The following documentation describes how end users can expect to receive and construct messages using version 1 of the API.

API Structure and Message Format

This is a custom, RPC-based API. The remote procedural calls/messages will be serialized using version 3 of Google's Protocol Buffers. You can read the documentation on Protocol Buffers here.

Most successful requests return a single response, but some return a stream of responses.

More on Protobuf

This project was built using protobuf version 3.15.8. To keep things consistent, you should use the same version.

API clients will be given a proto files, like `NinjaApiMessages.proto`, and an example Python client containing a pre-generated Python `protoc` output. You can bootstrap your setup off this example Python client or generate source files for a different language using `protoc`.

Note `protoc` is a source-to-source compiler also known as a transpiler. It translates proto source code into source code in another language. For example, you can compile the provided proto file to Go code via:

```
protoc --go_out=$DST_DIR $SRC_DIR/NinjaApiMessages.proto
```

In addition to Python and Go, C++, C#, Dart, Java, Kotlin, Objective-C, and Ruby are supported.

The proto file will start with the following lines.

```
syntax = "proto3";
package NinjaApiMessages;
```

Header

The header will be the first proto message we introduce. It is the first field in the `MsgContainer` message that wraps every API message.

```
message Header {
  enum MsgType {
    ERROR = 0;
    HEARTBEAT = 1;
    LOGIN_REQUEST = 2;
    LOGIN_RESPONSE = 3;
    NINJA_REQUEST = 4;
    NINJA_RESPONSE = 5;
    ACCOUNT_REQUEST = 6;
    ACCOUNT_RESPONSE = 7;
    SHEETS_REQUEST = 8;
    SHEETS_RESPONSE = 9;
    SHEET_RISK_REQUEST = 10;
    SHEET_RISK_RESPONSE = 11;
    SHEET_STATE_REQUEST = 12;
    SHEET_STATE_RESPONSE = 13;
    CONTRACTS_REQUEST = 14;
    CONTRACTS_RESPONSE = 15;
    CONTRACT_INFO_REQUEST = 16;
    CONTRACT_INFO_RESPONSE = 17;
    SETTLEMENTS_REQUEST = 18;
    SETTLEMENTS_RESPONSE = 19;
    PRICE_FEED_STATUS_REQUEST = 20;
    PRICE_FEED_STATUS_RESPONSE = 21;
    SECURITY_STATUSES_REQUEST = 22;
    SECURITY_STATUSES_RESPONSE = 23;
    START_MARKET_DATA_REQUEST = 24;
    MARKET_UPDATES = 25;
    STOP_MARKET_DATA_REQUEST = 26;
    WORKING_RULES_REQUEST = 27;
    WORKING_RULES_RESPONSE = 28;
    POSITIONS_REQUEST = 29;
    POSITIONS_RESPONSE = 30;
    ORDER_ADD_REQUEST = 31;
    ORDER_ADD_RESPONSE = 32;
    ORDER_ADD_EVENT = 33;
    ORDER_CHANGE_REQUEST = 34;
    ORDER_CHANGE_RESPONSE = 35;
    ORDER_CHANGE_EVENT = 36;
    ORDER_CANCEL_REQUEST = 37;
    ORDER_CANCEL_RESPONSE = 38;
    ORDER_CANCEL_EVENT = 39;
    CANCEL_ALL_ORDERS_REQUEST = 40;
    CANCEL_ALL_ORDERS_RESPONSE = 41;
    FILL_NOTICE = 42;
    ACTIVE_ORDERS_REQUEST = 43;
    ACTIVE_ORDERS_RESPONSE = 44;
  }

  MsgType msgType = 1;
  string version = 2;
}
```

The `MsgContainer` will hold the message header and its payload (generally a serialized proto message).

```
message MsgContainer {
  Header header = 1;
  bytes payload = 2;
}
```

Because we need to know the size of the `MsgContainer` when receiving it, we also have binary framing for the `MsgContainer`. Its only field is the size of the `MsgContainer`. Here is how adding that framing might look in Python.

```
container = NinjaApiMessages.MsgContainer()
# populate container fields
serialized_container = container.SerializeToString()
frame = struct.pack("i", len(serialized_container))
```

To read this binary framing you can `unpack` it.

```
sizeof_container = struct.unpack("i", frame)
```

If you use the provided example client, you will not have to worry about packing and unpacking the message frame. It will do that work for you.

