QuantConnect Documentation - LEAN Engine

Created on 05/26/2023

Copyright QuantConnect 2023

Table of Content

- 1 Getting Started
- 2 Contributions
- 2.1 Datasets
- 2.1.1 Key Concepts
- 2.1.2 Defining Data Models
- 2.1.3 Rendering Data
- 2.1.3.1 Rendering Data with Python
- 2.1.3.2 Rendering Data with CSharp
- 2.1.3.3 Rendering Data with Notebooks
- 2.1.4 Testing Data Models
- 2.1.5 Data Documentation
- 2.2 Brokerages
- 2.2.1 Setting Up Your Environment
- 2.2.2 Laying the Foundation
- 2.2.3 Creating the Brokerage
- 2.2.4 Translating Symbol Conventions
- 2.2.5 Describing Brokerage Limitations
- 2.2.6 Enabling Live Data Streaming
- 2.2.7 Enabling Historical Data
- 2.2.8 Downloading Data
- 2.2.9 Modeling Fee Structures
- 2.2.10 Updating the Algorithm API
- 3 Statistics
- 3.1 Capacity
- 4 Class Reference

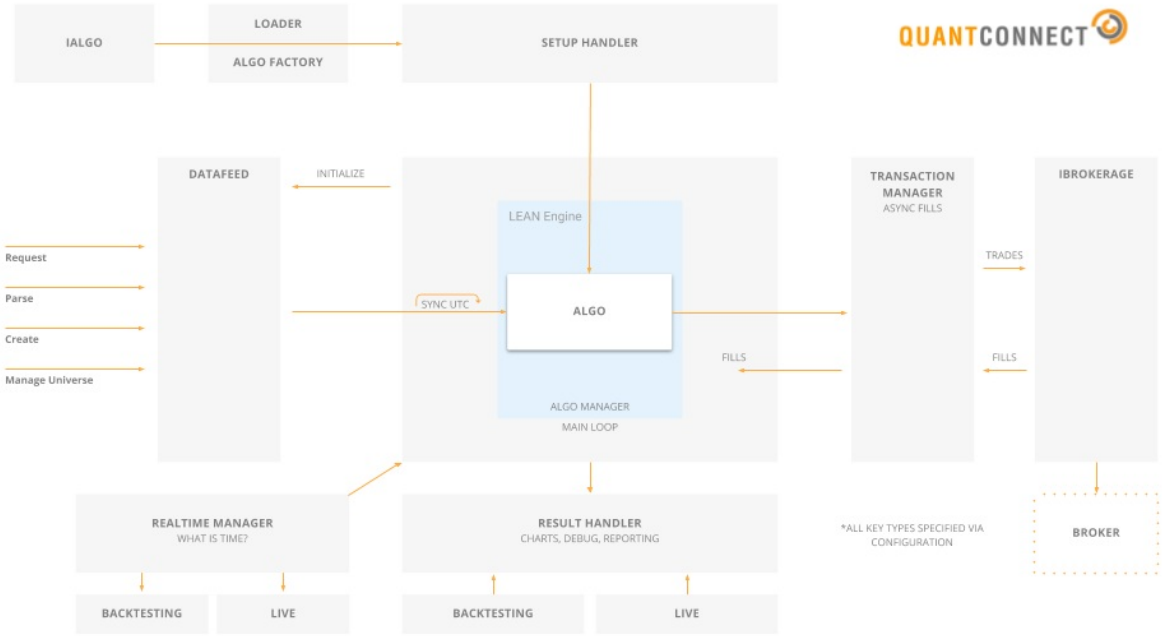
1 Getting Started

Introduction

Lean Engine is an open-source algorithmic trading engine built for easy strategy research, backtesting and live trading. We integrate with common data providers and brokerages so you can quickly deploy algorithmic trading strategies.

The core of the LEAN Engine is written in C#; but it operates seamlessly on Linux, Mac and Windows operating systems. It supports algorithms written in Python 3.8 or C#. Lean drives the web-based algorithmic trading platform QuantConnect .

System Overview



The Engine is broken into many modular pieces which can be extended without touching other files. The modules are configured in `config.json` as set "environments". Through these environments, you can control LEAN to operate in the mode required.

The most important plugins are:

Result Processing

An `IResultHandler` that handle all messages from the algorithmic trading engine. Decide what should be sent, and where the messages should go. The result processing system can send messages to a local GUI, or the web interface.

Datafeed Sourcing

An `IDataFeed` that connect and download the data required for the algorithmic trading engine. For backtesting this sources files from the disk, for live trading, it connects to a stream and generates the data objects.

Transaction Processing

An `ITransactionHandler` that process new order requests; either using the fill models provided by the algorithm or with an actual brokerage. Send the processed orders back to the algorithm's portfolio to be filled.

Realtime Event Management

An `IRealtimeHandler` that generate real-time events - such as the end of day events. Trigger callbacks to real-time event handlers. For backtesting, this is mocked-up a works on simulated time.

Algorithm State Setup

An `ISetupHandler` that configure the algorithm cash, portfolio and data requested. Initialize all state parameters required.

These are all configurable from the `config.json` file in the `Launcher` Project.

Developing with Lean CLI

QuantConnect recommends using Lean CLI for local algorithm development. This is because it is a great tool for working with your algorithms locally while still being able to deploy to the cloud and have access to Lean data. It is also able to run algorithms on your local machine with your data through our official docker images.

Reference QuantConnects documentation on Lean CLI here .

Installation Instructions

This section will cover how to install lean locally for you to use in your own environment.

Refer to the following readme files for a detailed guide regarding using your local IDE with Lean:

- VS Code

- VS

To install locally, download the zip file with the latest master and unzip it to your favorite location. Alternatively, install Git and clone the repo:

```
$ git clone https://github.com/QuantConnect/Lean.git
$ cd Lean
```

Mac OS

1. Install Visual Studio for Mac
2. Open `QuantConnect.Lean.sln` in Visual Studio

Visual Studio will automatically start to restore the Nuget packages. If not, in the menu bar,

1. click `Project > Restore NuGet Packages`
2. In the menu bar, click `Run > Start Debugging`

Alternatively, run the compiled `dll` file:

1. click `Build > Build All`
2. run the following code:

```
$ cd Lean/Launcher/bin/Debug
$ dotnet QuantConnect.Lean.Launcher.dll
```

Linux (Debian, Ubuntu)

1. Install dotnet 6
2. Compile Lean Solution

```
$ dotnet build QuantConnect.Lean.sln
```

3. Run Lean

```
$ cd Launcher/bin/Debug
$ dotnet QuantConnect.Lean.Launcher.dll
```

To set up Interactive Brokers integration, make sure you fix the `ib-tws-dir` and `ib-controller-dir` fields in the `config.json` file with the actual paths to the TWS and the IBController folders respectively. If after all you still receive connection refuse error, try changing the `ib-port` field in the `config.json` file from 4002 to 4001 to match the settings in your IBGateway/TWS.

Windows

1. Install Visual Studio
2. Open `QuantConnect.Lean.sln` in Visual Studio
3. Build the solution by clicking `Build Menu -> Build Solution`
4. Press `F5` to run

Python Support

A full explanation of the Python installation process can be found in the Algorithm.Python project.

Local-Cloud Hybrid Development

Seamlessly develop locally in your favorite development environment, with full autocomplete and debugging support to quickly and easily identify problems with your strategy. For more information please see the CLI documentation .

