**III- The Websocket layer**, 2

III.A-Authentication., 3

III.C-Send a new order to the exchange, 4

III.D Cancelling an existing order, 5

III.E Subscribe Order Book, 6

III.E Updating orders, 6, 7

III.F Subscribe Orders, 9

**I-Introduction**, 1

**II-The interfaces: FIX vs Websocket**, 1

**IV- The FIX protocol**, 11

IV.A - Introduction, 11

IV.B- Establishing a connection, 12

IV.B.1- Preparing the initiator.cfg file, 13

IV.B.2- Preparing the stunnel connection, 15

IV.B.3- Application Data Dictionary and Transport Data Dictionary, 15

IV.B.4 – Initiate a Session, 16

IV.B.5 – Sending Orders, 17

**I-Introduction**

The purpose of this document is to describe all the different mechanisms that the ATS has to expose to the outside world the trading against its CLOB (Central Limit Order Book) for the secondary market. This is intended to be a technical document, however we will try to make it clear enough so that non-technical users can also read and have a sense about the different interfaces that are available for the ATS.

**II-The interfaces: FIX vs Websocket**

The platform has two very specific technologies to communicate with the outside world: The FIX protocol and the Websocket layer.

When we refer to communicating with the outside world we refer to very specific functionalities as

* Sending new orders
* Cancelling or updating the previously created orders
* Receiving execution reports related to my account orders

In this context, every technology offers different strengths and weakness and it will be the external user the one that will have to decide which ones fits best their specific needs.

On one side, the FIX protocol is the standard communication protocol in the financial markets. Initiated in 1992, whoever has worked in an electronic trading project, should have dealt with this technology somehow during their career. Its features cover every possible action that could take place in the financial markets and is designed to be very performant, so only the basic and necessary information flows from one side to the other, optimizing the network performance. FIX has a significant reservoir of tags and values, dictionaries, groups, etc., which may seem intimidating if you are not familiar with the language/protocol. Rialto can point you in the right direction and provide guidance if needed on a limited basis.

On the other side, the *Websocket* technology that was developed to provide an easy and friendly enough bi-directional communication channel between two counterparties. This means, with the massive growth of websites and the proliferation of REST services on the in internet, more companies started to face scenarios where the server needed to send messages to the client and vice versa, as much as the other way around. Implementing this with the REST protocol would need the client to be constantly pinging the server, which is not performant nor safe. In this context, the *Websocket* technology offers an easy path to deal with this. This is why *Websocket* technology is popular among all the crypto exchanges, which allows different developers to safely and easily connect their trading robots to the venue.

In the ATS, both technologies have the ~~same~~ similar strengths and weakness as was mentioned before. The FIX layer implementation is as simple as the *Websocket* one and implements the same messages, but it is more scalable and it allows managing several accounts with just one connection. The *Websocket* layer implements a necessary number of fields for every message and it allows to connect to the *CLOB* as easy as you would connect to any crypto exchange. However, this simplicity can be a burden in the long term if you want to have additional functionality with the ATS.

Following, we will the technical aspects of these technologies so that you can choose the option that best fits to your needs.

**III- The Websocket layer**

The goal of this section is to describe all the messages that will flow from one side to the other. We do not pretend to go too deep into the protocol connectivity details, as this is very specific to every language and plenty of information can be found on the internet.

However, it is important to remember that in order to establish a *Websocket* connection, we need to keep in mind, the following concepts.

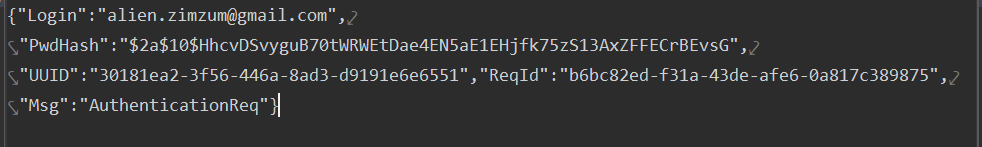
* The client will have to establish a connection to the server. Every technology has its own libraries, but you will have to provide the *Websocket* address to that library, and this library will do the work.
* A *Websocket* url looks like the following
  + wss://<url>:<port>
  + example: wss://70.37.85.42:3132
* Once the connection is established, there should be
  + A callback where all the *Websocket* messages are received
  + An asynchronous loop that will constantly wait for *Websocket* responses
* In summary, once the connection is established, every library will have
  + A method to send messages
  + A callback to receive responses
* And these messages will be implemented in the very popular ***JSon*** standard.

Following, we will describe all the existing messages that should be implemented. We will not describe every specific field meaning as they are designed to match all the fields available in the FIX protocol. Therefore, for further information, we will provide links that explain what all those fields are meant to be

III.A-Authentication.

Once the connection has been established, the server will want to know who you are and this is called authentication.

