Byron - an Event-Driven Microservices Framework

João Francisco Lino Daniel

Leonardo Lana Violin Oliveira

Monograph submitted
to the
Instituto de Matemática e Estatística
of the
Universidade de São Paulo
for the
Course Final Monograph
of the
Bachelor's in Computer Science

Course:
Computer Science

Advisor:
Prof. Dr. Alfredo Goldman

Co-advisor:
Prof. Dr. Eduardo Guerra

BsC. Renato Cordeiro Ferreira

Contents

Chapter 1

Introduction

In the past twenty years, the Web became ubiquitous to the point where availability and reliability are essencial to online systems [Wam19]. Recent studies focus on creating and improving of a series of techniques, principles and architectures to keep up with these demands – studying fine-grained distributed systems, data synchronization techniques, automation of infrastructure, to name a few topics. This demand pushed systems to become more complex. There are a lot of aspects that increase the chances of failure and there are a handful of practices that make systems highly coupled, when those failures trigger a domino effect – 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 is 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 do not fail too?

To reach this objective, Byron – an event-driven microservices framework – was built. 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.

Byron defines a Domain Specific Language (DSL) to represent the application's entities, their relationships and actions, allowing developers to focus on the business

domain. It also offers a declarative interface for dealing with events. Byron provides a Command Line Interface (CLI) that works as a tool to scaffold a working directory and to set up the app in a development environment, by bootstrapping basic resources and providing means to update whenever wanted.

To measure that Byron fulfills its goals, a set of scenarios of a real application implemented with Byron is presented. Each scenario lists external interactions with the app and automated tests that implement them. In the end, each one exposes some features that helps answering the research questions, strengthening the hypothesis that Byron works as a solution for decoupling microservices.

Development community provides a reasonable amount of tools that are used in the development process of microservices, from distributed solutions for databases to whole web frameworks. Byron stands out for being a cross-cutting solution that implements a service-level architecture: it has a message broker for event-sourcing fully integrated with microservices that listen to events and update its local cache; it makes declarative and clear the task of emitting events.

The development process of Byron was divided into two parts: an initial version developed without tests, and a final version developed using Test-Driven Development. Byron's first version was set aside after an experiment with real developers, after collecting a positive feedback, yet with some issues, of the impact of the framework. That's because, as it had no tests, it wasn't a good call to keep developing something that became untestable. The second version let the architecture emerge from tests and the result was a more cohesive software.

There are two other features of interest in Byron that differentiate it: first, as the architecture emerged from the tests, and more importantly, it emerged whilst the code was being developed, Byron has an architecture built to be extensible, so future contributions from the community can improve its forces; second, its clearer code, in comparison with the first version, makes the framework more maintainable, which suggests a more stable future of development.

Chapter 2 presents the conceptual background upon which this project was settled. **Chapter 3** introduces the Byron framework with its terminology, features and some contexts when the adopting of it is beneficial. **Chapter 4** presents the study case and its analysis. **Chapter 5** presents comparisons between Byron and other frameworks, namely *Ruby on Rails* and *NestJS*. Finally, **chapter 6** discusses the results and future work related to the project.

Chapter 2

Background

This chapter presents the theory supporting this project development, from the basis of architectural and infrastructural principles, to the articulation of those concepts and the created abstractions.

2.1 Microservices

The **Microservices Architecture** is an architectural model on which the system gets divided into a set of independent parts, each one with its own well-defined responsibility – each part is called a *microservice*. This implies that the system is composed by multiple code bases and, during runtime, it is the composition of multiple processes. It raised in popularity because it provides two ways of scalability: in team sizes, due to the division of code bases; and in volume of work to be executed, due to the multiple binaries in runtime [New15]. This independence between microservices enforces independent deployment, allowing different development teams to move at their own speed. It also enables each microservice to be scaled according to its own workload and demand [Ric15].

Although this architecture has become very popular, it is not the best solution for all systems. The Microservices Architecture focus on scalability. Whenever that is not an issue to the development or to the application, then it's better not to adopt this architecture [New15]. That is because microservices offer a series of difficulties to be overcome, since it creates a *distributed system*, and that can overpower the development team.

When this architecture applies, its difficulties can be dealt with some patterns, as follows:

- **API Gateway** provides to their clients an abstraction of the system's distributed APIs and its addresses, making the microservices system to externally look like

it's a monolith[1]; Figure 2.1 compares two systems – with and without API Gateway –, where *ms X* stands for *microservice X* and the arrows are representations of requests;

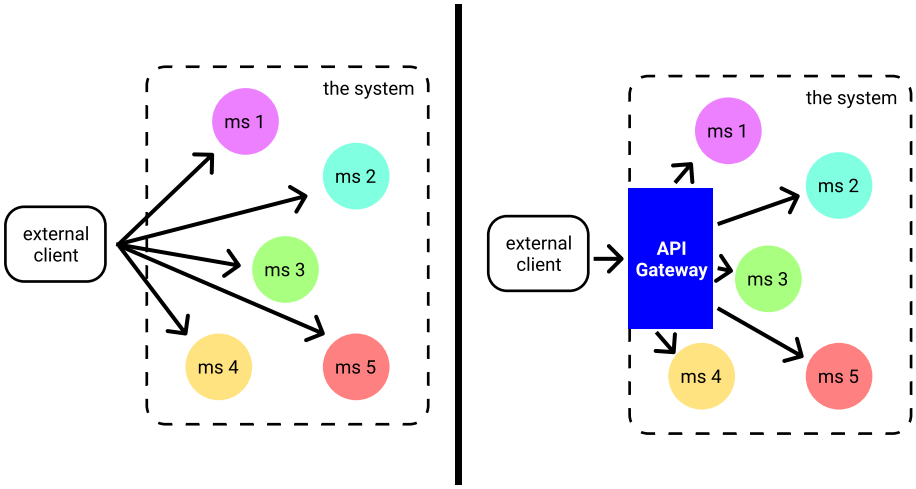

Figure 2.1: Comparison between two microservices systems: on the left, without API Gateway, the external client must address directly each microservice; on the right, adopting API Gateway, the system itself routes the requests to the correct microservices, centralizing the address to the external client

- **Command-Query Responsibility Segregation (CQRS)** decouples the reading and writing model of services, thus optimizing each operation according to the use case and external demand; Figure 2.2 compares two systems – with and without CQRS –, where *ms X* stands for *microservice X* and the gray object represents a database;