The version will be set to v1.0.0 to start. This API will use SemVer version formatting. When we fix a bug in version 1, you will notice the patch version number is incremented. For example, messages sent after the first bug fix will have version v1.0.1. If we add a new field to one of the messages, you will notice that the minor version number has been incremented. For example, messages sent after the first time a new field is added will have version v1.1.0. The major version number will be incremented whenever new messages become available or we make changes that are not completely backward compatible. For example, messages sent after version 2 is released will have version v2.0.0.

You must send a version in login messages or you will receive an error. The version you specify in your first successful login message will be the version we assume you are using when sending all subsequent messages unless you specify a different version in the message header of a later message. Please note, that the login message to each endpoint must provide a version number. That is, if you send a login request to the trading connection and position connection, both login messages must also have a populated version field. All inbound messages will specify a version number. Over time we may deprecate old versions. We will give ample warning before old versions are completely removed.

Connecting

Trade support will supply you with two to three endpoints to connect to. Each endpoint will include a hostname or IP and a port. All connections will be TCP connections (layer 4 of the OSI model). Connections will be stateful, meaning once you connect, the connection will stay open until you or the server closes the connection. Once the connection is closed, a new connection will have to be established if you wish to continue using the API.

Here is a description of the three connection points you will be given:

1. **Trading Connection:** This endpoint will connect directly to your ninja and allow you to query for both market data and submit orders. You can only have one of these per endpoint and must have a valid access token (described later) to connect.
2. **Position Connection:** This is a dedicated endpoint for position queries. It is ninja agnostic. When submitting a position query, you will submit it for accounts, not for a ninja. You can have multiple position connections but must have a unique access token (described later) for each connection you make.
3. **Market Data Connection (optional):** There are two reasons you might want to use this optional connection. (1) It reduces the work your ninja has to do because your ninja no longer needs to disseminate market data to your client. (2) It allows you to access products that your ninja does not receive market data for, allowing your algorithms to price off markets that your ninja does not know about. You can have multiple market data connections but must have a unique access token (described later) for each connection you make.

When a connection is established, the first message must be a login message.

Each login request must specify a connection type to reduce the likelihood of connecting to the wrong endpoint.

```
enum ConnectionType {
  TRADING_CONNECTION = 0;
  POSITION_CONNECTION = 1;
  MARKET_DATA_CONNECTION = 2;
}
```

Trade support will also provide you with an access token. This token grants access to a resource (often a ninja) and a connection type. It is meant to provide extra insurance, so errors like connecting to a production environment when testing your API client code, are less likely to occur.

```
message Login {
  string user = 1;
  string password = 2;
  ConnectionType connectionType = 3;
  string accessToken = 4;
}
```

The `user` field will be your tag 50 and the `password` field will be the password you regularly use to log in to your OptionsFe.

In response, you will receive the following message.

```
message LoginResponse {
  ConnectionType connectionType = 1;
  string ninja = 2;
  repeated string accounts = 3;
}
```

Both the `ninja` and `accounts` fields will be populated when connecting to a **Trading Connection**. You can use it to verify that you have been provided the correct endpoint. Only `accounts` will be populated when connecting to a **Position Connection**. Neither `ninja` nor `accounts` will be populated for the **Market Data Connection**.

Version 1 Messages

This section describes the messages you can send once logged into an API endpoint.

Errors

Order-handling responses will contain a status flag indicating if your action was a success or failure. Doing so allows you to tie order manipulation outcomes to a specific order. When other requests fail, you will receive a more generic error response.

