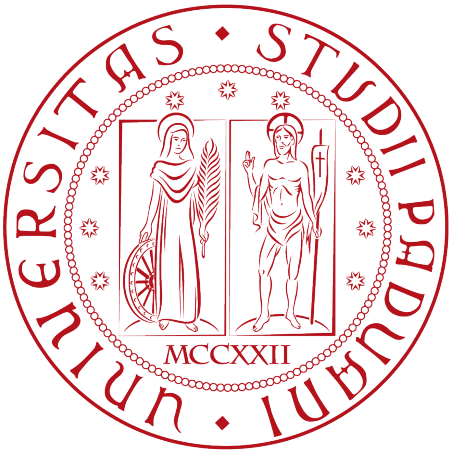
University of Padova

Mathematics Department "Tullio Levi-Civita"

Bachelor's Degree in Computer Science



Development of a Redis management platform with serverless microservices on AWS

Bachelor's Thesis

Advisor

Dr. Armir Bujari

Author

Alberto Schiabel - 1144672

Academic Year 2018-2019

I confirm that this Bachelor's thesis is my own work and I have documented all sources and material used.

Abstract

This document describes Queue Controller, a software designed and developed by the author during his internship at Pagination S.r.L. The allotted period for the internship has been 320 hours, spread between June and August 2019. Queue Controller offers a web-based solution to monitor and interact with business data stored in a Redis database. The system is hosted on Amazon Web Services and is composed of multiple microservices running in serverless AWS Lambda functions.

This thesis describes the reasons why this software was built, the technologies involved, the development phases and the intended future use. This document also includes some discussions about the importance of software testing and the advantages of automated deployments of software artifacts.

Due to the existence of a non-disclosure agreement with the hosting company, this document only presents a portion of the work done at Pagination S.r.L.

"Without requirements or design, programming is the art of adding bugs to an empty text file"

— Louis Srygley

Acknowledgements

First of all, I would like to express my utmost appreciation to Dr. Armir Bujari, who has been my thesis advisor, for the help and the attentive support he gave me during the internship and the writing of this thesis.

I owe my deepest gratitude to my parents for the support in these 3 years of sacrifices to balance study and work, and to my brother for being the first person to put me in front of a computer.

I want to thank the Pagination team for having welcomed me and guided me through the internship experience. In particular, I'm thankful to my tutor Simeone, who proved to be a very talented and capable person.

I also wish to express my gratitude to my previous company, Brainwise, which gave me the possibility to start my professional career as Software Developer from July 2016 to April 2019. I'm glad we parted ways in the best of terms.

Last but not less important, I'd like to thank the new friends found in Padova: in particular Francesco and Ciprian.

Padova, 25 September 2019 Alberto Schiabel

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The purpose of this thesis is to illustrate in detail the activities carried out by the undergraduate Alberto Schiabel during the internship held at Pagination S.r.l. from June 3, 2019 to August 9, 2019. The internship lasted a total of 320 hours and was supervised by the tutor Simeone Pizzi.

This chapter introduces Pagination S.r.l. and the reasons why I chose it, as well as the high-level project goals and structure of this thesis.

1.1 The Company

Pagination S.r.l. is a small business-to-business company that offers automatic layout cloud services, mainly for the generation of catalogs and price lists. Pagination was born in 2009, and despite its small size (only 6 employees) several national and international companies trust its products; some of them are *Fisher*, *Geox*, *Prestashop*, and *Gardner Denver*. A complete list of its customers can be found at https://pagination.com/customers.

Pagination S.r.l. stands out compared to other companies in the area due to the young age of its employees and the usage of cutting-edge technologies, such as *Go* microservices running on Amazon Web Services. It is also one of the few companies I know of that takes automatic software testing very seriously.

Figure 1.1: Pagination S.r.l. logo.

Pagination is composed of three main areas:

* Administrative area: it sets the business goals, handles the human and economical resources, and decides marketing strategies;
* Graphic design area: it deals with the visual aspect of the projects as well as

Name	Pagination S.r.l.
Address	Piazza Garibaldi, 8, 35122 Padova
Telephone	+39 049 490 6493
Email	luca.reginato@pagination.com
Website	https://pagination.com
VAT Number	IT04279160263

Table 1.1: Resume of the Pagination S.r.l. details.

the communication with the stakeholders to understand their design preferences. They produce layout templates for the customers' documents;

* Technical area: it develops and maintains the software tools and products used by the customers, as well as managing the automatic test and delivery processes of the created software.

Due to the small number of employees, these main areas aren't necessarily strictly separated: some graphic designers also deal with minor software programming tasks.

As a Full Stack Developer intern, I joined the software development team in the technical area.

1.2 The Core Product

Pagination provides a cloud software that allows its customers to generate documents where the data to be published automatically fits a custom layout template. This software runs on Amazon Web Services and uses *Redis* as real-time database. The generated documents are mostly catalogs and price lists, but many companies also use the product to create brochures, technical documents, and menus for restaurants. Each of these documents is generated from data expressed in a tabular format. Depending on the data provided by the customer, Pagination also offers multilingual support.

First of all, the customers communicate their requirements to Pagination and provide some sample documents as well as an example of the data format used to generate said documents. Then, the graphic designers prepare a template layout for the documents and suggest potential data edits to better satisfy the customer. Templates are usually built using Adobe InDesign. After this manual set-up phase, the customer can upload its data and launch the document generation workflow from the web interface of the product. This process can take many hours to complete and can be seen as a pipeline with tasks that can be partially executed in parallel. The user interaction in this automatic phase can be resumed by Image 1.2.

The customer data is often defined as structured formats, such as *Excel*, Structured Query Language and Comma-separated Values files. However, Pagination also offers web integration solutions to ingest structured data from a plethora of third-party sources, like *Salesforce*, *Akeneo* and *PimCore*.

1.3 The Internship Project

The automated layout solutions offered by Pagination S.r.l. are stuctured as a distributed Extract, Transform, Load pipeline running on AWS. Each pagination process

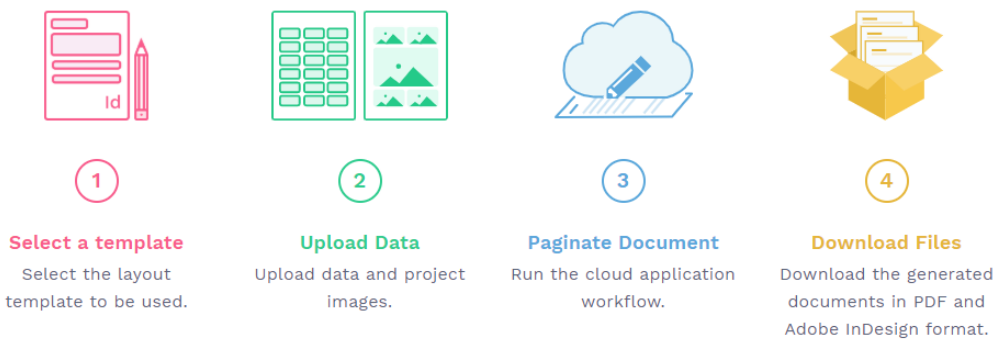


Figure 1.2: Steps required by a Pagination customer to generate a document with the given style and structured data.

requires validation, data extraction and transform stages. There also exist the possibility to generate a PDF document in parallel, crafting parts of it in multiple nodes running on the cloud and then merging them together. Such a complex process may fail in multiple situations, e.g. when the input data presents invalid entries, or when a bug in the company proprietary software causes a deadlock situation on a project.

The company uses Redis to implement priority queues for the pagination requests their software receives. While I cannot disclose the whole structure of this proprietary software, the following entities must be introduced to comprehend the internship project:

* **Stage**: it's a virtual concept that represents the environment where a pagination process is executed. It roughly translates to the standard *"development"* and *"production"* software stages;
* **Job Queues**: *Redis* lists that store information about single documents that should be paginated. They are identified by a key that contains the stage and the unique ID of the request. For each job, a number of other related data is stored and updated on other Redis data structures;
* **Locks**: when a pagination request arrives, a lock acquisition is attempted on the pagination's project. Locks are represented as Redis keys whose value is a unique code necessary used to authorize further edits to the same lock it refers to;
* **Request Queues**: *Redis* lists that contain information about the priority of the requests received, as well as their concurrency level (i.e. how many document splits should be paginated in parallel);
* **Job Result Lists**: for every pagination request there are 3 more *Redis* lists to track the failed, completed, and deleted jobs.

Pagination S.r.l. wanted a platform that offered an easy and rapid access to its code product internal process. Without this platform, apparently simple operations like monitoring the pending pagination requests, unlocking a project or deleting a pagination job may only be performed by a technician. It's a tedious project that involves opening multiple Secure Shell connections and manually running a number of

commands against a *Redis* database instance. The internship project, named *Queue Controller*, has the purpose of becoming this platform.

During the first internship week, the tutor proposed a number of constraints that didn't appear in the original work plan:

* The web interface should be as clean as possible and its User Experience should be extremely intuitive, as it should be used by non tech-savvy employees;
* Whenever possible, operations that alter the *Redis* database state should be run as atomic operations via *Lua* scripts;
* The Redis structure upon which Pagination S.r.l. software relies on is extremely volatile and may change frequently. I should assume that the data currently stored in a *Redis* list may be moved to a *Redis* hash in a matter of days. Therefore, *Queue Controller* should be as decoupled as possible from the actual *Redis* data structures it needs to interact with.

When I proposed to integrate *Queue Controller* with an authentication system, I was told that the company doesn't possess an internal authentication system yet. However, my code should be structured in such a way that can make it easy to integrate it in the future.

Reasons for the choice of this Internship

I met Pagination for the first time at the *StageIT 2019* event; it was one of the few companies that, to me, seemed worth an interview. Their project caught my attention mainly because it dealt with technologies that are highly on demand in the job market: cloud services, full stack development, and real-time databases. The company is based in Padova, exactly one kilometer far from the train station. Since I started the internship with 3 exams still pending, I opted to choose a company based in the same city of my University, in order to optimize train subscriptions and the time required to travel from the University to the office in the same day.

Unfortunately the location of the company required me to perform a quite long commute every day, but it was the nearest company I've found with an appealing project.

Other factors that proved to be fundamental for the choice of the internship company have been:

* The website: it's the virtual facade of the company, and I believe that a neat website reflects a comfortable workplace, whereas a neglected website may indicate a suboptimal workplace;
* The face-to-face interview: the first human interaction I've had with Pagination S.r.l. was really professional and quite fascinating. I was also given some hints about the knowledge of my soon-to-be tutor on the technological domain of the internship, which made me feel more welcomed;
* The age of the employees: this might seem like an odd reason, but after having worked 3 years with people generally much older than me, the possibility of having younger colleagues seemed appealing. For example, my tutor is only 2 years older than me;
* The quick company response: after glancing at my previous experiences and open-source projects, Pagination S.r.l. proposed to hire me as an intern less than 2 hours after the first face-to-face interview.

1.4 Thesis Structure

* The second chapter, "Business Context", introduces the company and the business context in which it operates;
* The third chapter, "Technical Background", describes the main tech tools chosen for the internship project;
* The fourth chapter, "Requirement Analysis", presents an analysis of the requirements of the Queue Controller project;
* The fifth chapter, "Design and Development", shows how the project evolved from the design phase to the actual software crafting phase;
* The sixth chapter, "Deployment", presents how Terraform was used to automate the deployment of the Queue Controller artifacts on AWS;
* The seventh chapter, "Verification and Validation", deals with the processes that have certified the quality of the Queue Controller project;
* The eighth chapter, "Conclusions", presents a critic evaluation of the achievements of the internship experience at Pagination;
* The appendix chapters present some snippets of the code and scripts written during the internship. They aren't exhaustive, as the developed product is not meant to be open-source.

Chapter 2

Business Context

This chapter explains the business context in which the internship host company, Pagination S.r.l., operates. As a temporary member of the software development team, I had the chance to observe the business process in act at Pagination, in particular the ones concerning the technical area of the company.

2.1 Project Management

Not every software built by Pagination engineers is customer facing: some projects are also intended for internal use, such as utilities to set up or update the cloud infrastructure or to link code repositories to Continuous Integration pipelines.

Both internal and external processes share the following phases:

* **Viability**: before a new project can start, its technological and economic viability must be assured. In particular, for external projects, the type of documents that the customer want to generate, as well as the way the input data is structured have a huge impact on the viability analysis of the project. If a project isn't deemed feasible, it's immediately shut down.
* **Planning and Requirements**: employees of the administrative area conduct interviews with the customer to define a list of requirements for the project. When every part agrees with the requirements, a project schedule is produced.
* **Prototypation**: a prototype is the first deliverable product prepared by the graphic designers and the development team that is shared with the *stakeholders*. The stakeholders must evaluate whether the prototype satisfies the requirements, even if it's too raw to be considered final. Eventual adjustments must be made in this phase in order to continue with the next phase of the project.
* **Development**: the actual product is developed. Most of the time, the product is built upon the first prototype. During development, employees of the technical area take care of configuring the cloud infrastructure, and other configuration items, and write the code to satisfy the customer requirements. At the end of this phase, the software is validated.

  From what I have had the possibility to see, most external projects are forks of a single core product with some limited customizations.

* **Delivery**: as soon as the user requirements are satisfied and validated, the software is shipped to a private cloud sandbox and the customer is given a secure access to it. At this point, the users are taught how to use the software and the customer is ready to generate its own documents in complete autonomy.

  Some external projects are isolated, in the sense that the user requirements are fixed and don't need maintenance or improvements. In other cases old customers propose new requirements, which cause a new project iteration.
  For what concerns the internal projects, the delivery phase always requires that the new product be included in the internal business processes.

2.2 Development

Pagination's software development team mostly deals with planning and development of the *back-end* of the cloud service. The entire Pagination platform runs on Amazon Web Services, which is one of the most famous cloud providers, with new services and improvements delivered every month. Most business features are implemented using the *Serverless* paradigm with **AWS Lambda**; other software services use a more standard paradigm and are executed on an **AWS Elastic Compute Cloud (AWS EC2)** cloud instance. *Back-end* development happens mostly using the *Go* programming language and *Node.js*. Pagination also uses **AWS Simple Storage Service (AWS S3)** as its primary data storage service: it's used to store the input data and the template layouts submitted by the customers, as well as to store configurations shared between multiple internal projects.

2.2.1 Configuration Management

Configuration management is the set of versions of items that are part of a system and the procedures needed to coordinate the system components together. Configuration management focuses on identifying a system configuration in the various moments of its lifetime, in order to guarantee a systematic control of changes, as well as the integrity and traceability of the system.

Pagination S.r.l. uses *Git* as distributed versioning system, relying on *tag*s to keep track of different baselines of its software. The company uses *Amazon Code Commit* as *Git* service. It also uses *npm* and *Go modules* to manage, respectively, *Node.js* and *Go* third-party dependencies.

2.3 Maintenance

Maintenance is needed when an already developed product requires some changes. This may happen for three reasons:

* To correct a *bug* in the system (*corrective maintenance*);
* To adapt the system to changes in the execution environment (*adaptive maintenance*);
* To add new features to a system (*evolutive maintenance*).

If and when Pagination S.r.l. products need maintenance, the team returns to the development process in order to apply the required changes.

2.4 Support Processes Tools

2.4.1 Editors and IDEs

The following paragraphs present the two main editors and IDEs used at Pagination S.r.l.. Even though the company didn't force me to use any of them, I gladly did because I was already comfortable with both.

Visual Studio Code

Visual Studio Code is a multiplatform editor developed by *Microsoft*. It includes native support for debugging Node.js, offers an advanced *Git* integration, as well as code autocompletion. Pagination S.r.l. software development team mainly uses this editor during the development process.

IntelliJ GoLand

GoLand is a *Go* editor developed by *IntelliJ*. Before this internship, I mainly used *Visual Studio Code* for *Go* development. However, during the first week of internship I noticed that its official *Go* extension was broken and incompatible with the new *Go Modules* system introduced by *Go 1.11.0*; hence I adopted *Intellij Goland*.

2.4.2 Git and AWS CodeCommit

The company development team uses *Git* to version control the code it writes. Pagination S.r.l. relies on Amazon CodeCommit cloud service to host repositories. Since the company has less than 5 developers, it can profit from CodeCommit Free Tier.

I'd much rather use *GitHub* since it's a more feature-rich platform with an arguably better User Interface, but I must recognize that *CodeCommit* is well integrated with the *AWS* ecosystem.

2.4.3 Pancake

Pagination uses *Pancake* to keep track of projects, tasks and hours planned for each job. It allows to monitor the company productivity and manage projects. Moreover, it's able to export complete reports of the time expenses that can be forwarded to customers.

2.4.4 G Suite

Pagination S.r.l. uses Google's productivity suite, *G Suite*, which offers services like *Gmail for Business*, *Google Docs*, *Google Sheet* and *Google Drive*. These tools are used across the company to manage file sharing, collaborative document editing, and canned responses on support emails. The most important advantage of *G Suite* is that it's easy to use even for the less tech-savvy employees.

2.5 Agile Methodology

Agile Development is one of the most popular and modern software development life-cycle models. It focuses on process adaptability and customer satisfaction, and

is based on rapid delivery of a working software product. The Agile model requires multiple iterations. A typical iteration can last from one to two weeks and is called *sprint*. Agile requires initial high-level planning and design phases, but then new requirements can be added on the fly as soon as the required features are built, tested, and reviewed by the stakeholders.

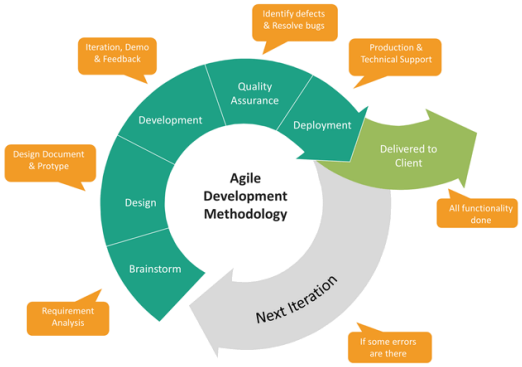


Figure 2.1: A visual representation of how the Agile development life-cycle.

I needed to select Agile as the development methodology as I quickly realized that many requirements would have been proposed by my tutor throughout the whole internship, and not only in the initial phase. In fact, the rationale for selecting Agile as the development methodology is due to its flexible nature. I also think that Agile can be particularly efficient when the development team is small, like this scenario.

Agile requires quick deliveries. Having decided to structure the project as a network of composable services exposed by a REST Application Program Interface, each new delivery implied adding some new API Endpoint that implemented a feature, or adjusting some modules behavior according to the newest specifications.

Chapter 3

Technical Background

This chapter introduces the main tools, programming languages, and frameworks used for the internship project, and explains why they were chosen.

3.1 Tools

In this section are listed the main tools I've used extensively throughout my internship experience.

3.1.1 Docker

Docker is a cross-platform tool that has revolutionized how software artifacts are distributed, offering guarantees of easy reproducibility on any system via software containers. Containers are standardized units of software that package up code and all its dependencies so the application is able to run quickly and reliably from one computing environment to another[1]. A *Docker* container image is a standalone software package that includes everything needed to run an application: system libraries, configurations, the code and the runtime.

