delf-rs: A DelF-style DDL and API

Mary McGrath
Brown University

1 Abstract

In a modern user-centric web application, many data points of various types (e.g. photos, posts, comments, likes) will be associated with a user. The European Union's General Data Protection Regulation (GDPR) [10] enshrines data subjects with the right to deletion. In a web application with a complex graph of types, dependencies, and interactions, guaranteeing that all of a user's data (and no extra data) is deleted upon request can be a difficult task requiring bespoke deletion code and deep knowledge of the code base. Facebook has introduced DelF [7] as their approach to solving this problem. DelF provides rich functionality and shows that its usage decreases deletion errors, and increases developer productivity, however it is deeply integrated and reliant on Facebook's unique infrastructure. delf-rs is a proof-of-concept which implements the core functionality of DelF in a system-agnostic and extensible manner that can be used by almost any web application.

2 Introduction & Background

Since GDPR enforcement began in 2018, GDPR Enforcement Tracker shows there have been 17 cases citing violations of article 17, the right to deletion [2]. Web applications have a moral and legal obligation to delete the data of a subject upon request [10]. However, the data of a subject can be sprawling - crossing the boundaries of product features and intertwined with the data of other users. In the example of a social media application, when a user is deleted, their account, posts, likes, and friendships should be deleted, but not the things they liked, or their friends. These deletion rules are typically incorporated throughout the code base by each developer as they add features which interact with the data.

This approach presents a few problems. (1) There is no guarantee that a deletion leaves no dangling data items (e.g. friendships where one friend no longer exists). (2) Developers must self-manage the deletion process without the help of static analysis, writing bespoke code for each deletion. (3) It requires developers to have extensive knowledge of the code base and how the feature they are building interacts with other features. Facebook introduced DelF [7] as their approach to solving these problems.

DelF is a deletion framework built upon Facebook's existing architecture for defining data in their various storage systems. The data definition language (DDL) describes the underlying storage of items as objects and edges, each with metadata about how to store the item. DelF adds deletion notations to this DDL. There are 6 deletion types for objects, and 3 for edges. The edge deletion types (`deep`, `shallow`, `refcount`) define whether when an edge is deleted, whether the object it pointed to should also be deleted. The object deletion types (`by_any`, `by_x_only`, `directly`, `directly_only`, `short_ttl`, `not_deleted`) indicate what can delete that object, and in the case of `short_ttl` (short time-to-live) the mechanism for deletion.

In addition to this annotation, Facebook's implementation of DelF includes an asynchronous service which guarantees deletion of all relevant items in the graph, and additionally provides the ability to undo a deletion, static analysis, and dynamic analysis. Facebook found in [7] that DelF improved accuracy of deletion, improved developer productivity, and didn't noticeably impact throughput of requests.

delf-rs aims to show that the core functionality of DelF can be implemented outside of Facebook's infrastructure. delf-rs implements a DDL inspired by Facebook's, but with only the information needed for deletion and not that needed for defining and creating the items in storage. It additionally implements the deletion algorithm for all nine deletion types and static analysis-based validation of the schema. The asynchronous service is implemented as a REST API, which allows for system and language agnostic use of the service by any program. The dynamic analysis and restoration of deleted data is outside of the scope of this work.

3 Design

delf-rs is implemented in Rust [9] as a crate that is available as both an executable (`delf`) and a library. The library has three

