Package 'aqr'

July 2, 2014

Version 0.4

Date 2014-02-04

Title Interface methods to use with an ActiveQuant Master Server

Author Ulrich Staudinger, ActiveQuant GmbH

Maintainer Ulrich Staudinger <ustaudinger@activequant.com>

Depends R (>= 2.1), xts, RCurl

Description This R extension provides methods to use a standalone ActiveQuant Master Server from within R. Currently available features include fetching and storing historical data, receiving and sending live data. Several utility methods for simple data transformations are included, too. For support requests, please join the mailing list at https://r-forge.r-project.org/mail/?group_id=1518

License GPL (>= 2)

URL http://www.activequant.com

Repository CRAN

Repository/R-Forge/Project aqr

Repository/R-Forge/Revision 61

Repository/R-Forge/DateTimeStamp 2014-03-01 14:12:50

Date/Publication 2014-03-01 18:18:11

NeedsCompilation yes

R topics documented:

`aqr-package` *Package level introduction*

Description

This package provides an R interface for using an AQ Master Server (AQMS). Preferrably, this package is used in conjunction with an AQMS, although the messaging layer works with any STOMP compliant messaging server, too. While I do not want this text to become an advertorial for AQMS, it is unavoidable to refer to AQ and AQMS.

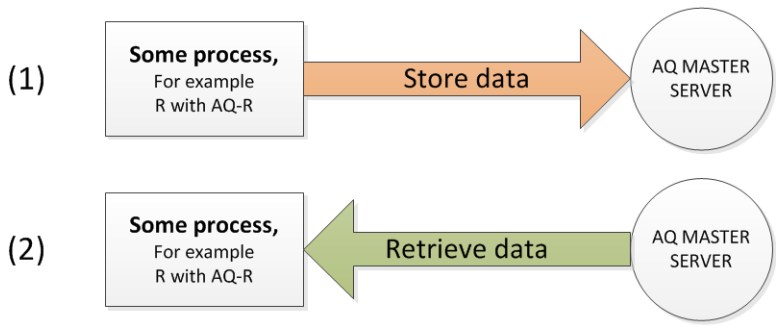
Some remarks upfront. An AQMS instance does not include data sources or data fetchers, it is a simple and dumb data store and data relay, built on open source components. Connectors to venues such as Yahoo, Bloomberg, Reuters, etc., are not within this extension's scope. The #1 rule to keep in mind: *what is inside, can go out*.

AQ-R tries to maintain a consistent variable naming scheme. Throughout this package, you will encounter the terms *seriesId*, *fieldId* and *channel*. All these terms are plain names, which you give meaning to and which you can choose freely within certain sanity boundaries. Note, that ActiveQuant itself generates IDs for instruments automatically. As soon as your R code interacts with other AQ components, these other components might specify the IDs of instruments. As we deal in this tutorial only with R, we are going to skip the technical details of how auto-generated IDs look like in the Java world and work with the fact that you are free to choose them for your own purposes. As said before, the server itself is very dumb and does not enforce a naming pattern or consistency between instrument definitions and timeseries data. What goes in, can go out.

This introductory section is separated into two parts, a) historical data and b) messaging realtime data.

Historical data

AQ-R provides methods to store and fetch historical timeseries data with an AQMS. Keep rule #1 in mind, you can't fetch what isn't in. So, in order to load 1 minute, 5 minute, 1 hour or tick data, data has to be to put in. There exist some ready-made data feeders within AQ, but you are free to write your own in python, Java or in R Although the AQMS interfaces are cross language compatible, we focus in this text on R. This basic structure of data feeders and data consumers is shown in the next figure.