Compared to virtual machines, containers follow the "Create once, use many times" ideology. After a container environment is defined in a special file called *Dockerfile*, it's available to be used when needed.

Nowadays Docker is a tool used by every top software company and has had a huge impact on the open-source community too. The main benefits that derive from the adoption of this containerization technology are:

* **Self-contained**: Docker containers are standalone, since they include everything needed for a seamless application execution. This allows developers to focus on the business logic of the application instead of worrying about compatibility issues a manual software installation in the system that will host it;
* **Easier deployments**: with Docker, a system administrator no longer needs to manually install dependencies in the system and link them all together, since they are already explicitly defined in the *Dockerfile* and are included in the *Docker* container environment;

[1] 12.

* **Faster than virtual machines**: Docker containers take only seconds to bootstrap and run the software they contain, whereas virtual machines take a significant amount of time (and generally have much more overhead too);
* **Widely used by the community**: choosing Docker brings the advantages related to the selection of a successful software technology. In fact, many popular applications and frameworks are available as containers and are available for download via public repositories.

3.1.2 Terraform

Terraform is a "tool for building, changing, and versioning infrastructure safely and efficiently"[2] which is used to manage public cloud provider services as well as custom in-house solutions. This tool is based on configuration files with a *.tf* extension defined using a custom syntax called HCL. Developers and DevOps specialists use these configuration files to describe every component needed to define a cloud environment. These configurations can then be fed into Terraform to generate an *execution plan*. In this phase, Terraform verifies the validity of the configuration code and summarizes the steps it will require to set up, update or tear down the described cloud infrastructure. Terraform uses a local backup to detect whether there are configuration changes and to create incremental execution plans accordingly.

The main features of Terraform can be resumed as:

* **Infrastructure as a Code**: every link between different services can be described using a high-level human-friendly configuration syntax;
* **Execution Plans**: before proceeding to alter or create the cloud infrastructure specified by the configuration files, Terraform lists every single step required to obtain the desired result;
* **Application environment control**: Terraform can be used to define the environment variables available at runtime in the applications running in the specified infrastructure;
* **Update Automation**: complex infrastructure updates can be applied with minimal human interaction, drastically cutting down the times compared to other solutions.

With Terraform, resources such as networking, DNS entries, and compute instances can be explicitly defined and saved. The main advantage of using configuration files to define a cloud infrastructure is that the same cloud infrastructure may be replicated by anyone in seconds. While toggling with the CLI/UI interface of a cloud provider may be tempting at first, especially when working with small projects, it leaves absolutely no documentation, nor trace of the steps needed to update or revert some configuration items. Terraform, instead, uses the configuration code as documentation, bringing easy reproducibility on the table. Moreover, the open-source community is continuously contributing with lots of small, composable packages that extend Terraform's features and the cloud support.

The development team at Pagination has had more than a year of usage experience with *Terraform*. Since this tool is recognized as the industry standard by the DevOps community and I had no previous experience with tools that automate cloud deployments, I agreed to use it when my tutor suggested it to me.

[2] 18.

3.1.3 Redis

Redis is an open-source in-memory data structure store which is primarily used as database, cache layer or message broker[3]. It doesn't support relations between its structures. It supports distributed storage via Redis Cluster, which provides built-in replication and automatic sharding and partitioning.

Redis offers a number of data structures based on the key-value concept. Since keys are used to identify a particular data structure, they can be compared to *primary keys* in *SQL* datastores. Redis keys are binary-safe, which means any binary sequence can be used as a key, as long as its size is less than 512 MB. Developers are often encouraged to define a certain schema to identify a key, such as *user:1000:followers*. Redis provides no means to enforce a particular schema on the keys it uses, so maintaining a proper key pattern consistency is up to the software that interacts with the database.

The most common Redis data structures are:

* **String**: it's the simplest type of value that can be associated with a Redis key. Both fields and values are saved as a string[4];
* **List**: data type that implements *linked lists* in Redis. Elements are sorted based on insertion order and saved as strings. From the algorithmic point of view, Redis lists suffer from the same drawbacks as the standard *linked lists*, so while appending an item to the head or the tail of the list requires constant time, accessing elements by index happens in linear time.[5];
* **Hash**: it's the equivalent of a key-value hash map data type. Hashes with few elements and small values are encoded in a special way that makes them extremely memory-efficient[6];
* **Set**: it's an unordered collection of strings. Sets support fast access to the middle of the collection. Redis also supports a sorted version of *Set*, called *Sorted Set*[7].

Different commands are required to retrieve data from a Redis database depending on the structures chosen for data representation. Redis supports Lua scripting: it's mainly used as a way to run multiple sequential commands as a single atomic operation.

The entire Pagination infrastructure relies on Redis. In fact, when I agreed to join the internship, the usage of Redis was one of the mandatory constraints.

This wasn't my first professional experience with Redis, as I had the chance to briefly work with it at Brainwise.

3.2 Cloud Services

Each service cited in the following subsections is provided by a single cloud provider: *Amazon Web Services*. Every cloud system developed by Pagination runs of *AWS*.

[3] 29.
[4] 35.
[5] 31.
[6] 30.
[7] 34.

One of the reasons why I chose this internship in the first place was to gain more proficiency with this cloud infrastructure.

3.2.1 AWS Lambda

AWS Lambda is a cloud computing service that executes code on demand as soon as some configurable events happen. The code execution may be automatically triggered from other AWS services or invoked directly as an HyperText Transfer Protocol endpoint from a web app. AWS Lambda supports code written in multiple programming languages and platforms, notably *Node.js*, *Java*, *Go*, *C#* and *Python*[8].

It uses the *Serverless* model, which is way different that the classic Infrastructure as a Service cloud computing model. In *IaaS*, the cloud provider only manages the low level of the system and the physical machines availability. Handling the high-level system infrastructure is up to the end user, which needs to take care of the number of virtual machines in use, their memory, storage and processor configuration, their operating system, load balancing, scalability and so on. With the *Serverless* model, instead, all these detail are handled by the cloud platform: the developer is only required to write the code for his software application. *IaaS* and *Serverless* have also drastically different pricing policies. With *IaaS* model, the user obtains the resources he needs and pays a subscription. Under the same conditions, the cost doesn't change whether the server is idle or not. In the *Serverless* model, however, costs are defined based on the time required to run the code.

The main benefits of AWS Lambda are:

* **No servers to manage**: AWS Lambda automatically runs the code without needing server management and provision;
* **Continuous Scaling**: it automatically scales the application by running each process in parallel, scaling precisely to the size of the workload;
* **Subsecond Metering**: the user is charged only for the time the code executes rounded up to the nearest 100ms, as well as the number of times the code is triggered. Contrary to the *IaaS* model, the user doesn't need to pay for idle servers.

Currently, Pagination S.r.l. uses *AWS Lambda* for free, since its monthly requirements fall under the AWS Lambda free tier. This free tier allows up to 1 million free requests per month and 400,000 GB-seconds of compute time per month with the default 128MB of RAM configuration.

3.2.2 AWS API Gateway

Amazon API Gateway is a scalable cloud service used to create, manage, monitor and secure APIs. It supports both REST and WebSocket APIs that expose entrypoints to access data, business logic, or features from back-end services. These back-end services can either be other AWS services (such as workloads running on Amazon EC2 or code running on AWS Lambda) or any other real-time web application[9].

A visual representation of how AWS API Gateway works is depicted in Figure 3.1.

[8] 6.
[9] 1.

Figure 3.1: A visual representation of how AWS API Gateway works.

API Gateway concurrently handles hundreds of thousands of API requests, dealing with traffic management, authorization and access control, processing, monitoring and API version management. The payment model for this service is only tied to the number of calls received and the amount of traffic transferred.

3.2.3 AWS CloudWatch

Amazon CloudWatch is the main monitoring service offered by AWS. It provides logs, metrics and event data for any AWS resource and service[10].

I've been using it to detect anomalous behaviors, inspect application logs and troubleshoot runtime issues.

3.2.4 AWS ECR

AWS Elastic Container Registry (AWS ECR) is a Docker container registry offered by Amazon that can be used to store, manage, and deploy Docker container images. Amazon also offers a quick deployment workflow to run images stored in ECR on Amazon ECS. In order to use ECR, the developers only have to pay for the amount of data stored and the data transferred to the Internet[11].

I've been using Amazon ECR to store the Docker image of the Redis instance used as database for the internship project. That instance was preloaded with about 2MB of realistic data dumped from the database used in production at Pagination.

3.2.5 AWS ECS

AWS Elastic Container Service (AWS ECS) is a high-performance container orchestration service that supports Docker containers that can execute and scale containerized applications on AWS. Amazon ECS relieves the developer from the need to install and administrate orchestration software, and manually manage and scale clusters of virtual

[10] 2.
[11] 3.

machines[12].

I've been using Amazon ECS to run the Redis database image stored in AWS ECR.

3.2.6 AWS Secrets Manager

Amazon Secrets Manager is a service to store and manage secrets needed at runtime by business applications or other services. With AWS Secrets Manager, the secrets can be easily rotated and retrieved via either a web UI dashboard or REST APIs[13].

I've been using it in conjunction with *AWS KMS* to retrieve database credentials and other sensitive or common information without needing to hardcode it in plain text.

3.2.7 AWS KMS

AWS Key Management Service (AWS KMS) lets the developers create and manage encrypted keys across most AWS Services. It uses hardware security modules to protect the stored keys[14].

I've been using the free-plan of AWS KMS to encrypt the Redis password stored in AWS Secrets Manager and decrypt it whenever the AWS Lambda microservices would start.

3.3 Programming Languages

This section presents the programming languages used in the *Queue Controller* project.

3.3.1 Go

Go (also known as *Golang*) is an *open-source* and multiplatform programming language released by Google in 2009[15]. *Go* syntax is clean, concise, and expressive, and quite similar to *C* or *Python*. *Go* is a compiled language with a dynamic type system whose main strengths are concurrency primitives and performance, which is on par with *C++*.

It has rapidly became the language of the cloud as most of the tools that empower the major cloud infrastructures are actually developed using *Go*. Example cloud-related technologies written using this language are Docker and Kubernetes, upon which many cloud services of the major three providers (*Google Cloud*, *Amazon Web Services*, *Microsoft Azure*) rely on[16].

Pagination asked me to choose a single programming language to develop the multitude of microservices needed for the internship project. While working with microservices may encourage using different programming languages according to the needs and the performance requirements, it may represent an obstacle to future developers, who may not necessarily know every language used, especially if the development team is as small as Pagination's.

[12] 4.
[13] 7.
[14] 5.
[15] 16.
[16] 10.

The company asked me to choose primarily between JavaScript (executed on the Node.js platform) and Go, with the discouraged option of proposing a different programming language too.

I decided to use Go due to the following reasons:

* Even though Pagination cloud platform was originally built using Node.js, the newest services built by the development team are being written solely in Go;
* I've been interested in Go since I was still in high school, but I've never had the possibility to use it at a professional level;
* I've been professionally using *Node.js* for more than 3 years, and I have also contributed to a number of open source Node.js library too, so I figured I'd rather spend some time improving my knowledge of a different language rather than sticking with my comfort zone;
* Go binaries generally consume much less memory than a Node.js interpreted application, which is an important point to consider when using a cloud service like *AWS Lambda*, since higher memory consumption leads to higher bills.

3.3.2 Lua

Lua is a fast and lightweight multiparadigm programming language. It supports dynamic typings and doesn't require compilation, as it is an interpreted language. Redis offers the possibility to run scripts written in a subset of the Lua programming language to interact with the database. The Lua code can either be sent by an inline script or as a script file, respectfully via the *EVAL* or *EVALSHA* Redis commands.

In Redis, the only way to perform complex operations and make them atomic is sending Lua scripts to the database. In fact, when a script is evaluated and executed, no other user can access the database. Redis also offers a powerful Lua script command-line debugger, which has been incredibly helpful to identify and fix some subtle bugs in the Lua code.

I didn't have any previous experience with the Lua language. It has been quite easy to learn, but it presents some differences in the way it evaluates data types, which can lead to errors for developers experienced in other dynamic languages like JavaScript.

3.3.3 TypeScript

TypeScript is an open-source "typed superset of JavaScript that compiles to plain JavaScript"[17]. It's built upon JavaScript's ECMAScript 2015+ specification, adding support to interfaces, method access modifiers (via the *private*, *protected* and *public* keywords), generics, namespaces and so much more. It has gained a lot of traction in the JavaScript community due to these main benefits:

* **Huge community**: Typescript is built and backed by **Microsoft**, is very actively maintained and extremely appreciated and used by the community. According to StackOverflow's 2019 survey, *TypeScript* is the third most loved programming language (behind *Rust* and *Python*) and is used by 23.5% of professional developers[18].

[17] 28.
[18] 36.

* **Enhanced readibility**: while the *Java*-like *TypeScript* syntax may be considered a little verbose, it has the beauty of being extremely explicit and clear.
* **Type safety**: explicit types prevent type errors, which are erroneous or undesirable program behaviors caused by discrepancies between differing data types;
* **Advanced Intellisense**: TypeScript type system has greatly improved the autocomplete features provided by IDEs;
* **100% compatible with JavaScript runtimes**:

Unfortunately, unlike completely type-safe languages like Java, TypeScript doesn't provide a runtime type checker, since its type safety can only be analyzed statically before transpiling the TypeScript source code to JavaScript.

I've decided to use *TypeScript* for writing the front-end of the internship project because:

* Both in my tutor's point of view and mine, a typed language is more prone to raise the quality of a project and eases the comprehension of the code for future developers of the project;
* Using TypeScript is often a requirement in many Front-end or Full Stack Developer job entries;
* I have a considerable professional experience using it, having adopted it since version 1.0.0, and I believe it's the "right tool for the job".

3.4 Frameworks

This section briefly discusses the software frameworks used during the internship and why they were chosen.

3.4.1 React.js

React.js is an open-source library used to build user interfaces in JavaScript[19]. It's maintained by Facebook and used by many top companies, such as Netflix[20], AirBnb, Dropbox and WhatsApp web[21]. React.js promotes a component-based approach to build user interfaces, and its team has created a new syntax to directly manipulate HyperText Markup Language from JavaScript, called JSX. It also features lots of utilities to manipulate the application state,

For what concerns the frameworks needed to create the front-end part of the project, I've been told to choose any of the most popular JavaScript-based framework: React.js, Angular, and Vue.js. I've decided to use React.js because:

* React.js is the only front-end framework used at Pagination;
* Most job offers require React.js experience;
* It officially supports *TypeScript*;

[19] 14.
[20] 13.
[21] 23.

Listing 3.1: JSX syntax example.

```
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
  </div>
);
```

Listing 3.2: Example of a unit test written in TypeScript using Jest.

```
import PaginatedListCacheManager from '../
    PaginatedListCacheManager';

describe('PaginatedListCacheManager', () => {
  let offset: number;
  let cacheManager: PaginatedListCacheManager<number>;

  beforeAll(() => {
    offset = 10;
    cacheManager = new PaginatedListCacheManager(offset);
  });

  it(`A fresh PaginatedListCacheManager should
      have 0 as nextIndex`, () => {
    expect(cacheManager.getNextIndex()).toBe(0);
  });
});
```

* Having used both frameworks for years at a professional level, I strongly prefer using React.js' *JSX* syntax rather than *Angular*'s templates to create dynamic *HTML* codes;
* React.js is not only the framework I'm most comfortable with, but it's also the one I enjoy using the most, both as a professional and hobbyist developer;
* I've been following its changes since version 0.13.0 (at the moment of writing, React.js has reached the 16.9.0 version). I have a deep admiration for the React team, because I've seen directly how features implemented in React.js have had an impact on the whole JavaScript community and beyond.

3.4.2 Jest

Jest is a testing framework for JavaScript developed by Facebook. It provides a rich, intuitive, and well documented assertion API, high performance via parallel processes, and its exceptions provide rich contextual details.

I have a couple of years of experience using Jest, and I found it's the most mature open-source JavaScript testing framework, as well as one of the most famous. Since Pagination left me freedom of choice, I opted for Jest as testing framework for the front-end application.

3.4.3 Ginkgo

Before starting my internship at Pagination, I didn't know any Go testing frameworks beside the standard (and quite limited) Go testing library. I have therefore accepted my tutor's suggestion and adopted *Ginkgo* to perform Unit Tests and Integration Tests on the Microservices written in *Go*.

Chapter 4

Requirements Analysis

This chapter presents part of the requirements analysis performed for the Queue Controller project. Due to secrecy reasons, I'm not allowed to reveal every aspect of the project.

4.1 Use cases

The formal language used to express the intent of each requirement and the relations between them is Unified Modeling Language. In particular, UML's Use Case diagrams have been used. *Use Case diagrams* are high-level UML diagrams dedicated to describe which are the services offered by a system to its end users. Use Case Diagrams describe how a system's features are perceived and used by the actor that interacts with the given system. Since the project doesn't require any authentication at all, and given that there's only a single main actor, the number of Use Cases is limited and there is no Use Case dedicated to user registration, authentication and authorization.

Each use cases is classified according to the following convention:

UC[Parent Code].*[ID Code]

* **Parent Code**: the ID of the generic use cases that generated the considered use case. If the use case isn't generated by other use cases, the *Parent Code* should be omitted;
* **ID Code**: it identifies the use case and is made of numbers only.

4.2 Actors

4.2.1 Primary Actors

A primary actor is a user that interacts with the system to achieve a goal. In Queue Controller, there's a single primary actor:

* **Company User**: Possibly non tech-savvy employee of the company that wants to interact with the system.

Secondary Actors

A secondary actor is a user that interacts with the system to help him reaching the goals of the primary actor. In Queue Controller there's no secondary actor.

4.3 Use Case Diagrams

The following paragraphs will present the UML Use Case diagrams for the Queue Controller project. The possible extensions are cited in the general use case diagram only (4.3.1).

4.3.1 UC0: General Use Case Diagram

Main Actors: Company User.

Description: The company user gains access to the system features.

Preconditions: The system offers the following features to the company user:

* UC1: Show jobs result lists;
* UC2: Show the list of completed pagination jobs;
* UC3: Show the list of pagination jobs with errors;
* UC4: Show the list of deleted pagination jobs;
* UC5: View error "Pagination job does not exist";
* UC6: Stop in progress pagination jobs;
* UC7: Delete a pagination job;
* UC8: Acquire a lock on a pagination job;
* UC9: Show list of pagination locks;
* UC10: Force delete any pagination lock;
* UC11: Delete a pagination lock;
* UC12: Force delete a pagination lock;
* UC13: Delete a pagination lock with a token authorization;
* UC14: View error "Unauthorized to delete lock";
* UC15: View error "Lock does not exist";
* UC16: Update pagination lock expiration timeout with a token authorization.

.

Postconditions: The system has offered the features to the company user.

