University of Padua

Department of Information Engineering

Design and development of a cloud serverless solution for single sign-on authentication

Author:
Giovanni Veronelli

Supervisor:
Prof. Nicola Ferro

A thesis submitted for the degree of

MSc Computer Engineering

Academic Year 2019/2020

Abstract

Serverless technology is a way of creating applications where cloud vendors deal with provisioning, scalability and maintenance of infrastructures, giving the developers a rapid way of development and deployment. The aim of this thesis is to extend knowledge on the serverless services and its possibility on Amazon Web Services (AWS): in order to support this architecture a practical example is given. The design and development of a web application was made in a real IT company that gave the possibility of a four months internship to the author of this thesis.
The application name is PASS, and its main function is to give a middle layer between an authentication single sign on (SSO) platform and a group of applications: the latter can share the same profiles of users, standardizing and managing the proliferation among applications.
The thesis is divided into four main parts: the first and the second focus on creating a background for the tools, concepts and services used, while the third explains the business requirements analyzing them and providing the architecture design of PASS. The fourth part shows how the practical implementation is done and how the application works, with views and the backend logic put in practice. In the end, conclusions are wrapped up considering the student's personal internship experience and the overall application work done.

Acknowledgements

This thesis is the result of the research and development activity carried out at SCAI ITEC during four months of internship. The aim of this project was to strengthen my knowledge on the cloud and the web development.

I would express my gratitude towards my company SCAI ITEC and Marco Stefani for the mentoring.

I would like to thank my supervisor, Nicola Ferro, for the advices and the reviewing work of my thesis.

to my family

Contents

1 Introduction **7**

2 Concepts **8**

2.1 Cloud computing 8

2.1.1 Definition 8

2.1.2 Architecture 8

2.2 Serverless computing 10

2.2.1 History of serverless 11

2.3 Nested Set Model 12

2.3.1 Example 12

2.4 Single sign-on 13

2.5 RESTful API 14

3 Technologies and services **16**

3.1 Languages 16

3.1.1 Typescript 16

3.1.2 Node.js 17

3.2 AWS Lambda 18

3.2.1 Runtime 18

3.2.2 Execution Context 19

3.3 Database 20

3.3.1 Amazon Aurora Serverless 20

3.4 AWS API Gateway 22

3.5 AWS Cognito 22

3.6 AWS Simple Storage Service 22

3.7 Serverless Framework 23

3.8 Express Js 23

3.9 Sequelize . . . 24
3.10 Angular . . . 24

4 Architecture design **26**
4.1 Business requirements . . . 26
4.2 Requirement analysis . . . 27
4.3 Functional requirements . . . 27
4.4 Cloud architecture . . . 28
4.5 Database design . . . 29
4.6 Entity relation schema . . . 31
4.7 Entity table . . . 31
4.8 Relationship table . . . 36
4.9 API design . . . 36

5 PASS demo **39**
5.1 Back-end . . . 39
5.2 Front-end . . . 40

6 Conclusion **48**
6.1 Internship experience . . . 48

A Lambda insights **50**
A.0.1 Permissions . . . 50
A.0.2 Invoking Functions . . . 51
A.0.3 Error handling and automatic retries . . . 53
A.0.4 Function Scaling . . . 54
A.0.5 Handler in Node.js . . . 55
A.0.6 Logging in Node.js . . . 56
A.0.7 Errors in Node.js . . . 58
A.0.8 Monitoring and troubleshooting . . . 59

Bibliography **62**

List of Figures

2.1 Cloud computing architecture 9
2.2 Serverless platform architecture 11
2.3 A hierarchy: types of clothing 13
2.4 Single sign-on basic schema 14

3.1 Aurora Serverless: logic schema 21

4.1 Cloud architecture 28
4.2 Entity relation schema 31

5.1 Front-end - general view 41
5.2 Front-end - application component view 42
5.3 Front-end - application add form view 42
5.4 Front-end - application delete form view 43
5.5 Front-end - application detail view 44
5.6 Front-end - user component view 45
5.7 Front-end - user detail view 46
5.8 Front-end - roles component view 47

A.1 Synchronous, Asynchronous and Stream invocations for AWS Lambda . . . 53
A.2 Example index.js file logging in AWS Lambda 57
A.3 Example log format in AWS Lambda 57
A.4 Example index.js file 58
A.5 Example reference error response JSON 58
A.6 CloudWatch metrics: view example 60

