame with the requested historic market data:

time	open	high	low	close	volume	symbol
2023-11-14 09:30:00-05:00	187.69	187.73	187.16	187.47	1168171.0	AAPL
2023-11-14 09:30:30-05:00	187.45	187.51	187.28	187.29	210746.0	AAPL
2023-11-14 09:31:00-05:00	187.28	187.48	187.21	187.23	229337.0	AAPL
2023-11-14 09:31:30-05:00	187.24	187.24	186.73	186.73	197331.0	AAPL
2023-11-14 09:32:00-05:00	186.74	186.90	186.69	186.83	163242.0	AAPL
...	...	...	...	...	...	...
2023-11-15 15:03:30-05:00	188.41	188.44	188.39	188.40	29656.0	AAPL
2023-11-15 15:04:00-05:00	188.40	188.42	188.34	188.34	15333.0	AAPL
2023-11-15 15:04:30-05:00	188.36	188.36	188.30	188.30	16014.0	AAPL
2023-11-15 15:05:00-05:00	188.31	188.34	188.28	188.34	35756.0	AAPL
2023-11-15 15:05:30-05:00	188.33	188.36	188.32	188.32	15280.0	AAPL

[1452 rows x 6 columns]

Figure 10.8: A pandas DataFrame containing requested historic market data

> *TIP*
>
> *If you're using an* **Interactive Development Environment** *(***IDE***) such as* **PyCharm** *or* **VSCode***, you can execute the code in debug mode during*

> *development. Debug mode lets you pause the execution of the code to inspect the variables.*

See also

To learn more about historical market data, see the documentation at this URL: **https://interactivebrokers.github.io/tws-api/historical_data.html**. For specifics on the following topics, see the associated documentation:

- Requesting historical market data: **https://interactivebrokers.github.io/tws-api/historical_bars.html#hd_request**
- Receiving historical market data: **https://interactivebrokers.github.io/tws-api/historical_bars.html#hd_receive**
- Duration, bar sizes, historical data types available, and available data per instrument: **https://interactivebrokers.github.io/tws-api/historical_bars.html#hd_duration**

Getting a market data snapshot

In the previous recipe, we learned how to get historical data. In some situations, we may need the current market price. In ***Chapter 12***, *Deploy Strategies to a Live Environment*, we'll use the current market price to create methods to target specific values or percentage allocations in our portfolio.

The API uses tick types, each representing a specific category of market data, such as last trade price, volume, or bid and ask. These tick types let us access real-time pricing information, which is important for making informed trading decisions. This recipe will show you how to get real-time market data.

Getting ready...

We assume you've created the **app.py**, **client.py**, and **wrapper.py** files in the **trading-app** directory. If not, do it now.

How to do it...

We'll update **app.py**, **client.py**, and **wrapper.py** to request the last price for a contract.

1. Open **client.py** and include the following method in the **IBClient** class after the **get_historical_data_for_many** method:

```
def get_market_data(self, request_id, contract,
    tick_type=4):
    self.reqMktData(
        reqId=request_id,
        contract=contract,
        genericTickList="",
        snapshot=True,
        regulatorySnapshot=False,
        mktDataOptions=[]
    )
    time.sleep(5)
    self.cancelMktData(reqId=request_id)
    return self.market_data[request_id].get(tick_type)
```

2. Open **wrapper.py** and include the following method in the **IBWrapper** class after the **historicalData** method:

```
def tickPrice(self, request_id, tick_type, price,
    attrib):
        if request_id not in self.market_data.keys():
            self.market_data[request_id] = {}
            self.market_data[request_id][tick_type] =
            float(price)
```

3. Add an instance variable in the **__init__** method under **self.historical_data = {}**:

```
self.market_data = {}
```

How it works…

The **`get_market_data`** method fetches a specific tick type from the IB API. It initiates a market data request for a given financial contract, which we identify by **`request_id`**, and requests all available ticks by using an empty string for the **`genericTickList`** argument. The function opts for a single data snapshot, rather than a continuous stream and pauses for five seconds to allow data reception and processing. After the pause, it cancels the data request to free up the request ID. Finally, the function retrieves and returns the specific market data from a dictionary using the request ID and tick type as keys. We default to tick type **`4`**, which is the last traded price.

