Data Aggregation in Microservice Architecture

https://doi.org/10.3991/ijoe.v15i12.11095

Ivo Damyanov
South-West University, Blagoevgrad, Bulgaria
damianov@swu.bg

Abstract—In a microservice architecture aggregation of data collected from different sources is a common task. Today's technology trends require us to exchange data that is no longer tabular. JSON data format has gained popularity among web developers, and has become the main format for exchanging information over the web. When we need to aggregate data collected from the web, storing it into relational database just to perform this task and pass it to the next unit to process or display it often is an exaggerated action. In this paper, we discuss a scenario and an implementation of in-memory preprocessing and aggregating data using lazy evaluation, value tuples and LINQ.

Keywords—Microservice architecture, JSON, data aggregation, LINQ, message broker, lazy evaluation.

1 Introduction

In the present distributed environment often we need to collect data from different sources adjust it, aggregate it and display it on screen or store it for future processing. Source format can be structured (hierarchical or tabular) or semi-structured (free text). There are few architectural patterns for building systems to collect and process data but mainly we can divide them into two groups, namely centralized and distributed. Centralized approach means that we have a single point of control over what and when we process, whether distributed applications are built as a bunch of independent nodes where each one performs a single task, independently, and communicates with the rest of the nodes via a messaging system. Maintaining changes in centralized application reflect in redeploy of the whole application. Implementing pluggable architecture also is not an easy solution. On the other hand, distributed approach has it benefits since we can maintain each node (worker) independently to keep it up to date with the changes in the source.

Microservice architectural pattern is one of the emerging architectural patterns nowadays, which is built around the following core concepts: separately deployed units, service components and distributed nature [1]. Microservice architectural pattern is a type of Service Oriented Software Architecture (SOA) with a bunch of independent (autonomous) service components that make up our app. The biggest advantages of applying this architectural pattern are the easily deployable and testable components as well as the ability to scale horizontally. Loose coupling and high cohe-

sion also characterize this pattern. In scenarios where we collect data from independent sources and process it (or at least aggregate it) is one of the motivated choices for applying this pattern especially when we aim for real-time data processing and visualizing data.

In this paper we discuss a common scenario for an application that retrieves data from several independent sources, adjusts its format and presentation (measures, precision, language, etc.), aggregates it and displays it as soon as new data is collected on a dashboard [2]. Since sources are independent, the data feeds may be processed in different order every time. Aggregated feed is displayed on screen dashboard. Such kind of scenario is common in betting industry where it was firstly implemented by the author. We can recognize similar scenarios also in scientific experiments, manufacture monitoring, etc. As was already motivated for such a scenario microservice architecture fits well. Every worker unit communicates and shares data via asynchronous messaging broker (like RabbitMQ [3]) or distributed streaming platform (like Kafka [4]) and data is processed in memory rather than using the power of relational database systems. We will discuss a solution that incorporates LINQ, value tuples and lazy evaluation implemented with C# language for .NET Framework.

2 Distributed Apps and Data Formats

Using heterogeneous and distributed environment means each independent node needs to exchange information with the rest of the application. After the era of binary data exchange protocols and formats, in the time of web apps and services, applications expose and exchange data in textual and human readable formats. Intermediate format for relational (tabular) data becomes the textual format of comma separated values (CSV). Despite the compact nature of this format, it is not self-descriptive, and usually we need to share additional information for specifying column data types or infer data types by preprocessing data, but often this leads to some mismatches. This weakness narrows the usage of this format and it is mainly used when we feed data into spreadsheet processor or relational databases.

Today's technology trends require us to exchange data that is no longer tabular. Over the past decades, XML [5] has enabled heterogeneous computing environments to share information over the web. XML data format is an independent and flexible platform. XML is well suited for data exchange, since XML documents are self-described, easily parsed and can represent complex data structures. One of the huge advantages compared to CSV format is that XML documents can be easily validated against XML Shema, which, in fact, is also presented in XML format. Used as a base data format on which SOAP, the main protocol in the beginning of the SOA era (now it is less popular), was based. Restful Services and Web UI frameworks bring JavaScript object notation format (JSON) to the focus.

