UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

Corso di Laurea Magistrale in Informatica

Serverless application for multiple datebases governance on AWS

Relatore:
Elena PAGANI
Correlatore:
Roy S. HALSTEAD

Tesi di Laurea di:
Michael DANIEL NAGUIB
Matricola: 923425

Anno Accademico 2020/2021

To Francesco,
a dear friend who has always helped me.
To my Brother-in-law,
who made me laugh whenever I was feeling down.
To my Sister,
who was always there when I needed her.
To my Father,
who has always supported me.
To my Mother,
the brightest Sun,
may she always shine on the path before me.

Contents

List of Figures

List of Tables

List of Code Samples

Chapter 1

Introduction

The problem of centralizing the management of large-scale manufacturing data plagues many companies that have decided to embark on the Digital Transformation of their infrastructure. For this thesis project, I was hired by Deloitte Touche Tohmatsu, one of the Big Four in auditing and consulting, to work on the Cloud Transformation process of a major client company in the manufacturing sector. This transformation consisted of creating a cloud-based platform for the company's Data Management department, where users could seamlessly interact with databases distributed nationally or internationally.

The existing infrastructure that was to be replaced consisted of an on-premise data platform that was not designed for the amount of data being generated across all of the client's factories scattered throughout the territory. This solution was not flexible with respect to the various databases it had to manage, because for each new database, and for each new table associated with it, a new interface had to be implemented to manage the data. Furthermore, it did not place any control on the quality of the data entered in the various tables, did not provide any possibility to extract the data for use elsewhere, did not place any restrictions on what operations were available, nor to whom these tables were accessible. Continuing to do development for this platform had become unsustainable for the client, as each new page that had to handle a single table took a long time to develop, as well as being a very expensive process.

The Data Entry Tool project stands as an alternative to the data entry and manipulation platform. The new web application that was to be developed had to be a plug and play solution that could handle any connected database, it had to provide a system to control access to tables, along with the ability to group them as needed, it had to provide a level of validation for data types and impose any constraints on values, give the ability to import and export data in bulk, while also providing a logging system for each operation performed on the data. All of this should be available in a graphical interface that adapts to user permissions and table structures.

In the following chapters I will introduce the concepts behind Cloud Transformation, such as different infrastructure models and service models. Emphasis will be placed on microservices infrastructure and Serverless architecture, as they lie at the heart of the Data Entry Tool. Next, the tools used within the project will be described, from the Cloud Provider to the programming language chosen for development. Going forward, I will show a deep dive of the implemented architecture, going from the Back-End and Front-End sub-architectures to the deployment system of the entire application. In addition, the most important elements of the implementation I have done will also be explained. As it will be explained in this chapter, in the Back-End section I mainly dealt with managing multiple databases, collecting data to facilitate the dynamic creation of tables, and defining the schema used by the database that manages the application, while for the Front-End section I created a standard structure to manage static

interfaces; moreover, I dealt with the Continuous Integration and Continuous Delivery of the application.

Towards the end, I will show some implementation gimmicks and the impacts they had on application performance, along with comparisons between the implemented solution and the pre-existing one to verify that the newly developed application had met the requirements. Finally, I will present the final considerations, as well as some references to possible future developments.

Chapter 2

Reference Technologies

The platform chosen to host the new web application is Amazon Web Services [1]. The choice was forced by the fact that the customer data platform was already in place and the application is closely tied to that environment. Before diving into the details of the project, let us see an overview of the reference technologies that power AWS.

2.1 Cloud Infrastructure

Cloud Infrastructure is a term used to describe the set of components, like hardware, abstracted resources, storage and network resources needed for cloud computing; think of it like the tools needed to build a cloud. Let us start with a definition of Cloud Computing:

Definition. *Cloud computing is a remote, virtual pool of on-demand, shared resources that can be rapidly deployed at scale.[2]*

The basic concept to understand from this definition is the *virtualization*: this is an abstraction technology, used to separate resources from physical hardware and pool them into clouds; in essence, it allows you to create multiple virtual machines, all of which run on a single server. Each one has its own operating system and set of applications, and can run all at the same time without being aware of each other's existence.

With virtualization, an organization can reduce its Capital Expenditure (CapEx), namely the funds used to acquire, upgrade and maintain the physical hardware of on-premise servers, and thus reduce its Operational Expenditure (OpEx), as it would not need to rent a facility to house them or pay salaries for any maintainer because all the costs involving the hardware are borne by the cloud provider.

There are 4 main types of resources that can be built with these virtual machines:

- Compute resources provide on-demand processing power to complete a workload; these are used to run programs or scripts in the cloud. An equivalent in the traditional on-premise environment would be the physicals servers that you would connect to remotely, install, and run software on.

- Storage resources allow to save and retrieve unstructured data; these are resources that allow you to store things like images or videos in the Cloud. Here the equivalent would be a DAS (direct attached storage), NAS (Network attached storage), or SAN (storage area network).

- Database resources allow to store structured sets of data to be used in an application.

- Network resources allow to control how all the other resources will communicate with each other. In a data center, this role would be taken by routers, switches, firewalls, and load balancers.

2.1.1 Cloud Service Models

A Cloud Vendor offers different type of services to the customers, where "as-a-Service" generally means a cloud computing service that is managed in lieu of the customer.

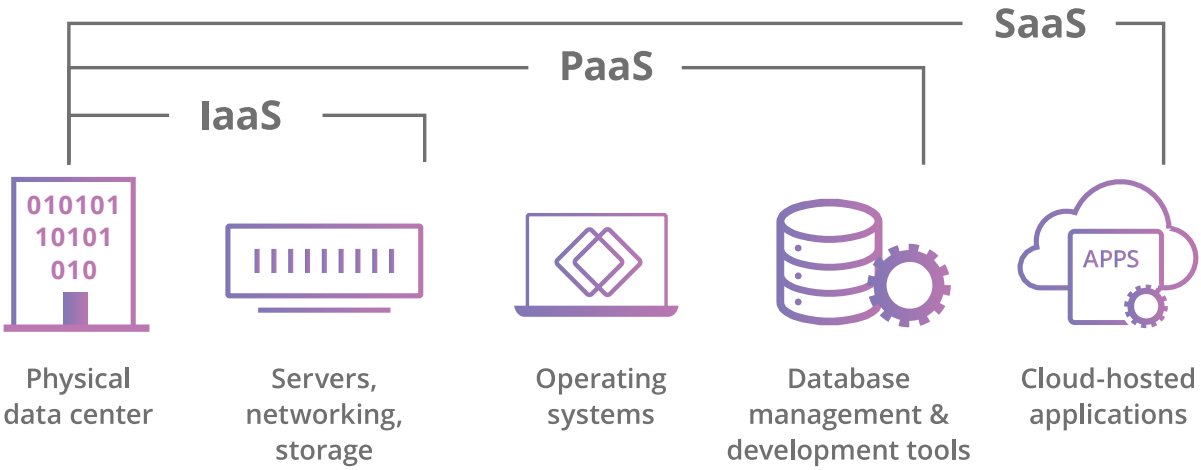


Figure 2.1: The elements included with each type of service. [3]

Infrastructure-as-a-Service (IaaS)

This is the most basic model where the cloud provider hosts the infrastructure, providing virtualization, storage, servers, wires and appliances that make connections between those machines, on behalf of their customers, essentially taking care of the hardware, so that the user does not have to have an on-premise data-center and does not have to worry about physically updating and maintaining those components; however, it is the responsibility of the customer to handle the applications, data, operating system, middle-ware, and runtimes.

As shown in Figure 2.2, the user can access the infrastructure via any Internet connection as opposed to on-premise solution, where the user has to physically be in the data-center.

Platform-as-a-Service (PaaS)

The cloud provider hosts the hardware and provides an application software platform, such as the operating system. It is primarily intended for developers and programmers as it allows them to develop, run, and manage their own apps without having to build and maintain the infrastructure or the platform. PaaS can be accessed over any Internet connection, making it possible to build an entire application in a web browser.

Among the tools provided with PaaS are:

- a set of development tools, sometimes offered together as a framework, including, a source code editor, a debugger, a compiler, and other essential tools.
- Middleware that sits in between user-facing applications and the machine's operating system, which is necessary for running the application, but end users do not interact with it.

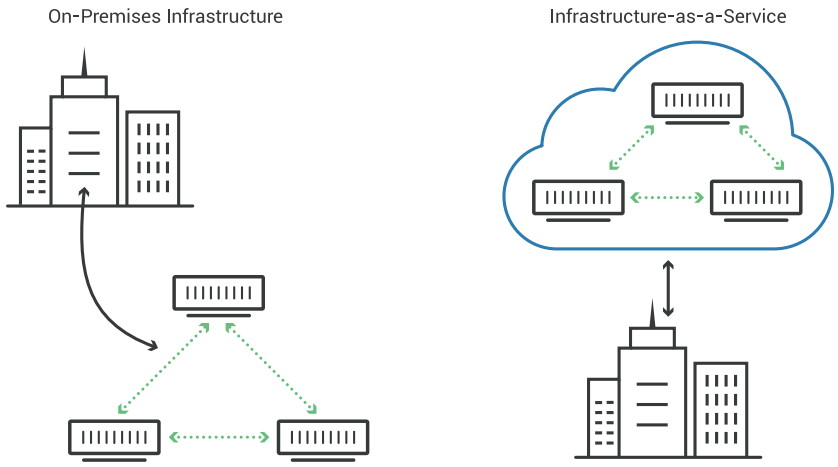


Figure 2.2: On-Premise vs. IaaS. [4]

- Operating Systems where the developers work on and the application run on.

- Databases administered and maintained by the vendor, which usually provides developers with a database management system.

This service is often used by developers for a number of reasons, like the ability to cut coding time by leveraging pre-coded application components built into the platform, the support for geographically distributed teams because the development environment is accessed over the Internet, and the efficient management of the application life-cycle by building, testing, deploying, managing, and updating within the same integrated environment.

Software-as-a-Service (SaaS)

This service is a form of cloud computing where users subscribe to an application rather than purchasing it once and installing it. Figure 2.3 shows that the actual application logic runs in the cloud and many users can log into it and use it from any compatible device. The SaaS model reduces the users' upfront costs by eliminating the need to permanently purchase software, however they should invest in fast network hardware since service performance is determined by Internet connection speeds.

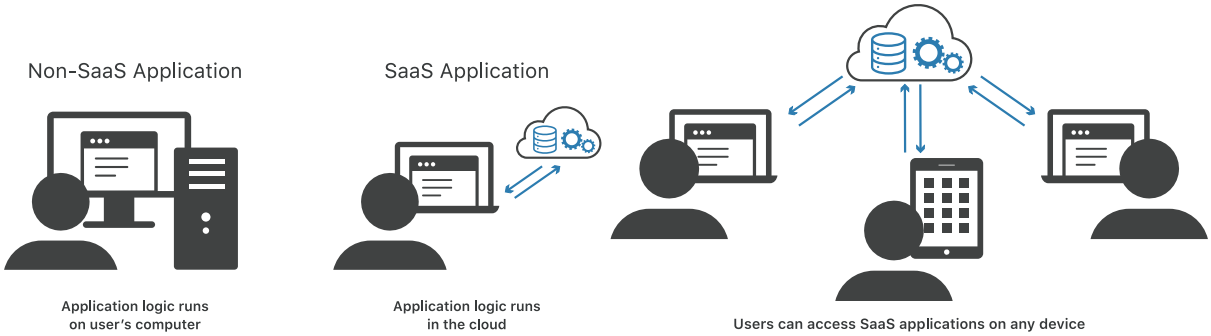


Figure 2.3: Where the application logic lies. [5]

2.1.2 Private, Public, and Hybrid

If the models above defined how the services are offered via the cloud, then these different cloud deployment models define where the cloud server are and who manages them.

Private Cloud

A private cloud is a server, data-center, or distributed network entirely dedicated to one organization. By using a private cloud, an organization can experience the benefits of cloud computing without sharing resources with other organizations. As shown in Figure 2.4, this type of cloud can either be located inside an organization (on-premise) or remotely managed by a third party and accessed over the Internet (off-premise). It should be noted that an internal private cloud is not the same as a traditional data-center, because, as a cloud model, it boasts all the features of the cloud architecture, like virtualization, which makes them more efficient, more powerful, and more scalable.

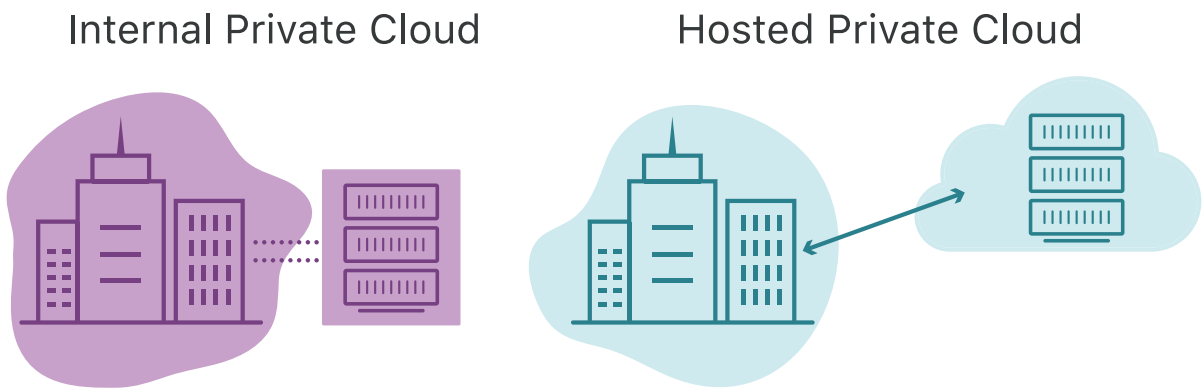


Figure 2.4: Different locations for the private cloud. [6]

A reason to use the private cloud model may depend on a company's security standard because it eliminates inter-company *multi-tenancy*, which is when multiple customers of a cloud provider are accessing the resources provided by the same physical server, like data or processes. This gives a company more control over the cloud security measures that are put in place, despite the increase in cost that this approach will need.

Public Cloud

A public cloud is a pool of virtual resources that is automatically provisioned and allocated among multiple clients through a self-service interface.

The distinction from private cloud model, highlighted in Figure 2.5, relies on the fact that the private model is a cloud service that is not shared with any other organization, which avoids multi-tenancy and increases data protection; by contrast, a public cloud is a cloud service that shares computing services among different customers, while protecting and hiding each customer's data and applications from one another.

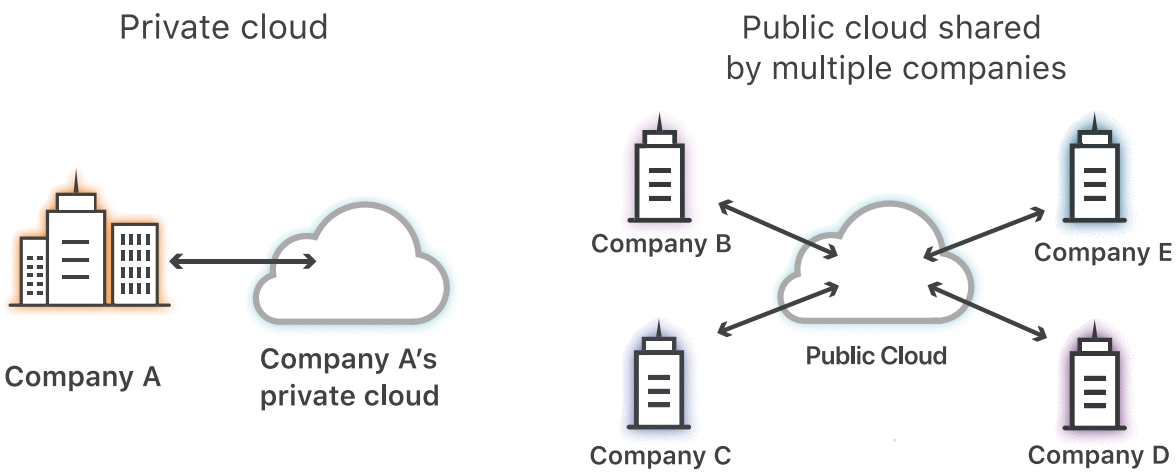


Figure 2.5: Distinct utilization for private and public cloud models. [7]

Hybrid Cloud

A hybrid cloud mixes two or more types of cloud environments, which combines public and private cloud, both on-premise and hosted. These different cloud environments must be tightly interconnected with each other, essentially functioning as one combined infrastructure.

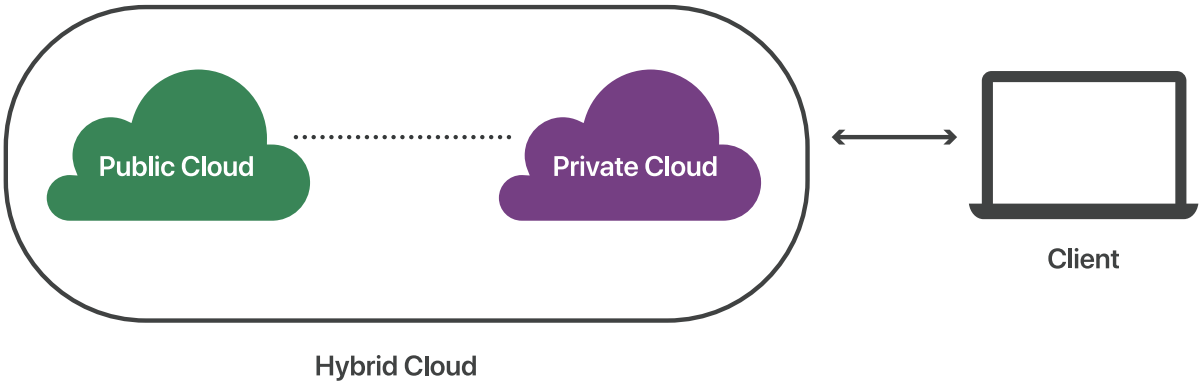


Figure 2.6: Each cloud environment is presented as one to the client. [8]

2.1.3 Microservice Architecture

Before cloud and cloud computing came to be, the classic way to build an application was with a Monolithic Architecture, usually hosted on a specific server or set of servers, which consist of a single stack: the User Interface on top, the Business Logic in the middle, and the database on the bottom. Such architecture has several disadvantages; for example, any change, does not matter how minimal, means the entire stack has to be updated, or if a portion of the application breaks, the entire application might fail.

With the cloud, the idea came forth of dividing a larger application in smaller, independent services, thus a Microservice Architecture was born. A microservice is a smaller portion of an application, that performs one service only, runs in its own environment independently from other parts of the application, and accesses its own data. Figure 2.7 shows a comparison between the two approaches; we can see that a user will not be able to distinguish between them.

This kind of architecture has several advantages:

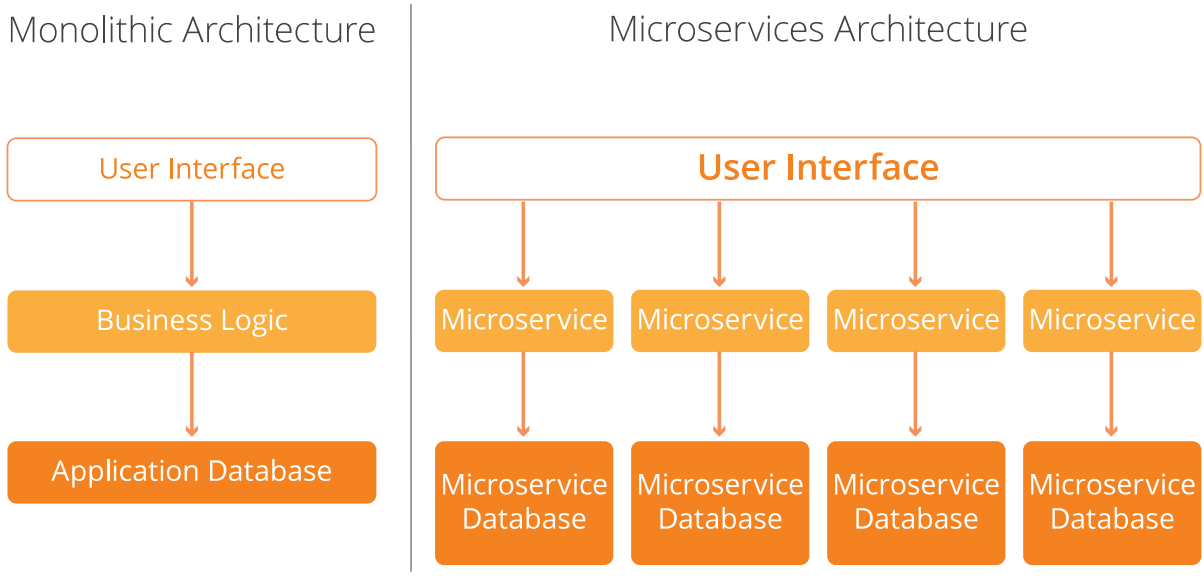


Figure 2.7: Side-by-side comparison between the architectures. [9]

- Resilience: as the services are independent from one another, the breaking or crashing of a service does not affect the rest of the application.
- Selective scalability: instead of scaling the entire application, only the service that receive a large amount of usage can be scaled.
- Easier to handle change: new features or updates can be rolled out one at a time, instead of re-deploying the entire application stack.
- Flexibility for developers: each service is independent, and as such could be written in different languages.

The main advantage of building an application with a Microservice Architecture in the cloud is most evident when used in conjunction with a Serverless architecture which allows a service, or even a function, to be executed only when needed.

2.1.4 Serverless Architecture

Definition. *Serverless computing is a method of providing back-end services on an as-used basis.*[10]

From this definition of Serverless, we can infer that the resources needed to provide computing power for a back-end infrastructure are not reserved and paid for upfront, instead they are provided on-demand and charged based on usage. There are still servers in serverless, but they are abstracted away from the application development.

The Serverless provider manages all the underlying infrastructure, allowing the users to only worry about code development and deployment, while ensuring auto-scaling of the service in case of peak activity. The Figure 2.8 shows the difference in costs between a classic cloud approach and a Serverless one. With traditional servers, developers and companies purchase a fixed number of servers, or an amount of server space, to ensure that a spike in traffic or activity does not exceed their monthly limits, thus breaking the application; this would mean that part of the purchase could be wasted. On the other hand, with the Serverless approach they can purchase back-end services on a pay-as-you-go basis.