Main Scenario:

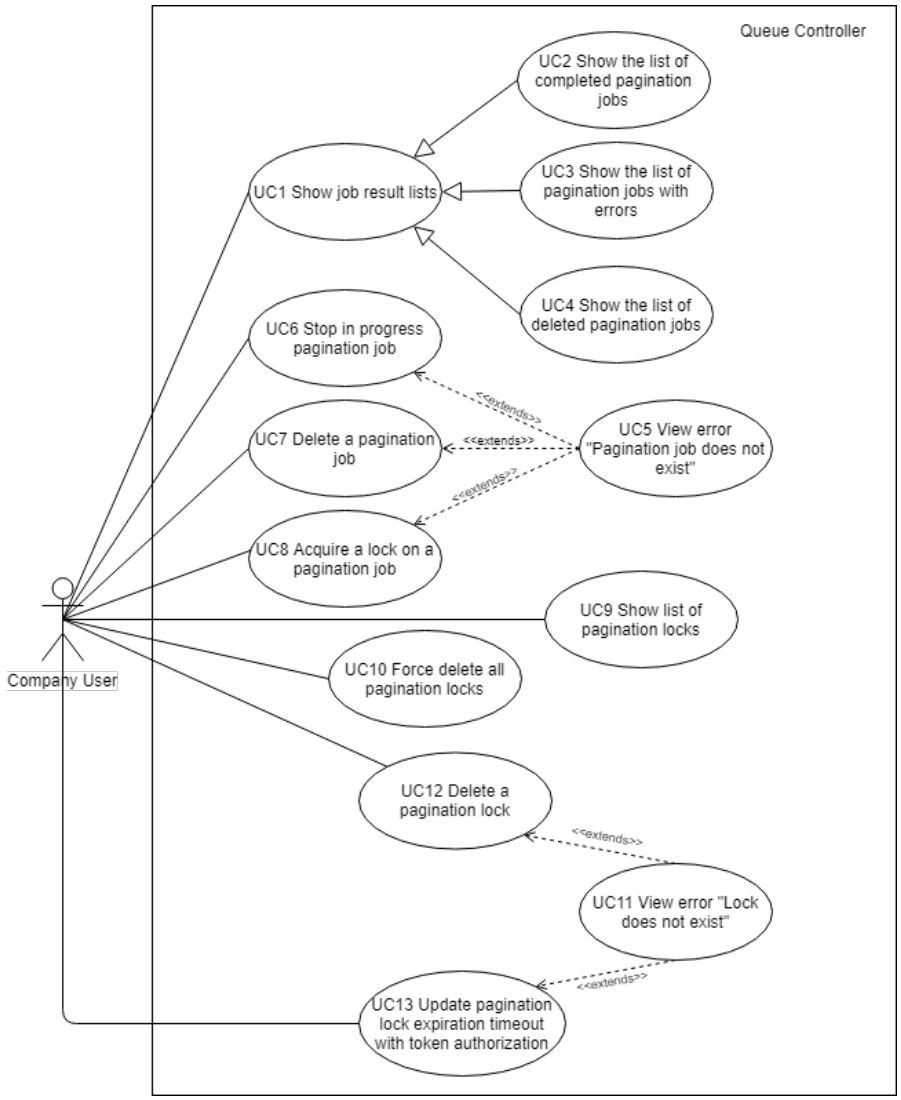


Figure 4.1: UML diagram of UC0: General Use Case Diagram

1. The company employee opens the user interface to interact with the Queue Controller system.

Extensions:

1. If the user tries to stop a pagination job that doesn't exist, an error message should be returned (UC5);
2. If the user tries to delete a pagination job that doesn't exist, an error message should be returned (UC5);
3. If the user tries to acquire a lock on a project that doesn't exist, an error message should be returned (UC5);

4. If the user tries to delete a pagination lock on an unlocked project, an error message should be returned (UC15);
5. If the user tries to delete a pagination lock with token authorization using a token that doesn't match with the lock's token, an error message should be returned (UC14).

4.3.2 UC1: Show job result lists

Main Actors: Company User.

Description: The company user must be able to view the list of pagination job results. They can be in three states: completed, deleted, or with errors.

Preconditions: The company user decides to view the lists of pagination job results.

Postconditions: The system allows the company user to view the list of pagination job results.

Main Scenario:

1. The company user selects a stage;
2. The company user selects the type of job result list to view:
   * Completed (UC2);
   * With errors (UC3);
   * Deleted (UC4);
3. The list of jobs of the selected type is presented to the user;
4. The user can select an item of the list to view its details.

4.3.3 UC2: Show the list of completed pagination jobs

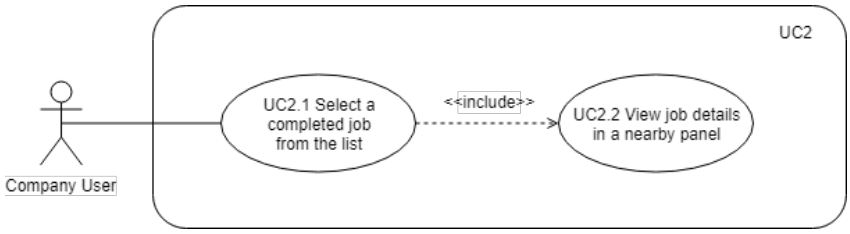


Figure 4.2: UML diagram of UC2: Show the list of completed pagination jobs

Main Actors: Company User.

Description: The company user must be able to view the list of completed pagination

job results.

Preconditions: The company user decides to view the lists of completed pagination job results.

Postconditions: The system allows the company user to view the list of completed pagination job results.

Main Scenario:

1. The company user selects a stage;
2. The company user selects to view the completed pagination job results;
3. The list of completed pagination job results is presented to the user;
4. The user can select an item of the list to view its details.

4.3.4 UC3: Show the list of pagination jobs with errors

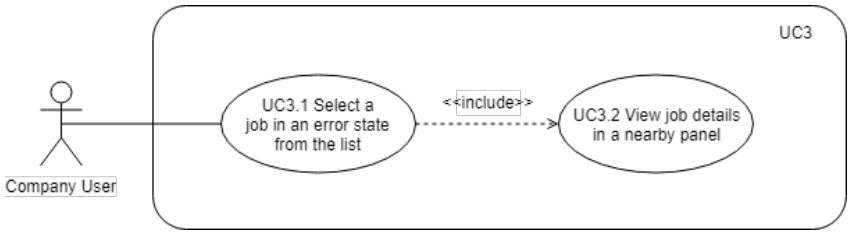


Figure 4.3: UML diagram of UC3: Show the list of pagination jobs with errors

Main Actors: Company User.

Description: The company user must be able to view the list of pagination job results interrupted due to processing or parsing errors.

Preconditions: The company user decides to view the lists of pagination job with errors.

Postconditions: The system allows the company user to view the list of pagination jobs in an error state.

Main Scenario:

1. The company user selects a stage;
2. The company user selects to view the completed pagination job results;
3. The list of completed pagination job results is presented to the user;
4. The user can select an item of the list to view its details.

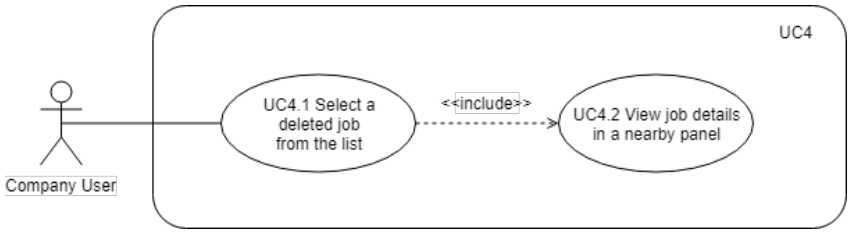


Figure 4.4: UML diagram of UC4: Show the list of deleted pagination jobs

4.3.5 UC4: Show the list of deleted pagination jobs

Main Actors: Company User.

Description: The company user must be able to view the list of pagination job results that were deleted in the previous hours.

Preconditions: The company user decides to view the lists of recently deleted pagination jobs.

Postconditions: The system allows the company user to view the list of pagination jobs that were deleted recently.

Main Scenario:

1. The company user selects a stage;
2. The company user selects to view the completed pagination job results;
3. The list of completed pagination job results is presented to the user;
4. The user can select an item of the list to view its details.

4.3.6 UC6: Stop in progress pagination job

Main Actors: Company User.

Description: The company user decides to suspend an in-progress pagination job.

Preconditions: The system allows the company user to suspend an in-progress pagination job.

Postconditions: The system has stopped a pagination job and the UI no longer offers the possibility to stop that job.

Main Scenario:

1. The company user selects a stage;
2. The company selects a job list to view;
3. The company user selects the job to suspend.

Extensions:

1. If the user tries to stop a pagination job that doesn't exist, an error message should be returned (UC5).

4.3.7 UC7: Delete a pagination job

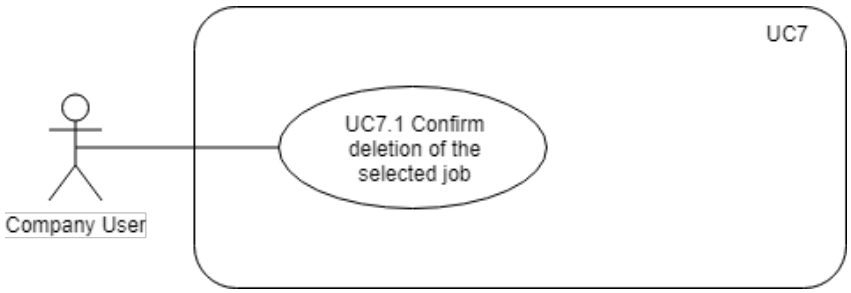


Figure 4.5: UML diagram of UC7: Delete a pagination job

Main Actors: Company User.

Description: The company user decides to delete a pagination job.

Preconditions: The system allows the company user to delete a pagination job.

Postconditions: The system has deleted the specified pagination job and the UI no longer displays it in the job list or in the job detail view.

Main Scenario:

1. The company user selects a stage;
2. The company selects a job list to view;
3. The company user selects the job to delete;
4. The company user confirms that the selected job must be deleted.

Extensions:

1. If the user tries to delete a pagination job that doesn't exist, an error message should be returned (UC5).

4.3.8 UC8: Acquire a lock on a pagination job

Main Actors: Company User.

Description: The company user decides to acquire a lock on a pagination job to prevent other users to perform operations on the locked job for a certain amount of time.

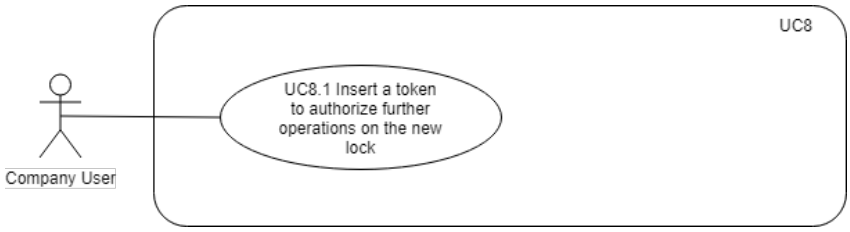


Figure 4.6: UML diagram of UC8: Acquire a lock on a pagination job

Preconditions: The system allows the company user to create a lock on a pagination job.

Postconditions: The system has created a lock on the specified pagination job.

Main Scenario:

1. The company user selects a stage;
2. The company user lists the pagination jobs;
3. The company user selects the job to lock;
4. The company user writes a possibly empty token that can be used later to perform further operations on the lock that will be created.

Extensions:

1. If the user tries to acquire a lock on a project that doesn't exist, an error message should be returned (UC5);

4.3.9 UC9: Show the list of pagination locks

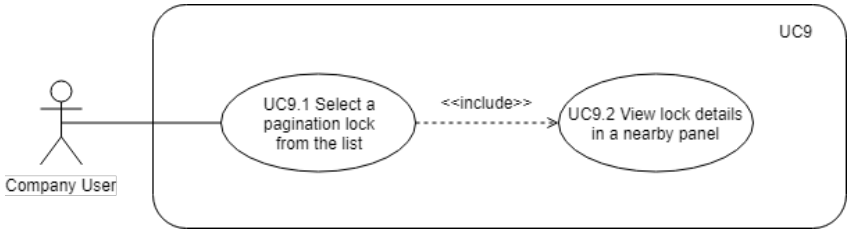


Figure 4.7: UML diagram of UC9: Show the list of pagination locks

Main Actors: Company User.

Description: The company user decides to view the list of pagination locks in a certain stage. If no lock is present, an empty list should be returned.

Preconditions: The system allows the company user to view the list of pagination locks in a certain stage.

Postconditions: The system has viewed the list of pagination locks in the specified stage.

Main Scenario:

1. The company user selects a stage;
2. The company user selects the job to lock;
3. The company user writes a possibly empty token that can be used later to perform further operations on the lock that will be created.

4.3.10 UC10: Force delete all pagination locks

Main Actors: Company User.

Description: The company user decides to unlock every pagination project in a certain stage. If no lock exists, the system doesn't change its state.

Preconditions: The system allows the company user to delete every pagination lock in a certain stage.

Postconditions: The system has deleted every pagination lock in the specified stage.

Main Scenario:

1. The company user selects a stage;
2. The company user deletes all locks in a single action.

4.3.11 UC12: Delete a pagination lock

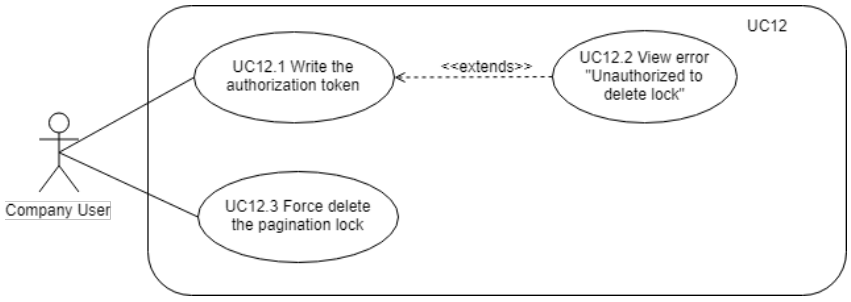


Figure 4.8: UML diagram of UC12: Delete a pagination lock

Main Actors: Company User.

Description: The company user decides to unlock a single pagination job in a specific stage.

Preconditions: The system allows the company user to delete the selected pagination lock.

Postconditions: The system has deleted the selected pagination lock.

Main Scenario:

1. The company user selects a stage;
2. The company user selects a lock to remove;
3. The company user either forces the deletion of the selected lock, or inserts the appropriate token to authorize the token deletion.

Extensions:

1. If the user tries to delete a pagination lock that doesn't exist, an error message should be returned (UC11).

4.3.12 UC13: Update pagination lock expiration timeout

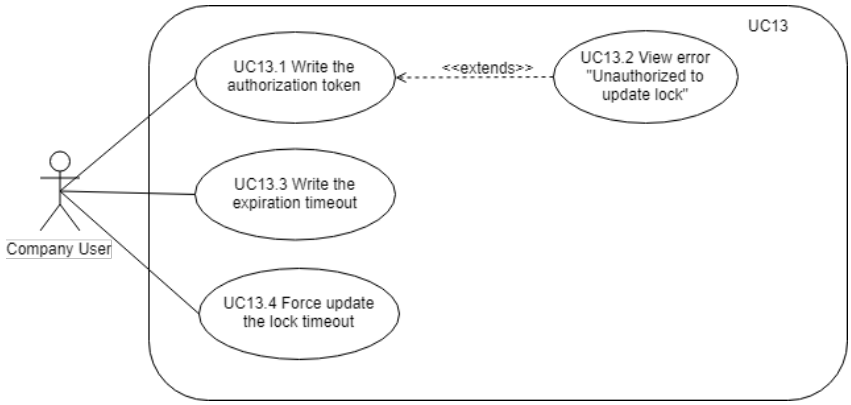


Figure 4.9: UML diagram of UC13: Update pagination lock expiration timeout

Main Actors: Company User.

Description: The company user decides change the timeout after which a particular locked project is automatically unlocked.

Preconditions: The system allows the company user to update a pagination lock in a certain stage.

Postconditions: The system has updated the expiration timeout of the selected pagination lock.

Main Scenario:

1. The company user selects a stage;
2. The company user views the lock list;

3. The company user selects a lock item from the list;
4. A default timeout is displayed to the company user, which can be modified in a detail panel;
5. The company user can either write the correct token to authorize the update operation on the lock or force the update operation.

4.4 Requirements

The presented requirements can be distinguished in the following categories:

* Functional requirements;
* Constraint requirements;
* Performance requirements;
* Quality requirements.

The identification code is the string that identifies any requirement and follows this particular notation:

R[Typology][Importance]-[Code]

* **Typology**:
  - **F**: it indicates a *Functional* requirement;
  - **C**: it indicates a *Constraint* requirement;
  - **Q**: it indicates a *Quality* requirement.
* **Importance**:
  - **M**: it indicates a *Mandatory* requirement;
  - **D**: it indicates a *Desirable* requirement;
  - **O**: it indicates an *Optional* requirement;
* **Code**: it is a progressive number expressed in a hierarchical fashion. It's composed by:

  [BaseID](.[SubCaseID])*

  - **BaseID**: code which, combined with *Typology*, identifies a general requirement;
  - **SubCaseID**: progressive code which may or may not be added that identifies possible subcases or specializations of a general requirement.

The sources used to decide the user and system requirements are the following:

* *Work Plan*: it denotes a requirement that was already defined in the Work Plan prepared before starting the actual internship.;

* *Use Cases diagram*: it indicates a requirement that emerged during the preparation of the Use Case diagrams, and should indicate the particular use case to which a particular requirement refers to;
* *Tutor*: it denotes a requirement that was either imposed by the company or suggested by the tutor;
* *Personal Decision*: it indicates a self-imposed requirement, that has been deemed coherent with the overall nature of the project by the tutor.

The requirements identified for the project are shown below with their respective identification code, brief description and source.

4.4.1 Functional Requirements

Code	**Description**	**Source**
RFM-1	The system should make the user understand whether the remote database is reachable or not	Tutor
RFM-2	The user must be able to see the list of queue jobs being currently processed	Tutor
RFM-3	The user must be able to delete every job being currently processed in a particular stage	Tutor
RFD-4	The user must be able to see the list of queue jobs that had errors	Tutor, Work Plan
RFM-5	The user must be able to delete every job that had errors in a particular stage	Tutor
RFM-6	The user must be able to see the list of queue jobs deleted recently	Tutor
RFM-7	The user must be able to configure the timeout after which a deleted job expires	Tutor
RFM-8	The user must be able to view the details of a single job	Tutor
RFM-9	The user must be able to delete a single job	Tutor
RFO-10	The user must be able to edit the priority of a single job	Tutor
RFM-11	The user must be able to list the locks in a particular stage	Tutor
RFM-12	The user must be able to acquire a lock on a single project	Tutor
RFM-12.1	The acquired lock token must be persisted on the client in order to authorize subsequent operation on locks	Tutor
RFM-13	The user must be able to release a lock on a project	Tutor
RFM-13.1	The user should provide the token of the lock he's attempting to release in order to authenticate himself	Tutor
RFM-13.2	The user may ask the system to force the lock release even if the lock token isn't known	Tutor

Table 4.1 continued from previous page

Code	Description	Source
RFM-14	The user must be able to refresh the timeout on an already acquired lock	Tutor
RFM-14.1	The user should provide the token of the lock he's attempting to refresh in order to authenticate himself	Tutor
RFM-14.2	The user may ask the system to force the lock release even if the lock token isn't known	Tutor
RFD-15	The user should be able to see the JSON content of a job	Tutor
RFD-15.1	The JSON content of a job's split should be sintax-highlighted on the Web UI	Tutor

Table 4.1: Functional requirements tracking table.

