



ArbitrageGainer - project specification...
PDF

This is my project spec and the following is my workflow on obtaining cross-traded cryptocurrencies. Provide me with some feedback.

Bounded Context "Data Computation System"

workflow „List of currency at an exchange retrieved“ =

```
input = CurrencyListRetrievallInitiated event AND exchangeToRetrieve  
output = ValidCurrencyPairsStored event
```

```
foreach Currency in exchangeToRetrieve  
do appendCurrencyToList  
do  
compareTradeInformation  
identifyCurrenciesTradedTwice  
if InvalidCurrencyFound then:  
    send InvalidCurrencyFilteredOut event  
convertToCrossTradeFormat  
identifyInvalidPairs  
if InvalidPairsFound then:  
    send InvalidPairsFilteredOut event  
filterOutInvalidPairs  
@ExternalSystem(Database)  
storeValidPairInformation  
stop
```

```
event InvalidCurrencyFilteredOut = InvalidCurrencyName AND  
FilterTimestamp  
event ValidCurrencyPairsStored = ValidPairsList AND  
StorageTimestamp  
event InvalidPairsFilteredOut = InvalidPairName AND FilterTimestamp
```



Here's feedback on your workflow, "List of currency at an exchange retrieved," based on the criteria and practices from Domain-Driven Design (DDD) principles:

Strengths:

1. **Clear Workflow Structure:** The workflow is clearly structured, capturing the essential steps of retrieving, filtering, and storing valid currency pairs. The pseudocode format aligns with DDD best practices, making it easy to follow.
2. **Event Usage:** Events like `InvalidCurrencyFilteredOut`, `ValidCurrencyPairsStored`, and `InvalidPairsFilteredOut` are well-defined and used appropriately, reflecting significant occurrences within the workflow.
3. **Handling Errors:** The workflow includes mechanisms to handle invalid data (`InvalidCurrencyFilteredOut` and `InvalidPairsFilteredOut`), which enhances the robustness of the system by explicitly managing domain errors.

Weaknesses:

1. **Missing Input/Output Definitions:** While events are defined, the workflow lacks clear input and output parameters for each step. For instance, the input `exchangeToRetrieve` should be more explicit about its structure, and `ValidPairsList` needs further clarification about how it's generated throughout the process.
2. **Stop Conditions:** The stop condition, though implied by `storeValidPairInformation` and `stop`, is not clearly defined for when the workflow ends. A more explicit description of the stop condition for the system is needed to ensure clarity.
3. **Cross-Exchange Currency Identification:** The workflow does not fully address how cross-traded cryptocurrencies are identified, as outlined in the project specification. You should include steps that handle comparing currency pairs across multiple exchanges, which is a core functional requirement.
4. **Market Data Comparison for Arbitrage:** The workflow should detail how the system maintains and compares currency values across exchanges to identify arbitrage opportunities. This step is crucial for identifying currencies that are traded on multiple exchanges, which seems missing here.

Workflow Adjustments:

- **Input and Output Definitions:** Specify the input structure (`exchangeToRetrieve`), the format for the valid currency list (`ValidPairsList`), and ensure output parameters are consistently defined for each step.
- **Cross-Exchange Processing:** Incorporate a mechanism that compares the retrieved currencies across multiple exchanges to identify the pairs that are traded on more than one exchange. This is a core part of the functionality for identifying arbitrage opportunities.
- **Stop Conditions:** Make the conditions for stopping more explicit. It is unclear if the workflow should halt upon storing valid pairs or if there are external signals (e.g., completion of all data processing) that trigger the stop condition.
- **Real-Time Trading Data:** If the retrieved currency pairs are meant to be used for real-time arbitrage, the workflow should extend to handle subscribing to live data feeds or managing currency comparisons for arbitrage, as this is a core requirement.

This feedback helps you enhance the clarity, completeness, and functionality of your workflow, especially focusing on how it integrates within the broader trading system.



Can you be more specific about stop conditions



Certainly! In the context of your workflow, **stop conditions** are crucial for clearly defining when the process halts. These conditions ensure that the workflow doesn't continue running indefinitely and stops at an appropriate point in the data processing cycle. Let's break it down in relation to your workflow:

Current Workflow:

You have a `stop` statement after storing valid currency pairs (`storeValidPairInformation`).

However, it's unclear if this is the only stop condition or if there are other conditions in which the system needs to terminate.

Specific Suggestions for Stop Conditions:

1. Completion of Currency Processing:

- **Condition:** The workflow should stop after all currencies from the `exchangeToRetrieve` list have been processed.
- **Implementation:** Check if all items in the list have been iterated over. Once the loop through `foreach Currency in exchangeToRetrieve` completes, this would be the most natural stop condition.
- **Suggestion:** Add an explicit condition at the end of the loop that checks if there are no more currencies to process.

Example:

```
pseudo Copy code
foreach Currency in exchangeToRetrieve
...
stop // when the last currency is processed
```

2. Invalid Data Handling:

- **Condition:** The workflow should stop or interrupt processing if an invalid currency or invalid pair is found and has been successfully filtered out.
- **Implementation:** After emitting `InvalidCurrencyFilteredOut` or `InvalidPairsFilteredOut` events, you may want the system to either retry, terminate the workflow for specific cases, or continue. The current version implies that it continues, but there should be more explicit control over whether this triggers a termination.