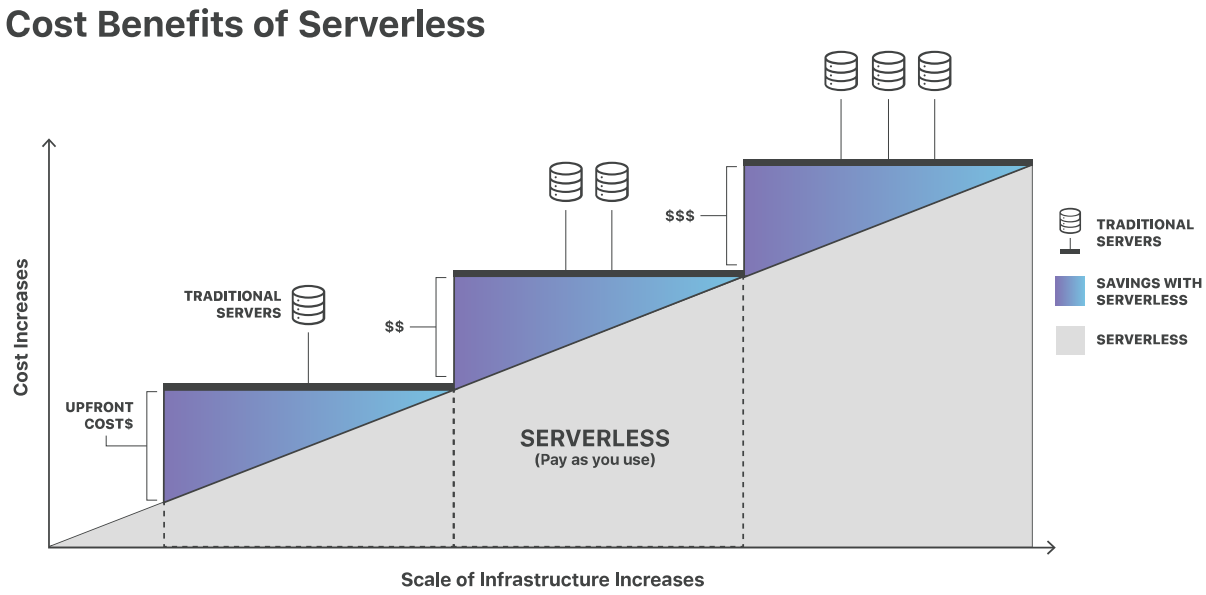


Figure 2.8: A representation of the benefits of Serverless. [10]

2.1.5 Function-as-a-Service

Function-as-a-Service (FaaS), or Serverless computing, is an event-driven execution model that runs in stateless containers that allow developers to build, compute, run, and manage application packages as functions without having to maintain their own infrastructure. This is a serverless way to implement a microservice architecture, and since this functions are stateless, they are inherently scalable.

Figure 2.9 shows the breakdown of a monolithic application in distinct microservices, which in turn are divided in functions where each is an implementation of FaaS.

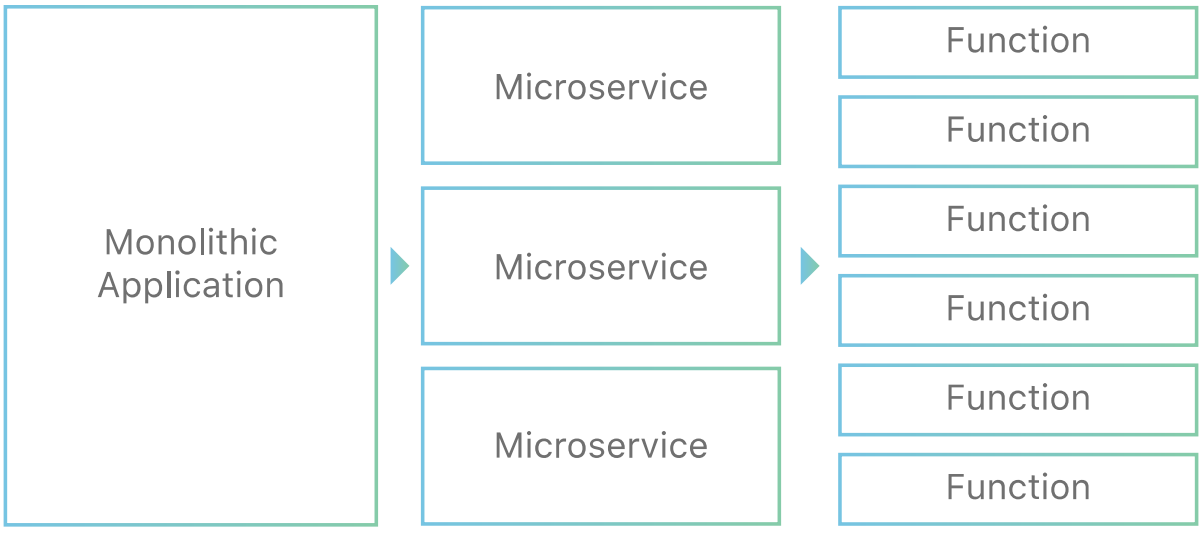


Figure 2.9: Breakdown of a monolithic application into simple functions. [11]

We can see that, by using FaaS as building blocks of microservices, we can build a truly modular application.

PaaS vs. Serverless computing

PaaS and Serverless computing are similar in that for both a developer has to worry only about writing and uploading code, while the vendor handles all the back-end processes. However, there are key points of difference:

- Scaling is vastly different when using the two models since applications built using Serverless computing, or FaaS, will scale automatically, whereas PaaS applications will not unless programmed to do so.
- While PaaS applications are more like traditional applications, and have to be running most of the time or all of it in order to be immediately available for users, Serverless applications can be up and running almost instantly.
- With Serverless computing vendors do not provide development tools or frameworks, while they do with PaaS.
- PaaS billing is not nearly as precise as in Serverless computing, in which charges are broken down to the number of seconds or fractions of a second each instance of a function runs.

2.2 Amazon Web Services

This section will outline the key AWS services used in the project.

AWS API Gateway

In general, an API gateway is an API management tool that sits between a client and a collection of back-end services, and acts as a reverse proxy to accept all API calls, aggregate the various services required to fulfill them, and return the appropriate result.

Amazon API Gateway [12] is a service for building comprehensive RESTful API that are HTTP-based, that enable stateless client-server communication, and implement standard HTTP Create, Read, Update, and Delete methods.

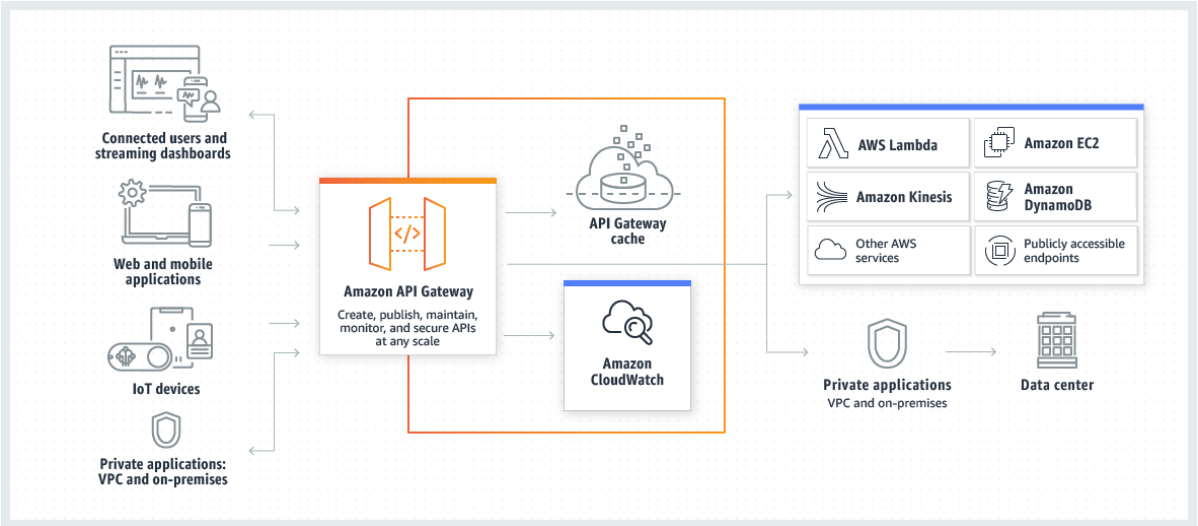


Figure 2.10: API Gateway diagram. [12]

The diagram in Figure 2.10 illustrates how the API Gateway provides a developer with an integrated and consistent development experience for building AWS Serverless applications. API Gateway handles all the tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls, essentially acting as a "front door" for applications to access data, business logic, or functionalities from the back-end services.

AWS Lambda

AWS Lambda [13] is an implementation of FaaS, or Serverless computing. As mentioned in 2.1.5, code can run for virtually any type of application or back-end service, and the Lambda takes care of everything required to run it and scale it with high availability.

Amazon Aurora

Amazon Aurora [14] is a relational database engine that combines the speed and reliability of high-end commercial databases with the simplicity and cost-effectiveness of open source databases. As shown in Figure 2.11, it is designed to replicate 6 copies of the data across 3 different Availability Zones, thus creating a distributed, fault-tolerant, and self-healing storage system.

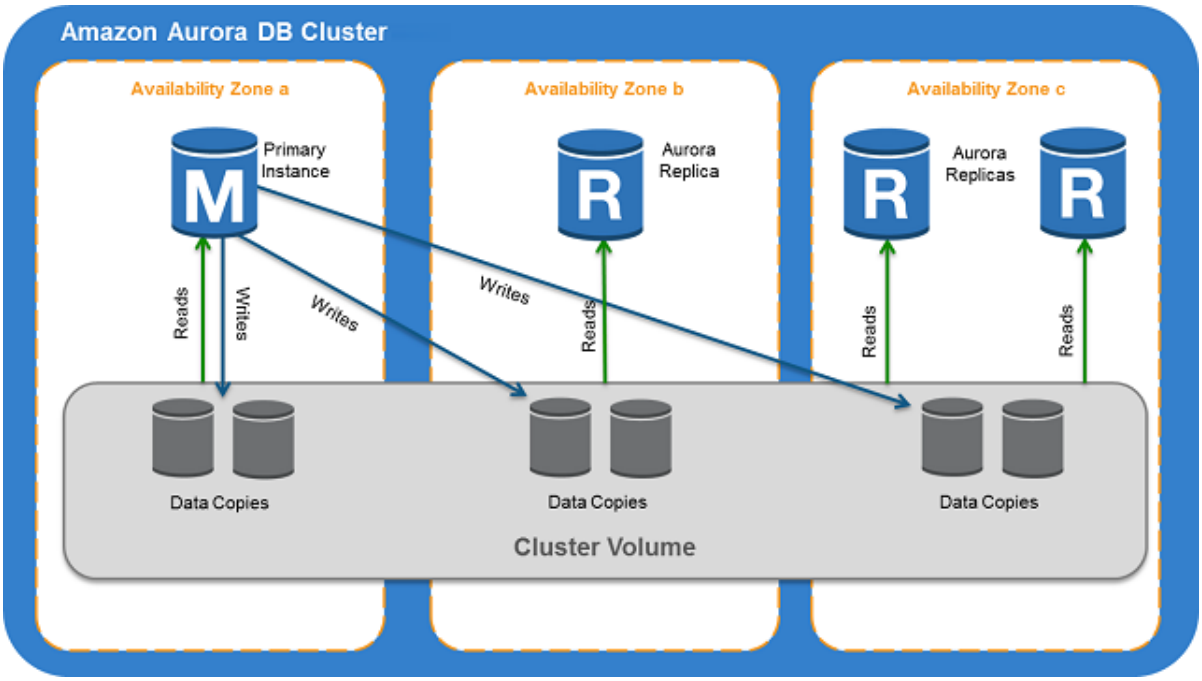


Figure 2.11: Aurora data replication schema. [15]

Amazon Simple Queue Service

SQS [16] is a managed queuing service that enables the secure message exchange, decoupling and scaling of microservices and serverless applications from one another. With it, it was possible to perform the logging of the operations performed and the asynchronous invocation of actions, while also providing a near real-time experience for the users.

Amazon Elastic Container Registry

Amazon ECR [17] is a managed container registry used by developers to share and deploy container images, keep track of the created versions, and manage their lifecycle. Moreover, these images are transferred to the registry via the HTTPS protocol and are encrypted at rest.

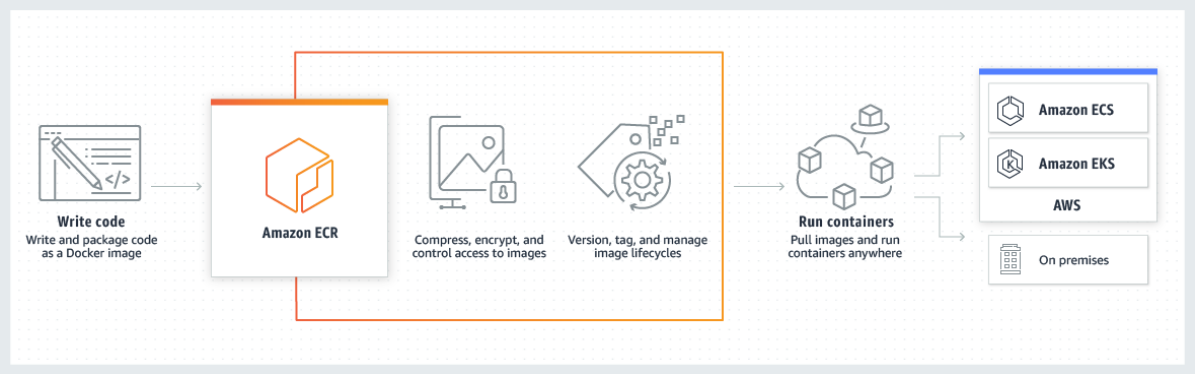


Figure 2.12: ECR diagram. [17]

Amazon CloudFormation

CloudFormation [18] is a service that uses template files to automate the configuration of AWS resources. As further defined in 2.4.2, it can be described as an Infrastructure-as-Code tool and a Cloud Automation solution, which is the use of automated tools and processes to perform workflows in a cloud environment that would otherwise have to be performed manually, such as configuring servers or setting up a network, thereby enabling the automated setup and deployment of virtually any AWS service.

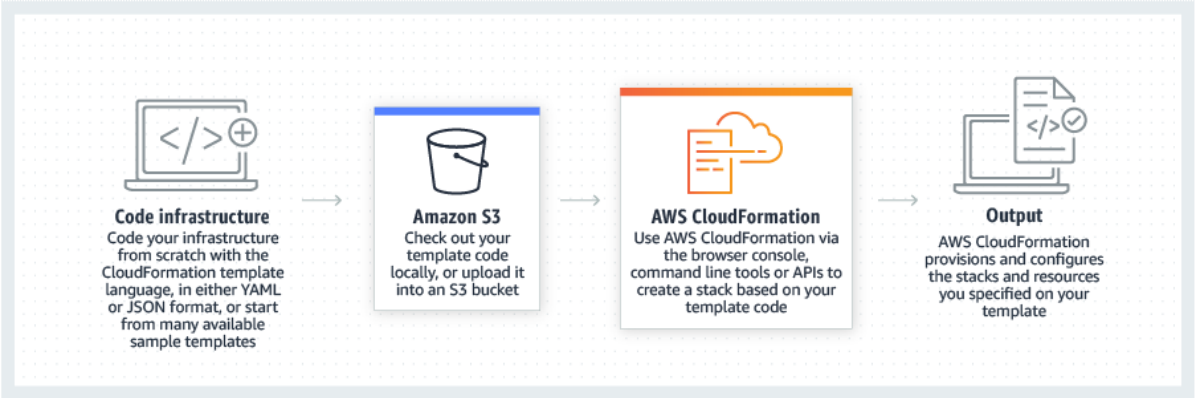


Figure 2.13: CloudFormation deployment process. [18]

2.3 Big Data Platform

In the process of Digital Transformation of the customer, aimed at digitizing processes and enabling new business models, we find the Integrated Data Platform, whose goal is to make the most of the data produced every day by the company and turn them in valuable elements for the business. It is the cornerstone of the business' data strategy that ensures centralized

information governance and enables coexistence and inter-operability of on-premise hosted, private and public cloud data resources.

The main benefits of the Data Platform will be:

- Data-driven approach that will support the decision making process.
- Smart Manufacturing that will allow factories to monitor, analyze, and improve production.
- A centralized, integrated location that helps information cross between central systems and factories.
- Predictive Analysis applicable to both production facilities and new types of products enabling the creation of new business models.
- The Data Lake, with its large amount of data, computing power and services offered by the Cloud Provider enable the use of AI/ML.

The Data Platform is designed to support the numbers shown in Figure 2.14 in terms of integrated data and their processing:

Figure 2.14: Amount of data and processing power of the Data Platform.

- The platform handles the loading of more than 60 TB/Month composed of both Near Real Time and event-driven Batch streams.
- Since the Data Platform must retain a 5-year history and based on the amount of data produced each day, it is estimated to have a Data Lake with a size rater than 4 PB.
- To support data loading, transformation, consolidation, and visualization, we are provided with a high computing power of more than 600 vCPUs.

2.4 ToolBox

Several development tools were required for this project in addition to the cloud infrastructure. This section will give a brief introduction of each tool.

2.4.1 Node.js

Among the supported languages for AWS Lambda there is also Node.js [19], an open-source, cross-platform, back-end JavaScript runtime environment that runs on Chrome's V8 engine and executes JavaScript code outside of a web browser.

Definition. *Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. [20]*

It is designed to build scalable network applications by leveraging its single-threaded and asynchronous event-driven runtime. A Node.js application runs in a single process, without creating a new thread for each request; when it performs an I/O operation, instead of locking the thread and wasting CPU cycles waiting, Node.js will make use of a thread taken from a thread pool to perform the task and will resume the operations when the response comes back. This allows Node.js to handle thousands of concurrent connections with a single server without the burden of handling thread concurrency, which could be a significant source of bugs. This is possible thanks to the Event Loop, depicted in Figure 2.15.

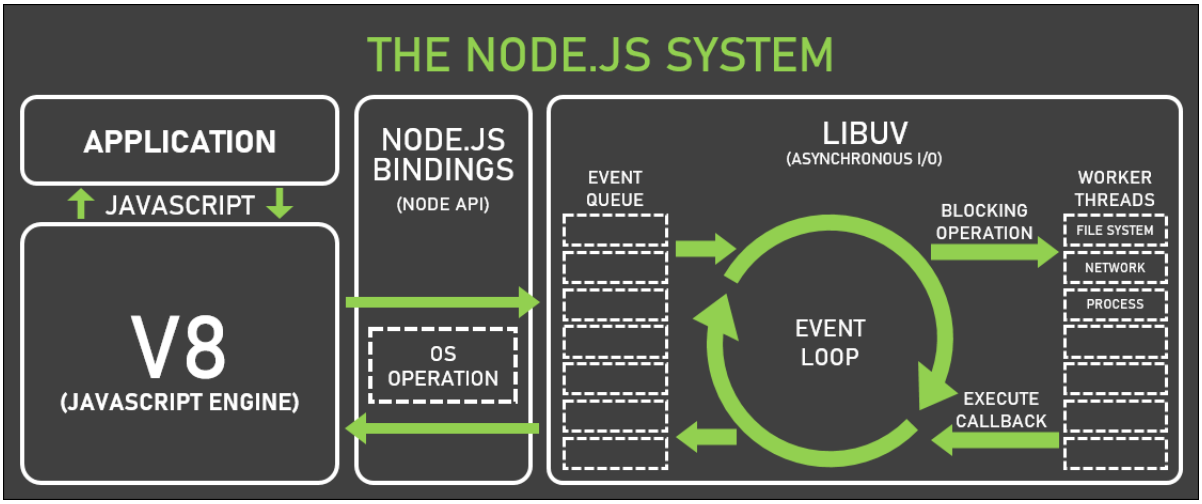


Figure 2.15: Event Loop that powers Node.js. [21]

TypeScript

The language used for back-end and front-end development is TypeScript [22], which is an object-oriented super-set of JavaScript that was developed to overcome code complexity for large projects.

Javascript is a dynamically typed language, meaning that the software will not treat type differences as errors until runtime, which often results in bugs. TypeScript on the other hand offers optional static typing. Once static typing is declared, a variable does not change its type, and during compilation, the compiler alerts developers to any type of errors (syntactic or semantic), which results in early bug detection.

2.4.2 Infrastructure as Code

Infrastructure provisioning has historically been a time-consuming and expensive manual process, completed by accessing a management portal and using a point-and-click approach to deploying resources. Infrastructure as Code (IaC) is the managing and provisioning of infrastructure through code instead of manual processes. With this approach, configuration files that contain the infrastructure specifications are created, thus ensuring that the same environment is provisioned every time. By deploying the infrastructure as code also means that it is possible to divide it into modular components that can be automatically combined in different ways, and since it is provisioned via code, the configuration can be added to the version control system.

The ultimate goal of IaC is automation as it is possible to integrate the process of provisioning, or dismantling, resources in any DevOps flow.

Serverless Framework

To this end, the Serverless Framework [23] was used, which uses a command-line tool and configuration files to deploy both code and cloud infrastructure leveraging Cloud Formation templates, while also managing the lifecycle of the serverless architecture.

2.4.3 OpenAPI Specification

Definition. *The OpenAPI Specification defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. [24]*

With it, the API for each Lambda function was defined describing its path, HTTP method, possible path or query parameters, an authorizer to check user permissions to access a resource, validation of the request body, and the body of the response to the API call.

2.4.4 Flyway

Flyway [25] is an open-source database migration tool, and as such it allows you to transfer data from one type of database to another. However, we did not use it for data migration in the strict sense, but rather as a tool that allowed us to add changes to the database schema to the versioning system. With this tool, it is not necessary to interact directly with the Aurora instance to update the schema. At the time of migration, Flyway would check the back-log saved in the database itself, comparing the result of previous migrations with the current state of the schema.