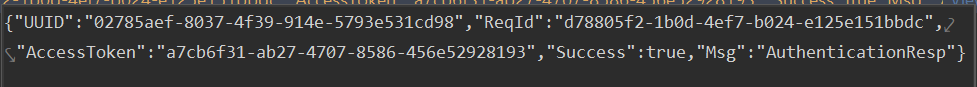
The ***json*** that we should send looks as follows



Where

* **Msg**=”*AuthenticationReq*”
* **Login**: Email of the ATS users on behalf the client will be interacting with the ATS. This means, the users that will be sending orders and trading against the platform.
* **PwdHash**: Encrypted password of the previous user using the *BCrypt* encryption library.
  + <https://en.wikipedia.org/wiki/Bcrypt>
  + This is a generic library also available in several programming languages and it should be used out of the shelve, with no salt or no special settings.
* **UUID**: Unique identifier of the client. Therefore, if you have two different clients in the same computer connecting to the ATS, this field should have two different values identifying each client. Usually the best option is to send a *uuid* data type that will allow 100% unique ids across different clients. Otherwise the authentication request will be rejected.
* **ReqId**: String value that will be echoed back in the response. This is mostly used for clients that have several connections and want to know for a response what is the request that it belongs to.

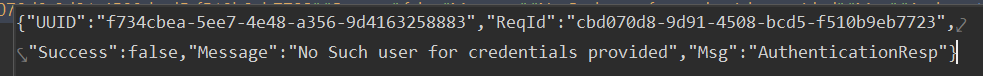
This is how the response will look if everything is ok



The most important fields here are

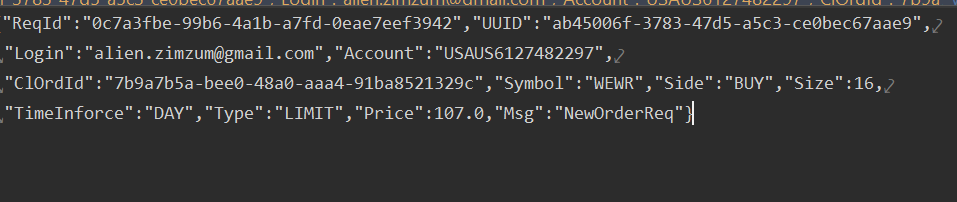
* **Msg**=”*AuthenticationResp*”
* **Success=”**True**”** indicatingthat theauthentication could be properly made
* **AccessToken=** This is an important key that will have to be sent on every protected request (subscribing to orders and order routing) and it will allow the *CLOB* engine to know that it is interacting with the client with the right privileges. In summary, the client will have to save this key in memory and send it with all the messages that require security (new orders, cancel orders, order subscription, etc.)

If something went wrong, however, the message will look like the following:



III.B-Send a new order to the exchange

Once we are authenticated to the exchange, we can start sending orders. Just send the following messages:

****

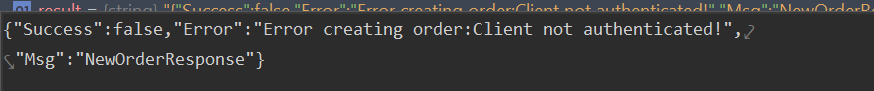
Where

* **Msg**: “*NewOrderReq*”
* **ReqId**: Is the request id that will be echoed in the *NewOrderResp* message.
* **UUID**: Is the unique id that identifies the client (any ***Guid*** should be fine)
* **Login**: is the ATS shareholder who is sending the order.
* ***Account***: Is the account that will provide/receive the funds for the order. If no account is provided, the default account will be used. Remember that this account has to be populated as a *Piermont* account and we should use here the account number (***account\_number*** field in the ***fund\_account\_details*** table)
* **ClOrdId**: Is the unique client id of the order. Once the order gets to the exchange it will have an *OrderId*, which is assigned, by the exchange, but you should be able to identify your order by your client Id as this order might never make it to the exchange and you will still need to properly identify the executions reports that refer to this order.
* **Symbol**: The security that we are trading i.e. AAPL or MSFT, etc.
* **Side**: “*BUY*” or “*SELL*”
* **Size**: The number of shares to trade
* **TimeInForce**: Order expiration. It can be:
  + “*DAY*” (Expires at the end of the day)
  + “*GTC*” (Never expires)
* **Type**: only “*LIMIT*” for the moment
* **Price**: The price the order will have

For more clarifications about every field, refer to the following link

<https://www.fixtrading.org/online-specification/business-area-trade/#msg014>

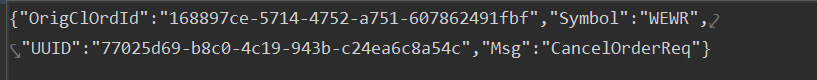
After sending the order, you will receive a ***NewOrderResponse*** that will tell us if the order was received or not by the exchange. In the following error message the order was rejected because the user was not authenticated, but if ***Success=True***, we would still have to wait to receive an execution report with ***Status=New*** , that confirms that the order was not only received but also accepted in the exchange. If it was rejected, we would receive a **Status**=*Rejected* execution report (***ExecutionReportMsg***) with the ***Text*** field describing the rejection reason.

****

One important thing to mention is that we will always be automatically subscribed to an order we just sent to the exchange, no matter if we previously sent the subscription message. Therefore, after sending a ***NewOrderReq*** message, we will start receiving execution reports (***ExecutionReportMsg***) just like if we had subscribed to that order.