AQMS is built on HBase and Hadoop, an ultra-scalable NoSQL solution which enables you to build large storage clusters capable of handling Petabytes of data. Try that with plain file-based storage of HDF5 files. But let's move on. Also with HBase, data gets separated into tables, rows and columns. Specific to the AQMS approach is that time series data is stored into one table per timeframe. This means, all `timeFrame = RAW` data goes into the RAW table, all `timeFrame = EOD` goes into the EOD table, etc. There is no logical enforcement that all data is indeed of the specified granularity, but there is a logical enforcement that table names are of specific values only. It is for the time being within the responsibility of the user to put data where it belongs. A series can contain an arbitrary amount of fields. The *seriesId* specifies the logical name of the series, typically it contains

the instrument ID, but it is literally just a string used to identify. Examples of a seriesId are *CNX.MDI.EURUSD* or *BBG.FUT.GXZ12_INDEX*. Let's move on to fields. FieldIDs, similar to seriesIDs are plain strings used to identify a field within a series. The user is responsible for maintaining a naming scheme, within the data feeders of AQ, we use the same field naming conventions. Part of the convention is to use only upper-case field names. Examples of field names are *OPEN*, *HIGH*, *PX_SETTLE*, *SMA10*, *IMPLVOL*, etc., but these are just examples. In case of doubt, rule #1 applies: what goes in, goes out.

Tutorial

In the context of this tutorial, we assume you have your AQMS server up and running. At first we will create a small script that uses quantmod to fetch end-of-day historical data from Yahoo. We will then store that data in AQMS. Because it is so much fun, we will also calculate the simple-moving-average and store this one in AQMS, too. As the final step, we will write another script and fetch former stored data from AQMS.

Let's fetch data for Microsoft and SAP from Yahoo.

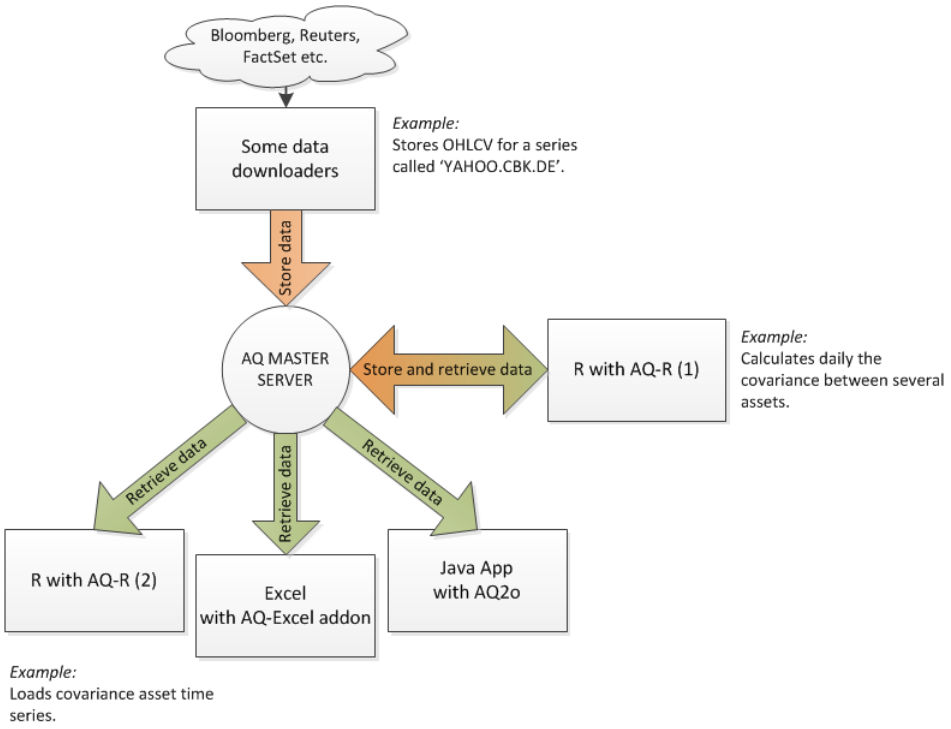
```
require(aqr)
require(quantmod)
# fetch them via quantmod
getSymbols(c("MSFT", "SAP"))
# visual check
candleChart(MSFT)
candleChart(SAP)
# we have to clean the column names of quantmod.
colnames(MSFT) <- c("OPEN", "HIGH", "LOW", "CLOSE", "VOLUME", "ADJUSTED")
colnames(SAP) <- c("OPEN", "HIGH", "LOW", "CLOSE", "VOLUME", "ADJUSTED")
# store them.
aqStoreMatrix("myMSFT", "EOD", MSFT)
aqStoreMatrix("mySAP", "EOD", SAP)
```

Once data has been stored in AQMS, it is much faster to retrieve data in the future from AQMS than it is to fetch it from Yahoo or Google. Keep in mind that some providers' data usage policies prohibit storing data locally.