Response (connection type(s): all)

```
message Error {
  enum Type {
    UNSPECIFIED = 0;
    AUTHENTICATION = 1;
    AUTHORIZATION = 2;
    NOT_LOGGED_IN = 3;
    RECEIVE_ERROR = 4;
    UNSUPPORTED_MESSAGE = 5;
    INVALID_MESSAGE = 6;
    INVALID_VERSION = 7;
    CONNECTION_INACTIVE = 8;
    LOGIN_REQUEST_MALFORMED = 9;
    NINJA_REQUEST_MALFORMED = 10;
    SHEETS_REQUEST_MALFORMED = 11;
    SHEET_RISK_REQUEST_MALFORMED = 12;
    SHEET_STATE_REQUEST_MALFORMED = 13;
    CONTRACTS_REQUEST_MALFORMED = 14;
    CONTRACT_INFO_REQUEST_MALFORMED = 15;
    SETTLEMENTS_REQUEST_MALFORMED = 16;
    SECURITY_STATUSES_REQUEST_MALFORMED = 17;
    START_MARKET_DATA_REQUEST_MALFORMED = 18;
    STOP_MARKET_DATA_REQUEST_MALFORMED = 19;
    WORKING_RULES_REQUEST_MALFORMED = 20;
    POSITION_REQUEST_MALFORMED = 21;
    RATE_LIMIT_EXCEEDED = 22;
  }

  Type type = 1;
  string msg = 2;
}
```

You can use the `errorType` field to determine the type of error that occurred and the `msg` field to gather more information about the specific error.

It is worth describing a few of the error types. If you get an `AUTHENTICATION` error that means your username or password was incorrect. If you get an `AUTHORIZATION` error that means that trade support gave you the wrong endpoint or they need to update your permissions because you are not authorized to access the resources at that endpoint. You will receive a `RECEIVE_ERROR` if the server handling your requests has trouble processing your request. This might be because you sent a bad size in the message frame, failed to serialize the message correctly, sent a header message type that didn't match the payload, or sent more than a gigabyte of traffic to the server in a very short time frame. You will only receive an `INVALID_VERSION` error if the version you sent in the header for your `Login` message is invalid. This is because we require you to specify a valid version when you log in so that we know what version of the response we should send you for all subsequent messages. Your login requests will fail until you send a valid version in `SemVer` format. The `CONNECTION_INACTIVE` error will occur whenever the server has not received a message from your client for more than 5 seconds, which implies that it missed a heartbeat. If it misses more than 2 heartbeats, you will receive a `CONNECTION_INACTIVE` error then the server will close the connection. The `UNSUPPORTED_MESSAGE` error will occur if you send a message to an endpoint that is not designed to receive it. For example, if you send a position request to the trading connection.

Heartbeats

Every connection type requires you to send a message within 5 seconds so we know the connection is still active. We will close the connection if we don't hear anything from your client for 10.5 seconds. You should send a heartbeat to the server if you have not sent other messages in the last 5 seconds. However, it does not hurt to send out a heartbeat every 5 seconds regardless of what other messaging is happening.

Similarly, each API endpoint will send you heartbeats every five seconds.

Ninja

The connection that trade support gives you points to a ninja. Although this information is part of the login response, you may wish to query it separately.

Ninja Request (connection type(s): trade connection)

Here you can send the header with the message type `NINJA_REQUEST`. No message body is needed.

Ninja Response (connection type(s): trade connection)

```
message NinjaInfo {
  string name = 1;
}
```

Accounts

You also might wish to know the accounts you can trade into or the accounts you can receive positions for (keep in mind they may be different).

Account Request (connection type(s): trade connection, position connection)

Here you can send the header with the message type `ACCOUNT_REQUEST`. No message body is needed.

Account Response (connection type(s): trade connection, position connection)

```
message Accounts {
  repeated string accounts = 1;
}
```

Contracts and Settlements

Most likely, the first thing you'll be interested in after you're logged in is what instruments are available for you to trade. This section describes the messages you can send to discover available contracts and their properties.

Contracts Request (connection type(s): trade connection, market data connection)

Here you can send the header with the message type `CONTRACTS_REQUEST`. No message body is needed.