To use Flyway's migration capability, migration files must be written in SQL and use the following naming convention:

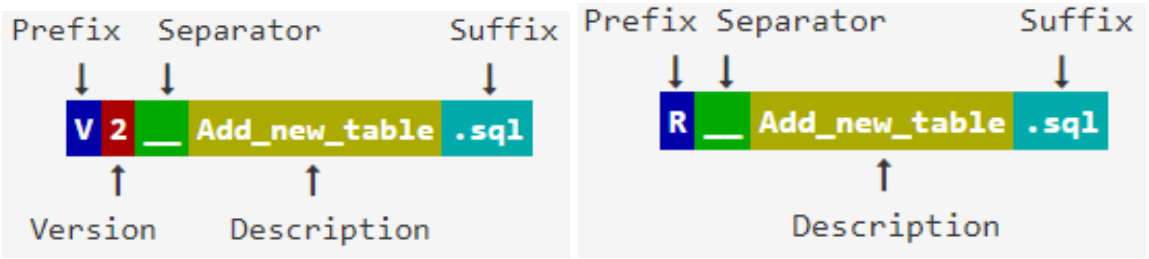


Figure 2.16: Naming convention for, respectively, versioned and repeatable migrations. [26]

- A prefix that represents the type of migration.
  - **V** for versioned and **R** for repeatable.
- in case of a versioned migration, a version number to distinguish one version from another; the use of semantic versioning can help to keep track of the change introduced.
- Two underscores as separator.
- A description of the introduced change.

- The file suffix.

The versioned migrations are applied in order exactly once and each version migration has a unique version, a description, and a checksum that is used to detect accidental changes to an already applied version. In the project, this kind of migration has been used to keep record of the schema changes.

Repeatable migrations are applied after all pending versioned migrations have been executed and in the order of their description. Like the versioned migrations, they have a description and a checksum, but no version; moreover, they are applied whenever their checksum changes. In the project, they are used for the creation of views.

2.4.5 React

React [27] is a declarative, efficient, and flexible JavaScript library for building interactive user interfaces. It lets you compose complex UIs from small and isolated pieces of code called *components*. We used it with the AntDesign Pro framework to build the front-end side of the application.

AntDesign Pro

AntDesign Pro [28] is an open source code for enterprise-level UI design languages and React UI library. It comes with a set of high-quality React components, with theme customization capability.

2.4.6 DevOps

As a DevOps platform we used GitLab to host the GIT repository, and its CI/CD tool for the schema migration with Flyway, the deployment of the serverless infrastructure, and the build and deployment of the front-end application; we created 2 main branches and several development branches:

- A Master branch that contains the production environment that has been thoroughly checked and tested.
- A Dev branch that contains the developed features that were to be tested before the merge into the Master branch and from where feature branches spread out to follow their development.

The deployment of each branch consists of a pipeline definition that utilizes custom Docker images that are identical to the development environment.

2.4.7 Docker

A container is a standard unit of software that packages code and all its dependencies so that the application runs quickly and reliably from one computing environment to another. A Docker container image [29] is a lightweight, self-contained, executable software package that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings. As shown in Figure 2.17, containers isolate the software from its environment and ensure that it runs smoothly despite differences, for example, between development and production.

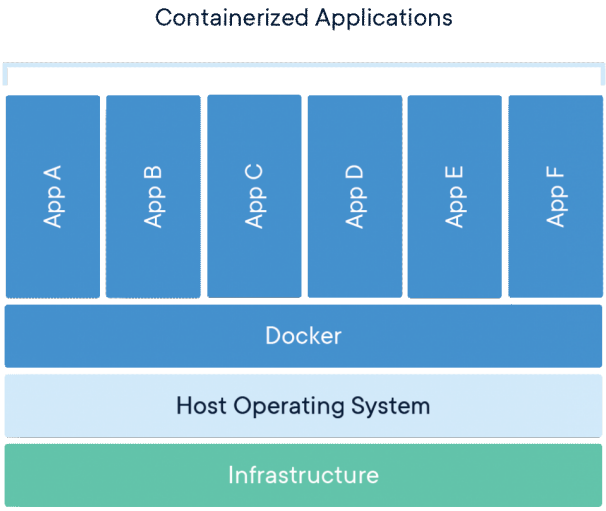


Figure 2.17: Structure of containerized applications. [30]

2.4.8 Kubernetes

In a production environment there is a need to manage containers running applications and ensure there is no downtime, and Kubernetes (K8s)[31] provides a framework to run distributed systems in a resilient manner, by taking care of the scaling and failover, and providing deployment patterns.

Some of the features provided by Kubernetes are:

- *Service discovery and load balancing*: exposing a container via a DNS name or IP address, and if the traffic to the container is high, it can create a new instance of the same container image and perform load balancing by distributing network traffic between them. This feature is widely used in the context of the project.

- *Storage orchestration*: allows to automatically mount a chosen storage system, being it local, cloud hosted, or some other place.

- *Automated rollouts and rollbacks*: gives the possibility of automatically deploy and remove containers, and move resources between them.

- *Automatic bin packing*: once decided the amount of CPU and memory (RAM) needed by a container, it automatically place containers in node clusters to make the best use of the resources.

- *Self-healing*: automatically restart failed containers, replaces them, and kills unresponsive ones.

- *Secret and configuration management*: allows to store sensitive information that can be deployed or updated without the need to rebuild the container images.

Rancher

The tool used for the orchestration of Kubernetes clusters is Rancher [32], an open source platform that makes it easy to create, manage, and monitor them.

Definition. *Rancher is a software product to manage Kubernetes clusters. This includes not only managing existing clusters, but building new clusters as well. [33]*

Chapter 3

The Data Entry Tool

The Data Entry Tool is part of an enterprise customer's digital transformation, consisting of a robust and integrated solution that enables the coexistence and interoperability of on-premise, private and public cloud data resources. The required platform guarantees access to and manipulation of data from multiple sources offering on-demand requests in near real-time. It will be the tool's job to validate and redirect the data to the corresponding database, as well as grant a user the access only to a subset of tables, according to the given permissions.

3.1 Requirements

The requirement for the tool were the following:

- There has to be a distinction between a normal User and an Administrator of the Data Entry Tool;
- Tables can insist on several databases and the supported engine are PostgreSQL, Oracle, and MSSQL Server;
- Tables can be organized in groups in such a way as to create a 2-level depth hierarchy;
- Each table could be assigned an action in JSON format that would be performed on-demand;
- A subset of permissions is assigned to each table, spanning from simple visualization to direct alteration of values;
- Permissions to users are provided by creating associations between groups of users and a table or groups of tables;
- Each Field in a table is associated with a data type which will help its validation;
- The possibility of importing/exporting data from/to Excel files must be arranged;
- Each operation performed by a User must be tracked.

3.2 Architecture

The overall architecture is composed by three different elements: the underlying database of the tool, the Back-End, and the Front-End, as seen in Figure 3.1. Firstly, the database allows to map an existing database that belongs to the client; this database could be located anywhere,

either in the cloud or on-premise, as long as it is reachable via the AWS subnet. This is where all users of the tool are registered, with the respective permissions on each table, as well as a set of logs regarding all the operation performed by any of them.

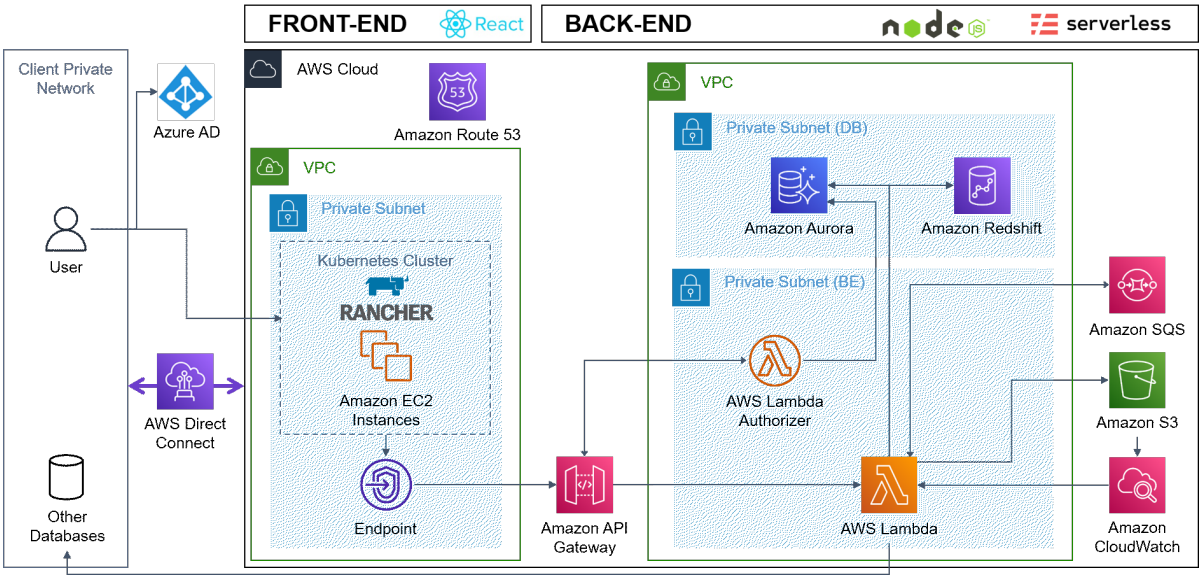


Figure 3.1: High level overview of the Data Entry Tool.

Then, the Back-End contains all the logic regarding database mapping, the inclusion, authentication, and authorization of the users, the granting and revoking of the permissions, external tables manipulation and data validation, and the logging of all operations.

Finally, the Front-End is where users and administrators interact with all registered databases: an administrator insert new users, databases, and tables, and creates associations between users and tables with all the required permissions; a user performs operations on the permitted tables without any knowledge to where they are stored or which underlying database is working with, all with an homogeneous view for all the tables.

As part of my work, I was involved in the design of the whole architecture and the development of the Database, Back-End, Front-End, and CI/CD aspects.

3.2.1 Database

The DBMS chosen to handle the tool-related data is an instance of Amazon Aurora PostgreSQL. For the versioning of the database schema we used Flyway, which, as said before, is a tool that performs database migrations. To make use of it in the CI/CD pipeline, I leveraged the available docker image and created a docker compose file which, by automatically obtaining the connection parameters from the environment variables and mapping the SQL versioning files, performs the migrations of the schema.

In the ER Schema in Figure 3.2, we can see all the elements used to map a database in the tool, as well as handling of the authentications and authorizations to perform operations.

- **Users**: an entity who represents the users of the Data Entry Tool; they can be differentiated in normal user and administrator, where the administrators are a subset of all the users, which means that they can also utilize the tool as normal users do.
- **Audit Log**: all the operations carried out by users are registered here; each log contains on which table it was performed, the type of the operation, the status, the data used and eventual errors, and when it was performed.

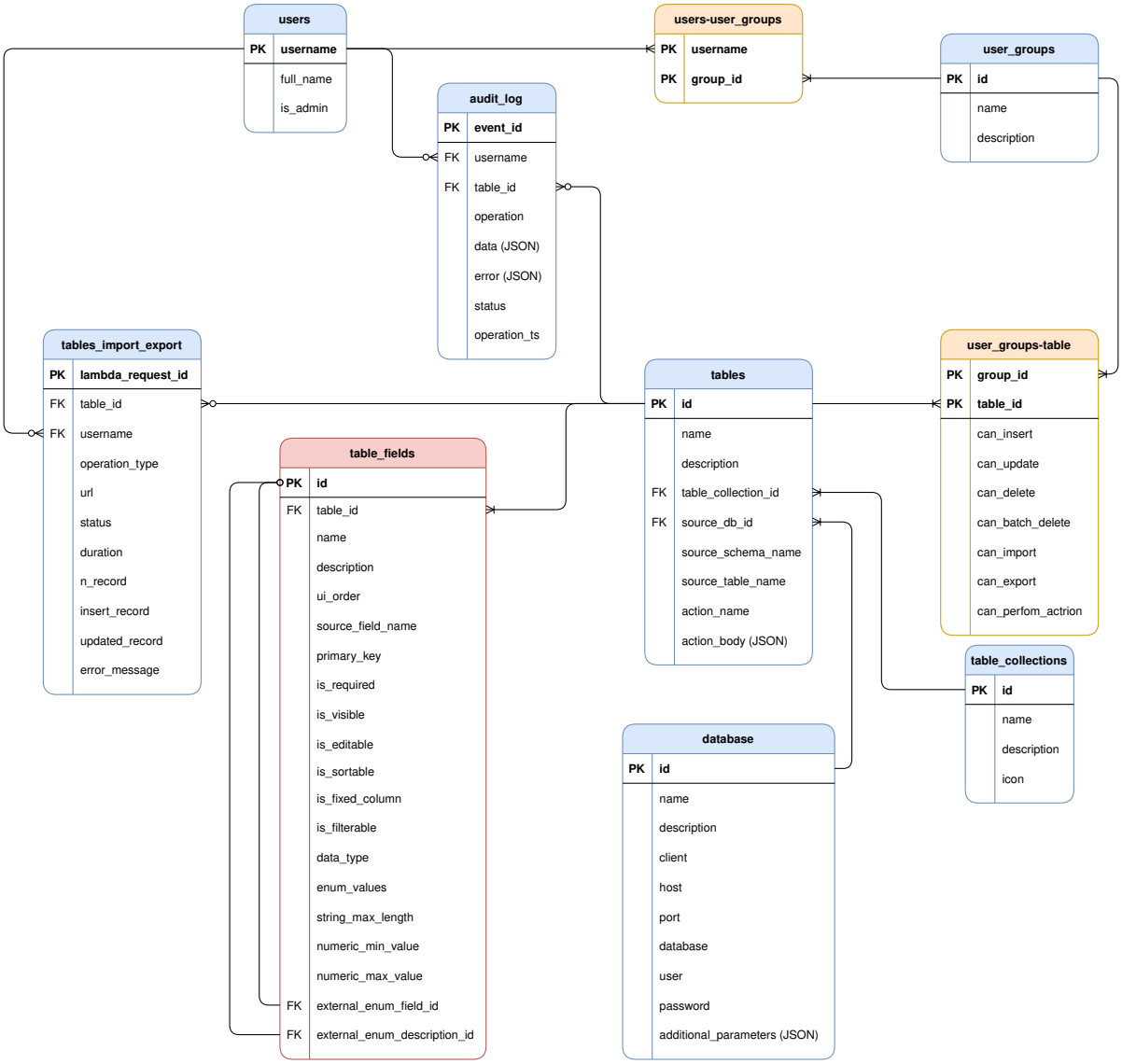


Figure 3.2: Entity-Relationship Schema

- **Databases**: this table contains the connection parameters for each database that the tool has to manage.
- **Tables**: here all the tables of all the registered databases are gathered; each one contains the original name of the schema and table in the respective database. Also, a table may contain the definition of an action that is performed upon request of the user.
- **Table Fields**: this table contains all the column of a table in a given database. Each field maintain the details of a column, from the constraints like being a primary key column to the data type, from a list of possible values to the maximum number of chars that constitute it. There is also the possibility to set features for each field:
  - *Primary Key*: marks a field as Primary Key; multiple fields can be Primary Key;
  - *Is Required*: the user has to provide a value for this field;
  - *Visible*: sets the field as visible by the user that views the table;

  - *Editable*: makes the field editable by the user;
  - *Sortable*: Allows the user to sort the table according to this field;
  - *Filterable*: Allows the user to filter the table according to this field;
  - *Fixed Column*: does not allow the column to move during horizontal scroll of the table;
  - Note: the distinction between the "*Primary Key*" and the "*Is Required*" features is due to the fact that a primary key may not be required; as an example, consider an auto-increment primary key: in this case, the user should not be allowed to manually enter a value as it would cause inconsistency in the database.

- **Table Collections**: collections are a way of aggregating tables and displaying them to the user in an orderly fashion. Only tables that are included in a collection will be shown to users.

- **Tables Import Export**: this table contains all the import and export operations with the relative status.

- **User Groups**: Each user group contains the association between users and tables (N-to-M); these associations tell a user which tables are visible and what kind of manipulation s/he can perform on them. If a User has access to the same table through distinct groups, the sum of the permissions given to each group would be granted. E.g. if group A gives users the ability to delete data in table X, and group B the ability to insert new data in X, then the users that are part of both group A and B will be granted the ability to delete and insert data in X.

To speed up data retrieval from this database, we have created views and indexes.

3.2.2 Back-End

The aforementioned database represents the state of the tool, while the Back-End represents the stateless part. Here, through the Amazon API Gateway, all the functions that manipulate the tool database and all the external ones can be called with a specific REST API. Moreover, many APIs also use something called *Path Parameters* and *Query Parameters*. Path parameters are variable parts of a URL path. They are used to point to a specific resource within a collection, such as a user group identified by an ID. A URL can have several path parameters, each indicated by curly brackets. Query parameters on the other hand are key-value pairs that are an optional part of the path and are primarily used to perform the sorting, pagination and filtering of the tables from the Back-End side.

As we can see in Figure 3.3, before performing any operations on the tool, any user who is logging in, whether an administrator or a user, will be authenticated and authorized by the *Lambda Authorizer* to perform specific operations on the tool. Some Lambdas are also used to perform scheduled cleaning of tables used as support for certain operations.

Regarding the APIs, there are 3 main paths that can be used:

- `/me`: from this API, any user can retrieve information regarding his/her own profile.

- `/admin`: with this API an administrator can manage the entire tool, like granting permissions to users, handle the various databases, review the logs, and so on.

- `/user`: via this API an authenticated user can perform operations on the tables it has access to.

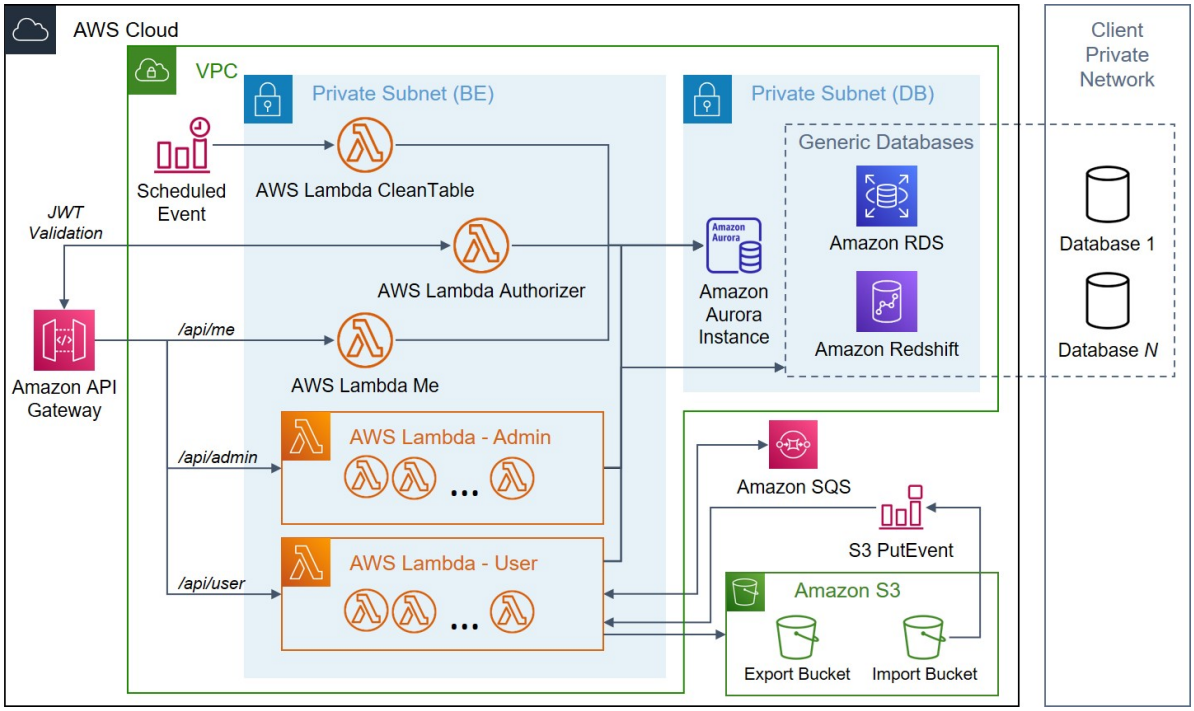


Figure 3.3: High level overview of the Back-End architecture.

All of the APIs listed can be accessed via the HTTP method *GET*, so that when a request comes from the Front-End, they can answer with the current state of the tool for that particular API. Moreover, almost all of the APIs provide access for the methods *POST*, *PUT*, *DELETE*, effectively creating CRUD APIs, to manipulate the state of the tool (e.g. adding a new database, update the records of a table, deleting a table collection, and so on).

Authentication & Authorization

Upon landing on the platform, the user will be redirected to the Microsoft login page to enter the company's credentials. The authentication process is performed via a Json Web Token (JWT) issued by the customer's Azure Active Directory, a cloud-based identity and access management service. The token, encrypted by Microsoft, contains the username of the user being authenticated, the time of issue, and the time of validity. If the token has not expired yet, the username will be searched for in the Aurora instance. If the user exists in the database, by default s/he is authorized the access to the base APIs `/me` and `/user`, and if the user is also an administrator, s/he is authorized the access also to the base API `/admin`; otherwise if the user is not present in the database, the access to all the resources is denied.

A note regarding the access to user API: even if the user has access to this path, if s/he is not allowed to perform any operation on any table, s/he will land on the homepage with an empty dashboard.

Administrator Side

Before we get into the details of the administration section, it is important to understand the data flow that needs to be entered into the tool in order to display it properly from the user side.

