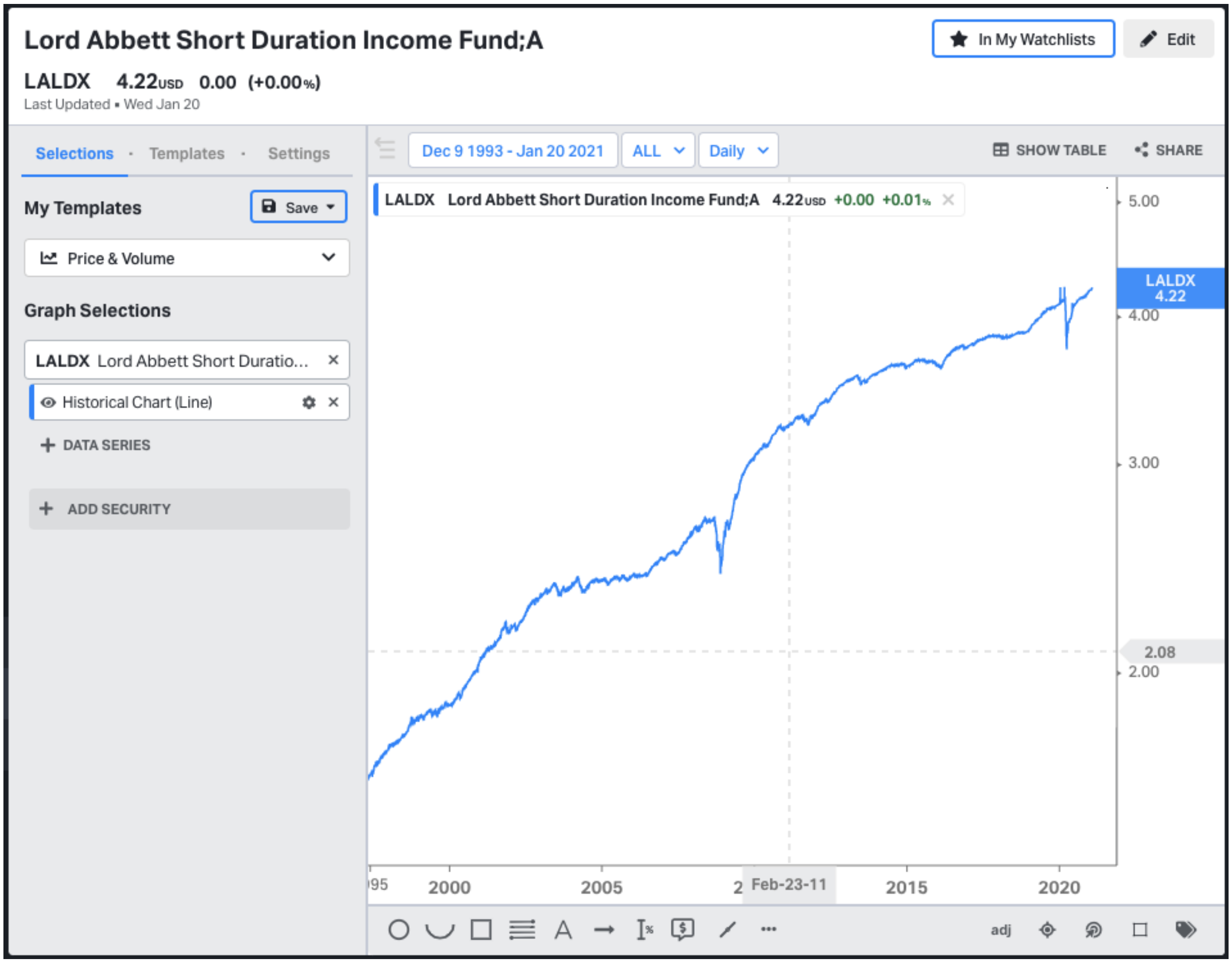
Data Engineer Take Home Test

Background:

- At Koyfin, we process data from different sources and provide a customized way to explore that via our front end application.
- In this exercise, you will solve a real world ETL problem with Apache Spark.
- In this example we have gathered data from the past 12 months and provided you with json, xml format for a single Mutual Fund.

The Feature:

- In the Koyfin app, for each Mutual Fund, we want to chart a time series of its adjusted and unadjusted historical prices. The adjusted price accounts for cash distributions paid by the fund, and the unadjusted price does not, it represents the market price per share.
    - The adjusted chart below shows how the fund performed, for an example investor, since 1993.