Now let's assume we are in a new R session. We'll first load the data from yesterday and will then calculate the SMAs and store these, too.

```
#let's load what we stored.
aqLoadOHLC("myMSFT", "EOD", 19900101, 20200101)
aqLoadOHLC("mySAP", "EOD", 19900101, 20200101)
# let's calc SMAs
smaMsft = SMA(MSFT[,4])
smaSap = SMA(SAP[,4])
# let's store it.
aqStoreSeriesField("MSFT", "SMA14", "EOD", smaMsft)
# It should say: Wrote 1478 lines.
aqStoreSeriesField("SAP", "SMA14", "EOD", smaSap)
# let's load the SMA series that we stored.
aqLoadSeriesField("MSFT", "SMA14", "EOD", 19900101, 20200101)
```

More complex scenario The following figure shows you a more developed setup for historical data, where instead of R, other applications, like Excel play the role of data consumers. The builtin cross-language support of AQMS enables R applications to share data through AQMS with other environments, for exampe Excel - imagine some R processes calculating some risk parameters and some other non-scientific person viewing this data without installing ODBC drivers, etc. The AQMS contains an CSV-over-HTTP interface, which returns data in CSV format, so that any application, able to view a webpage can access data. Isn't that neat? And way easier than SQL, ODBC or other fancy technology, but that's all for now.

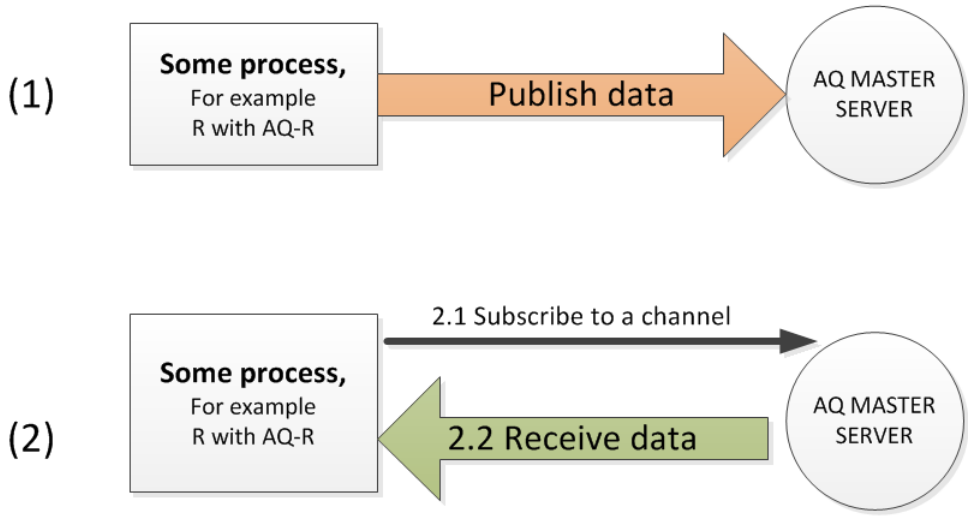


Messaging realtime data

Messaging happens in channels. All messages sent to a channel are broadcasted to all subscribers. Several data consumers can subscribe to the same channel and several data producers can publish into the same channel. Subscribers subscribe to a channel by specifying the *channel name* to which they would like to subscribe.

The channel name is not governed by conventions, although some data feeders use similar naming conventions. These channel names are plain string representations, for example "TEXT", "PNL", "CNX.MDI.EURUSD". The messages transmitted in a channel are not standardized either, although some data feeders (particularly the AQ data feeders) send messages in a consistent, google protocol buffers based format.

Using the messaging solution involves always the same flow. Some data consumer has subscribed to a channel. Some data publisher sends a message to a channel. All subscribed data consumers will receive this message. The following diagram summarizes this.