4.4.2 Constraint Requirements

Code	Description	Source
RCM-1	It must be possible to interact with the software product via a web interface	Tutor
RCM-1.1	The user must access the web app via the internal Pagination network	Tutor
RCM-2	Front-end framework decision	Work Plan
RCD-2.1	The front-end web app should be developed using either React.js, Angular or Vue.js	Tutor
RCO-2.2	The front-end web app must be developed using React.js	Personal Decision
RCM-3	Browser support	Tutor, Personal Decision
RCM-3.1	The user must be able to browse the web app using at least Google Chrome v74	Tutor
RCO-3.2	The user may be able to browse the web app using Microsoft Edge v44	Personal Decision
RCD-4	The system should be deployed on AWS	Tutor
RCM-5	The system must use AWS Lambda functions to define the product core logic	Tutor
RCM-6	The system must use AWS API Gateway to aggregate the various AWS Lambda functions	Tutor
RCM-6.1	Each AWS API Gateway endpoint should be paired with a unique AWS Lambda function	Tutor
RCM-7	The product source code and its documentation must be stored in AWS CodeCommit	Tutor
RCM-8	The product source code must be versioned using Git	Tutor
RCM-9	Back-end code language decision	Personal Decision, Work Plan
RCM-9.1	The language used to define the AWS Lambda functions must be either Go, Node.js or both	Work Plan
RCM-9.2	The unique language decided to be used to define the AWS Lambda functions is Go	Personal Decision
RCM-9.2.1	The intern must use at least Go v12.0.0	Personal Decision

Table 4.2 continued from previous page

Code	Description	Source
RCM-9.2.2	The Go code must rely on the Go Modules feature of Go	Personal Decision
RCM-10	The back-end should expose a REST API interface	Work Plan
RCO-11	The back-end may also expose a GraphQL interface	Personal Decision
RCM-12	The structures that define the public contract of each AWS Lambda function should be written using ProtoBuffers	Tutor, Personal Decision
RCM-12.1	The version of ProtoBuffers to use is v3	Personal Decision
RCM-13	The project code should be compiled in a virtual environment using Docker	Personal Decision
RCM-13.1	Each microservice should have a related Dockerfile used to build that single service	Personal Decision
RCM-13.2	The build process should also generate a zip file ready to be sent to create or update the related AWS Lambda function	Personal Decision
RCM-14	The project should be tested in a virtual environment using Docker	Personal Decision
RCM-15	The back-end must interact with a Redis backend	Tutor
RCM-15.1	The back-end must interact with a Redis backend hosted on AWS and accessible by the AWS lambda functions via a private AWS VPC subnetwork	Tutor
RCM-16	The Redis operations that cannot be executed in concurrency with other users should be defined in Lua scripts, whose content must be sent to the Redis database	Tutor, Personal Decision
RCM-16.1	To reduce costs, the Lua scripts cannot be read at runtime by the Lambda function	Tutor
RCD-16.2	The Lua scripts shouldn't be manually defined inline in the Go files	Personal Decision
RCM-16.3	The Lua scripts can be defined in separate Lua files and be included in the Go executable at compile time using Go Generate	Personal Decision
RCM-16.4	The Lua scripts' content must be minimized and stripped of comments before being sent to Redis	Tutor
RCM-16.5	The Lua scripts' content must be minimized and stripped of comments at compile time, without overriding the original Lua script	Personal Decision
RCM-17	The REST API communication should use the JSON format	Tutor
RCD-18	The front-end should use the component react-ace to display code highlighting of the content of a Job	Tutor

Table 4.2 continued from previous page

Code	Description	Source
RCD-19	The intern may use Terraform to automatically deploy lambda functions on the AWS cloud	Tutor

Table 4.2: Constraint requirements tracking table.

4.4.3 Quality Requirements

Code	Description	Source
RQD-1	The product should follow the 12 Factor Apps principles	Personal Decision
RQD-2	The project's repository should adhere to the standard Go project layout	Tutor
RQM-3	The intern should create a manual to document how to use and extend the application	Work Plan
RQM-4	The REST endpoints of the product should be documented in the OpenAPI format	Personal Decision
RQM-4.1	The REST endpoints documentation should also be exported using Redoc	Tutor
RQM-4.1.1	The intern should use Redoc to export a human-readable, styled HTML file, given an OpenAPI compatible file	Tutor
RQD-5	The code written in Go should take at least an A+ grade with the static analysis tool GoReport	Tutor
RQM-6	Each public variable or function must contain a meaningful comment	Tutor
RQM-7	The Go code must be verified using the standard Go lint tool	Personal Decision
RQM-8	Each public function should have at least a meaningful unit test related	Personal Decision
RQD-9	Each Lambda function should have at least a meaningful integration test related	Personal Decision
RQO-10	Continuous Integration should be set up for the product	Personal Decision, Work Plan
RQO-10.1	The Continuous Integration agent to be used should be Jenkins	Tutor
RQM-11	The code that contains the core logic of the product must be separated from the code that handles the Lambda Functions	Personal Decision, Tutor
RQM-12	The intern should prepare a document in which he presents the overall architecture and the design patterns used in the system	Work Plan

Table 4.3: Quality requirements tracking table.

4.4.4 Summary

A summary of the project requirements is presented in Table 4.4.

Type	Total	Mandatory	Desirable	Optional
Functional	21	17	3	1
Constraint	38	30	5	3
Quality	15	9	4	2

Table 4.4: Summary of the project requirements.

Chapter 5

Design and Development

This chapter presents and explains the architecture of the components that belong to the Queue Controller project, and how they were implemented during the internship.

5.1 System Design

The Queue Controller project is based on the client-server architecture. The client application, also called front-end, is a Single Page Application written in React.js and TypeScript. On the other hand, the server, also called back-end, is composed of several microservices exposed as API endpoints to the client. Client-server communication makes use of the common *JSON* format to interchange data: it's widely used in the industry and it's natively supported by many programming languages.

Each microservice is defined in an AWS Lambda function. These functions aren't directly exposed as they are: they are instead mapped to REST API endpoints. This job is performed by AWS API Gateway, and has little to no overhead in terms of the performance of the system.
Each of these functions requires to connect to configuration servers and to the Redis database during their lifetime. The Redis database lives in a private region of the cloud that isn't directly exposed to Internet. That same Redis database is read and updated by the Pagination's core product pipeline, and it's the only resource shared with Queue Controller.

The most important system details are presented in a top-down approach in the following sections. The back-end and front-end of the Queue Controller project are studied separately. Section 5.2 treats the architecture and the subsystems that form the back-end of Queue Controller, whereas section 5.3 discusses the structure of the client-side web application, i.e. the front-end.

5.2 Back-end

I decided that the most appropriate server-side architecture for this project would be Microservice architecture. Microservices are often considered the opposite of the more standard monolithic architecture, where a system is made of highly coupled components residing in a single application.

A monolithic architecture presents many problems that eventually lead to higher code debt:

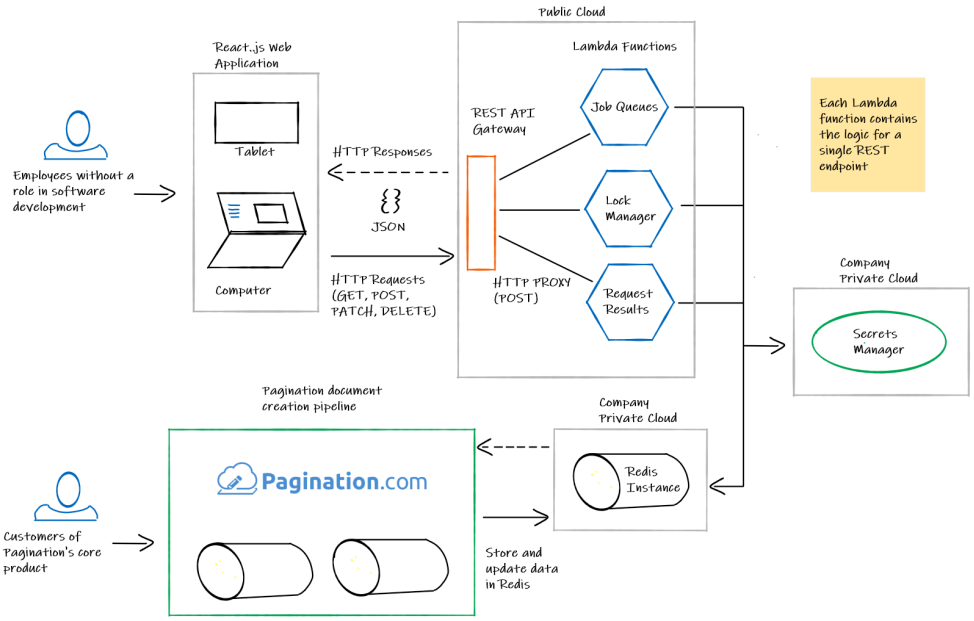


Figure 5.1: End-to-end view of the Queue Controller system.

* Tight coupling between components, as everything is contained in a single application;
* It's difficult to perform maintenance, as the source files often contain too many lines of code and it may be unclear where a component responsibility starts and ends;
* Slow build and deployment processes, as the system must be built and delivered as a whole;
* Code duplication, since the reusability of code is reduced to the minimum;
* Slower startup times.

Microservice architecture, on the other hand, treats the system as multiple independent services that cooperate with one another, each of which solves a single problem and offers a specific feature. Microservices are loosely coupled and follow the **Single Responsibility Principle** of SOLID. Services can communicate using either synchronous protocols such as REST over HTTP, or asynchronous protocols like Advanced Message Queuing Protocol. These characteristics give the possibility to develop and deploy microservices independently of one another.

The Microservice architecture possesses a lot of advantages:

* Better code modularity: since the single microservices are independent, the system is loosely coupled and the app is quicker to develop, test and improve;
* Microservices are, as the name suggest, relative small. This is due to the fact that their purpose is very limited;

* The app can scale according to the needs. While in a monolithic architecture there's no way to scale a component that causes bottlenecks since it's too entangled with other services, with microservices an intervention targeted to increase performance on a slow service is possible. Furthermore, the single services may be configured to run on special hardware or particular operative systems whenever needed;
* They increase the productivity of a development team. Having multiple microservices at stake gives the possibility to better parallellize the work of the team, as any developer can fiddle with a single service and re-deploy it without risking to interfere with the job of the colleagues;
* It's easier to update the system without downtime. In fact, when using microservices there's no need to re-deploy the entire system for every code change, as it's sufficient to update the interested service. It's also possible (and suggested) to run the new version of the system alongside the old one and progressively migrate the user requests to new version;
* The system is more resilient and the errors are sandboxed, since if there's an error, a single microservice is faulted, whereas the other ones continue to work as they should. This means that even with errors the system isn't completely halted;
* The build and deployment processes take much less time, given the possibility to use Continuous Delivery;
* The system isn't bound to a single technology stack. Microservices can be developed with different tools and languages, so the developers aren't bound to technological choices made in the past.

However, Microservices carry their own set of drawbacks:

* Develop a distributed system is generally more challenging and complex;
* Testing the interactions between services is more difficult;
* Developer tools are generally oriented on building monolithic applications and don't provide explicit support for developing distributed applications:
* Continuous Delivery processes may be difficult to set up for the first time;
* Increased memory consumption, as the microservice architecture replaces N monolithic application instances with NxM services instances. If each service runs in its own virtual machine or Docker container, which is usually necessary to isolate the instances, then there is the overhead of M times as many virtual runtimes.

I decided to use Microservice architecture because it fits Queue Controller particularly well: different components perform a single feature that's independent to the other modules. Each service is accessed via its own REST endpoint configured with AWS API Gateway.

5.2.1 About Redis and AWS VPC

Given the very limited number of users of the Queue Controller application, there's currently a single Redis database in the system. For security reasons, it isn't exposed to the Internet directly. Instead, it resides in an AWS VPC, which is a logically isolated area of the AWS Cloud where can be run in a virtual network. Services outside a VPC can only communicate with services in the private network by establishing a VPN connection with an AWS-managed virtual private gateway.

Lambda functions can be connected to VPC to access private resources during their execution. By default, AWS Lambda runs the function code securely within a VPC. In order to authorize a Lambda function to access resources inside a private VPC, some additional VPC-specific configuration information must be provided to AWS, such as the list of private subnet identifiers and security group identifiers. This information is used by AWS Lambda to set up Elastic Network Interfaces (ENIs) that enable a Lambda function to connect securely to other resources within the specified private VPC[8].

Every Lambda function in Queue Controller connects to the Redis database running on Amazon ECS inside the VPC. VPCs can be shared between multiple services. In fact, for cost-related reasons I didn't create any new private cloud, I simply used an existing VPC configured by Pagination.

5.2.2 Secrets Management

Secrets are credentials like passwords, SSH-keys or API-keys that a service needs to know to authenticate and communicate with other services. Even in a small-to-medium cloud infrastructure with only a dozen of microservices, it's easy to imagine how the number of secrets can increase along with them. As the number of microservices grows, managing and control secret credentials can only become more impractical.

A centrally automated secret management solution is offered by Amazon Secrets Manager. It's a service that offers easy rotation, management and retrieval strategies for secret credentials. Web services that need to retrieve secrets can use the Secrets Manager APIs, eliminating the need to hardcode sensitive information in plain text. Configurations and passwords should be as much decoupled from the code as possible, since they can change frequently.

Another alternative to a secrets management system is using system environment variables. They are much less secure and trivially accessible by every process running on a system, but eliminate the complexity and the latency related to establishing a secure connection to a service like AWS Secrets Manager. The Pagination development team uses the following question-based approach to decide when they should use a secret management solution or when using environment variables is fine:

* Should the string accessed at runtime be used as configuration parameter, for example to specify a feature flag or an immutable URL to call? Use environment variables;

* Should the string accessed at runtime be used as a credential or authorization token, or an URL that may vary frequently (for example, a VPC subnet IP)? Use a solution like AWS Secrets Manager.

As a temporary member of the company's development team, I used the latter approach during the internship.

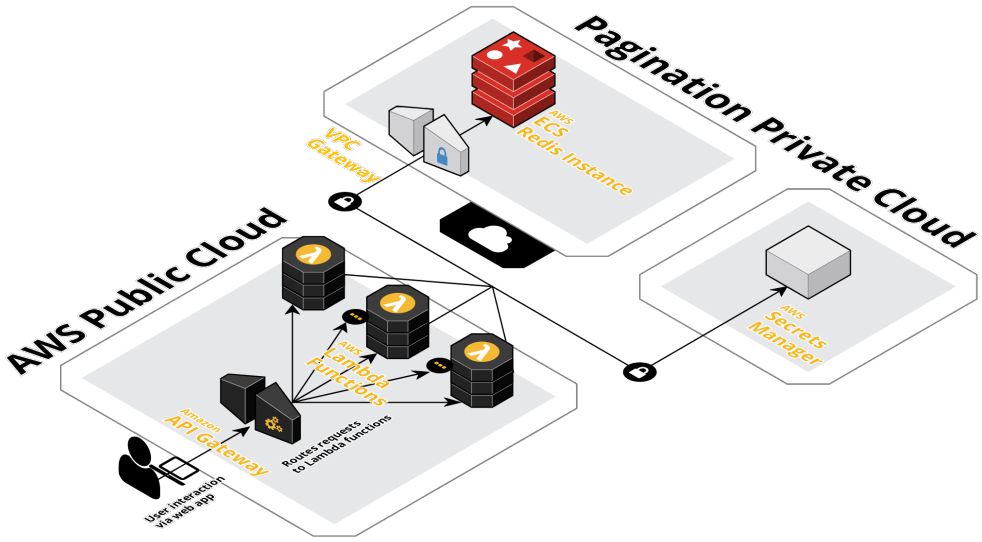


Figure 5.2: Queue Controller cloud infrastructure.

The back-end of the project exposes a REST API interface via Amazon API Gateway.

As written in 3, the back-end is composed of serverless microservices running on AWS Lambda.

5.2.3 Protocol Buffers

Protocol Buffers are a language and platform-neutral serialization mechanism for structured data developed at Google; it can be imaged thought of as smaller, faster and simpler alternative to Extensible Markup Language or JavaScript Object Notation. From a simple data definition written in the Protobuf specification language, and highly optimized source code can be generated to easily write and read that structured data to and from a variety of data streams[17]. Protobuf models are defined in files with the *.proto* extension, which once transpiled to *Go* source files have the *.pb.go* file extension.

Protocol Buffers can only be used in server-to-server communications and are often used in Remote Procedure Call systems for their performance benefits over *JSON*, which is slower to parse. Even though they're overkill for this project as their potential cannot be fully expressed in *REST* communications, the tutor asked me to use them nevertheless because:

* In case new microservices written in another programming languages need to be introduced in the system, they can use the same Protobuf models, as they can be converted in source code written in a plethora of programming languages;
* Currently the software runs on *Amazon Web Services*, but in the future it may be migrated to Google Cloud, which offers Protobuf integrations;
* He thought spending some time studying and applying Protocol Buffer would be useful for me in future jobs.

Listing 5.1: Protobuf model example.

```
// protobuf-example.proto
syntax = "proto3";
package commonproto;

// import the GoGo Protobuf extension
import "github.com/gogo/protobuf/gogoproto/gogo.proto";

// define package of the generated Go source code
option go_package = "pagination.com/queue-controller/internal/
    commonproto";

message PaginationLock {
    string id = 1 [(gogoproto.customname) = "ID", (gogoproto.
        jsontag) = "id"];
    string key = 2;
    int64 ttl = 3 [(gogoproto.customname) = "TTL", (gogoproto.
        jsontag) = "ttl"];
}
```

A small drawback is that they require some more complicated step in the build process of the various microservices, as they need to use a special Protocol Buffer compiler to autogenerate source code from the Protobuf models.

Actually, I didn't use Protobuf directly. Instead, I've used "GoGo Protobuf" a superset of the Protobuf specification which provides more control on the generated source code and can deliver faster serialization strategies. It's well-trusted in the Go community, as massive projects like *Etcd*, *Kubernetes*, and *Mesos* use it as well[15].

An example Protobuf model is presented on Listing 5.1.

5.2.4 Redis Client Wrapper

There exists 2 main open-source *Redis* clients for the *Go* programming language:

* **Redigo**: it's the most used client at Pagination, and currently has 10 Github issues and 6423 stars;
* **Go Redis**: it's the most famous Redis client, and currently has 33 Github issues and 6776 stars.

I decided to use *Redigo* because it had fewer issues and the development team was already comfortable using it. However, I didn't want to couple every AWS Lambda microservice with this particular Go client. In particular:

* *Redigo* offers an intricated *API* to run scripts;
* It offers way more features than what was required by Queue Controller;
* I wanted to make it easy to change Redis client once and for all dependant microservices.

The class diagram of this Redis Client wrapper is presented in Figure 5.3.