List of Tables

2.1 The resulting rapresentation 12

4.1 Adjacency list, nested set model comparison 30

4.3 Relationship table 36

4.4 API definitions table 37

Chapter 1

Introduction

The company SCAI ITEC deals with Consulting, System Integration and Application Management in the ICT field, operating in all the main market sectors: banking and insurance, industry, public administration and services. It is located in Padua and works with customers in the entire Veneto region. ITEC belongs to a group named SCAI; its aims are to have companies to work in all fields of ICT in order to satisfy all customers needs nationally and internationally.
During his four months internship in SCAI ITEC the grad student designed and developed a web application to standardize and shave user profiles between different applications.
The application name is PASS: it is consists of an independent client made in Angular and a backend developed with serverless framework based on a cloud infrastructure. This work will analyze the technologyes and the basic concepts behind the realization of the application. The thesis is divided into four main parts: the first and the second focus on creating a background for the tools, concepts and services used, while the third explains the business requirements analyzing them and providing the architecture design of PASS. The fourth part shows how the practical implementation is done and how the application works, with views and the backend logic put in practice. In the end, conclusions are wrapped out considering the student's personal internship experience and the overall application work done.

Chapter 2

Concepts

In this chapter paradigms and models are explained for a better understanding of the design of the application in chapter 4.

2.1 Cloud computing

2.1.1 Definition

The National Istitute of Standards and Technology (NIST) gives a detailed and formalized definition: *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction* [1].

2.1.2 Architecture

The architecture of a cloud computing environment can be divided into 4 layers [2]: the hardware/datacenter layer, the infrastructure layer, the platform layer and the application layer, as shown in Fig. 2.1. A description of the layers can be the following:

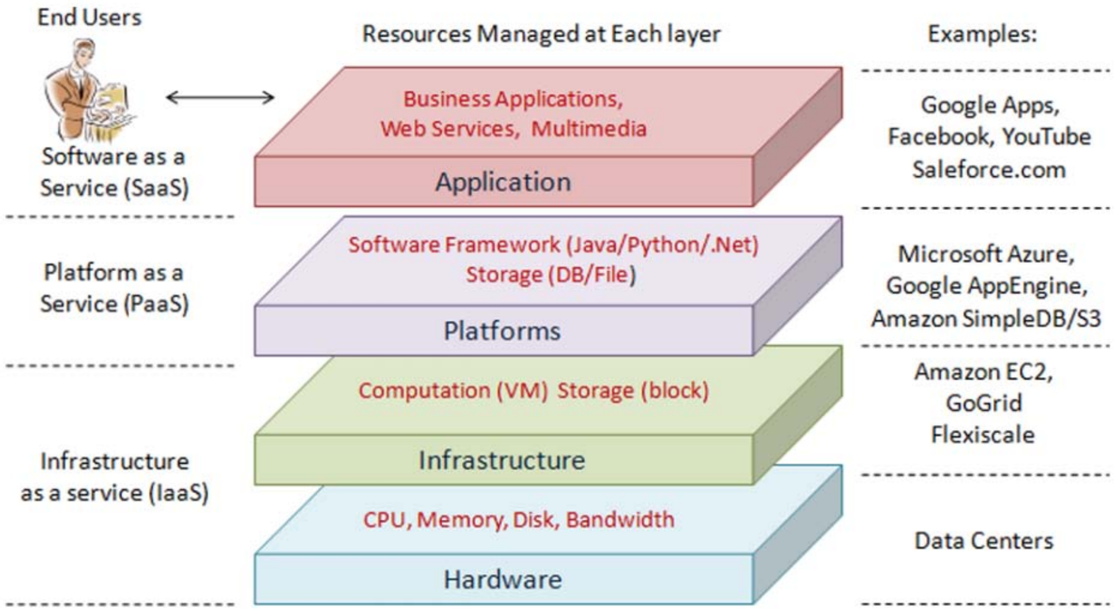

Figure 2.1: Cloud computing architecture

- The hardware layer - this layer is responsible for managing the physical resources of the cloud, including physical servers, routers, switches, power and cooling systems. In practice, the hardware layer is typically implemented in data centers. A data center usually contains thousands of servers that are organized in racks and interconnected through switches, routers or other fabrics. Typical issues at hardware layer include hardware configuration, fault tolerance, traffic management, power and cooling resource management;

- The infrastructure layer - also known as the virtualization layer, the infrastructure layer creates a pool of storage and computing resources by partitioning the physical resources using virtualization technologies;

- The platform layer - Built on top of the infrastructure layer, the platform layer consists of operating systems and application frameworks. The purpose of the platform layer is to minimize the burden of deploying applications directly into VM containers;

- The application layer - at the highest level of the hierarchy, the application layer consists of the actual cloud applications. Different from traditional applications, cloud applications can leverage the automatic-scaling feature to achieve better performance, availability and lower operating cost. The architectural modularity allows

cloud computing to support a wide range of application requirements while reducing management and maintenance overhead.

