Byron, an Event-Driven Microservices Framework

João F. L. Daniel[1], **Leonardo L. V. Oliveira**[1], **Renato C. Ferreira**[1], **Eduardo M. Guerra**[2], **Thatiane O. Rosa**[1,3], **Alfredo G. vel Lejbman**[1]

[1]Institute of Mathematics and Statistics – University of São Paulo (USP)
São Paulo – SP – Brazil

[2]Facoltà de Scienze e Tecnologie informatiche – Libera Università di Bolzano
Bolzano – Italia

[3]Federal Institute of Tocantins
Paraíso de Tocantins – TO – Brazil

{joaofran,renatocf,gold,thatiane}@ime.usp.br, {leolanavo,guerraem}@gmail.com

Abstract. *The rise of technological dependency made some requirements crucial to online systems, such as availability, and scalability. The microservices architectural style provides improvements to scalability and software maintainability and has been broadly adopted. Although, microservices highlight trade-offs between consistency and coupling level. This work presents Byron, an event-driven microservices framework as a solution to mitigate these problems. It implements a reactive architecture in an Event-Sourcing environment.*

1. Introduction

The technological dependency, on behalf of people and companies, has risen lately – online systems are constantly and massively accessed. For that, availability, reliability, and scalability are features that become crucial for online systems [Wampler 2019].

To achieve such features, development teams have been adopting the microservices architectural style. This style provides improvements to scalability, in terms of development teams, and software deployment [Garrison and Nova 2018]. Furthermore, it enhances continuous delivery, which improves the system evolution process.

A microservices-based system becomes more complex than a monolithic one [Bonér 2017], because it highlights the trade-offs between data consistency and level of coupling among parts, similar to Brewer's Theorem [Sadalage and Fowler 2013]. To reach decoupling, implementing microservices is not enough, it is necessary to adopt an asynchronous communication model [Bonér 2017].

Thus, considering modern context requirements and the difficulties microservices architecture offers, we created a framework to assist the process to build software. We designed Byron as an event-driven microservices framework to help the implementation of the asynchronous communication model – which provides decoupling and eventual data consistency – and also to mitigate the time-consuming task of integrating other tools, such as Docker, NATS-Streaming, and MongoDB.

2. What is Byron?

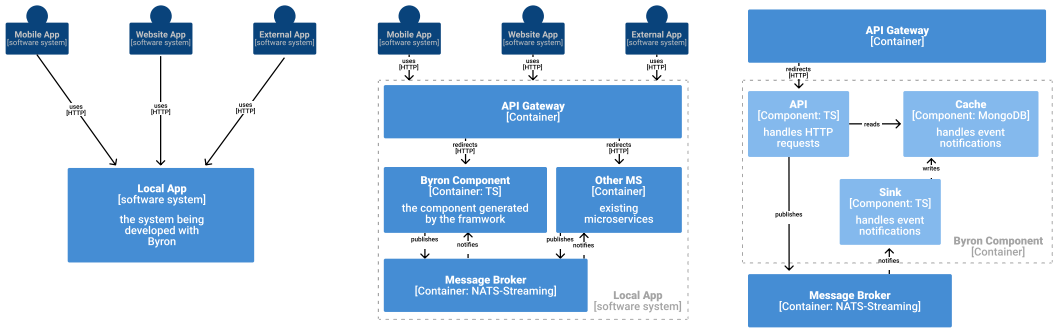
Byron is an event-driven microservices framework, i.e., it adopts the event as the core abstraction in the communication model. It is written in TypeScript, generates GraphQL

APIs, uses MongoDB as local cache and Mongoose as Object-Document Mapping, and adopts NATS-Streaming as the message broker. Byron is an open-source project and it can be accessed in `https://gitlab.com/byron-framework/cli`.

2.1. Architecture

The selected way to explain Byron's architecture is by following the C4 Model [Brown 2011]. It has an approach that values abstractions on which it is possible to zoom-in or zoom-out accordingly to the interest: more details or more context, respectively.

In this section, we present Byron's architecture with narratives, beginning with the description of the Context, the full zoomed-out perspective; it is followed by zooming into the Containers; until reaching the full zoomed-in perspective, the Components.



(a) (b) (c)

Figure 1. (a) Context, (b) Containers, and (c) Components

Context: Byron is a framework to be used in general-purpose web systems – it has no specialization in other tasks, such as machine learning, for instance.

Another characteristic is that Byron requires the whole Context system to implement the Event-Sourcing pattern [Richardson 2018]. That is, the microservice developed with the framework must be placed within a system that adopts a message broker persisting the state of the application as a sequence of changes, marked by events. Such a broker must offer an API to subscribe to updates and also to publish events. Fig. 1(a) presents this idea.

Containers: We call Byron Component the main object built by the framework – despite the ambiguity, it is not a C4-Component, it is a C4-Container. It is meant to implement a coherent set of well-defined operations of the domain, i.e., a Bounded Context [Evans 2003].