From the road-map in Figure 3.4, we can see that the workflow starts from the creation of

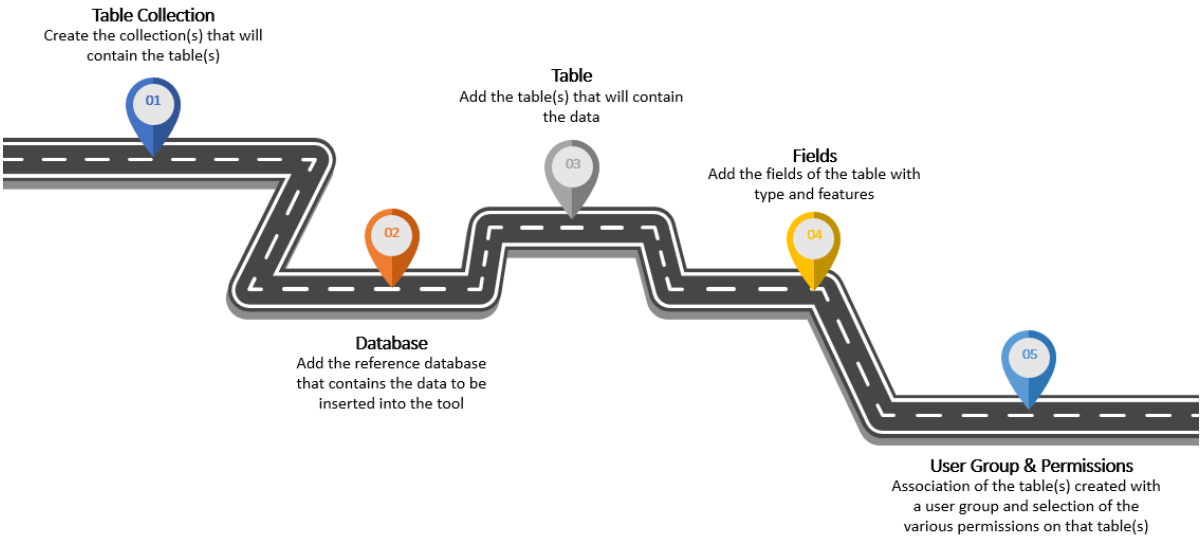


Figure 3.4: Administrators' road-map.

table collections; this is done because it would be a good practice to have a collection in place when the tables are defined. After that, it is time to connect a database to the tool via the provided host, port, username and the password, which will be encrypted before storing in the Aurora instance. After a database connection test, to make sure that the provided connection parameters are correct, all necessary tables can be added, and whenever a new table is added, all fields that belong to the table are automatically added as well.

When the tables are in place, it is time to add users to the tool and assign them to user groups so that they have the appropriate permissions to operate on them.

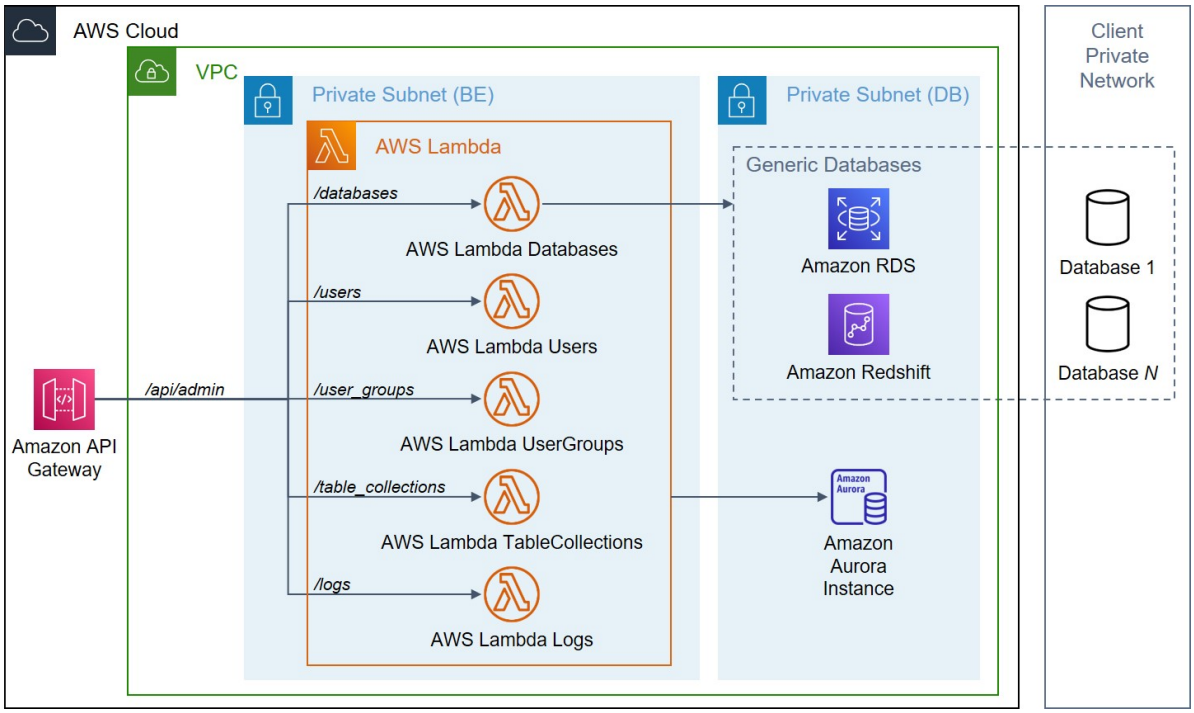


Figure 3.5: Administrators' side Back-End architecture.

In Figure 3.5 we can see the different functionalities for an administrator:

- `/databases`: from this API an administrator can connect new databases to the tool, register new tables, and for each table register all fields; moreover, not all tables need to be manipulated by users; some of them can be added only to be referenced in other tables, as with foreign keys, to prevent integrity constraints problems.
- `/users`: this API is used to manage all users of the tool; keep in mind that the users of the tool are a subset of the users of the company.
- `/user_groups`: from this API you can manage user groups, where all users and tables are associated with each other with different permission levels.
- `/table_collections`: all table collections are maintained with this API. These collections are useful in the Front-End to display all tables from a drop-down menu, so that they are easily accessible by users.
- `/logs`: this API gives access to the logs of all operations performed on the registered tables by all users of the tool.

The full list of the APIs for the Administrators can be seen in Table 3.1.

Table 3.1: List of APIs for the Administrator side.

API List	Description
`/users`	Provide CRUD operations on the users of the tool
`/user_groups`	Manages all the user groups
`/user_groups/{group_id}`	Manage a given user group
`/user_groups/{group_id}/tables`	View and manage the tables in a group
`/user_groups/{group_id}/users`	View and manage the users in a group
`/table_collections`	Manage all the collections
`/table_collections/{table_collection_id}`	View and manage the tables in a specific collection
`/tables`	Returns all the tables
`/databases`	To manage all the databases saved in the tool
`/databases/{database_id}/tables`	To manage the tables in a database
`/databases/{database_id}/check_connection`	Check if the connection to the database is working properly
`/databases/{database_id}/tables/{table_id}/fields`	To manage the fields in a table
`/databases/{database_id}/tables/{table_id}/fields/order`	To change the order of the columns displayed to the user
`/logs`	To view the logs of the operations in the tool

User Side

The user side of the Back-End contains all the logic of the possible manipulations they can perform on the actual tables distributed in the different databases. Note that no user can access tables for which s/he does not have permissions.

In contrast to the administrator side, where the structure of the tool is static, meaning that each administrator will have the same graphical interface available, each user will have the interface adapted to their role in the tool. To be able to do it, it is necessary to provide the *metadata* of the accessible tables to the Front-End, and checking on the Back-End side that each operations on the actual data is allowed. From a security perspective, this allows to thwart Man-in-the-Middle tampering.

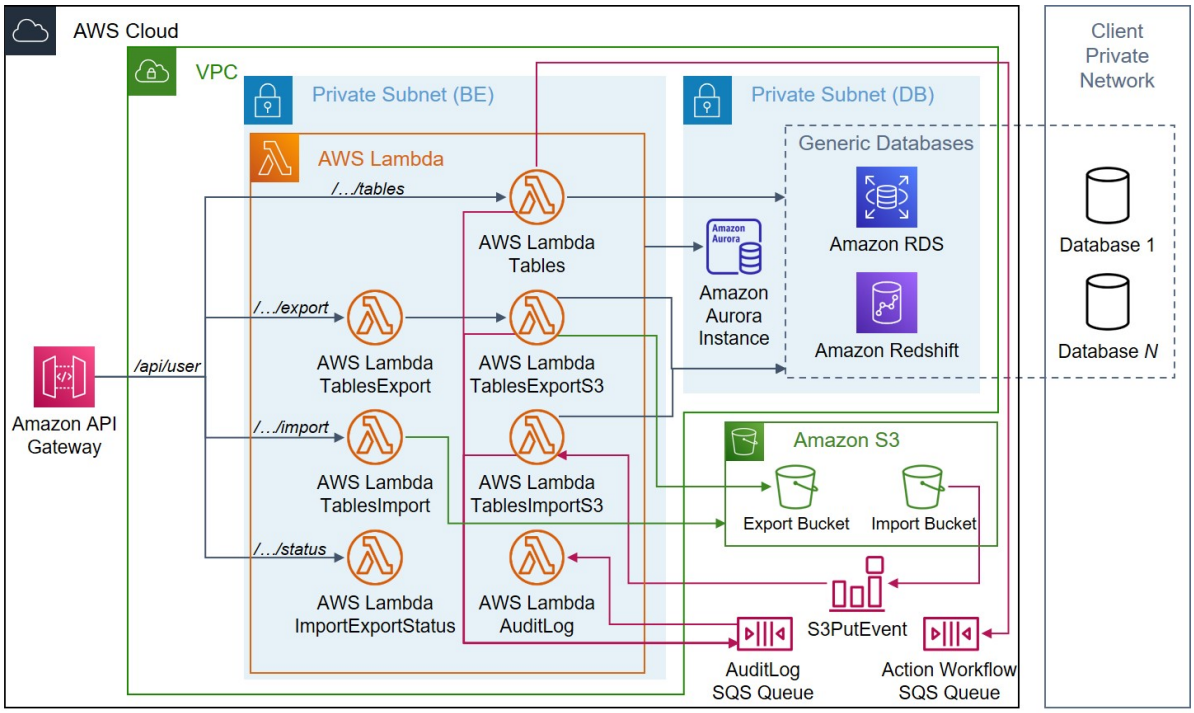


Figure 3.6: Users' side Back-End architecture.

Figure 3.6 represents the architecture behind the main users of the tool. After the successful execution of an allowed operation, that operation must be logged, and to do so, a message is sent via the SQS queue *AuditLog* to another Lambda that will log it. The queue is used to decouple the operation execution to its logging so that the Lambda invoked by the user can return a response as quickly as possible to maintain some semblance of a real time feeling. The same reasoning is used also in the case of a table export, where no allowed operation is restricted during the export process, but not in case of import because there would be conflicts with any operations performed by the user, and for this reason no operation is allowed until the import process is completed.

Each API stems from the same base root, so each operation can refer the same table:

`/table_collections/{collection_id}/tables/{table_id}`

The most important endpoints accessible by the user are:

- `/tables`: this API allows users to see and modify the content of all the tables they have permissions to.

- `/export`: if the user is allowed to, with this API they can export the table in Excel format on their local machine.
- `/import`: depending on the permissions given to the user, to update records, insert records or both, this API allows to load an Excel file to modify the table.
- `/status`: with this API it is possible to keep track of the import/export status.

Table 3.2 shows the complete list of APIs for the User Side.

Table 3.2: List of APIs for the User side.

API List	Description
`/table_collections`	Returns all collections with their own tables and user permissions
`/.../metadata`	Returns the metadata for a given table (e.g. table name, permissions, fields, ...)
`/.../records`	Gives access to CRUD operations on data according to permissions
`/.../`	If available for a table, trigger the action execution
`/.../batchDelete`	If allowed, allows to delete multiple row in a table
`/.../export`	If allowed, triggers the table export
`/.../import`	According to permissions, allow the import of an excel in *INSERT*, *UPDATE*, or *UPSERT* mode
`/.../{operation_type}/status`	Get the status of the current user's import and export operations on the table
`/.../{operation_type}/status/{request_id}`	Get the status of the given import/export operation

3.2.3 Front-End

The APIs described in the Back-End section are not directly accessible by a user; they will interact with a Graphical User Interface to perform operations on the Data Entry Tool or the distributed databases. The Front-End section will use those API transparently to the user, while still showing some of them in the URI of the Tool website to make it clear to the user what resource it is trying to access. The complete list of APIs available to the users can be seen in Table 3.3.

Table 3.3: APIs exposed by the Front-End.

API List	Description
`/user`	User's dashboard and main page
`/user/table_collections/:tableCollectionId/tables/:tableId`	Gives access to view and manipulate the given table
`/admin/databases`	To manage the databases
`/admin/databases/:dbId/tables`	To manage the tables in a given database
`/admin/databases/:dbId/tables/:tableID/fields`	to manage the fields in a given table
`/admin/users`	To manage the users of the Tool
`/admin/user-groups`	To create the association between users and tables and granting the permissions
`/admin/table-collections`	To manage all the table collections
`/admin/logs`	To view logs of operations performed by users

As shown in Figure 3.7, the Front-End application consists of a Docker container that is hosted on an instance of Rancher, which exposes a URL to access the Tool that is accessible only from the corporate intranet.

Web Application Container

The aforementioned container is based on an NGINX [34] image, an open source software which has capabilities as web server and reverse proxy. The web server, listening on port 80, is used to serve the Tool's end users the Front-End of the application, while the reverse proxy capability is used to redirect user requests to the appropriate Back-End server, in this case to the proper Back-End API.

Web Application Instance

To create an instance of the Front-End container image, which is called a *Pod* in Rancher, other than the image itself, it is necessary to provide an entry point to the Tool. This is done via the `Ingress` service: through it, the NGINX web server URL and listening port are bound to the Ingress service URL and listening port. Moreover, it is provided a TLS certificate to ensure security and encryption to the data exchanged between the user and the application.

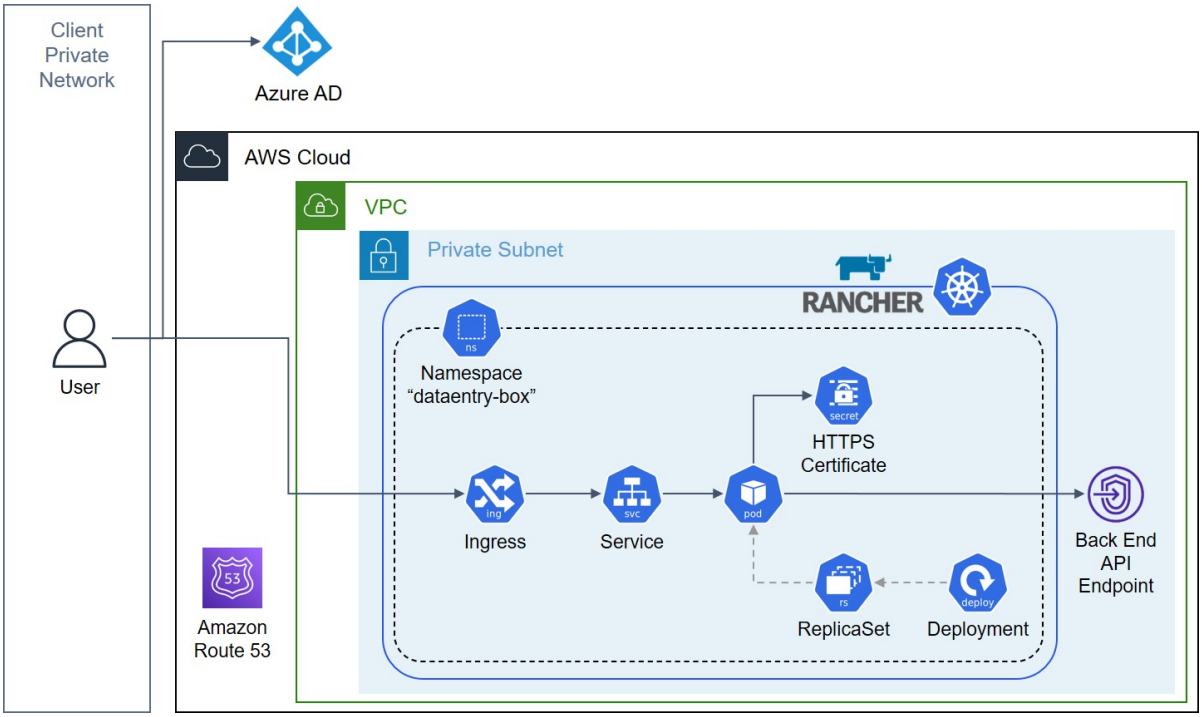


Figure 3.7: Front-End architecture.

Authentication & Authorization

As described before, in order to access the Tool a user has to provide his/her own username to the company's Azure Active Directory which will issue a token that will be verified via the Lambda Authorizer seen in the previous section. If the user is not present in the Aurora database, or if s/he is present but does not have access to any table yet, the pages represented in Figure 3.8 will be shown.

Common Features

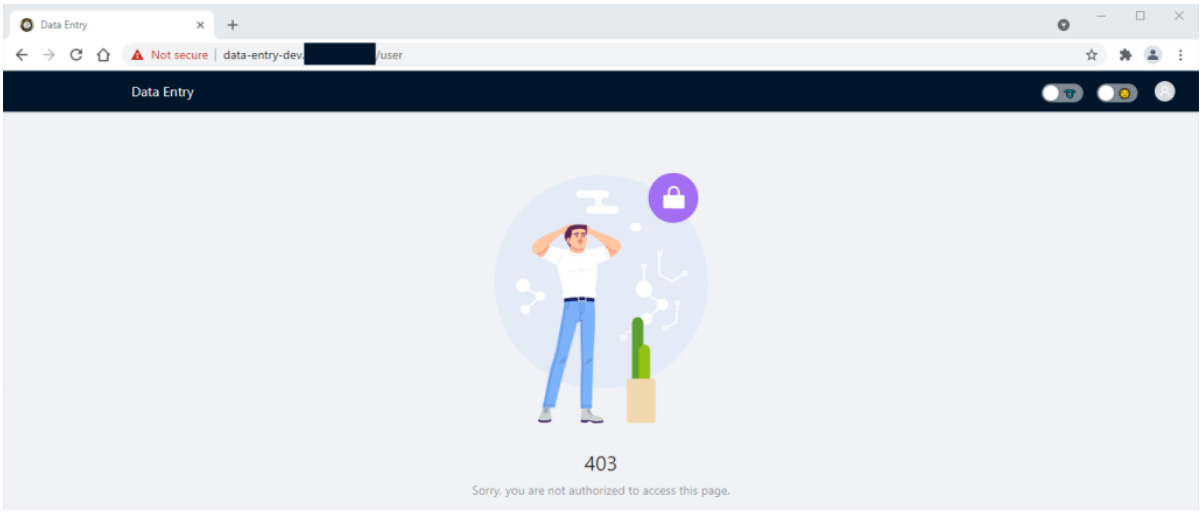
All the tables, shown in the figures that will follow, have some basic operations in common that are performed on the Back-End side to avoid overloading the client and because it is easier to perform while retrieving the data from the different databases.

Pagination

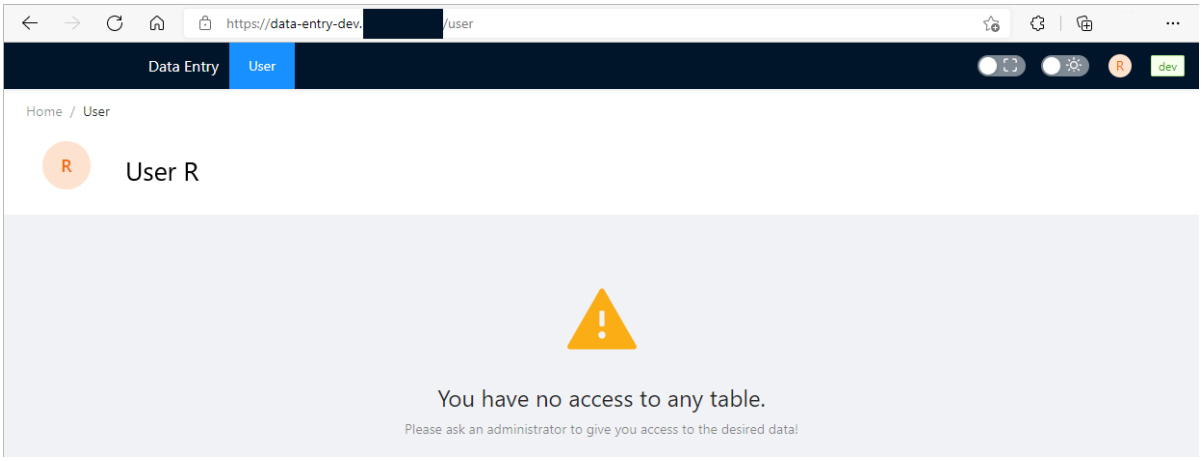
All the tables have the option to be paginated, which means changing the number of rows displayed to the user; this is done by adding the optional query parameters `page` and `pageSize` to the request to the Back-End. These two parameters represent, respectively, the page number and the number of rows to be displayed in the table, and they are used in the process of data retrieval to set the amount of rows to be fetched and with which offset to take them, as shown in the Code Sample 3.1.

```
SELECT ...
FROM table
LIMIT pageSize OFFSET page
```

Code Sample 3.1: SQL representation of the pagination process.



(a) Unauthorized access.



(b) No table available.

Figure 3.8: Restricted access pages.

Filtering

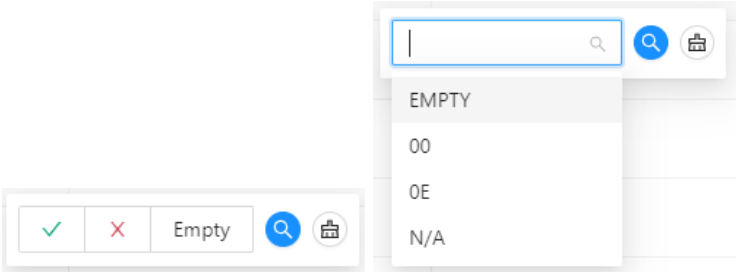
Whenever a magnifying glass icon is displayed next to a column name, it means that you can filter the table based on a value provided for that column. Also, if the columns allow it, it is possible to apply a filter on more than one column at a time, because the filtering process is additive. This is possible thanks to the query parameter `filters` which contains an array of filters to apply to the table; a filter is composed of:

- an operation on how to filter the rows, like *equality* to get the rows with the same value as the provided one, *greater than* to get all the elements which are greater than the provided one. The full list of operations is available in Table 3.4 and the representation of those filters can be seen in Figure 3.9;
- the name of the column to filter for;
- the value for which to filter; this could be a single value, a couple of values, and a list of values.