Sponsorships

Sponsor QuantConnect to support our developers as we improve a revolutionary quantitative trading platform LEAN, in an open, collaborative way. We will continue to level the playing field with industry-grade tools and data accessibility. We use sponsorship funds to achieve the following goals:

- To continue the development of LEANâ€™s infrastructure
- To create free, high-quality research and educational material
- To provide continued support for our community
- To make terabytes of data accessible in the Dataset Market
- To bring LEAN to global financial markets
- To increase live trading brokerage connections
- To connect more individuals with financial institutions so individuals can gain income for their ideas at scale

To become a QuantConnect sponsor, see the Sponsorship page on GitHub.

2 Contributions

2.1 Datasets

2.1.1 Key Concepts

Introduction

Listing Process

Datasets contributed to LEAN can be quickly listed in the QuantConnect Dataset Marketplace, and distributed for sale to more than 235,000 users in the QuantConnect community. To list a dataset, reach out to the QuantConnect Team for a quick review, then proceed with the data creation and process steps in the following pages.

Datasets must be well defined, with realistic timestamps for when the data was available ("point in time"). Ideally datasets need at least a 2 year track record and to be maintained by a reputable company. They should be accompanied with full documentation and code examples so the community can harness the data.

Data Sources

The `GetSource` method of your dataset class instructs LEAN where to find the data. This method must return a SubscriptionDataSource object, which contains the data location and format. We host your data, so the `transportMedium` must be `SubscriptionTransportMedium.LocalFile` and the `format` must be `FileFormat.Csv` .

TimeZones

The `DataTimeZone` method of your data source class declares the time zone of your dataset. This method returns a NodaTime .DateTimeZone object. If your dataset provides trading data and universe data, the `DataTimeZone` methods in your Lean.DataSource.<vendorNameDatasetName> / <vendorNameDatasetName>.cs and Lean.DataSource.<vendorNameDatasetName> / <vendorNameDatasetName>Universe.cs files must be the same.

Linked Datasets

Your dataset is linked if any of the following statements are true:

- Your dataset describes market price properties of specific securities (for example, the closing price of AAPL).
- Your alternative dataset is linked to individual securities (for example, the Wikipedia page view count of AAPL).

Examples of unlinked datasets would be the weather of New York City where data is not relevant to a specific security.

When a dataset is linked, it need to be mapped to underlying assets through time. The `RequiresMapping` boolean instructs LEAN to handle the security and ticker mapping issues.

2.1.2 Defining Data Models

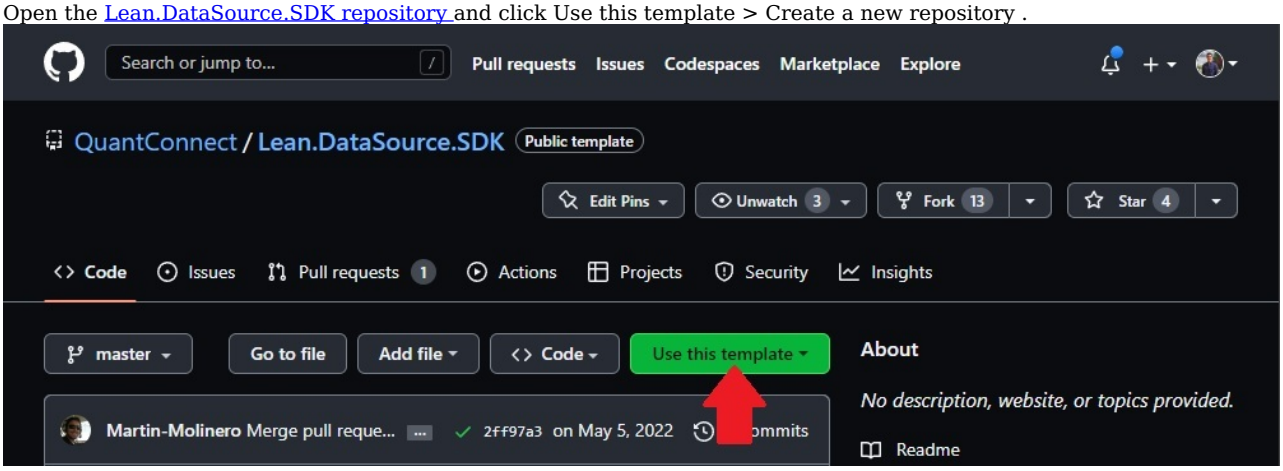
Introduction

This page explains how to set up the data source SDK and use it to create data models.

Part 1/ Set up SDK

Follow these steps to create a repository for your dataset:

1. Open the Lean.DataSource.SDK repository and click Use this template > Create a new repository .



   Start with the SDK repository instead of existing data source implementations because we periodically update the SDK repository.

2. On the Create a new repository from Lean.DataSource.SDK page, set the repository name to Lean.DataSource.<vendorNameDatasetName> (for example, Lean.DataSource.XYZAirlineTicketSales).

   If your dataset contains multiple series, use <vendorName> instead of <vendorNameDatasetName> . For instance, the Federal Reserve Economic Data (FRED) dataset repository has the name Lean.DataSource.FRED because it has many different series .

3. Click Create repository from template .
4. Clone the Lean.DataSource.<vendorNameDatasetName> repository.

   ```
   $ git clone https://github.com/username/Lean.DataSource.<vendorNameDatasetName>.git
   ```

5. If you're on a Linux terminal, in your Lean.DataSource.<vendorNameDatasetName> directory, change the access permissions of the bash script.

   ```
   $ chmod +x ./renameDataset
   ```

6. In your Lean.DataSource.<vendorNameDatasetName> directory, run the renameDataset.sh bash script.

   ```
   $ renameDataset.sh
   ```

   The bash script replaces some placeholder text in the Lean.DataSource.<vendorNameDatasetName> directory and renames some files according to your dataset's <vendorNameDatasetName>.

Part 2/ Create Data Models

The input to your model should be a CSV file that's in chronological order.

```
1997-01-01,905.2,941.4,905.2,939.55,38948210,978.21
1997-01-02,941.95,944,925.05,927.05,49118380,1150.42
1997-01-03,924.3,932.6,919.55,931.65,35263845,866.74
...
2014-07-24,7796.25,7835.65,7771.65,7830.6,117608370,6271.45
2014-07-25,7828.2,7840.95,7748.6,7790.45,153936037,7827.61
2014-07-28,7792.9,7799.9,7722.65,7748.7,116534670,6107.78
```

Follow these steps to define the data source class:

1. Open the Lean.DataSource.<vendorNameDatasetName> / <vendorNameDatasetName>.cs file.
2. Follow these steps to define the properties of your dataset:
   1. Duplicate lines 32-36 for as many properties as there are in your dataset.
   2. Rename the `SomeCustomProperty` properties to the names of your dataset properties (for example, `Destination`).
   3. If your dataset is a streaming dataset like the Benzinga News Feed , change the argument that is passed to the `ProtoMember` members so that they start at 10 and increment by one for each additional property in your dataset.
   4. If your dataset isn't a streaming dataset, delete the `ProtoMember` members.
   5. Replace the â€œSome custom data propertyâ€ comments with a description of each property in your dataset.