Lord Abbett Short Duration Income Fund;A

LALDX 4.22USD 0.00 (+0.00%)

Last Updated • Wed Jan 20

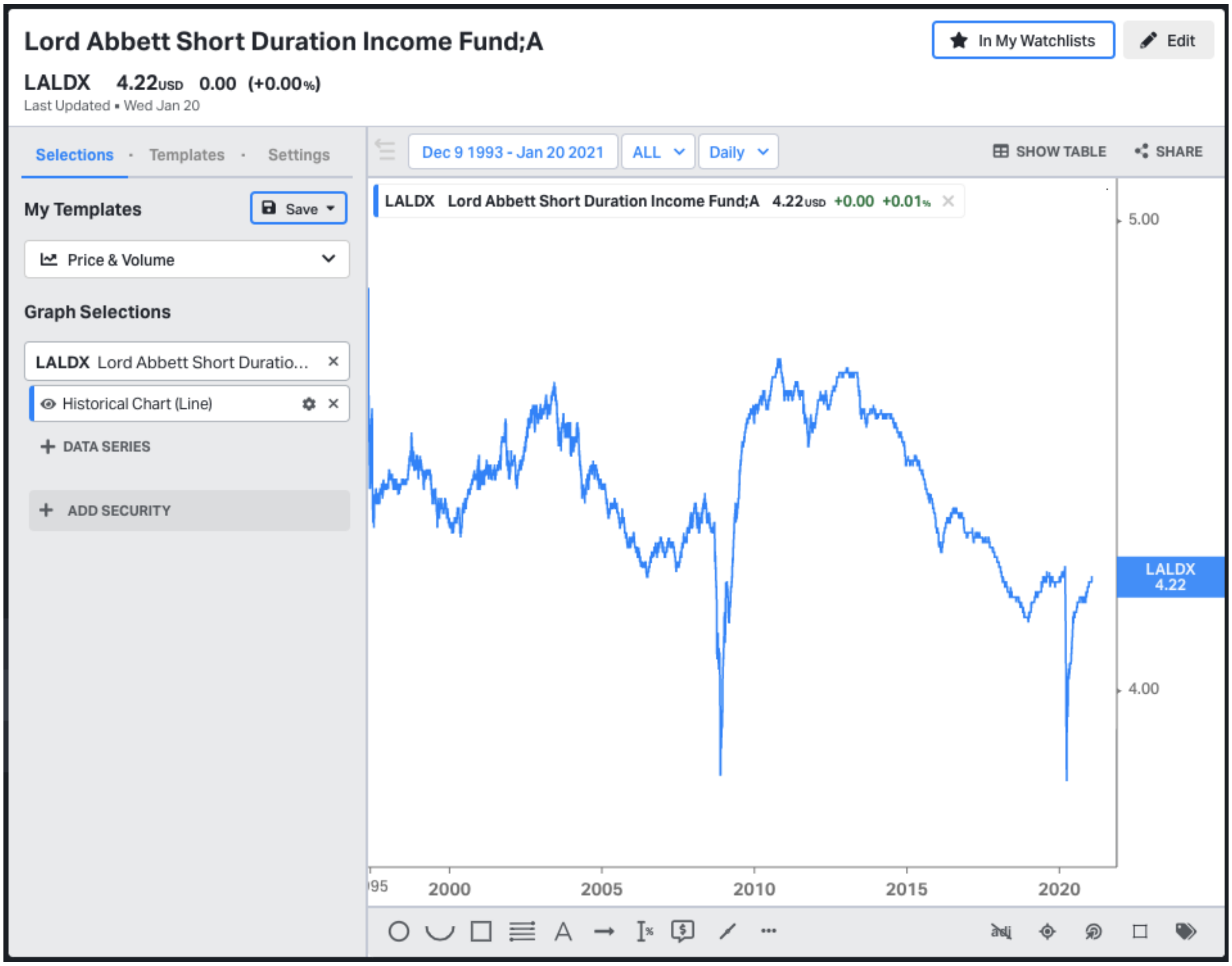
My Templates: Price & Volume

Graph Selections: LALDX Lord Abbett Short Duratio... — Historical Chart (Line)

Dec 9 1993 - Jan 20 2021 · ALL · Daily

LALDX Lord Abbett Short Duration Income Fund;A 4.22USD +0.00 +0.01%

- The unadjusted chart below shows the market price remained relatively constant since 1993, dipping after each cash distribution.



Lord Abbett Short Duration Income Fund;A

LALDX 4.22USD 0.00 (+0.00%)

Last Updated • Wed Jan 20

Dec 9 1993 - Jan 20 2021 ALL Daily

LALDX Lord Abbett Short Duration Income Fund;A 4.22USD +0.00 +0.01%

Input:

- We get the unadjusted market price series, and a cumulative adjustment factor series from the data vendor (Acme Data Inc.), and we have to derive the adjusted price in our ETL pipeline. Each day, the vendor drops a JSON file into an FTP directory, with new data for us to append to the series. We need to load any new files as soon as they're available on the FTP server. For this example we are just going to concentrate on the existing file system and not worry about copying any data from ftp server.