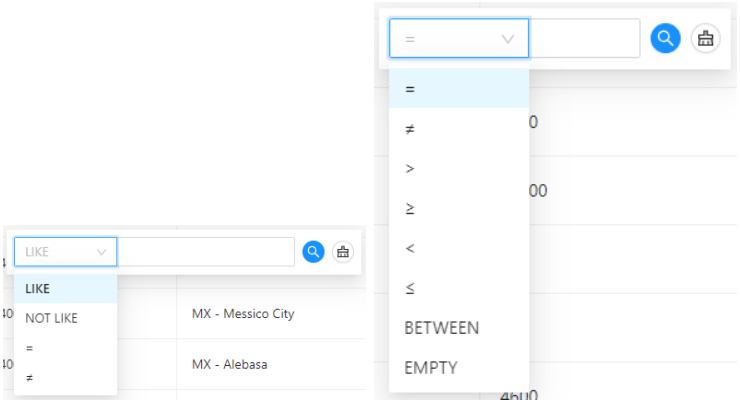
As shown in the Code Sample 3.2, the filtering process is translated into a set of *WHERE* conditions, and since the filtering process is additive, each condition is linked with a *AND* statement.

```
SELECT ...
FROM table
WHERE columnName1 = toCompare1 AND columnName2 > toCompare2 AND ...
```

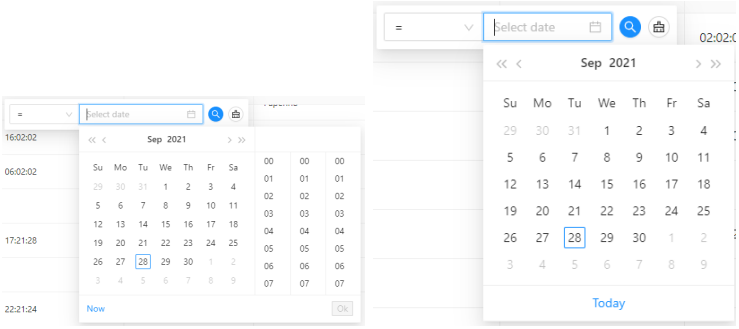
Code Sample 3.2: SQL representation of the filtering process.



(a) Filter by a boolean value or from a list of possible values.



(b) Filter by a given alphanumeric word or a number.



(c) Filter by a date and a given time or just the date.

Figure 3.9: Different filtering options based on column type.

Sorting

Lastly, if the up and down arrows are present next to the column name, it is possible to sort the rows of the table by the values in that column, either in ascending or descending order; also, just like the filtering operation, the sorting operation is additive, meaning that it is possible

Table 3.4: List of the available operations.

Operation	Description
EQ	Takes the values that are equal to the provided one
NOT_EQ	Takes only the values that are **not** equal to the input
GT	Tkes the values that are greater than the input
GTQ	Takes the values that are greater or equal than the input
LT	Takes the values that are less than the input
LTQ	Takes the values that are less or equal than the input
IN	Takes the values that are present in the provided list
NOT_IN	Takes the values that are **not** present in the provided list
LIKE	Takes the values that are similar to the provided one (case sensitive)
NOT_LIKE	Takes the values that are **not** similar to the provided one (case sensitive)
I_LIKE	Takes the values that are similar to the provided one (case insensitive)
BETWEEN	Takes the values that are in between the provided input couple
NOT_BETWEEN	Takes the values that are **not** in between the provided input couple
IS_NULL	Takes the values that are null

to sort a table by multiple columns. Like the previous operations, this one also uses a query parameters, `orderBy`, that contains a list of columns to order by and which order to use for each column. To make it easier for the user to figure out which columns they applied the sort to, if it was applied to more than one, a number appears next to the column name, as shown in Figure 3.10.

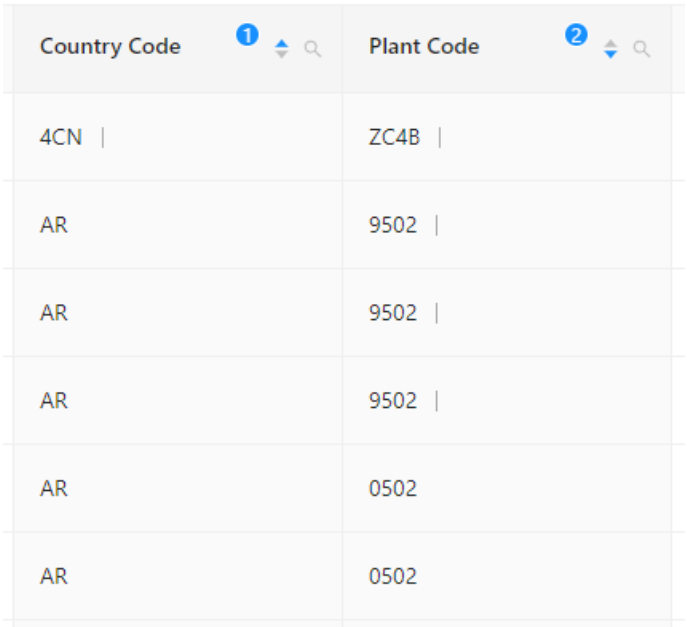


Figure 3.10: Sort order for multiple columns.

Actions on records

The previous ones were operations that affected the entire table, while the following ones are actions that interest the single rows of a table. As shown in Figure 3.11, there are 5 possible actions that can be performed on the records:

- *New Record*: this button allows the insertion of a new record in the table by opening a menu on the right of the table to enter the specific values.
- *Edit*: this button opens the same menu as "*New Record*", but filled with the values of the selected record; from here it is possible to change the values according to the necessary update;
- *Delete*: this operation deletes the record. Since it can be a dangerous operation, before deleting a row a confirmation pop up appears asking if you want to proceed. It is also possible to enable a table for the deletion of multiple rows.
- *Duplicate*: duplicates the current record. This feature helps a user not to have to retype all the values for a new record if most of them are the same as for another one;
- *Refresh*: with this button the table can be refreshed at any moment to visualize any new values.

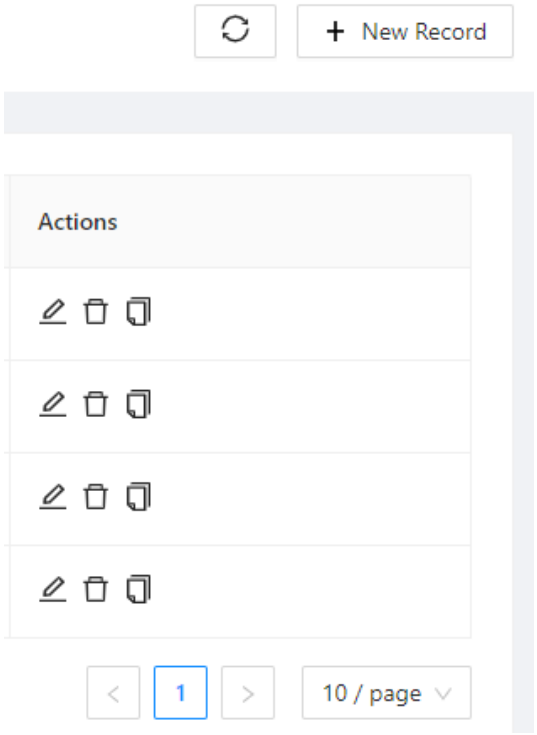


Figure 3.11: Actions allowed on records.

Keep in mind that all of these actions and operations are always enabled for the Administrator side, while for the User side, excluding pagination and refresh, they must be enabled by granting permission to the user group that will access the particular table.

Administrator Side

If the user is present in the database and is recognized to be also an administrator, s/he will be redirected to the first page of the Administrator Side, the *Databases* page, shown in Figure 3.12, where s/he will see all the databases already registered to be managed by the Tool.

Databases

From this page an administrator can add new databases and manage the existing ones. In addition to the basic operations established above, for each database it is also possible to check at any time that the connection to it is working.

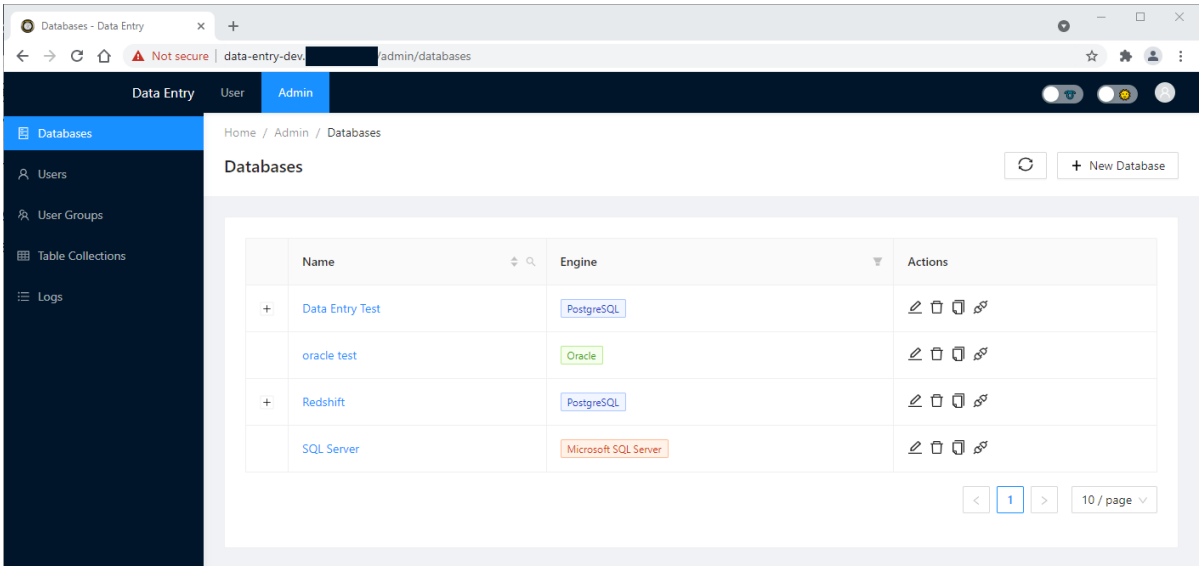


Figure 3.12: Administrator's homepage.

The information required to add, or edit, a database, as shown in the Figure 3.13, is:

- *Database Name*: this is the name that is going to be displayed in the Tool;
- *Engine*: the Database Management System that powers the database; the available options are **PostgresSQL**, **OracleDB**, and **Microsoft SQL Server**;
- *Hostname & Port*: this is the connection string to the database with the relative port;
- *Instance Name*: this is the actual name of the database on its hosting machine;
- *Database User & Password*: these are the credentials used to access the database; all tables that are intended to be used by the Tool must be accessible to this user.

Tables

Once a database has been added, it is time to add to the Tool the tables that will be used by the users. To do this, we need to enter the database we have just created, where we will see the page in Figure 3.14. From here it is possible to add or edit any table to which, as mentioned before, the user of the database used for the connection has access.

Figure 3.15 shows the minimum information needed to add or edit a table:

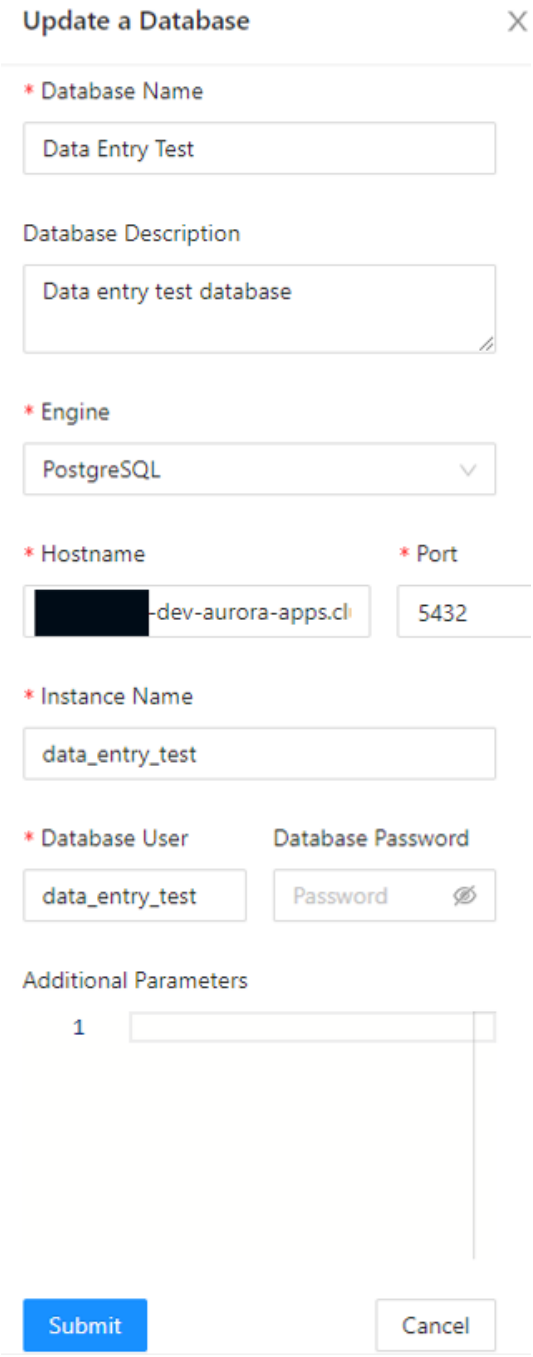


Figure 3.13: Add/edit form for a database.

- *Table Name*: the name that will be displayed in the Tool;
- *Source Schema Name*: the schema associated to the table;
- *Source Table Name*: original name of the table in the database

The original schema and table name are used to retrieve information about the table, in this case, as we will see in the next section, the columns that make it up. As for deleting tables, not all of them can be deleted at will, because they can be used by others as lookup tables from which to retrieve data, much like foreign keys.

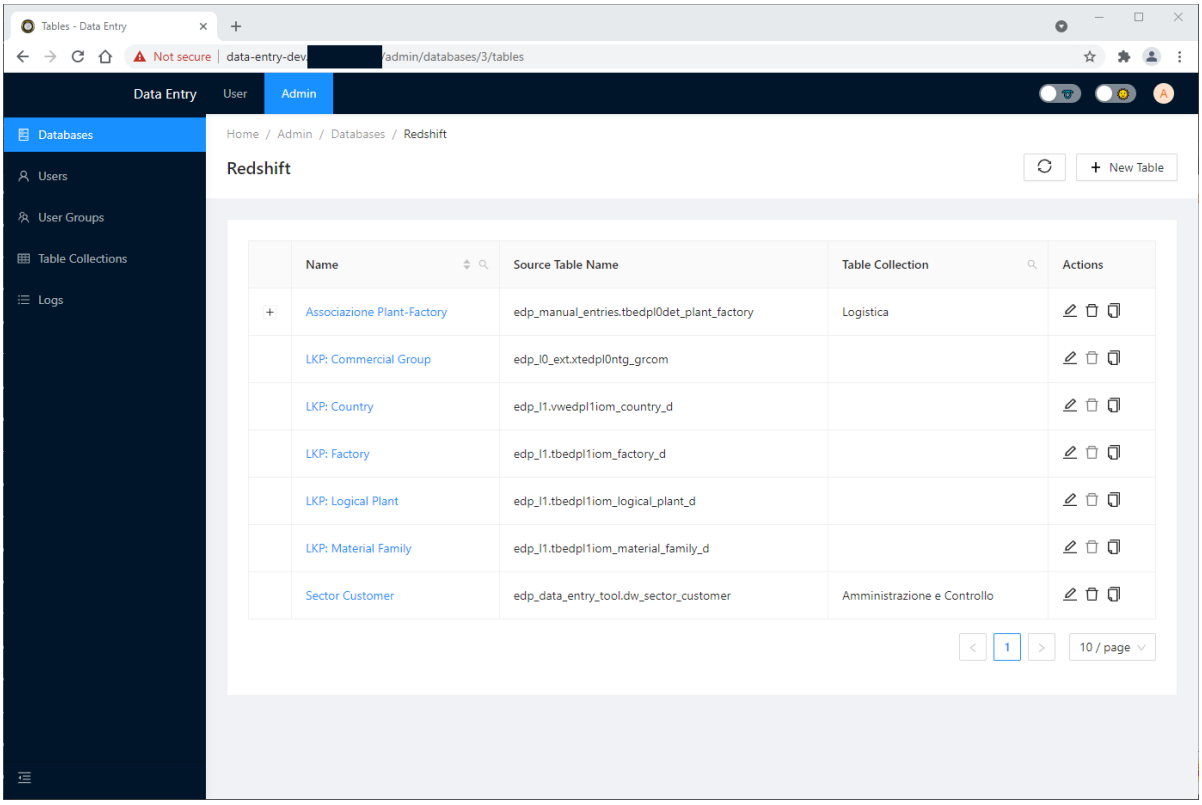


Figure 3.14: Tables management page.

Fields

When a table is added, all fields associated to it are automatically added to the Tool, and by looking at the breadcrumbs at the top of the page, it is possible to see to which table the fields shown in Figure 3.16 belong.

During the field retrieval process, the column datatype is parsed and associated with the field in the Tool; this datatype is important because the filtering and sorting operations and data entry constraints depend on it. For example, a *Numeric* field may have constraints such as a minimum or maximum value, or floating-point precision, and can be ordered by increasing value. This information is retrieved during the table insertion, if available, otherwise it needs to be added by hand in the form shown in Figure 3.17a. Here the bare minimum information needed to edit a filed is:

- *Field Name*: the name that will be displayed in the Tool. It defaults to be the same as the Source Field Name;
- *Source Field Name*: original name of the column in the table;
- *Data Type*: the datatype of the column. Note that different datatype may have more required information, as shown in Figure 3.17b

Users

The next item in the left-hand menu is the *Users* page, displayed in Figure 3.18. From this page an administrator can add a user to the database, which means granting access to the Tool, provided that this user is present in the client's Azure Active Directory. From here you can

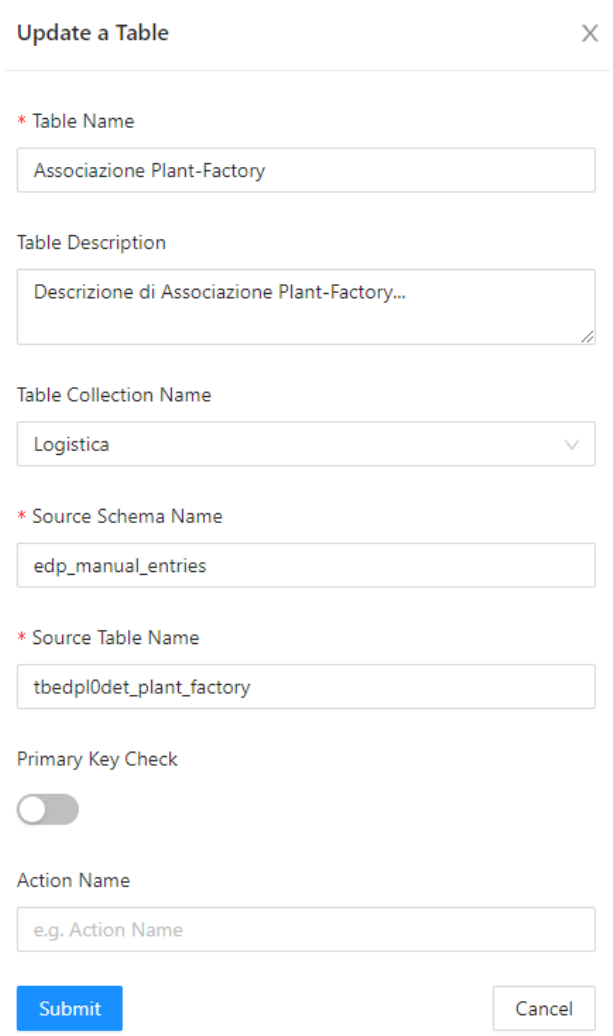


Figure 3.15: Add/edit form for a table.

grant or revoke a user's administrator status, however an administrator cannot revoke his/her own status.

User Groups

Moving forward in the menu there is the *User Groups* page, shown in Figure 3.19. Here all users are associated with tables and have different permissions based on the group. As we can see, there is a counter of how many users and tables each group has. Also, as described above in the Database section, the same user can be associated with different groups, and the same goes for tables; this means that the permission level on the same table can vary between groups, and if the user belongs to different groups where the same table exists, the permission level on that particular table would be the sum of the permissions of each group.

Figure 3.20a shows the form used to make associations in a group, i.e. users and tables, other than the name of the group, while Figure 3.20b shows the permissions that can be given the users accessing a table.

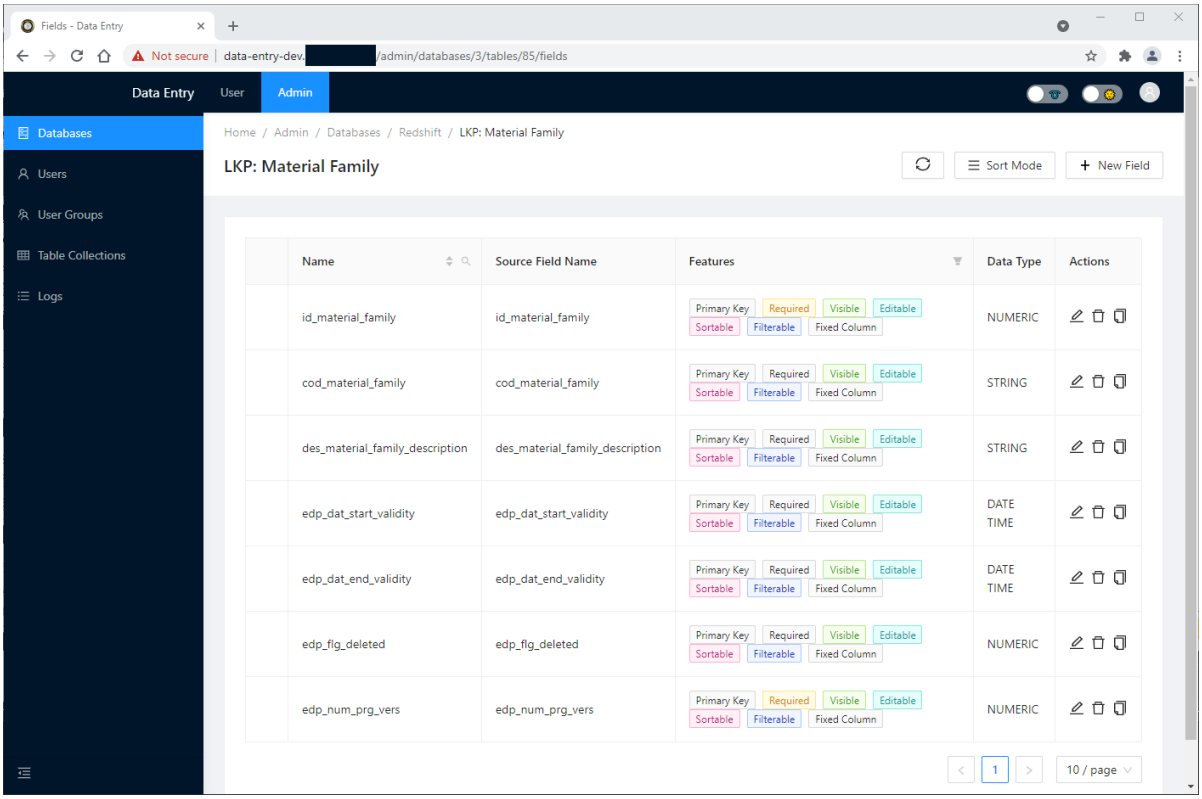


Figure 3.16: Fields management page.