2.2 Serverless computing

Function as a service (FaaS), is the service which serverless computing is based, it is a category of cloud computing services that provides a platform allowing customers to develop, run, and manage application functionalities without the complexity of building and maintaining the infrastructure Considering fig. A.6 FaaS is in the platforms layer as PaaS. Serverless computing is a cloud computing model in which code is run as a service without the need for the user to maintain or create the underlying infrastructure. This does not mean that serverless architecture does not require servers, but instead that a third party is managing these servers so they are abstracted away from the user.
There are a lot of misconceptions surrounding serverless starting with the name. Servers are still needed, but developers need not concern themselves with managing those servers. Decisions such as the number of servers and their capacity are taken care of by the serverless platform, with server capacity automatically provisioned as needed by the workload. This provides an abstraction where computation (in the form of a stateless function) is disconnected from where it is going to run. The core capability of a serverless platform is that of an event processing system, as depicted in Fig. 2.2. The service must manage a set of user-defined functions, take an event sent over HTTP or received from an event source, determine which function(s) to which to dispatch the event, find an existing instance of the function or create a new instance, send the event to the function instance, wait for a response, gather execution logs, make the response available to the user, and stop the function when it is no longer needed.
The challenge is to implement such functionality while considering metrics such as cost, scalability, and fault tolerance. The platform must quickly and efficiently start a function and process its input. The platform also needs to queue events, and based on the state of the queues and arrival rate of events, schedule the execution of functions, and manage stopping and deallocating resources for idle function instances. In addition, the platform needs to carefully consider how to scale and manage failures in a cloud environment. [3]

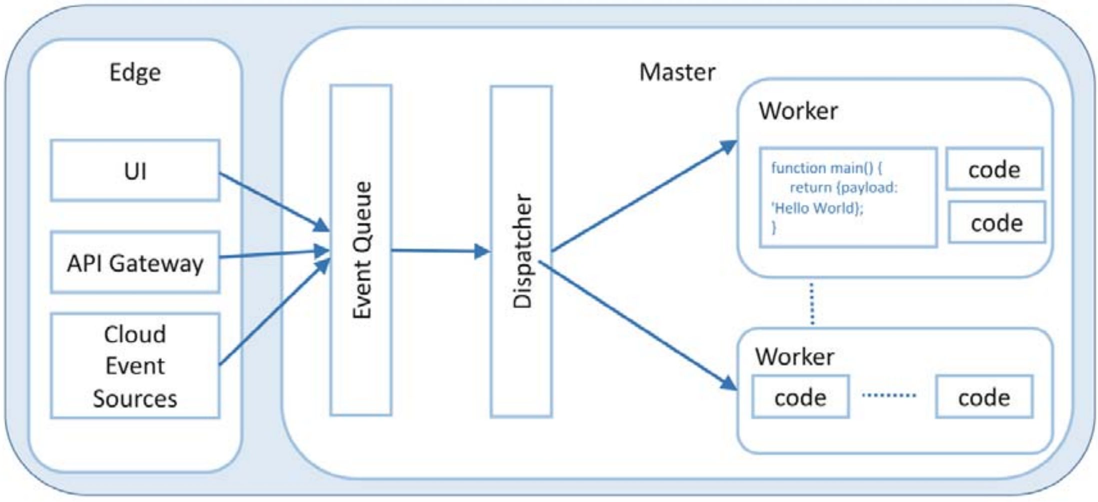

Figure 2.2: Serverless platform architecture

2.2.1 History of serverless

Serverless computing was popularized by Amazon in the re:invent 2014 session. Other vendors followed in 2016 with the introduction of Google Cloud Functions, Microsoft Azure Functions, and IBM OpenWhisk. However, the serverless approach to computing is not completely new. It has emerged following recent advancements and adoption of virtual machine (VM) and then container technologies.
Before traiting Serverless architecture where the abstraction of the infrastructure is really strong, it is better to talk about the past of this technology.
Infrastructure as a Service (IaaS) provides computing infrastructure, physical or virtual machines and other resources like virtual-machine disk image library, block, and file-based storage, firewalls, load balancers, IP addresses, and virtual local area networks. An example of this is an Amazon Elastic Compute Cloud (EC2) instance.
Platform as a Service (PaaS) provides computing platforms which typically includes the operating system, programming language execution environment, database, and web server. Some examples include AWS Elastic Beanstalk, Azure Web Apps, and Heroku. Software as a Service (SaaS) provides a dev with access to application software. The installation and setup are removed from the process and develeoper left with the application.