While sending data is a trivial call to `aqSend()`, receiving messages involves not only subscribing to a channel, but also either waiting for data or looking for data at regular intervals. The call to `aqWaitForData()` is a blocking call and will return a list of channels for which data is available. A subsequent call to `aqPoll()` will return all data received since its last call. An event driven R script would always call `aqWaitForData()`, followed immediately by `aqPoll()`. A message independent system can call `aqPoll()` at regular intervals, for example as soon as some other computations conclude.

Technicalities

Feel free to skip the next paragraphs and go straight to the tutorials, if you are not so technical. To my knowledge - without checking ALL existing packages of R - there is no easy and generic way to do realtime messaging in R. This partly owing to the fact that R is single threaded. This means of course that at some point within the messaging infrastructure, some sort of buffering has to occur. AQ-R solves this by spawning a background thread in its C part, this messaging interface buffers a limited amount of incoming data until it has been processed by R. On the communication protocol side, AQ-R uses the STOMP protocol to implement a two-way messaging solution. Technically, you do not need to use AQMS, as any STOMP compliant messaging server may be used.

On the technical side, the default way to messaging is through a *topic*, rather than a *queue* - but queues are also implementable should there be a serious need. The distinction between a topic and a queue is, a topic is a broadcast to all subscribers in a channel, whereas a queue means the message gets sent to the next available subscriber.

Tutorial

In this tutorial we build a simple message producer and a simple message consumer. Assuming the latest AQMS is up and running on *localhost*, we need two R instances, one for sending and one for

receiving data. At first we write the data sender. Our data sender should send out a random number every second. The trivial code is shown next.

```
require(aqr)
while(1){
  # generate a message containing a number between 1 and 1000.
  msg = toString(sample(1000,1))
  # send the message to channel RAND_DAT_CHAN
  aqSend("RAND_DAT_CHAN", msg)
  # sleep for a second.
  Sys.sleep(1);
}
```

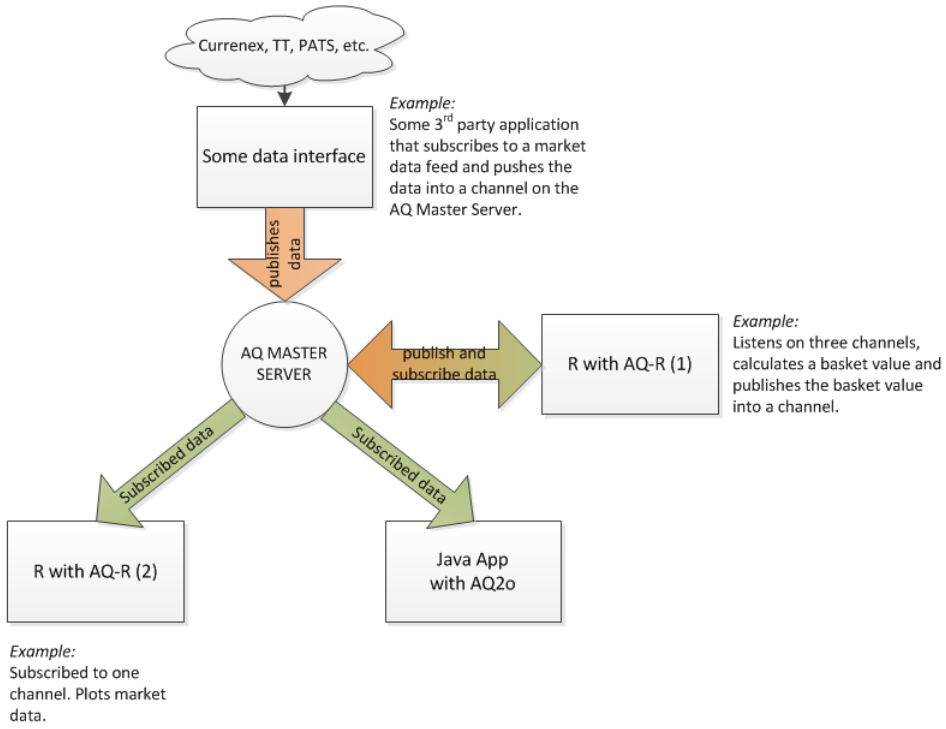
Now, let's build the receiver side. The two key function are `aqPoll()`, which will return at the time of this writing all received messages separated by a newline character and `aqWaitForData`, which is a blocking call and which will wait until data has been received. `aqPoll` will fetch all messages for all channels as a two dimensional matrix, one row corresponding to one channel. It is the responsibility of the R code to further process these messages.

