



11

Manage Orders, Positions, and Portfolios with the IB API

In algorithmic trading, efficient management of orders, positions, and portfolio data is critical. Luckily for us, we can do it all using Python. Managing orders encompasses a range of activities, including executing new trades, canceling existing orders, and updating orders to adapt to changing market conditions or shifts in trading strategies. Managing positions involves monitoring and analyzing live position data to track **profit and loss (PnL)** in real time. This immediate insight into the performance of individual trades enables traders to make informed decisions on whether to hold, sell, or adjust positions. Further, real-time (or near real-time) portfolio data can generate real-time (or near real-time) risk statistics to improve overall risk management. Portfolio data management involves a comprehensive analysis of the portfolio to assess its performance, understand risk exposure, and make strategic adjustments for optimizing returns. This is especially true when trading stocks on margin or futures where there is a financial cost and opportunity cost in holding losing positions.

As we've seen, the IB API uses a consistent request-callback pattern that can be used in several aspects of our trading app, including order management, position management, and accessing portfolio details. This pattern allows for the initiation of a request, such as placing or modifying an order, retrieving current position data, or gathering portfolio information, followed by a callback function that handles the

response. We'll use this pattern throughout the chapter. By the end of this chapter, we'll be able to submit and modify orders to IB through Python, obtain key portfolio details including information on liquidity, margin requirements, and open positions, and compute our portfolio's profit or loss.

In this chapter, we present the following recipes:

- Executing orders with the IB API
- Managing orders once they're placed
- Getting details about your portfolio
- Inspecting positions and position details
- Computing portfolio profit and loss

Executing orders with the IB API

In [*Chapter 10, Set Up the Interactive Brokers Python API*](#), we created **contract** and **order** objects. Using these, we can use the IB API to execute trades. But before we can execute trades, we have to understand the concept of the next order ID.

The next order ID (**nextValidOrderId**) is a unique identifier for each order. Since up to 32 instances of a trading app can run in parallel, this identifier makes sure individual orders are traceable within the trading system. **nextValidOrderId** is used to preserve order integrity and prevent overlap between multiple orders submitted simultaneously or in rapid succession. When our trading app connects to the IB API, it receives an integer variable called **nextValidOrderId** from the server that is unique to each client connection to TWS. This ID must be used for the first order submission. Subsequently, we are responsible for incrementing this identifier for each new order.

Getting ready

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

How to do it...

First, we'll add the code to deal with the integer **nextValidOrderId** in the **wrapper.py** file and print out the details of the order's execution:

1. Add an instance variable, **nextValidOrderId**, in the **__init__** method of our **IBWrapper** class:

```
self.nextValidOrderId = None
```

2. Add an implementation of the **nextValidId** method overriding the method from the inherited **EWrapper** class in our **IBWrapper** class:

```
def nextValidId(self, order_id):  
    super().nextValidId(order_id)  
    self.nextValidOrderId = order_id
```

3. Add an implementation of the **orderStatus** method overriding the method from the inherited **EWrapper** class:

```
def orderStatus(  
    self,  
    order_id,  
    status,  
    filled,  
    remaining,  
    avg_fill_price,  
    perm_id,  
    parent_id,  
    last_fill_price,  
    client_id,  
    why_held,  
    mkt_cap_price,  
):  
    print(  
        "orderStatus - orderid:",  
        order_id,
```

```

        "status:",
        status,
        "filled",
        filled,
        "remaining",
        remaining,
        "lastFillPrice",
        last_fill_price,
    )

```

4. Add an implementation of the **openOrder** method overriding the method from the inherited **EWrapper** class:

```

def openOrder(self, order_id, contract, order,
               order_state):
    print(
        "openOrder id:",
        order_id,
        contract.symbol,
        contract.secType,
        "@",
        contract.exchange,
        ":",
        order.action,
        order.orderType,
        order.totalQuantity,
        order_state.status,
    )

```

5. Add an implementation of the **execDetails** method overriding the method from the inherited **EWrapper** class:

```

def execDetails(self, request_id, contract, execution):
    print(
        "Order Executed: ",
        request_id,
        contract.symbol,
        contract.secType,
        contract.currency,
        execution.execId,
        execution.orderId,
        execution.shares,
    )