III.C Cancelling an existing order

To cancel an order we should just send the following message

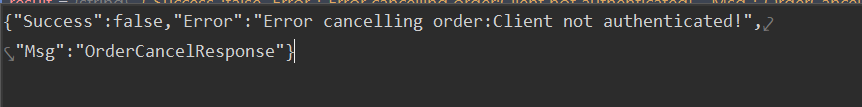


Where we have to keep in mind the following concepts

* **Msg**=”*CancelOrderReq*”
* **OrigClOrdId** :Is the client order id of the cancelled order.
* **OrderId**: If we don’t want to provide the client order id but the exchange order id, we can do it through this field
* **Symbol**: For security reasons, we not only have to provide the order id but also the security ***symbol.*** If the order id and symbol provided mismatch, the cancellation will be rejected.
* **UUID**: The unique client id specified in the login process

We will receive a message confirming the arrival or rejection of the cancellation request. If ***Success=true***, we would be just be confirming that the cancellation request has arrived to the exchange. Whatever happens after sending a cancellation message, it will be received through the traditional ***ExecutionReportMsg*** that we described in the order subscription.

An example of a cancellation received but rejected before getting to the exchange (ex: user not authenticated) would be received in the ***OrderCancelResponse*** message and would look as follows:



Remember that you will only be able to cancel orders that you are subscribed or that you have sent to the exchange, where you would be automatically subscribed to that order.

III.D Updating orders

At the moment of writing this document, orders cannot be updated also known as cancel-replace. They can just be created, cancelled and re-created. However, future versions pretend to implement this functionality.

III.E Subscribe Order Book

So far, we have dealt with synchronous operations. We send an order and we get the response immediately. However, the reason why exchanges usually use the Websocket technology is because its asynchronisity. This means, there are certain entities where the UI might want to know its evolution through time

+ The order book

+ The orders

+ Anything else whose state might change in different points in time

In this context, we have the order book subscription which is the process by which the UI is able to build the different levels of the order book in a way that is effective but also performant as we avoid sending more messages than necessary.

The subscription processes have the following basic 3 steps

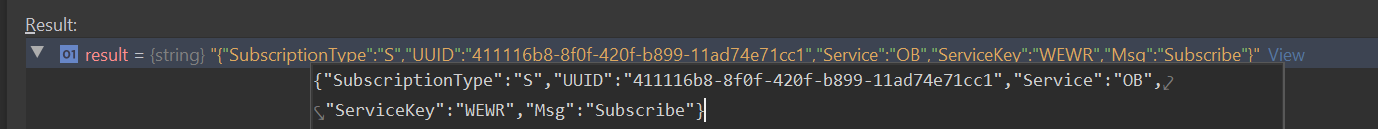
#1- Subscription

#2- Subscription response

#3- Asynchronously we will receive the update messages

III-E.#1 - Subscription message

The subscription message looks as follows

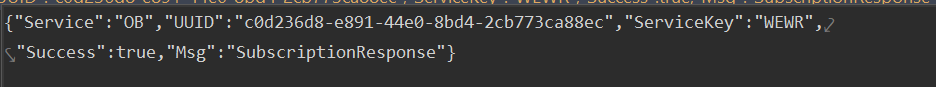


Where

* + **Msg**=”***Subscribe***”
  + **SubscriptionType** =”***S***” (“***S***” for subscriptions, “***U***” for unsubscriptions”)
  + **UUID**: GUID unique identifier of your client. 2 browsers/clients should have 2 different UUIDs
  + **Service**: “***OB***” (**OB** is for order book. Other subscriptions will have other keys)
  + **ServiceKey**: ”***WEWR***” (symbol of the security we are subscribing to )

III-E.#2 - Unsubscription message

After sending the previous message , you will receive a message like the following

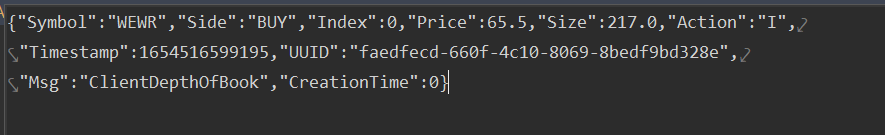


Where

* + **Msg**=”***SubscriptionResponse***” is indicating the message type
  + **SubscriptionType** =”***U***” (“***S***” for subscriptions, “***U***” for unsubscriptions”)
  + **UUID**: Unique identifier of the client sent in the subscription request. It is usually useful when many subscriptions are triggered through the same channel.
  + **Service**: “***OB***” (**OB** is for order book. Other subscriptions will have other keys)
  + **ServiceKey**: ”***WEWR***” (symbol of the security we are subscribing to )
  + **Success= true/false** If the subscription was successful or not
  + **Message:** Not present in the previous example, but here we would find the error description, in case ***Success=false***

III-E.#3 – Processing the order book messages.