   If your dataset contains multiple series, like the FRED dataset , create a helper class file in Lean.DataSource.<vendorNameDatasetName> directory to map the series name to the series code. For a full example, see the LIBOR.cs file in the Lean.DataSource.FRED repository. The helper class makes it easier for members to subscribe to the series in your dataset because they don't need to know the series code. For instance, you can subscribe to the 1-Week London Interbank Offered Rate (LIBOR) based on U.S. Dollars with the following code snippet:

   ```
   AddData<Fred>(Fred.LIBOR.OneWeekBasedOnUSD);
   // Instead of
   // AddData<Fred>("USD1WKD156N");

   self.AddData(Fred, Fred.LIBOR.OneWeekBasedOnUSD)
   # Instead of
   # self.AddData(Fred, "USD1WKD156N")
   ```

3. Define the GetSource method to point to the path of your dataset file(s).

   Set the file name to the security ticker with `config.Symbol.Value` , which is the string value of the argument you pass to the AddData method

when you subscribe to the dataset. An example output file path is / output / alternative / xyzairline / ticketsales / dal.csv .

4. Define the `Reader` method to return instances of your dataset class.

   Set `Symbol = config.Symbol` and set `EndTime` to the time that the datapoint first became available for consumption.

   Your data class inherits from the `BaseData` class, which has `Value` and `Time` properties. Set the `Value` property to one of the factors in your dataset. If you don't set the `Time` property, its default value is the value of `EndTime` . For more information about the `Time` and `EndTime` properties, see Periods .

5. Define the DataTimeZone method.

```
public class VendorNameDatasetName : BaseData
{
    public override DateTimeZone DataTimeZone()
    {
        return DateTimeZone.Utc;
    }
}
```

   If you import `using QuantConnect` , the `TimeZones` class provides helper attributes to create `DateTimeZone` objects. For example, you can use `TimeZones.Utc` or `TimeZones.NewYork` . For more information about time zones, see Time Zones .

6. Define the `SupportedResolutions` method.

```
public class VendorNameDatasetName : BaseData
{
    public override List<Resolution> SupportedResolutions()
    {
        return DailyResolution;
    }
}
```

   The `Resolution` enumeration has the following members:

7. Define the `DefaultResolution` method.

   If a member doesn't specify a resolution when they subscribe to your dataset, Lean uses the `DefaultResolution` .

```
public class VendorNameDatasetName : BaseData
{
    public override Resolution DefaultResolution()
    {
        return Resolution.Daily;
    }
}
```

8. Define the `IsSparseData` method.

   If your dataset is not tick resolution and your dataset is missing data for at least one sample, it's sparse. If your dataset is sparse, we disable logging for missing files.

```
public class VendorNameDatasetName : BaseData
{
    public override bool IsSparseData()
    {
        return true;
    }
}
```

9. Define the RequiresMapping method.

```
public class VendorNameDatasetName : BaseData
{
    public override bool RequiresMapping()
    {
        return true;
    }
}
```

10. Define the `Clone` method.

```
public class VendorNameDatasetName : BaseData
{
    public override BaseData Clone()
    {
        return new VendorNameDatasetName
        {
            Symbol = Symbol,
            Time = Time,
            EndTime = EndTime,
            SomeCustomProperty = SomeCustomProperty,
        };
    }
}
```

11. Define the `ToString` method.

```
public class VendorNameDatasetName : BaseData
{
    public override string ToString()
    {
        return $"{Symbol} - {SomeCustomProperty}";
    }
}
```

Part 3/ Create Universe Models

If your dataset doesn't provide universe data, follow these steps:

1. Delete the Lean.DataSource.<vendorNameDatasetName> / <vendorNameDatasetName>Universe.cs .

2. Delete the Lean.DataSource.<vendorNameDatasetName> / <vendorNameDatasetName>UniverseSelectionAlgorithm.* files.
3. In the Lean.DataSource.<vendorNameDatasetName> / tests / Tests.csproj file, delete the code on line 8 that compiles the universe selection algorithms.
4. Skip the rest of this page.

The input to your model should be a CSV file where the first column is the security identifier and the second column is the point-in-time ticker.

```
A R735QTJ8XC9X,A,17.19,109700,1885743,False,0.9904858,1
AA R735QTJ8XC9X,AA,71.25,513400,36579750,False,0.3992678,0.750075
AAB R735QTJ8XC9X,AAB,16.38,5000,81900,False,0.9902758,1
...
ZSEV R735QTJ8XC9X,ZSEV,10.5,800,8400,False,0.8981684,1
ZTR R735QTJ8XC9X,ZTR,9.56,102300,977988,False,0.0803037,3.97015016
ZVX R735QTJ8XC9X,ZVX,10,15600,156000,False,1,0.666667
```

Follow these steps to define the data source class:

1. Open the Lean.DataSource.<vendorNameDatasetName> / <vendorNameDatasetName>Universe.cs file.
2. Follow these steps to define the properties of your dataset:
    1. Duplicate lines 33-36 or 38-41 (depending on the data type) for as many properties as there are in your dataset.
    2. Rename the `SomeCustomProperty` / `SomeNumericProperty` properties to the names of your dataset properties (for example, `Destination` / `FlightPassengerCount`).
    3. Replace the â€œSome custom data propertyâ€ comments with a description of each property in your dataset.
3. Define the GetSource method to point to the path of your dataset file(s).

    Use the `date` parameter as the file name to get the date of data being requested. An example output file path is / output / alternative / xyzairline / ticketsales / universe / 20200320.csv .

4. Define the `Reader` method to return instances of your universe class.

    The first column in your data file must be the security identifier and the second column must be the point-in-time ticker. With this configuration, use `new Symbol(SecurityIdentifier.Parse(csv[0]), csv[1])` to create the security `Symbol` .

    The date in your data file must be the date that the data point is available for consumption. With this configuration, set the `Time` to `date - Period` .

5. Define the DataTimeZone method.

    ```
    public class VendorNameDatasetNameUniverse : BaseData
    {
        public override DateTimeZone DataTimeZone()
        {
            return DateTimeZone.Utc;
        }
    }
    ```

    If you import `using QuantConnect` , the `TimeZones` class provides helper attributes to create `DateTimeZone` objects. For example, you can use `TimeZones.Utc` or `TimeZones.NewYork` . For more information about time zones, see Time Zones .

6. Define the `SupportedResolutions` method.

    ```
    public class VendorNameDatasetNameUniverse : BaseData
    {
        public override List<Resolution> SupportedResolutions()
        {
            return DailyResolution;
        }
    }
    ```

    Universe data must have hour or daily resolution.

    The `Resolution` enumeration has the following members:

7. Define the `DefaultResolution` method.

    If a member doesn't specify a resolution when they subscribe to your dataset, Lean uses the `DefaultResolution` .

    ```
    public class VendorNameDatasetNameUniverse : BaseData
    {
        public override Resolution DefaultResolution()
        {
            return Resolution.Daily;
        }
    }
    ```

8. Define the `IsSparseData` method.

    If your dataset is not tick resolution and your dataset is missing data for at least one sample, it's sparse. If your dataset is sparse, we disable logging for missing files.

    ```
    public class VendorNameDatasetNameUniverse : BaseData
    {
        public override bool IsSparseData()
        {
            return true;
        }
    }
    ```

9. Define the RequiresMapping method.

    ```
    public class VendorNameDatasetNameUniverse : BaseData
    {
        public override bool RequiresMapping()
        {
            return true;
        }
    }
    ```

10. Define the `Clone` method.

```
public class VendorNameDatasetNameUniverse : BaseData
{
    public override BaseData Clone()
    {
        return new VendorNameDatasetName
        {
            Symbol = Symbol,
            Time = Time,
            EndTime = EndTime,
            SomeCustomProperty = SomeCustomProperty,
        };
    }
}
```

11. Define the `ToString` method.

```
public class VendorNameDatasetNameUniverse : BaseData
{
    public override string ToString()
    {
        return $"{Symbol} - {SomeCustomProperty}";
    }
}
```

2.1.3 Rendering Data

2.1.3.1 Rendering Data with Python

Introduction

This page explains how to create a script to download and process your dataset with Python for QuantConnect distribution.