```

```
        execution.lastLiquidity,
    )
```

6. Now, open the **client.py** file. Here, we'll add a custom method, **send_order**, under the **__init__** method that accepts a **contract** object and **order** object, increments the **nextValidOrderId** variable, and sends the order to the exchange:

```
def send_order(self, contract, order):
    order_id = self.wrapper.nextValidOrderId
    self.placeOrder(orderId=order_id,
                    contract=contract, order=order)
    self.reqIds(-1)
    return order_id
```

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

```
import time
import pandas as pd
from utils import Tick, TRADE_BAR_PROPERTIES
from ibapi.client import EClient
class IBClient(EClient):
    def __init__(self, wrapper):
        EClient.__init__(self, wrapper)
    def send_order(self, contract, order):
        order_id = self.wrapper.nextValidOrderId
        self.placeOrder(orderId=order_id,
                        contract=contract, order=order)
        self.reqIds(-1)
        return order_id
<snip>
```

How it works...

First, we'll cover the overridden methods in the **IBWrapper** class.

When orders are submitted through the **placeOrder** method from the **IBClient** class, **orderStatus**, **openOrder**, **execDetails** from the

IBWrapper class are invoked depending on the life cycle of the order. They accomplish the following:

- **orderStatus** receives updates about the status of submitted orders
- **openOrder** provides information about orders that have been submitted but not fully executed
- **execDetails** provides detailed information about the execution of an order

Each of these callbacks accepts parameters that are passed from the IB API. In this recipe, we just print out the information. In more sophisticated applications, we can use the events to trigger risk analysis, portfolio updates, or alerts.

When **send_order** is called, it takes a **contract** object and an **order** object. The **contract** object represents the financial instrument to be ordered, while the **order** object contains the details of the order. The first line within the method, **order_id = self.wrapper.nextValidOrderId**, retrieves the next valid order ID from the **IBWrapper** object. The next line, **self.placeOrder(orderId=order_id, contract=contract, order=order)**, is a call to the **placeOrder** method of the IB API. This method is responsible for placing the order with the given **order_id**, **contract**, and **order** details on the IB server. The subsequent call, **reqIds(-1)**, is a request to the server to increment the internal counter for the next valid order ID.

There's more...

We now have everything in place to send orders to IB through the API using Python. To send an order for execution, import our **order** type and the **BUY** constant at the top of the **app.py** file:

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

Then, add the following code after the line that defines the trading app:

```
limit_order = limit(BUY, 100, 190.00)
app.send_order(aapl, limit_order)
```

This uses the **contract** object we set up in the *Creating a Contract object with the IB API* and the **order** object we set up in the *Creating an Order object with the IB API* recipes from [Chapter 9](#).

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

```
import threading
import time
import sqlite3
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.create_table()
        self.connect(ip, port, client_id)
        thread = threading.Thread(target=self.run,
                                   daemon=True)
        thread.start()
        time.sleep(2)
    <snip>
if __name__ == "__main__":
    app = IBApp("127.0.0.1", 7497, client_id=11)
    aapl = stock("AAPL", "SMART", "USD")
    gbl = future("GBL", "EUREX", "202403")
    pltr = option("PLTR", "BOX", "20240315", 20, "C")
    limit_order = limit(BUY, 100, 190.00)
    app.send_order(aapl, limit_order)
```

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

After running this code, you will see a series of messages in the terminal. These messages are a result of the **orderStatus**, **openOrder**, and **execDetails** callbacks being called after the **placeOrder** method was invoked:

```
openOrder id: 9 AAPL STK @ SMART : BUY LMT 100 PreSubmitted
orderStatus - orderid: 9 status: PreSubmitted filled 0 remaining 100 lastFillPrice 0.0
Order Executed: -1 AAPL STK USD 0000e0d5.6554a46e.01.01 9 100 2
openOrder id: 9 AAPL STK @ SMART : BUY LMT 100 Filled
orderStatus - orderid: 9 status: Filled filled 100 remaining 0 lastFillPrice 188.66
openOrder id: 9 AAPL STK @ SMART : BUY LMT 100 Filled
orderStatus - orderid: 9 status: Filled filled 100 remaining 0 lastFillPrice 188.66
```

Figure 11.1: Message indicating the limit order was executed

IMPORTANT NOTE

In this example, we enter a buy limit order at a price above the current ask, which, at the time of writing, was \$188.66. This means that even though the order is a limit order, it will be executed immediately. This is done only for demonstration and testing purposes. In most scenarios, we'd enter a limit order under the best bid to wait for our desired price.