A Byron Component adopts, for the external communication interface, a model different than the adopted for internal communication. The Component exposes a GraphQL API, a strongly-typed alternative to REST. As the internal communication, it exchanges Domain Events [Richardson 2018] with the broker, publishing and listening updates. Fig. 1(b) illustrates this level of abstraction.

Components: As shown in Fig. 1(c), a Byron Component is divided into three microservices. First, the **API** is responsible for exposing a GraphQL API and for handling HTTP requests. When it needs data, the API queries the **Cache**, the second microservice

of a Byron Component. It is an in-memory database storing the most recent and relevant data came from the broker. This storage provides decoupling between components because, in case of failure elsewhere, a Byron Component has its reliable source of data, even if they are slightly outdated – because with that, any request can be answered with some data. The third is the **Sink**, responsible for listening to events of interest in the broker and then updating the Cache accordingly.

This set of microservices implements a pattern called Persistent Storage [Richardson 2018]: it extracts the Byron Component state into a separated process. This practice provides data consistency among copies of the same microservice – in case of scaling up. Furthermore, the segregation between API and Sink around the Cache implements CQRS [Richardson 2018], providing better scalability to them.

2.2. Implementation with Byron

Byron works in a declarative way. It uses a Domain Specific Language to centralize all declarations into a single file, called `schema.yaml`. It extends GraphQL definitions for the API – with its own **types**, **inputs**, and **commands** – to wrap also all event-handling functions – called **handlers** – and functions related to lifecycle – called **hooks**.

The online documentation, hosted at `https://byron.netlify.com`, presents further information about Byron technical details.

3. The use of Byron and similar tools

Byron is a tool to implement general-purpose business logic. Its architecture supports contexts often related to layers that are closer to external clients.

The adoption of microservices is a measure mostly driven by team scalability. This framework is a tool to implement, for instance, a new service within an international e-commerce platform that already adopts microservices, because the company has several development teams. A counter-example is a new start-up building a functional prototype to validate its business model. In that case, as it is still a small company, the adoption of microservices becomes overwhelming.

Event-Sourcing inspired Byron's architecture, which is event-driven. Hence, it offers eventual data consistency. That makes the adoption of this framework more suitable in an eventually consistent tolerant environment – for instance, a chatting system.

There are other tools available that work similar to Byron in terms of language, approach, communication model and architectural decisions. Table 1 compares Byron with NestJS (`https://nestjs.com/`) and Moleculer (`https://moleculer.services/`).

Different from Byron, NestJS is not a microservices-exclusive framework. It offers two modes: a monolithic and a microservices-oriented. When it comes to the latter, it does not offer a defined architecture, it considers a microservice an application that simply does not uses HTTP requests. Along a software evolution, when it follows a defined architecture, the project benefits from having ground base to decision making and risk/cost analysis. Since Byron defines an architecture, it stands out.

Moleculer has a much more fine-grained approach to microservices, in which the division happens in a function-level. This approach is more similar to the server-

		Byron	NestJS	Molecular
Language		TypeScript	TypeScript	Node.js
Approach		declarative	imperative	imperative
Comm. Model	*async*	yes	yes	yes
	orientation	events	events, messages	messages
	protocol	pub-sub	pub-sub, req-res	req-res
Architectural Decisions		defined architecture	no architecture	no architecture

Table 1. Comparison between Byron, NestJS, and Molecular

less paradigm, which injects function objects into machines already with servers running, as this layer of code was part of the infrastructure itself.

4. Conclusion and Future Work

In this work, we presented Byron, an event-driven microservices framework. It is a framework to develop general-purpose business logic adopting the microservices architectural style, and it works as a solution to provide data consistency and decoupling between parts. Its successful results are proof-of-concept.

Byron is under constant development seeking improvements and can be enhanced with some future works. We thought of the following:

- experimenting with a system implemented with Byron running under critical scenarios,
- creating extensions to integrate other cloud-computing platforms,
- providing more flexibility of its architecture to create a wider diversity of Byron Components, and
- experimenting with the developer experience with the framework to better understand the value provided by the well-defined architecture.

References

Bonér, J. (2017). *Reactive Microsystems - The Evolution of Microservices at Scale*. O'Reilly, 1st edition edition.

Brown, S. (2011). The C4 model for visualising software architecture.

Evans, E. (2003). Domain-Driven Design Tackling Complexity in the Heart of Software. Technical report.

Garrison, J. and Nova, K. (2018). *Cloud Native Infrastructure - Patterns for Scalable Infrastructure and Applications in a Dynamic Environment*. O'Reilly, 1st edition edition.

Richardson, C. (2018). *Microservices Patterns*. Manning, 1st edition edition.

Sadalage, P. J. and Fowler, M. (2013). *NoSQL distilled - a brief guide to the emerging world of polyglot persistence*. Addison-Wesley, 1st edition edition.

Wampler, D. (2019). *Fast Data Architectures for Streaming Applications - Getting Answers Now from Data Sets That Never End*. O'Reilly, 2nd edition edition.