Using Processing Framework

During this part of the contribution process, you need to edit the Lean.DataSource.<vendorNameDatasetName> / DataProcessing / process.sample.py file so it transforms and moves your raw data into the format and location the GetSource methods expect. Before you start editing the script, change the structure of the Lean.DataSource.<vendorNameDatasetName> / output directory to match the path structure you defined in the `GetSource` methods (for example, output / alternative / xyzairline / ticketsales).

Follow these steps to set up the downloading and processing script for your dataset:

1. In the Lean.DataSource.<vendorNameDatasetName> / DataProcessing / process.sample.py file, add some code to time how long it takes to process the entire dataset and how long it takes to update the dataset with one day's worth of data.
   You need this information for when you provide the dataset documentation . We need to know how long it takes to process your dataset so we can schedule its processing job.
2. In the processing file, load the raw data from your source.

   You can fetch data from any of the following sources:

Source	Considerations
Local Files	It can help to first copy the data into location.
Remote API	Stay within the rate limits. You can use the rate gate class.

   You should load and process the data period by period. Use the date range provided to the script to process the specific dates provided.

3. If your dataset is for universe selection data and it's at a higher frequency than hour resolution, resample your data to hourly or daily resolution.
4. If any of the following statements are true, skip the rest of the steps in this tutorial:
   - Your dataset is not related to Equities.
   - Your dataset is related to Equities and already includes the point-in-time tickers.

   If your dataset is related to Equities and your dataset doesn't account for ticker changes, the rest of the steps help you to adjust the tickers over the historical data so they are point-in-time.

5. If you don't have the US Equity Security Master dataset , contact us .
6. In the Lean.DataSource.<vendorNameDatasetName> / DataProcessing / Program.cs file, remove the statements of the `Main` method
7. In a terminal, compile the data processing project.

   ```
   $ dotnet build .\DataProcessing\DataProcessing.csproj
   ```

   This step generates a file that the `CLRImports` library uses.

8. In the Lean.DataSource.<vendorNameDatasetName> / DataProcessing / process.sample.py file, import the `CLRImports` library.

   ```
   from CLRImports import *
   ```

9. Create and initialize a map file provider.

   ```
   map_file_provider = LocalZipMapFileProvider()
   map_file_provider.Initialize(DefaultDataProvider())
   ```

10. Create a security identifier.

    ```
    sid = SecurityIdentifier.GenerateEquity(point_in_time_ticker,
        Market.USA, True, map_file_provider, csv_date)
    ```

After you finish editing the process.sample.py script, run it to populate the Lean.DataSource.<vendorNameDatasetName> / output directory.

```
$ python process.sample.py
```

Note: The pull request you make at the end must contain sample data so we can review it and run the demonstration algorithms.

Python Processor Examples

The following examples are rendering datasets with Python processing:

- Lean.DataSource.BitcoinMetadata
- Lean.DataSource.BrainSentiment
- Lean.DataSource.CryptoSlamNFTSale
- Lean.DataSource.QuiverQuantTwitterFollowers
- Lean.DataSource.Regalytics

2.1.3.2 Rendering Data with CSharp

Introduction

This page explains how to create a script to download and process your dataset with C# for QuantConnect distribution.

Using Processing Framework

During this part of the contribution process, you need to edit the Lean.DataSource.<vendorNameDatasetName> / DataProcessing / Program.cs file so it transforms and moves your raw data into the format and location the GetSource methods expect. Before you start editing the script, change the structure of the Lean.DataSource.<vendorNameDatasetName> / output directory to match the path structure you defined in the `GetSource` methods (for example, output / alternative / xyzairline / ticketsales).

Follow these steps to set up the downloading and processing script for your dataset:

1. In the Lean.DataSource.<vendorNameDatasetName> / DataProcessing / Program.cs file, add some code to time how long it takes to process the entire dataset and how long it takes to update the dataset with one day's worth of data.
   You need this information for when you provide the dataset documentation . We need to know how long it takes to process your dataset so we can schedule its processing job.
2. In the processing file, load the raw data from your source.

   You can fetch data from any of the following sources:

Source	Considerations
Local Files	It can help to first copy the data into location.
Remote API	Stay within the rate limits. You can use the rate gate class.

   You should load and process the data period by period. Use the date range provided to the script to process the specific dates provided.
3. If your dataset is for universe selection data and it's at a higher frequency than hour resolution, resample your data to hourly or daily resolution.
4. If any of the following statements are true, skip the rest of the steps in this tutorial:
   - Your dataset is not related to Equities.
   - Your dataset is related to Equities and already includes the point-in-time tickers.

   If your dataset is related to Equities and your dataset doesn't account for ticker changes, the rest of the steps help you to adjust the tickers over the historical data so they are point-in-time.
5. If you don't have the US Equity Security Master dataset , contact us .
6. In the Lean.DataSource.<vendorNameDatasetName> / DataProcessing / Program.cs file, create and initialize a map file provider.

   ```
   var mapFileProvider = new LocalZipMapFileProvider();
   var mapFileProvider.Initialize(new DefaultDataProvider());
   ```
7. Create a security identifier.

   ```
   var sid = SecurityIdentifier.GenerateEquity(pointInIimeTicker,
       Market.USA, true, mapFileProvider, csvDate)
   ```
8. In a terminal, compile the data processing project to generate the process.exe executable file.

   ```
   $ dotnet build .\DataProcessing\DataProcessing.csproj
   ```

After you finish compiling the Program.cs file, run the process.exe to populate the Lean.DataSource.<vendorNameDatasetName> / output directory.

Note: The pull request you make at the end must contain sample data so we can review it and run the demonstration algorithms.

CSharp Processor Examples

The following examples are rendering datasets with C# processing:

- Lean.DataSource.BinanceFundingRate
- Lean.DataSource.CoinGecko
- Lean.DataSource.CryptoCoarseFundamentalUniverse
- Lean.DataSource.QuiverInsiderTrading
- Lean.DataSource.VIXCentral

2.1.3.3 Rendering Data with Notebooks

Introduction

This page explains how to create a script to download and process your dataset with Jupyter Notebooks for QuantConnect distribution.

Using Processing Framework

During this part of the contribution process, you need to edit the Lean.DataSource.<vendorNameDatasetName> / DataProcessing / process.sample.ipynb file so it transforms and moves your raw data into the format and location the GetSource methods expect. Before you start editing the script, change the structure of the Lean.DataSource.<vendorNameDatasetName> / output directory to match the path structure you defined in the `GetSource` methods (for example, output / alternative / xyzairline / ticketsales).

Follow these steps to set up the downloading and processing script for your dataset:

1. In the Lean.DataSource.<vendorNameDatasetName> / DataProcessing / process.sample.ipynb file, add some code to time how long it takes to process the entire dataset and how long it takes to update the dataset with one day's worth of data.
   You need this information for when you provide the dataset documentation . We need to know how long it takes to process your dataset so we can schedule its processing job.
2. In the processing file, load the raw data from your source.

   You can fetch data from any of the following sources:

Source	Considerations
Local Files	It can help to first copy the data into location.
Remote API	Stay within the rate limits. You can use the rate gate class.

   You should load and process the data period by period. Use the date range provided to the script to process the specific dates provided.