See also

To learn more about the famous **nextValidOrderId** and details about order execution, see the documentation here:

https://interactivebrokers.github.io/tws-api/order_submission.html. This URL details the **openOrder** and **orderStatus** callbacks.

Managing orders once they're placed

Effective order management is important and typically involves canceling or updating existing orders. Canceling orders is straightforward: we may enter a limit or stop loss order that we no longer want. Market conditions change or our strategy indicates a different entry or exit position. In these cases, we'll use the IB API to completely cancel the order.

On the other hand, we may want the order to remain in the order book but with different attributes. Traders frequently update orders to change the quantity being traded, which allows them to scale into positions in response to market analysis or risk management requirements. Adjusting limit prices is another common update, which lets us set a new maximum purchase price or minimum sale price, depending on market conditions. Similarly, modifying stop prices is a strategic move to manage potential losses or lock in profits, especially in volatile markets.

We can update existing orders using the IB API by calling the **placeOrder** method with the same fields as the open order, except with the parameter to modify. This includes the order's ID, which must match the existing open order. IB recommends only changing order price, size, and **time in force**. Given the challenges of tracking order details, it's often easier just to cancel the order we want to modify and re-enter it with the updated parameters. That's the approach we'll take in this recipe.

Getting ready

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

How to do it...

We'll add three new methods to our **client.py** file to manage orders:

1. Add the **cancel_all_orders** method to our **IBClient** class directly under the **__init__** method:

```
def cancel_all_orders(self):
    self.reqGlobalCancel()
```

2. Add the **cancel_order_by_id** method next:

```
def cancel_order_by_id(self, order_id):
    self.cancelOrder(orderId=order_id,
                      manualCancelOrderTime="")
```

3. Finally, add **update_order**:

```
def update_order(self, contract, order, order_id):
    self.cancel_order_by_id(order_id)
    return self.send_order(contract, order)
```

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

```
<snip>
class IBClient(EClient):
    def __init__(self, wrapper):
        EClient.__init__(self, wrapper)
    def cancel_all_orders(self):
        self.reqGlobalCancel()
    def cancel_order_by_id(self, order_id):
        self.cancelOrder(orderId=order_id,
                          manualCancelOrderTime="")
    def update_order(self, contract, order, order_id):
        self.cancel_order_by_id(order_id)
        return self.send_order(contract, order)
<snip>
```

How it works...

We start by creating a function to cancel all open orders. The **cancel_all_orders** method executes the **reqGlobalCancel** method, which is a command to cancel all open orders placed through the cur-

rent session, ensuring that no pending orders remain active in the trading system.

IMPORTANT NOTE

*Calling **cancel_all_orders** will cancel all open orders, regardless of how they were originally placed. That means it will cancel orders manually entered through TWS in addition to those entered through the IB API.*

To cancel a single order, we use **cancel_order_by_id**, which cancels a specific order identified by its integer **order_id**. When invoked, it calls the **cancelOrder** method, passing the integer **order_id** as an argument, along with an empty string for **manualCancelOrderTime**. The **send_order** method returns the **order_id** that is used to cancel the order.