Contracts Response (connection type(s): trade connection, market data connection)

```
enum Exchange {
  UNKNOWN_EXCHANGE = 0;
  CME = 1;
  ICE = 2;
  ENDEX = 3;
  LIFFE = 4;
  CFE = 5;
  EUREX = 6;
  LME = 7;
  B3 = 8;
  JPX = 9;
  FMX_FUT = 10;
  FMX_UST = 11;
}

message Contract {
  Exchange exchange = 1;
  string secDesc = 2;
  string whName = 3;
}

message Contracts {
 repeated Contract contracts = 1;
}
```

The `Contract` message is used ubiquitously throughout the API. In a handful of request messages, you can send `Contract` messages as part of the request. When populating a contract message, the `exchange` and `secDesc` must be set. You can optionally supply a `whName`. Inbound messages will always have `secDesc` and `whName` populated.

Contract Info Request (connection type(s): trade connection, market data connection)

Once you have contracts, you will probably want contract information.

```
message GetContractInfo {
  repeated Contract contracts = 1;
}
```

You must supply at least one contract. If the repeated `contracts` field is empty, a `CONTRACT_INFO_REQUEST_MALFORMED` error will be returned.

Contract Info Response (connection type(s): trade connection, market data connection)

```
message Date {
  uint32 year = 1;
  uint32 month = 2;
  uint32 day = 3;
}

message Side {
  UNKNOWN_SIDE = 0;
  BUY = 1;
  SELL = 2;
}

message ContractLegInfo {
  Side side = 1;
  int32 ratio = 2;
  Contract contract = 3;
}

message ContractInfo {
  enum ContractType {
    FUTURE = 0;
    OPTION = 1;
    BOND = 2;
  }

  enum OptionType {
    PUT = 0;
    CALL = 1;
  }

  Contract contract = 1;
  string secGroup = 2;
  string cfiCode = 3;
  ContractType contractType = 4;
  OptionType optionType = 5;
  Date lastTradeDate = 6;
  double strike = 7;
  double tickSize = 8;
  double tickAmt = 9;
  string currency = 10;
  double highLimitPrice = 11;
  double lowLimitPrice = 12;
  Contract underlying = 13;
  repeated ContractLegInfo legs = 14;
}

message ContractInfoList {
  repeated ContractInfo contractInfoList = 1;
}
```

Please note, some of these fields only apply to options, like `putCall`, `strike`, and `underlying`. They will not be populated for future contracts.

Settlement Request (connection type(s): trading connection, market data connection)

For exchanges that provide settlement information, you can also request settlements.

```
message GetSettlements {
  repeated Contract contracts = 1;
}
```

You must supply at least one contract. If the repeated `contracts` field is empty, a `SETTLEMENTS_REQUEST_MALFORMED` error will be returned.

Settlement Response (connection type(s): trading connection, market data connection)

```
message Settlement {
  Contract contract = 1;
  Date date = 2;
  optional double prelim = 3;
  optional double final = 4;
}
```

If you request settlements early in the day, you might receive yesterday's settlements. If the server does not have yesterday's preliminary settlement stored, you will only receive yesterday's final settlement. If you request settlements after the prelim settlement has been released but before the final settlement is released, only the `prelim` field will be populated.

Sheets

The following set of messages will help you figure out what sheets the ninja has created. A lot of detail about each sheet and the strategies they contain will be added in later versions of the API.

In version 1, you will only be able to view future sheets. Contrary to what the name suggests, you can trade options on future sheets.

Sheets Request (connection type(s): trade connection)

```
message GetSheets {
  enum SheetType {
    FUTURE_SHEETS = 0;
  }

  SheetType type = 1;
}
```

Sheets Response (connection type(s): trade connection)

```
message Sheet {
  string name = 1;
  string account = 2;
  repeated Contract contracts = 3;
}

message Sheets {
  repeated Sheet sheets = 1;
}
```

Some users may prefer obtaining a list of contracts that are listed on future sheets rather than querying for all available contracts. If this applies to you, you can obtain a list of contracts via a **Sheets Request** rather than a **Contracts Request**. The list of contracts provided by a **Sheets Request** will often be less overwhelming.