- The sample JSON files contain real data, for one mutual fund. .

Field	**Type**	**Description**
`date`	string (YYYY-MM-DD)	We receive one json per day and the date field is populated with it.
`assetId`	integer	This **assetId** is of the source from where this data is consumed, so we do store this for data lineage but it is not used to expose via our REST endpoint
`nav`	string	The **nav** field represents the Net Asset Value of the fund, aka the unadjusted market price when the market closed, on that date
`adjustmentFactor`		The **adjustmentFactor** field, representing the latest cumulative adjustment factor used to compute the adjusted price, taking effect on that date.

Here’s an example file with both nav and adjustmentFactor, for one mutual fund.

```
[
  {
    "date": "2019-12-27",
    "assetId": 4000123,
    "nav": "18.46",
    "adjustmentFactor": "10.0449181987999"
  }
]
```

We store both the unadjusted and adjusted prices in a database so that it’s easy to serve API calls from our client, for charting these series. In our database, we want to index this data using our own, internal unique ID for this trading item, the Koyfin ID (**KID**). Assume these IDs are already available, in a flat json file, like this:

Field	**Type**	**Description**
KID	string	Koyfin ID (**KID**)
ticker	string	Publicly traded ticker value
qualifiedTicker	string	At Koyfin we have international data, so we add additional information to make it unique per country
ISIN	string	International Securities Identification number
acmeAssetID	string	Asset id mentioned in the input above.

```
[
  {
    "KID": "mf-5s7ebl",
    "ticker": "FSDAX",
    "qualifiedTicker": "FSDAX:US",
    "ISIN": "UA123456789",
    "acmeAssetID": 4000123
  },
  {
    "KID": "mf-xyz123",
    "ticker": "LALDX",
    "qualifiedTicker": "LALDX:US",
    "ISIN": "UA123456788",
    "acmeAssetID": 4000567
  }
]
```

Formula:

Here’s the formula for the adjusted series:

```
adjPrice(t) = marketPrice(t) * adjFactor(t) / adjFactor(max(t))
```

max(t) = the timestamp at the head of the series

Note that when a new Adjustment factor is added, the historical adjusted prices will change. This table shows the progression of `adjFactor(max(t))` as new data is imported.

Insert Time	Date	Market Price	Latest Adj Factor	Adj Price @Time 2	Adj Price @Time 6	Max Adj Factor @Time 1	Max Adj Factor @Time 2	Max Adj Factor @Time 3	Max Adj Factor @Time 4	Max Adj Factor @Time 5	Max Adj Factor @Time 6
6	2021-01-06	9.3	10		9.3						10
5	2021-01-05	2.3	10		2.3					10	10
4	2021-01-04	8.4	10		8.4				10	10	10
3	2021-01-03	14.3	9		12.87			9	10	10	10
2	2021-01-02	12.3	9	12.3	11.07		9	9	10	10	10
1	2021-01-01	10	1	1.11111	1	1	9	9	10	10	10

Resources:

This exercise requires you to install docker on your laptop. Once docker is installed read through README attached.

Once setup is complete you should have a Zeppelin environment running to execute your program but it is not mandated to use. Please feel free to use and implement this exercise in any other format but will be preferred if you use spark framework to complete it.

Apache Spark - provided in the Docker container
Elasticsearch - provided in the Docker container
Redis - provided in the Docker container
MongoDB - provided in the Docker container
input/koyfinSymbologyService.json - flat file mapping different types of IDs for the same mutual fund
input/json - archive of daily feed files in json format
input/xml.zip - archive of daily feed files in xml format
output/expected_adjusted_values.csv - spreadsheet of actual adjusted values for mutual fund 4000123, for reference

Goal:

- Choose a database and describe the schema of your data model
- Implement using apache spark framework to load this data into your new database
- Defend your design and discuss the tradeoffs
- Describe in plain english how you would test this system
- Describe in plain english how you would monitor this system
    - To detect missing data points
    - To alert engineers of corrupt input data
    - To alert engineers of infrastructure outage
- Bonus: make a REST API to serve up the unadjusted and adjusted series to the client

Evaluation :

Your program will be evaluated based on the following criteria, in decreasing order of importance:

1. *Correctness* – Does the assignment produced by your program satisfy all the required covenants? Does the assignment stay within each facility's capacity?

2. *Clarity* – Is your code well-organized and easy to read?

3. Extensibility – Is your solution architected in a manner that makes it easy to collaborate with multiple engineers, and allow them to add and modify features in a consistent, testable way?