Ephemeral Data Handling in Microservices

Fabrizio Montesi[1], Larisa Safina[1,2], and Stefano Pio Zingaro[3]

[1]University of Southern Denmark, [2]Innopolis University, [3]Università di Bologna/INRIA

Introduction. Modern application areas for software systems, like eHealth, the Internet of Things, and Edge Computing, need to address two requirements: velocity and variety [1]. Velocity concerns managing high throughput and real-time processing of data. Variety means that data might be represented in heterogeneous formats, complicating their aggregation, query, and storage. Recently, in addition to velocity and variety, it has become increasingly important to consider *ephemerality* (of data handling) [2], where data must be processed in real-time but not persist. The rise of ephemeral data is due to scenarios with heavy resource constraints (e.g., storage, battery), as in the Internet of Things and Edge Computing, or new regulations that may limit what data can be persisted, like the GDPR, as in eHealth.

Programming data handling correctly can be time consuming and error-prone with a general-purpose language. Thus, often developers use a query language, paired with an engine to execute them [3]. When choosing the query execution engine, developers can either *A*) use a database management system (DBMS) executed outside of the application, or *B*) include a library that executes queries using the application memory.

Problem statement. Approach *A*) is the most common. Since the early days of the Web, programmers integrated application languages with relational (SQL-based) DBMSs for data persistence and manipulation. This pattern continues nowadays, where relational databases share the scene with new NoSQL DBMSs, like MongoDB [4] and Apache CouchDB [5], which are document-oriented. Document-oriented databases natively support tree-like nested data structures (typically in the JSON format). Since data in modern applications is typically structured as trees (e.g., JSON, XML), this removes the need for error-prone encoding/decoding procedures with table-based structures, as in relational databases. However, when considering ephemeral data handling, the issues of approach *A*) overcome its benefits even if we consider NoSQL DBMSs:
Drivers and Maintenance. An external DBMS is an additional standalone component that needs to be installed, deployed, and maintained. To interact with the DBMS, the developer needs to import in the application specific drivers (libraries, RESTful outlets). As with any software dependency, this exposes the applications to issues of version incompatibility.
Security Issues. The companion DBMS is subject to weak security configurations and query injections, increasing the attack surface of the application.
Lack of Tool Support. Queries to the external DBMS are typically black-box entities (e.g., encoded as plain strings), making them opaque to analysis tools available for the application language (e.g., type checkers).
Decreased Velocity and Unnecessary Persistence. Integration bottlenecks and overheads degrade the velocity of the system. Bottlenecks derive from resource constraints and slow application-DB interactions; e.g., typical database connection pools represent a potential bottleneck in the context of high data-throughput. Also, data must be inserted in the database and eventually deleted to ensure ephemeral data handling. Overheads also come in the form of data format conversions.
Burden of Variety. The DBMS typically requires a specific data format for communication, forcing the programmer to develop ad-hoc data transformations to encode/decode

data in transit (to insert incoming data and returning/forwarding the result of queries). Implementing these procedures is cumbersome and error-prone.

On the other side, approach *B*) is less well explored. However, it holds potential for ephemeral data handling. Approach *B*) avoids by design the first two issues of approach *A*) . The issue on *tools* is sensibly reduced, since both queries and data can be made part of the application language. The issue on *velocity* is also tackled by design. There are less resource-dependent bottlenecks and no overhead due to data insertions (there is no DB to populate) or deletions (the data disappears from the system when the process handling it terminates). Data transformation between different formats is still an issue here since, due to *variety*, the developer must convert incoming/outgoing data into/from the data format supported by the query engine. Examples of implementations of approach *B*) are LINQ [3] and CQEngine [6]. While LINQ and CQEngine grant good performance (velocity), variety is still an issue. Those proposals either assume an SQL-like query language or rely on a table-like format, which entail continuous, error-prone conversions between their underlying data model and the heterogeneous formats of the incoming/outgoing data.

Contribution. Inspired by approach *B*, we implemented a framework in the Jolie programming language [7] for ephemeral data handling in microservices; the building blocks of software for our application areas of interest. Our framework includes a query language and an execution engine, that we formalised instantiating a sound version of the MongoDB query language [8].