In the **`tickPrice`** callback, it first checks whether the **`request_id`** exists in the **`market_data`** dictionary and if not, it initializes an empty dictionary for that ID. It then updates the dictionary, setting the **`tick_type`** key to the received **`price`**, converted to a float.

The **`tickPrice`** method will typically retrieve seven different tick types:

Code	Tick Name	Description
1	Bid Price	Highest priced bid for the contract.
2	Ask Price	Lowest price offer on the contract.
4	Last Price	Last price at which the contract traded.
6	High	High price for the day.
7	Low	Low price for the day.
9	Close Price	The last available closing price for the *previous* day.
14	Open Tick	Current session's opening price .

Figure 10.9: Different tick types returned by the IB API

There's more…

To get the last closing price for AAPL, add the following code to **`app.py`** after we define our AAPL contract:

```
data = app.get_market_data(
    request_id=99,
```

```
        contract=aapl
    )
```

The `data` variable contains a float representing the last traded price of AAPL.

See also

To learn more about how to request and receive market data using the IB API, see the following resources:

- A list of different tick types: https://interactivebrokers.github.io/tws-api/tick_types.html
- Requesting market data: https://interactivebrokers.github.io/tws-api/md_request.html
- Receiving market data: https://interactivebrokers.github.io/tws-api/md_receive.html

Streaming live market data

Requesting tick-by-tick data involves a real-time, granular approach to market data acquisition. This process begins by invoking the `reqTickByTickData` method, where we specify the unique request identifier, the financial instrument's contract details, and the type of tick data we are interested in (such as `BidAsk`). Upon this request, the IB API starts transmitting data for each individual market **tick**, which is a single change or update in market data.

Unlike bulk historical data retrieval, this method provides data almost instantaneously as market events occur, which lets us build near real-time algorithmic trading applications. The received data is handled through a callback function that is triggered for each tick, capturing detailed information such as price, size, and the time of the tick. By the end of this receipe, you'll be able to stream near-real-time market data from the IB API.

Getting ready...

We assume you've created the **app.py**, **client.py**, and **wrapper.py** files in the **trading-app** directory. If not, do it now.

How to do it...

We'll update **app.py**, **client.py**, and **wrapper.py** to request historic market data.

1. Open **client.py** and include the following import at the top of the file:

```
from dataclasses import dataclass, field
```

2. Then add a **dataclass** to represent each price tick below the **TRADE_BAR_PROPERTIES** constant:

```
@dataclass
class Tick:
    time: int
    bid_price: float
    ask_price: float
    bid_size: float
    ask_size: float
    timestamp_: pd.Timestamp = field(init=False)
    def __post_init__(self):
        self.timestamp_ = pd.to_datetime(self.time,
            unit="s")
        self.bid_price = float(self.bid_price)
        self.ask_price = float(self.ask_price)
        self.bid_size = int(self.bid_size)
        self.ask_size = int(self.ask_size)
```

3. Inside the **IBClient** class in the **client.py** file, add the functions that will start and stop the streaming data:

```
def get_streaming_data(self, request_id, contract):
    self.reqTickByTickData(
        reqId=request_id,
        contract=contract,
```

```
            tickType="BidAsk",
            numberOfTicks=0,
            ignoreSize=True
        )
        time.sleep(10)
        while True:
            if self.stream_event.is_set():
                yield Tick(
                    *self.streaming_data[request_id])
                self.stream_event.clear()
    def stop_streaming_data(self, request_id):
        self.cancelTickByTickData(reqId=request_id)
```