Sheet Risk Request (connection type(s): trade connection)

Another thing you might be interested in is the risk for a sheet. Please note, these are the risk parameters you can set as a user, not the risk parameters that are set by trade support. The risk parameters sent by trade support still apply even though they are not shown here.

```
message GetSheetRisk {
  repeated string sheets = 1;
}
```

Notice that this is a repeated field, so you can request this information for several sheets at once. If you leave this field blank, you will get risk for all future sheets.

Sheet Risk Response (connection type(s): trade connection)

```
message SheetRisk {
  string sheet = 1;
  uint32 clipSize = 2;
  uint32 ordersSent = 2;
  uint32 maxOrders = 3;
}

message SheetRiskList {
  repeated SheetRisk riskForSheets = 1;
}
```

This message will expand once functionality is added that allows you to create ‘Price Setter’ strategies through the API.

You might also be interested to know if the sheet is on or not. For version one of the API, this does not affect your ability to enter orders or receive market data.

Sheet State Request (connection type(s): trade connection)

```
message GetSheetStates {
  repeated string sheets = 1;
}
```

Notice that this is a repeated field, so you can request this information for several sheets at once. If you leave this field blank, you will get the state of all future sheets.

Sheet State Response (connection type(s): trade connection)

```
message SheetState {
  enum Status {
    DISABLED = 0;
    OFF = 1;
    ON = 2;
  }

  string sheet = 1;
  Status status = 2;
}

message SheetStates {
  repeated SheetStates sheetStates = 1;
}
```

Market Data

After you have established what contracts you are interested in, you can query for their market data.

Price Feed Status Request

Before getting market data, it is important to check if the source you are receiving market data from has market data to give you. If it does, we say its “Price Feed” is up. If not, we say its “Price Feed” is down.

You can send the header with message type `PRICE_FEED_STATUS_REQUEST` to make this request. No message body is needed.

Price Feed Status Response

In response to a price feed status request, you will get a message describing if the price feed is up or down.

```
message PriceFeedStatus {
  enum Status {
    DOWN = 0;
    UP = 1;
  }

  Status status = 1;
}
```

If any price feed goes up or down and you are listening to live market data, you will also receive one of these messages to denote the change in the price feed state.

Start Market Data Request (connection type(s): trade connection, market data connection)

```
message DurationMilliseconds {
  uint32 duration = 1;
}

message StartMarketData {
  repeated Contract contracts = 1;
  DurationMilliseconds cadence = 2;
  bool includeImplieds = 3;
  bool includeTradeUpdates = 4;
}
```

You must provide a list of contracts you wish to receive market data for or you will receive a `START_MARKET_DATA_REQUEST_MALFORMED` error. You can optionally select the cadence in which you wish to receive market data in milliseconds. If you don’t set a cadence or set it to 0, it will send out market data as often as possible. Otherwise, the minimum accepted cadence is 100 milliseconds. There may be some jitter in the system, so even if you specify a cadence of 100 milliseconds, in actuality the market data may arrive at a slightly different and variable cadence. We will do our best to get market data to you when you want it.

Market Data Subscription Response (connection type(s): trade connection, market data connection)

```
message EpochNanoseconds {
  fixed64 timestamp = 1;
}

message TobUpdate {
  uint32 bidOrders = 1;
  uint32 askOrders = 2;
  uint32 bidQty = 3;
  uint32 askQty = 4;
  double bidPrice = 5;
  double askPrice = 6;
  EpochNanoseconds lastChange = 7;
}

enum Aggressor {
  UNKNOWN_AGGRESSOR = 0;
  BUYER = 1;
  SELLER = 2;
  NO_AGGRESSOR = 3;
}

enum MarketTradeType {
  REGULAR_TRADE = 0;
  OPENING_TRADE = 1;
  BLOCK_TRADE = 2;
  OTHER_TRADE = 3;
}

message TradeUpdate {
  uint32 tradedQty = 1;
  double tradePrice = 2;
  EpochNanoseconds transactTime = 3;
  Agressor aggressor = 4;
  MarketTradeType marketTradeType = 5;
  bool tradeAffectsVolume = 6;
  uint32 sqn = 7;
}

message MarketUpdates {
  Contract contract = 1;
  TobUpdate tobUpdate = 1;
  repeated TradeUpdate tradeUpdates = 2;
}
```

Remember, these messages will be disseminated at the cadence you sent unless there has been no update since the data was last sent or you cancel your subscription.