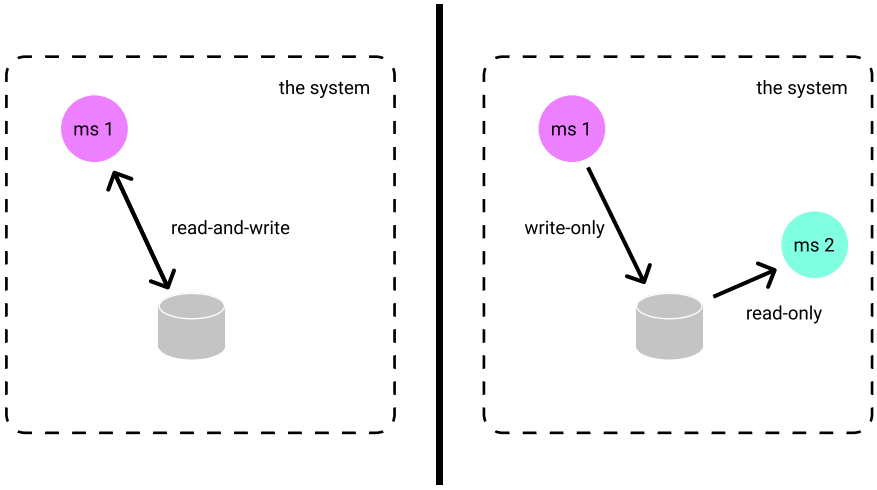

Figure 2.2: Comparison between two approaches: on the left, without CQRS, on which scaling *ms 1* scales at the same time read and write potential; on the right, adopting CQRS, on which is possible to better fit the requirements of read and writes that might be different, by scaling independently *ms 1* or *ms 2*.

[1]For further reading, search for *Monolith* and *Layered Architecture* at [Ric15]

- **Event Sourcing** enables microservices to take care of their own state in isolation, whereas notifying changes to other components via message-passing; Figure 2.3 compares two systems – with and without Event Sourcing –, where *ms X* stands for *microservice X*, two-headed arrows are synchronous requests and single-headed arrows are message passing;

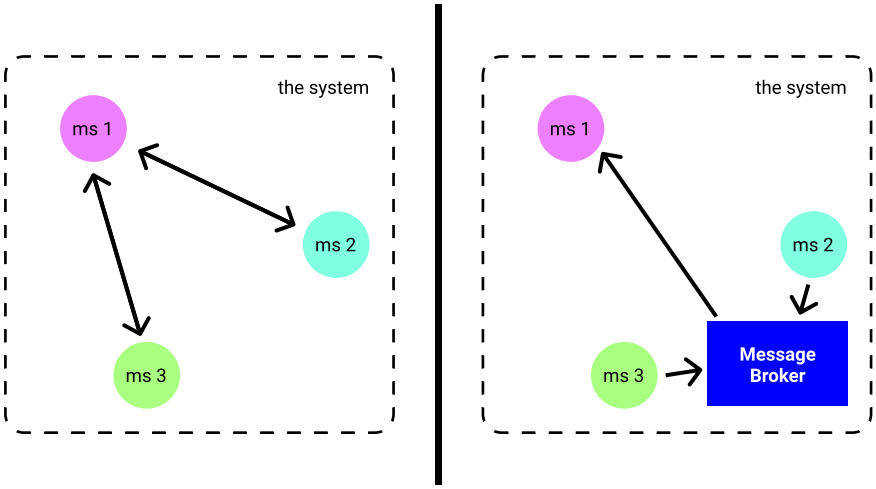

Figure 2.3: Comparison between two approaches: on the left, without Event Sourcing, on which microservices communicate synchronously one with another; on the right, adopting Event Sourcing, on which *ms2* and *ms3* don't have to wait for *ms1* to process the request.

The idea of an **event** is a significant transition in state – an event doesn't *exist*, it simply *occurs*. A **message** is the implementation of an event, an object used to notify that the event happened – it may have a payload within, where data is stored. The **Message Broker** is the composition of a Message Queue (MQ) used to broadcast the messages, and an Event Log model, i.e., what is stored is the sequence of changes over the time – differently from a database that is often used to store the final state only.

Reactive Systems is the set of principles that makes systems more robust, more resilient and more flexible [Bon+14], following four characteristics: **responsiveness** sets focus on providing consistent and reasonable response times, so that it enhances reliability and simplifies error handling by an external client; **resiliency** keeps the system responsive in face of a failure; **elasticity** keeps the system responsive in various workloads – without bottlenecks; and **message-driven** provides loose coupling by exchanging asynchronous messages through the boundaries of the system.

Responsiveness can be implemented by returning an answer to the client as soon as the request handler has it, i.e., either if the request fails or if it succeeds, it immediately responds accordingly. The emission of an event, if it is due, happens almost simultaneously to the response. Figure 2.4 compares two sequence diagrams – with higher and with lower responsiveness –, where arrows represent actions and the sooner they happen, the higher they are in the diagram.

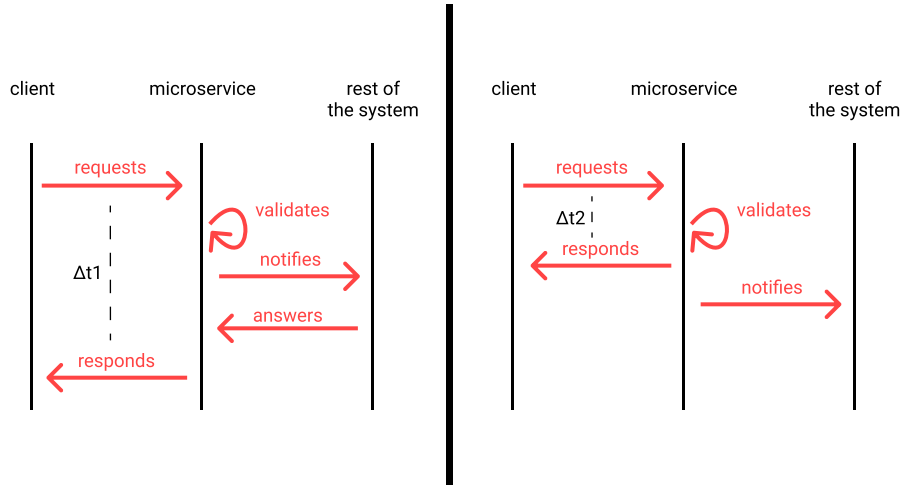

Figure 2.4: Comparison of two systems: on the left, with larger $\Delta t(\Delta t_1 > \Delta t_2)$, since the response happens after the rest of the system gets notified, hence lower responsiveness; on the right, with smaller $\Delta t(\Delta t_2 < \Delta t_1)$, since the response happens as soon as possible, hence higher responsiveness.