The result of the changes is the following code in **`client.py`**:

```
import time
import pandas as pd
from dataclasses import dataclass, field
from ibapi.client import EClient
TRADE_BAR_PROPERTIES = ["time", "open", "high", "low",
    "close", "volume"]
@dataclass
class Tick:
    time: int
    bid_price: float
    ask_price: float
    bid_size: float
    ask_size: float
    timestamp_: pd.Timestamp = field(init=False)
    def __post_init__(self):
        self.timestamp_ = pd.to_datetime(self.time,
            unit="s")
        self.bid_price = float(self.bid_price)
        self.ask_price = float(self.ask_price)
        self.bid_size = int(self.bid_size)
        self.ask_size = int(self.ask_size)
class IBClient(EClient):
    def __init__(self, wrapper):
        EClient.__init__(self, wrapper)
    <snip>
    def get_streaming_data(self, request_id, contract):
        self.reqTickByTickData(
```

```
            reqId=request_id,
            contract=contract,
            tickType="BidAsk",
            numberOfTicks=0,
            ignoreSize=True
        )
        time.sleep(10)
        while True:
            if self.stream_event.is_set():
                yield Tick(
                    *self.streaming_data[request_id])
                self.stream_event.clear()
    def stop_streaming_data(self, request_id):
        self.cancelTickByTickData(reqId=request_id)
```

4. Open **`wrapper.py`** and include the following import at the top of the file:

```
import threading
```

5. Add the following lines of code to the **`__init__`** method in the **`IBWrapper`** class:

```
self.streaming_data = {}
self.stream_event = threading.Event()
```

6. Next, add the following method to the **`IBWrapper`** class:

```
def tickByTickBidAsk(
        self,
        request_id,
        time,
        bid_price,
        ask_price,
        bid_size,
        ask_size,
        tick_atrrib_last
):
    tick_data = (
        time,
        bid_price,
        ask_price,
        bid_size,
```

```
            ask_size,
        )
        self.streaming_data[request_id] = tick_data
        self.stream_event.set()
```

The result of the changes is the following code in **`wrapper.py`**:

```
import threading
from ibapi.wrapper import EWrapper
class IBWrapper(EWrapper):
    def __init__(self):
        EWrapper.__init__(self)
        self.nextValidOrderId = None
        self.historical_data = {}
        self.streaming_data = {}
        self.stream_event = threading.Event()
    <snip>
    def tickByTickBidAsk(
            self,
            request_id,
            time,
            bid_price,
            ask_price,
            bid_size,
            ask_size,
            tick_atrrib_last
    ):
        tick_data = (
            time,
            bid_price,
            ask_price,
            bid_size,
            ask_size,
        )
        self.streaming_data[request_id] = tick_data
        self.stream_event.set()
```

How it works…

Getting streaming data through the IB API requires two steps: first we request it, then we process it through a callback. The tick-by-tick data

we stream is the data that corresponds to the data shown in the TWS Time and Sales window.

> *IMPORTANT NOTE*
>
> *Time and Sales refers to a real-time data feed that provides detailed information about executed trades for a specific financial instrument. This feed includes the exact time of each trade, the price at which it was executed, and the size of the trade. Time and Sales data is useful for traders who want to analyze order book dynamics in real time.*

Requesting streaming data

We start by defining a Python **dataclass** called `Tick`. A Python **dataclass** is a decorator that automatically generates special methods such as `__init__`, `__repr__`, and `__eq__` for classes, primarily used for storing data, simplifying, and reducing boilerplate code.

The `Tick` **dataclass** represents market data ticks with attributes for time, bid and ask prices, and sizes. An additional attribute, `timestamp_`, is defined as a pandas `Timestamp` and excluded from automatic `__init__` method generation using `field(init=False)`. The `__post_init__` method of the **dataclass** converts the `time` attribute from a Unix timestamp to a more readable pandas `Timestamp` format, using `to_datetime`.

The `get_streaming_data` method requires a request ID and a contract to start streaming data. The method requests and yields real-time bid and ask tick data for a specified financial instrument, represented by the `contract` parameter.

The method begins by calling `reqTickByTickData`, a function of the IB API that requests real-time tick-by-tick data. The parameters passed to this function include the following:

- `reqId`: A unique identifier for the data request.

- **`contract`**: Specifies the financial instrument for which data is being requested.
- **`tickType`**: Specifies the type of data to stream. Available options are **`Last`**, **`AllLast`**, **`BidAsk`**, and **`MidPoint`**.
- **`numberOfTicks`**: A flag to request a specific set of historic ticks or to stream until canceled.
- **`ignoreSize`**: Whether updates to bid and ask sizes are required.