2.3 Nested Set Model

The nested set model is a technique for representing nested sets (also known as trees or hierarchies) in relational databases. The nested set model is to number the nodes according to a tree traversal, which visits each node twice, assigning numbers in the order of visiting, and at both visits. This leaves two numbers for each node, which are stored as two attributes (LFT and RGT). Querying becomes inexpensive: hierarchy membership can be tested by comparing these numbers. Updating requires renumbering and is therefore expensive. Refinements that use rational numbers instead of integers can avoid renumbering, and so are faster to update, although much more complicated. [4]

2.3.1 Example

In a clothing store catalog, clothing may be categorized according to the hierarchy given in figure 2.4.

Node	**Left**	**Right**
Clothing	1	22
Men's	2	9
Women's	10	21
Suits	3	8
Slacks	4	5
Jackets	6	7
Dresses	11	16
Skirts	17	18
Blouses	19	20
Evening Gowns	12	13
Sun Dresses	14	15

Table 2.1: The resulting rapresentation

The "Clothing" category, with the highest position in the hierarchy, encompasses all subordinating categories. It is therefore given left and right domain values of 1 and 22, the latter value being the double of the total number of nodes being represented. The next hierarchical level contains "Men's" and "Women's", both containing levels within themselves that must be accounted for. Each level's data node is assigned left and right

domain values according to the number of sublevels contained within, as shown in the table data.

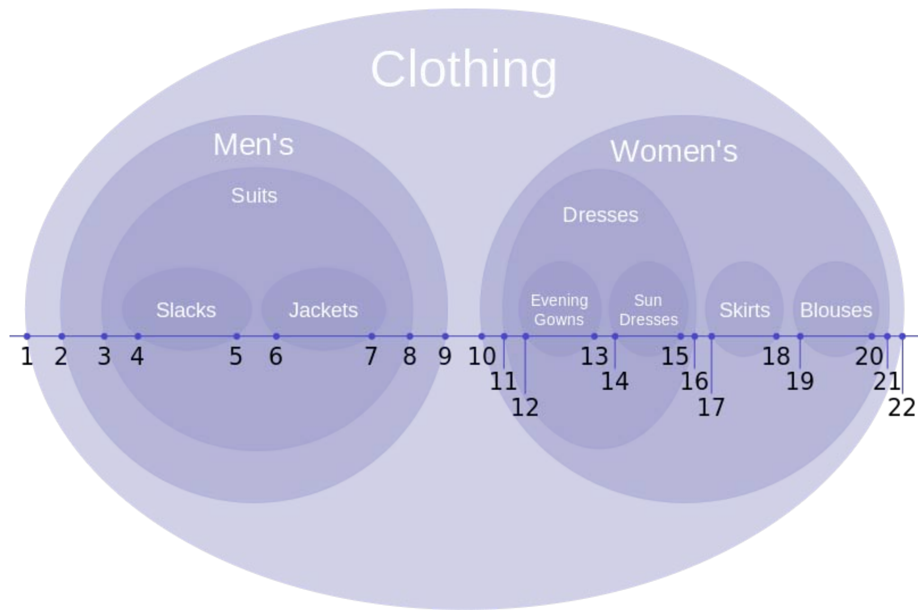

Figure 2.3: A hierarchy: types of clothing

2.4 Single sign-on

Single sign-on (SSO) is a session and user authentication service that permits a user to use one set of login credentials (e.g., name and password) to access multiple applications. SSO can be used by enterprises, smaller organizations, and individuals to mitigate the management of various usernames and passwords. In a basic web SSO service, an agent module on the application server retrieves the specific authentication credentials for an individual user from a dedicated SSO policy server, while authenticating the user against a user repository such as a lightweight directory access protocol (LDAP) directory. The service authenticates the end user for all the applications the user has been given rights to and eliminates future password prompts for individual applications during the same session. [5]