3. If your dataset is for universe selection data and it's at a higher frequency than hour resolution, resample your data to hourly or daily resolution.
4. If any of the following statements are true, skip the rest of the steps in this tutorial:
   - Your dataset is not related to Equities.
   - Your dataset is related to Equities and already includes the point-in-time tickers.

   If your dataset is related to Equities and your dataset doesn't account for ticker changes, the rest of the steps help you to adjust the tickers over the historical data so they are point-in-time.

5. If you don't have the US Equity Security Master dataset , contact us .
6. In the Lean.DataSource.<vendorNameDatasetName> / DataProcessing / Program.cs file, remove the statements of the `Main` method
7. In a terminal, compile the data processing project.

   ```
   $ dotnet build .\DataProcessing\DataProcessing.csproj
   ```

   This step generates a file that the `CLRImports` library uses.

8. In the Lean.DataSource.<vendorNameDatasetName> / DataProcessing / process.sample.ipynb file, import the `CLRImports` library.

   ```
   from CLRImports import *
   ```

9. Create and initialize a map file provider.

   ```
   map_file_provider = LocalZipMapFileProvider()
   map_file_provider.Initialize(DefaultDataProvider())
   ```

10. Create a security identifier.

    ```
    sid = SecurityIdentifier.GenerateEquity(point_in_time_ticker,
        Market.USA, True, map_file_provider, csv_date)
    ```

After you finish editing the process.sample.ipynb script, run its cells to populate the Lean.DataSource.<vendorNameDatasetName> / output directory.

Note: The pull request you make at the end must contain sample data so we can review it and run the demonstration algorithms.

Notebook Processor Examples

The following examples are rendering datasets with Jupyter Notebook processing:

- Lean.DataSource.KavoutCompositeFactorBundle
- Lean.DataSource.USEnergy
- Lean.DataSource.FRED

2.1.4 Testing Data Models

Introduction

The implementation of your Data Source must be thoroughly tested to be listed on the Dataset Market .

Run Demonstration Algorithms

Follow these steps to test if your demonstration algorithm will run in production with the processed data:

1. Open the Lean.DataSource.<vendorNameDatasetName> / QuantConnect.DataSource.csproj file in Visual Studio.
2. In the top menu bar of Visual Studio, click Build > Build Solution .

   The Output panel displays the build status of the project.

3. Close Visual Studio.
4. Fork the LEAN repository .
5. Clone the Lean repository.

   ```
   $ git clone https://github.com/<username>/Lean.git
   ```

6. Copy the contents of the Lean.DataSource.<vendorNameDatasetName> / output directory and paste them into the Lean / Data directory.
7. Open the Lean / QuantConnect.Lean.sln file in Visual Studio.
8. In the Solution Explorer panel of Visual Studio, right-click QuantConnect.Algorithm.CSharp and then click Add > Existing Itemâ€¦ .
9. In the Add Existing Item window, click the Lean.DataSource.<vendorNameDatasetName> / <vendorNameDatasetName>Algorithm.cs file and then click Add .
10. In the Solution Explorer panel, right-click QuantConnect.Algorithm.CSharp and then click Add > Project Reference... .
11. In the Reference Manager window, click Browseâ€¦ .
12. In the Select the files to referenceâ€¦ window, click the Lean.DataSource.<vendorNameDatasetName> / bin / Debug / net6.0 / QuantConnect.DataSource.<vendorNameDatasetName>.dll file and then click Add .

    The Reference Manager window displays the QuantConnect.DataSource.<vendorNameDatasetName>.dll file with the check box beside it enabled.

13. Click OK .

    The Solution Explorer panel adds the QuantConnect.DataSource.<vendorNameDatasetName>.dll file under QuantConnect.Algorithm.CSharp > Dependencies > Assemblies .

14. Copy the Lean.DataSource.<vendorNameDatasetName> / <vendorNameDatasetName>Algorithm.cs file and paste it in Lean / Algorithm.CSharp directory.
15. In the Lean / Algorithm.CSharp / <vendorNameDatasetName>Algorithm.cs file, write an algorithm that uses your new dataset.
16. In the Solution Explorer panel, click QuantConnect.Lean.Launcher > config.json .
17. In the config.json file, set the following keys:

    ```
    "algorithm-type-name": "<vendorNameDatasetName>Algorithmâ€,
    "algorithm-language": "CSharp",
    "algorithm-location": "QuantConnect.Algorithm.CSharp.dll",
    ```

18. Press F5 to backtest your demonstration algorithm.
19. Copy the Lean.DataSource.<vendorNameDatasetName> / <vendorNameDatasetName>Algorithm.py file and paste it in Lean / Algorithm.Python directory.
20. In the Lean / Algorithm.Python / <vendorNameDatasetName>Algorithm.py file, write an algorithm that uses your new dataset.
21. In the Solution Explorer panel, click QuantConnect.Lean.Launcher > config.json .
22. In the config.json file, set the following keys:

    ```
    "algorithm-type-name": "<vendorNameDatasetName>Algorithmâ€,
    "algorithm-language": "Python",
    "algorithm-location": "../../../Algorithm.Python/<vendorNameDatasetName>Algorithm.py",
    ```

23. Press F5 to backtest your demonstration algorithm.

    Important: Your backtests must run without error. If your backtests produce errors, correct them and then run the backtest again.

24. Copy the Lean / Algorithm.CSharp / <vendorNameDatasetName>Algorithm.cs file to Lean.DataSource.<vendorNameDatasetName> / <vendorNameDatasetName>Algorithm.cs .
25. Copy the Lean / Algorithm.Python / <vendorNameDatasetName>Algorithm.py file to Lean.DataSource.<vendorNameDatasetName> / <vendorNameDatasetName>Algorithm.py .

Run Unit Tests

You must run your demonstration algorithms without error before you set up unit tests.

In the Lean.DataSource.<vendorNameDatasetName> / <vendorNameDatasetName>Tests.cs file, define the `CreateNewInstance` method to return an instance of your `DataSource` class and then execute the following commands to run the unit tests:

```
$ dotnet build tests/Tests.csproj
$ dotnet test tests/bin/Debug/net6.0/Tests.dll
```

2.1.5 Data Documentation

Introduction

This page explains how to provide documentation for your dataset so QuantConnect members can use it in their trading algorithms.

Required Key Properties

You need to process the entire dataset to collect the following information:

Property	Description
Start Date	Date and time of the first data point
Asset Coverage	Number of assets covered by the dataset
Data density	Dense for tick data. Regular or Sparse according to the frequency.
Resolution	Options: Tick, Second, Minute, Hourly, & Daily.
Timezone	Data timezone. This is a property of the data source.
Data process time	Time and days of the week to process the data.
Data process duration	Time to process the entire the dataset.
Update process duration	Time to update the dataset.

Provide Documentation

To provide documentation for your dataset, in the Lean.DataSource.<vendorNameDatasetName> / listing-about.md and Lean.DataSource.<vendorNameDatasetName> / listing-documentation.md files, fill in the missing content.

Next Steps

After we review and accept your dataset contribution, we will create a page in our Dataset Market . At that point, you will be able to write algorithms in QuantConnect Cloud using your dataset and you can contribute an example algorithm for the dataset listing.

2.2 Brokerages

Creating a fully supported brokerage is a challenging endeavor. LEAN requires a number of individual pieces which work together to form a complete brokerage implementation. This guide aims to describe in as much detail as possible what you need to do for each module.

The end goal is to submit a pull request that passes all tests. Partially-completed brokerage implementations are acceptable if they are merged to a branch. It's easy to fall behind master, so be sure to keep your branch updated with the master branch. Before you start, read LEAN's coding style guidelines to comply with the code commenting and design standards.