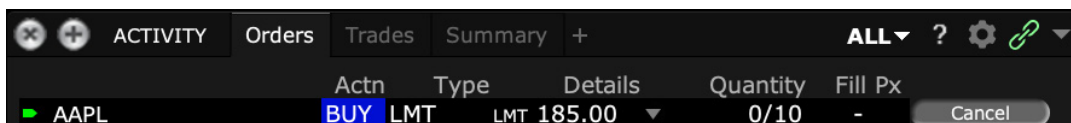
To update orders, we combine two methods. The **update_order** method first cancels an existing order identified by **order_id** using the **cancel_order_by_id** method. It then creates and sends a new order with the specified **contract** and **order** details using the **send_order** method, effectively updating the original order by replacing it with a new one. This is the recommended method of updating orders using the IB API.

There's more...

Let's test out our new method. In the **app.py** file, add the following code after the line that defines our AAPL (jas rev. 2024-07-03):

```
order_1 = limit(BUY, 10, 185.0)
order_1_id = app.send_order(aapl, order_1)
```

Once you run this code, you'll see an order in the **Orders** section of TWS:



The screenshot shows the 'Orders' tab in the TWS interface. A single order is listed for AAPL, with a green status icon. The order details are: Actn: BUY, Type: LMT, Details: LMT 185.00, Quantity: 0/10, Fill Px: -. A 'Cancel' button is visible next to the order.

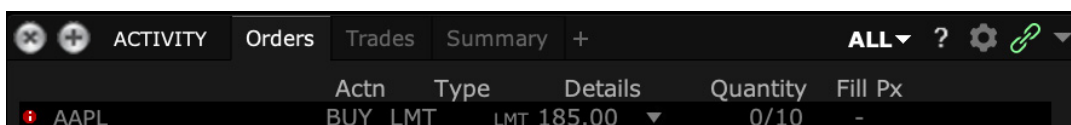
	Actn	Type	Details	Quantity	Fill Px
AAPL	BUY	LMT	LMT 185.00	0/10	-

Figure 11.2: Our AAPL limit order safely resting off the market

To cancel the order, run the following code:

```
app.cancel_order_by_id(order_1_id)
```

You'll now see our order canceled:



The screenshot shows the 'Orders' tab in the TWS interface. The order for AAPL now has a red status icon, indicating it is canceled. The order details remain: Actn: BUY, Type: LMT, Details: LMT 185.00, Quantity: 0/10, Fill Px: -. The 'Cancel' button is no longer visible.

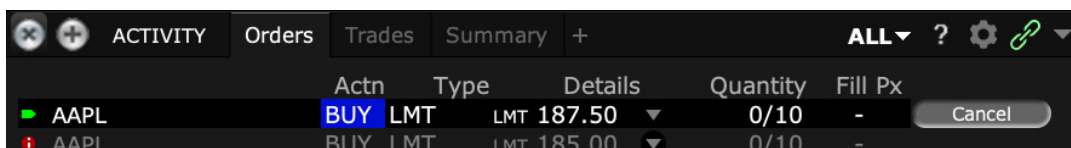
	Actn	Type	Details	Quantity	Fill Px
AAPL	BUY	LMT	LMT 185.00	0/10	-

Figure 11.3: Our AAPL limit order is canceled

Let's reenter the order, create a second order with a different limit price, and update it:

```
order_2 = limit(BUY, 10, 187.50)
app.update_order(aapl, order_2, order_1_id)
```

Our original order is canceled, and the new one is pending:



The screenshot shows the 'Orders' tab in the TWS interface. There are now two orders listed for AAPL. The top order is new, with a green status icon: Actn: BUY, Type: LMT, Details: LMT 187.50, Quantity: 0/10, Fill Px: -. The bottom order is the original one, with a red status icon: Actn: BUY, Type: LMT, Details: LMT 185.00, Quantity: 0/10, Fill Px: -. A 'Cancel' button is visible next to the new order.