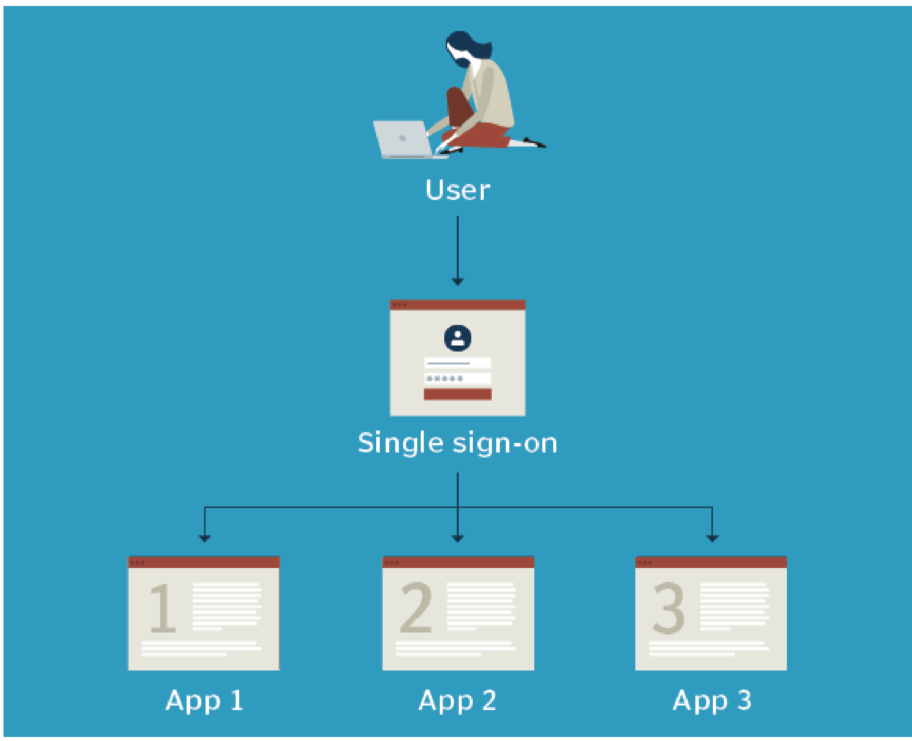

Figure 2.4: Single sign-on basic schema

2.5 RESTful API

REST (REpresentational State Transfer) is an architectural style for developing web services. REST is popular due to its simplicity and the fact that it builds upon existing systems and features of the internet's Hypertext Transfer Protocol (HTTP) in order to achieve its objectives, as opposed to creating new standards, frameworks and technologies.

A RESTful API is an application program interface (API) that uses HTTP requests to GET, PUT, POST and DELETE data. A RESTful API is based on representational state transfer (REST) technology. RESTful API design was defined by Dr. Roy Fielding in his 2000 doctorate dissertation [6]. In order to be a true RESTful API, a web service must adhere to the following six REST architectural constraints:

- Use of a uniform interface (UI) - resources should be uniquely identifiable through a single URL, and only by using the underlying methods of the network protocol, such as DELETE, PUT and GET with HTTP, should it be possible to manipulate

a resource;

- Client-server based - there should be a clear delineation between the client and server. UI and request-gathering concerns are the client's domain. Data access, workload management and security are the server's domain. This loose coupling of the client and server enables each to be developed and enhanced independent of the other;
- Stateless operations - all client-server operations should be stateless, and any state management that is required should take place on the client, not the server;
- RESTful resource caching - all resources should allow caching unless explicitly indicated that caching is not possible;
- Layered system - REST allows for an architecture composed of multiple layers of servers;
- Code on demand - most of the time a server will send back static representations of resources in the form of XML or JSON. However, when necessary, servers can send executable code to the client.

Chapter 3

Technologies and services

In this chapter the technologies, languages, frameworks and services used in the project are presented. The aim is to provide a background to understand the following sections of design and implementation.

3.1 Languages

The languages used in this project are HTML, CSS and typescript with Angular framework, Node.js with Serverless framework.
Since html and css are not new languages, they are given for known, instead typescript and node introductions follows.

3.1.1 Typescript

TypeScript is an open-source object-oriented language developed and maintained by Microsoft, licensed under Apache 2 license. It is a typed superset of JavaScript that compiles to plain JavaScript. TypeScript was developed under Anders Hejlsberg, who also led the creation of the C# language. TypeScript was first released in October 2012.[7]

TypeScript Features

- Cross-Platform - TypeScript runs on any platform that JavaScript runs on. The TypeScript compiler can be installed on any operating system such as Windows, MacOS and Linux;

- Object Oriented Language - TypeScript provides powerful features such as Classes, Interfaces, and Modules;

- Static type-checking - TypeScript uses static typing. This is done using type annotations. It helps type checking at compile time. Additionally, using the type inference mechanism, if a variable is declared without a type, it will be inferred based on its value;
- Optional Static Typing - TypeScript also allows optional static typing if JavaScript's dynamic typing is used;
- DOM Manipulation - Just like JavaScript, TypeScript can be used to manipulate the DOM for adding or removing elements;
- ES 6 Features - TypeScript includes most features of planned ECMAScript 2015 (ES 6, 7) such as class, interface, Arrow functions etc.