Following the data request, we delay execution for 10 seconds to ensure that some data is received and buffered before the method starts yielding data.

The method then enters an infinite loop, continuously checking if the thread event **`stream_event`** is set and when it is, the method yields the **`Tick`** object with the latest data for the given **`request_id`**, and then clears the event, readying it for the next batch of data.

> *IMPORTANT NOTE*
>
> *A Python generator is a special type of function that returns an iterator, allowing for the generation of items on the fly rather than storing them all in memory at once. When a generator function is called, it doesn't execute its code immediately. Instead, it returns a generator object that can be iterated over. Each iteration over a generator object resumes the function's execution from where it last yielded a value, using the* **`yield`** *statement, until it either encounters another* **`yield`** *or reaches the end of the function, at which point it raises a* **`StopIteration`** *exception.*

Receiving streaming data

Depending on the **`tickType`** requested in the **`reqTickByTickData`**, IB will use a different callback function. They are as follows:

tickType	Callback
Last	tickByTickAllLast
AllLast	tickByTickAllLast
BidAsk	tickByTickBidAsk
MidPoint	tickByTickMidPoint

Figure 10.10: Callbacks used for each tick type requested

> *IMPORTANT NOTE*
>
> *We only implement* **`tickByTickBidAsk`** *to respond to the* **`BidAsk`** *tick type. You can read more about the type of data that is returned in each of the callbacks at this URL:* **https://interactivebrokers.github.io/tws-api/tick_data.html**

We override the **`tickByTickBidAsk`** provided by the IB API. This callback method is invoked for every tick. Upon invocation, the method first constructs a tuple, **`tick_data`**, comprising the time, bid price, ask price, bid size, and ask size.

After receiving a tick, the method constructs a **`tick_data`** tuple with the bid and ask prices and respective volumes. The method then updates a dictionary **`streaming_data`**, keyed by **`request_id`**, with this tick data and sets a **`stream_event`** threading event, signaling that new data has been received and is ready for processing in the **`get_streaming_data`** method.

There's more...

We now have the code built to request and receive streaming tick data. Open the **`app.py`** file and add the following code block under the definitions of the contracts:

```
eur = future("EUR", "CME", "202312")
for tick in app.get_streaming_data(99, es):
    print(tick)
```

The result of the changes is the following code in **`app.py`**:

```
import threading
import time
from wrapper import IBWrapper
from client import IBClient
from contract import future
class IBApp(IBWrapper, IBClient):
    def __init__(self, ip, port, client_id):
        IBWrapper.__init__(self)
        IBClient.__init__(self, wrapper=self)
        self.connect(ip, port, client_id)
        thread = threading.Thread(target=self.run,
            daemon=True)
        thread.start()
        time.sleep(2)
if __name__ == "__main__":
    app = IBApp("127.0.0.1", 7497, client_id=10)
    eur = future("EUR", "CME", "202312")
    for tick in app.get_streaming_data(99, eur):
        print(tick)
    time.sleep(30)
    app.disconnect()
```

After running this code, you'll see a list of **`Tick`** objects printed to the console with the details of each tick:

```
Tick(time=1700522344, bid_price=1.0956, ask_price=1.09565, bid_size=Decimal('2'), ask_size=Decimal('38'), timestamp_=Timestamp('2023-11-20 23:19:04'))
Tick(time=1700522353, bid_price=1.09555, ask_price=1.09565, bid_size=Decimal('63'), ask_size=Decimal('22'), timestamp_=Timestamp('2023-11-20 23:19:13'))
Tick(time=1700522355, bid_price=1.0956, ask_price=1.09565, bid_size=Decimal('2'), ask_size=Decimal('35'), timestamp_=Timestamp('2023-11-20 23:19:15'))
Tick(time=1700522361, bid_price=1.09555, ask_price=1.09565, bid_size=Decimal('55'), ask_size=Decimal('32'), timestamp_=Timestamp('2023-11-20 23:19:21'))
Tick(time=1700522361, bid_price=1.0956, ask_price=1.09565, bid_size=Decimal('2'), ask_size=Decimal('32'), timestamp_=Timestamp('2023-11-20 23:19:21'))
Tick(time=1700522400, bid_price=1.09555, ask_price=1.09565, bid_size=Decimal('57'), ask_size=Decimal('26'), timestamp_=Timestamp('2023-11-20 23:20:00'))
Tick(time=1700522403, bid_price=1.0956, ask_price=1.09565, bid_size=Decimal('4'), ask_size=Decimal('29'), timestamp_=Timestamp('2023-11-20 23:20:03'))
Tick(time=1700522410, bid_price=1.09555, ask_price=1.09565, bid_size=Decimal('71'), ask_size=Decimal('32'), timestamp_=Timestamp('2023-11-20 23:20:10'))
Tick(time=1700522416, bid_price=1.0956, ask_price=1.09565, bid_size=Decimal('4'), ask_size=Decimal('33'), timestamp_=Timestamp('2023-11-20 23:20:16'))
```

