Treating Microservices as a Unifed, Distributed Database

Brennan Saeta
Coursera Inc.
381 E. Evelyn Ave.
Mountain View, CA
saeta@coursera.org

ABSTRACT

Microservices have become a popular application architecture,[2] allowing multiple teams within an organization to efficiently build a complex system. Additionally, the rise of native mobile apps and interactive javascript web apps bring sophisticated business logic into the client. A principled approach–inspired by database architectures–enables frontend developers to treat the backend as if it were a database, resulting in efficient bandwidth use, and avoids "chatty" APIs, without sacrificing developer productivity. Naptime is an open source API framework that allows independent micro services to present a unified, queryable API to clients. Naptime operates on a graph-like subset of relational algebra that is amenable for efficient execution of distributed queries.

JSON-based APIs over HTTP provide an easily consumable cross-platform lingua-franca for both Javascript-based web applications, and native mobile applications. Additionally, microservices often interact with each other via HTTP and JSON as well. Unfortunately, developers implement business logic and interact with persistence systems typically in an ad-hoc fashion as per the application needs. An extreme approach is the Backend-for-Frontend pattern,[3] where presentation logic is split between the client and the server. Alternatively, Facebook has developed GraphQL[1] however that requires abandoning existing infrastructure to support JSON/HTTP APIs, and the microservice architecture–until now.

The data needs of interactive clients inevitably place relational-algebra-like demands upon the underlying API. Let us take a course catalog as a motivating example. An API to power this will need to support pagination to allow the user to browse the list of courses. Additionally, a user may search for a specific course, so this course API must support selection. The rich UI should display detailed information about the instructor for each course, and thus the API must support natural joins across related data types in a single query. Finally, only the necessary data should be transferred across a WAN or mobile network, and so the API must support projections. Coupled with additional concerns such as admission control to mitigate DoS and DDoS attacks, API frameworks should borrow from the rich experience of SQL database systems.

The Naptime data model centers around a resource. Read queries fall into 3 main categories: (1) multi-get, (2) get-all, and (3) finders. Responses from all three types of handlers invariably return a paginated list of elements; in short, a resource is a SQL table, and the handlers implement relational-algebra-like selections. The Naptime framework automatically adds field-projection capabilities to each handler. Finally, an engine sits on top of all the individual resources and can automatically link related resources into a single response based on the client query. The engine supports both "forward" joins, where the id of the related resource is a field in the root model, as well as "reverse" joins, where the related resource must be queried to determine the set of related elements. This model supports both one-to-one, one-to-many, and many-to-many relations, and is more flexible than a graph-based nodes-and-edges model.

Naptime has been in production for years, and powers 98% of all new APIs at Coursera. By providing a higher level of abstraction than arbitrary JSON and HTTP request/response pairs, developers are more productive. The Naptime toolkit implements high-level testing APIs, and turns imperative business logic into declarative code. Additionally, the toolkit leverages advanced Scala language features to catch bugs at compile time. The resulting unified, queryable API is much more flexible as applications evolve than a custom hand-implemented HTTP- or RPC-based API. We have even developed a loose bijection between Naptime's HTTP- and JSON-based APIs and Facebook's GraphQL, allowing us an easy incremental transition. As we learned in the evolution from SQL to NoSQL to NewSQL, the principles of database systems are inescapable.

© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

1. REFERENCES

[1] L. Byron. Graphql specification. http://facebook.github.io/graphql/, 2016.

[2] J. Lewis and M. Fowler. Microservices: a definition of this new architectural term. http://martinfowler.com/articles/microservices.html, 2014.

[3] S. Newman. Pattern: Backends for frontends. http://samnewman.io/patterns/architectural/bff/, 2015.