3.1.2 Node.js

Node.js is an open source, cross-platform runtime environment for developing server-side and networking applications. Node.js applications are written in JavaScript, and can be run within the Node.js runtime on OS X, Microsoft Windows, and Linux.
Node.js also provides a rich library of various JavaScript modules which simplifies the development of web applications.[8]

Node.js Features

Following are some of the important features that make Node.js the first choice of software architects.

- Asynchronous and Event Driven – All APIs of Node.js library are asynchronous, that is, non-blocking. It essentially means a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call;
- Very Fast – Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution;
- Single Threaded but Highly Scalable – Node.js uses a single threaded model with event looping. Event mechanism helps the server to respond in a non-blocking way and makes the server highly scalable as opposed to traditional servers which create limited threads to handle requests. Node.js uses a single threaded program and

the same program can provide service to a much larger number of requests than traditional servers like Apache HTTP Server;

- No Buffering − Node.js applications never buffer any data. These applications simply output the data in chunks;

- License − Node.js is released under the MIT license.

An important mention goes to map reduce and filter functions that are used to manage data, also promises and async/await methods to handle with asynchronous methods.

3.2 AWS Lambda

Amazon Web Service (AWS) Lambda is the core of a serverless application in the Amazon cloud. AWS Lambda is an event-driven, serverless computing platform. It is a computing service that runs code in response to events and automatically manages the computing resources required by that code. The runtime and execution context follows, meanwhile for more information about permisses, invocation methods, error handling, function scaling, Node.js interfaces and monitoring read appendix A.

3.2.1 Runtime

AWS Lambda supports multiple languages through the use of runtimes. A developer can choose a runtime when creating a function, and can change runtimes by updating his function configuration.
The underlying execution environment provides additional libraries and environment variables that can be accessed by its function code.
When a function is invoked, Lambda attempts to re-use the execution environment from a previous invocation if one is available. This saves time, preparing the execution environment, and allows to save resources (like database connections and temporary files in the execution context) to avoid creating them every time the function runs.
A runtime can support a single version of a language, multiple versions of a language, or multiple languages; runtimes specific to a language or framework version are deprecated when the version reaches end of life.
A custom runtime can be implemented so as to use other languages in Lambda.[9]

3.2.2 Execution Context

When AWS Lambda executes a Lambda function, it provisions and manages the resources needed to run the function. When a Lambda function is created, configuration information, such as the amount of memory and maximum execution time, are declared.
When a Lambda function is invoked, AWS Lambda launches an execution context based on the configuration settings provided. The execution context is a temporary runtime environment that initializes any external dependencies of a Lambda function code, such as database connections or HTTP endpoints: this allows subsequent invocations with better performances, because there is no need to reinitialize those external dependencies.
Lambda takes time to set up an execution context and do the necessary "bootstrapping", which adds some latency each time the Lambda function is invoked.
This latency can both be seen when a Lambda function is invoked for the first time or after it has been updated, because AWS Lambda tries to reuse the execution context for subsequent invocations of the Lambda function.
After a Lambda function is executed, AWS Lambda maintains the execution context for some time in anticipation of another Lambda function invocation. In practice, the service "freezes" the execution context after a Lambda function completes, and "thaws" the context for reuse, if AWS Lambda chooses to reuse the context when the Lambda function is invoked again. This execution context reuse approach has the following implications:

1. Objects declared outside of the function's handler method remain initialized, providing additional optimization when the function is invoked again. For example, if a Lambda function establishes a database connection, instead of reestablishing the connection, the original connection is used in subsequent invocations. A best practise is to add logic in the code to check if a connection exists before creating one;

2. Each execution context provides 512 MB of additional disk space in the /tmp directory. The directory content remains when the execution context is frozen, providing transient cache that can be used for multiple invocations;

3. Background processes or callbacks initiated by Lambda function that did not complete when the function ended resume if AWS Lambda chooses to reuse the execution context. A good practise is that a developer should make sure any background processes or callbacks in his code are complete before the code exits.

3.3 Database

3.3.1 Amazon Aurora Serverless

Amazon Aurora is a fully-managed relational database engine that's compatible with MySQL and PostgreSQL. Amazon Aurora Serverless is an on-demand, autoscaling configuration for Amazon Aurora. Aurora Serverless DB cluster is a DB cluster that automatically starts up, shuts down, and scales up or down its compute capacity based on application's needs.