Figure 10.11: Streaming tick data as Tick objects

See also

To learn more about the available types of streaming data, check the following URL: **https://interactivebrokers.github.io/tws-api/market_data.html**. For specifics, check the following URLs:

- Level 1 (top of book) streaming and snapshot data. This is another option for streaming real-time data that was not covered in this

recipe: **https://interactivebrokers.github.io/tws-api/md_request.html**.

- Streaming tick data covered in this recipe: **https://interactivebrokers.github.io/tws-api/tick_data.html**.

Storing live tick data in a local SQL database

We discussed storing financial market data in ***Chapter 4***, *Store Financial Market Data on Your Computer*. In this chapter, we learned several ways of downloading different types of market data for free and storing it in a variety of formats. In ***Chapter 13***, *Advanced Recipes for Market Data and Strategy Management*, we'll look at how to use the cutting-edge ArcticDB library to store petabytes of market data. For now, we'll extend the recipes in this chapter and the recipes in ***Chapter 4*** to store the streaming tick data in a local SQLite database.

Getting ready...

We assume you've read the *Storing Data On-Disk with SQLite* recipe in ***Chapter 4***, *Store Financial Market Data on Your Computer*. If you haven't, please review it now. We also assume you've created the **`app.py`**, **`client.py`**, **`wrapper.py`** , and **`utils.py`** files in the **`trading-app`** directory. If not, do it now.

We'll also move some of the code currently in **`client.py`**, to the **`utils.py`** file. Copy the following code from **`client.py`** and paste it into **`utils.py`**:

```
import pandas as pd
from dataclasses import dataclass, field
TRADE_BAR_PROPERTIES = ["time", "open", "high", "low",
    "close", "volume"]
@dataclass
class Tick:
```

```
    time: int
    bid_price: float
    ask_price: float
    bid_size: float
    ask_size: float
    timestamp_: pd.Timestamp = field(init=False)
    def __post_init__(self):
        self.timestamp_ = pd.to_datetime(self.time,
            unit="s")
        self.bid_price = float(self.bid_price)
        self.ask_price = float(self.ask_price)
        self.bid_size = int(self.bid_size)
        self.ask_size = int(self.ask_size)
```

Now, remove the code from the **`client.py`** file and import it from **`utils.py`**. The result of the changes is the following code in **`client.py`**:

```
import time
import pandas as pd
from utils import Tick, TRADE_BAR_PROPERTIES
from ibapi.client import EClient
class IBClient(EClient):
    <snip>
```

How to do it...

We'll add methods to our **`IBApp`** class that will create the SQLite database, generate a connection to our database, and insert tick data into the table.

1. Add the **`connection`** method to the **`IBApp`** class in **`app.py`** and decorate it with the **`property`** decorator:

   ```
   @property
   def connection(self):
       return sqlite3.connect("tick_data.sqlite",
           isolation_level=None)
   ```

2. Add the **`create_table`** method to the **`IBApp`** class:

```
def create_table(self):
    cursor = self.connection.cursor()
    cursor.execute(
    "CREATE TABLE IF NOT EXISTS bid_ask_data (
        timestamp datetime, symbol string,
        bid_price real, ask_price real,
        bid_size integer, ask_size integer)")
```