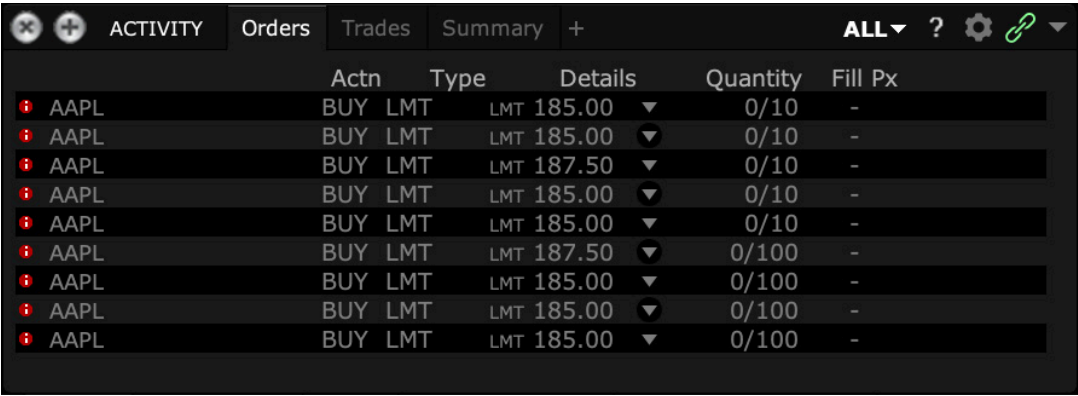
	Actn	Type	Details	Quantity	Fill Px
AAPL	BUY	LMT	LMT 187.50	0/10	-
AAPL	BUY	LMT	LMT 185.00	0/10	-

Figure 11.4: Our original AAPL order is canceled and our new AAPL order is entered

Finally, cancel all open orders:

```
app.cancel_all_orders()
```

As a result, all open orders are canceled:



The screenshot shows the 'Orders' tab in the TWS interface. It displays a table of open orders, all of which are marked as 'CANCELED' with a red 'X' icon. The table has columns for 'Actn', 'Type', 'Details', 'Quantity', and 'Fill Px'. The orders are for AAPL, all with a quantity of 100 and a price of 185.00. The 'Details' column shows 'LMT' (Limit) and 'LMT 185.00'. The 'Quantity' column shows '0/100', indicating that the orders have not been filled.

Actn	Type	Details	Quantity	Fill Px
BUY	LMT	LMT 185.00	0/100	-
BUY	LMT	LMT 185.00	0/100	-
BUY	LMT	LMT 187.50	0/100	-
BUY	LMT	LMT 185.00	0/100	-
BUY	LMT	LMT 185.00	0/100	-
BUY	LMT	LMT 187.50	0/100	-
BUY	LMT	LMT 185.00	0/100	-
BUY	LMT	LMT 185.00	0/100	-
BUY	LMT	LMT 185.00	0/100	-

Figure 11.5: All open orders are canceled

See also

Read more about modifying orders here:

https://interactivebrokers.github.io/tws-api/modifying_orders.html.

Getting details about your portfolio

The IB API offers a comprehensive snapshot of portfolio data, returning 157 different portfolio values through a single API call. This data provides a detailed view of our portfolios, encompassing a wide range of metrics and data points. Account values delivered via `updateAccountValue` can be classified in the following way:

- **Commodities:** Suffixed by **-C**
- **Securities:** Suffixed by **-S**
- **Totals:** No suffix

In this recipe, we'll build the code to get those data points.

Getting ready

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

How to do it...

The first step is to incorporate the account number into our **IBApp** class. While an account number is optional for requesting account-level data in a single account structure, it's best practice to specify it in the case of multiple accounts. Then, we'll add the callback to the **IBWrapper** class and add the request method to the **IBClient** class:

1. Modify the **__init__** method of our **IBApp** class by adding **account** as an argument:

```
def __init__(self, ip, port, client_id, account):
```

2. Add the following dictionary to the **__ini__** method in the **IBWrapper** class to store our account details:

```
self.account_values = {}
```

3. Then, add the callback function that responds to the TWS messages:

```
def updateAccountValue(self, key, val, currency,
    account):
    try:
        val_ = float(val)
    except:
        val_ = val
    self.account_values[key] = (val_, currency)
```

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

```
import threading
from ibapi.wrapper import EWrapper
class IBWrapper(EWrapper):
    def __init__(self):
        EWrapper.__init__(self)
        self.nextValidOrderId = None
```

```

        self.historical_data = {}
        self.market_data = {}
        self.streaming_data = {}
        self.stream_event = threading.Event()
        self.account_values = {}
<snip>
    def updateAccountValue(self, key, val, currency,
        account):
        try:
            val_ = float(val)
        except:
            val_ = val
        self.account_values[key] = (
            val_, currency)

```

4. At the end of the **IBClient** class, add the following method:

```

def get_account_values(self, key=None):
    self.reqAccountUpdates(True, self.account)
    time.sleep(2)
    if key:
        return self.account_values[key]
    return self.account_values

```

How it works...