In a new R instance, the following code will print the received message as soon as the event hits the R instance.

```
require(aqr)
aqSubscribeChannel("RAND_DAT_CHAN")
while(1){
  aqWaitForData()
  # fetch all data.
  text = aqPoll()
  # browser()
  message("Message received: ", text[,2])
}
```

Now that messages have been received, you could for example convert it to a double. The open nature of this messaging solution enables creating arbitrarily complex messaging scenarios. The only real restriction is a maximum message size of 4096 bytes within the R extension.

More complex scenario The following diagram presents a more complex messaging scenario with various data producers and consumers. Again, this messaging solution is not R specific.



Thanks for your attention, now on to the function documentation.

`approximateSLTP` *Approximates StopLoss/TakeProfit for a given PNL series and an HLC series.*

Description

This function uses the generatePnlCurve function to forward generate a PNL curve. Best, test it with your own series to understand how it works (or contribute documentation).

Usage

```
approximateSLTP (high, low, close, takeProfit, stopLoss, runningPosition, messages=FALSE)
```

Arguments

`high`	an XTS object containing the highs of a price series
`low`	an XTS object containing the lows of a price series
`close`	an XTS object containing the closes of a price series
`takeProfit`	an absolute value when to trigger a take profit action
`stopLoss`	an absolute value when to trigger a stop loss action
`runningPosition`	the running position
`messages`	whether to print informational message or not, defaults to FALSE

Value

a two column matrix with position and pnl, where position is a rewritten version of the input

---

`aqDataReady` *Is data ready?*

Description

This message retrieves a list of all messaging channels for which there is data. The result set is a list with channel names.

This function is a quick call. Other than the aqWaitForData method, this method will not block until data becomes available.

Usage

```
aqDataReady()
```

Examples

```
## Not run:

> aqDataReady()
     [,1]
[1,] "/topic/R-PROCESSOR-2"
>


## End(Not run)
```

`aqDayOfWeekStat` *applies a function to all values per weekday.*

Description

applies a function to all values per weekday.

Usage

```
aqDayOfWeekStat(x, f = mean)
```

Arguments

`x`	the input xts object
`f`	the function to apply

Value

a matrix that contains weekly figures

aqDisableDebugMessages *Disable debug messages*

Description

Disables debug messages in messaging related C-parts of AQ-R.

Usage

```
aqDisableDebugMessages()
```

aqDropHour *removes all data that belongs to a specific hour from an input data set.*

Description

removes all data that belongs to a specific hour from an input data set.

Usage

```
aqDropHour(x, hour)
```

Arguments

hour	the hour to remove from this data set, e.g. 8 or 15, etc.
x	an input xts object

Value

a dataset in which all information for this hour has been dropped.

aqDropHours *Drops data of several hours, delegates on to aqDropHour*

Description

Drops data of several hours, delegates on to aqDropHour

Usage

```
aqDropHours(x, hours)
```

Arguments

x	the input xts data set
hours	the vector of hours to drop

Value

the resulting data set

aqEnableDebugMessages *Enable debug messages*

Description

Enables debug messages in messaging related C-parts of AQ-R. Debug messages provide a variety of additional information, such as the raw messages received.

Usage

```
aqEnableDebugMessages()
```

`aqFilterOHLCSD` *Removes outliers based on standard deviation filters. Overwrites these with the open value.*

Description

Removes outliers based on standard deviation filters. Overwrites these with the open value.

Usage

```
aqFilterOHLCSD(ohlcv, sdFilterAmount = 10)
```

Arguments

`ohlcv`	an input Open/High/Low/Close/Volume dataset
`sdFilterAmount`	the amount of standard deviations a value has to be off, to be considered errenuous data

Value

retuns a filtered ohlcv object

`aqHourIndex` *Returns for an XTS input list the hour index per element.*

Description

Returns for an XTS input list the hour index per element.

Usage

```
aqHourIndex(xtsSeries)
```

Arguments

`xtsSeries`	the input object of type XTS.