After subscribing to the order book, you will start receiving messages like the following



Where

* + **Msg**=” ***ClientDepthOfBook***”
  + **CreationTime**= Epoch the json was created
  + **Timestamp**= Epoch the json was updated
  + **UUID**= The UUID Guid generated in the subscription so you are sure which client this message belongs to
  + **Symbol**= The subscribed symbol

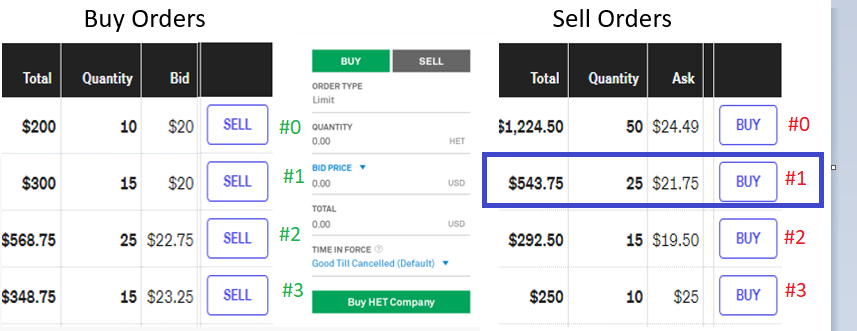
Then we have the most important fields

* Index
* Side: “BUY” or “SELL”
* Size
* Price
* Action: “I”(insert), “U”(update), “D” (delete)

In the following images we have 10 rows (5 buy and 5 sells), so every “*ClientDepthOfBook*” message will identify one specific row.

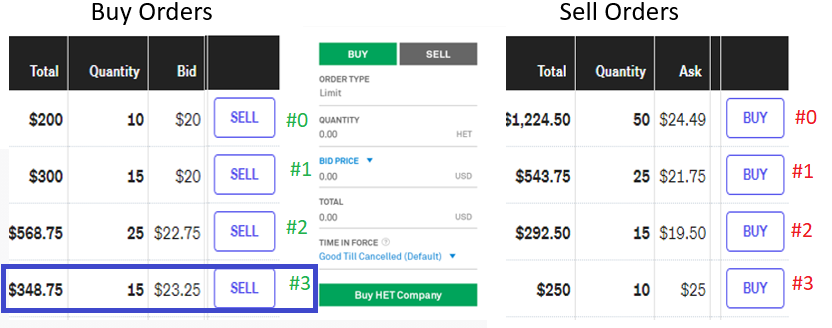
For example, the row in the blue rectangle, will be referred with a message with

* Side=”SELL”
* Index=1



On the same side, the following row in the blue rectangle, will be referred with

* Side=” BUY”
* Index=3



So basically, when you have the row identified, you know that you will have to take 3 actions

* Update the price (Bid or Ask)
  + You receive an **action**=“***I***” for an empty or filled row. You insert or update its ***Quantity*** and ***Bid*** with the ***Size*** and ***Price*** fields respectively
* Update the size (Quantity)
  + You receive an **action**=”***U***” for an already existing row. You update its ***Quantity*** and ***Bid*** with the ***Size*** and ***Price*** fields respectively
  + You receive an **action**=”***U***” for a row that is empty. That is just wrong. You show an error and prevent the order entry panel buttons from entering any order
* Delete the row
  + You receive an **action**=”***D***” for an already existing row. You clear that row. This means that the price and quantity of that row should be seen as a slash (“-“)
  + You receive an **action**=”***D***” for an empty row. That is just wrong and it is a bug alarm. You show an error and prevent the order entry panel buttons from entering any order.

Therefore, given the previous rules, If you update every price level following the previous rules for every message received you will have an order book constantly updated.

In practice, you will see that when you subscribe to any given order book, you will receive an initial snapshot with all the N price levels (5 price level in the test securities), followed by small sporadic updates, every time a new action, like someone creating an order, takes place.

III.F Subscribe Orders

The same way we have subscribed to the order book, there is another entity that will have a very specific cycle in any given exchange: the orders.

It is beyond the scope of this document to describe all the different statuses an order can have in an order book, but basically an order will go from creation, execution and then closing (filled, cancelled, expiring, etc.)

So in order of being notified of all the intermediate statuses, we will also have the following messages

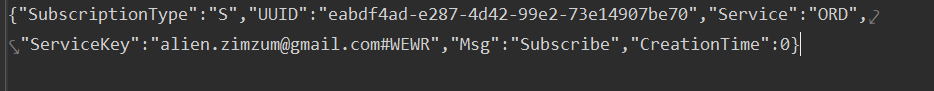
#1- Subscription

#2- Subscription response

#3- Asynchronously we will receive the update messages

III-F.#1 – Order Subscription message

The subscription message looks as follows



* Where
  + **Msg**=”***Subscribe***”
  + **SubscriptionType** =”***S***” (“***S***” for subscriptions, “***U***” for unsubscriptions”)
  + ***UUID***: GUID unique identifier of your client. 2 browsers should have 2 different UUIDs
  + **Service**: “***ORD***” (***ORD*** is for ALL the orders of that specific shareholder. Other subscriptions will have other keys)
  + **ServiceKey**:”***ShEmail@gmail.com#WEWR***” (symbol of the security we are subscribing to)

Where

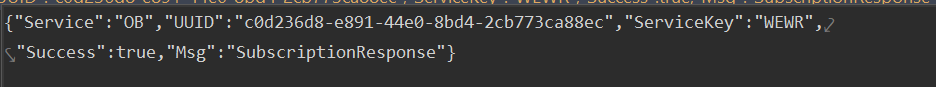
* [ShEmail@gmail.com](mailto:ShEmail@gmail.com) is the email of the logged shareholder for who you are requesting the subscription
* ***WEWR*** is the symbol that is open

Remarks

* In future CLOB engine versions, we will create the possibility to subscribe to ALL the orders of ALL the securities. For now we will deal, one security at a time.
* Make sure to implement the unsubscription message. Even when the CLOB engine will automatically unsubscribe you when he stops receiving ping messages from the client, it is a good practice to avoid receiving no longer used messages.