Example:

```
pseudo  
  
if InvalidCurrencyFound then:  
    send InvalidCurrencyFilteredOut event  
    stop // Stop if critical errors in currency data occur
```

3. External Trigger (Optional):

- **Condition:** The workflow could stop based on an external condition, like a timeout, user intervention, or a notification from another system. This is common in real-time systems where an external system might signal the end of data retrieval.
- **Implementation:** Incorporate a step that listens for a termination signal from an external system (e.g., user-requested stop).

Example:

```
pseudo  
  
if StopSignalReceived then:  
    stop // External system sends a stop signal
```

4. Completion of Data Storage:

- **Condition:** After the valid currency pairs are stored in the database, the workflow stops.
- **Implementation:** Ensure that after `storeValidPairInformation`, the process halts if there's no further operation, especially if this is the final step.

Example:

```
pseudo  
  
storeValidPairInformation  
stop // Workflow stops after all valid pairs are stored
```

5. Final Data Consistency Check (Optional):

- **Condition:** The workflow could halt after performing a final consistency check to ensure that all valid currency pairs have been stored without errors.
- **Implementation:** Add an explicit check that validates the stored pairs. If everything is valid, the workflow stops.

Example:

```
pseudo  
  
if AllPairsStoredSuccessfully then:  
    stop // Workflow ends after confirming successful storage
```

In Summary:

The primary stop condition in your workflow should likely be when the loop finishes processing all currencies in the `exchangeToRetrieve` list. This should be clearly indicated in the workflow pseudocode. However, you should also include handling for invalid data (`InvalidCurrencyFilteredOut` and `InvalidPairsFilteredOut`) and ensure the system stops appropriately after these conditions are met.

You can also consider external stop signals (e.g., a timeout or user request), depending on how your system will be used in real-time scenarios.



Condition: The workflow should stop or interrupt processing if an invalid currency or invalid pair is found and has been successfully filtered out. Filtering out invalid pairs should not be a condition to stop the whole process



Thank you for the clarification! If filtering out invalid currencies or pairs should not stop the entire workflow but rather allow it to continue, then here's a more detailed adjustment:

Handling Invalid Currencies and Invalid Pairs:

In this scenario, the workflow should continue processing other currencies even if some are invalid or filtered out, ensuring that only the valid ones are stored. To achieve this, you can update the conditions for handling invalid data without stopping the workflow entirely. Instead, the system should just skip over the invalid data and proceed to the next steps.

Suggested Changes:

1. Handling Invalid Currencies:

- Condition: When an invalid currency is detected, the system should emit an `InvalidCurrencyFilteredOut` event and then continue to the next currency in the list without stopping the entire process.

Implementation: Instead of using `stop`, use a control flow mechanism like `continue` (or similar) to move to the next iteration in the loop. This ensures that invalid currencies are filtered out, but the workflow processes all remaining valid currencies.

Example:

```
pseudo  
foreach Currency in exchangeToRetrieve  
do appendCurrencyToList  
if InvalidCurrencyFound then:  
    send InvalidCurrencyFilteredOut event
```

2. Handling Invalid Pairs:

- **Condition:** Similar to handling invalid currencies, invalid currency pairs should be filtered out without halting the workflow. If an invalid pair is found, it should trigger an `InvalidPairsFilteredOut` event, filter the pair out, and then proceed with processing the rest of the pairs.

Implementation: Like with invalid currencies, ensure the system only skips over invalid pairs rather than stopping the entire workflow.

Example:

```
pseudo Copy code  
  
if InvalidPairsFound then:  
    send InvalidPairsFilteredOut event  
    filterOutInvalidPairs // Filter out invalid pairs and continue processing valid on
```

Overall Workflow Adjustments:

Here's how you can incorporate these changes while ensuring the workflow doesn't terminate upon encountering invalid data:

```
pseudo Copy code  
  
workflow „List of currency at an exchange retrieved“ =  
  
    input = CurrencyListRetrievalInitiated event AND exchangeToRetrieve  
    output = ValidCurrencyPairsStored event  
  
    foreach Currency in exchangeToRetrieve  
        do appendCurrencyToList  
        if InvalidCurrencyFound then:  
            send InvalidCurrencyFilteredOut event  
            continue // Do not stop, skip to next currency  
        do compareTradeInformation  
        do identifyCurrenciesTradedTwice  
        if InvalidPairsFound then:  
            send InvalidPairsFilteredOut event  
            filterOutInvalidPairs // Filter invalid pairs but continue processing  
        do convertToCrossTradeFormat  
    @ExternalSystem(Database)  
    storeValidPairInformation  
    stop
```

Why Use `continue` Instead of `stop`:

- `continue` ensures that the workflow moves to the next currency when an invalid one is found, without halting the entire process. This fits your requirement of filtering out invalid pairs without stopping the workflow.

Based on the feedback, I improved my workflow by specifying that when invalid inputs are encountered, we continue to the next element instead of terminating the whole workflow. I also added actual filtering actions when such inputs are met instead of simply sending event indicating we found something invalid.