The root of the brokerage system is the algorithm job packets, which hold configuration information about how to run LEAN. The program logic is a little convoluted. It moves from *config.json* > *create job packet* > *create brokerage factory matching name* > *set job packet brokerage data* > *factory creates brokerage instance* . As a result, we'll start creating a brokerage at the root, the configuration and brokerage factory.

Setting Up Your Environment

Set up your local brokerage repository.

Laying the Foundation

(`IBrokerageFactory`) Stub out the implementation and initialize a brokerage instance.

Creating the Brokerage

(`IBrokerage`) Instal key brokerage application logic, where possible using a brokerage SDK.

Translating Symbol Conventions

(`ISymbolMapper`) Translate brokerage specific tickers to LEAN format for a uniform algorithm design experience.

Describing Brokerage Limitations

(`IBrokerageModel`) Describe brokerage support of orders and set transaction models.

Enabling Live Data Streaming

(`IDataQueueHandler`) Set up a live streaming data service from a brokerage-supplied source.

Enabling Historical Data

(`IHistoryProvider`) Tap into the brokerage historical data API to serve history for live algorithms.

Downloading Data

(`IDataDownloader`) Save data from the brokerage to disk in LEAN format.

Modeling Fee Structures

(`IFeeModel`) Enable accurate backtesting with specific fee structures of the brokerage.

Updating the Algorithm API

(`ISecurityTransactionModel`) Combine the various models together to form a brokerage set.

See Also

Dataset Market
Purchasing Datasets

2.2.1 Setting Up Your Environment

Introduction

This page explains how to set up your coding environment to create, develop, and test your brokerage before you contribute it to LEAN.

Prerequisites

Working knowledge of C#. You also need to install .NET 6.0 .

Set Up Environment

Follow these steps to set up your environment:

1. Fork Lean and then clone your forked repository to your local machine.
2. Open the Lean.Brokerages.Template repository and click Use this template .
3. On the Create a new repository from Lean.Brokerages.Template page, set the repository name to Lean.Brokerages.<brokerageName> (for example, Lean.Brokerages.XYZ).
4. Click Create repository from template .
5. Clone the Lean.Brokerages.<brokerageName> repository.

   ```
   $ git clone https://github.com/username/Lean.Brokerages.<brokerageName>.git
   ```

6. If you're on a Linux terminal, in your Lean.Brokerages.<brokerageName> directory, change the access permissions of the bash script.

   ```
   $ chmod +x ./renameBrokerage
   ```

7. In your Lean.Brokerages.<brokerageName> directory, run the renameBrokerage.sh bash script.

   ```
   $ renameBrokerage.sh
   ```

   The bash script replaces some placeholder text in the Lean.Brokerages.<brokerageName> directory and renames some files according to your brokerage name.

2.2.2 Laying the Foundation

IBrokerageFactory	
Primary Role	Create and initialize a brokerage instance.
Interface	IBrokerageFactory.cs
Example	BitfinexBrokerageFactory.cs
Target Location	Lean.Brokerages.<brokerageName> / QuantConnect.<brokerageName>Brokerage /

Introduction

The IBrokerageFactory creates brokerage instances and configures LEAN with a Job Packet . To create the right `BrokerageFactory` type, LEAN uses the brokerage name in the job packet. To set the brokerage name, LEAN uses the `live-mode-brokerage` value in the configuration file .

Prerequisites

You need to set up your environment before you can lay the foundation for a new brokerage.

Lay the Foundation

Follow these steps to stub out the implementation and initialize a brokerage instance:

1. In the Lean / Launcher / config.json file, add a few key-value pairs with your brokerage configuration information.

   For example, `oanda-access-token` and `oanda-account-id` keys. These key-value pairs will be used for most local debugging and testing as the default. LEAN automatically copies these pairs to the BrokerageData member of the job packet as a dictionary of `<string,string>` pairs.

2. In the Lean.Brokerages.<brokerageName> / QuantConnect.<brokerageName>Brokerage / <brokerageName>Factory.cs file, update the `BrokerageData` member so it uses the `Config` class to load all the required configuration settings from the Lean / Launcher / config.json file.

   For instance, `Config.Get("oanda-access-token")` returns the `"oanda-access-token"` value from the configuration file. For a full example, see the BrokerageData member in the `BitfinexBrokerageFactory` .

   In the `IBrokerageFactory` examples, you'll see code like `Composer.Instance.AddPart<IDataQueueHandler>(dataQueueHandler)` , which adds parts to the Composer . The Composer is a system in LEAN for dynamically loading types. In this case, it's adding an instance of the `DataQueueHandler` for the brokerage to the composer. You can think of the Composer as a library and adding parts is like adding books to its collection.

3. In the Lean / Common / Brokerages folder, create a <brokerageName>.cs file with a stub implementation that inherits from the DefaultBrokerageModel .

   Brokerage models tell LEAN what order types a brokerage supports, whether we're allowed to update an order, and what reality models to use. Use the following stub implementation for now:

   ```
   namespace QuantConnect.Brokerages
   {
       public class BrokerageNameBrokerageModel : DefaultBrokerageModel
       {

       }
   }
   ```

   where `BrokerageName` is the name of your brokerage. For example, if the brokerage name is XYZ, then `BrokerageNameBrokerageModel` should be `XYZBrokerageModel` . You'll extend this implementation later.

4. In the Lean.Brokerages.<BrokerageName> / QuantConnect.<brokerageName>Brokerage / <brokerageName>BrokerageFactory.cs file, define `GetBrokerageModel` to return an instance of your new brokerage model.

   ```
   public override IBrokerageModel GetBrokerageModel(IOrderProvider orderProvider)
   {
       return new BrokerageNameBrokerageModel();
   }
   ```

5. If your brokerage uses websockets to send data, in the Lean.Brokerages.<brokerageName> / QuantConnect.<brokerageName> / <brokerageName>Brokerage.cs file, replace the `Brokerage` base class for `BaseWebsocketsBrokerage` .
6. In the Lean.Brokerages.<brokerageName> / QuantConnect.<brokerageName>Brokerage / <brokerageName>Brokerage.cs file, update the constructor to save required authentication data to private variables.
7. In the Lean.Brokerages.<brokerageName> / QuantConnect.<brokerageName>Brokerage / <brokerageName>BrokerageFactory.cs file, define the `CreateBrokerage` method to create and return an instance of your new brokerage model without connecting to the brokerage.

   The Brokerage Factory uses a job packet to create an initialized brokerage instance in the `CreateBrokerage` method. Assume the `job` argument has the best source of data, not the `BrokerageData` property. The `BrokerageData` property in the factory are the starting default values from the configuration file, which can be overridden by a runtime job.

8. In the Lean / Launcher / config.json file, add a `live-<brokerageName>` key.

   These `live-<brokerageName>` keys group configuration flags together and override the root configuration values. Use the following key-value pair as a starting point:

   ```
   // defines the 'live-brokerage-name' environment
   "live-brokerage-name": {
     "live-mode": true,

     "live-mode-brokerage": "BrokerageName",

     "setup-handler": "QuantConnect.Lean.Engine.Setup.BrokerageSetupHandler",
     "result-handler": "QuantConnect.Lean.Engine.Results.LiveTradingResultHandler",
     "data-feed-handler": "QuantConnect.Lean.Engine.DataFeeds.LiveTradingDataFeed",
     "data-queue-handler": [ "QuantConnect.Lean.Engine.DataFeeds.Queues.LiveDataQueue" ],
     "real-time-handler": "QuantConnect.Lean.Engine.RealTime.LiveTradingRealTimeHandler",
     "transaction-handler": "QuantConnect.Lean.Engine.TransactionHandlers.BacktestingTransactionHandler"
   },
   ```

   where `brokerage-name` and `"BrokerageName"` are placeholders for your brokerage name.