Resiliency can be enhanced by implementing data replication and event sourcing, to provide to a microservice all the data it needs to create a response to any request made to it. Not all data a microservice needs is created within its domain of actions, but it can receive the data via messages, whenever the data is created elsewhere. Then, adopting a local cache database for each microservice, thus creating data replication, provides a degree of decoupling that whenever a request is made, the responsible microservice does not depend on any other to respond. This implementation improves resiliency by reducing the chances of a domino effect – when the failure of a microservice implies on the failure of others, and so on.

Elasticity is a feature achievable via implementing scalable microservices. When the system is a monolith, it tends to have bottlenecks and its scalability gets compromised. With microservices, it is possible to have scalability on demand, as Figure 2.5 shows. It compares two systems – a monolith and one following microservices architecture –, where each gray box is a copy of the whole monolithic system, *ms X* stands for *microservice X* and the number of stacked copies of each microservice can be different.

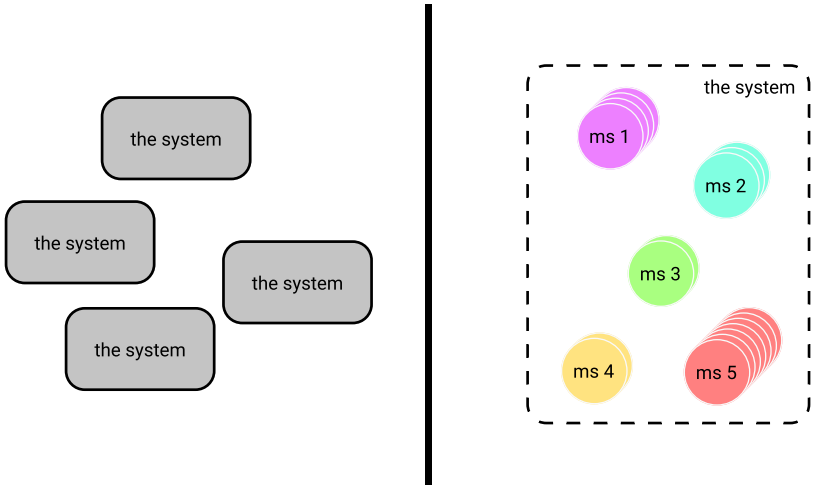

Figure 2.5: Comparison of two systems: on the left, a monolith, represented on a scaled scenario, on which there are many copies of the entire system; on the right, one following microservices architecture, on which its parts can be scaled up individually according to their needs.

Adopting message-driven is a paradigm change, in sense that communication becomes based on reaction, other than being active as request-oriented. In a message-driven system, when there's a part that depends on data from others, it listens to events around that data, so that when it needs to use that data, it already has. Figure 2.6 shows two sequence diagrams representing this paradigm change – the blue dashed box focuses on the complexity of the microservice B that depends on external data.

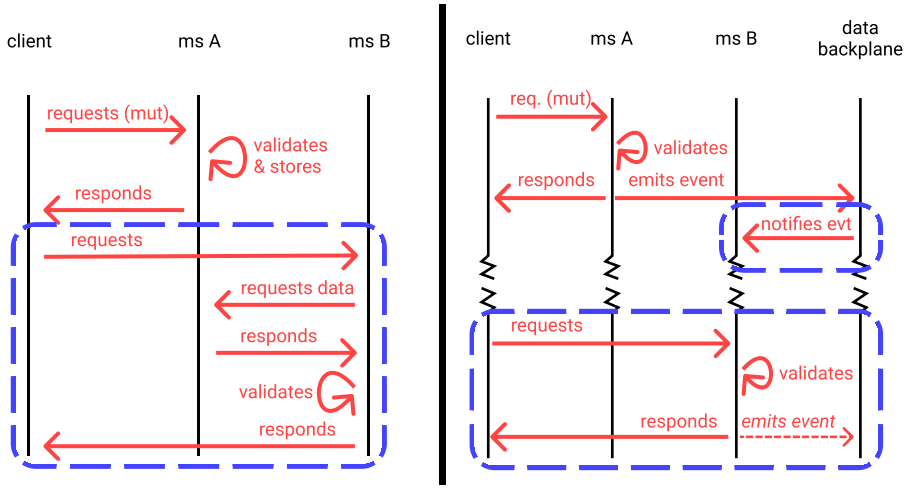

Figure 2.6: Comparison between two systems: on the left, request-based paradigm, on which *ms B* doesn't have all data it needs and requests it to *ms A*; on the right, message-driven paradigm, on which, although the system is based on data replication, its microservices don't have a direct dependency between each other.

Message-driven is a feature that fits well with Reactive Systems and with another concept: **Fast Data Architecture (FDA)**. Treating messages as they come in is based on the policy of stream-orientation, and this configures a **Fast Data System**.

It's interesting to note that, following this approach, batch-orientation is a sub-case where streams are finite. FDA also relies on a strong, resilient, stream-oriented data back-plane: log systems and message queues – a system can implement it with a message broker.

2.2 Infrastructure

Virtualization is the technique of emulating in software the hardware, which overcomes the difficulties to configure and maintain, and avoids the waste of resources caused by running physical machines [GN18]. The use of **Virtual Machines (VMs)** makes possible to isolate two applications running on the same physical machine, which provides improvements in the usage of resources of the physical machine, and in the application's portability [GN18].

When a VM is launched, the operating system of the physical machine – called *host O.S.* – emulates over itself an entire operating system for the VM – called *guest O.S.*. This adds an overhead compared to running the application directly on the host OS. **Containerization** is a technique that encapsulates code, dependencies, networking and file-system into a single process. There's a resemblance with VMs, but the major difference is that a container is an isolated process created directly on the host OS – it solves VM's overhead of stacking guest OS. **Docker** is a container engine based on Linux namespaces for providing isolation.

Cloud-Native Infrastructure (CNI) is an infrastructure that is hidden behind useful abstractions, controlled by APIs, managed by software and has the purpose of running applications [GN18]. These features determine a scalable and efficient pattern to manage infrastructure.

The adoption of useful abstractions enables CNI to manage the underlying IaaS – Infrastructure as a Service, refer to Figure 2.7 – by creating a new layer of objects above it. It then exposes an API to be consumed avoiding re-implementation. Management by software is a core feature of CNI, since it enhances scalability, resiliency, provisioning and maintainability of the infrastructure [GN18]. That is because the management software enters in between the CNI layer and the underlying IaaS, mapping the former's abstractions into latter's entities.