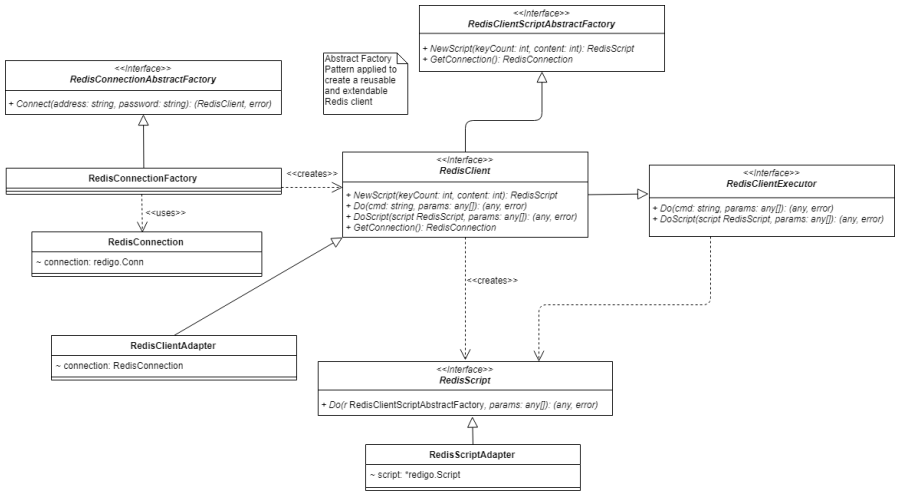


Figure 5.3: Class diagram of Redis Client wrapper.

5.2.5 Standard Redis Iterator Interface

According to my tutor, one of the most important things about this project is the possible evolution of the data structures it relies on in the *Redis* database. In particular, right now the data structures used to extract paginated lists of data aren't always optimized from the algorithmic point of view. These data structures are *Redis* lists and top-level keys.

About iterating over Redis Lists

Redis lists use the *LRANGE* command, which returns the specified elements of the list stored at a given key. It requires 2 zero-based indexes as start and stop offset, with 0 being the first element of the list (i.e. the head of the list), 1 being the next element, and so on. The -1 stop index can be used to retrieve every item from the specified start index to the last one. Redis doesn't produce out-of-range errors: if the start index is larger than the end of the list, an empty list is returned[32].

The signature of the *LRANGE* command is:

LRANGE key start stop

The time complexity of iterating over Redis lists is $O(S + N)$, where S is the distance of start offset from the head for small lists, from nearest end (head or tail) for large lists; and N is the number of elements in the specified range.

About iterating over Redis Keys

Redis offers a practical command to retrieve every top-level key available in a Redis database, also offering blob support to filter results. However, it should never be used in a production application since it runs as a blocking operation and may take a considerable amount of time and resources according to the number of keys stored.

Redis keys use the *SCAN* command to efficiently iterate over the set of keys in the connected database. Other data structures such as Redis sets, hashes or sorted sets use closely related commands: *SSCAN*, *HSCAN*, *ZSCAN*. Since these commands allow for

incremental iteration, returning only a small number of elements per call, they "can be used in production without the downside of commands like KEYS that may block the server for a long time when called against big collections of keys or elements"[33]. These commands have a drawback though: while blocking commands like *KEYS* are able to provide all the values in the top-level keys in a given moment, the *SCAN* family of commands only offer limited guarantees about the returned elements. This is due to the fact that the collection being incrementally iterated can change during the iteration process.

SCAN is a cursor-based iterator, so at every call of the command, the server returns an updated cursor that the user needs to use as the cursor argument in the next call. This implies that the whole state of a scan must be persisted on the client, and not on the server. An iteration starts when the cursor is set to 0 and terminates when the cursor returned by the server is 0. SCAN returns an array of two values: the first value is the new cursor to use in the next call, the second value is an array of elements.

The signature of the *SCAN* command is:

SCAN cursor *[MATCH pattern] [COUNT count]*

The time complexity of iterating over Redis keys is $O(1)$ for every call, and $O(N)$ for a complete iteration, including enough command calls for the cursor to return back to 0. N is the number of elements inside the collection.

Even though *SCAN* does not provide guarantees about the number of elements returned at every iteration, it is possible to empirically adjust the behavior of the command using the *COUNT* option. *COUNT* allows the user to specify the amount of work that should be done at every call in order to retrieve elements from the collection. Contrary to the SQL *"SELECT * ... LIMIT x"* construct, in Redis the *COUNT* option is just a hint; it's perfectly plausible for the user to receive 0 elements on the first two calls and $x + n$ elements on the third call. The number of items returned in a single *SCAN* call can be more than the specified COUNT when the saved data is so small that *Redis* can store it in an optimized data structure.

Similarly to *KEYS*, the *SCAN* command offers *glob-style* pattern matching with the *MATCH* option. A drawback is that if the pattern matches very little elements inside the collection, *SCAN* will likely return no elements in most iterations.

Redis Iterator Module

The tutor asked me to inspect the common points between the *LRANGE* command and the *SCAN* family of commands to create a generic way to retrieve paginated lists of data from *Redis*.

I have been inspired by the Iterator Pattern, which is a design pattern used to traverse a container and access the container's elements. Considering the drawbacks of the native *Redis* commands explained in the subsections 5.2.5 and 5.2.5, a generic *Redis* iterator would need:

* a function to return the next N elements. The number of elements may be only an estimate (to accomodate *SCAN* limits);
* a function that returns true if and only if the iterator has other items to traverse;
* a function that returns the latest offset index reached by the iterator, or 0 if the structure has been completely iterated over;

Listing 5.2: Redis Iterator interface in Go.

```
// iterator.go
package redisiterator

// Iterator is the interface needed to implement the Iterator
    pattern to create
// a uniform contract to traverse Redis data structures.
type Iterator interface {
        // Next returns at the next elements, whose cardinality's
            upper bound
        // is set to count.
        Next(count int) ([]interface{}, error)

        // HasNext returns true if and only if the aggregate
            associated with the iterator
        // has at least another element.
        HasNext() (bool, error)

        // GetLastIndex returns the last index reached by the
            iterator, or 0 if
        // it has completed traversing the structure.
        GetLastIndex() uint64

        // GetOptimisticResultLength returns an estimate of the
            expected maximum size of
        // the slice resulting from the iterations.
        // This should be used to initialize the capacity of the
            array needed to capture the results.
        GetOptimisticResultLength() uint64
}
```

* an estimate of the maximum size of the returned list. This is only and optimization strategy to initialize the capacity of the *Go* slice needed to store the results.

The *Go* interface is shown in Listing 5.2.

Go cannot be considered a proper object-oriented programming language, and has no concept of classes, thus class diagrams are somewhat impossible to design precisely in Go. For example, in Go, the idiomatic way to create a "constructor" for a *struct* object is defining a public function called *New* in the scope of the represented module.

A class diagram that explains how the *ListIterator* and *KeysIterator* are related the *Iterator* is shown in Figure 5.4. A particularity of the *KeysIterator* is that not only it exploits the native *Redis* glob-style matching feature, but it also sports an in-memory glob-style exclusion filter written in *Go*. This was necessary due to the suboptimal Redis structure in use at Pagination. I proposed an alternative structure that would simplify the algorithms needed to retrieve items of data to be displayed in paginated lists, but while the tutor agreed with my alternative, such a change would have taken many days of work and has been dismissed indefinitely.

Listing 5.3 shows an example usage of *KeysIterator*. Listing 5.4 shows an example usage of *ListIterator*.

Listing 5.3: Keysiterator example usage.

```
// creates a new keysiterator
it := keysiterator.New(
    &conn, // Redis connection
    &keysPattern, // glob-style pattern that identifies the keys
        to read

    // approximated count limit
    keysiterator.KeysIteratorLimit(params.Limit),

    // the index returned from the previous iteration
    keysiterator.KeysIteratorLastIndex(params.NextIndex),
)
count := int(params.Limit)
// rawKeysToRead contains the keys read from Redis
rawKeysToRead := make([]interface{}, 0, it.
    GetOptimisticResultLength())

for {
    ok, err := it.HasNext()
    if err != nil {
        return nil, 0, err
    }
    if !ok {
        break
    }
    // performs SCAN {lastIndex} MATCH {keysPattern} COUNT {limit
        }.
    rawNextBatch, err := it.Next(count)
    if err != nil {
        return nil, 0, err
    }
    result = append(rawKeysToRead, rawNextBatch...) // its type
        is []interface{}
}

// the last index of the iteration.
// If it's 0, it means the list is completely traversed.
lastIndex := it.GetLastIndex()
```

Listing 5.4: Listiterator example usage.

```
// creates a new listiterator
it := listiterator.New(
        &conn, // Redis connection
        listKey, // key that identifies the list to read

        // number of items to read for the current page
        listiterator.ListIteratorDefaultOptimisticResultLength(
            params.Limit),

        // the index returned from the previous iteration
        listiterator.ListIteratorLastIndex(params.NextIndex),
)

count := int(params.Limit)
// rawKeysToRead contains the items read from the specified Redis
    list
rawKeysToRead := make([]interface{}, 0, it.
    GetOptimisticResultLength())

// This cyle is guaranteed to run exactly once. It has this shape
    to make listiterator
// and keysiterator easily replaceable.
for {
        ok, err := it.HasNext()
        if err != nil {
                return nil, 0, err
        }
        if !ok {
                break
        }

    // performs LRANGE {listKey} {params.NextIndex} {limit +
        params.NextIndex + 1}
    // and discards the latest element.
        rawNextBatch, err := it.Next(count)
        if err != nil {
                return nil, 0, err
        }

        rawKeysToRead = append(rawKeysToRead, rawNextBatch...)
}

// the last index of the iteration.
// If it's 0, it means the list is completely traversed.
lastIndex := it.GetLastIndex()
```

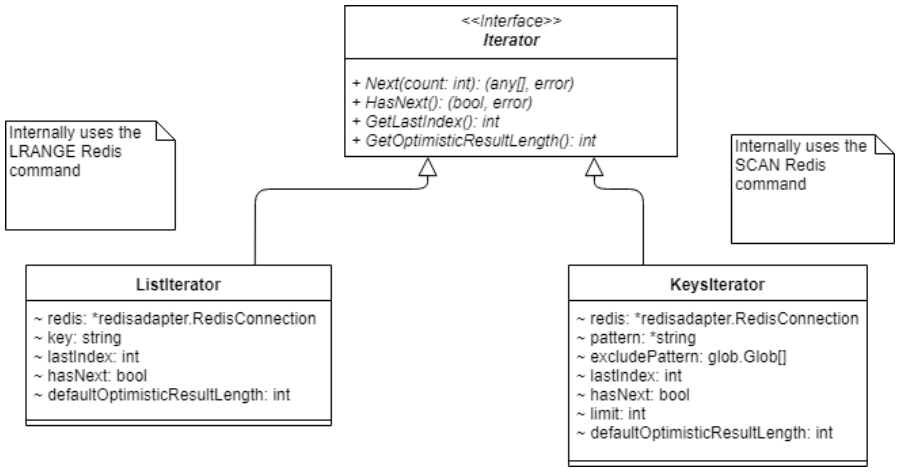


Figure 5.4: Class diagram that shows the relation between the List and Keys iterator implementations and the Iterator interface.

5.2.6 REST Parameters Extraction and Validation

The REST API exposed by Queue Controller is quite complex, and makes use of various types of dynamic parameters to receive input data from the user.

Query Strings

Query strings are the final part of a URL that assigns values to parameters. A query string starts with the question mark (?) symbol and the ampersand (&) symbol is used as a separator. As the name suggests, query strings accept only string values, so any value that's supposed to be a boolean or an integer must be manually converted server-side. I have used query strings to specify the next index of a Redis scan and an estimate of the number of element to be returned (e.g. /locks?nextIndex=153&limit=10).

URL Path Parameters

URL path parameters are dynamic values embedded in the URL path. They must be explicitly configured in AWS API Gateway. I have used them to indicate a single element of a collection (e.g. */jobs/123*) or to identify the current stage, which could be either "development" or "production": */stages/production/...*.

Body Payload

HTTP operations like PUT, POST and PATCH specify their input data in the so-called "request body". Usually REST APIs use JSON payloads as request bodies, as I and the tutor decided to do. JSON payloads are generally used to read structured data that should be inserted in a database. For instance, I have used JSON payloads in conjunction with the POST and PATCH operations to create and update a pagination lock definition. Contrary to query strings and URL path parameters, the payload in a request body isn't visible in the URL of the request.

Listing 5.5: JSON Queue Controller REST response example.

```
{
    "data": [],
    "meta": {
        "nextIndex": 143
    }
}
```

HTTP Headers

HTTP Headers are a key-value map of strings that is generally used to send metadata to the server. I have used them for 2 primary reasons:

1. Specify CORS headers to navigate properly using the project web application;
2. Send the token needed to operate on a pagination lock to the server.

Just like the body payload, headers aren't visible in the URL of the HTTP request.

Since many parameters with the same name and format were used to read data from multiple Lambda functions, I put the logic to extract and verify them in a separate utility module. Unfortunately, I didn't think about it when I first started working on the project. However, performing this code refactoring made me notice some trivial typos that caused unexpected real-time errors.

5.2.7 Common Response structure

In my opinion, one of the most important aspects of creating a *REST* API that can be easily extended by other developers is predictability. An *API* user should be able to quickly develop confidence by predicting how some configuration parameters may be called without consulting the documentation, or how a reply to an HTTP request should be structured. With this in mind, I defined a simple policy to construct responses, where every data to be returned should be defined as the value of the *data* key, whereas any additional info can be optionally defined in the *meta* object. For instance, the Listing 5.5 shows the JSON response of a Redis scan, in which the *meta* object contains the index cursor to be used for the next iteration.

This policy to construct JSON response is defined in a utility module called "successresponse". The constructor of this response utility accepts any number of *MetaResponseField* objects, where each one of them defines a single object to be appended to the *meta* object. If the *meta* object is empty, it isn't included in the JSON response. Generics would have come really handy in this case, since the *data* object can contain any structure, but unfortunately *Go* doesn't support generics yet. I have used Go's interfaces to emulate generics; in *Go*, an empty interface type (written as "*interface{}*") is the equivalent of the *any* type in *TypeScript*.

The common response structure of "successresponse" can be seen in Listing 5.10. The whole "successresponse" module code can is shown in Appendix B.

Listing 5.6: Common Queue Controller REST response structure.

```
package successresponse

import "encoding/json"

// SuccessResponse defines the structure of the JSON response
    returned in case of success.
type SuccessResponse struct {
        Data interface{}    `json:"data"`
        Meta *MetaResponse  `json:"meta,omitempty"`
}
```

5.3 Front-end

5.3.1 Standard Redux Architecture

Redux is a predictable state container for JavaScript applications. One of the most difficult aspects of software development is state management, which can be the source of many bugs. Redux was originally developed by Dan Abramov, which now works at Facebook and has contributed to core React.js features. Redux is a simplified implementation of Facebook's Flux architecture, which is a Model-View-Controller framework. It lessens the complexity of the model using Reducers. In Redux, Reducers are pure functions, i.e. functions without side effects; they are given a certain state and a Redux Action and compute the next application state.

Redux is inspired by functional programming and has 3 main principles:

* The application state is stored as a single object. State centralization makes the process of testing and debugging faster;
* The application state is immutable, in the sense that it cannot be modified directly. The only way to update the state is to provide a Redux Action, which is an immutable object that describe state changes. In order to prevent race conditions, Actions are executed in order;
* Reducers define how actions transform the state. They centralize data mutations and can act on all or part of the state. Being pure functions, reducers can also be combined and reused.

The Redux architecture can greatly increase the scalability for large and complex apps. It also enables powerful developer tools (the most notable of which is Redux DevTools, developed by the original Redux creator) because every mutation can be traced to the action that caused it. Figure 5.5 shows the Redux architecture and how it integrates with React.js components, which are part of the View.

When a user interacts with a React.js component, an event is emitted asking the dispatcher to return a Redux action. The dispatcher then checks whether the Action requires sending an HTTP request to a back-end API using middlewares. HTTP requests are asynchronous, and if such interactions are needed, the dispatcher returns the action and waits for the HTTP response. Then, the action is passed back and sent along with the current state to reducers. The reducer that knows how to handle that particular action creates the new state and replaces the old one. The Redux Store,

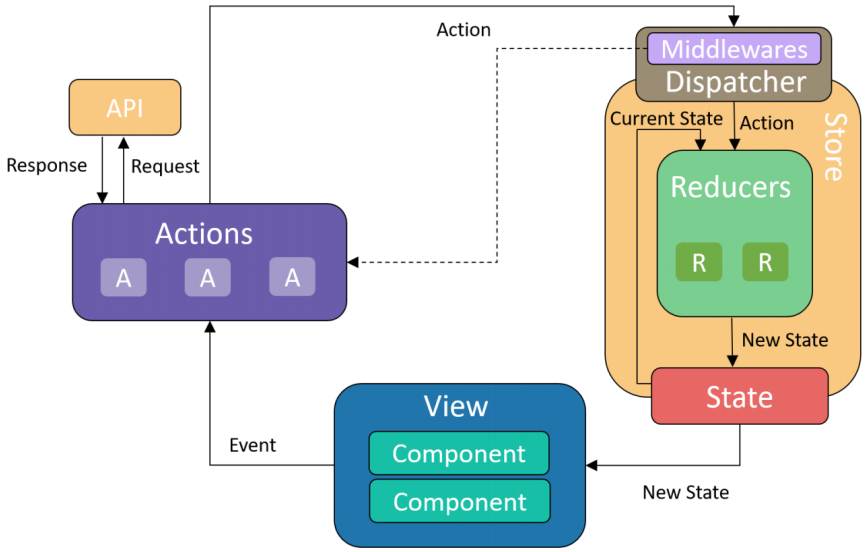


Figure 5.5: Redux Architecture and integration with React.js View components.

which contains the updated state, notifies any View component that uses the changed portions of the state. Those components are then re-rendered accordingly.

The Queue Controller front-end has about 50 composable components and containers. Without a centralized state solution like Redux, I would have needed to pass down React.js dynamic properties for chains of 4 to 8 components: it definitely wouldn't have scaled.

As Redux best practices suggest, I have used a combination of Redux and the native state management in React.js:

* I would use React.js component state system to handle local dynamic values, such as UI-related flags;
* I would use Redux whenever an API call was needed or when and action in a component should reflect in another component of the React state tree.

I have decided to use Redux because, even though other centralized state management solutions exist, it's still the most popular and widely used in the community. I've also studied it extensively: I've been using it for about 4 years now and have already used it in a professional environment. Considering the amount of new topics I've discovered with this internship, re-using some old knowledge of technology that fits the purpose helped me respect the deadlines and lessen the project risks.

5.3.2 Client-side Routing

I have used *React Router* to perform client-side navigation on the React.js web application. I have enjoyed the declarative style of React Router because it fits perfectly with the other React code, it's natural and, since it's written in JSX, it's very easily readable. It also offers the possibility to define dynamic routes that map to React.js components rendered after the first web page render. A nice feature I was able to implement with this third-party library was being able to navigate between the

application web pages and toggling detail views using the URL. This is particularly useful, because with a simple copy-paste of the URL of the application used by a customer, the developers can reproduce the same dynamic web page shown to the user.

5.3.3 List Cache Module

In general, the fewer HTTP requests are sent to AWS Lambda, the less the company needs to pay to AWS for the usage fees. Even though the web traffic related to Pagination's Lambda functions is low enough to be free (less than a million requests a month), it may rapidly increase in the future, thus the company would benefit from a traffic optimization.

The tutor asked me to create a client-side cache module for the API operations that returns paginated lists of data, i.e. lists that are potentially too big to be retrieved all at once, and require multiple partial iterations to be traversed efficiently.