By now, we're settling into a common pattern: request something to happen in the **IBClient** class and add the results of the call to a dictionary in the **IBWrapper** class. The **updateAccountValue** method is a callback that accepts a key (representing a specific account attribute), a value, a currency, and an account identifier, then stores the value and currency as a tuple in the **account_values** dictionary, keyed by the provided attribute key.

The **get_account_values** method requests account updates, **reqAccountUpdates**, then pauses execution for two seconds to allow time for the account data to be updated. If a specific key is provided, the method returns the value associated with that key from the

account_values dictionary. If no key is specified, it returns the entire **account_values** dictionary, providing a snapshot of all account-related values.

The IB API returns 157 different account values. Some of the more common are as follows:

- **AvailableFunds**
- **BuyingPower**
- **CashBalance**
- **Currency**
- **EquityWithLoanValue**
- **FullAvailableFunds**
- **NetLiquidation**
- **RealizedPnL**
- **TotalCashBalance**
- **UnrealizedPnL**

There's more...

Let's get the net liquidation value of the account. In the **app.py** file, replace the code after the **IBApp** class with the following:

```
if __name__ == "__main__":
    app = IBApp("127.0.0.1", 7497, client_id=10,
               account="DU7129120")
    account_values = app.get_account_values()
    net_liquidation = app.get_account_values(
        "NetLiquidation")
    app.disconnect()
```

The result is the **account_values** dictionary, which contains all 157 account values. Here's a sample:


```
{
    'AccountReady': ('true', ''),
    'AccountType': ('INDIVIDUAL', ''),
    'AccruedCash': (0.0, 'BASE'),
    'AccruedCash-C': (0.0, 'USD'),
    'AccruedCash-P': (0.0, 'USD'),
    'AccruedCash-S': (0.0, 'USD'),
    'AccruedDividend': (0.0, 'USD'),
    'AccruedDividend-C': (0.0, 'USD'),
    'AccruedDividend-P': (0.0, 'USD'),
    'AccruedDividend-S': (0.0, 'USD'),
    'AvailableFunds': (3195.34, 'USD'),
    'AvailableFunds-C': (0.0, 'USD'),
    'AvailableFunds-P': (0.0, 'USD'),
    'AvailableFunds-S': (3195.34, 'USD'),
    ...
    'SMA': (2312528.09, 'USD'),
    'SMA-S': (2312528.09, 'USD'),
    'SegmentTitle-C': ('US Commodities', ''),
    'SegmentTitle-P': ('Crypto at Paxos', ''),
    'SegmentTitle-S': ('US Securities', ''),
    'StockMarketValue': (-769.01, 'BASE'),
    'TBillValue': (0.0, 'BASE'),
    'TBondValue': (0.0, 'BASE'),
    'TotalCashBalance': (5175.9625, 'BASE'),
    'TotalCashValue': (5175.96, 'USD'),
    'TotalCashValue-C': (0.0, 'USD'),
    'TotalCashValue-P': (0.0, 'USD'),
    'TotalCashValue-S': (5175.96, 'USD'),
    'TotalDebitCardPendingCharges': (0.0, 'USD'),
    'TotalDebitCardPendingCharges-C': (0.0, 'USD'),
    'TotalDebitCardPendingCharges-P': (0.0, 'USD'),
    'TotalDebitCardPendingCharges-S': (0.0, 'USD'),
    'TradingType-S': ('STKNOPT', ''),
    'UnrealizedPnL': (-12.9, 'BASE'),
    'WarrantValue': (0.0, 'BASE'),
    'WhatIfPMEEnabled': ('true', '')
}
```

Figure 11.6: A sample of the data provided by `get_account_values`

The result is the `net_liquidation` tuple, which contains the account's net liquidation value and the currency the value is denominated in. Here's what it looks like:

```
net_liquidation
(4408.76, 'USD')
```

Figure 11.7: Net liquidation value of the account

See also

For a complete list of account values and their descriptions, see the following URL: https://interactivebrokers.github.io/tws-api/interfaceIBApi_1_1EWrapper.html#ae15a34084d9f26f279abd0bdeab1b9b5.

