Byron - a Cloud-Native Reactive Microservices Framework

João Francisco Lino Daniel
Leonardo Lana Violin de 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
Renato Cordeiro Ferreira

Contents

Chapter 1

Introduction

Building and deploying complex distributed systems is hard. There are a lot of aspects that increase the chances of failure and there are a handful of tasks that make the development process repetitive and time consuming. Automation, observability, availability and continuous delivery are some of the key concerns in order to succeed in this task.

The aim of this project is to build Byron, a framework that helps developers to work around the initial overload in their processes to build cloud-native reactive microservices applications, i.e., systems that use a state of the art architecture to achieve these goals.

Byron will define a Domain Specific Language (DSL) to represent the application's entities and their relationships, allowing developers to focus on the business domain. It will also count with a Command Line Interface (CLI) that will be the single tool to set up the app in the cloud by bootstrapping basic resources and rolling new releases. This way, Byron will make it easier to create and deploy distributed systems for complex domains.

In the next sections, **chapter 2** presents a literature review about architectural principles and infrastructure technologies that will be used. **Chapter 3** details the proposal of the project, delineating the development workflow intended for Byron. Finally, **chapter 4** presents: Lean Inception Methodology adopted to define the development of the framework and the CLI along the year as well as the general work plan to make this project.

Chapter 2

Literature Review

2.1 Architecture

The software architecture of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both. Documenting Software Architectures, Bass et al

2.1.1 Layered Architecture

The **Layered Architecture**, also known as **n-tier**, it is the most common software architecture, because it works very well as a starting point for most applications ([Ric15]). It is often implemented in four layers: presentation, business, persistence and database ([Ric15]), in which a given layer provides to another layer strictly above it an abstraction of all layers beneath it. Hence, two powerful features of this architecture are separation of concerns and layer isolation. The former provides a very clear organization for the structure of the system, whereas the latter means that changes within a layer should not impact other layers.

Even though the Layered Architecture fits very well in the beginning of development, there are two major problems with it. First, there is the **architecture sinkhole anti-pattern** – many requests may be attended by a single layer deep in the architecture, but to get into this layer, they need to run through all layers, thus causing a unnecessary overhead. Second, this architecture tends to lead to a monolithic application ([Ric15]) – a structure that keeps all application features tied up into a single executable or binary –, which compromises software deployability and scalability.

2.1.2 Microservices Architecture

The **Microservices Architecture** is an architectural model on which the system gets divided into a set of independent components called **microservices** – not a single codebase, executable nor binary anymore. It rose in popularity because it provides two ways of scalability: in team sizes and in volume of work to be executed ([New15]). This independence between parts of the system enforces independent deployment of components, allowing different development teams to move at their own speed. It also enables each microservice to be scaled up or down independently, according to its own workload ([Ric15]).

Since this model of architecture is very new, there is still a debate on how to divide an application into microservices. Currently there are two main guidelines: by **functionality**, where each one gets its on microservice; or by **context**, where each *group of functionalities* – defining a context – gets its own service ([New15]).

Although this architecture has become very popular, it is not the best solution for all systems. The Microservices Architecture focused scalability. Whenever that is not an issue to the application, then it is better not to adopt this architecture ([New15]). That is because microservices offers a series of difficulties to overcome since it creates a *distributed system*. Those difficulties can be dealt with some microservices patterns, as follows:

- **API Gateway** provides an abstraction of the system APIs to their clients, making the microservices system to look like it is a single component system.
- **Service Mesh** provides the system a communication model based on rules and policies, managed by a service-independent component, enhancing the decoupling of responsibilities.
- **Sidecar** deals with peripheral tasks, such as dealing with redirections and telemetry, hence providing each microservice an abstraction of the rest of the system.
- **Event Sourcing** enables microservices to care of their own state in isolation, whereas notifying changes to other components via message-passing.
- **Command-Query Responsibility Segregation (CQRS)** decouples the reading and writing model of services, thus optimizing each operation according to the use cases.