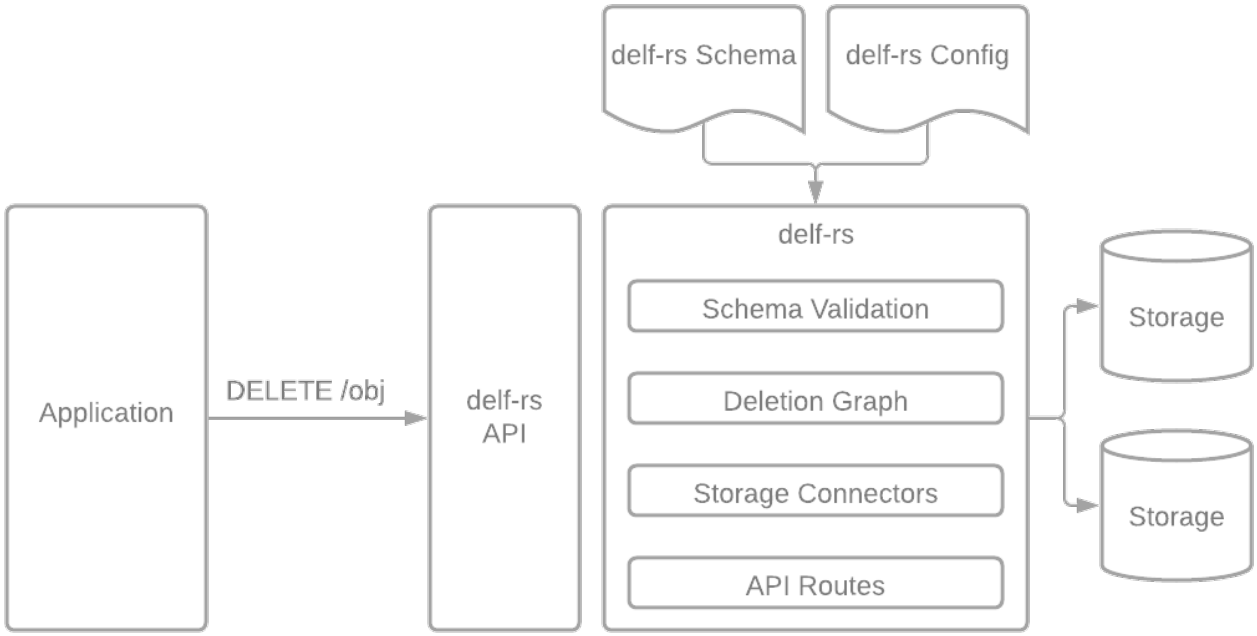

Figure 1: **delf-rs architecture** The core funcionality of delf-rs is schema validation, the deletion graph, connectors to storage, and the api routes. All of this functionality is dependent on a user-provided schema (using the DDL) and configuration for how to connect to the storages. The CLI provides programs to start the REST API server as well as validate the graph.

sub-modules: 1) `graph`, which is the core data structure and where the deletion algorithm and validation is implemented; 2) `storage`, an implementable trait as well as any implementations (currently only MySQL [3]); and 3) `api` which defines the endpoints for the REST API. The executable is a command line interface (CLI) with two programs: `run`, which starts the REST API, and `validate`, which checks the schema is valid and complete. Figure 1 shows the architecture of delf-rs.

3.1 Data Definition Language (DDL)

The DDL is a YAML [6] specification, taking a similar form to the DDL Facebook uses for defining and creating items in their storage systems. Because delf-rs only deletes items, but does not create them, it requires less information about the objects and edges in the storage. delf-rs needs the unique identifiers and relationships between objects, as well as some minimal type information, but does not need specific type information, nor information about non-identifier attributes of the object (e.g. a user's name, or favorite color).

Listing 3.1 contains a sample schema written using the delf-rs DDL. A complete description of the DDL can be found in the delf-rs docmuentation (`https://mcmcgrath13.github.io/delf-rs/delf/index.html`).

3.2 REST API

To allow delf-rs to be used in a system agnostic manner, the deletion methods are exposed as a deletion microservice via a REST API. This also allows a common deletion graph to be used by several webservices that use the same underlying storage and data. The library includes the definition of the routes as well as the definition of the microservice. The CLI `run` command starts the service.

Objects deleted via the API are considered directly deleted, meeting the criteria of the `directly` and `directly_only` deletion types.

The current implementation of delf-rs completes the deletion synchronously within the HTTP `DELETE` request. However, future work could include making this asynchronous with retries to better match Facebook's implementation and the robustness that model provides. This proof-of-concept will return an error code to the requester if the deletion failed, however that does not guarantee that no deletions were completed before the failure was encountered.

As a side process to the REST API, a thread periodically checks `short_ttl` objects if they are ready for deletion based on an expiration timestamp that is an attribute of the object. In combination, the REST API and `short_ttl` thread jointly provide a working implementation of all nine deletion types.

```yaml
object_type:
  name: photo
  storage: mysql
  deletion: directly
  id: photo_id
  edge_types:
    - name: handle
      to:
        object_type: photo_blob
        field: photo_id
      deletion: deep
    - name: created_by
      to:
        object_type: user
        field: photo_id
        mapping_table: user_photos
      deletion: shallow
      inverse: created_photo

---
object_type:
  name: photo_blob
  id: id
  storage: blob
  deletion: by_any
  edge_types: []

---
object_type:
  name: user
  storage: mysql
  deletion: directly_only
  id: user_id
  edge_types:
    - name: created_photo
      to:
        object_type: photo
        field: user_id
        mapping_table: user_photos
      deletion: shallow
      inverse: created_by
```

Listing 1: A sample schema using the delf-rs DDL. Each YAML document describes an object in the database and its outbound edges.

3.3 Deletion Graph

The deletion graph implements the algorithm for the deletion of objects and edges according to their edge type and the state of the data. It holds the parsed and validated delf schema in memory. Upon a call to delete an item in the graph, the deletion is validated against the type of each item, then if the deletion is valid, it is deleted and any outbound edges it has are deleted and traversed to determine if any of the connected objects should also be deleted. Deletion terminates when all objects and edges downstream of the traversal starting point have been deleted, or determined to not be eligible for deletion. The traversal of the graph is depth first.

The deletion types used in delf-rs are as described in DelF [7] with the following exceptions. `custom` deletion types are not supported by delf-rs. These were largely included in DelF to allow for backwards compatibility during the transition period, then very rarely selected in usage with a DelF-driven deletion system. The DelF implementation of `short_ttl` scheduled the object for deletion on its creation, however, since delf-rs is not as tightly integrated with the storage, it instead requires objects using the `short_ttl` deletion type store the time they should be deleted as an attribute or field of the object. delf-rs periodically checks this deletion time then if it is before the present time, deletes the object.