The 12-Factor App (12-Factor) – a methodology developed by the Heroku platform team, inspired by their experience and Martin Fowler's books Patterns of Enterprise Application Architecture and Refactoring – lists a dozen of practices developers should follow in order to create apps that better fit the reality of PaaS – Platform as a Service, refer to Figure 2.7.

Figure 2.7 presents a table comparing three cloud models: *on-premises*, IaaS and

PaaS.

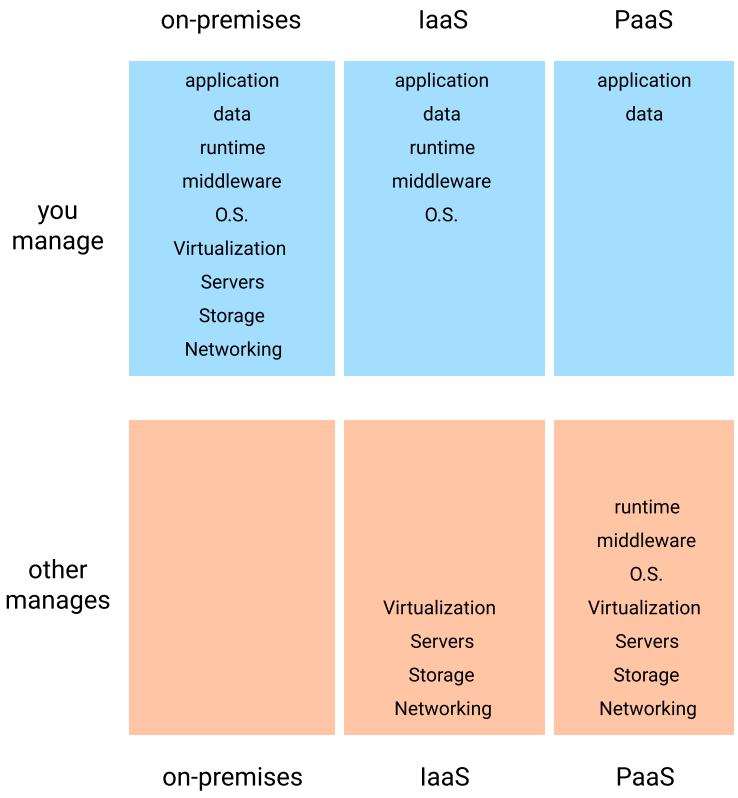

Figure 2.7: Cloud models: a comparison between on-premises, IaaS and PaaS. On the blue zone, assets the developer must manage; on the beige zone, assets the provider manages.

Changes caused by CNI into the relationship between business and infrastructure not only impact how the application runs, but also take it a step further into what 12-Factor proposed, consolidating the **Cloud-Native (CN)** principle. Applications should have four characteristics: **resiliency** embraces failure instead of trying to prevent them; **agility** allows fast deployments and quick iterations; **operability** adds the control of the lifecycle to the application instead of relying on external processes and monitors; and **observability** provides information to answer questions about the app state [GN18].

Resiliency, in the interpretation of CN, can be implemented in three main ways: *design for failure*, ergo each microservice is built to embrace failure and is prepared for a fast reboot; *graceful degradation*, which is implemented by each microservice storing in a local cache some information, even if it might be a little outdated – it's better to give some answer than none; and *declarative communication* due to all communication across the system happening over the network, when the specification of the requirements happens by defining *what* is wanted, other than *how*.

Agility and observability are two factors that come from known features: the adop-

tion of microservices as the core architectural decision implies CN agility – each microservice can be developed, delivered and scaled independently from each other; making events the main abstraction for communication and adopting a message broker as backbone for fast data architecture enhances observability, as every change in state gets registered in the logs of the broker.

Chapter 3

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. The code of the framework is open-source, hosted in a GitLab repository and can be accessed at https://gitlab.com/byron-framework/cli. There's also an online documentation, hosted at https://byron.netlify.com/.

3.1 Architecture

Byron's architecture is complex and it's better explained following the C4 Model 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.

The first concept to understand is a **Component**: it's the highest abstraction the framework defines, it's supposed to implement a subset of the system's business logic. When deployed, it's placed within a back-end of a web server, so it can be the target of a request coming from an external client, such as a mobile app or another back-end. An API-Gateway can be used to abstract externally every detail of inner division, and its role is to identify the respective API and to redirect the request. Whenever there's an event, the Component interacts with the broker, via a publication/subscription protocol. Figure 3.1 illustrates this context.

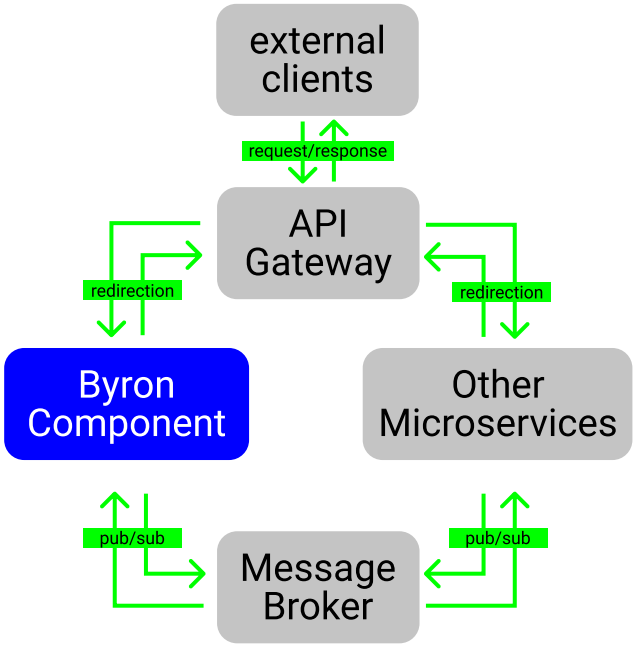

Figure 3.1: Representation of the context in which a Byron Component (in blue) is placed; arrows (in green) represent interactions with other elements (in gray)

A Component is internally a set of three microservices: an **API**, a **Sink** and a local **Cache**. When a request comes from the Gateway – or straight from the client, when there's no gateway – , the API is responsible for extracting information from the request and run a command. If it requires stored data, the API reads the Cache, does some validation and then sends a response back. If the request causes any change in the application state, then 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. Figure 3.2 represents this level of detail.