2.2 Infrastructure

2.2.1 Virtualization

Running servers *on-premises* – self-managing physical machines – is often expensive due to its demand of time to configure and maintain. **Virtualization** – emulating the hardware in software – is an alternative to that.

Virtual Machines

For many years, **Virtual Machines (VMs)** were the trending solution to achieve idempotence – the ability to reproduce behavior based only on the input: same input, same output, no interference of environment. With a hypervisor running multiple VMs, it is possible to isolate an application from others in the same physical machine, so there is no interference between apps; it also enables a better fit between the application and the environment, therefore spending less resources; and improves portability, allowing the app to run on any physical machine ([GN18]).

When a VM is launched, the Operating System (OS) of the physical machine (host) emulates another entire OS (guest) over itself – it is a big overhead compared to running the application directly in the host. For that reason, VMs are no longer the trending solution in virtualization – although it is still important when it comes to the cloud (subsection 2.2.2).

Containers

Containerization is a technique that encapsulates code, dependencies, networking and file-system into a single process. There is a resemblance with what a VM does, but there's a major difference: a container is an isolated process created directly by the host OS – there is no need to stack another OS.

Among the containers engines, Docker is currently the most relevant. For Docker, an "image is a lightweight, standalone executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings" ([Inc19]). In addition, a Docker container is simply an instance of the image running in memory.

2.2.2 Cloud

Infrastructure as a Service

Cloud providers offer **Infrastructure as a Service** (IaaS), that means all the infrastructure – networking, storage and compute – can be consumed as needed ([GN18]). IaaS has changed the relationship between infrastructure and business: the demand for infrastructure goes from buying hardware to requesting VMs via an API. This format creates new possibilities for work automation, even though it still requires infrastructure management.

Platform as a Service

Platform as a Service (PaaS) goes one step higher in abstraction level: it abstracts the operating system and environment from applications – it is platform's job to manage the infrastructure.

This abstraction of the O.S. forces developers to write code in a different way, as the platform will be managing it – with PaaS, there is no more SSHing into a VM to install dependencies. The Twelve-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 platform reality.

Even though PaaS is a very high-level solution to infrastructure, there is a trade-off: developers have to give application flexibility in order to make the infrastructure being someone else's responsibility ([GN18]).

Cloud-Native Infrastructure

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.

Adopting useful abstractions enables CNI to manage the IaaS by creating a new layer of abstract objects above it. It then exposes an API to be consumed avoiding re-implementation. Being managed by software is a core feature of CNI: it enhances scalability, resiliency, provisioning and maintainability of the infrastructure. That is because the management software enters in

between the layer of abstraction created by CNI and the underlying IaaS, mapping the new abstract objects into preexisting IaaS entities.

These changes in the relation with infrastructure not only impacts how it runs, but also takes the application running over it a step further into what 12-Factor proposed, consolidating what is called the **Cloud-Native Principle**.

2.3 Application Principles

2.3.1 Cloud-Native

Cloud-Native (CN) is a set of architectural principles for applications that run on a cloud-native platform. Cloud-Native applications should follow four principles: **resiliency** embraces failure instead of trying to prevent them; **agility** allows fast deployments and quick iterations; **operability** adds control of the lifecycle from inside the app instead of relying on external processes and monitors; and **observability** provides information to answer questions about the app state ([GN18]). These design characteristics can be achieved with a few methods:

- **Resiliency** can be implemented with three concepts in mind: *design for failure*, *graceful degradation* and *declarative communication;*. Traditional infrastructure was built to resist failure, but CN apps embrace failure and implement solutions for fast starts and restarts; CN apps also are exposed to a huge amount of traffic and must handle situations where there is a massive incoming load. Apps with Graceful Degradation always answer under these extreme situations, even if the only possible answer is a partial or outdated (cached) response ([GN18]). In a CN environment, communication happens through the network, which allows the parts to send high-level declarative message – implementation details get covered by an exposed remote API. The adoption of this behavior moves the abstraction ruler upwards and generates a standardized communication model ([GN18]).