With Aurora Serverless, a database endpoint can be created without specifying the DB instance class size. The minimum and maximum capacity can be configured. The database endpoint connects to a proxy fleet that routes the workload to a fleet of resources that are automatically scaled. Because of the proxy fleet, connections are continuous as Aurora Serverless scales the resources automatically based on the minimum and maximum capacity specifications. Database client applications don't need to change to use the proxy fleet. Aurora Serverless manages the connections automatically. Scaling is rapid because it uses a pool of "warm" resources that are always ready to service requests.

Instead of provisioning and managing database servers, Aurora capacity units (ACUs) are specified in the configurations. Each ACU is a combination of processing and memory capacity. Database storage automatically scales from 10 GiB to 64 TiB, the same as storage in a standard Aurora DB cluster.

The minimum and maximum ACU can be specified. The minimum Aurora capacity unit is the lowest ACU to which the DB cluster can scale down. The maximum Aurora capacity unit is the highest ACU to which the DB cluster can scale up. Based on settings, Aurora Serverless automatically creates scaling rules for thresholds for CPU utilization, connections, and available memory.

Aurora Serverless manages the warm pool of resources in an AWS Region to minimize scaling time. When Aurora Serverless adds new resources to the Aurora DB cluster, it uses the proxy fleet to switch active client connections to the new resources.

The capacity allocated to an Aurora Serverless DB cluster seamlessly scales up and down based on the load (the CPU utilization and number of connections) generated by a client application. It also scales to zero capacity when there are no connections for a 5-minute period.

Aurora Serverless scales up when capacity constraints are seen in CPU, connections, or memory. It also scales up when it detects performance issues that can be resolved by scaling up.