3. Add the **`stream_to_sqlite`** method to the **`IBApp`** class. Don't worry, we'll go through it in detail next:

```
def stream_to_sqlite(self, request_id, contract,
    run_for_in_seconds=23400):
        cursor = self.connection.cursor()
        end_time = time.time() + run_for_in_seconds + 10
        for tick in app.get_streaming_data(request_id,
            contract):
                query = "INSERT INTO bid_ask_data (
                    timestamp, symbol, bid_price,
                    ask_price, bid_size, ask_size)
                    VALUES (?, ?, ?, ?, ?, ?)"
                values = (
                tick.timestamp_.strftime(
                   "%Y-%m-%d %H:%M:%S"),
                contract.symbol,
                tick.bid_price,
                tick.ask_price,
                tick.bid_size,
                tick.ask_size
            )
            cursor.execute(query, values)
            if time.time() >= end_time:
                break
        self.stop_streaming_data(request_id)
```

4. In the **`__init__`** method of the **`IBApp`** class, add the following line under the calls to **`IBWrapper`** and **`IBClient`**:

```
self.create_table()
```

The result of the changes is the following code in **`IBApp`**:

```
class IBApp(IBWrapper, IBClient):
    def __init__(self, ip, port, client_id):
        IBWrapper.__init__(self)
        IBClient.__init__(self, wrapper=self)
        self.create_table()
        self.connect(ip, port, client_id)
        thread = threading.Thread(target=self.run,
            daemon=True)
        thread.start()
        time.sleep(2)
    @property
    def connection(self):
        return sqlite3.connect("tick_data.sqlite",
            isolation_level=None)
    def create_table(self):
        cursor = self.connection.cursor()
        cursor.execute(
            "CREATE TABLE IF NOT EXISTS bid_ask_data (
                timestamp datetime, symbol string,
                bid_price real, ask_price real,
                bid_size integer, ask_size integer)")
    def stream_to_sqlite(self, request_id, contract,
        run_for_in_seconds=23400):
            cursor = self.connection.cursor()
            end_time = time.time() + run_for_in_seconds + 10
        for tick in app.get_streaming_data(request_id,
            contract):
                query = "INSERT INTO bid_ask_data (
                    timestamp, symbol, bid_price,
                    ask_price, bid_size, ask_size) VALUES (
                        ?, ?, ?, ?, ?, ?)"
                values = (
                    tick.timestamp_.strftime(
                        "%Y-%m-%d %H:%M:%S"),
                    contract.symbol,
                    tick.bid_price,
                    tick.ask_price,
                    tick.bid_size,
                    tick.ask_size
                )
```

```
                cursor.execute(query, values)
                if time.time() >= end_time:
                    break
            self.stop_streaming_data(request_id)
```

How it works...

We first create a class property called **`connection`**, which establishes and returns a connection to our SQLite database. The **`@property`** decorator is used to create a property, allowing the **`connection`** method to be accessed like an attribute without the need to call it as a method.

When the **`connection`** property is accessed, it creates a connection to an SQLite database file named **`tick_data.sqlite`**. The **`isolation_level=None`** parameter sets the transaction isolation level to **`None`**, which means that the **autocommit** mode is enabled. In **autocommit** mode, changes to the database are committed immediately after each statement, without needing to explicitly call **`commit`** after each database operation.

The **`create_table`** method creates a database table named **`bid_ask_data`**. When this method is called, it first establishes a database cursor, which is an intermediary for executing database commands. The code then executes a **`create`** SQL command using the **`execute`** method, which creates the table if it does not exist. When **`IBApp`** is initialized, we call **`self.create_table`**, which creates the table if it does not exist.

The **`stream_to_sqlite`** method stores the tick data in our SQLite database for a specified duration. It starts by creating a database cursor from the established SQLite connection. The method calculates an **`end_time`** value by adding the specified **`run_for_in_seconds`** duration (defaulting to 23,400 seconds, or 6.5 hours) to the current time, plus an additional 10 second buffer which makes up for the time we wait for the tick stream to start. It then enters a loop, retrieving streaming data

from the **`get_streaming_data`** method, which yields **`Tick`** objects containing market data for the given **`request_id`** and **`contract`**. For each tick received, the method constructs a SQL insert query to add the tick data into the **`bid_ask_data`** table, formatting the timestamp and including relevant data such as the contract's symbol, bid price, ask price, bid size, and ask size. The loop continues until the current time exceeds the calculated **`end_time`**, at which point it breaks, limiting the data streaming to the specified duration.