- **Agility** can be achieved by adopting microservices architecture (subsection 2.1.2): each microservice can be developed by separate teams and can be delivered independently – the key is to keep valid the contracts between them.

- **Operability** can be obtained when an application exposes health check with a command or endpoint, because the application is the one who should know if it's running properly, or in which conditions is it running on (the spectre is wider than only "healthy" or "unhealthy").

- **Observability** is enhanced with telemetry data, that focus in service information. There's a considerable resemblance between Health Check reports and Telemetry Data, but the former doesn't need to trigger any alert due to the automation of the platform underneath – it should be prepared to restart the app in case of failure, for instance –, but the latter deals with data that should be alerted in unexpected cases ([GN18]).

2.3.2 Reactivity

Today's demand for high availability and low response times led organizations to create a new set of patterns for developing software. Reactive Systems is the set of patterns that makes systems

more robust, more resilient and more flexible ([Bon+14]).

Reactive Systems follow four characteristics: **responsiveness**, they focus on providing consistent and reasonable response times, so that it enhances reliability and simplifies error handling; **resiliency**, they keep the system responsive in face of a failure; **elasticity**, without bottlenecks, the system can stay responsive in various workloads; and **message driven**, interchanging asynchronous messages along the system provides loose coupling by defining boundaries between components.

- **Responsiveness** can be implemented by returning a response to the user as soon as the system identifies the data is consistent.
- **Resiliency** is achieved by four actions: *replication*, when a component has many copies; *containment*, as a system keeps its own state isolated from other systems; *isolation*, meaning that parts are independent in lifecycles and have independent processes; and *delegation*, when a process delegate a task to another, it gets free to do another task or even to observe the original task in the other process.
- **Elasticity** means that the system is designed in compatibility with up scales and down scales, hence presenting no bottlenecks.
- **Message-Driven** is a away to define boundaries between components: asynchronous message passing does not violate the isolation of internal state, furthermore it allows communication between the system components. It can be implemented using a *message broker* (section 2.4.2)

2.3.3 Fast Data

For years, big data applications have been processing data in batches. Nowadays the scale of communication over the internet made this batch-orientation a commercial disadvantage and modern systems are migrating to stream-oriented data processing. This stream-orientation allows data to be processed as soon as it arrives in the system, creating what's called **Fast Data Systems (FDS)** ([Wam19]), which handles data as possibly infinite streams. Batch processing, in turn, is just a subcase where streams have an end.

To follow a **Fast Data Architecture (FDA)**, a system needs to implement a stream-oriented data back-plane, relying on log systems and message queues. It also needs to make logs its main abstraction – streams comes from beneath it, such as telemetry data, service logs with implementation details and write-ahead for databases CRUD transactions.

2.4 Databases

When building system, more often than not, we need to save the data generated by the usage of the system. A common way to do so is using **databases**, a collection of related data, known facts that can be recorded and have implicit meaning ([EN16]).

2.4.1 Relational

A **Relational database** is based on the model of mathematical relations - which means tables of values. They generaly use the **Structured Query Language (SQL)** to manage the data. Relational databases raise in popularity due to the soft learning curve of SQL along with the simplicity of the model. This query language made relational databases also known as SQL databases.

Although efficient and simple, the use of tables to store the data is a rigid model, not allowing easy changes to the structure – if one item in the table changes its structure, the whole table must be changed. Hence, making structural changes in production relational databases a troublesome task.

Another limitation of the relational model is that not all data is better represented by a table. This is often referred as **impedance mismatch**, where a formatting layer must be added between the application and the database – lists are an example, because SQL databases do not support the representation of arrays.

Relational databases were not designed to run on clusters, but there are workarounds. One alternative is **Sharding**, a technique on which different sets of data can be store in different databases ([SF13]), but the application must control which database is used to save its data; another is using **Clustered Relational Databases**, like Microsoft SQL Server, that uses a cluster-wide file system to store data, but it still has only one single point of failure ([SF13]).