9. In the Lean / Launcher / config.json file, set the `environment` value to the your new brokerage environment.

   For example, `"live-brokerage-name"` .

10. Build the solution.

    Running the solution won't work, but the stub implementation should still build.

2.2.3 Creating the Brokerage

	IBrokerage
Primary Role	Brokerage connection, orders, and fill events.
Interface	IBrokerage.cs
Example	BitfinexBrokerage.cs
Target Location	Lean.Brokerages.<brokerageName> / QuantConnect.<brokerageName>Brokerage /

Introduction

The IBrokerage holds the bulk of the core logic responsible for running the brokerage implementation. Many smaller models described later internally use the brokerage implementation, so its best to now start implementating the `IBrokerage` . Brokerage classes can get quite large, so use a `partial` class modifier to break up the files in appropriate categories.

Prerequisites

You need to lay the foundation before you can create a new brokerage.

Brokerage Roles

The brokerage has many the following important roles vital for the stability of a running algorithm:

1. Maintain Connection - Connect and maintain connection while algorithm running.
2. Setup State - Initialize the algorithm portfolio, open orders and cashbook.
3. Order Operations - Create, update and cancel orders.
4. Order Events - Receive order fills and apply them to portfolio.
5. Account Events - Track non-order events (cash deposits/removals).
6. Brokerage Events - Interpret brokerage messages and act when required.
7. Serve History Requests - Provide historical data on request.

Brokerages often have their own ticker styles, order class names, and event names. Many of the methods in the brokerage implementation may simply be converting from the brokerage object format into LEAN format. You should plan accordingly to write neat code.

The brokerage must implement the following interfaces:

```
class MyBrokerage : Brokerage, IDataQueueHandler, IDataQueueUniverseProvider { ... }
```

Implementation Style

This guide focuses on implementing the brokerage step-by-step in LEAN because it's a more natural workflow for most people. You can also follow a more test-driven development process by following the test harness. To do this, create a new test class that extends from the base class in Lean / Tests / Brokerages / BrokerageTests.cs . This test-framework tests all the methods for an `IBrokerage` implementation.

Connection Requirements

LEAN is best used with streaming or socket-based brokerage connections. Streaming brokerage implementations allow for the easiest translation of broker events into LEAN events. Without streaming order events, you will need to poll for to check for fills. In our experience, this is fraught with additional risks and challenges.

SDK Libraries

Most brokerages provide a wrapper for their API. If it has a permissive license and it's compatible with .NET 6, you should utilize it. Although it is technically possible to embed an external github repository, we've elected to not do this to make LEAN easier to install (submodules can be tricky for beginners). Instead, copy the library into its own subfolder of the brokerage implementation. For example, Lean.Brokerages.<brokerageName> / QuantConnect.<brokerageName>Brokerage / BrokerLib / * . After you add a library, build the project again to make sure the library successfully compiles.

LEAN Open-Source. If you copy and paste code from an external source, leave the comments and headers intact. If they don't have a comment header, add one to each file, referencing the source. Let's keep the attributions in place.

Define the Brokerage Class

The following sections describe components of the brokerage implementation in the Lean.Brokerages.<brokerageName> / QuantConnect.<brokerageName>Brokerage / <brokerageName>Brokerage.cs file.

Base Class

Using a base class is optional but allows you to reuse event methods we have provided. The `Brokerage` object implements these event handlers and marks the remaining items as `abstract` .

LEAN provides an optional base class `BaseWebsocketsBrokerage` which seeks to connect and maintain a socket connection and pass messages to an event handler. As each socket connection is different, carefully consider before using this class. It might be easier and more maintainable to simply maintain your own socket connection.

Brush up on the `partial` class keyword. It will help you break-up your class later.

Class Constructor

Once the scaffolding brokerage methods are in place (overrides of the abstract base classes), you can focus on the class constructor. If you are using a brokerage SDK, create a new instance of their library and store it to a class variable for later use. You should define the constructor so that it accepts all the arguments you pass it during the `CreateBrokerage` method you implemented in the Lean.Brokerages.<brokerageName> / QuantConnect.<brokerageName>Brokerage / <brokerageName>BrokerageFactory.cs file.

The following table provides some example implementations of the brokerage class constructor:

Brokerage	Description
Interactive Brokers	Launches an external process to create the brokerage.

OANDA	Creates an SDK instance and assigns internal event handlers.
Coinbase	Offloads constructor work to `BrokerageFactory` and uses the `BaseWebsocketBrokerage` base class.

`string Name`

The `Name` property is a human-readable brokerage name for debugging and logging. For US Equity-regulated brokerages, convention states this name generally ends in the word "Brokerage".

`void Connect()`

The `Connect` method triggers logic for establishing a link to your brokerage. Normally, we don't do this in the constructor because it makes algorithms and brokerages die in the `BrokerageFactory` process. For most brokerages, to establish a connection with the brokerage, call the connect method on your SDK library.

The following table provides some example implementations of the `Connect` method:

Brokerage	Description
Interactive Brokers	Connects to an external process with the brokerage SDK.
OANDA	Simple example that calls the brokerage SDK.
Coinbase	Establishes the WebSocket connection and monitoring in a thread.

If a soft failure occurs like a lost internet connection or a server 502 error, create a new `BrokerageMessageEvent` so you allow the algorithm to handle the brokerage messages . For example, Interactive Brokers resets socket connections at different times globally, so users in other parts of the world can get disconnected at strange times of the day. Knowing this, they may elect to have their algorithm ignore specific disconnection attempts.

If a hard failure occurs like an incorrect password or an unsupported API method, throw a real exception with details of the error.

`void Disconnect()`

The `Disconnect` method is called at the end of the algorithm before LEAN shuts down.

`bool IsConnected`

The `IsConnected` property is a boolean that indicates the state of the brokerage connection. Depending on your connection style, this may be automatically handled for you and simply require passing back the value from your SDK. Alternatively, you may need to maintain your own connection state flag in your brokerage class.

`bool PlaceOrder(Order order)`

The `PlaceOrder` method should send a new LEAN order to the brokerage and report back the success or failure. The `PlaceOrder` method accepts a generic `Order` object, which is the base class for all order types. The first step of placing an order is often to convert it from LEAN format into the format that the brokerage SDK requires. Your brokerage implementation should aim to support as many LEAN order types as possible. There may be other order types in the brokerage, but implementing them is considered out of scope of a rev-0 brokerage implementation.

Converting order types is an error-prone process and you should carefully review each order after you've ported it. Some brokerages have many properties on their orders, so check each required property for each order. To simplify the process, define an internal `BrokerOrder ConvertOrder(Order order)` method to convert orders between LEAN format and your brokerage format. Part of the order conversion might be converting the brokerage ticker (for example, LEAN name "EURUSD" vs OANDA name "EUR/USD"). This is done with a `BrokerageSymbolMapper` class. You can add this functionality later. For now, pass a request for the brokerage ticker to the stub implementation.

Once the order type is converted, use the `IsConnected` property to check if you're connected before placing the order. If you're not connected, throw an exception to halt the algorithm. Otherwise, send the order to your brokerage submit API. Oftentimes, you receive an immediate reply indicating the order was successfully placed. The `PlaceOrder` method should return true when the order is accepted by the brokerage. If the order is invalid, immediately rejected, or there is an internet outage, the method should return false.