Table Collections

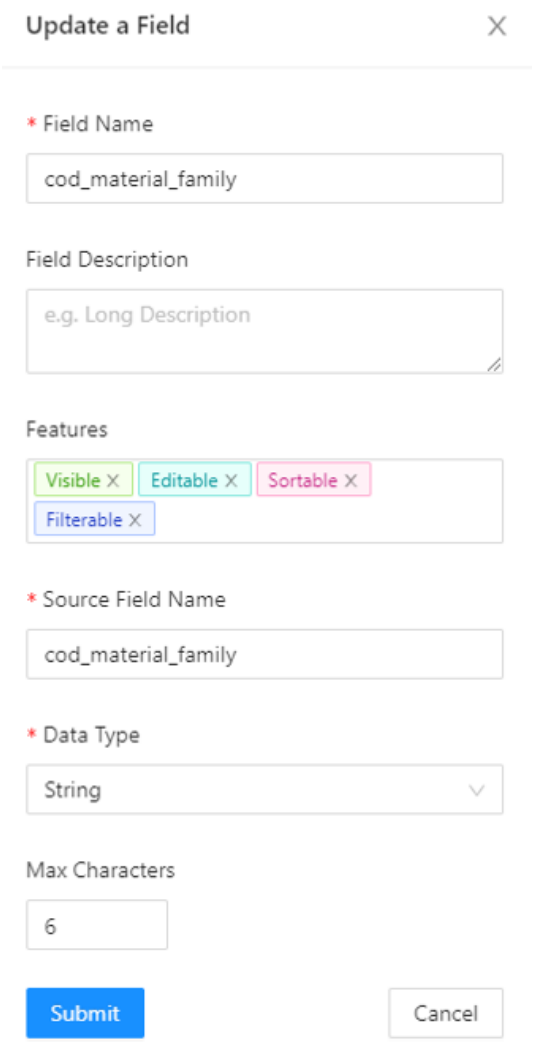
Figure 3.21 shows the next entry in the menu, the *Table Collections* page. The collections managed here are used to group the tables, thus making their use well organized for the user.

In addition to the name, as shown in Figure 3.22, a collection is characterized by an icon; this icon is used to identify the collection when the left menu is minimized to give more room to the table.

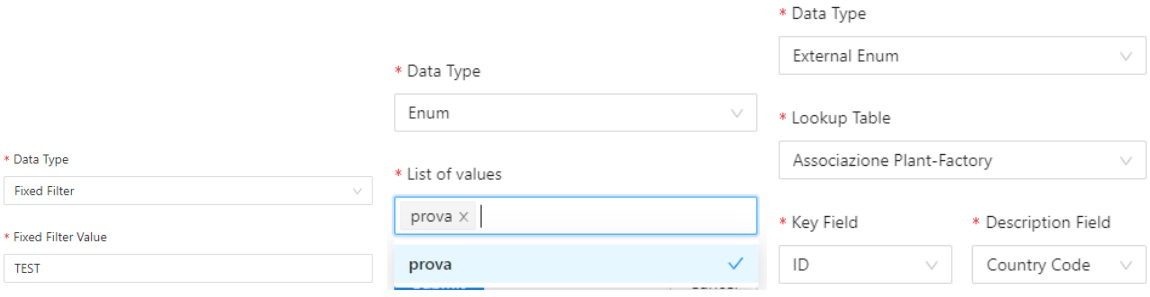
Logs

The last entry in the menu for the Administrator side is the *Logs* page, here represented by Figure 3.23. This page is used to observe all the operations performed by the users of the Tool and the result of each of them.

In case of a failed operation, it is possible to observe the error returned from the Tool, here in Figure 3.24, and the input that caused it.



(a) Edit form for a field.



(b) Different constraints for the datatype.

Figure 3.17: Field management form.

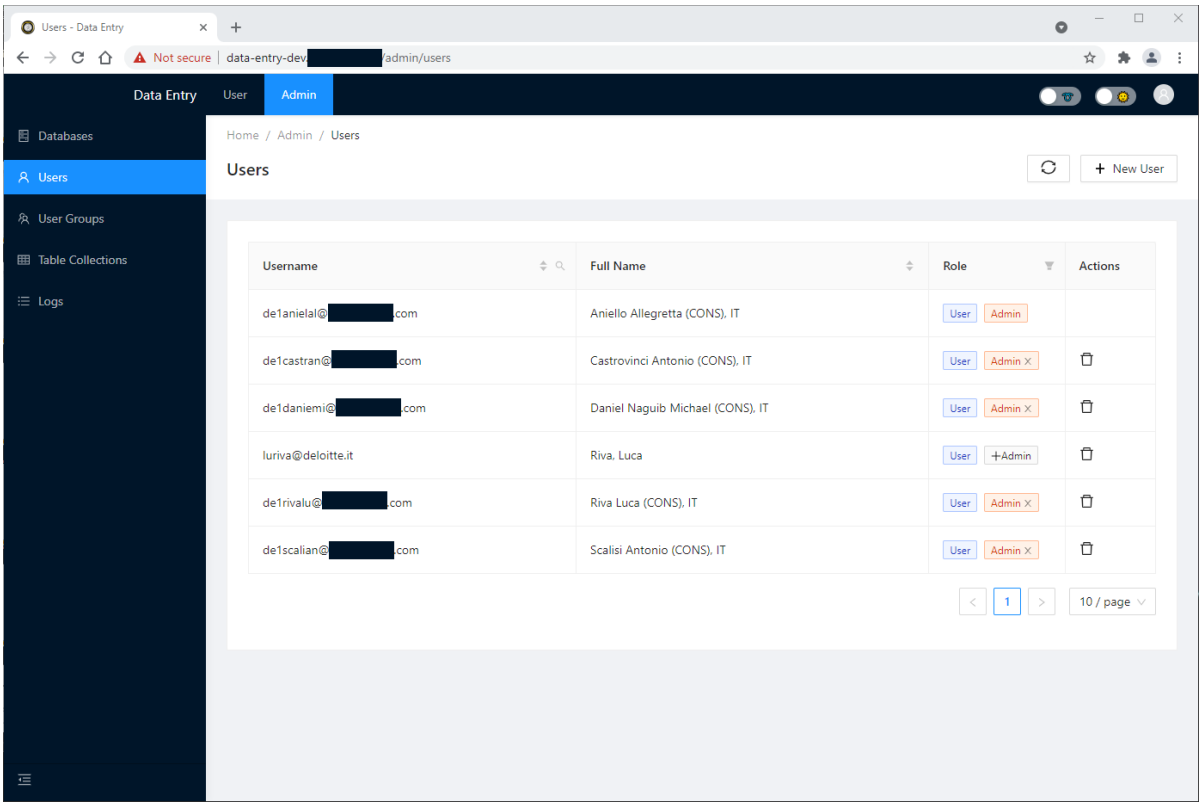


Figure 3.18: Users management page.

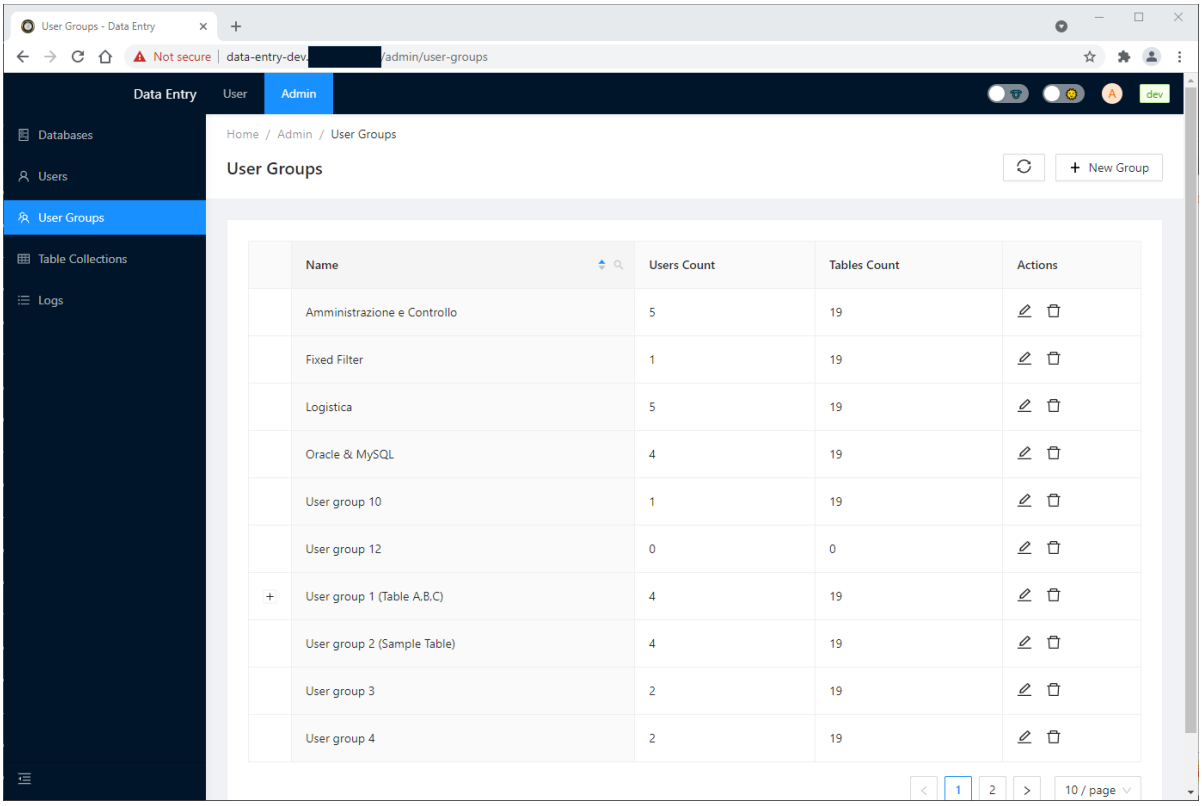
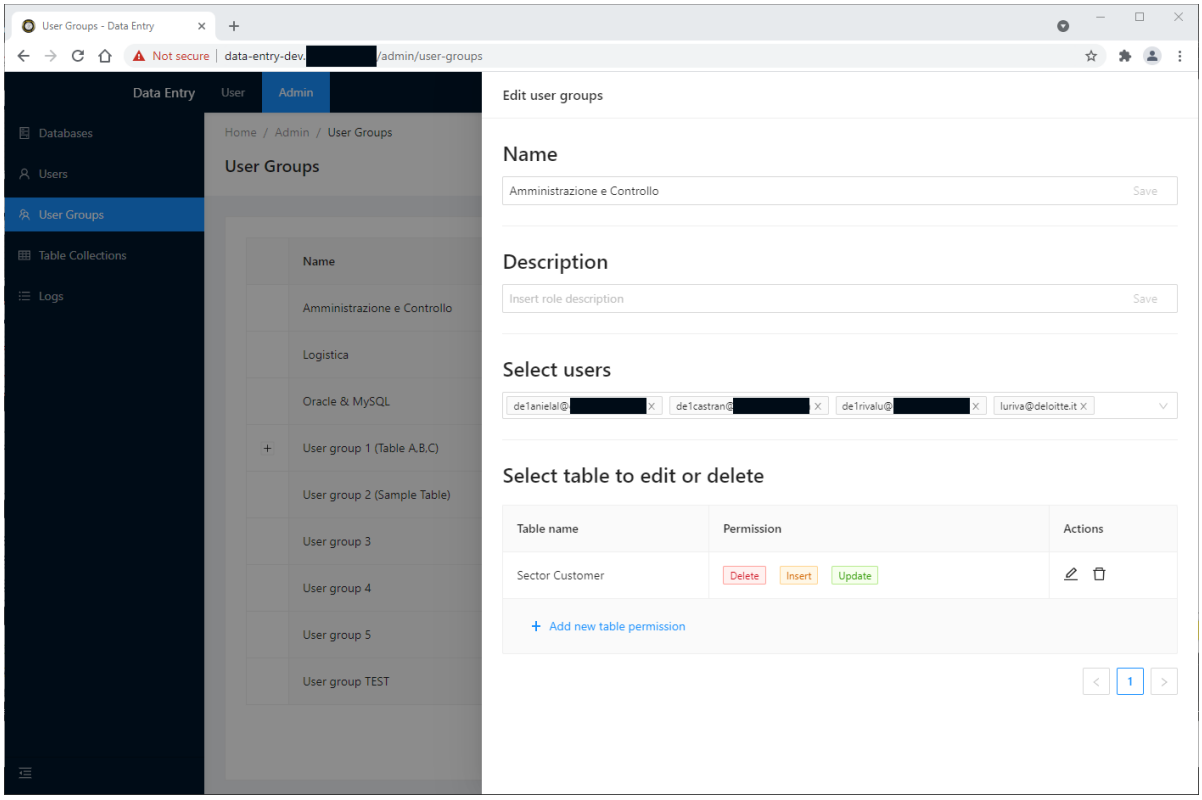
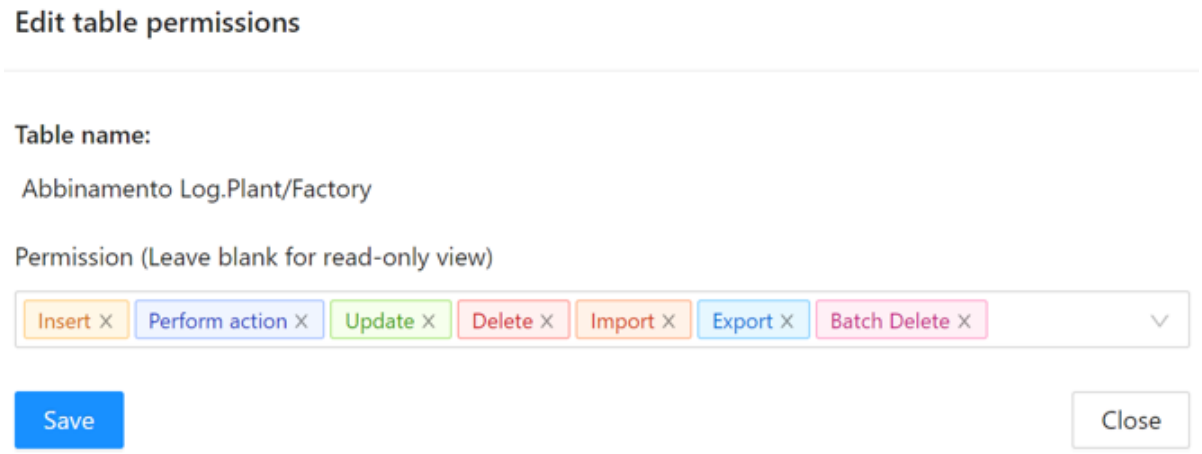


Figure 3.19: User groups management page.



(a) Add/edit form for a user group.



(b) Permissions level form.

Figure 3.20: Group management form.

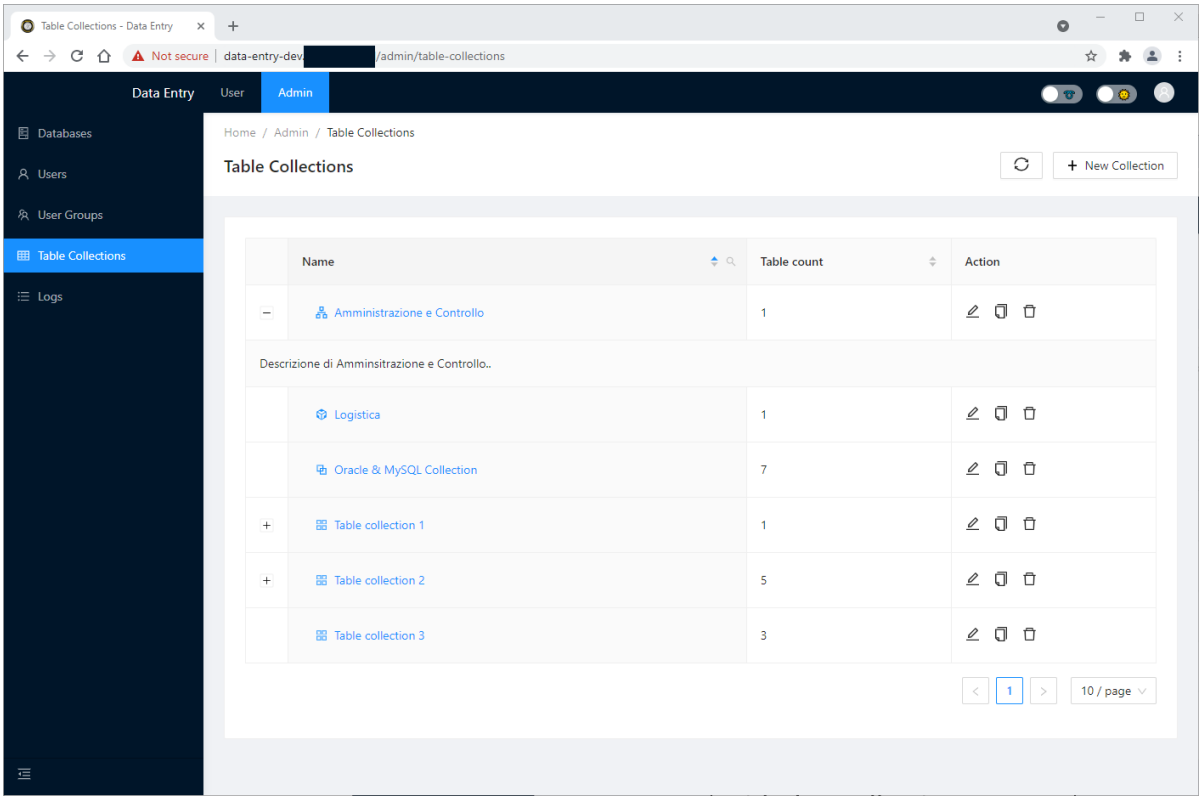


Figure 3.21: Table collections management page.

Figure 3.22: Add/edit form for a collection.

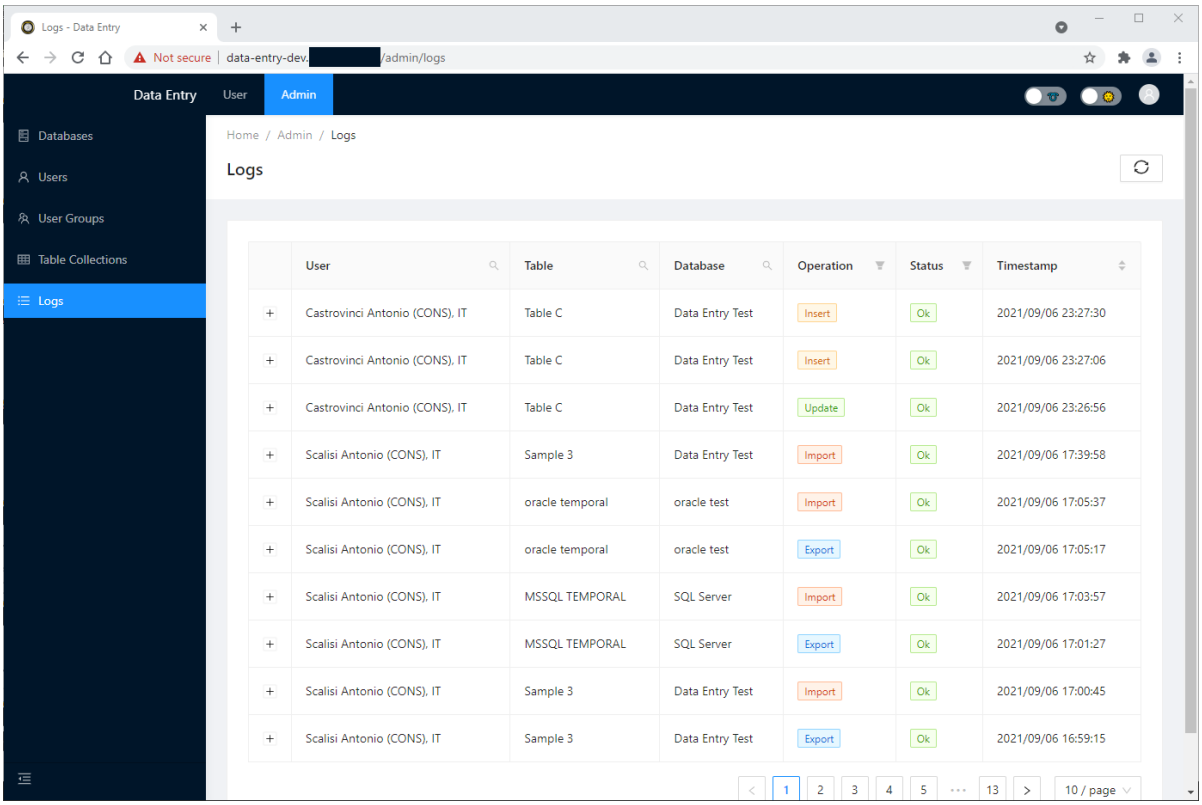


Figure 3.23: Logs page.