In our presentation we will illustrate the features of our framework through a non-trivial eHealth use case, which describes the handling of the data and the workflow of the diagnostic algorithm taken from [9], where the authors delineate a diagnostic algorithm to detect cases of encephalopathy. The handling follows the principle of "data never leave the hospital" in compliance with the GDPR [10]. While the algorithm described in [9] considers a plethora of clinical tests to signal the presence of the neurological condition, we focus on two early markers for encephalopathy: fever in the last 72 hours and lethargy in the last 48 hours. That data is collectible by commercially-available smart-watches and smart-phones [11]: body temperature and sleep quality. We report in Listing 1, in a JSON-like format, code snippets exemplifying the two kinds of data structures. At lines 1–2, we have a snippet of the biometric data collected from the smart-watch of the patient. At lines 4–6 we show a snippet of the sleep logs [12]. Both structures are arrays, marked `[ ]`, containing tree-like elements, marked `{ }`. At lines 1–2, for each `date` we have an array of detected temperatures (`t`) and heart-rates (`hr`). At lines 4–6, to each year (`y`) corresponds an array of monthly (`M`) measures, to a month (`m`), an array of daily (`D`) logs, and to a day (`d`), an array of logs (`L`), each representing a sleep session with its start (`s`), end (`e`) and quality (`q`).

Listing 1: Snippets of biometric (line 1) and sleep logs (lines 3–5) data.

```
1 [{date:20181129,t:[37,...],hr:[64,...]},
2  {date:20181130,t:[36,...],hr:[66,...]},...]
3
4 [{y:2018,M:[...,{m:11,D:[{d:29,L:[{s:"21:01",e:"22:12",q:"good"},
5 {s:"22:36",e:"22:58",q:"good"},...]},{d:30,L:[
6 {s:"20:33",e:"22:12",q:"poor"},...]},...]},...]},...]
```

On the data structures above, we define a Jolie microservice, reported in Listing 2, which describes the handling of the data and the workflow of the diagnostic algorithm,

using our implementation of TQuery. The example is detailed enough to let us illustrate all the operators in TQuery: `match`, `unwind`, `project`, `group`, and `lookup`. Note that, while in Listing 2 we hard-code some data (e.g., integers representing dates like `20181128`) for presentation purposes, we would normally use parametrised variables.

In Listing 2, line 1 defines a request to an external service, provided by the **HospitalIT** infrastructure. The service offers functionality `getPatientPseudoID` which, given some identifying `patientData` (acquired earlier), provides a pseudo-anonymised identifier — needed to treat sensitive health data — saved in variable `pseudoID`.

At lines 2–6 (and later at lines 9–17) we use the chaining operator `|>` to define a sequence of calls, either to external services, marked by the `@` operator, or to the internal TQuery library. The `|>` operator takes the result of the execution of the expression at its left and passes it as the input of the expression on the right.

At lines 2–6 we use TQuery operators `match` and `project` to extract the recorded temperatures of the patient in the last 3 days/72 hours.

At line 2 we evaluate the content of variable `credentials`, which holds the certificates to let the Hospital IT services access the physiological sensors of a given patient. In the program, `credentials` is passed by the chaining operator at line 3 as the input of the external call to functionality `getMotionAndTemperature`. That service call returns the biometric data (Listing 1, lines 1–2) from the **SmartWatch** of the patient. While the default syntax of service call in Jolie is the one with the double pair of parenthesis (e.g., at line 1 Listing 2), thanks to the chaining operator `|>` we can omit to specify the input of `getMotionAndTemperature` (passed by the `|>` at line 3) and its output (the biometric data exemplified at Listing 1) passed to the `|>` at line 4. At line 4 we use the TQuery operator `match` to filter all the entries of the biometric data, keeping only those collected in the last 72 hours/3 days (i.e., since `20181130`). The result of the `match` is then passed to the `project` operator at line 5, which removes all nodes but the temperatures, found under `t` and renamed `in temperatures` (this is required by the interface of functionality `detectFever`, explained below). The `projection` also includes in its result the `pseudoID` of the patient, `in` node `patient_id`. We finally store (line 6) the prepared data in variable `temps` (since it will be used both at line 7 and 16).

At line 7, we call the external functionality `detectFever` to analyse the temperatures and check if the patient manifested any fever, storing the result in variable `hasFever`.

Listing 2: Encephalopathy Diagnostic Algorithm.

```
1  getPatientPseudoID@HospitalIT( patientData )( pseudoID );
2  credentials
3  |> getMotionAndTemperature@SmartWatch
4  |> match {date == 20181128 || date == 20181129 || date == 20181130 }
5  |> project {t in temperatures, pseudoID in patient_id }
6  |> temps;
7  detectFever@HospitalIT( temps )( hasFever );
8  if( hasFever ){
9    credentials
10   |> getSleepPatterns@SmartPhone
11   |> unwind {M.D.L }
12   |> project{y in year,M.m in month,M.D.d in day,M.D.L.q in quality}
13   |> match {year == 2018 && month == 11 && ( day == 29 || day == 30 ) }
14   |> group { quality by day, month, year }
15   |> project {quality, pseudoID in patient_id }
16   |> lookup { patient_id == temps.patient_id in temps }
17   |> detectEncephalopathy@HospitalIT }
```