2.4.2 Non Relational

The **Non-Relational databases (NoSQL)** are databases that don't follow the mathematical model of relations – some model their data as a graph, other as a document, for instance. The term NoSQL came out of an relational database that didn't use SQL as query language ([SF13]).

Document-Oriented Database

Document-Oriented Database uses document as its main model. It usually stores and retrieves documents written in data serialization languages – such as JSON, YAML or XML –, which allows to write the data in a **hierarchical structure**. Maps (lists), collections (hash) or scalar values (integers, strings and booleans) are possible values in that structure.

There is a couple of benefits of using Document-Oriented Databases. The first is the fact that documents are weakly structured, which means that the keys contained in each document can vary – this is possible due to the lack of rigid structure. The second is high availability: these databases use a primary-replica setup, where the data is replicated in other instances of the database, freeing the application of concerning about which nodes are available for reading.

The **replica set** setup is also really beneficial for scalability in heavy-read applications – those apps where the ratio between read and write is very big –, as we can just add another replica to the set to use it for reading. When it comes to heavy-write applications – where the ratio between read and write is very small –, the sharding model can be used, similarly to relational databases.

Message Broker

To explain what a **Message Broker** is, we need to define some terms first. An **event** is a significant transition in state, for example, consider a car going from "for sale" to "sold" – an event doesn't *exist*, it only *occurs*. A **message** is the implementation of an event, it is an object used to notify the system an event has occurred.

The Message Broker uses a **Message Queue (MQ)** to make circulate the messages – usually asynchronously emitted – from some component of the system to other components. The MQ is both responsible for receiving messages and broadcasting its arrival across the system.

Message Brokers also use the **Event Log** model to store their data. This means that they don't store the final (or current) state of the system, they store the history of changes – the broadcast messages – and build the state using this history.

Chapter 3

Proposal

Developing software that attends modern needs, such as scalability, became a complex task. Even though each demand may be individually solved by the adoption of an architectural principle or an infrastructure tool, there is still the challenge of implementing solutions together coherently.

This project proposes to build Byron – a framework that aims to assemble the state-of-art architectural principles. Byron will make easy to build cloud-native reactive microservice systems easy by providing a simple development workflow implemented in a command-line interface (CLI).

3.1 Developers Interface and Workflow

Byron's Command-Line Interface (CLI) will be the tool to guide the development workflow within Byron Framework. The CLI will bootstrap a directory structure on which the developer will work upon. This structure provides organization of the files, according to:

- `customResolvers/`: a directory to store all **custom resolvers** the developer may use;
- `hooks`: a directory to store all **lifecycle hooks**, functions to be called in a given situation;
- `config.byron`: a file containing infrastructure configuration, based on Helm Charts;
- `credentials.byron`: a file containing credentials to access the cloud provider;
- `schema.byron`: a file describing the GraphQL API for that *component*, along with Byron annotations to indicate custom behavior;

Once it is done, the developer using Byron might define the entities his software will deal with using the DSL. Byron will also provide a set of annotations in order to specify some operations and code-related configurations, from validation of type's attributes to configuring the granylarity of the generated events.

Whereas annotations in the DSL provide standard behaviors, the developer will probably want to define into the code some business logic that differs from standards. **Life cycle hooks** and **custom resolvers** supply ways of doing that. The former is to be used for ensuring policies and debugging purposes; the latter creates new features in the API along with its implementation.

The developer will be able to manage the deployment of that application in the cloud provider using Byron's CLI tool, as long as he keep his interactions via this presented interface, not editing the generated code. Byron allows the user to access this code, but once it is done, the CLI will not manage the deployment anymore. This might cause estrangement, but it follows the idea of using Byron exclusively as a bootstrapping to the system, overcomming the initial overhead of setup.

3.2 Generated System's Architecture

Byron follows a rigid and well-defined architecture, presented as follows.

3.2.1 Context

Byron will build a single API back-end that will be consumed by external clients – either front-end sites and mobile apps, or other systems – controlled by users. All this communication will happen via HTTP requests, following GraphQL standards. All the system will be hosted at a Kubernetes cluster.