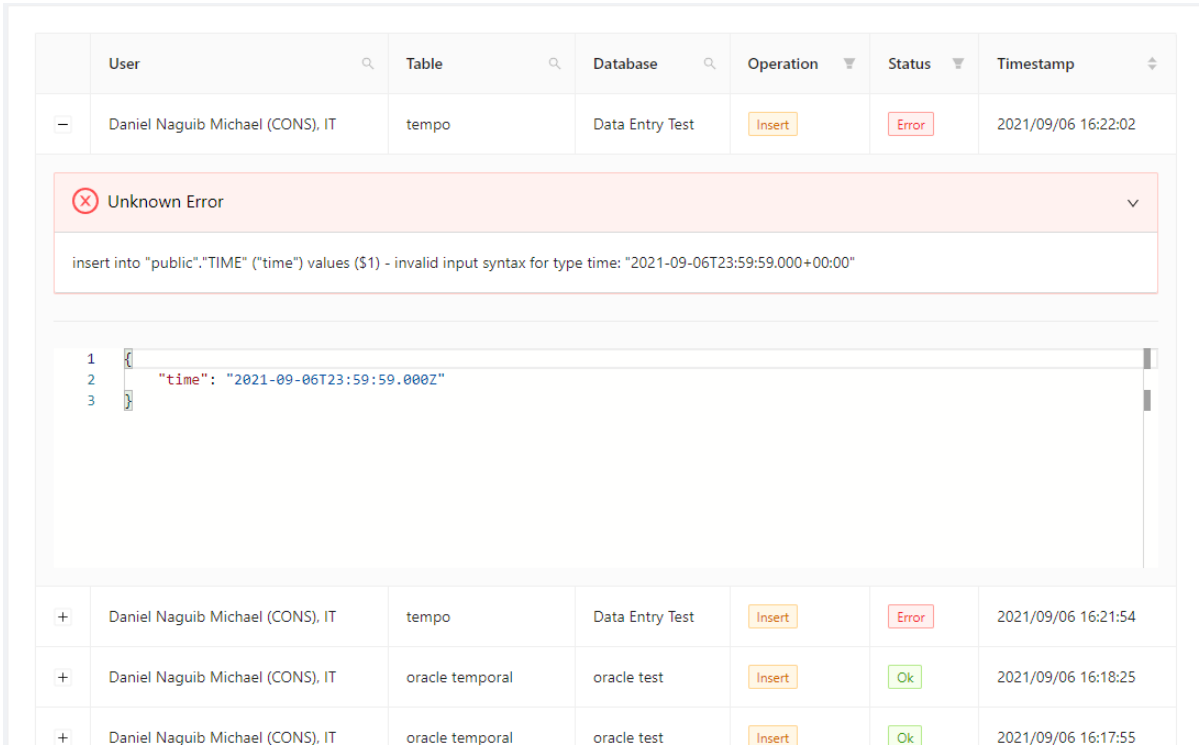


Figure 3.24: Failed operation with the associated error.

User Side

As anticipated in the Back-End section, the User Side of the Front-End is very dynamic due to the fact that each user can access different tables. This is very evident from the homepage that welcomes the users of the Tool shown in Figure 3.25. As we can see, users will see a series of cards representing the different table collections they have access to. Within each card, all accessible tables are listed with an overview of the permissions enabled for each. Note also that the same collections are visible and easily accessible in the menu on the left.

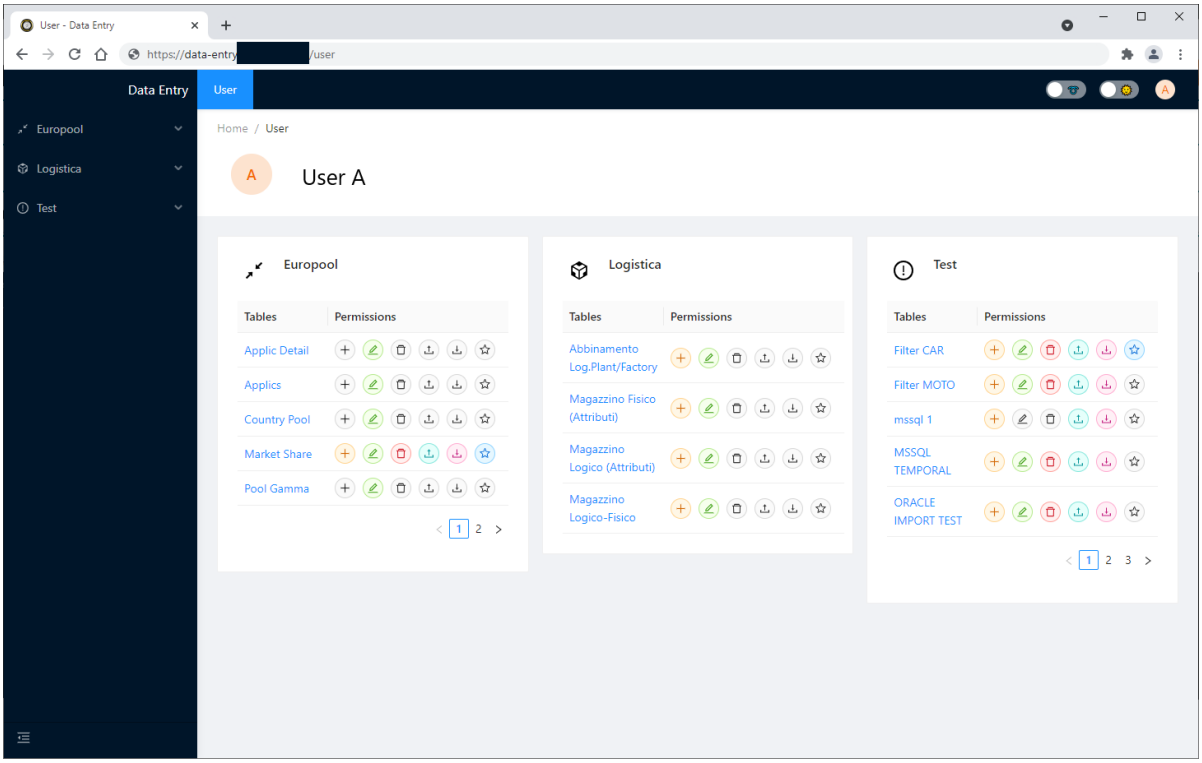
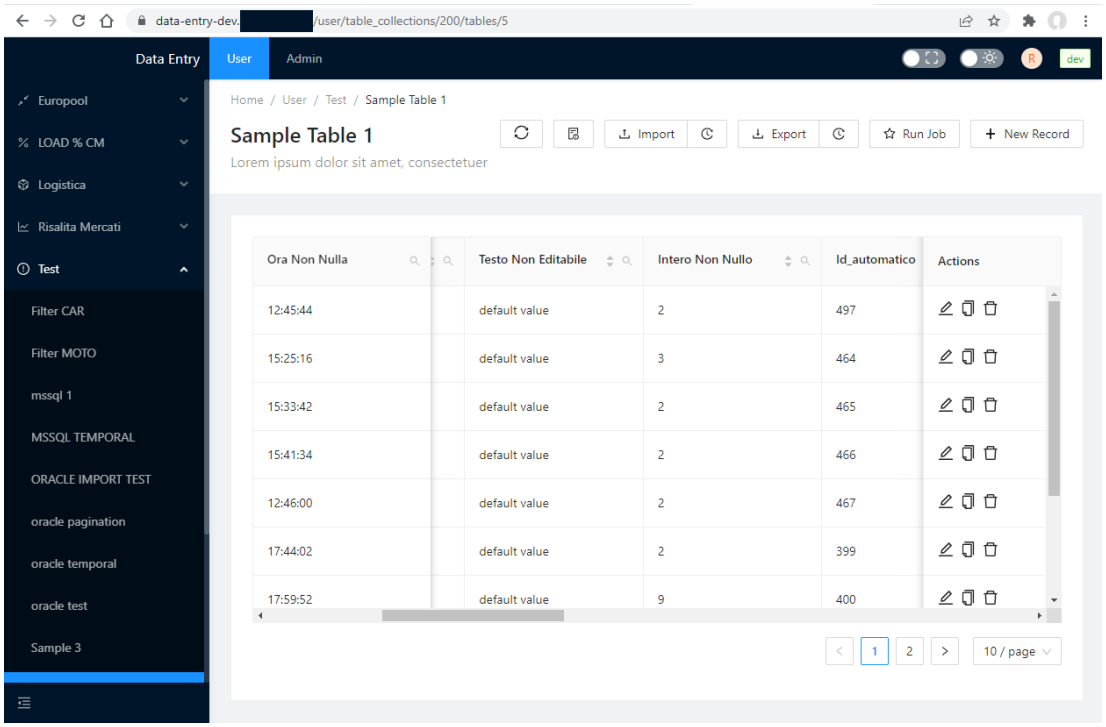


Figure 3.25: Users' homepage.

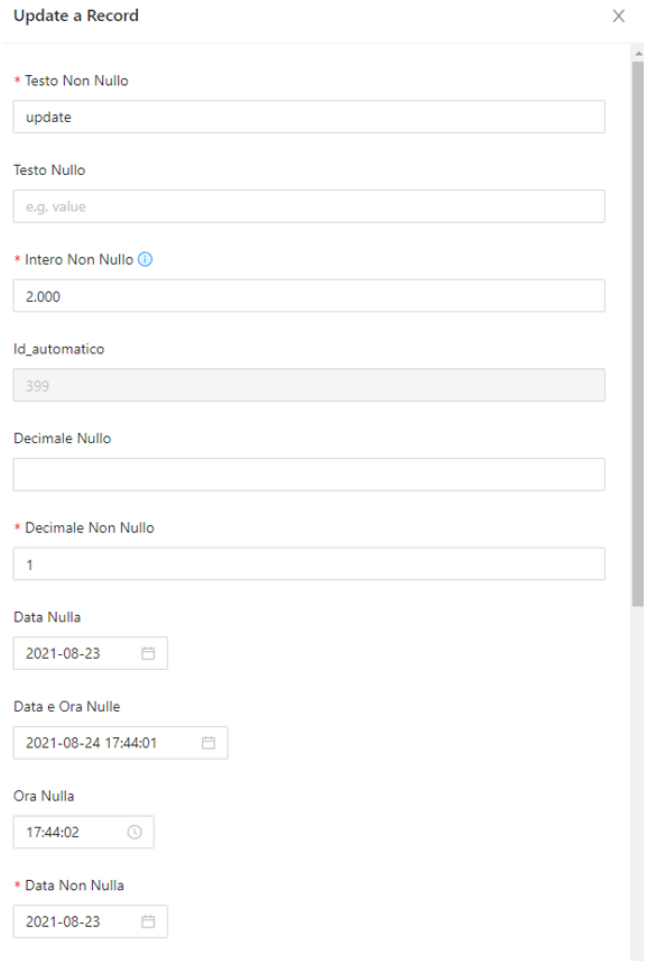
When the chosen table is opened, it needs to be built from scratch; for this purpose, all metadata of the table is retrieved from the Back-End and parsed to build it. This metadata contains:

- the basic information of the table: the table name, the description, the collection it belongs to, and the executable action, if available;
- the permissions enabled for the user of the table, since the presence of action icons depends on them;
- all the visible fields of the table; each field contains the information needed to display the column properly, like the datatype or if it is possible to sort or filter by it, and the information to create the form for the insertion or update.

The result of the processing of these metadata can be seen in Figure 3.26a. In it we can see the actions enabled for the user in the upper part of the table and in the rightmost column, a fixed column on the left side of the table, with which one can only filter, and some columns with different datatype, with which one can filter and sort. Instead, in Figure 3.26b we can see the form generated for inserting or updating a record; here are the fields available in display mode and those available for editing, each with their own datatype constraints.



(a) Example of a generated table.



(b) Example of a generated form.

Figure 3.26: Result of the metadata processing.

3.3 Deployment

As mentioned earlier, the deployment of the tool is performed via GitLab's CI/CD tool, using a deployment pipeline for each branch. The pipeline execution depends on *runners* provided by the company; a runner is an agent that runs the CI/CD jobs, where a job is a definition of what to do. Because the number of runners is limited, and knowing that the Data Entry Tool is not the only project in development for the client, we ran into difficulties because sometimes, in order to test a minor fix, we had to wait for a runner to become available, thus increasing the wait time between each deployment. To this end I created a convention on the branch names and the pipeline executions.

Table 3.5: Pipeline execution mode.

Branch name	Environment	Automatic	Manual	Description
dev-*	dev		X	Branches for feature development
dev	dev	X		Main development branch
master	prod		X	Production release branch

Table 3.5 represents the different execution mode of the pipelines depending on the branch:

- the only branch that would be automatically deployed through the pipeline was the `dev` branch, because it is the collection point for all completed features, allowing integration testing between new and existing features;
- if the branch name starts with `dev-`, the pipeline for that branch must be manually triggered, because not every update on those branches implies a completed feature and we do not want to monopolize the available runners;
- the pipeline for the `master` branch has to be manually triggered, because its execution represents a release of a completed and tested set of features and we wanted full control on when the deploy started.

3.3.1 Pipeline definition

The pipeline used in each branch is divided into 4 phases, as shown in the figure 3.27, and each phase represents the deployment of a section of the Data Entry Tool, and since each phase is independent of the other, it has been made so that each can be deployed independently by monitoring changes in their specific directory.

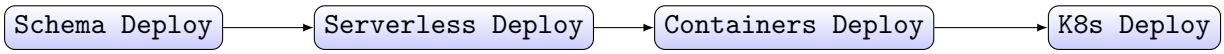


Figure 3.27: Deployment pipeline steps.

1. **Schema Deployment**: in this stage the schema of the database on which the tool work is updated.
2. **Serverless Deploy**: in this stage all the Back-End, meaning all the Lambdas, SQS queue, APIGateway, the encription keys, and sensitive keys, are deployed or updated via the Serverless Framework;
3. **Containers Deploy**: this stage manages the creation and upload of a Docker container for the Front-End application;

4. **K8s Deploy**: in this last stage the container created in the previous stage is taken and deployed via Rancher.

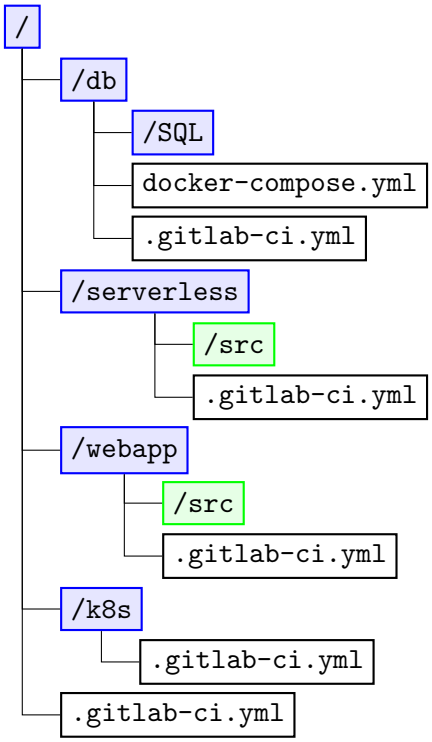


Figure 3.28: Directory structure of the project.

From Figure 3.28 we can see an excerpt of the directory tree with the division into sections of the Data Entry Tool; each contains a file `.gitlab-ci.yml` including the definitions of both all the actions to be performed at the time of deployment, and the files to look at to start the specific stage of the pipeline. The **Containers Deploy** and **K8s Deploy** stages are tightly bound together because whenever there is an update on the Graphical User Interface, not only the container has to be recreated, but it has to be deployed via the orchestration system for it to be available for use.

3.3.2 Schema Deploy

The Aurora Instance at the heart of the project must exist before this phase of the pipeline is executed because, as shown in the Listing 3.3, the Flyway command `migrate` needs a connection string to the database and the user and password used to perform operations on the specified schema.

```
version: '3.9'
services:
    flyway:
        image: public.ecr.aws/XXXXXXXX/datamanagement/gitlab_ci/
            flyway/flyway:7.10-alpine
        command: -url=jdbc:postgresql://${DB_HOST_URL}:${DB_PORT}/
            data_entry
             -schemas=data_entry
             -user=${DB_USER}
```

```
    -password=${DB_PSW}
    migrate
volumes:
  - ./SQL:/flyway/sql
```

Code Sample 3.3: Docker-compose.yml used in the schema deployment.

The `migrate` command is an idempotent operation that scans the file-system for available migrations, in this case in the directory `/flyway/sql` where we mapped the content of the `/SQL` directory in the docker-compose file. After the scan, it will compare them to the migration previously applied to the database, and as shown in Figure 3.29, if any difference is found, it will migrate the database to close the gap.

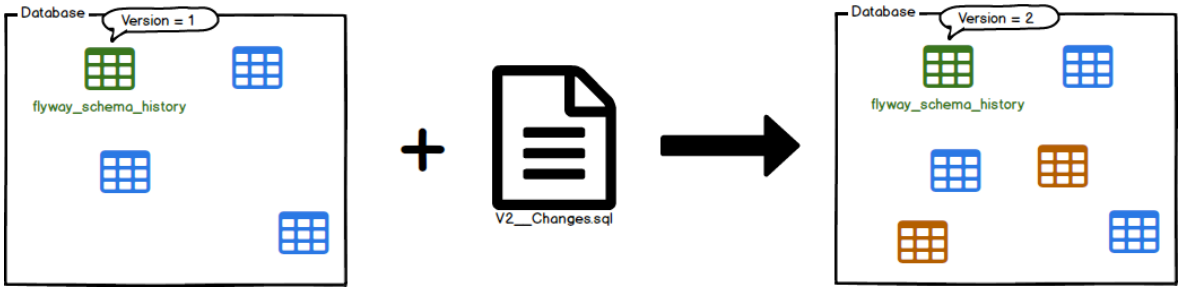


Figure 3.29: Migration process.

3.3.3 Serverless Deploy

In this phase, the Back-End infrastructure is deployed using the Serverless Framework via AWS CloudFormation. For this purpose I created a custom docker image based on Alpine Linux, which was chosen because it has a small size. In the docker image I added Node v14, because it is the same version used in the Lambda's environment and the most recent version at the time of development, TypeScript, because it is the language chosen for development and is used to compile source code into JavaScript when the pipeline is running, and Serverless to deploy directly from the pipeline.

When this phase of the pipeline begins, the operations performed are:

1. install all the dependencies;
2. compile the TypeScript source code into JavaScript to be deployed in the Lambdas;
3. copy of all the dependencies for the Lambda functions;
4. deploy of the infrastructure.

3.3.4 Containers Deploy

For this phase, just like the Serverless Deploy phase, I created a custom Docker image, this time, in addition to Node v14, with the AWS Command Line Interface. Here the TypeScript source files are compiled into JavaScript via the AntDesign Pro specific build operation. Then the compiled files are copied into the Front-End container image, alongside the configuration files for NGINX, and uploaded to the client ECR instance with the help of the AWS CLI.

3.3.5 K8s Deploy

In this last phase, the information of the just created container image is used to create the Pod in the Rancher instance. To do so, the operations performed are:

1. creation of the Pod, by using the Front-End container image;
2. exposition of the listening port 80 of the Pod and binding of the API gateway. In this way the Back-End APIs are reachable from the Pod;
3. binding of the Pod listening port 80 to the Ingress port 80, exposition of the Tool URL to be reachable from the client, and addition of the TLS certificate to ensure secure data exchange.

3.4 Implementation

In this next section, I will convey the most important implementation aspects of the creation of the Data Entry Tool.

3.4.1 Multiple Database Governance

Interacting with different databases

One of the challenges I had to deal with was having to manage more than one database at a time. All databases are different from one another, because of the DBMS used, the very different structure of the tables that exist in those databases, or the different Entity-Relationship schemas that they have to manage. This is one of the reasons I did not use an Object-Relational Mapping; this kind of technique makes it possible to query and manipulate data from a database using an object-oriented paradigm; however, to do so it is needed to know in advance the underlying data structure of the tables it tries to manipulate. With the Data Entry Tool this is not possible since the databases, or the tables that reside there, are not known in advance. Another reason not to use an ORM library is that there is no library in TypeScript that can handle all the required DBMSs at the same time.

With the ORM out of the equation, I had to find another solution to handle the databases. Fortunately, I found Knex.js [35], an SQL query builder that allows to create queries via method chaining, shown in Code Sample 3.4, has transaction support, and works with all the required DBMSs. The ability to build a query through method chaining made it possible to create different queries based on the execution flow.

```
import knex from 'knex'

const db = knex(...)

db.select(...)
    .from(...)
    .where(...)
```

Code Sample 3.4: Example of query creation with Knex.

Knex uses the parameters provided when a database is first added to the Tool, i.e. the database name, connection host and port, database user and password, to establish a connection and perform operations on it. However, since establishing a connection takes time, whenever a user has to perform an operation on a database, after creating the connection, I cache it in a

dictionary. This is done to follow the temporal locality principle: if a user needs to perform an operation on a given table in a given database, then s/he is likely to perform other operations on that particular table in the near future. This way, any user who needs to access that database, even if for a different table, does not have to establish a new connection to perform operations. Moreover, any operation performed on any database registered in the Tool, or on the Aurora instance on which it is based, is performed through a transaction. This is done to prevent any concurrency while interacting with the data, given the dynamic nature of the databases and their distributed access.

Automatic column retrieval

When a new table is added to the Tool, it starts the process of automatic column retrieval. This process is done to save the user the tedious work of manually inserting all the columns of a table. However, since different DBMSs have different places where they store the metadata for each table, I had to create custom queries to retrieve the data.

Fortunately, both PostgreSQL and Microsoft SQL Server use the same set of tables, but OracleDB uses a different set, as displayed in Table 3.6.

Table 3.6: Tables containing metadata.

Engine	Schema	Tables
PostgreSQL Microsoft SQL SERVER	information_schema	• columns • table_constraints • key_column_usage
OracleDB	SYS	• ALL_TAB_COLUMNS • ALL_CONSTRAINTS • ALL_CONS_COLUMNS

Through the process of table joining, with the original name and schema of the new table added as background information, it is possible to get the elements that make up a column, or at least the set of essential information. This set consists of:

- knowing whether a column is a primary key;
- knowing whether the column can have empty/null values;
- knowing if the column can be modified. This is done by checking if it is a primary key and if it has a default value;
- the data type of the column. Since we cannot use all the possible data types available in all the DBMSs, the column data type is mapped to a generic data type that represents it. The data types available in this automatic process are:
  - BOOLEAN
  - STRING
  - NUMERIC
  - DATE
  - TIME
  - DATETIME

A View to ease information retrieval

The information regarding the association between a user and a table is spread across many tables. This means that, in order to reach the metadata of a table, given a user, many tables must be traversed, and doing so each time would have made the source code difficult to read and debug in case of problems. For this reason, I created a view that gathers in one place the essential information needed to retrieve metadata, to check a user's permissions, to correctly display a table, and to connect to the original database to perform operations.