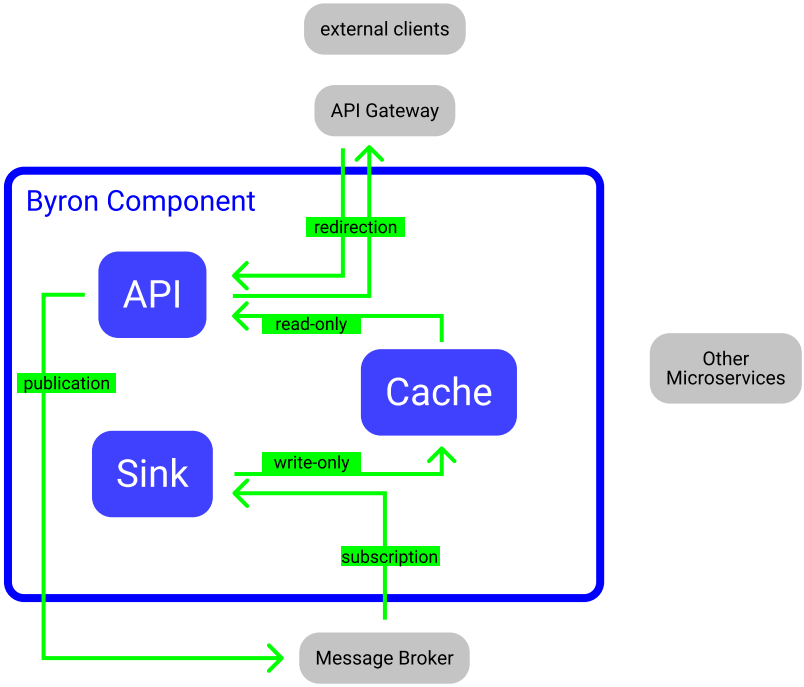

Figure 3.2: Representation of a Byron Component (blue border) internal microservices: API, Sink and Cache (all in blue filling)

The relationship between API and Sink around the Cache implements Command-Query Responsibility Segregation. The former has a *read-only* access to the Cache, while the latter, a *write-only* access. It's interesting to mention that this behavior allows not only independent scale to each kind of access, but also improves responsiveness as the API is able to respond the request without having to update the state of the Cache.

Going deeper into details, Figure 3.3 represents what elements each microservice of the Component holds. The API, since it implements a GraphQL API, has a schema – the DSL inspired by GraphQL[1] – and all commands and command hooks provided by the developer. To query the Cache, the API also has all mongoose models. The Cache stores all MongoDB Documents. The Sink has the same mongoose models as the API, once it refers to the same database. It also contains all handlers and handler hooks provided by the developer.

Going deeper into details, each of the component's microservices holds its own set of files. The API holds all **commands** – all actions that the API offers to the exterior – and all command **hooks** – functions that are called in a given moment, either before or after a command. The Sink holds all **handlers** – functions that implement reactions to

[1]For further information, refer to GraphQL Docs

events in the broker – and all handler hooks – which works similarly to command hooks, but either before or after handlers. As the API and the Sink access the Cache, both hold a copy of Mongoose Models – an object that models the structure of documents in the MongoDB database and implements access methods. Figure 3.3 represents this level of detail.

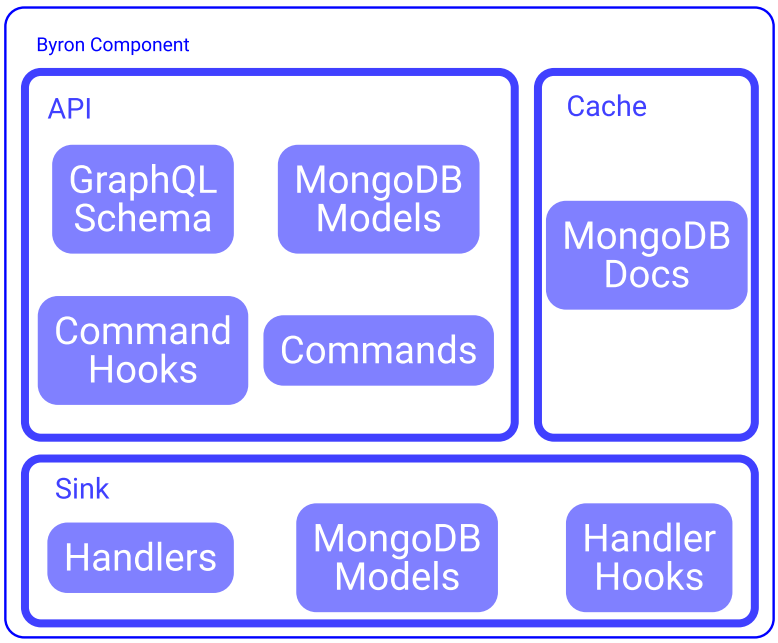

Figure 3.3: representation of what the API, the Sink and the Cache hold inside with more details

3.2 *Hello, Byron!*

This section presents a simple case for using Byron, in order to make things concrete: Calisto, a system to manage undergrad, with a component Students, on which each student have a name, a course and ingress year.

To begin the development, Byron provides a CLI with the command `byron component init NAME`, where `NAME` is an argument to be passed with the desired name of the component. It scaffolds a working directory structured as follows:

Listing 1 the directory structure Byron uses

```
NAME/
 |-- api/
 |     |-- commands/
 |-- sink/
 |     |-- handlers/
 |-- hooks/
 |-- schema.yaml
```

Byron works in a declarative way with the use of a Domain Specific Language (DSL), centering all declarations in a `schema.yaml` file. For this toy example, the Student component deals with the type `Student`, that have name, course and ingress year. The component has the commands `getStudents` – that returns a list of `Students` – and `createStudent` – that create a new student entry and returns it. And in order to update the Cache, the Sink has the handler `newStudent` – that listens to `new.student` event and updates the Cache. Listing 2 presents the `schema.yaml`:

Listing 2 a `schema.yaml`, an example containing **types**, **commands** and **handlers**, a basic running application

```yaml
name: "Student"
namespace: "Calisto"

content:
  types:
    - name: "Student"
      attributes:
        - name: "name"
          type: "String!"
        - name: "course"
          type: "String!"
        - name: "ingress"
          type: "Int!"
  commands:
    - name: "getStudents"
      type: "query"
      returnType: "[Student!]!"
    - name: "createStudent"
      type: "mutation"
      returnType: "Student!"
      parameters:
      - name: "name"
        type: "String!"
      - name: "course"
        type: "String!"
      - name: "ingress"
        type: "Int!"
  handlers:
    - name: "newStudent"
      event: "new.student"
```

From the `type` named `Student`, Byron will create:

- the GraphQL Student type, and
- the Mongoose Student model.