Value

a vector of the same length as xtsSeries, containing the hour

`aqHourlyStat` *applies a function across hour slots. Internally, it iterates over 0:23 and selects all rows which fit into this hour.*

Description

applies a function across hour slots. Internally, it iterates over 0:23 and selects all rows which fit into this hour.

Usage

```
aqHourlyStat(x, f = mean)
```

Arguments

`x`	the input xts object
`f`	the function to apply

Value

a matrix that contains hourly data

`aqInit` *This method builds an object that carries necessary configuration values. The resulting object is a list, which you can modify outside of this function. Currently contains tsHost, tsPort, openField,closeField, highField, lowField and volField.*

Description

This method builds an object that carries necessary configuration values. The resulting object is a list, which you can modify outside of this function. Currently contains tsHost, tsPort, openField,closeField, highField, lowField and volField.

Usage

```
aqInit()
```

Value

This function returns a plain list with configuration settings.

aqInitMessaging *Initializes the messaging layer*

Description

This function can be used to specify a host and a port different from localhost and 61618. It is recommended practice to call this function before using AQ-R's messaging functionality.

Usage

```
aqInitMessaging(host = "localhost", port = 61618)
```

Arguments

host	the STOMP host
port	the STOMP port

aqLoadOHLC *Loads OHLC from an AQ Master Server*

Description

Loads OHLC from an AQ Master Server

Usage

```
aqLoadOHLC(seriesId, freq, startDate, endDate, con = aqInit(),
  useCache = FALSE, cacheDir = getwd())
```

Arguments

seriesId	a series ID
freq	frequency in enumeration form, f.e. HOURS_1, MINUTES_1
startDate	a Date8
endDate	a Date8
con	a fully initialized connection definition
useCache	a boolean that says whether you want use and cache data
cacheDir	a directory name that will be used for caching if enabled

Value

a XTS object

aqLoadSeriesField *Loads one series field from an AQ Master Server*

Description

Loads one series field from an AQ Master Server

Usage

```
aqLoadSeriesField(seriesId, fieldId, freq, startDate, endDate, con = aqInit())
```

Arguments

seriesId	the series name
fieldId	the field name
freq	the frequency, must be one of ActiveQuant's enums
startDate	a start date in date8 format (yyyyMMdd)
endDate	an end date in date8 format (yyyyMMdd)
con	a connection object

Value

the loaded series as XTS object

aqLoadXtsFromCsv *Loads a XTS object from CSV, to be used with our aqSaveXtsToCsv function. This method assumes that the fie's first column contains an interpretable timestmap.*

Description

Implementation in progress (16 Feb 2014)

Usage

```
aqLoadXtsFromCsv(filename)
```

Arguments

filename	the csv file which to load as XTS.

Value

an XTS object

aqLoadYahooEOD *Loads EOD data from Yahoo and returns an XTS object.*

Description

Loads EOD data from Yahoo and returns an XTS object.

Usage

```
aqLoadYahooEOD(instrument, start = oneMonthAgo(), end = today())
```

Arguments

instrument	a Yahoo Instrument ID
start	a POSIXlt start date
end	a POSIXlt end date

Value

instrument prices as XTS object

aqPoll *Poll data from the messaging bridge*

Description

Used to poll data from the messaging bridge. As the time of this writing (0.2), individual messages are separated by "\n". In a future version, this function will return a list object.

Usage

```
aqPoll()
```

Examples

```
## Not run:
> aqPoll()
     [,1]
[1,] "/topic/TEXT"
     [,2]
[1,] "TEST1\nTEST2\nTEST3\n"
>

## End(Not run)
```

aqSaveXtsToCsv *Saves an XTS object to csv file.*

Description

Saves an XTS object to csv file.

Usage

```
aqSaveXtsToCsv(filename, historyXts)
```

Arguments

filename	where to save data to
historyXts	the input xts object

aqSend *Send data to a messaging channel*

Description

Sends data as raw bytes to the messaging channel.