You will also receive streamed price feed status updates and security status updates if the status changes for a contract you are actively listening to is affected.

Stop Market Data Request (connection type(s): trade connection, market data connection)

```
message StopMarketData {
  repeated Contract contracts = 1;
}
```

If the contracts filter is empty, it will stop sending market data. If the filter is populated with contracts, it will stop market data for only those contracts.

In response to this message, you will see the market data stream stop or receive an error message if your request fails.

Security Statuses Request

You can request security statuses at any time.

```
message GetSecurityStatuses {
  repeated Contract contracts = 1;
}
```

You must provide a list of contracts to receive security statuses or you will receive a `SECURITY_STATUSES_REQUEST_MALFORMED` error.

Security Statues Response

```
message SecurityStatus {
  enum Status {
    UNKNOWN = 0;
    TRADING_HALT = 1;
    CLOSE_SCHEDULED = 2;
    CLOSE_MKT_EVENT = 3;
    OPEN_SCHEDULED = 4;
    OPEN_MKT_EVENT = 5;
    UNAVAILABLE = 6;
    PRE_OPEN_SCHEDULED = 7;
    PRE_OPEN_MKT_EVENT = 8;
    PRE_OPEN_NO_CANCEL = 9;
    NEW_PRICE_INDICATION = 10;
    PRE_CROSS = 11;
    CROSS = 12;
    PRE_CLOSE = 13;
    CLOSE = 14;
    POST_CLOSE = 15;
    IMPLIEDS_OFF_SCHEDULED = 16;
    IMPLIEDS_OFF_MKT_EVENT = 17;
    IMPLIEDS_ON_SCHEDULED = 18;
    IMPLIEDS_ON_MKT_EVENT = 19;
    TECHNICAL_HALT = 20;
  }

  Contract contract = 1;
  Status status = 2;
}

message SecurityStatuses {
  repeated SecurityStatus statuses = 1;
}
```

There are a lot of different market states. It is up to you, the user, to determine what the market state means for you. Some market states imply that you can’t trade, cancel, or add orders. How exchanges define when you can trade, cancel, and add orders varies.

As previously mentioned, if you are subscribed to market data for a product and its security status changes, you will automatically receive a `SecurityStatuses` message with the updated status.

Working Rule Prefixes

Before you submit orders, you might be interested in which working rules are available. You can obtain this information from the “Working Rules” tab of your OptionsFe, but this interface allows you to access working rule information from inside the API.

Working Rules Request (connection type(s): trade connection)

Here you can send the header with message type `WORKING_RULES_REQUEST`. No message body is needed.

Working Rules Response (connection type(s): trade connection)

```
message WorkingRule {
  enum WorkType {
    LIMIT                = 0;
    WORK_STOP            = 1;
    STOP_LIMIT           = 2;
    HBLA_RETIRED         = 3;
    OTHER_MONTH          = 4;
    CASCADE              = 5;
    SIZE_CUTOFF          = 6;
    STERL_HBLA           = 7;
    BOR_HBLA             = 8;
    LEAD_MONTH           = 9;
    LEG_SPLIT            = 10;
    FADE_STOP_LIMIT      = 11;
    FADE_SIZE_CUTOFF     = 12;
    FADE_LIMIT           = 13;
    SCO_SPREAD           = 14;
    BEST_MONTH           = 15;
    SMART_SPLIT          = 16;
    OUTRIGHT_SMART_SPLIT = 17;
    SCO_MINI             = 18;
    OPTION_SCRATCH       = 19;
    DYNAMIC              = 20;
    STOP_PRICE           = 21;
    SOFR_HBLA            = 22;
    DYNAMIC_HEDGE_OFFSET = 23;
    EAGLE                = 24;
    PARTIAL              = 25;
    AGRO_EE              = 26;
    AGRO_PLUS_ONE        = 27;
  }

  string prefix = 1;
  WorkType workType = 2;
}

message WorkingRules {
  repeated WorkingRule workingRules = 1;
}
```

Notice that no details are sent with the working rule. This is intentional. Working rule parameters can vary depending on the `WorkType` so to avoid complicating the message these parameters have been left off. In version 2 of the API, there will be a separate “Get Working Rule Details” request to get additional information about working rules, but for now, you’ll have to reference your OptionsFe for this information.