Given that there's a `command` named `getStudent`, Byron will search for a func-

tion file `api/commands/getStudents.ts` in the working directory, with the structure as follows in listing 3:

Listing 3 a query command function file *api/commands/getStudents.ts* that returns all students stored – the parameters are inherted from GraphQL standards[2]

```
async function getStudents(
  parent: any,
  arguments: any,
  context: any,
  info: any
): Promise<any> {
  // 'context' contains useful objects, such as DB models
  const { db: { Student } }: any = ctx;

  return await Student.find();
}

// the convention is to export as 'default'
export default getStudents;
```

Similarly, there's `createStudent`, a mutation command placed in *api/commands/createStudent.ts* with the structure as follows in listing 4:

Listing 4 a mutation command function file *api/commands/createStudent.ts* that expect `name, course, ingress` as arguments, guarantees uniqueness of name and emits an event in case of all validations are ok.

```
1  async function createStudent(
2    parent: any,
3    args: any,
4    ctx: any,
5    info: any
6  ): Promise<any> {
7    const { db: { Student }, Error, uuid }: any = ctx;
8
9    // destruct 'args' into variables
10   const { name, course, ingress }: any = args;
11
12   // consider 'name' as unique
13   let students: any = await Student.find({ name });
14   if (students.length !== 0) {
15     const msg: string =
16       `Student with name [${name}] already exists`;
17
18     throw new Error(msg, '409');
19   }
20
21   student = { _id: uuid(), name, course, ingress };
22   Event.emit('new.student', student);
23   return student;
24 }
25
26 export default createStudent;
```

Note that in `line 22`, an event is emitted to notify all the system of that change in the application's state.

Finally, from the `handler` named `newStudent`, Byron will search from a function file `sink/handlers/newStudent.ts` in the working directory, with the structure as follows in listing 5

Listing 5 a handler function file *sink/handlers/newStudent.ts* that implements the reaction for the event `new.student`, storing data in the database

```
declare type Handler = async (
  ctx: any,
  msg: any
) => Promise<void>;

const newUser: Handler = async (
  ctx: any,
  msg: any
): Promise<void> => {
  const data: any = JSON.parse(msg.getData());

  const user: any = ctx.db.User.create({ ...data });
  console.log(user);
};

export default newUser;
```

This is the minimum code and configuration a developer needs to successfully build and run a component using Byron.

In order to test, listing 6 defines an automated unit test that inserts a new student using the mutation `createStudent`, then search for it with the query `getStudent`:

Listing 6 a simulation of an automated test that runs a `createStudent` mutation and asserts that the user was indeed created, by running a `getStudents` query and comparing results

```
describe('Byron basic example', () => {
  let client, server, data;

  beforeEach(() => {
    server.reset();
    server.uuid = mockFunction(() => '42');

    data = {
      name: "John Doe",
      course: "Computer Science",
      ingress: 2020
    };

    client = setupMockClient();
    client.performMutation(gql`{
      createStudent (
        name: ${data.name},
        course: ${data.course},
        ingress: ${data.ingress}
      ) {
        _id
      }
    }`);
  });

  test('querying Students returns inserted data', () => {
    const result = client.performQuery(gql`{
      getStudents {
        _id,
        name,
        course,
        ingress
      }
    }`);

    expect(result).toBe([{ _id: '42', ...data }]);
  });
});
```

3.3 Features

Byron provides features to help in the development process. At the same time it offers code generation, the framework also has support for custom code provided by the developer. In other words, Byron automates part of the code writing, while let developers focus on implementing business logic.

- **code generation**: Byron automates the setup for its microservices, scaffolding code with imports and configurations out of the box. More specifically, the framework generates:
  - *Mongoose Models*: each type defined in the `schema.yaml` is used as base for writing a Mongoose model, a structure that implements a MongoDB schema;
  - *GraphQL Schema*: this file defines all types the API manipulates, such as extended types, mutations and inputs. From `schema.yaml`, Byron extracts all definitions and generates `schema.graphql`;
  - *Docker files*: Docker is the base of the development environment, and Byron generates a `Dockerfile` and a `docker-compose.yaml`.
- **customization**: in order to prevent every system generated with Byron to look the same, the framework supports customization. A developer using Byron can provide its own:
  - *business types*: within the `schema.yaml`, a developer can define its own types that extends GraphQL basic types;
  - *commands*: commands are all action that an API knows how to do. Commands can be *queries* when it's just to fetch data, or *mutations* that change the application state;
  - *handlers*: handlers implement the behavior a given system have as a response to an event;
  - *hooks*: lifecycle hooks tweak commands and handlers behavior, by adding more actions into them;
- **scripting**: Byron, with its homonymous CLI tool, provides an wrap of Docker Compose commands, so there's no need to worry about the generated code – in a future work, a new feature will be implemented to let developers have easy access to the generated code, if they intend to simply scaffold a configured system.

3.4 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.

3.4.1 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.

So, this situation represents a case where Byron fits well.

3.4.2 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.

Thus, this situation represents a case where it's better **not to adopt** Byron. But note: that's only for the core business – other parts of the system, as it's going to be microservices, can benefit from Byron.

3.4.3 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 the system;
- the system doesn't have a big demand yet, so system scalability is not an issue either.

Thus, this situation represents a case where it's better **not to adopt** Byron – a monolith system fits better. But note: that's only for the initial development – once the team begins to grow and the system demand raises, then Byron might fit better on a evolution process from monolith to microservices.

Chapter 4

Study Case

To evaluate Byron's effectiveness, a study case was performed by using the framework to develop Perses, a system for management of scientific initiation in schools. The system developed doesn't intend to be complete and fulfill its needs, it was used for the single purpose of a proof-of-concept that Byron provides decoupling by being event-driven. This chapter presents the study case, along with its application scenarios and the final analysis.

4.1 Context

Perses is a system for managing scientific initiation in schools. It supports multiple **schools**, each *coordinated* by an **user**, counting a handful of **classes** with a *teacher* and *students*.

The system is implemented, accordingly to Byron's terms, into two *components*:

- Auth, for dealing with user authentication;
- Groupings, for dealing with schools, coordinators, classes, teachers and students.

This study case aims to provide details in order to answer the research questions, presented in Section 1. For a better fit with the scenarios, the research questions are here rephrased with more concrete information:

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

Adopting the proposed architecture and considering a single component, is it possible for the Cache to be updated by the Sink with pertinent data?

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

Adopting the proposed architecture, considering more than one component, is it possible for each Cache to be updated by its own Sink with pertinent data?

