What’s inside the Autotrader?

You will start off with something like this:

```
class AutoTrader(BaseAutoTrader):
    def __init__(self, loop: asyncio.AbstractEventLoop):
        """Initialise a new instance of the AutoTrader class."""
        super(AutoTrader, self).__init__(loop)

    def on_error_message(self, client_order_id: int, error_message: bytes) -> None:
        """Called when the exchange detects an error."""
        pass

    def on_order_book_update_message(self, instrument: int, sequence_number: int,
                                     ask_prices: List[int], ask_volumes: List[int],
                                     bid_prices: List[int], bid_volumes: List[int]
                                    ) -> None:
        """Called periodically to report the status of an order book."""
        pass

    def on_order_status_message(self, client_order_id: int, fill_volume: int,
                                remaining_volume: int, fees: int) -> None:
        """Called when the status of one of your orders changes."""
        pass

    def on_position_change_message(self, future_position: int, etf_position: int) -> None:
        """Called when your position changes."""
        pass

    def on_trade_ticks_message(self, instrument: int,
                               trade_ticks: List[Tuple[int, int]]) -> None:
        """Called periodically to report trading activity on the market."""
        pass
```

The *AutoTrader* class has several callbacks that are called when the exchange sends it a message. They are inherited from the *BaseAutoTrader* class which provides default implementations that have no effect, so it is possible to omit callbacks that aren’t used.

In addition to the callbacks, the *BaseAutoTrader* class provides three additional methods that may be used to send messages to the exchange:

```
def send_amend_order(self, client_order_id: int, volume: int) -> None:
    """Amend the specified order with an updated volume."""
    # Implementation omitted

def send_cancel_order(self, client_order_id: int) -> None:
    """Cancel the specified order."""
    # Implementation omitted

def send_insert_order(self, client_order_id: int, side: Side, price: int,
                      volume: int, lifespan: Lifespan) -> None:
    """Insert a new order to buy or sell the ETF."""
    # Implementation omitted
```

Note that each time you insert an order it must have a unique *client order id* and these must be strictly increasing. Furthermore, the price is in cents and must be a multiple of the *tick size*, which is $1. Amend messages may only be used to *reduce* the volume of an order, not increase it. While it is possible for you to override these methods with your own implementations, it should not be necessary to do so.

The *BaseAutoTrader* class also has a couple of useful properties including:

```
    self.event_loop: asyncio.AbstractEventLoop  # Use to get the time or schedule callbacks
    self.logger: logging.Logger  # Use this to log events for debugging or analysis
```

What information do the callbacks provide?

The callbacks provide information about:

1. The state of the order books for both the future and the ETF
2. The state of your orders
3. The volume of trades for both the future and the ETF.

The state of the order books

Order book update messages contain information about the top five price levels in the order book for an instrument. For example:

Ask Prices	11700	11800	11900	12000	12100
Ask Volumes	1250	920	600	250	1410
Bid Prices	11600	11400	11300	11200	10900
Bid Volumes	1650	400	1120	990	1110

All prices are in cents. Ask prices are in ascending order and bid prices are in descending order so that both the best (i.e. lowest) ask price and the best (i.e. highest) bid price come first. If there are less than five price levels in the order book, the missing entries in the price and volume lists will be zero.

It is possible, though unlikely, for order book update messages to be missed or arrive out of order. A *sequence number* is provided which can be used to detect if this has happened.

The state of your orders

Order status messages describe the current state of your orders and contain three pieces of information:

fill volume The number of lots that have already traded

remaining volume The number of lots that have yet to trade

fees The total fees for this order.

Remember that you pay fees for being the *aggressor* (also known as the taker) in a trade but you receive fees for being the initiator of the passive order (also known as the maker). This means that the *fees* value may be negative.

The volume of trades

Trade ticks messages provide information about the number of lots that have traded at each price level for an instrument. Each trade tick is a pair containing a price and the number of lots that have traded at that price since the last trade ticks message.