JSON is a data format based on the data types of the JavaScript programming language. It has gained tremendous popularity among web developers, and has become the main format for exchanging information over the web [6]. JSON format is self-descriptive and with much more compact syntax compared to XML. Unlike XML it

lacks standard schema for defining the legal building blocks in a given JSON document. JSON Schema [7] is a general attempt to define a schema language for JSON documents. The definition is still far from being standard, but there is already a growing number of applications that support JSON schema definition. In the next lines of code CSV, XML and JSON data formats are demonstrated.

```
Sport, Championship, Event
"Soccer", "Friendly Games", "Sweden - Norway"
"Soccer", "Friendly Games", "Kosovo - Malta"
```

```
<Data>
<Sport name="Soccer">
  <Championship name="Friendly Games">
    <Event name="Sweden - Norway" />
    <Event name="Kosovo - Malta" />
  </Championship>
</Sport>
</Data>
```

```
{ "Data": [
        {
            "Name": "Soccer",
            "Championships": [
                {
                    "Name": "Friendly Games",
                    "Events": [
                        {
                            "Name": "Sweden - Norway"
                        },
                        {
                            "Name": "Kosovo - Malta"
                        }
                    ]
                }
            ]
        }
    ]
}
```

As we already said, technology trends require us to process hierarchical data and JSON nature is hierarchical. JSON is natively maintained format by major messaging brokers, and it can be easily processed from C# code using sophisticated libraries with good performance and easy to use API.

3 Scalable Data Aggregation wit LINQ and Value Tuples

With .NET Framework 3.5 in 2007, a Language INtegrated Queries (or LINQ) was introduced in .NET universe to bridge the gap between programming languages and databases and this became one of the most useful features for data processing [8]. LINQ defines an API pattern that enables querying of any data collection. The query operator set includes filtering (where), mapping (select), monadic bind (selectMany), sorting (orderby) and partitioning (groupBy). We can freely compose any queries. [9] In .NET Languages like C# LINQ uses the canonical interface for collections IEnumerable<T> and uses delegates Func<S,T> to represent computations. The standard query operators are defined in the Linq.Enumerable class with prototypes as shown with the next code lines.

```
IEnumerable<T> Select<S,T>(IEnumerable<S> source,
  Func<S,T> selector)
IEnumerable<T> SelectMany<S,T>(IEnumerable<S> source,
  Func<S,IEnumerable<T>> selector)
IEnumerable<T> Where<T>(IEnumerable<T> source,
  Func<T,bool> predicate)
```

LINQ syntax is like XQuery comprehensions in the form from-where-select, as show bellow.

```
from sport in sports
where sport.UsesBall()
select sport.Name
```

If we assume that our data is highly hierarchical we are forced to fit all the processing (traversal walk-through data) in a nested loops. For example, if we have a hierarchy Level 1 → Level 2 → Level 3 → Data and we need to map levels from different sources (for example they are in different languages) we can loop thorough data with code like this

```
foreach (var lvl1 in jsonData) {
  foreach (var lvl2 in lvl1.Levels) {
    foreach (var lvl3 in lvl2. Levels) {
      foreach (var data in lvl3.Data) {
        //process data
      }
    }
  }
}
```

Getting data that are more complex will make our code hardly maintainable. Instead, we can use LINQ to make it more declarative in the following way:

```
var query = from lvl1 in jsonData
  from lvl2 in lvl1.Levels
  from lvl3 in lvl2.Levels
  from data in lvl3.Data
  select {
    //process data
  }
```

Any preliminary data processing for instance mapping level's names or turning data in common measure units, etc. can be done in a separate method over source data and yield amended values. The only requirement is to align IEnumerable as result type and to implement the necessary method for amending (i.e. the method Preprocess).

```
IEnumerable<Level> PreprocessLevel(
      IEnumerable <Level> levels) {
  foreach (var lvl in levels) {
    yield return Preprocess(lvl);
  }
}
```