Q3 Does the proposed architecture provide decoupling between components, in a way that in case of one failing, the others do not fail too?

The code developed for this study case is available at https://gitlab.com/byron-framework/study-case, which is a group of repositories. **Auth** and **Grouping** are the two components coded, while **scenarios-tests** is a code base that runs tests described in the following section. All the following sections present a scenario and steps to reproduce it assuming these basic steps:

- install Byron CLI

  `npm install -global @byronframework/cli`
  (or `yarn global add @byronframework/cli`)

- install **Docker** and **Docker Compose**

  this process depends on which distribution is used, refer to docs.docker.com/install/ and docs.docker.com/compose/install/

- clone the repositories with `git clone <repo url>`, with the URLS:

  ```
  https://gitlab.com/byron-framework/study-case/auth
  https://gitlab.com/byron-framework/study-case/grouping
  https://gitlab.com/byron-framework/study-case/scenarios-tests
  ```

- build the components

  ```
  byron component build path/to/Auth
  byron component build path/to/Grouping
  ```

4.2 Methodology

The study case is based on scenarios, each one stressing a desired feature of Byron, intended to provide means to answer a research question.

4.2.1 Scenario - Consistency within a Single Component

research question	[Q1] Adopting the proposed architecture and considering a single component, is it possible for the Cache to be updated by the Sink with pertinent data?
high level description	creates an user by sending a `createUser` command to the *Auth API* and then sends a `getUser` with the `ID` provided on the first command.
metrics	the result of the second command (doesn't fail and brings correctly the just-created user)

In this scenario, the creation of an user is tested: a user is created accessing the Auth API directly with a `createUser` command, then the same user is requested via `getUser`, to check if there's no error and if the data matches with those provided in the creation.

The steps to reproduce the results of this scenario are as follows:

1. initialize the infrastructure with a broker

   ```
   byron infrastructure up -N StudyCase
   ```

2. run the Auth component

   ```
   byron component up path/to/Auth
   ```

3. enter the *scenarios-tests* directory and run the first scenario

   ```
   docker-compose run rm tester yarn first
   ```

After running this scenario, the clean up steps are:

- remove the scenario-tests container from its directory

  ```
  docker-compose down
  ```

- remove Auth component

  ```
  byron component down path/to/Auth
  ```

- remove the infrastructure

  ```
  byron infrastructure down -N StudyCase
  ```

The action and the metric where chosen because, since a component implements CQRS and Event-Sourcing, at the same time as it responds as soon as the data is

validated, this test gives enough information to confirm that the Auth component's Cache is being updated by the Sink.

Listing 7 presents an interesting part of the testing code:

Listing 7 the main function for running the first scenario, where the user is created, then it is queried

```
1 async function test(): Promise<void> {
2   const id: string = await createUser();
3   await listenToNewUser(id);
4   await sleep(1000);
5   await readUser(id);
6 }
```

One interesting thing to highlight from listing 7 is line 4: to work, the test depends on a *waiting* period of $1000ms$. This snippet of code highlights the *eventual consistency* on the system: the validated data must be broadcast from `createUser` command via event, through the broker into all interested Sinks, before the Cache gets updated and the data gets available for `getUser` command to fetch it.

4.2.2 Scenario - Consistency across Multiple Components

research question	[Q2] Adopting the proposed architecture, considering more than one component, is it possible for each Cache to be updated by its own Sink with pertinent data?
high level description	creates an user by sending a `createUser` command to the `Auth API`, then sends a `createSchool` command to the `Grouping API` with the user ID and finally sends a `getSchool` command
metrics	the result of the last command (doesn't fail and brings correctly the just-created school)

In this scenario, the broadcast of events is tested across multiple components: the creation of a school only happens successfully when a user is provided, so the Grouping component must have updated its cache with the event of new user emitted by the Auth component.

The steps to reproduce the results of this scenario are as follows:

1. initialize the infrastructure with a broker

```
byron infrastructure up -N StudyCase
```

2. run Auth and Grouping components

```
byron component up path/to/Auth
byron component up path/to/Grouping
```

3. enter the *scenarios-tests* directory and run the second scenario

```
docker-compose run -rm tester yarn second
```

After running this scenario, the clean up steps are:

- remove the scenario-tests container from its directory

```
docker-compose down
```

- remove Auth and Grouping components

```
byron component down path/to/Auth
byron component down path/to/Grouping
```

- remove the infrastructure

```
byron infrastructure down -N StudyCase
```

These actions and this metrics were chosen because it's a way to check whether the Grouping component indeed received from the event the just-created user data. This test gives enough information to confirm that Grouping Cache is being updated by Grouping Sink with data from the event emitted by Auth API.

Listing 8 presents two snippets from each of components' `schema.yaml`:

Listing 8 fragments of Auth/schema.yaml (lines 1 - 13) and `Grouping/schema.yaml` (lines 16 - 29) highlighting that the commands are in different components

```
1  name: Auth
2  namespace: StudyCase
3
4  content:
5    # ...
6    commands:
7      - name: createUser
8        type: mutation
9        returnType: 'User!'
10       parameters:
11         - name: data
12           type: CreateUserInput
13
14 ---
15
16 name: Grouping
17 namespace: StudyCase
18
19 content:
20   # ...
21   commands:
22     - name: createSchool
23       type: mutation
24       returnType: 'School!'
25       parameters:
26         - name: data:
27           type: CreateSchoolInput
28         - name: userID
29           type: 'ID!'
```

4.2.3 Scenario - Decoupling of Multiple Components and One Failing

research question	[Q3] Does the proposed architecture provide decoupling between components, in a way that in case of one failing, the others do not fail too
high level description	creates an user by sending a `createUser` command to the Auth API, then sends a command that causes Auth component to stop (simulation of a failure), creates a school and finally checks whether the event of role update is emitted properly and the school is created with the user id
metrics	the result of the creation of the school (doesn't fail and brings correctly the just-created school with the user id)

The steps to reproduce the results of this scenario are as follows:

1. initialize the infrastructure with a broker

   ```
   byron infrastructure up -N StudyCase
   ```

2. run Auth and Grouping components

   ```
   byron component up path/to/Auth
   byron component up path/to/Grouping
   ```

3. enter the *scenarios-tests* directory and run the third scenario

   ```
   docker-compose run -rm tester yarn third
   ```

After running this scenario, the clean up steps are:

- remove the scenario-tests container from its directory

  ```
  docker-compose down
  ```

- remove Auth and Grouping components

  ```
  byron component down path/to/Auth
  byron component down path/to/Grouping
  ```

- remove the infrastructure

  ```
  byron infrastructure down -N StudyCase
  ```

This scenario is interesting to prove that the Byron provides decoupling between components, such that, even with Auth component being killed, Grouping component managed to provide an answer to the `createSchool` command.