3.4 Storage Connectors

A web application may have multiple types of storage specialized to the type of data being stored (e.g. relational, graph, blob). While the proof-of-concept includes only support for MySQL, it is designed to be extended to many storage types. A generic high-level API defines a delf-rs compatible storage. Any storage type that implements those methods would be compatible with delf-rs. This enables usage with web apps with a a variety of backing storage, as well as allowing the deletion graph to span multiple backing storage types.

3.5 Validation

The heuristics designed in [7] improved the reliability and usage of DelF by software engineers. delf-rs includes the static analysis described in DelF, but does not implement the dynamic analysis as it requires tighter integration with the underlying storage. The static analysis requires that all objects in the deletion graph are traversed when starting at an object with the delete type of `directly`, `directly_only`, `short_ttl` or `not_deleted` and following edges with deletion type of `deep` or `refcount`. This ensures no objects can be left dangling upon deletion of their connected objects.

In addition to the static analysis described in DelF, delf also implements validation that the underlying storage is as described in the schema. This step is not required in DelF as the schema also defines and creates those objects in the storage, meaning they can not be out of sync.

4 Implementation

delf-rs is implemented in 926 lines of Rust [9] code. The delf graph is built on top of petgraph's [4] directed graph. The REST API uses the rocket [5] web framework. The proof-of-

concept storage is implemented using diesel [1], an ORM, to connect to and query the MySQL database.

Type safety is a prominent feature of Rust, which ensures that if a schema is successfully read into a deletion graph and validation passes, that runtime errors are unlikely. However, this also requires assigning a specific type to the inputs and results of any queries to the storage. delf-rs only needs to know how to delete an item in the storage, but not the exact type. This is addressed by assigning fields a type of `number` or `string` allowing the queries to the storage to be correctly formatted and executed.

5 Evaluation

HotCRP [8], an open-source conference management platform, was used to validate the proof-of-concept. The data model centers around users and papers with other objects existing in support of those objects. HotCRP is backed by a MySQL database containing 25 tables. Most of these tables are considered objects by delf-rs, but a few are functionally edges, providing a many-to-many mapping between two object tables. HotCRP's database was containerized and loaded with test data to validate delf-rs.

The database was able to be described as a delf schema with 310 lines of YAML. The schema consists of 16 objects and 33 edges. All deletion types were used in the schema, though a few usages were partially contrived based on how HotCRP could use delf rather than its current deletion logic. This was done to ensure all deletion types were validated.

With a freshly loaded test database, the `delf validate` and `delf run` programs were run. All items in the delf graph existed in the MySQL database as described and all objects were reachable via the static analysis described in sub-section 3.5. An erroneous version of the schema was also validated and found to have all of the introduced errors. `DELETE` requests of both objects and edges were made to the running API and manually validated to have deleted all relevant objects. This included the validation that a `refcount` edge only deletes the object it is point to if it is the last remaining reference, and also the validation that the `short_ttl` objects were deleted as scheduled as well as by an incoming edge. All deletion types were shown to work as described in DelF [7].

6 Future Work

Future work is needed to study delf-rs and mature it from a proof-of-concept to a production-ready system. In addition to implementing the fully asynchronous deletion with retries, and the ability to recover deletions as in DelF, implementing more storage types is important to validating that delf-rs can be a viable solution for any web service. Additionally, adapting more test applications to use delf-rs would be valuable in ensuring the DDL covers typical data models found across a variety of use cases and storage types.

7 Conclusion

The abstraction and delegation of deletion logic to a dedicated service has been shown to improve accuracy of deletion and reduce bugs [7]. This decreases the likelihood of a violation of article 17 of the GDPR [10]. delf-rs shows that the approach of DelF can be implemented agnostic of Facebook's infrastructure and in a manner that is agnostic to the technology stack of the web application.

References

[1] Diesel. http://diesel.rs/. Accessed: 2020-12-06.

[2] Gdpr enforcement tracker. https://www.enforcementtracker.com/. Accessed: 2020-12-08.

[3] Mysql. https://mysql.com/. Accessed: 2020-10-02.

[4] petgraph. https://github.com/petgraph/petgraph. Accessed: 2020-12-06.

[5] Rocket. https://rocket.rs/. Accessed: 2020-10-02.

[6] Yaml ain't markup language. https://www.yaml.org. Accessed: 2020-10-02.

[7] Katriel Cohn-Gordon, Georgios Damaskinos, Divino Neto, Joshi Cordova, Benoît Reitz, Benjamin Strahs, Daniel Obenshain, Paul Pearce, and Ioannis Papagiannis. {DELF}: Safeguarding deletion correctness in online social networks. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[8] Eddie Kohler. Hotcrp. https://github.com/kohler/hotcrp/. Accessed: 2020-12-06.

[9] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[10] Council of European Union. Regulation (eu) 2016/679 of the european parliament and of the council, 2016. https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016R0679.