Byron - an Event-Driven Microservices Framework

authors:

João F. Daniel[1] Leonardo Lana[1]

advisors:

Prof. Dr. Alfredo Goldman[1] Prof. Dr. Eduardo Guerra[2] BsC. Renato Cordeiro[1]

[1] Instituto de Matemática e Estatística - Universidade de São Paulo [2] Instituto Nacional de Pesquisas Espaciais

Introduction

In the past twenty year, the Web became ubiquitous to the point where availability and reliability are essencial to online systems. This demand pushed systems to become more complex: there are a lot of aspects that increase the chance of failure and there are a handful of practices that make them highly coupled, when a failure in one part causes failures all across the system.

The aim of this project is to create a solution for developing microservices that values decoupling, an architecture that provides decoupling and data consistency. To guide this project, three research questions were defined:

Q1 Is it possible to propose an architecture that provides data consistency within a system component, even if it's eventual consistency?

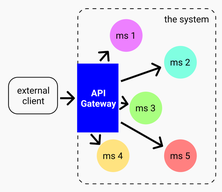
Q2 Assuming the proposed architecture and considering more than one component, does it sustain the eventual consistency across all components?

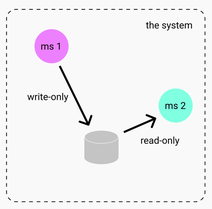
Q3 Does the proposed architecture provide decoupling between components, so that in case of one failing, the others doesn't fail too?

To reach this objective, it was built Byron – an event-driven microservices framework. It is based on three major principles: reactivity addresses the demand issue, stressing qualities about response times and internal communication standards; Cloud-Native focus on guiding the development of an application that is conceived to run with an automated, fully managed infrastructure; and Fast-Data models data handling as soon as it arrives.

Background

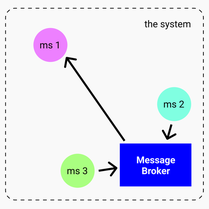
- adopting API Gateway, the system itself routes the requests to the correct microservices, unifying the address to the external client

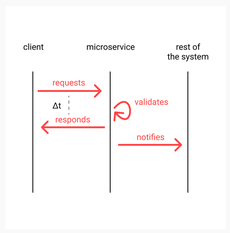




- adopting CQRS, on which is possible to better fit the requirements of read and writes that might be different, by scaling independently ms 1 or ms 2

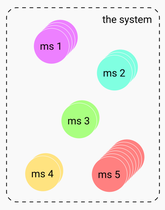
- adopting Event Sourcing, on which ms2 and ms3 don't have to wait for ms1 to process the request.

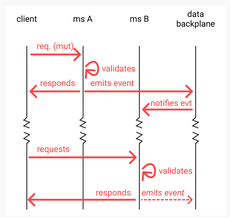




- on the right, a small Δt, since the response happens as soon as possible, hence higher responsiveness.

- following microservices architecture, on which its parts can be scaled up individually according to their needs.





- message-driven paradigm, on which, although the system is based on data replication, its microservices don't have a direct dependency between each other.

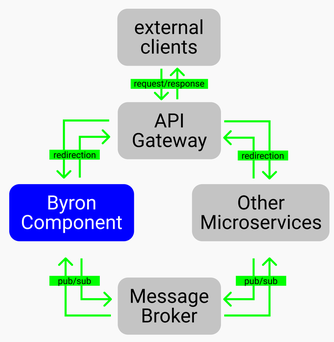
What is Byron?

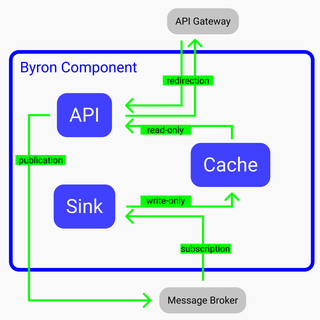
Byron is an Event-Driven Microservices Framework. It is a framework to develop systems following Microservices Architecture that adopts events as its core abstraction in the communication model, in order to provide decoupling between parts. It's a TypeScript framework that generates GraphQL APIs, uses MongoDB as the Cache with Mongoose as Object-Document Mapping and adopts NATS Streaming as Message Broker.