There's more...

We now have the code built to store streaming tick data. Add the following code block under the definitions of the contracts:

```
es = future("ES", "CME", "202312")
app.stream_to_sqlite(99, es, run_for_in_seconds=30)
```

The result of the changes is the following code in **`app.py`**:

```
<snip>
if __name__ == "__main__":
    app = IBApp("127.0.0.1", 7497, client_id=10)
    es = future("ES", "CME", "202312")
    app.stream_to_sqlite(99, es, run_for_in_seconds=30)
    app.disconnect()
```

We can inspect the data stored in our SQLite database with DB Browser for SQLite. It's a free tool that gives us a graphical interface to inspect data in SQLite databases. You can download it here: https://sqlitebrowser.org/. Once installed, open up the **`tick_data.sqlite`** file and navigate to the **Browse Data** tab and you'll see the tick data.

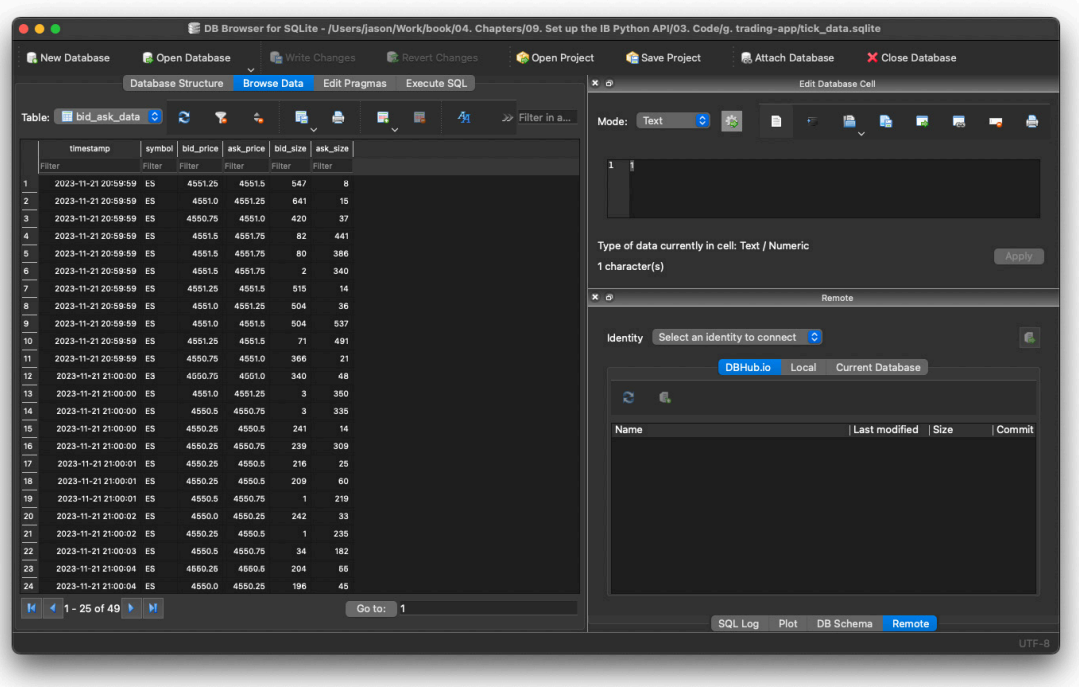

Figure 10.12: Browsing bid ask tick data in our SQLite database

See also

The SQLite documentation was covered in ***Chapter 4***, *Store Financial Market Data on Your Computer*, but if you're interested in more details about the **`Connection`** and **`Cursor`** objects, you can check the following URLs:

- Documentation for the SQLite **`connect`** method: **https://docs.python.org/3/library/sqlite3.html#sqlite3.connect**
- Documentation for the SQLite **`Cursor`** object: **https://docs.python.org/3/library/sqlite3.html#cursor-objects**