Inspecting positions and position details

To get position-level details from the IB API, including data such as position size, market price, value, average cost, and PnL, we can utilize specific API calls. These calls request detailed information for each position held in an account, with the API responding with the requested data for each position. This enables us to have a comprehensive view of our holdings. In this recipe, we introduce how to get the position data. In the next chapter, we'll make use of it to build a trading strategy.

Getting ready

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

How to do it...

In the previous recipe, *Getting details about your portfolio*, we used the `reqAccountUpdates` request method in the `IBClient` class to request account details. Calling `reqAccountUpdates` triggers two callbacks. The first is `updateAccountValue`, which we overrode in the `IBWrapper` method. This method returns details about the account. `reqAccountUpdates` also triggers the `updatePortfolio` callback, which returns details about the positions in the account. We'll use the `updatePortfo-`

lio method to get account details using the same **reqAccountUpdates** method:

1. Add a dictionary to the end of the **__init__** method in the **IBWrapper** class to store position details:

```
self.positions = {}
```

2. Add the following method to the end of the **IBWrapper** class:

```
def updatePortfolio(
    self,
    contract,
    position,
    market_price,
    market_value,
    average_cost,
    unrealized_pnl,
    realized_pnl,
    account_name
):
    portfolio_data = {
        "contract": contract,
        "symbol": contract.symbol,
        "position": position,
        "market_price": market_price,
        "market_value": market_value,
        "average_cost": average_cost,
        "unrealized_pnl": unrealized_pnl,
        "realized_pnl": realized_pnl,
    }
    self.positions[contract.symbol] = portfolio_data
```

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

```
<snip>
class IBWrapper(EWrapper):
    def __init__(self):
        EWrapper.__init__(self)
        self.nextValidOrderId = None
        self.historical_data = {}
```

```

        self.streaming_data = {}
        self.stream_event = threading.Event()
        self.account_values = {}
        self.positions = {}
<snip>
def updatePortfolio(
    self,
    contract,
    position,
    market_price,
    market_value,
    average_cost,
    unrealized_pnl,
    realized_pnl,
    account_name
):
    portfolio_data = {
        "contract": contract,
        "symbol": contract.symbol,
        "position": position,
        "market_price": market_price,
        "market_value": market_value,
        "average_cost": average_cost,
        "unrealized_pnl": unrealized_pnl,
        "realized_pnl": realized_pnl,
    }
    self.positions[contract.symbol] = portfolio_data

```

4. Now, add the following method to the end of the **IBClient** class:

```

def get_positions(self):
    self.reqAccountUpdates(True, self.account)
    time.sleep(2)
    return self.positions

```

How it works...

When **reqAccountUpdates** is called, the **updatePortfolio** callback is triggered for every position in the account. TWS passes details about

the position to the method, which are captured in the **positions** dictionary, and keyed by the contract's symbol.

The **get_positions** method triggers the callback through the **reqAccountUpdates** method, waits two seconds, and returns the dictionary containing the positions.

There's more...

To get the current positions in the account, add the following code to the end of the **app.py** file:

```
positions = app.get_positions()
```

The result is a dictionary with position details:

```
{'AAPL': {'contract': 4966399376: 265598, AAPL, STK, ,0,0,,, NASDAQ, USD, AAPL, NMS, False, , , , combo: ,
'symbol': 'AAPL',
'position': Decimal('10'),
'market_price': 191.30000305,
'market_value': 1913.0,
'average_cost': 191.66,
'unrealized_pnl': -3.6,
'realized_pnl': 0.0},
'NVDA': {'contract': 4966396112: 4815747, NVDA, STK, ,0,0,,, NASDAQ, USD, NVDA, NMS, False, , , , combo: ,
'symbol': 'NVDA',
'position': Decimal('-10'),
'market_price': 487.3200073,
'market_value': -4873.2,
'average_cost': 487.27595595,
'unrealized_pnl': -0.44,
'realized_pnl': 0.0}}
```

Figure 11.8: Position details of the account

See also

Read more about getting position data using the **reqAccountUpdates** method at this URL: https://interactivebrokers.github.io/tws-api/account_updates.html.