III-F.#2 - Unsubscription message

After the subscription you will receive

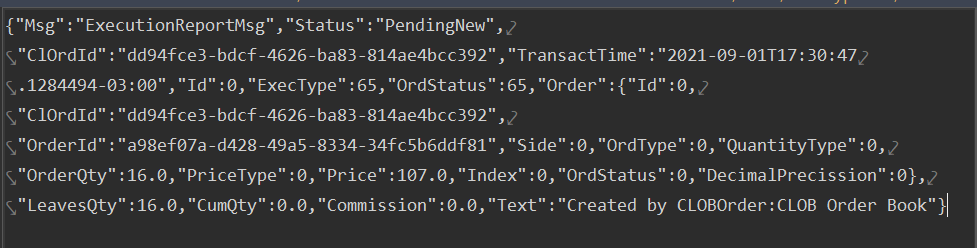


With all the same fields than III.F order book subscription message, but with the only difference than:

* **Service**=”***ORD***”
* **ServiceKey**=” [ShEmail@gmail.com#WEWR](mailto:ShEmail@gmail.com#WEWR)”

III-F.#3 – Processing the order subscription messages.

Once subscribed, you will receive json messages like the following



As you can see, the previous message fields have one outstanding characteristic: all the naming convention belong to the ***Execution Report*** (35=8) field name. Therefore, if you want to know, what the *ClOrdId* for is, you just have to go to the FIX protocol specification and search for this specific field. For further info, refer to section IV- The FIX protocol.

Therefore, the most important fields are

* Status: Order status in the exchange
* ClOrdId: It is the client order id
* Order Id : It the exchange order id assigned to the order

For more information about every field use and behavior , refer to the following link

<https://www.onixs.biz/fix-dictionary/4.4/msgtype_d_68.html>

In summary, after creating the order, every time a new event has taken place that changes one of the previous fields, a new **ExecutionReportMsg** will be received by the client.

**IV- The FIX protocol**

IV.A - Introduction

From its inception the FIX protocol was designed to be a secure and performant mechanism to transmit messages in a financial exchange.

There are three kind of messages that the protocol is prepared to deal with:

* Order flow messages
  + New Order Single (new order requests)
  + Order Cancel-Replaces (cancel/update orders)
  + Execution Reports (reports with all the information about what happened to an order through its lifecycle)

It is beyond the purpose of this document to provide a complete explanation of the FIX protocol implementation, as there are lot of documentation related to this protocol throughout the internet.

One of the best resources can be found in the following link which is used as a reference guide for FIX developers throughout the world

[**https://www.onixs.biz/**](https://www.onixs.biz/)

[**https://www.fixtrading.org/**](https://www.fixtrading.org/)

The protocol manages a client-server connection where the client is called an ***initiator*** and the server an ***acceptor***.

During the connection lifecycle, the *initiator* will connect to the *acceptor*, it will send (if needed) the login and password, and the *acceptor* will stay listening to receive any form of connection from the initiator, which are most commonly security list requests, market data requests or new order routings.

In FIX, fields do not have a descriptive name like the one we saw with the Websocket layer. It is mostly a collection of ***<FieldNumber>=<FieldValue>*** all chained in a single string so we how to know exactly what every field means and how different hierarchy behaves.

All these tags will be structured in 3 sections

* The header: we have all the fields here that describe the message like the message type (tag 35), the message sequence number for synchronization (tag 34), etc.
* The message fields with all the required message information. For example, when sending an order we will have the order client id (FIX tag 11), the order quantity (FIX tag 40), etc.
* There might be eventually some inner collections, like the order participants, an order book price levels, etc.

For example, the following is a message where a given initiator called ***TestClient*** (49=TestClient) connects (35=A) to the ***CLOB engine*** (56=CLOB\_ENGINE) and sends its credentials in fields 553 and 554.



As it was mentioned before, we will not go into too many details about all the FIX protocol nuances, but we will describe everything to know to establish a FIX connection in case you are already familiar with this protocol

IV.B- Establishing a connection

In the external client – Rialto ATS CLOB interaction, the external client will take the role of an Initiator.

This means that Rialto will assign him what is called ***SenderCompID*** that it will have to include in its FIX configuration files.