Obtaining Positions

One other bit of information you might wish to know before trading is your position in the market.

Position Request (connection type(s): position connection)

```
message GetPositions {
  repeated string accounts = 1;
  repeated Contract filters = 2;
  bool includeSpec = 3;
}
```

If you leave the account field blank, you will receive positions for all accounts to which your access token grants access. If you leave the filters field blank, you will get positions for all contracts for the specified accounts. If you leave both fields blank, you will receive all positions for all contracts in all accounts to which your access token grants access.

The `includeSpec` field is a toggle that allows you to include speculative positions. Speculative positions are mainly used by options trading desks. If you are unsure if you have speculative positions, you most likely don’t and will be safe setting this toggle to `true` or `false`.

Position Response (connection type(s): position connection)

```
message Position {
  string account = 1;
  Contract contract = 2;
  Date tradeDate = 3;
  int32 dailyPos = 4;
  int32 openPos = 5;
  int32 totalPos = 6;
}

message Positions {
  repeated Position positions = 1;
}
```

If you specified accounts in your request, the account field of the response will be populated, otherwise, it will be empty. You will get a `Position` message for every unique account and contract grouping.

Submitting Orders

After you have contracts, positions, and market data, you are ready to start adding, changing, or canceling orders. Submitting orders as described in this section is equivalent to submitting click orders through OptionsFe.

As a safety precaution, an OptionsFe must be open and connected to the ninja before submitting orders. Any order management done via the API when an OptionsFe is not open will return an error.

You'll notice many of the following messages have a field for sheet and account. When submitting a request, one of the two fields must be populated. If you populate both and the account does not match the sheet's account, your request will fail. Whenever you specify a sheet, the ninja will perform the requested operation via the specified future strategy sheet. When you receive a response, if it is clear that an order belongs to a sheet, the sheet field will be populated. Responses will always have the account field populated.

Order Add Request (connection type(s): trade connection)

```
message TimeInForce {
  enum Type {
    DAY = 0;
    GTC = 1;
    IOC = 2;
  }

  Type type = 1;
}

message OrderAdd {
  string sheet = 1;
  string account = 2;
  uint64 id = 3;
  Contract contract = 4;
  TimeInForce timeInForce = 5;
  Side side = 6;
  uint32 qty = 7;
  double price = 8;
  string prefix = 9;
}
```

The ID field will help tie your order to a response. You are responsible for coming up with a unique ID for the order. IDs only have to be unique while the session is connected. If you disconnect and establish a new session, you can reuse the IDs from a previous session. If you have 2 or more connected sessions running simultaneously, you are responsible for ensuring they do not use overlapping IDs.

Please note, `TimeInForce` is a separate message in case we add a GTD option in the future, which would require an optional `date` field.

Order Add Response (connection type(s): trade connection)

```
message Status {
  enum Outcome {
    FAILURE = 0;
    SUCCESS = 1;
  }

  Outcome outcome = 1;
  string msg = 2;
}

message OrderAddResponse {
  uint64 id = 1;
  Status status = 2;
  string orderNo = 3;
  string sheet = 4;
  string account = 5;
}
```

Once you receive this message, you will start using the order number to track your order instead of the ID.

Order Add Event (connection type(s): trade connection)

You will receive an `OrderAddEvent`, when a change to one of your API orders triggers an order add within the ninja. You will see this most when using certain working rules. In later versions of the API, you may see it alongside ninja induced hedges as well.

```
message OrderAddEvent {
  string orderNo = 1;
  string triggerOrderNo = 2;
  Contract contract = 3;
  TimeInForce timeInForce = 4;
  Side side = 5;
  uint32 qty = 6;
  double price = 7;
  string prefix = 8;
  string sheet = 9;
  string account = 10;
}
```

You can derive what order triggered this new order by looking at the `triggerOrderNo` field.