Regarding permissions on a table, since a user can have access to one across multiple user groups, it makes sense to always enable the sum of permissions on that table regardless of where they access the table. For this purpose, the view contains a sub-query, visible in Code Sample 3.5, which performs an aggregation based on the username and the associated table. With it and the aggregation function `bool_or()`, it is possible to perform the logical addition of the permissions present in a user group for a given table accessible to a given user. In addition, an aggregation is done on the table fields, more precisely to check if among the fields of a table there is also one marked as primary key. This is necessary to prevent a user from trying to update or delete a record in a table where a primary key is not defined. Otherwise, when deleting or updating using the `where` condition, all elements that satisfy the condition are manipulated.

```
WITH users_tables_agg AS (
    SELECT
        u.username,
        ugt.table_id,
        bool_or(ugt.can_insert) AS can_insert,
        bool_or(ugt.can_update) AS can_update,
        bool_or(ugt.can_delete) AS can_delete,
        bool_or(ugt.can_import) AS can_import,
        bool_or(ugt.can_export) AS can_export,
        bool_or(ugt.can_perform_action) AS can_perform_action,
        bool_or(ugt.can_batch_delete) AS can_batch_delete,
        bool_or(tf.primary_key) AS has_primary_key
    FROM
        users u
    JOIN users_user_groups uug ON
        u.username = uug.username
    JOIN user_groups_tables ugt ON
        uug.group_id = ugt.group_id
    LEFT JOIN table_fields tf ON
        ugt.table_id = tf.table_id
    GROUP BY
        u.username,
        ugt.table_id
)
```

Code Sample 3.5: Sub-query of the view.

This view, however, is not a materialized view. The reason is that, since metadata is very dynamic, pre-computing a materialized view, and keeping it up-to-date, would take more time than would be saved by using it, since it has to be updated every time an element in the view changes.

The Filtering conundrum

In both the Administrator and User views of the Data Entry Tool, it is possible to filter the elements of a table according to the values of one or more columns. For the Administrator side, the columns to filter are predefined and never change; however the same cannot be said for the User side. On their side, the tables are dynamically generated and do not always reflect the structure present of their source databases. Just as the structure is dynamic, the number and type of columns to filter for is dynamic as well. Therefore, it was necessary to create a flexible and easy-to-use way to represent a filter. For this reason, I created a filter type with a structure that could be easily used by both the Administrator and User side with Knex's query building capabilities.

```
type Filter = {
    operation: Operations
    column: string
    toCompare: unknown | [unknown, unknown] | Array<unknown>
}
```

Code Sample 3.6: Composition of a filter.

In the Code Sample 3.6, we can see that a filter consists of an operation, described in Table 3.4, the name of the column to be filtered, and a comparison element represented by a union type. A *Union Type* describes a value that can be one of several types. Since it is not possible to know a priori the actual type of the element of comparison used in the process of filtering, this Union Type conceptually represents the number of elements for which to compare the values of a column. The represented types are, in order:

- *unknown*: this type represents a single value that can be used with comparison operations, i.e. greater than, equal, or similar to a value in the given column. This representation is used by columns that contains numbers, boolean, or alphanumeric strings.
- *[unknown, unknown]*: this type represents a range used to check if a value in the given column belongs to it or not. This representation is used by columns that contains numbers or temporal data.
- *Array<unknown>*: a generalized version of the previous type that is used by columns with alphanumeric values.

3.4.2 Serverless management

Lambda's environment

When building Serverless applications, it is quite common to have code that is shared between Lambda functions. This can be custom code, which is used by more than one function, or a standard library, which is used to simplify the implementation of the business logic. Instead of adding the dependencies to each Lambda, effectively increasing the storage size that they will use and the time needed to deploy a function, it is possible to centralize all the common dependencies they will use. A *Lambda Layer* is an archive that can contain libraries, a custom runtime environment, data, or configuration files. When a layer is included in a function, the content is extracted from the archive and used in the execution environment.

For the creation of the Lambda functions for the Data Entry Tool, I created two different layers, shown in Code Sample 3.7.

```
nodeModulesLayer:
    path: layers/layer_node_modules
    name: nodeModulesLayer
    description: Node Modules Layer

oracleClientLayer:
    path: layers/oracle_client
    name: oracleClientLayer
    description: Oracle Client Layer to manage database connection
```

Code Sample 3.7: Layers defined in the project.

The first layer, *nodeModulesLayer*, consists of all NodeJs dependencies used in all Lambda functions; since OracleDB is a proprietary DBMS, a specific client is needed to access it, and so the second layer, *oracleClientLayer*, is the client that is used to access it. However, for security reasons, only a small number of Lambda functions can use this layer.

With these layers, the average size of a Lambda function drops from 70MiB to 15KiB.

API Gateway naming problem

While developing the Serverless infrastructure, I discovered an issue regarding the definition of the API Gateway. Since OpenAPI is used to define the APIs used in the Back-End side, each time the API definition is updated, a new Back-End deployment must be performed. However, the deployment failed as the existing API Gateway was updated in place. To get around this problem, I created a script that runs during the Serverless Deploy step of the pipeline. The purpose of this script is to read the configuration file where the API Gateway is defined, and change the resource name that is being created by adding the deployment timestamp at the end of it. This way, the created resource is always "new", and the deployed one will be destroyed.

3.4.3 Front-End

Standardize the experience

Because the Administrator side of the Front-End has a static interface, meaning that it does not change amongst users who use it, many similar components can be reused with little or no change throughout the web application. Moreover, some elements can be reused in the end user side as well.

The main actions available with every table, i.e., insert, update, delete, and duplicate a row, are parameterized components and agnostic to the tables they must manage. While the actions of paging, sorting and filtering have only a common basic structure, but their functioning depends more strictly on the tables that use them.

For the administrator side, there are 4 sections whose behavior is more or less the same, namely pages for table collections, databases, tables and fields. Each one has a table with a specific number of columns and a form for the insertion or update of a row. For these pages, I created templates for the table and the form to facilitate their development, since what needs to change is the number of columns and the endpoint from which to retrieve them.

The end user side is very dynamic since it must adapt the interface according to the user. For this reason, it is not possible to use a predefined template such as a "silver bullet" for the construction of all possible tables. Therefore, their creation depends on the metadata received from the Back-End. As mentioned before, this metadata contains, other than the actual table information and list of permissions, the list of fields retrieved from the table "`Table Fields`"; after removing all the fields that are not visible, those that remain are mapped to the table

columns by checking the data type, applying the badges for filtering and sorting, and finally sorted in the order decided by the administrator.

Chapter 4

Performance and Validation

No performance requirements were established in the specifications provided for the development of the Data Entry Tool. However, since the project is based on the concept of Serverless, which, as mentioned in Chapter 2, means that resources are used on-demand and paid for based on usage, what we could do was optimize their use.

4.1 Implementation enhancements

One of the features that we improved was the API that retrieves the metadata for the dynamic creation of the tables for the end user.

The `/metadata` API not only retrieves information about the table and their permissions, but also collects all the fields that will be used to build the table. Of these fields, those with an *External Enum* datatype represent a list of possible values extracted from a lookup table, also recorded in the Tool. An example of this type of field can be seen in Figure 3.17b. These values represent the relationship between the tables as they are in effect foreign keys. Since the list of possible values for foreign keys is necessary for the proper construction of the table, and for any record addition or update, it means that, for each field representing a foreign key it is necessary to collect all possible values that can be used to refer to a record in the lookup table.

To collect the possible values for each External Enum field, you must query the source database of the lookup table to retrieve the set of values. It goes without saying that as the number of foreign keys for a table increases, the number of queries increases, and each query requires a connection to the database, which can increase the latency and therefore the waiting time for the creation and use of the table by the end user. This is all considering a single end user of the Tool; with multiple users, the situation is multiplied.

One improvement that I made, described in Section 3.4.1, was reusing a previously created connection to a database instead of creating a brand new one every time. However, this shrewdness does not resolve the problem entirely, because it does not take into account either the amount of time needed by the database to actually perform the query, or the latency of the network where the information passes through. Network latency depends on where the databases are located relatively to the datacenter where the Data Entry Tool resides, while the time required for the database to complete the query processing depends on the amount of data to retrieve and the computing capabilities of the hosting machine. In addition, the API Gateway that routes requests to the appropriate Lambda function limits its duration to 29 seconds. This means that if the data cannot be retrieved in this time frame, an operation timeout will be returned to the user. This is an inherent limitation of AWS API Gateway that ensures a near real-time experience for applications using it. However, in these 29 seconds, the same Lambda function can accept multiple requests from the same or different users, optimizing its execution

by not tying it to a single user.

4.2 Cost-dependent performance

Taking into account the timing that cannot be controlled by the Tool, what I could do was run the queries in parallel to gather the data. However, performing parallel requests to retrieve the data requires a certain amount of CPU power. As mentioned earlier, resources are used on-demand and paid for based on usage, and by default Lambda functions are set to use a default amount of resources. This can reduce the expense, but also the computational performance of the function.

AWS Lambda functions are charged monthly based on **number of requests** and **duration**, which is the code execution time. A request is represented by the moment when the code starts running in response to an event, which in our case comes from the Gateway API, and AWS Lambda charges the total number of requests across all the functions used. Instead, the duration is calculated on the time interval from when the code start executing until it returns or is terminated, rounded to the nearest upper 100 milliseconds. This means that 100 milliseconds of execution is charged even if it takes less than that. The final price of the Lambda function depends on the amount of memory (RAM) allocated to the function; furthermore, the computational power associated with the function depends linearly on the allocated memory. The final price calculation is composed of the sum of the monthly compute charges with the monthly request charges.

$$RequestCharges = \rho * \sigma$$

$$ComputeCharges = \rho * \alpha * \frac{\gamma}{1024\,\text{MB}} * \chi$$

$$TotalMonthlyCharges = ComputeCharges + RequestCharges$$

- $\boldsymbol{\rho} \rightarrow$ requests per month.
- $\boldsymbol{\sigma} \rightarrow$ price per million of requests.
- $\boldsymbol{\alpha} \rightarrow$ average Lambda function execution time in seconds.
- $\boldsymbol{\gamma} \rightarrow$ amount of RAM provided in MegaByte.
- $\boldsymbol{\chi} \rightarrow$ price per GigaByte used in a second.

Let us say, as an example, that the Lambda function that retrieves metadata receives one million requests in a month (ρ) at a per-million price of \$0.20 (σ), has an average execution time of one second (α), and 1024 MB of RAM provided (γ) at a price of \$0.000016667 (χ) per GB used in one second. With this data we can see that in one month AWS would charge \$16.86 just for this function alone.

Knowing how AWS Lambda charges for its usage, we can take the amount of memory as a variable and see how much its variation affects computational performance, thus determining the execution time of the Lambda function that collects the metadata. Figure 4.1 shows the trend of the execution time on a sample of requests taken directly from the AWS console of the Lambda function to avoid latency due to connection, API Gateway and Lambda Authorizer. In the figure, we can see the different computational performance as the memory increases. When switching from 512 MB to 1024 MB of RAM, the execution time is almost halved, while doubling it further to 2048 MB or 4096 MB we reach a point of diminishing returns where the

performance gain is not such as to justify, respectively, doubling or quadrupling the costs. The Table 4.1 further emphasizes the increase in performance.

Keeping in mind the balance between cost and computational performance, I decided to allocate an amount of memory equal to 1024 MB to this Lambda function, which deals not only with collecting metadata, but also with performing manipulations on the table data on behalf of the user. Following this logic, I have reduced the computational power of those Lambda functions that do not require many resources to be executed in a performant way. All the functions managed on the Administrator side belong to this category.

Table 4.1: Performance gain by increasing memory.

	512 MB	1024 MB	2048 MB	4096 MB
MIN	1842.00 ms	1.00 ms	8.00 ms	523.00 ms
MAX	3364.00 ms	1854.00 ms	1743.00 ms	1249.00 ms
AVG	2025.83 ms	1121.24 ms	769.25 ms	712.60 ms
95%	2245.60 ms	1437.75 ms	1207.15 ms	1031.80 ms

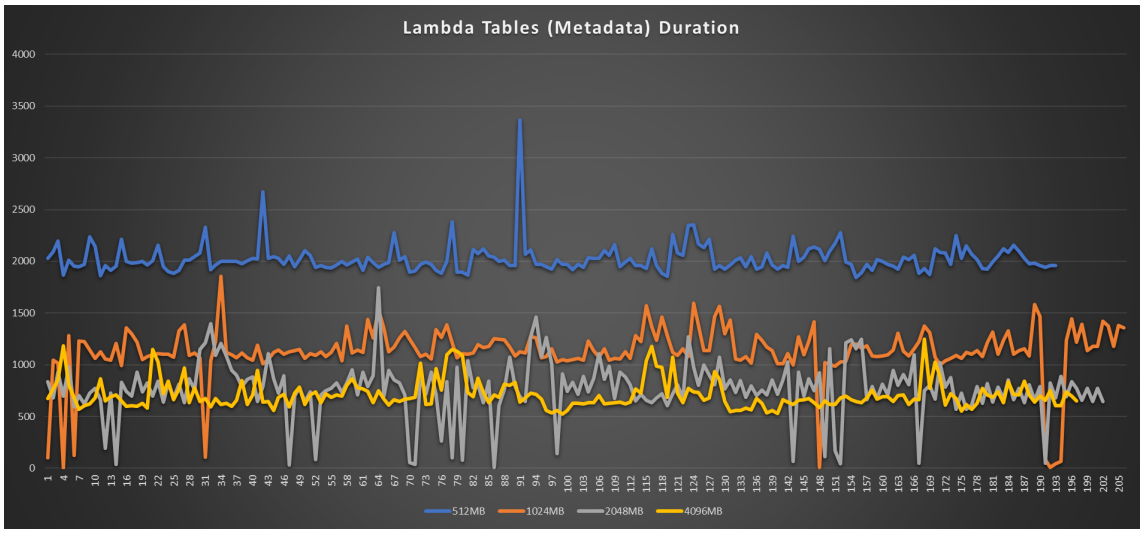


Figure 4.1: Lambda execution time based on the memory provided.

4.3 Validation

4.3.1 Testing during development

To ensure that the Data Entry Tool met the requirements, it underwent several tests. During the development of the Back-End side, the tests were performed through the use of the Serverless Framework which allows the entire Back-End code to run locally on the development machine. This way there was no need to deploy unfinished APIs in the development environment. The Serverless Framework instantiated a local server, listening on port 3000, to which it attached the APIs, in effect creating a local API Gateway. Lambda functions that are tied to the APIs were run on the local machine, so that it was possible to see the execution steps and logs on the terminal to verify that the function was working as expected. However, the Aurora instance at the core of the Tool, for both development and production, and all test databases used were hosted in the AWS datacenter.

On the other hand, during the development of the Front-End side of the tool, just like the Back-End, we were running the application locally; however, we were using the versions of the APIs deployed in the development environment as endpoints for the Back-End requests. The tests focused on the correct display of the information received from the Back-End and the correct interaction with the data located in the different types of databases, always taking into account the speed of data display to maintain a near real-time experience.

4.3.2 Meeting the requirements

To ensure that the Data Entry Tool met the expectations of the client company, it was tested through various use cases and underwent the User Acceptance Test. Some of the use cases that have been provided by the client are:

- Division of user access according to the given permissions.
- Using tables independently of the underlying database.
- Using tables with tens of columns with different datatypes.
- Loading the Excel file to populate the table.
- Logging of every operation.

After developing the frontend side of the Data Entry Tool, the project would periodically go through the User Acceptance Test (UAT), where the client would test it to personally verify that everything they had requested was in place, and if there were adjustments or improvements to be made, these would be fixed and resubmitted at the next UAT. This process of reiterating development with UATs was maintained for an additional 2 weeks after delivery of the finished project, as a hypercare period was established in the contract in case of problems.

4.3.3 Compared to the pre-existing solution

The pre-existing solution to manipulate data in multiple tables existing in different databases consisted in a web page tailored for each table. This means that every time there was a need to interact with a new table created in a database, the client had to spend some time creating a new web page to interact with directly, without any check either on the datatype or whether there were any constraints to respect. These web pages had no concept of permissions, meaning that any user had indiscriminate access to any table, and could potentially modify its contents improperly. Moreover, since it was not possible to import data by uploading an Excel file to the page for a particular table, they had decided to store these files in a shared directory without updating the corresponding tables, which in many cases had led to inconsistent data. With the expanding infrastructure the client was experiencing, this ongoing effort and expense of creating web page after web page for each table was no longer sustainable.

By developing this project as a plug and play solution, the previous problems have disappeared; administrators need just to make a particular database reachable from their private network, register it in the Tool, add the desired tables, and give permissions to a set of users to interact with them. Everything else is taken care of by the Data Entry Tool.

Chapter 5

Conclusions

As always when you create something new, you learn a lot in the process. Through this experience, I learned a lot about creating Cloud-based applications and how they are used in real-world solutions. The Data Entry Tool developed is able to sustain heavy loads of requests coming from many users and handle the management of the different databases used by the client. If there is a need to add support for other DBMS the process is quite simple, as you would only need to add in the Back-End the client needed to interact with the database, and verify that all data types can be mapped to the types described above.

Now the client company, through this application, will not have to spend too much time and resources in creating interfaces to manage all the possible tables available in its databases, since, as per the initial goal, they will be created dynamically and will only have to be managed by designated administrators.

Moreover, to conclude, a peculiarity of this application is that it is not strictly linked to the environment of the client company for which it was created; since its operation, as already mentioned, is agnostic with respect to the databases it has to manage, it can be used as a stand-alone application and proposed to other client companies.

5.1 Future Development

The Data Entry Tool could be further developed to add more functionalities. Some example of this would be:

- Adding a system for administrators to approve changes made by end users to tables. This would allow all changes made to sensitive tables to remain in a pending state until they are marked as approved.

- The integration of data analysis elements, with the ability to create graphs within the application. Since the framework used for Front-End development provides the necessary to create graphs based on the input data, it would make them significantly clearer to understand, rather than having to export them, perhaps in considerable size, to create the graphs elsewhere. Moreover, with the integration of a real-time information stream from the databases, some graphs could be added to the end user's dashboard where they would be able to see the graphs update in real time.

- A system to promote tables between environments: instead of adding a table directly in the production environment, there could be a simulated copy of it in the development environment, structurally identical, where it would be possible to fine-tune the behaviour

of the interface. Once ready, the created configuration can be promoted to the original copy.

Bibliography

[1] *Cloud computing services - amazon web services (aws).* [Online]. Available: https://aws.amazon.com/.

[2] *Cloud computing.* [Online]. Available: https://cloudacademy.com/course/cloud-fundamentals-1845/cloud-computing/.

[3] *Main cloud service models.* [Online]. Available: https://www.cloudflare.com/learning/cloud/what-is-the-cloud/.

[4] *Infrastructure as a service.* [Online]. Available: https://www.cloudflare.com/learning/cloud/what-is-iaas/.

[5] *Software as a service.* [Online]. Available: https://www.cloudflare.com/learning/cloud/what-is-saas/.

[6] *Private cloud.* [Online]. Available: https://www.cloudflare.com/learning/cloud/what-is-private-cloud/.

[7] *Public cloud.* [Online]. Available: https://www.cloudflare.com/learning/cloud/what-is-a-public-cloud/.

[8] *Hybrid cloud.* [Online]. Available: https://www.cloudflare.com/learning/cloud/what-is-hybrid-cloud/.

[9] *Monolithic vs. microservices.* [Online]. Available: https://www.cloudflare.com/learning/serverless/glossary/serverless-microservice/.

[10] *Benefits of serverless.* [Online]. Available: https://www.cloudflare.com/learning/serverless/what-is-serverless/.

[11] *Function as a service.* [Online]. Available: https://www.cloudflare.com/learning/serverless/glossary/function-as-a-service-faas/.

[12] *Amazon api gateway.* [Online]. Available: https://aws.amazon.com/api-gateway/.

[13] *Amazon lambda.* [Online]. Available: https://aws.amazon.com/lambda.

[14] *Amazon aurora.* [Online]. Available: https://aws.amazon.com/rds/aurora/.

[15] *Aurora replication schema.* [Online]. Available: https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.Overview.html.

[16] *Amazon simple queue service.* [Online]. Available: https://aws.amazon.com/sqs.

[17] *Amazon elastic container registry.* [Online]. Available: https://aws.amazon.com/ecr.

[18] *Aurora replication schema.* [Online]. Available: https://aws.amazon.com/cloudformation.

[19] *Nodejs.* [Online]. Available: https://nodejs.org/.

[20] J. Stein, "Chapter 8: Creating an api with node.js using mongodb and mysql," in *Reactjs cookbook.* Packt Publishing Limited, 2016, p. 265.

[21] *Nodejs event loop*. [Online]. Available: https://www.tutorialandexample.com/node-js-event-loop.

[22] *Typescript language*. [Online]. Available: https://www.typescriptlang.org/.

[23] *Typescript language*. [Online]. Available: https://www.serverless.com/.

[24] *Openapi specification*. [Online]. Available: https://swagger.io/specification/.

[25] *Flyway - database version control system*. [Online]. Available: https://flywaydb.org/.

[26] *Migration process*. [Online]. Available: https://flywaydb.org/documentation/concepts/migrations#sql-based-migrations.

[27] *React*. [Online]. Available: https://reactjs.org/.

[28] *Antdesign pro*. [Online]. Available: https://pro.ant.design/.

[29] *Docker*. [Online]. Available: https://www.docker.com/.

[30] *Containerized applications*. [Online]. Available: https://www.docker.com/resources/what-container.

[31] *Kubernetes*. [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[32] *Rancher*. [Online]. Available: https://rancher.com/.

[33] *What is rancher*. [Online]. Available: https://www.openlogic.com/blog/what-is-rancher.

[34] *Nginx*. [Online]. Available: https://www.nginx.com/.

[35] *Knex*. [Online]. Available: http://knexjs.org/.