In this context, the client will have to

* Properly set an ***initiator.cfg*** file with the SenderCompID provided
* Prepare the ***stunnel*** connection
* Properly store a **FIX 5.0 SP2** application dictionary and transport data dictionary
* Initiate a session using the selected library (username and password)
* Send the user and password of the external client shareholder in the ATS
* Start sending orders

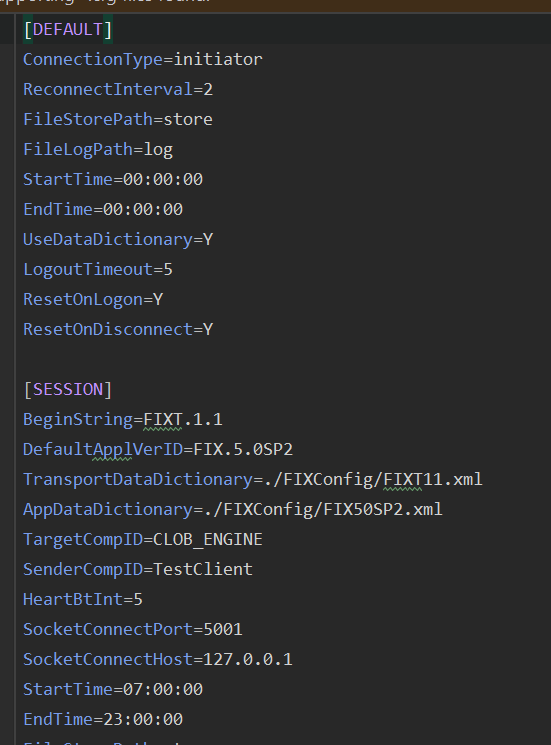
Following we show the most important parameters of all these different steps

IV.B.1- Preparing the initiator.cfg file

Every FIX library (like *Quickfix*) will have its own version of the ***initiator.cfg*** file. So the first step in every FIX connection would be to setup this file.

The following file is just an example and we will mention the most important parameters. However, as these parameters change through time, we will include a separate document with all the specific parameters at any given point in time.

The file is a simple text file of extension ***cfg*** that has a combination of *key-values* separated in two sections: ***Default*** (applies to all the acceptors) and ***Session*** (applies to a specific acceptor in case we want to have connections ready for multiple acceptors).



Where the most important parameters are

* We will have to provide in ***TransportDataDictionary*** and ***AppDataDictionary*** of the FIX dictionary files (see IV.B.1)
* Values of ***TargetCompID*** and ***SenderCompID*** will be provided upon real connection, depending on
  + Which Rialto environment are we connecting to
  + The ***SenderCompID*** assigned to a specific client
  + ***ResetOnLogon*** and ***ResetOnDisconnect*** should be Y as all the sequence number are reset after every session. This is just a technical parameter that defines that tag 34 (message sequence number) will always start from one after every connection and reconnection
  + ***StartTime* and *Endtime*** will be provided at the time of receiving the credentials, as this will depend on the exchange operational timetables.
  + ***SocketConnectHost*** and ***SocketConnectPort*** should be the one set in the stunnel app which will grant an encrypted connection.

For more information about all the fields in the initiator.cfg files you can refer to the following link

<http://quickfixn.org/tutorial/configuration.html>

[Home • FIX Trading Community](https://www.fixtrading.org/)

But remember, there will be a specific document describing all the specific parameters for every environment.

IV.B.2- Preparing the stunnel connection

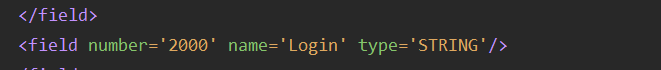
**<TO BE COMPLETED: if stunnel if the technology implemented or different alternatives are considered>**

IV.B.3- Application Data Dictionary and Transport Data Dictionary

A data dictionary is a file that describes the structure of all the fields that form every FIX message so that both parties agree that they will be using the same fields, and no missing fields or unknown fields will unexpectedly appear.

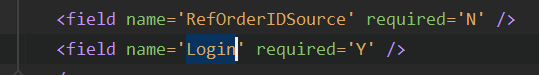
In the Rialto ATS CLOB engine, we will be using version 5.0 SP2 of the FIX protocol. So you just safely store this file in a folder and populate its relative path to the ***TransportDataDictionary*** and ***AppDataDictionary*** keys in the initiator.cfg file.

This way, all the messages that are created or received will be validated against these files and they will be rejected if they are malformed or unexpected or missing tags are founds.

The only custom change that has to be implemented for these files is the custom tag ***Login***, which should be appended at the end of the ***AppDataDictionary*** file.

This file will be part of the *NewOrderSingle* message (**35=D**) to specify the user’s login of a specific order.