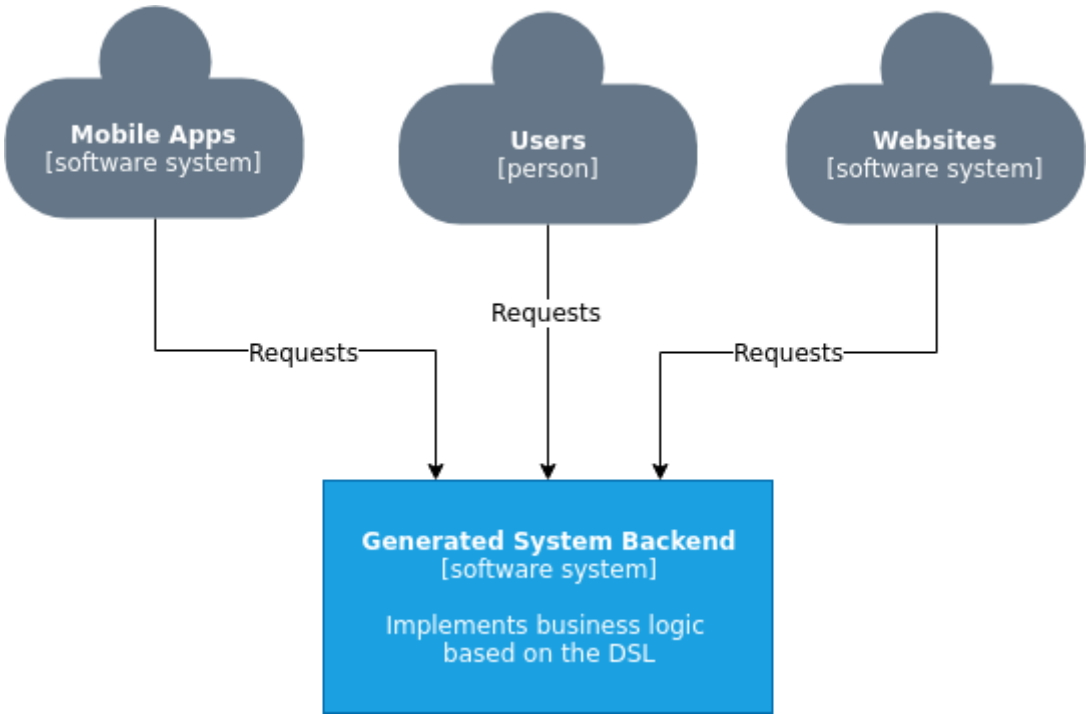

Figure 3.1: Context: a model presenting the application context.

3.2.2 Containers

The back-end is composed by three major CONTAINERS: the **API Gateway**, the **Message Broker** and a set of ***Components***. The API Gateway unites each *component*'s API into a single one, so when a request comes in, the Gateway redirects it to the correspondent *component*. Each *component* is responsible for treating and validating a subset of the system's data model. Frequently, the same piece of data is required by different *components*, so they must communicate

with each other to spread the it. The Message Broker is the backbone that supports all the communication between *components*.