Since in our scenario we do not know how often a source can provide data and we cannot have strict order of receiving data (they arrive as soon as they are available but in asynchronous manner) we can model our aggregator as two LINQ queries with join operation. Since data comes from different sources, we do not have primary keys or other uniquely matching identities, so we need to perform matching by value tuples, where each tuple contains (some) values and (some) data from the whole hierarchy (Level1, Level2, Level3, Data). Then the resulting queries are as follows:

```
  var newDataQuery =
  from lvl1 in PreprocessLevel(source, jsonData)
  from lvl2 in PreprocessLevel(source,lvl1.Levels)
  from lvl3 in PreprocessLevel(source,lvl2.Levels)
  from data in PreprocessData(source, lvl3.Data)
    select new {
        //create anonymous class object
      lvl1SomeValue = lvl1.SomeValue,
      lvl2SomeValue = lvl2.SomeValue,
      lvl3SomeValue = lvl3.SomeValue,
      dataSomeData = data.SomeData
    };

var aggregateDataQuery =
  from lvl1 in jsonAggregatedData
  from lvl2 in lvl1.InnerLevels
  from lvl3 in lvl2.InnerLevels
```

```
from data in lvl3.Data
join entity in newDataQuery on
(lvl1.SomeValue, lvl2.SomeValue,
      lvl3.SomeValue, data.SomeData)
equals
(entity.lvl1SomeValue, entity.lvl2SomeValue,
      entity.lvl3SomeValue, entity.dataSomeData)
select Aggregate(
    (lvl1.SomeValue, lvl2.SomeValue,
        lvl3.SomeValue, data.SomeData),
    (entity.lvl1SomeValue, entity.lvl2SomeValue,
        entity.lvl3SomeValue, entity.dataSomeData),
    source);
```

Now evaluation of aggregate Data Query will return new feed of aggregated data. We need to create a custom Aggregate function that will do the aggregation as required by the system design.

This solution frees us from hardcoded loops and we have much more linear structures than nested blocks. Using anonymous classes and tuples makes it much simpler to match data that needs to be aggregated.

4 Conclusion

Processing data in a distributed environment is a common task nowadays. The presented solutions with lazy evaluation, LINQ and value tuples give us freedom to process data as soon as it arrives regardless of the order in which we collected it. This is important especially when we need to display data in a real-time manner on screen.

5 References

[1] Mark Richards (2015), Software Architecture Patterns. Understanding Common Architecture Patterns and When to Use Them, O'Reilly Media Inc.
[2] Damyanov, I., & Tsankov, N. (2019). On the Possibilities of Applying Dashboards in the Educational System, TEM Journal, 8(2), 424- 429.
[3] https://www.rabbitmq.com
[4] https://kafka.apache.org
[5] https://www.w3.org/XML
[6] Pezoa, F., Reutter, J. L., Suarez, F., Ugarte, M., & Vrgoč, D. (2016). Foundations of JSON schema. In Proceedings of the 25th International Conference on World Wide Web (pp. 263-273). https://doi.org/10.1145/2872427.2883029
[7] http://json-schema.org/
[8] Meijer, E. (2011). The world according to LINQ. Queue, 9(8).
[9] Meijer, E., Beckman, B., & Bierman, G. (2006). LINQ: reconciling object, relations and XML in the .NET framework. In Proceedings of the 2006 ACM SIGMOD international

conference on Management of data (pp. 706-706). ACM. https://doi.org/10.1145/1142473.1142552

6 Author

Ivo Damyanov has a master's degree in Mathematics and he holds a doctor's degree in Computer science. He is an assistant professor at the Department of Informatics. His professional and scientific research interests are in the fields of: metaprogramming, domain-specific languages and code generation, discrete functions, e-learning and distance learning.

Article submitted 2019-05-23. Resubmitted 2019-07-15. Final acceptance 2019-07-19. Final version published as submitted by the authors.