So , at the end of the NewOrderSingle message we will have to append this tag



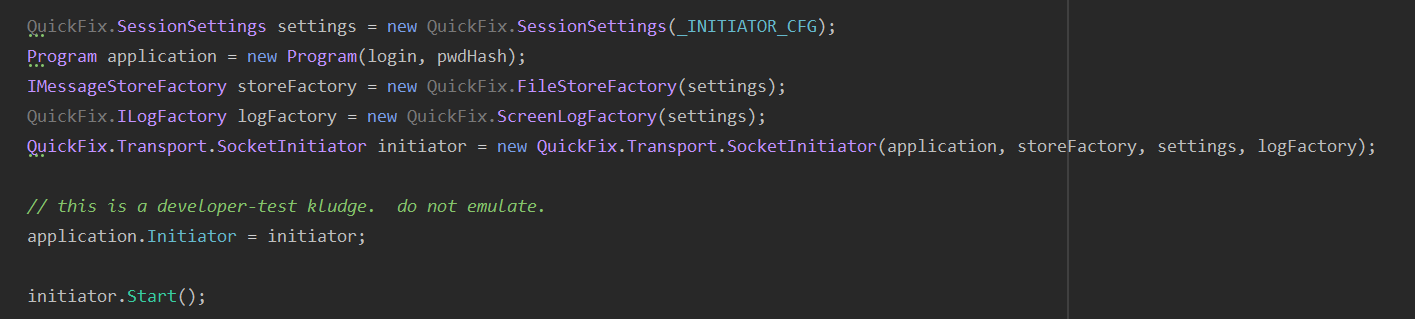
If you do not know how to do this, you can ask the customized ***FIX 5.0 SP2*** dictionary files and they will be provided.

IV.B.4 – Initiate a Session

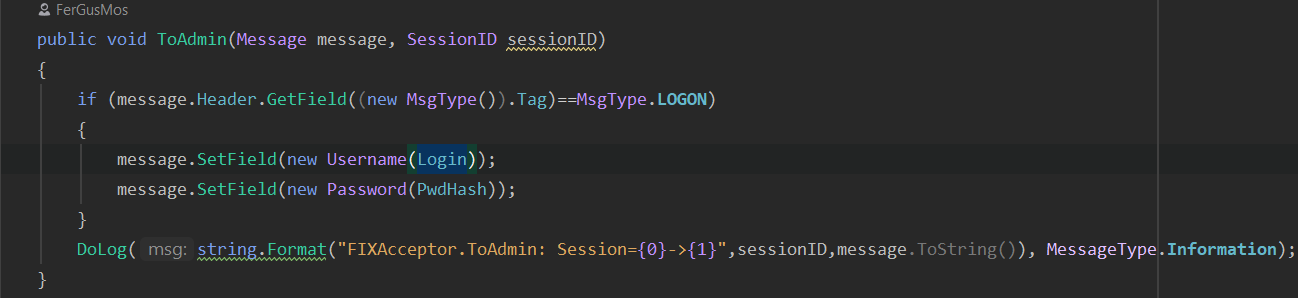
Once all the configuration files have been properly setup, you will have to write the code to initiate a session with the acceptor.

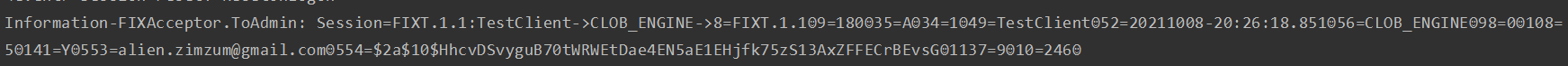
There is plenty of information about how to do this in every specific language, but the process is always the same

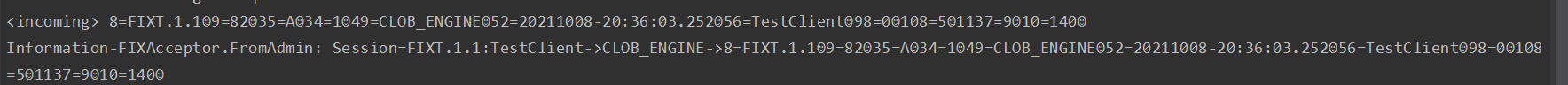
* Instantiate an ***Initiator*** class or similar
* Invoke the ***Start*** method or similar of this class
* A message flow of session initiator message (***35=A***) will start flowing between both counterparties until a connection is established
* From that point on, session keep alive messages will flow between the initiator and the acceptor (***35=0),*** which are messages meant to validate that the other side of the connection is still active
* This is usually done automatically by the FIX library implemented (ex: ***Quickfix*** library) and no coding or specific management should be done.

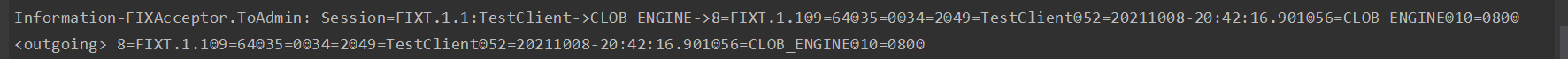
For example, the following snippet creates a connection in the C# language using the ***Quickfix*** library

And the following code, appends the external client login and username just before sending the session initiation message (35=A).



For example, in the following message we can see the *Quickfix* log where a *SenderCompID* named *TestClient* send a logging message (***35=A***) and send the credentials for a user whose login is [alien.zimzum@gmail.com](mailto:alien.zimzum@gmail.com).

Immediately, the engine (***49=CLOB\_ENGINE***) answers that the authentication has been successfully processed (***35=A***)

After the connection has been successfully established, nothing will happen until orders are sent to the exchange. A ping will take place every 5 seconds (see HeartBtInt variable in the config file)

If there were a problem with the authentication, the engine would have returned a message ***35=5*** indicating that the connection was rejected and then it would immediately close the connection.