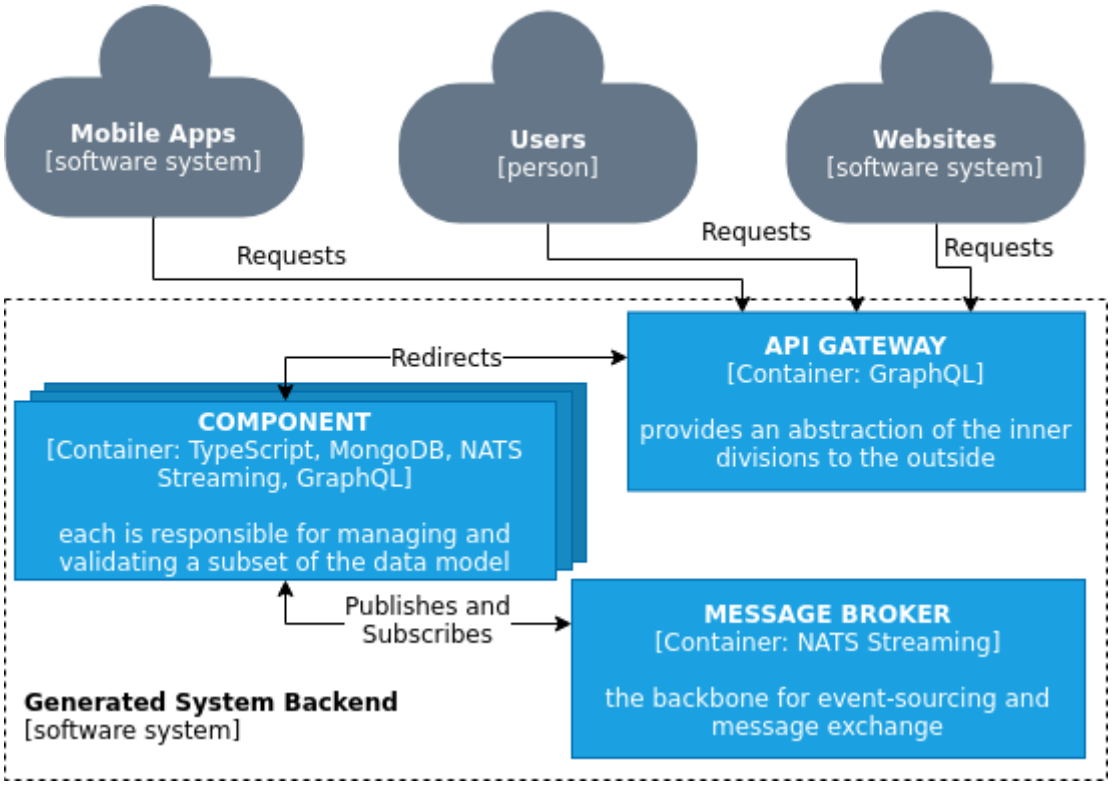

Figure 3.2: Containers: a model presenting the application containers.

3.2.3 Components

Each element in the set of *components* is subdivided into three components: the **API**, the **Cache** and the **Sink**. The capabilities of that *component* are exposed to the system via API – and furthermore, to the outside world via API Gateway. When a request comes in, the API is the one who deals with it. First it validates the data, applying validation rules that often requires reading the Cache. When it's all fine, the API takes two actions: it responds through the Gateway back to the client and it also wraps data into events and publishes them into the Broker. The Sink is responsible for listening to the broker for relevant messages, so that when they are broadcast, it unwraps the data and updates the *component*'s state by inserting that data into the Cache.