For example:

Trade Ticks	(11700, 25)	(11800, 10)	(11900, 8)	(12000, 40)	(12100, 15)

A simple example Autotrader

As an example, consider a (very) simple trading strategy that attempts to purchase one lot of the ETF at the best bid price of the future, and sell one lot of the ETF at the best ask price of the future. This strategy aims to profit from the difference between the bid and ask prices and also from maker fees since it uses good-for-day orders.

```python
class AutoTrader(BaseAutoTrader):
    def __init__(self, loop: asyncio.AbstractEventLoop):
        """Initialise a new instance of the AutoTrader class."""
        super(AutoTrader, self).__init__(loop)
        self.order_ids = itertools.count(1)
        self.ask_id = self.ask_price = self.bid_id = self.bid_price = 0

    def on_error_message(self, client_order_id: int, error_message: bytes) -> None:
        """Called when the exchange detects an error."""
        self.logger.warning("error with order %d: %s", client_order_id,
                            error_message.decode())
        self.on_order_status_message(client_order_id, 0, 0, 0)

    def on_order_book_update_message(self, instrument: int, sequence_number: int,
                                     ask_prices: List[int], ask_volumes: List[int],
                                     bid_prices: List[int], bid_volumes: List[int]
                                    ) -> None:
        """Called periodically to report the status of an order book."""
        if instrument == Instrument.FUTURE:
            best_bid = bid_prices[0]
            best_ask = ask_prices[0]

            if self.bid_id != 0 and best_bid != self.bid_price and best_bid != 0:
                self.send_cancel_order(self.bid_id)
                self.bid_id = 0
            if self.ask_id != 0 and best_ask != self.ask_price and best_ask != 0:
                self.send_cancel_order(self.ask_id)
                self.ask_id = 0

            if self.bid_id == 0 and best_bid != 0:
                self.bid_id = next(self.order_ids)
                self.bid_price = best_bid
                self.send_insert_order(self.bid_id, Side.BUY, best_bid, 1,
                                       Lifespan.GOOD_FOR_DAY)

            if self.ask_id == 0 and best_ask != 0:
                self.ask_id = next(self.order_ids)
                self.ask_price = best_ask
                self.send_insert_order(self.ask_id, Side.SELL, best_ask, 1,
                                       Lifespan.GOOD_FOR_DAY)

    def on_order_status_message(self, client_order_id: int, fill_volume: int,
                                remaining_volume: int, fees: int) -> None:
        """Called when the status of one of your orders changes."""
        if remaining_volume == 0:
            if client_order_id == self.bid_id:
                self.bid_id = 0
            elif client_order_id == self.ask_id:
                self.ask_id = 0
```

Improving the example Autotrader

The example Autotrader will work well when the price of the future fluctuates up-and-down but if the price consistently moves in one direction it will eventually breach the position limit. A simple way to deal with this problem is for the Autotrader to keep track of its position and refrain from placing an order if it would breach the limit. However, this would lead to extended periods of time where the Autotrader is unable to trade because it has reached the limit.

Another way to deal with the problem is for the Autotrader to adjust the buy and sell prices up or down depending on its current position. Only the `__init__` and `on_order_book_update_message` methods need to change and a new `on_position_change` method needs to be added:

```python
    def __init__(self, loop: asyncio.AbstractEventLoop):
        """Initialise a new instance of the AutoTrader class."""
        super(AutoTrader, self).__init__(loop)
        self.order_ids: Iterator[int] = itertools.count(1)
        self.ask_id = self.ask_price = self.bid_id = self.bid_price = self.position = 0

    def on_order_book_update_message(self, instrument: int, sequence_number: int,
                                     ask_prices: List[int], ask_volumes: List[int],
                                     bid_prices: List[int], bid_volumes: List[int]
                                    ) -> None:
        """Called periodically to report the status of an order book."""
        if instrument == Instrument.FUTURE:
            new_bid_price = bid_prices[0]-self.position*100 if bid_prices[0] != 0 else 0
            new_ask_price = ask_prices[0]-self.position*100 if ask_prices[0] != 0 else 0

            if self.bid_id != 0 and new_bid_price not in (self.bid_price, 0)
                self.send_cancel_order(self.bid_id)
                self.bid_id = 0
            if self.ask_id != 0 and new_ask_price not in (self.ask_price, 0):
                self.send_cancel_order(self.ask_id)
                self.ask_id = 0

            if self.bid_id == 0 and new_bid_price != 0 and self.position < 100:
                self.bid_id = next(self.order_ids)
                self.bid_price = new_bid_price
                self.send_insert_order(self.bid_id, Side.BUY, new_bid_price, 1,
                                       Lifespan.GOOD_FOR_DAY)

            if self.ask_id == 0 and new_ask_price != 0 and self.position > -100:
                self.ask_id = next(self.order_ids)
                self.ask_price = new_ask_price
                self.send_insert_order(self.ask_id, Side.SELL, new_ask_price, 1,
                                       Lifespan.GOOD_FOR_DAY)

    def on_position_change_message(self, future_position: int, etf_position: int) -> None:
        """Called when your position changes."""
        self.position = etf_position
```

With this change the Autotrader can stay in the market indefinitely at the cost of forgoing some profitable trading opportunities.