Architecture

Byron's architecture is complex and it's better explained following the C4 Model (https://c4model.com) for visualizing software architecture. This model adopts an "abstraction-first" approach, similarly to what Google Maps does: it's possible to zoom in or zoom out, depending on the interest – more detail or more context, respectively.

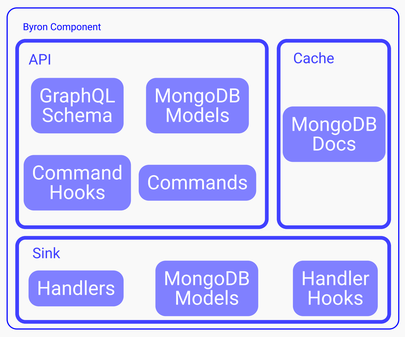
A **Component** is the highest abstraction defined, it's supposed to implement a subset of the system's business logic. It's deployed within a back-end of a web server, being target of a request from an external client. An API Gateway can be used to abstract externally the inner division. Whenever there's an event, the Component interacts with the broker, via a pub/sub protocol.





A Component a set of three microservices: **API**, **Sink** and local **Cache**. When a request comes in, the API is responsible for extracting information from it and run a command. If it needs stored data, it reads the Cache. If the request causes any change in the application state, the API is also responsible for emitting an event notifying the rest of the change. Any Sink interested in that event then reads it and updates the Cache with the payload data.

Going deeper into details, each of the component's microservices holds its own set of files. The API holds all commands and all command hooks. The Sink holds all handlers and all handler hooks. As the API and the Sink access the Cache, both hold a copy of Mongoose Models.



When to use

To discuss the situations when the adoption of Byron is beneficial and when it's not, this section presents three different cases.

Social Media - beneficial

Suppose there's a social media with a lot of users across the world, with a great amount of development teams and a huge demand for content all over the days. This social media supports some features, such as creating a profile and an institutional page, posting to a news feed, linking people and interests but does not have a chat system. This chat is to be developed and for that they intend to adopt Byron.

- the development staff is big, with many teams, which benefits from autonomy provided by microservices architecture;
- the system is used all across the world with a great demand, also benefiting from system scalability;
- the feature of chat tolerates eventual data consistency and, thus, asynchronous messages.

Bank - non-beneficial

Suppose there's a bank that is re-building its monolith system now adopting microservices architecture. It's a big bank supporting uncountable online transactions per day, with a large development staff. The core of the system deals with money, transactions and security. This core is to be rebuilt and for that they intend to use Byron.

- the development staff is big, with many teams, so it benefit from the autonomy for development teams provided by microservices architecture;
- the core business **does not** tolerate eventual data consistency – in that case, a better approach would be to adopt a synchronous communication model between the core microservices.

Startup on validation stage - non-beneficial

Suppose there's a small startup building its validation system. The development staff only has a five or six developers and the idea is not yet validated in the market. This system is to be first built and for that they intend to use Byron.

- the development staff is small, so it doesn't demand a lot of autonomy – and it's better not to use microservices, due to the complexity added to system;
- the system doesn't have a big demand yet, so system scalability is not an issue either.

Conclusion and Future Work

In order to verify that the framework provided decoupling and data consistency, a study case was conducted. Three scenarios were designed out of a real application, proving that Byron's architecture data consistency is eventual – an implication of the asynchronous message-driven communication model –, either Component-wise and across components, and also verifying that components are decoupled – so there's no domino effect in case of one's failure.

The framework would benefit from extensions, such as:

- adapters/drivers for other databases to use instead of MongoDB in the local cache;
- Kubernetes configuration file generation, following the same approach of Docker's;
- pluggable objects to generate the API in other standards, such as REST in replacement of GraphQL.

References

- Jonas Bonér et al. **The Reactive Manifesto**. Tech. rep. 2014.
- Justin Garrison and Kris Nova. **Cloud Native Infrastructure - Patterns for Scalable Infrastructure and Applications in a Dynamic Environment**. 2018.
- Sam Newman. **Building Microservices - Design Fine-Grained Systems**. 2015.
- Mark Richards. **Software Architecture Patterns - Understanding Common Architecture Patterns and When to Use them,** 2015.
- Dean Wampler. **Fast Data Architectures for Streaming Applications - Getting Answers Now from Data Sets That Never End**. 2019.