IV.B.5 – Sending Orders

Sending orders in the FIX protocol is done through what is called **a *New Order Single*** message (**35=D**). Of course, you can send orders that are more complex but in the first version of the FIX layer, we will start dealing with simple order’s scenarios.

Once we are authenticated (**msg 35=A)** we will be able to start sending orders and receiving execution reports for the orders that we sent. This means, that at the time of writing this document, we are not able to subscribe to all my ***execution reports*** but the ones that belong to the orders that I have sent to the exchange in the current session.

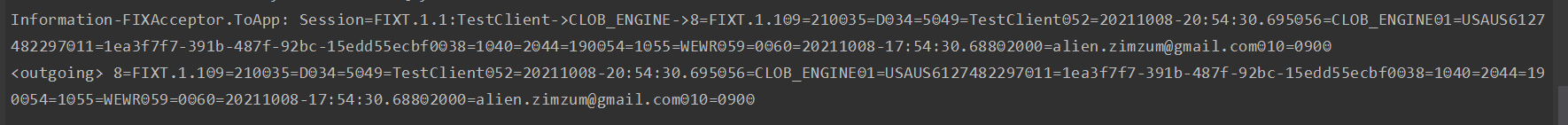
The fields that to be filled when sending a new order are (see ***https://www.onixs.biz/fix-dictionary/4.4/msgtype\_d\_68.html***)

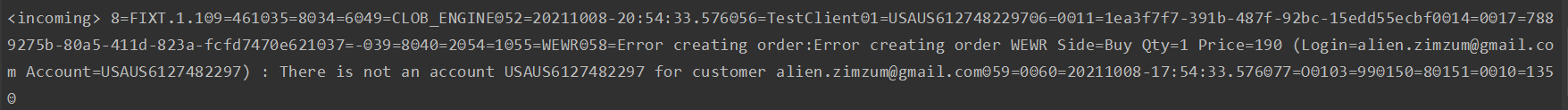
* **Symbol**: Symbol that the order is addressed to. If the symbol does not exist, you will later receive a Rejected execution report
* **Login**: The custom field ***2000*** referenced in section IV.B.3
* **Price**: The price of the limit order
* **Size**: The number of shares to buy or sell
* **Side**: The side of the order (Buy or Sell).
  + [Home • FIX Trading Community](https://www.fixtrading.org/)
  + See <https://www.onixs.biz/fix-dictionary/4.4/tagNum_54.html>
  + There are additional sides but the Rialto FIX layer only supports buys and sells *at this time…*
* **OrdType**:
  + https://www.onixs.biz/fix-dictionary/4.4/tagNum\_40.html
  + This field has to always be Limit in the current version of the FIX layer
* **ClOrdId**: Client order id that the client assigns to the order. Remember that there are two ids and this be the id that the client assigns to the order.
* **Account**: The ATS account the order is addressed. Otherwise, the default account will be used.
* **TimeInforce**:
  + Expiration of the order
  + At the time of writing this document only Day and GTC time inforce are implemented
  + See https://www.onixs.biz/fix-dictionary/4.4/tagNum\_59.html

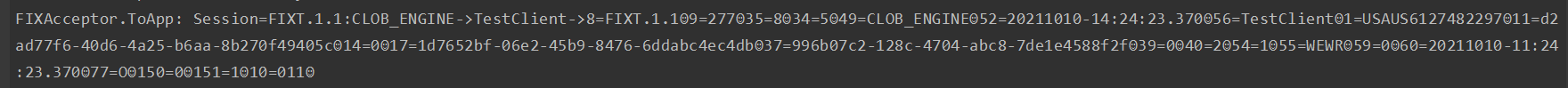
After sending an order to the exchange, you will start receiving execution reports updating the status of that order in the exchange. This order could be rejected, accepted (New), partially filled, filled, expired, etc.

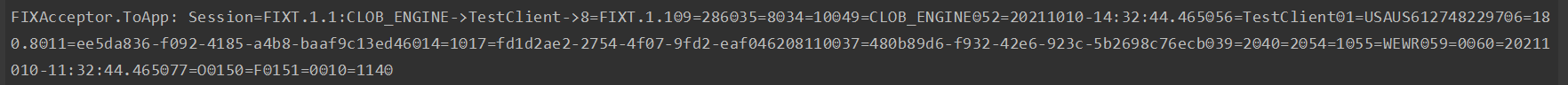
This will be communicated through the execution report message which is described in the following link:

* + https://www.fixtrading.org/online-specification/business-area-trade/

For example, the following message shows a new order (***35=D***) being sent to the exchange for the security ***WEWR***, for 1 share by 190$:

After sending an order is sent, we will start receiving execution reports (***35=8***) indicating what happened with that order in the exchange. For example, the following message rejects the order because we provided the wrong account:

But if the order had been accepted in the exchange, we would have received a ***New*** order status (***39=0***)

 If there were a trade, we would receive an execution report like the following:

See how *ExecType* field (***tag 150***) shows that there was a Trade (***150=F***), which make the order to go into order status (***tag 39***) Filled (***39=2***)

This is the basic message flow for the FIX layer. Future versions will incorporate new messages like market data and security lists.