Every HTTP GET operation exposed by a REST endpoint that returns a list of items shares the same JSON-formatted response structure. This structure was already shown in Listing 5.10. The cache module uses the *nextIndex* returned by the HTTP response and the relative data to construct an incremental list cache.

The cache module exposes the following characteristics:

* It should only perform calls for paginated portions of data not yet retrieved from the cloud;
* It should be possible to delete some items in a page without compromising the cache;
* The cache can be reset at any time if the user wants so;
* Given a starting list page N, accessing the previous $N - 1$ list pages and then returning to page N doesn't require any API call;
* Its main logic is decoupled from every third-party library or components. The cache module is simply used to compute and maintain the list cache. The portion of the state relative to the currently visualized list page is then copied to a Redux object to make it accessible to React.js page components. Redux, however, is unaware of the indexing strategies and objects needed to maintain the cache.

The cache module alone has more than 700 lines of test code and 17 unit test cases. These unit tests have 100% code coverage, which have been greatly appreciated by my tutor.

Contrary to list traversal, keys traversal on Redis can only be performed in a single direction, from the start to the end the set of keys. This is due to the fact that the *SCAN* command only returns the next index of the scan, and not the previous one. So how does a user view the items returned in the previous page of the key set? The cache module solves this problem too, since it can retrieve the previous page without performing additional API calls.

5.3.4 SCSS

Even though the visual layout of the client-side application wasn't a priority concern of the company that proposed me the project, I attempted to make it as good-looking as possible, with an eye on pleasant color contrast. I opted for a minimalistic design to avoid as much clutter as possible.

Listing 5.7: Common Queue Controller REST response structure.

```
/**
 * The soon-to-be-deleted job is in the current cache page of
    listClient.
 * Every pages that comes after the current page and the current
    page itself must be invalidated.
 * If the current page has only a single job, the previous page
    should be loaded.
 * If the previous page doesn't exists, the entire cache should
    be deleted (i.e. just call resetListJobs()).
 */

// listClient has type `ListProxy<PaginationJob>``
const listClient = jobCache.get(apiClientConfig, stageID);

// calls the API to delete `job` from Redis
await actionJobClient.delete(stageID, job);

// Removes `job` from the cache if it exists.
// The `areEqual` callback is used to perform the equality check
// on `PaginationJob` objects stored in the cache.
// `job` has type `PaginationJob`, and `job.jobID` is its unique
// ID.
// `response` has type `PaginatedListResponse<PaginationJob>`
const response = listClient.deleteAtCurrentPage({
  matcherValue: job,
  areEqual: (actualJobID, expectedJobID) => actualJobID.jobID ===
      expectedJobID.jobID,
});
```

Listing 5.8: Common Queue Controller REST response structure.

```
export interface DeleteFilter<T> {
  matcherValue: T;
  areEqual: (collectionItem: T, matcherValue: T) => boolean;
}
```

Listing 5.9: Common Queue Controller REST response structure.

```
export type PaginatedListResponse<T> = {
  data: T[];
  hasNext: boolean;
  hasPrevious: boolean;
};
```

Listing 5.10: Common Queue Controller REST response structure.

```
export type ScanPaginatedState <T> = Readonly<{
  // Business entities needed to handle the cache logic
  // and compute the next paginated state
  entities: {
    //
    data: T[],

    // Latest index read from the API
    nextIndex: number,

    // .
    // Every new index is appended to the tail of the list
    scanIndexSequence: number[],

    // Index that marks where the current slice
    sliceStart: number;
    sliceEnd: number;
  },

  // Simple cache state ready to be used by the view components
  ui: {
    // Portion of entities.data corresponding to the
    // current page
    slice: T[],

    // True iff the next page has elements
    hasNext: boolean,

    // True iff the previous page has elements
    hasPrevious: boolean,
  },
}>;
```

Instead of using the standard CSS, I decided to use SCSS to define the layout style of the web application, which improves and adds features to CSS, but requires a compilation process in order to be interpreted by web browsers. It has numerous benefits[1]:

* It's CSS-syntax friendly: valid CSS code is also valid SCSS code;
* It gives the possibility to define variables and simple utility functions, which provides better code organization, more maintainable code, and a syntax that is often more natural and easier to read;
* It gives the possibility to split the layout codebase in multiple, hierarchical files;
* It has a large open-source community and is well documented;
* It can be used to enhance famous SCSS frameworks, such as Bootstrap 4 or Bulma.

In Queue Controller, I have used a customised version of Bulma as SCSS framework. Moreover, its integration with React.js is seamless, and CSS preprocessors like SCSS or Stylus are widely used in the Web Design industry.

[1] 27.

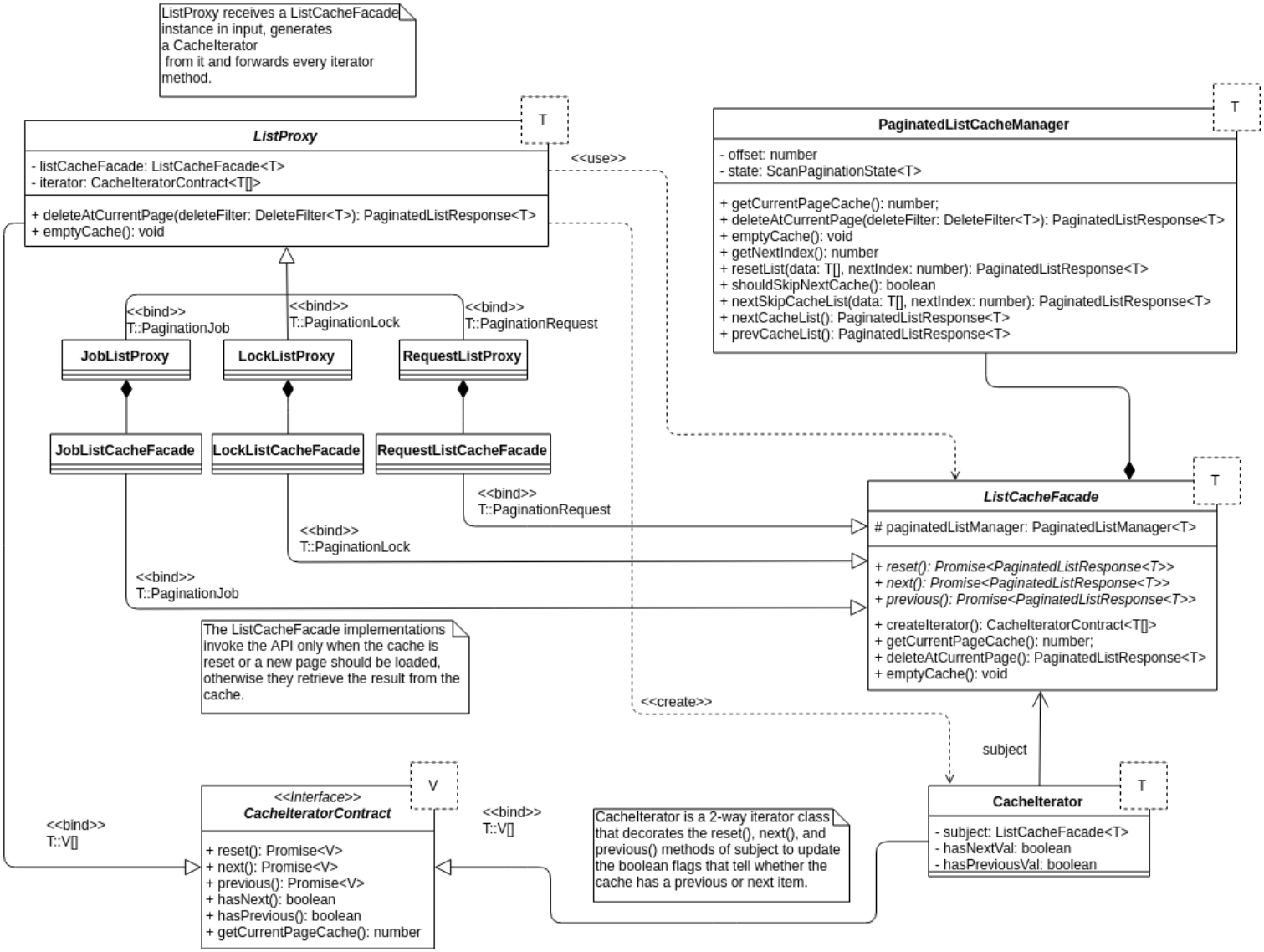


Figure 5.6: Class diagram detail of the cache module.

Chapter 6

Deployment

This chapter briefly explains the deployment process for the Queue Controller project using Terraform.

6.1 Automated Deployment

In software engineering, deployment is the process of preparing and configuring a software application to run and operate in a specific environment and install it in said environment. Deployment can either be carried out manually or through automated systems. Manual deployment is discouraged for multiple reasons:

* It necessarily requires human intervention;
* It's often a time consuming process, especially for bigger applications;
* It's error prone, since the deployment steps must be performed in a particular order and may require the usage of a terminal and the insertion of secure credentials;
* It's easy to collide with another ongoing deployment performed by a colleague (this actually happened a lot of times in my previous company);
* It doesn't provide fallback mechanisms that revert the status of the application running on the server when something goes wrong;

The only perceived advantage of manual deployment is simplicity, since oftentimes an File Transfer Protocol client or an SSH client are the only tools needed to set up an application on a system. But what good is simplicity when it requires manual and repetitive work?

A good automated deployment system should have the following characteristics[25]:

* It should be triggered by one action, such as a single command on a terminal;
* The deployment steps should be pre-defined, reproducible, predictable;
* Should require little to no human intervention;
* It should be performed as an atomic operation, where either each step is performed correctly or the system is reverted to its previous state;

* An in-progress deployment should lock the project to avoid other concurrent deployments on the same resources.

My tutor and I decided to perform automated deployments with *Terraform*.

6.1.1 What is Terraform

Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently. Its architecture is plugin-based, enabling developers to extend it by writing new plugins or modifying existing plugins.

Terraform is logically split into two main parts: Terraform Core and Terraform Plugins. Terraform Core discovers and load the plugins to use and communicates with Terraform Plugins. Terraform Plugins, on the other hand, expose an implementation for a specific service, such as AWS, or provisioner, such as bash.

Terraform Core

The main responsibility of Terraform Core are:

* Reading and parsing the infrastructure as a code files and modules;
* Constructing the resource graph;
* Plan execution, i.e. execute the steps necessary to perform the deployment;
* Communicate with plugins over Remote Procedure Call.

Terraform Plugins

Terraform Provider Plugins are responsible of:

* Initialize any included libraries used to make API calls;
* Authenticate with the infrastructure provides (such as AWS);
* Define resources that map to specific services offered by the infrastructure provider.

Each plugin is written in Go and exposes an implementation for a specific service or provisioner. All providers and provisioners used in Terraform configurations are plugins. They are executed as a separate process and communicate with the main Terraform binary over an RPC interface. Terraform Plugins are responsible for the domain-specific implementation of their type.

Terraform plugins are installed and discovered with the `terraform init` command. When this command is run, Terraform reads configuration files in the working directory to determine which plugins are necessary, searches for installed plugins in several locations, decides which plugin versions to use, and writes a lock file to ensure Terraform will use the same plugin versions until the same command is run again.

6.1.2 Terraform Configuration

Terraform configurations can get messy quite soon. After experimenting a bit and confronting myself with the tutor, I decided to group the automated deployment configurations in logical groups, called *modules*.

Lambda Module

It defines the boilerplate and variables needed to define the AWS Lambda resources via related Terraform plugins. Some values must be read from the parent module, which must specify, for example, the location of the executable binary, the name of the Lambda function, or the environment variables to be set once the AWS Lambda function is started.

Routes Module

This module mirrors the REST routes defined in the AWS API Gateway configuration. It contains a folder for every entity cited in section 1.3:

* *stages* for the resources related to the root REST endpoint that contains a variable stage identifier parameter;
* *jobs* for the resources related to the functions that create, list or delete pagination jobs;
* *locks* for the resources related to the functions that create, list or delete pagination locks;
* *requests* for the resources related to the other functions that interact with Redis.

The resources in the jobs, locks, and requests folders inherit the definitions in stages, since their endpoints contain a stage parameter. These submodules contain, for each REST endpoint, a configuration file named as the corresponding Lambda function containing the necessary AWS API Gateway method and integration resources. These resources are injected into the lambda module, which is imported. They also use a third-party plugin to set up CORS.

Main Module

The main module is tes Modulthe core of the configuration. It acts as a central hub that links the aforementioned resources and modules together. It defines the policies and permissions required by AWS to interact with their cloud services, and it maps the AWS Lambda resources defined in the lambda module into AWS API Gateway endpoints, specifying the appropriate HTTP verbs and dynamic parameters. Moreover, the main module reads the VPC configuration in use at Pagination from a private AWS S3 bucket.

6.1.3 Integration between AWS API Gateawy and AWS Lambda

Lambda functions are only reachable via HTTP POST requests. However, it would be inadequate to perform a POST request to retrieve data without altering the state: a more appropriate HTTP verb is GET. AWS API Gateway has been configured to map the incoming requests to POST requests to the Lambda functions that compose Queue Controller.

Listing 6.1: Snippet extracted from the Terraform resource configuration for AWS Lambda functions.

```
# Terraform resource template for AWS Lambda functions
resource "aws_lambda_function" "function_template" {
  # The local file to use as the lambda function.
  filename = "${var.zip_location}/${var.zip_file}"

  # A unique name to give the lambda function.
  function_name = var.function_name

  # IAM (Identity and Access Management) policy for the lambda
      function.
  role = var.lambda_role_arn

  # Language runtime
  runtime = var.lambda_runtime

  # The entrypoint of the Lambda function
  handler = var.lambda_handler

  # The maximum timeout in seconds for the execution of the
      Lambda function
  timeout = var.lambda_total_timeout_s

  # Source code hash to detect whether the source code of the
      lambda function
  # has changed. If it changed, Terraform will re-upload the
      lambda function
  # zip file.
  source_code_hash = filebase64sha256("${var.zip_location}/${var.
      zip_file}")

  environment {
    # The environment variables map
    variables = local.lambda_environment
  }
}
```

Chapter 7

Verification and Validation

This chapter explains the verification and validation processes which have certified the quality of Queue Controller.

7.1 Verification

According to the ISO/IEC 12207:2008 standard, *verification* is a support process that ensures that a certain activity didn't introduce errors during the evaluation period. There are two types of verification: *static* and *dynamic*.

Static verification is useful because it doesn't need the whole product to be executed to check for certain errors.

Dynamic verification requires running portions of code, it must be repeatable and can be automatized with proper instruments, called *test suites*. Tests are one of the main examples of dynamic verification.

7.1.1 On TDD

During the internship, I have attempted to follow the Test Driven Development practice, which consists in creating tests beforehand, writing the code necessary to make those tests run successfully, and refactoring both the test and the running code once all the tests pass.

Table 7.1 shows the benefits and drawbacks of using *Test Driven Development*.

	Benefits
Drawbacks	
Increased external quality[21, 37]	Increased development time[21, 22, 37]
Increased test coverage[37]	Requires high discipline[24]
Tests executed more frequently[22]	Can lead to brittle tests[9, 11]
More precise test cases[22]	

Table 7.1: Benefits and drawbacks of using Test Driven Development.

Benefits of using TDD are not completely clear and exact. In 2010, Kollanus found

weak evidence regarding better external software quality with TDD, and even less evidence about internal software quality. Even though results still hinted to better quality, the evidence was not uniform. External quality was measured comparing the successful acceptance tests and the number of defects reported by the customer. Internal quality had various metrics, such as code coverage, method size, cyclomatic complexity, number of cases. Often times, the metrics were deemed contradicting and the internal quality had unclear consensus. In 2016, Bissi et al discovered more promising results about TDD: 88% of studies revealed significant increase in software quality, and 76% of studies showed significant increase in internal software quality. In this case, internal quality was only inspected using code coverage. According to Williams et al., when using TDD developers are also encouraged to produce more precise and accurate test cases.

TDD requires and increase in development time and is sometimes referred to as a design technique, making it maybe a too demanding practice for junior developers. Even senior developers seem to lack the discipline required to perform TDD properly. Besides, TDD might have a tendency to shift the developers viewpoint to verifying the system state rather than its behavior, which can lead to brittle tests.

In my experience, I wasn't able to follow the TDD practice completely. Sometimes I found myself skipping the test code refactoring part; some other times I needed to run the code interactively to understand some edge cases of the interaction with Redis. I would say that TDD is not the silver bullet for software testing, but it could be argued that, in the right context, benefits of TDD outweigh its drawbacks.

7.1.2 Verification purpose

The verification activity was aimed to the following goals:

* notice and troubleshoot code errors;
* notice requirement changes;
* highlight design changes;
* identify components with an ambiguous or partially unknown behavior;
* identify wrong integrations between components.

A critic point has been deciding a balanced number of test cases to produce without risking of breaking the planned deadlines. Due to this, me and my tutor agreed to produce at least a unit test per method and the most general integration test cases per module. Anyhow, I was able to obtain 100% coverage in 5 back-end modules and in the front-end module that managed the list cache subsytem.

The metrics obtained after the verification process are:

* test coverage;
* ratio of tests successful vs tests run (which cannot fall below 100%);
* system requirements coverage.

7.1.3 Static Analysis

In order to comply with the best practices and prevent basic typos, all the modules have been subjected to static analysis. For the back-end written in Go, I've used the standard tools: *Go Lint* as linter and the *Go Vet* code inspector. The tutor and I decided that the Lua code to be run on Redis didn't need to be subjected to static analysis, so I didn't use any Lua linter. For the back-end, I've used the TypeScript linter embedded in the standard React.js boilerplate, *create-react-app*.

At the end of the internship experience, every module had 0 linter warnings and errors.

7.1.4 Test Development

Unit Test

I developed unit tests for every internal module that didn't involve external sources, such as connections to Redis or AWS Secrets Manager.

Due to time constraints, for what concerns the front-end, I only performed unit tests on a couple of router utilities and the whole list cache management system explained in section 5.3.3.

Integration Test

I developed integration tests for every AWS Lambda submodule. The common integration setup for Redis Integration tests was the following:

* Spin up an empty Redis database on the same machine that runs the test using a Docker image;
* Initialize a Ginkgo test suite and connect to the local Redis instance. In this case, the credentials needed to connect to Redis were read from the environment variables, since the local Redis instance is fictious in any case;
* Configure Ginkgo to reset the Redis connection and clean the data at each time.
* Initialize the variables to assert in the test suite definition;
* I would combine the BeforeEach and JustBeforeEach methods offered by the Ginkgo framework to update the variables to assert in the inner test cases;
* Define the data to be inserted into Redis directly into the

When the same resource is involved for multiple integration tests, it's important to run them sequentially, otherwise the tests may result in undefined behavior.

Since I didn't use Jenkins until the last three weeks of the project, I initially wrote a Docker Compose definition to perform both unit tests and integration tests locally. That Docker Compose definition would spawn a local Redis instance without

7.1.5 Continuous Integration with Jenkins

Jenkins is one of the most famous open-source continuous integration tools. It offers its user the possibility to define projects that can perform automated tasks, such as running the build process of a software component, executing tests and gathering their results, or deploying the artifact on the target infrastructure.