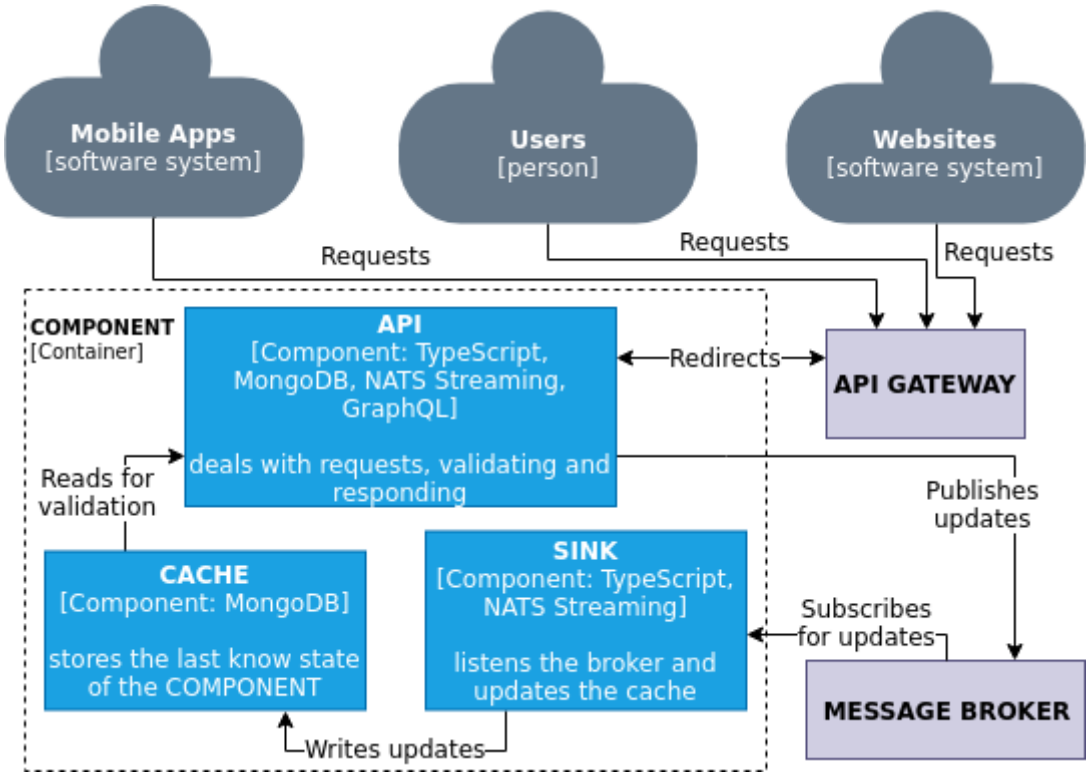

Figure 3.3: Components: a model presenting the application components.

Chapter 4

Work plan

4.1 Lean Inception

The **Lean Inception** is a process created with the objective of pinning down which features should be present in the **Minimum Viable Product (MVP)** and how to implement them, aligning technical knowledge and business agreement along the team about the product ([Car14]).

This method was elected to be followed, because it provides ways of aligning the team's vision, and also guides the definition of what should and should not be developed. First we achieved a common vision about the framework, then we define the order and priority of the features we should implement. Our notes made during this process can be found in the attachment section.

4.2 Calendar

The next steps are described as follows. **Inception and Prototyping** unites what is left to do in terms of preparation and prototyping: finish the lean inception in order to define a clear sequence of features and close the prototyping in order to draw the outlines of the DSL interface. **Partial Monograph** is relative to the tasks of writing a mid-process monograph describing the on-going project, along with definitions that came after this proposal. **Green**, **Yellow** and **Red Features** are consequences of the lean inception: each tier is a set of features to be developed, and the sequence green-yellow-red represents the relevance and value for the MVP – the specification of the features in each tier will be announced in the partial monograph. **Study Case** stands for the study case over which Byron will be used as a proof of concept: the idea is to build a micro blog inspired on Twitter, an application that is known to need a high scalability and availability – a good match of requirements for Byron. Finally, **Final Assets** details the tasks for the conclusion of this project, such as writing the final monograph and preparing the poster.

Task	Duration	Deadline
Inception and Prototyping	1 month and a half	June, 10th
Partial Monograph	3 weeks	mid July
Green Tier Features	3 weeks	July, 1st
Yellow Tier Features	4 weeks	August, 1st
Red Tier Features	4 weeks	September, 1st
Study Case	3 weeks	September, 25th
Final Assets	5 weeks	November, 1st

Table 4.1: Calendar table

Bibliography

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

[Car14] Paulo Caroli. **Lean Inception**. 2014.

[EN16] Ramez Elmasri and Shamkant B Navathe. **Fundamentals of Database Systems**. 2016.

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

[Inc19] Docker Inc. **What is a Container? | Docker**. 2019.

[New15] Sam Newman. **Building Microservices**. Tech. rep. O'Reilly, 2015.

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

[SF13] Pramod J. Sadalage and Martin Fowler. **NoSQL distilled - a brief guide to the emerging world of polyglot persistence**. Addison-Wesley, 2013, p. 164.

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