`bool UpdateOrder(Order order)`

The `UpdateOrder` method transmits an update request to the API and returns true if it was successfully processed. Updating an order is one of the most tricky parts of brokerage implementations. You can easily run into synchronization issues.

The following table provides some example implementations of the `UpdateOrder` method:

Brokerage	Description
Interactive Brokers	Updates multiple asset classes with an external application.
OANDA	Simple example that calls the brokerage SDK.
Coinbase	Throws an exception because order updates are not supported.

`bool CancelOrder(Order order)`

`bool UpdateOrder(Order order)`

`List<Order> GetOpenOrders()`

`List<Holding> GetAccountHoldings()`

`List<Cash> GetCashBalance()`

`bool AccountInstantlyUpdated`

`IEnumerable<BaseData> GetHistory(HistoryRequest request)`

`bool AccountInstantlyUpdated`

2.2.4 Translating Symbol Conventions

Introduction

This brokerage development guide is still under construction.

2.2.5 Describing Brokerage Limitations

Introduction

This brokerage development guide is still under construction.

2.2.6 Enabling Live Data Streaming

Introduction

This brokerage development guide is still under construction.

2.2.7 Enabling Historical Data

Introduction

This brokerage development guide is still under construction.

2.2.8 Downloading Data

Introduction

This brokerage development guide is still under construction.

2.2.9 Modeling Fee Structures

Introduction

This brokerage development guide is still under construction.

2.2.10 Updating the Algorithm API

Introduction

This brokerage development guide is still under construction.

3 Statistics

3.1 Capacity

Introduction

Capacity is a measure of how much capital a strategy can trade before the performance of the strategy degrades from market impact. The capacity calculation is done on a rolling basis with one snapshot taken at the end of each week. This page outlines how LEAN performs the entire calculation.

Security Capacity

The first step to determine the capacity of the strategy is to compute the capacity of each security the strategy trades.

Market Capacity Dollar Volume

Following each order fill, LEAN monitors and records the dollar-volume for a series of bars. To get an estimate of the available capacity, we combine many second and minute trade bars together. For hourly or daily data resolutions, we only use one bar.

```
_marketCapacityDollarVolume += bar.Close * _fastTradingVolumeDiscountFactor * bar.Volume * conversionRate * Security.SymbolProperties.ContractMultiplier;
```

Crypto Volume

Crypto trade volume is light, but there is significant capacity even at the very top of the order book. The estimated volume of Crypto is based on the average size on the bid and ask.

Forex and CFD Volume

Forex and CFD assets do not have a trade volume or quote size information so they were approximated as deeply liquid assets with approximately $25,000,000 depth per minute.

Volume Accumulation Period

The number of bars we use to calculate the market volume estimate depends on the asset liquidity. The following table shows the formulas LEAN uses to determine how long of a period the market capacity dollar volume is accumulated for after each order fill, as a function of the security resolution. The *AvgDollarVolume* in the table represents the average dollar volume per minute for the security you're trading. Notice that for the edge case where the average dollar volume is zero, the calculations use 10 minutes of data.

Resolution	Timeout Period
Second	$k = \begin{cases} \frac{100,000}{AvgDollarVolume}, & \text{if } AvgDollarVolume \neq 0 \\ 10, & \text{otherwise} \end{cases}$ $\min(120,\ \max(5, k)) \in [5, 120]$ minutes
Minute	$k = \begin{cases} \frac{6,000,000}{AvgDollarVolume}, & \text{if } AvgDollarVolume \neq 0 \\ 10, & \text{otherwise} \end{cases}$ $\min(120,\ \max(1, k)) \in [1, 120]$ minutes
Hour	1 hour
Daily	1 day

Only a fraction of the market capacity dollar volume is available to be taken by a strategyâ€™s orders because there are other market participants. The data resolution of the security determines how much of the market capacity dollar volume is available for the strategy to consume. The following table shows what percentage of the market capacity dollar volume is available for each of the data resolutions:

Resolution	Available Portion of Market Capacity Dollar Volume (%)
Daily	2
Hour	5
Minute	20
Second	50
Tick	50

Fast Trading Volume Discount Factor

To accommodate high-frequency trading strategies, the `_fastTradingVolumeDiscountFactor` variable scales down the market capacity dollar volume of the security proportional to the number of trades that it places per day for the security. The more frequently the strategy trades a security, the lower the capacity of the security goes since it becomes harder to get into a larger position without incurring significant market impact. The formula that LEAN uses to discount the capacity of the securities that the algorithm trades intraday is

$$d_i = \begin{cases} 1, & \text{if } i = 1 \\ \min(1,\ \max(0.2, d_{i-1} * \frac{m}{390})), & \text{if } i > 1 \end{cases}$$

where $d_i \in [0.2, 1]$
is the fast trading volume discount factor after order i
and m

is the number of minutes since order $i - 1$
was filled. We divide m
by 390 because there are $390 = 6.5 * 60$
minutes of trading in a regular Equity trading day.

Sale Volume

In addition to the market capacity dollar volume, for each security the strategy trades, LEAN also accumulates the weekly sale volume of the order fills. The sale volume scales down the weekly snapshot capacity.

```
SaleVolume += orderEvent.FillPrice * orderEvent.AbsoluteFillQuantity * Security.SymbolProperties.ContractMultiplier;
```

Portfolio Capacity

Now that we have the values to calculate the capacity of each security, we can compute the capacity of the portfolio.

Snapshot Capacity

To calculate the strategy capactiy, weekly snapshots are taken. When itâ€™s time to take a snapshot, the capacity of the strategy for the current snapshot is calculated by first selecting the security with the least market capacity dollar volume available. The fraction of trading volume that was available for this security is scaled down by the number of orders that were filled for the security during the week. The result is scaled down further by the largest value between the weight of the securityâ€™s sale volume in the portfolio sale volume and the weight of the securityâ€™s holding value in the total portfolio value. The result of this final scaling is the strategyâ€™s capacity in the current snapshot.

$$Snapshot\ Capacity = \frac{\frac{Market\ Capacity\ Dollar\ Volume}{Number\ Of\ Trades}}{\max\left(\frac{Sale\ Volume}{Portfolio\ Sale\ Volume}, \frac{Buying\ Power\ Used}{Total\ Portfolio\ Value}\right)}$$

When any of the denominators are 0 in the preceding formula, the quotient that the denominator is part of defaults to a value of 0. After the snapshot is taken, the sale volume and market capacity dollar volume of each security is reset to 0.

Strategy Capacity

Instead of using the strategyâ€™s capacity at the current snapshot as the final strategy capacity value, the strategy capacity is smoothed across the weekly snapshots. First, the capacity estimate of the current snapshot is calculated, then the final strategy capacity value is set using the following exponentially-weighted model:

$$Strategy\ Capacity = \begin{cases} S_i, & \text{if } i = 1 \\ 0.66 * S_{i-1} + 0.33 * S_i, & \text{if } i > 1 \end{cases}$$

where S_i
is the snapshot capacity of week i
.

Summary

Strategies that have a larger capacity are able to trade more capital without suffering from significant market impact. In general, a strategy that trades a large weight of the portfolio in liquid securities with high volume will have a large capacity. To avoid reducing the strategy capacity too much, only trade a small portion of your portfolio in illiquid assets with low volume.

4 Class Reference

{ "type": "link", "heading": "Class Reference", "subHeading": "", "content": "", "alsoLinks": [], "href": "https://www.lean.io/docs/v2/lean-engine/class-reference/" }