Usage

```
aqSend(channel, message)
```

Arguments

channel	input xts data set
message	a set of hours to drop

Examples

```
## Not run:
aqSend("R-PROCESSOR-1", "I am done.")

## End(Not run)
```

`aqStoreMatrix` *stores a matrix onto an AQ Master Server*

Description

stores a matrix onto an AQ Master Server

Usage

```
aqStoreMatrix(seriesId, freq, data, con = aqInit(), silent = FALSE)
```

Arguments

`seriesId`	a series ID to store
`freq`	the frequency, must be one of AQ's enums
`data`	the data as XTS object
`con`	a connection object, will be initialized by aqInit by default
`silent`	whether it should print storage diagnostics.

`aqStoreSeriesField` *Stores one series field to an AQ Master Server, typicall called from aqStoreSeries. This function assumes that data is either a zoo object, or that is a matrix with two columns where the first column contains a time series index in NANOSECONDS(!!!)*

Description

Stores one series field to an AQ Master Server, typicall called from aqStoreSeries. This function assumes that data is either a zoo object, or that is a matrix with two columns where the first column contains a time series index in NANOSECONDS(!!!)

Usage

```
aqStoreSeriesField(seriesId, fieldId, freq, data, con = aqInit(),
  silent = FALSE)
```

Arguments

`seriesId`	a series name
`fieldId`	the field ID of this data series
`freq`	a frequency string, must be one of AQ's suported enum names
`data`	the data as XTS object
`con`	a connection object, will be initialized by aqInit by default
`silent`	whether it should print storage diagnostics.

aqSubscribeChannel *Subscribe to a messaging channel*

Description

Subscribes to a messaging channel. Data will flow into the bridge and it will accumulate that data.

Usage

```
aqSubscribeChannel(channel)
```

Arguments

channel	one channel name

Examples

```
## Not run:

>aqSubscribeChannel("R-PROCESSOR-2")
[1] "Subscribed."
>


## End(Not run)
```

aqTestCallToDynLib *Test function*

Description

Tests whether the dynamic library works or not.

Usage

```
aqTestCallToDynLib(testMessage)
```

Arguments

testMessage	some test message

aqUnsubscribeChannel *Unsubscribe from a messaging channel*

Description

Unsubscribes from a messaging channel. The bridge will send an unsubscribe command to the messaging server.

Usage

```
aqUnsubscribeChannel(channel)
```

Arguments

channel	one channel name

aqWaitForData *Wait for data at the bridge*

Description

A blocking call that waits for data at the messaging bridge. The call will return a list of channels for which data is available, once the bridge contains data.

Usage

```
aqWaitForData()
```

buildArchiveURL *Builds an archive URL, based on connection parameters, seriesId, field, frequency and start and end date.*

Description

Builds an archive URL, based on connection parameters, seriesId, field, frequency and start and end date.

Usage

```
buildArchiveURL(con = aqInit(), seriesId, field, freq, startDate, endDate)
```

Arguments

con	connection parameters, will be initialized with aqInit() if void
seriesId	the series name
field	the field to load
freq	a frequency string, such as HOURS_1
startDate	the start date
endDate	the end date

Value

the complete archive URL as character

`generatePnlCurve` *method to generate a pnl curve from a running position. bids, asks and running position must have the same length. Can compute the pnl from one price to the other, but only for one asset! Does not take time into account - if you need signal delays, lag all input data on your own.*

Description

method to generate a pnl curve from a running position. bids, asks and running position must have the same length. Can compute the pnl from one price to the other, but only for one asset! Does not take time into account - if you need signal delays, lag all input data on your own.

Usage

```
generatePnlCurve(bidPrices, askPrices, runningPosition, messages = FALSE)
```

Arguments

bidPrices	an array of bid prices
askPrices	an array of ask prices
runningPosition	an array that contains a vector of the position
messages	specifies whether you want to have debug messages or not, defaults to FALSE

Value

This function returns a plain double array with pnl changes (uncumulated) and not an XTS series.

Note

all input arrays must have the same length.

oneMonthAgo *returns the date one month (30 days) ago as date8*

Description

returns the date one month (30 days) ago as date8

Usage

```
oneMonthAgo()
```

Value

a POSIXlt object pointing at thirty days ago

today *returns today as date8.*

Description

returns today as date8.

Usage

```
today()
```

Value

a POSIXlt object of now.

Index