Computing portfolio profit and loss

To get portfolio PnL details from the IB API, we can utilize specific API calls. These calls request the aggregate daily profit or loss, total unrealized profit or loss, and total realized PnL, with the API responding with the requested data. Getting PnL allows us to compute periodic portfolio returns, which, in turn, unlock a suite of risk metrics based on portfolio returns. We'll make use of the dollar profit or loss to compute periodic portfolio returns in the next chapter. In this recipe, we focus on requesting and receiving the portfolio profit or loss.

Getting ready

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

How to do it...

We'll follow the same pattern in this recipe as in the last few:

1. Add a dictionary to the end of the `__init__` method in the `IBWrapper` class to store position details:

```
self.account_pnl = {}
```

2. Add the following method to the end of the `IBWrapper` class:

```
def pnl(self, request_id, daily_pnl, unrealized_pnl,
        realized_pnl):
    pnl_data = {
        "daily_pnl": daily_pnl,
        "unrealized_pnl": unrealized_pnl,
        "realized_pnl": realized_pnl
    }
    self.account_pnl[request_id] = pnl_data
```

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

```
<snip>
class IBWrapper(EWrapper):
    def __init__(self):
        EWrapper.__init__(self)
        self.nextValidOrderId = None
        self.historical_data = {}
        self.streaming_data = {}
        self.stream_event = threading.Event()
        self.account_values = {}
        self.positions = {}
        self.account_pnl = {}
    def pnl(self, request_id, daily_pnl,
            unrealized_pnl, realized_pnl):
        pnl_data = {
            "daily_pnl": daily_pnl,
            "unrealized_pnl": unrealized_pnl,
            "realized_pnl": realized_pnl
        }
        self.account_pnl[request_id] = pnl_data
<snip>
```

4. Now, add the following method to the end of the **IBClient** class:

```
def get_pnl(self, request_id):
    self.reqPnL(request_id, self.account, "")
    time.sleep(2)
    return self.account_pnl
```

How it works...

The **pnl** method is a callback function to handle PnL data. When called, it receives an integer **request_id** along with three types of PnL data: **daily_pnl**, **unrealized_pnl**, and **realized_pnl**. The method then constructs a **pnl_data** dictionary with these values and stores it in the **account_pnl** dictionary, keyed by **request_id**, effectively updating the account's PnL information associated with that specific request.

The `get_pnl` method requests PnL data for a specific account. It calls `reqPnL` to send a request to the IB API for PnL information associated with the given `request_id` and account, then pauses for two seconds to allow time for the data to be received and processed, before returning the updated PnL data stored in `account_pnl`. Unrealized profit or loss is the potential financial gain or loss on an investment that has not yet been sold for cash. Realized profit or loss is the gain or loss on the asset once it's been sold.

There's more...

At the bottom of the `app.py` file, add the following code to get the account's PnL:

```
pnl = app.get_pnl(request_id=99)
```

The result is a dictionary with the daily, unrealized, and realized PnL:

```
{99: {'daily_pnl': 5.859764794922285,  
      'unrealized_pnl': 5.859724294922216,  
      'realized_pnl': 0.0}}
```

Figure 11.9: The result of calling `get_pnl` is a dictionary with the account PnL

See also

The IB API can retrieve PnL data from two different sources: the TWS Account window and the TWS Portfolio window. These sources have different update times, which may result in differing portfolio PnL values. The method described in this recipe requests PnL data from the TWS Portfolio window with a reset schedule specified in the global configuration.

For more details on how portfolio PnL is calculated, see the following URL: <https://interactivebrokers.github.io/tws-api/pnl.html>.