4.3 Analysis

The previous section brought all three scenarios this study case implemented, with their descriptions and some details about the implementation. This section summarizes the analysis into one.

The objective with the study case was to reason about three main questions presented in Chapter 1 and rephrased in Section 4.1:

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

Adopting the proposed architecture and considering a single component, is it possible for the Cache to be updated by the Sink with pertinent data?

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

Adopting the proposed architecture, considering more than one component, is it possible for each Cache to be updated by its own Sink with pertinent data?

Q3 Does the proposed architecture provide decoupling between components, in a way that in case of one failing, the others do not fail too?

Scenario **Consistency within a Single Component** (Subsection 4.2.1) presented elements to answer **Q1**. Those elements were evidences to prove that, even with some delay, the architecture of a **Component** is data consistent (eventual consistency), i.e., Auth's sink indeed listened to the event emitted by the API and broadcast by the broker, and the Cache was updated. Scenario **Consistency across Multiple Components** (Subsection 4.2.2) presented elements to answer **Q2**, evidencing the architecture provides data consistency across multiple components. Scenario **Decoupling of Multiple Components and One Failing** (Subsection 4.2.3) presented elements to answer **Q3**, which proves that the Byron components are decoupled in a level they hold up eventual failures without causing a domino effect.

With positive answers for the research questions, it's enough to say that Byron fulfill its proposals of being data consistent – even if it's eventual consistency – and of providing decoupling between its components – as much as it stands a component failing without causing fails across the system.

Chapter 5

Comparison between Byron and other frameworks

There are a plenty of frameworks for web development available [1]. Ruby on Rails and NestJS are two interesting frameworks due to their popularity – the former is very known, the latter is still rising – and to their approaches of how to organize a system.

5.1 Ruby on Rails

Ruby on Rails is a framework that implements MVC architecture – Model-View-Controller, a layered architecture. This architectural style provides good organization for code, which fits well for bootstrapping the system. Since its communication model is via function call – it runs as a single process –, Rails supports strong data consistency without latency.

On the other hand, as presented in Subsection 3.4.3, Byron adds much complexity to early moments of the development, when the development staff doesn't count with many teams and the system offer a small set of features.

In the long-term, as the system becomes more complex and more accessed, hence the requirements for availability and scalability raises, a Rails system ends up facing the monolith downsides. As a monolith is a single code base that is deployed into a single process that holds all features of the system, scalabilty becomes an issue: first, because the management of the only repository with many teams tends to get confuse and time consuming; then, when deployed, the system gets resource consuming and hard to optimize since its a single process running.

Byron fits better as a tool for developing a version of the system when these issues show up, such as the growth in the development staff and the raise in complexity of the

[1] assuming that WikiPedia doesn't count every single framework available, it's safe to say that there's *at least* a few dozens of web frameworks available – here https://en.wikipedia.org/wiki/Comparison_of_web_frameworks

system. As Byron provides a system meant to follow microservices architecture, then there's gain in both ways: there might be more than one code base, so the management of the repository gets easier; the deployment of the system get less resource consuming, since microservices provides scalability on demand, i.e., each microservice gets scaled up or down accordingly to its own needs.

5.2 NestJS

NestJS is a framework that is rising in popularity[2] that follows the Reactive System principle set, among others, such as Object Orientation and Functional Programming. Its supported features go from providing a couple of platforms as HTTP Server, to a progressive architecture – that goes from layered up to microservices.

NestJS adopts an imperative style, where every functionality must be explicitly coded by the developer, and, with that, one can fully configure the desired behavior. Byron adopts a more declarative style, as is the case of `types` in the `schema.yaml`. What's underneath this style is that Byron provides code generation, saving the developer some work. Also, the `schema.yaml` provides a fast way one can know what the component does.

Another difference between NestJS and Byron is related to code generation: Byron provides Docker configuration to the developer, i.e., a component coded with Byron has out-of-the-box benefits of a containerized system. As future work, it's aimed to support generation of configuration files for **Kubernetes**[3] – a cloud native container orchestration tool.

[2] a few companies are already adopting it (https://docs.nestjs.com/discover/companies) and its repository on GitHub points tens of thousands of other repositories having it as dependency (https://github.com/nestjs/nest/network/dependents?package_id=UGFja2FnZS00NTI3NzIzMzQ%3D)

[3] official website: https://kubernetes.io/

Chapter 6

Conclusion

Modern conditions of software development are extreme in some cases – multiple development teams over the same system, deployments that work under massive workload, and high demands for availability and short response times [Wam19]. A plenty of techniques and architectures – such as microservices, command-query responsibility segregation and fast-data architecture – were created to solve those problems, but there's yet a handful of practices that let systems vulnerable to a domino effect.

The aim of this project was to create a solution for developing microservices that values decoupling, an architecture that provides decoupling while keeping 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 component, even if it is eventual consistency?

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

Q3 Does the proposed architecture provide decoupling between components, in a way that in case of one failing, the others do not fail too?

This project presented Byron, an event-driven microservices framework that reunites a collection of principles and architectural standards aiming decoupled yet data consistent components. A framework that offers, via a CLI and a DSL, a declarative approach for defining a component – a set of an API for handling HTTP requests, a local Cache to guarantee independence from others, and a Sink to listen to events and update local cache.

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.

During the development of this project, a few ideas emerged for future works around Byron. 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;

Other relevant works around Byron are scientific experimentation. Up to now, there are two main ideas:

1. a research about the developer experience with the framework – how its abstractions and its workflow benefits development teams using it;
2. since data consistency is eventual, a benchmarking of the time involved in the broadcast of events;

Bibliography

[Bon+14] Jonas Bonér et al. **The Reactive Manifesto**. Tech. rep. 2014.

[GN18] Justin Garrison and Kris Nova. **Cloud Native Infrastructure - Patterns for Scalable Infrastructure and Applications in a Dynamic Environment**. 2018.

[New15] Sam Newman. **Building Microservices - Design Fine-Grained Systems**. 2015.

[Ric15] Mark Richards. **Software Architecture Patterns - Understanding Common Architecture Patterns and When to Use them**. 2015.

[Wam19] Dean Wampler. **Fast Data Architectures for Streaming Applications - Getting Answers Now from Data Sets That Never End**. 2019.