It's easily configurable and for most use cases it works "out of the box". Jenkins also has a huge plugin ecosystem maintained by the open-source community. It's based on a master-slave architecture, where the main server instance (the master) schedules and manages the tasks on one or more slaves. Slaves simply perform the tasks assigned to them by the master instance.

Jenkins Pipeline

A Jenkins Pipeline is a suite of plugins that define the tasks performed by Jenkins. With Jenkins, every change to the software committed in source control goes through a complex process before being released. This process involved building and testing the software in a reliable and repeatable manner, as well as deploy the artifact generated from the source code to the target environment. Jenkins uses a domain-specific language syntax to model the steps to be performed by the pipeline. These steps are usually defined in a special file called *Jenkinsfile*. Creating a Jenkinsfile and adding it to the version control system has many benefits:

* It automatically creates a pipeline build process for all branches of the Git repository;
* It allows to edit the steps directly on the pipeline dashboard;
* It provides a single source of truth for the pipeline, which can be viewed and edited my every team member of the project. This last aspect is very similar to the "Infrastructure as a Code" concept of Terraform.

Jenkins uses two pipeline syntaxes, the *declarative* and the *scripted* pipelines. Since the scripted pipeline is the only one that supports Docker integration, and given that I needed Docker to perform integration tests with a temporary Redis instance, I decided to use the scripted pipeline. The Jenkinsfile and the scripts run by it are listed in Appendix A. The most relevant aspect of it is that 2 Docker images are run and linked together. The main steps performed by Jenkins in the Queue Controller project are the following:

* It creates a Docker image of the environment with everything needed to build the microservices;
* It installs the Go third-party dependencies, minifies the Lua scripts and embeds them in Go files;
* It generates the Go sources for the Protobuf definitions;
* It creates a Go production executable for every Lambda function;
* It runs the unit and integration tests. First, a Redis Docker image is run, and its context is captured in a variable called *redis*. Then the *queue-controller-builder* image created earlier is used to run tests. The container that runs *queue-controller-builder* is configured to reach the network of the Redis image. In this case, the Redis credentials are defined as environment variables;
* If the tests are successful, the binary executables generated by the build process are wrapped in a zip file, and are ready to be deployed to AWS.

I have used TravisCI as continuous integration software for years, and have struggled a bit using Jenkins due to the huge difference in defining a CI pipeline. For this project, I decided to use Jenkins for two primary reasons:

* It's used in many companies, so some experience with it may help me with my future jobs;
* It's the CI tool used at Pagination, and I wanted to follow along with the development team I was working with.

7.2 Validation

Validation is the process of evaluating software during or at the end of the development process to determine whether it satisfies specified business requirements. It's needed to ensure that the final product actually meets the needs of the user and that the specifications were defined correctly. Validation demonstrates that the product fulfills the user requirements and that it works correctly in its intended environment.

The tutor and I decided to verify the project improvements defining baselines roughly every 7 working days. The tutor has evaluated the quality and scalability of the project for each baseline and the final version of the product. The cumulative evaluation of the production has been: **beyond expectation**.

Chapter 8

Conclusions

This chapter reports a critic evaluation of the project goals achieved and analyzes the technical and professional knowledge acquired during the internship at Pagination. Moreover, it proposes a personal comparison between work and the University.

8.1 Fulfillment of Project Goals

As reported in the introduction, before starting the internship experience, my tutor and I had redacted the work plan. Besides other relevant information, it contained the definition of the goals to fulfill within 320 hours of work, presented as mandatory, desired and optional.

I managed to completely achieve the mandatory and desirable objectives. I was also able to set up a Continuous Integration process for building and testing every Microservice, as well as the system as a whole. During the first week of the internship, after a discussion with the tutor, we realized that a *GraphQL* interface wouldn't suit the project, so we removed it from the optional requirements.

The original work plan goals are reported in Table 8.1.

Code	Goal	Achieved
O01	Requirements Analysis Redaction	YES
O02	System Architecture Design and Documentation	YES
O03	Design of the services that manage Pagination processes in Redis queues	YES
O04	Design of the service that delete Pagination processes in Redis queues	YES
O05	API REST design for the Redis Queue management system	YES
O06	Design of a UI interface with a framework of choice between React.js, Angular, Vue.js	YES
O07	Implementation of the designed system	YES
O08	Unit Testing of the system	YES
O09	Create the back-end using microservices running on AWS Lambda choosing between Node.js and Go	YES
D01	Creation of a module to report errors in Redis queues	YES
D02	Integration Testing of the system	YES
D03	Automatic deployment of the system on Amazon Web Services	YES

Table 8.1 continued from previous page

Code	Goal	Achieved
F01	Creation of a GraphQL interface for handling Pagination Redis queues	NO
F02	Set up an automated Continuous Integration process for the system	YES

Table 8.1: Resume of the goals defined in the original work plan.

8.2 Final Hours Accounting

In the preliminary work plan, I had compiled a list of software engineering activities that I expected to perform during the internship, as well as an estimation of the time needed to complete each of them. These activities are reported in Table 8.2, where their estimated hours are compared to the actual hours needed.

My initial previsions have been sufficiently similar to the actual hours taken by the various project activities. However, since many requirements weren't known at the time the work plan was created, and I didn't know I would have set up an automated deployment configuration with Terraform, the *"Requirements Analysis and Tech Stack choice"* activity and the *"Independent study of the required technologies and experimentation"* activity had been underestimated. Also, the project design and implementation took less than foreseen.

Activity	Estimated Hours	Actual Hours
Independent study of the required technologies and experimentation	22	36
Requirements Analysis and Tech Stack choice	26	38
Definition of reference architecture and related documentation	80	66
Back-end design and related tests	56	48
Front-end design and related tests	16	10
Documentation related to the system design	8	8
Implementation of the system and related tests	136	128
Back-end implementation and related tests	72	64
Front-end implementation and related tests	48	48
Documentation related to the developer documentation	16	16
Manual Verification	32	28
Deployment and Continuous Integration	8	10
Final Documentation Draft	12	12
Demo and Presentation	4	2

Table 8.2: Comparison between the estimated and actual hours taken to complete the project activities.

8.3 Requirements Status Tracking

This section presents again the requirements defined in chapter 4 and marks each of them with their completion status, which can either be *Completed* or *Abandoned*.

8.3.1 Functional Requirements Status Tracking

Code	Status
RFM-1	Completed
RFM-2	Completed
RFM-3	Completed
RFD-4	Completed
RFM-5	Completed
RFM-6	Completed
RFM-7	Completed
RFM-8	Completed
RFM-9	Completed
RFO-10	**Abandoned**
RFM-11	Completed
RFM-12	Completed
RFM-12.1	Completed
RFM-13	Completed
RFM-13.1	Completed
RFM-13.2	Completed
RFM-14	Completed
RFM-14.1	Completed
RFM-14.2	Completed
RFD-15	Completed
RFD-15.1	Completed

Table 8.3: Functional requirements status tracking table.

8.3.2 Constraint Requirements Status Tracking

Code	Status
RCM-1	Completed
RCM-1.1	Completed
RCM-2	Completed
RCD-2.1	Completed
RCO-2.2	Completed
RCM-3	Completed
RCM-3.1	Completed
RCO-3.2	Completed
RCD-4	Completed
RCM-5	Completed
RCM-6	Completed
RCM-6.1	Completed
RCM-7	Completed
RCM-8	Completed

Table 8.4 continued from previous page

Code	Status
RCM-9	Completed
RCM-9.1	Completed
RCM-9.2	Completed
RCM-9.2.1	Completed
RCM-9.2.2	Completed
RCM-10	Completed
RCO-11	**Abandoned**
RCM-12	Completed
RCM-12.1	Completed
RCM-13	Completed
RCM-13.1	Completed
RCM-13.2	Completed
RCM-14	Completed
RCM-15	Completed
RCM-15.1	Completed
RCM-16	Completed
RCM-16.1	Completed
RCD-16.2	Completed
RCM-16.3	Completed
RCM-16.4	Completed
RCM-16.5	Completed
RCM-17	Completed
RCD-18	Completed
RCD-19	Completed

Table 8.4: Constraint requirements status tracking table.

8.3.3 Quality Requirements Status Tracking

Code	Status
RQD-1	Completed
RQD-2	Completed
RQM-3	Completed
RQM-4	Completed
RQM-4.1	Completed
RQM-4.1.1	Completed
RQD-5	Completed
RQM-6	Completed
RQM-7	Completed
RQM-8	Completed
RQD-9	Completed
RQO-10	Completed
RQO-10.1	Completed
RQM-11	Completed
RQM-12	Completed

Table 8.5: Quality requirements status tracking table.

8.3.4 Summary

A summary of the status of the requirements at the end of the project is presented in Table 8.6.

Type	Total	Completed	Abandoned
Functional	21	20	1
Constraint	38	37	1
Quality	15	15	0

Table 8.6: Summary of the status of the requirements at the end of the project.

Only 2 requirements were abandoned:

* **RFO-10** was abandoned because when I exposed my doubts to the tutor he acknowledged that I couldn't perform that operation with the current software structure;
* **RC0-11** was abandoned because the nature of this project doesn't make it suitable for using GraphQL.

8.4 Knowledge acquired

I am particularly glad to have completed the assigned project because it allowed me to reinforce previous skills and to acquire new knowledge. I summarized this new knowledge in the following paragraphs.

Cost-benefit analysis

As a side aspect of the project, I've learned the basics of how to perform a proper cost-benefit analysis. Before this project, I didn't really realized how much it's important, especially in the IT sector. My tutor experience has been fundamental for selecting the proper cloud services and configuring them for minimizing the costs. I've also had fun crawling the Amazon documentation to retrieve the variable costs for some services, like *AWS Lambda*, and I've used them to create a practical cost calculator in JavaScript.

Cloud services

Ever since high school, I've been fascinated by the cloud and how it revolutionized part of modern software development as well as posing new challenges. This internship project gave me the possibility to study in deep how to combine together various cloud services, and in particular to deepen my knowledge about the Serverless paradigm. I've also understood better the pricing model of different cloud services, and how planning a good cloud architecture is crucial to minimize the costs. Many companies have switched to cloud systems in the latest years, and I think they represent a revolution that is here to stay for quite a long time.

Amazon Web Services

Amazon Web Services are remarkably popular among software development companies, so I think that having dealt with them every day for two months has increased the value of my tech-related skills.

Continuous Integration with Jenkins

The only two University courses that present Continuous Integration to students are *Software Engineering* and *Open Source Technologies*, which are both scheduled at the third year. In my opinion, the *Open Source Technologies* course would fit better if scheduled in the first year.

During this internship I was able to use and configure a Jenkins pipeline for the first time. While I've had previous experiences with CI systems like Travis CI or GitLab CI, this has been my first time ever with Jenkins. I'm glad I was able to combine the usage of Docker with a Jenkins pipeline to automatically build and run tests on the system.

Deployment Automation with Terraform

In my previous company, Brainwise, I was friend with a DevOps engineer who induced in me the interest in software systems automation and the automated deployments. Even if I appreciated the chats with him, I never really had the occasion to put his DevOps advice in practice until this internship project. My tutor Simeone was patient enough to guide me through the first steps with *Terraform* and showed me how the cloud infrastructure is usually deployed at Pagination. After spending some time reading *Terraform* documentation and reviewing different online developers implementations, I was also able to structure Terraform code in a way that my tutor judged "better and neater" than other Pagination projects. I'm sure this knowledge will help me negotiate a higher salary in the future.

Software Testing in Go

In times where job offers often require appliers to know Test Driven Development and *automated tests*, my Go knowledge would be incomplete without experience in at least a Go testing suite. My tutor introduced me to *Ginkgo*, and although there were some differences compared to Node.js testing suites (with which I'm more comfortable) I'm glad to have learned how to use it. I will definitely use this knowledge again in my future back-end projects.

8.5 Post Internship evaluation

Overall, I'm extremely satisfied with what I've learned and achieved during this internship.

The office

In my experience, Pagination's office has been the most wide and less cluttered so far. It's located in a palace right in the center of Padova, in front of *Garibaldi Square*, and

has 4 big square rooms separated by sliding doors. The office mood has always been calm and casual, which helped me feel positive and work without stress.

The perks

Every day, Pagination orders takeaway food for lunch, with every expense covered by the company. Every day, we would be proposed a different dish to eat. Once a week, there was also the possibility to hang out together at a restaurant to encourage socializing among colleagues. I think this is a great perk for a company, it feels the employees appreciated for their work and keeps the team's mood high.

Possible Improvements

There are, however, some aspects that could have been better:

* Air cooler system: unfortunately, it had numerous defects, and some days our room reached 34°C. I think that a good office temperature is essential (especially in the hottest months) for the wellness and the sense of tiredness of the team;
* Commute time: even though I was aware of the distance between my home and the company office when I chose this company, I have expressed some critic considerations about commute time in 8.5.1.

8.5.1 About Commute Time

I've come to understand the difficulties and the social impact of more than 3 hours of commute time every working day for 2 months. During the internship, I would wake up at 6.20AM, leave from the train station at 7.01 and arrive at the office at 8.30AM. Then I would leave the office at 5.30PM, mount on the 5.51PM train and arrive at home at 7.30PM.

I was a commuter student only during the first year of university, but I would stay in Padova only about 4 hours every day. The following years I did not attend the lessons and I reached the University classrooms almost only for the exams.

* Excluding the first 3 weeks of work, I wouldn't need to interact much with the tutor face to face, I could have just chatted with him;
* I would have been more productive if I didn't have to spend that much time travelling every day to reach the workplace. In particular, I would have been less tired, so I could better focus on the work.

8.5.2 About the Degree Courses

As a student-worker with 3 years of experience, I enjoyed studying new technologies during the internship and didn't really have too many doubts or struggles implementing the project.

The Bachelor's Degree in Computer Science offers its students the necessary knowledge needed to be comfortable learning new technologies, programming languages or frameworks on their own. To my mind, the courses more relevant to this internship experience have been Object-Oriented Programming, Algorithms and Data Structures, Concurrent and Distributed Programming, Open Source Technologies, and Software

Engineering (SWE). In particular, topics like SOLID principles, data structures drawbacks, design patterns, and network communications in a distributed environment are fundamental for a 21st-century software developer. I believe that the group project assignment offered by the Software Engineering course is quite similar to the internship experience, so I'm grateful for having trained with that project first.

However, I would have preferred a slightly different course structure. To get started, in my opinion, the Open Source Technologies course should be proposed to the first-year students. It doesn't have any pre-requirements and is extremely illuminating, as it explains the importance of software testing, code versioning, and continuous integration. I think I would have struggled less in the past if I had known how to properly perform continuous integration and tests at a younger age.

I would use structure the Web Technologies project different. While I sincerely appreciated the lectures about accessibility and user experience, the tech-oriented parts of the course and the scope of the project seem tremendously outdated.

Finally, I would reduce the number of people needed for each group for the SWE course project. 7 to 8 people with no prior group collaboration experience are just too many, in my opinion. The risk that someone isn't willing to collaborate and share ideas and may instead take advantage of the situation is tremendously high.

8.5.3 Comparison between University and Work

I think that I wouldn't have been able to adopt new technologies so quickly if I hadn't maturated a professional experience in the software development field alongside my University career. Finding the right balance between the day-to-day tasks at work and the University studies have been a tough experience, which I believe made me a better person and a more proficient student in the end.

Here are expressed some personal views about the different between University and work:

* Being able to negotiate schedule, quality, and features without disappointing the stakeholders is much more important at work. This is because there's money at stake, and not just exam marks;
* In general, small errors are admitted in a work environment, but being too careless may cost someone his own job;
* In some cases, working with colleagues with the same age may be more inspiring that working with much older colleagues that may have lost passion for their job. In other cases, however, older employees may be able to teach tips or more mature points of view on a particular technology or programming language feature;
* The projects developed for University can easily span across multiple months with little hours dedicated to them each week. In an office, instead, a developer would probably work 8 hours each day at the same project, which reduces the possibility to clear his mind and change focus for someday;
* Working for a software development firm and gaining a wage from that job may be more motivating than studying without earning money;
* I believe that working collaboratively on a team of people with different backgrounds and experience levels is easier in a professional environment, because people without willing to help and cooperate would probably be left at home, or would work on solo projects.

Appendix A

Jenkins Scripts

In this chapter are listed the project Jenkinsfile, Dockerfile, and a sample of the bash scripts needed to build Queue Controller.

Listing A.1: Jenkinsfile for the Queue Controller project

```
#!/usr/bin/env groovy

node {
  checkout scm

  def environment = ["HOME=/tmp", "GOCACHE=/tmp/.cache", "
      REDIS_TEST_PASSWORD="]
  stage ('setup') {
    sh '''mkdir -p /tmp/.cache'''
  }

  stage('Build queue-controller-builder Docker image') {
    sh "./scripts/jenkins/build-queue-controller-builder-image.sh
        "
  }

  stage('Build Go code using queue-controller-builder Docker
      image') {
    docker.image('queue-controller-builder:latest').inside {
      withEnv(environment) {
        sh "go mod download" # downloads Go third-party
            dependencies
        sh "./scripts/jenkins/minify-lua-scripts.sh"
        sh "./scripts/jenkins/generate-go-code.sh"
        sh "./scripts/jenkins/restore-lua-scripts.sh"
        sh "./scripts/jenkins/compile-go-protobuf.sh"
        sh "./scripts/jenkins/build-go-code.sh"
      }
    }
  }

  stage('Integration tests') {
    /**
     * In order to communicate with the Redis server, this
```

```
        Pipeline explicitly
     * maps the port (6379) to a known port of the host machine
     */
    docker.image('redis:5.0.0-alpine').withRun() { redis ->
      docker.image('queue-controller-builder:latest').inside("--
          link ${redis.id}:redis") {
        withEnv(["HOME=/tmp", "GOCACHE=/tmp/.cache", "
            REDIS_TEST_PASSWORD=", "REDIS_TEST_URL=redis:6379"]) {
          sh "./scripts/jenkins/test-go-code.sh"
        }
      }
    }
  }

  stage('Create zip artifacts') {
    docker.image('queue-controller-builder:latest').inside {
      withEnv(environment) {
        sh "./scripts/jenkins/artifacts-to-zip.sh"
      }
    }
  }

  stage('Cleanup') {
    deleteDir()
    dir("${workspace}/tmp/.cache") {
      deleteDir()
    }
  }
}
```