If there is a working rule attached to this new order, it will be defined by the `prefix` field.

Order Change Request (connection type(s): trade connection)

```
message OrderChange {
  string orderNo = 1;
  uint32 qty = 2;
  double price = 3;
  string prefix = 4;
}
```

Order Change Response (connection type(s): trade connection)

```
message OrderChangeResponse {
  string orderNo = 1;
  Status status = 2;
  string sheet = 3;
  string account = 4;
}
```

Order Change Event (connection type(s): trade connection)

You may receive order change events. For example, if a working order responds to something in the market. You must always be ready to handle order change events when you have active orders in the market.

```
message OrderChangeEvent {
  string orderNo = 1;
  Contract contract = 2;
  Side side = 3;
  uint32 newQty = 4;
  double newPrice = 5;
  string channel = 6;
  string sqn = 7;
  string sheet = 8;
  string account = 9;
}
```

If possible, we will give you the market data channel number and sequence number that caused the change event. You can use this information to look up the event in the tick database.

Order Cancel Request (connection type(s): trade connection)

```
message OrderCancel {
  string orderNo = 1;
}
```

Order Cancel Response (connection type(s): trade connection)

```
message OrderCancelResponse {
  string orderNo = 1;
  Status status = 2;
  string sheet = 3;
  string account = 4;
}
```

Order Cancel Event (connection type(s): trade connection)

You may also receive order cancellation events. For example, if an SMP cancel occurs. You must always be ready to handle order cancellation events when you have active orders in the market.

```
message OrderCancelEvent {
  string orderNo = 1;
  Contract contract = 2;
  string reason = 3;
  string sheet = 4;
  string account = 5;
}
```

We will provide a reason for the cancellation whenever possible.

Cancel All Orders Request (connection type(s): trade connection)

You may want to "turn off" your API at some point by canceling all active orders submitted (added) through the API. Please note, the API implements cancel-on-disconnect (CoD) functionality so when you disconnect ALL connected API clients, all API submitted day orders will automatically be canceled.

```
message CancelAllOrders {
  bool cancelGTCs = 1;
}
```

Cancel All Orders Response (connection type(s): trade connection)

```
message CancelAllOrdersResponse {
  repeated OrderCancelResponse canceledOrders;
}
```

Fill Notice (connection type(s): trade connection)

You will immediately be notified if any order the connected ninja manages is filled, so you must be prepared to handle this message whenever an active order is submitted to the market.

```
message FillNotice {
  string orderNo = 1;
  bool isApiOrder = 2;
  Contract contract = 2;
  double price = 3;
  Side side = 4;
  uint32 qty = 5;
  bool isPartialFill = 6;
  EpochNanoseconds transactTime = 7;
  Aggressor aggressor = 8;
  MarketTradeType marketTradeType = 9;
  bool tradeAffectsVolume = 10;
  string sheet = 11;
  string account = 12;
  bool isQuote = 13;
  string strategyGroup = 14;
  string strategy = 15;
  repeated FillNotice legFills = 16;
}
```

The `isApiOrder` field will be set to `true` for orders that were submitted (added) via the API and that are not being managed by a user connected via an OptionsFe. An OptionFe user can take control of an API submitted order at any time.

Active Orders Request (connection type(s): trade connection)

Finally, you can request that all active orders be sent to you anytime.

```
message GetActiveOrders {
  bool showOnlyApiOrders = 1;
  repeated Contract filters = 2;
}
```

Again, an “API order” is an order that was submitted (added) via the API and that is not being managed by a user connected via an OptionsFe. You can optionally specify a contract filter when submitting a request for active orders.

Active Orders Response (connection type(s): trade connection)

```
message ActiveOrder {
  string orderNo = 1;
  Contract contract = 2;
  Side side = 3;
  uint32 qty = 4;
  double price = 5;
  string prefix = 6;
  string sheet = 7;
  string account = 8;
}

message ActiveOrders {
  repeated ActiveOrder activeOrders = 1;
}
```