After the analysis on the temperatures, `if` the patient `hasFever` (line 8), we continue testing for lethargy. To do that, at lines 9–10, we follow the same strategy described for lines 2–3 to pass the `credentials` to functionality `getSleepPatterns`, used to collect the sleep logs of the patient from her **`SmartPhone`**. Since the sleep logs are nested under years, months, and days, to filter the logs relative to the last 48 hours/2 days, we first flatten the structure through the `unwind` operator applied on nodes `M.D.L` (line 11). For each nested node, separated by the dot (`.`), the `unwind` generates a new data structure for each element in the array reached by that node. Concretely, the array returned by the `unwind` operator at line 11 contains all the sleep logs in the shape:

```
[    {year:2018,M:[{m:11,D:[{d:29,L:[{s:"21:01",e:"22:12",q:"good"}]}]}]},
     {year:2018,M:[{m:11,D:[{d:29,L:[{s:"22:36",e:"22:58",q:"good"}]}]}]}  ]
```

where there are as many elements as there are sleep logs and the arrays under `M`, `D`, and `L` contain only one sleep log. Once flattened, at line 12 we modify the data-structure with the `project` operator to simplify the subsequent chained commands: we rename the node `y` `in` `year`, we move and rename the node `M.m` `in` `month` (bringing it at the same nesting level of `year`); similarly, we move `M.D.d`, renaming it `day`, and we move `M.D.L.q` (the log the quality of the sleep), renaming it `quality` — `M.D.L.s` and `M.D.L.e`, not included in the `project`, are discarded. On the obtained structure, we filter the sleep logs relative to the last 48 hours with the `match` operator at line 13. At line 14 we use the `group` operator to aggregate the `quality` of the sleep sessions recorded in the same day (i.e., grouping them `by` `day`, `month`, and `year`). Finally, at line 15 we select, through a `projection`, only the aggregated values of `quality` (getting rid of `day`, `month`, and `year`) and we include under node `patient_id` the `pseudoID` of the patient. That value is used at line 16 to join, with the `lookup` operator, the obtained sleep logs with the previous values of temperatures (`temps`). The resulting, merged data-structure is finally passed to the **`HospitalIT`** services by calling the functionality `detectEncephalopathy`.

References

[1] D. P. Mehta and S. Sahni, *Handbook of data structures and applications*. Chapman and Hall/CRC, 2004.

[2] E. Shein, "Ephemeral data," *Communications of the ACM*, vol. 56, no. 9, pp. 20–22, 2013.

[3] J. Cheney, S. Lindley, and P. Wadler, "A practical theory of language-integrated query," *ACM SIGPLAN Notices*, vol. 48, no. 9, pp. 403–416, 2013.

[4] MongoDB Inc., "MongoDB Website." https://www.mongodb.com/, 2018.

[5] Apache Software Foundation, "CouchDB Website." https://couchdb.apache.org/, 2018.

[6] Niall Gallagher, "CQEngine - Collection Query Engine." https://github.com/npgall/cqengine, 2018.

[7] F. Montesi, C. Guidi, and G. Zavattaro, "Service-oriented programming with jolie," in *Web Services Foundations* (A. Bouguettaya, Q. Z. Sheng, and F. Daniel, eds.), pp. 81–107, Springer, 2014.

[8] E. Botoeva, D. Calvanese, B. Cogrel, and G. Xiao, "Expressivity and complexity of mongodb queries," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 98, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[9] F. Vigevano and P. D. Liso, "Chapter 11 - differential diagnosis," in *Acute Encephalopathy and Encephalitis in Infancy and Its Related Disorders* (H. Yamanouchi, S. L. Moshé, and A. Okumura, eds.), pp. 81 – 85, Elsevier, 2018.

[10] N. Rose, "The human brain project: Social and ethical challenges," *Neuron*, vol. 82, no. 6, pp. 1212 – 1215, 2014.

[11] J. A. Bunn, J. W. Navalta, C. J. Fountaine, and J. D. REECE, "Current state of commercial wearable technology in physical activity monitoring 2015–2017," *International journal of exercise science*, vol. 11, no. 7, p. 503, 2018.

[12] S. M. Thurman, N. Wasylyshyn, H. Roy, G. Lieberman, J. O. Garcia, A. Asturias, G. N. Okafor, J. C. Elliott, B. Giesbrecht, S. T. Grafton, S. C. Mednick, and J. M. Vettel, "Individual differences in compliance and agreement for sleep logs and wrist actigraphy: A longitudinal study of naturalistic sleep in healthy adults," *PLOS ONE*, vol. 13, pp. 1–23, 01 2018.