Listing A.2: Dockerfile that provides the build environement for every Queue Controller microservice.

```
# This Dockerfile is specifically tailored for Go applications.

# Default Go version
ARG GO_VERSION=1.12.0

FROM golang:${GO_VERSION}

# Protobuf version
ARG PROTOBUF_VERSION=3.8.0

# Terraform version
ARG TERRAFORM_VERSION=0.12.3

# Set current directory
WORKDIR /tmp

# Create the user and group files that will be used in the
    running container to
# run the process as an unprivileged user.
RUN mkdir /user && \
echo 'nobody:x:65534:65534:nobody:/:' > /user/passwd && \
echo 'nobody:x:65534:' > /user/group
```

```
# unzip is required to extract data from zip archives
RUN apt update && apt install -y zip

# Install protobuf to /usr/bin/protoc
RUN curl -L https://github.com/protocolbuffers/protobuf/releases/
    download/v${PROTOBUF_VERSION}/protoc-${PROTOBUF_VERSION}-linux
    -x86_64.zip --output ./protobuf.zip
RUN chmod +x ./protobuf.zip
RUN ls -al .
RUN unzip ./protobuf.zip -d ./protobuf

# Copy protoc executable to /usr/bin
RUN cp -a ./protobuf/bin/. /usr/bin/

# Copy default ProtoBuffers definitions to /usr/include
RUN cp -a ./protobuf/include/. /usr/include/

RUN GO111MODULE=off go get -u github.com/golang/protobuf/protoc-
    gen-go

# Install gogo/protobuf
RUN GO111MODULE=off go get github.com/gogo/protobuf/proto
RUN GO111MODULE=off go get github.com/gogo/protobuf/protoc-gen-
    gofast
RUN GO111MODULE=off go get github.com/gogo/protobuf/gogoproto

# Install ginkgo
RUN GO111MODULE=off go get github.com/onsi/ginkgo/ginkgo
RUN GO111MODULE=off go get github.com/onsi/gomega/...

# Install Terraform
RUN curl -L https://releases.hashicorp.com/terraform/${
    TERRAFORM_VERSION}/terraform_${TERRAFORM_VERSION}_linux_amd64.
    zip --output ./terraform.zip
RUN chmod +x ./terraform.zip
RUN unzip ./terraform.zip -d ./terraform

# Copy terraform executable to /usr/bin
RUN cp ./terraform/terraform /usr/bin/
RUN chmod +x /usr/bin/terraform

# Install AWS Lambda Packager
RUN go get github.com/aws/aws-lambda-go/cmd/build-lambda-zip

# Install Lua runtime
RUN apt install lua5.1

RUN git clone https://github.com/stravant/LuaMinify /luaminify
RUN chmod -R 777 /luaminify

# Create the folder that will contain the AWS Lambda zip file
RUN mkdir /package
```

```
# Tell ‘go build‘ to create a statically linked application
ENV CGO_ENABLED 0

# Tell ‘go build‘ to compile for linux
ENV GOOS linux

# Tell ‘go build‘ to compile for a 64 bit architecture
ENV GOARCH amd64

VOLUME /queue-controller
WORKDIR /queue-controller

RUN chmod 777 -R ${GOPATH}
```

Listing A.3: Bash script needed to build Go code. It's executed in the project Docker image.

```
#!/bin/bash

set -e;

BASEDIR=$(dirname "$0")
PROJECTPATH="$( cd $BASEDIR; cd ../..; pwd -P )"

# Build a statically linked executable to /queue-controller/cmd/${SERVICE}/app
# -ldflags:
#   -s: Omit the symbol table and debug information.
#   -w: Omit the DWARF symbol table.
function build() {
    SERVICE=$1
    echo "Building code"
    go build \
        -ldflags="-s -w" \
        -installsuffix 'static' \
        -o ${PROJECTPATH}/build/package/${SERVICE}/app \
        ${PROJECTPATH}/cmd/${SERVICE}
}


# All the services to compile reside in the cmd folder of the project
find ${PROJECTPATH}/cmd/* -prune -type d | while IFS= read -r directory; do
    echo "Building service $(basename "$directory")"
    build $(basename "$directory")
done
```

Listing A.4: Bash script that builds the Docker image necessary to build the project microservices

```
#!/bin/bash

set -e;
```

```
BASEDIR=$(dirname "$0")
PROJECTPATH="$( cd $BASEDIR; cd ../..; pwd -P )"
IMAGE_NAME="queue-controller-builder"

echo "Creating $IMAGE_NAME Docker image"

# create 'queue-controller-builder' Docker image
docker build -t ${IMAGE_NAME} -f ${PROJECTPATH}/build/docker/
    Dockerfile.jenkins ${PROJECTPATH}
```

Listing A.5: Bash script that searches every Protobuf source file in the project and generates the equivalent Go source file

```
#!/bin/bash

set -e

# Generate ProtoBufferbuild's equivalent Go code
echo "Generating ProtoBuffer's code"
find . -type f -iname '*.proto' | \
    while IFS= read -r line ; \
    do protoc -I=. -I=$GOPATH/src \
        -I=$GOPATH/src/github.com/gogo/protobuf/protobuf \
        --gofast_out=paths=source_relative:. \
        ${line}; \
    done;
```

Appendix B

Success Response

This chapter presents the *successresponse* submodule, which shared among every AWS Lambda microservice and ensures every response from AWS API Gateway follows the same structure.

Listing B.1: Source of the successresponse.go file

```
// successresponse.go
package successresponse

import "encoding/json"

// SuccessResponse defines the structure of the JSON response
    returned in case of success.
type SuccessResponse struct {
        Data interface{}    `json:"data"`
        Meta *MetaResponse `json:"meta,omitempty"`
}

// New builds a new SuccessResponse. `data` accepts the content
    of the data to return to the users,
// whereas `fields` is a variadic variable that accepts Meta
    field definitions.
// If no field is passed to this function, then the resulting
    JSON response won't have any `meta` field.
func New(data interface{}, fields ...MetaResponseField) *
    SuccessResponse {
        mf := metaResponseFields{}
        for _, field := range fields {
                field.f(&mf)
        }

        response := &SuccessResponse{
                Data: data,
        }

        if mf.hasContents {
                meta := &MetaResponse{
                        NextIndex: mf.nextIndex,
                }
```

```
                response.Meta = meta
        }

        return response
}

func (sr *SuccessResponse) ToString() string {
        successResponseJSON, _ := json.Marshal(&sr)
        return string(successResponseJSON)
}
```

Listing B.2: Source of the metaresponse.go file

```
// metaresponse.go
package successresponse

// MetaResponse is the structure of the ‘meta‘ field inside the
    SuccessResponse structure.
type MetaResponse struct {
        NextIndex *uint64 ‘json:"nextIndex,omitempty"‘
}

type MetaResponseField struct {
        f func(*metaResponseFields)
}

type metaResponseFields struct {
        nextIndex *uint64

        // hasContents is a boolean flag that indicates whether
            any of the fields enumerated in metaResponseFields
        // has a value different than the default value (which is
             nil, since every field is a pointer).
        hasContents bool
}

func MetaNextIndex(nextIndex uint64) MetaResponseField {
        return MetaResponseField{func(mf *metaResponseFields) {
                mf.nextIndex = &nextIndex
                mf.hasContents = true
        }}
}
```

Glossary

Advanced Message Queuing Protocol Advanced Message Queuing Protocol (AMQP) and is an open standard application layer protocol. AMPQ is efficient, portable, multichannel and secure, and enables encrypted and interoperable messaging between organizations and application. 38, 87

Amazon Web Services Amazon Web Services is a cloud computing platform that provides customers with a wide array of cloud services. AWS can be defined as a secured cloud services platform that offers compute power, database storage, content delivery and various other functionalities. 1, 87

Application Program Interface An Application Program Interface (API) is a set of routines, protocols, and tools for building software applications. Basically, an API is a contract that specifies how software components should interact. 10, 87

AWS Virtual Private Cloud Amazon Virtual Private Cloud (Amazon VPC) is a logically isolated section of the AWS Cloud where the user launch AWS resources in a configurable virtual network. The user has complete control over his virtual networking environment. Amazon VPC provides advanced security features, such as security groups and network access control lists. 40, 87

Comma-separated Values Comma-separated values (CSV) is a simple file format used to store tabular data, such as a spreadsheet or database. Files in the CSV format can be imported to and exported from programs that store data in tables, such as Microsoft Excel. CSV stands for Comma-separated values. 2, 87

Continuous Integration Continuous Integration (CI) is the process of automating the build and testing of code every time a team member commits changes to version control. CI encourages developers to share their code and unit tests by merging their changes into a shared version control repository after every small task completion. Committing code triggers an automated build system to grab the latest code from the shared repository and to build, test, and validate the full master branch. 67, 87

Domain Name System DNS tands for "Domain Name System." Domain names serve as memorizable names for websites and other services on the Internet. DNS translates domain names into IP addresses, allowing users to access an Internet location by its domain name.. 12, 87

Docker Docker is a container technology that allows developers to package up an application with all the dependencies and the run-time systems it needs. 72

Endpoint The URI that goes after the base URL and points towards the requested API functionality. 10

eXtensible Markup Language Extensible markup language is a format that is used to describe documents and data. 41, 88

Extract, Transform, Load ETL pipeline refers to a set of processes extracting data from one system, transforming it, and loading into some database or data-warehouse. ETL stands for "Extract, Transform, Load", and is the common paradigm by which data from multiple systems is combined to a single database, data store, or warehouse for legacy storage or analytics. 2, 87

File Transfer Protocol File Transfer Protocol (FTP) is a protocol designed for transferring files over the Internet. 57, 87

GET The HTTP method for retrieving resources from a RESTful API. 52

GitLab CI GitLab CI is a continuous integration service integrated with the GitLab repository management platform. 72

HyperText Markup Language HTML stands for "Hypertext Markup Language." HTML is the language used to create webpages. "Hypertext" refers to the hyperlinks that an HTML page may contain. "Markup language" refers to the way tags are used to define the page layout and elements within the page. 18, 87

HyperText Transfer Protocol HyperText Transfer Protocol (HTTP) is the protocol used by websites and APIs communicate over the internet. 14, 87

Integration Test In software testing, an **integration test** performs assertions on groups of units of a software. It has the purpose of verifying whether there are faults in the interaction between the integrated units. Defects may arise when there's a mismatch between the signature of the unit functions and the actual unit arguments, or whenever there's a complex communication involved (for example, a network communication necessary to query a database from a class method unit). 20, 84

JavaScript Object Notation Javascript Object Notation is a data format commonly used for APIs requests parameters and response body. It's human-readable and faster than XML to parse. 41, 87

Jenkins Jenkins is an open-source continuous integration software tool written in Java for testing and reporting on isolated changes in a larger code base in real-time. The software enables developers to find and solve defects in a code base rapidly and to automate testing of their builds. 72

Microservice Whereas a monolithic application is built as a single unit, applications built with the microservice architectural style are composed of multiple services and the interfaces that link those services together. 20, 67, 84

PATCH The HTTP method for applying partial modifications to a resource. 48

POST The HTTP method for creating resources with a RESTful API. 48

PUT The HTTP method for updating resources with a RESTful API. 48

Redis Redis is an open source, non-relational, in-memory data structure store. It is used as a database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, and sorted sets with range queries. 3, 62, 85

Remote Procedure Call Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer on a network without having to understand the network's details. 41, 58, 87

REpresentational state transfer REpresentational State Transfer (REST) is a client-side architectural style for distributed hypermedia systems. It has numerous benefits, as it allows an architecture to be composed of hierarchical layers, it's stateless and cacheable for read operations, and provides a uniform interface to a system. 10, 87

Secure Shell SSH, also known as Secure Shell, is a network protocol that gives system administrators a secure way to access a computer over an unsecured network. 3, 87

Serverless Serverless computing is a method of providing backend services on an as-used basis. A Serverless provider allows users to write and deploy code without the hassle of worrying about the underlying infrastructure. A company that gets backend services from a serverless vendor is charged based on their computation and do not have to reserve and pay for a fixed amount of bandwidth or number of servers, as the service is auto-scaling. 71

SOLID SOLID is an acronym for the first five object-oriented design principles formulated by Robert C. Martin. These principles, when combined together, help a programmer developing software that is easy to maintain and extend. They also guide developers to avoid code smells, easily refactor code, and are also a part of the agile or adaptive software development. The SOLID principles are: Single-responsibility, Open-closed, Liskov substitution, Interface segregation, Dependency Inversion. 38

Structured Query Language The Structured Query Language (SQL) standardized format for querying and managing relational databases.. 2, 87

Test Driven Development Test Driven Development (TDD) is a software engineering practice that requires unit tests to be written before the code they are supposed to validate. 61, 72, 88

Travis CI Travis CI is a hosted continuous integration service used to build and test software projects. 72

Unified Modeling Language In Software Engineering, Unified Modeling Language (UML) is a standardized modeling language enabling developers to specify, visualize, construct and document artifacts of a software system. Thus, UML makes these artifacts scalable, secure and robust in execution. UML is an important aspect involved in object-oriented software development. It uses graphic notation to create visual models of software systems. 21, 88

Unit Test In software testing, a unit test performs assertions on individual units or components of a software. It has the purpose of verifying whether each unit of the software performs as designed. A unit is the smallest testable part of any software, which usually has few inputs and a single output. A unit is often a single function, procedure or class method. 20, 86

User eXperience In design, User Experience (UX) is the process design teams use to create products that provide meaningful and relevant experiences to users. This involves the design of the entire process of acquiring and integrating the product, including aspects of branding, design, and usability. 4, 88

UI In design, User Interface (UI) is the process of making interfaces in software or computerized devices with a focus on looks or style. Designers aim to create designs users will find easy to use and pleasurable. UI design typically refers to graphical user interfaces but also includes others, such as REST interfaces. 9, 88

Acronyms

AMQP Advanced Message Queuing Protocol. 83

API Application Program Interface. 83

AWS Amazon Web Services. 8, 15, 16, 83

AWS EC2 AWS Elastic Compute Cloud. 8

AWS ECR AWS Elastic Container Registry. 15

AWS ECS AWS Elastic Container Service. 15

AWS KMS AWS Key Management Service. 16

AWS S3 AWS Simple Storage Service. 8

AWS VPC AWS Virtual Private Cloud. 83

CI Continuous Integration. 83

CSV Comma-Separated Values. 83

DNS Domain Name System. 83

ETL Extract, Transform, Load. 84

FTP File Transfer Protocol. 84

HTML HyperText Markup Language. 84

HTTP HyperText Transfer Protocol. 84

JSON JavaScript Object Notation. 84

REST REpresentational state transfer. 85

RPC Remote Procedure Call. 85

SQL Structured Query Language. 85

SSH Secure Shell. 85

TDD Test Driven Development. 85

UI User Interface. 86

UML Unified Modeling Language. 85

UX User eXperience. 86

XML eXtensible Markup Language. 84

Bibliography

Bibliography references

[9] E. Amodeo. *Learning Behavior Driven Development with JavaScript.* Packt Publishing, 2015 (cit. on p. 61).

[11] Z. Dennis et al D. Chelimsky D. Astels. *On the Effectiveness of Unit Test automation at Microsoft.* Pragmatic Bookshelf, 2010 (cit. on p. 61).

[20] Daniel T. Jones James P. Womack. *Lean Thinking, Second Editon.* Simon & Schuster, Inc., 2010.

[21] S. Kollanus. *Test Driven Development - still a promising approach?* IEEE, 2010 (cit. on p. 61).

[22] N. Nagappan L. Williams G. Kudrjavets. *On the Effectiveness of Unit Test automation at Microsoft.* 2009 (cit. on p. 61).

[24] M. A. Gerosa M. F. Aniche. *Most common mistakes in test-driven development practice: Results from an online survey with developers.* IEEE, 2010 (cit. on p. 61).

[37] M. C. F. P. Emer W. Bissi A. G. S. S. Neto. *The effects of test driven development on internal quality, external quality and productivity: A systematic review.* 2016 (cit. on p. 61).

Web sources

[1] Amazon. *Amazon API Gateway.* URL: `https://aws.amazon.com/api-gateway/` (cit. on p. 14).

[2] Amazon. *Amazon CloudWatch.* URL: `https://aws.amazon.com/cloudwatch/` (cit. on p. 15).

[3] Amazon. *Amazon Elastic Cloud Registry.* URL: `https://aws.amazon.com/ecr/` (cit. on p. 15).

[4] Amazon. *Amazon Elastic Cloud Service.* URL: `https://aws.amazon.com/ecs/` (cit. on p. 16).

[5] Amazon. *Amazon Key Management Service.* URL: `https://aws.amazon.com/kms/` (cit. on p. 16).

[6] Amazon. *Amazon Lambda*. URL: https://aws.amazon.com/lambda/ (cit. on p. 14).

[7] Amazon. *Amazon Secrets Manager*. URL: https://aws.amazon.com/secrets-manager/ (cit. on p. 16).

[8] Amazon. *Configuring a Lambda Function to Access Resources in an Amazon VPC*. URL: https://docs.aws.amazon.com/lambda/latest/dg/vpc.html (cit. on p. 40).

[10] David Cassel. *Go: the progrmaming language of the cloud*. URL: https://thenewstack.io/go-the-programming-language-of-the-cloud/ (cit. on p. 16).

[12] Docker. *What is a Container?* URL: https://www.docker.com/resources/what-container (cit. on p. 11).

[13] Netflix Engineering. *Netflix likes React.js*. URL: https://medium.com/netflix-techblog/netflix-likes-react-509675426db (cit. on p. 18).

[14] Facebook. *React.js website*. URL: https://reactjs.org/ (cit. on p. 18).

[15] GoGo. *GoGo Protobuf Github page*. URL: https://github.com/gogo/protobuf (cit. on p. 42).

[16] Google. *Go Documentation*. URL: https://golang.org/doc/ (cit. on p. 16).

[17] Google. *Protocol Buffer documentation*. URL: https://developers.google.com/protocol-buffers/ (cit. on p. 41).

[18] Hashicorp. *Introduction to Terraform*. URL: https://www.terraform.io/intro/index.html/ (cit. on p. 12).

[19] IEEE. *ISO/IEC 12207:2008*. URL: https://klevas.mif.vu.lt/~adamonis/iso/ISO_IEC_12207_2008(E)-Character_PDF_document.pdf.

[23] Connor Leech. *Companies in the Bay Area that use React.js*. URL: https://medium.com/employbl/companies-in-the-bay-area-that-use-react-js-841d04d0f0a8 (cit. on p. 18).

[25] Geshan Manandhar. *The best automated deployment tool is... the one that fits your needs*. URL: https://dev.to/geshan/the-best-automated-deployment-tool-is-the-one-that-fits-your-needs-3o8 (cit. on p. 57).

[26] *Manifesto Agile*. URL: http://agilemanifesto.org/iso/it/.

[27] Carlos Mauri. *7 benefits of using SASS over conventional CSS*. URL: https://www.mugo.ca/blog/7-benefits-of-using-SASS-over-conventional-CSS (cit. on p. 55).

[28] Microsoft. *Introduction to TypeScript*. URL: https://www.typescriptlang.org/ (cit. on p. 17).

[29] Redis. *Introduction to Redis*. URL: https://redis.io/topics/introduction (cit. on p. 13).

[30] Redis. *Redis Hashes*. URL: https://redis.io/topics/data-types-intro#redis-hashes (cit. on p. 13).

[31] Redis. *Redis Lists*. URL: https://redis.io/topics/data-types-intro#redis-lists (cit. on p. 13).

[32] Redis. *Redis LRANGE command documentation*. URL: https://redis.io/commands/lrange (cit. on p. 43).

[33] Redis. *Redis SCAN command documentation*. URL: https://redis.io/commands/scan (cit. on p. 44).

[34] Redis. *Redis Sets*. URL: https://redis.io/topics/data-types-intro#redis-sets (cit. on p. 13).

[35] Redis. *Redis Strings*. URL: https://redis.io/topics/data-types-intro#redis-strings (cit. on p. 13).

[36] StackOverflow. *Developer Survey 2019: Most Popular Programming, Scripting, and Markup Languages*. URL: https://insights.stackoverflow.com/survey/2019#technology-_-programming-scripting-and-markup-languages (cit. on p. 17).