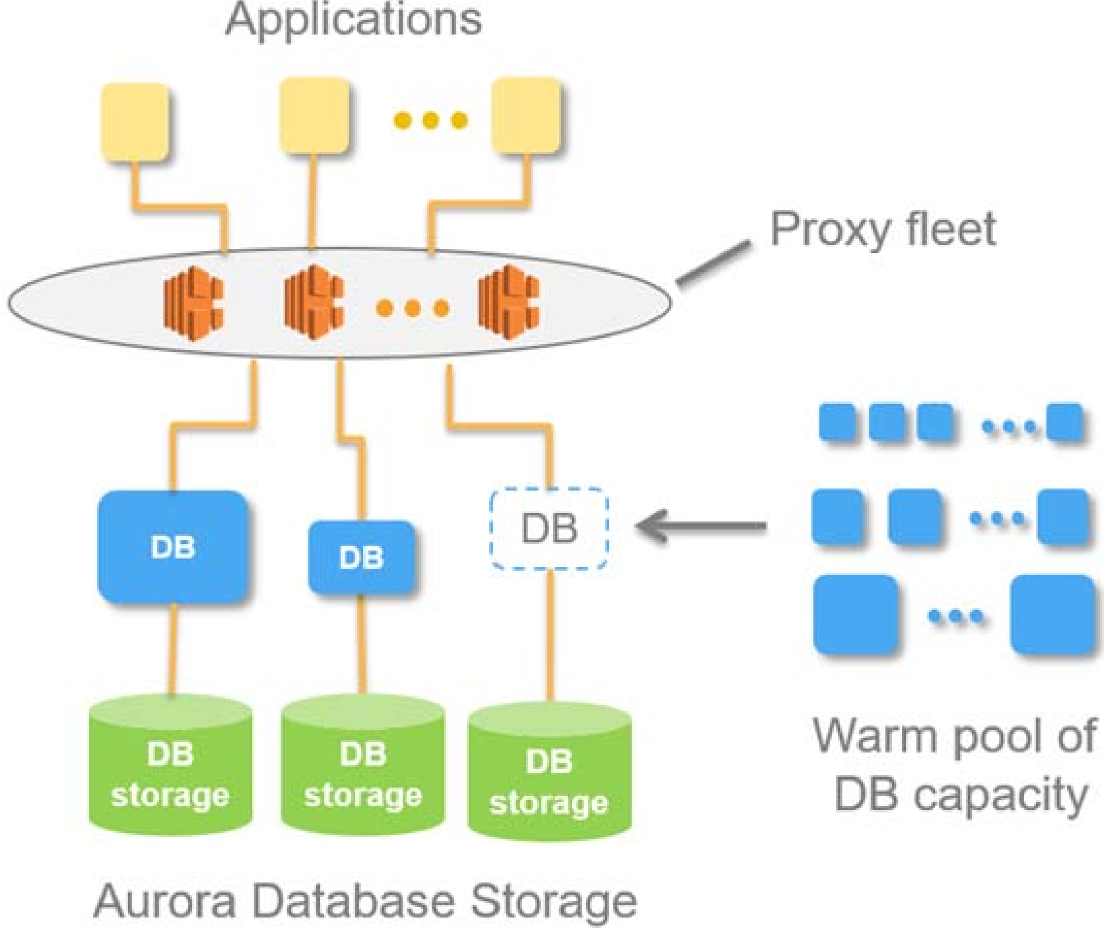

Figure 3.1: Aurora Serverless: logic schema

After scaling up, the cooldown period for scaling down is 15 minutes. After scaling down, the cooldown period for scaling down again is 310 seconds.

3.4 AWS API Gateway

Amazon API Gateway is an AWS service for creating, publishing, maintaining, monitoring, and securing REST APIs at any scale. API developers can create APIs that access AWS or other web services as well as data stored in the AWS Cloud.
API Gateway creates REST APIs that: are HTTP-based; adhere to the REST protocol, which enables stateless client-server communication; implement standard HTTP methods such as GET, POST, PUT, PATCH and DELETE. API Gateway acts as a "front door" for applications to trigger Lambdas.
Together with AWS Lambda, API Gateway forms the app-facing part of the AWS serverless infrastructure. Lambda can be used to interact with other AWS services and to expose its functions through API methods through API Gateway. To enable serverless applications, API Gateway supports streamlined proxy integrations with AWS Lambda and HTTP endpoints.

3.5 AWS Cognito

Amazon Cognito provides authentication, authorization, and user management for web and mobile apps. Users can sign in directly with a user name and password, or through a third party such as Facebook, Amazon, Google or Apple.
The component used of Amazon Cognito is the user pool. User pool is an user directory that provide sign-up and sign-in options for the app.
Easy integrable with API Gateway.

3.6 AWS Simple Storage Service

Amazon Simple Storage Service (Amazon S3) is a scalable, high-speed, web-based cloud storage service designed for online backup and archiving of data and applications on Amazon Web Services. Amazon S3 was designed with a minimal feature set and created to make web-scale computing easier for developers. Amazon S3 is used to host the static web page and make it publically accessible.

3.7 Serverless Framework

The Serverless Framework [10] helps building serverless apps with less overhead and cost. The Serverless Framework consists of an open source command-line interface (CLI) that makes it easy to develop, deploy and test serverless apps across different cloud providers, as well as a hosted Dashboard that includes features designed to further simplify serverless development, deployment, and testing, and enable to easily secure and monitor serverless apps.
The core of the Serverless Framework is the use of a single file yaml configuration that fully describe the cloud infrastructure. In that way a cloud architecture can be easy reproducible and shareable. Here is a list of all available properties in serverless.yml when the provider is set to aws. When the deploy command is invoked all the cloud infrastructure is created, obviously an AWS account is needed and fully configured in the CLI.
The advantage of Serverless Framework is the possibility of invoking the lambda locally with this cli command:

```
$ serverless invoke local --function functionName
```

This runs code locally by emulating the AWS Lambda environment. It's not a 100% perfect emulation, there may be some differences, but it works for testing without uploading the code each time and use the Lambda service, uselessly increasing the bill.
A lot of useful examples can be found in the following github repository: `https://github.com/serverless/examples`
For a fast configuration serverless components gives templates for common needs, findable in github repository here `https://github.com/serverless/components`.

3.8 Express Js

Express.js [11], or simply Express, is a web application framework for Node.js, released as free and open-source software under the MIT License. It is designed for building web applications and APIs. It has been called the de facto standard server framework for Node.js. To use Express in serverless environment serverless-http is adopted. This module allows to 'wrap' all APIs for serverless use, satisfying lambda handler function needs. More information are in the following github repository `https://github.com/dougmoscrop/serverless-http`.

3.9 Sequelize

Sequelize [12] is a promise-based Node.js Object Relational Mapping (ORM) for Postgres, MySQL, MariaDB, SQLite and Microsoft SQL Server. It features solid transaction support, relations, eager and lazy loading, read replication and more. To handle the tree structures the Nested Set Model (NSM) is adopted, fortunately there is a well done github project that implements all the formulation of NSM: `https://github.com/fremail/sequelize-nested-set`.

3.10 Angular

Angular is a TypeScript-based open-source web application framework led by the Angular Team at Google and by a community of individuals and corporations. Angular is a complete rewrite from the same team that built AngularJS. It is a platform and framework for building client applications in HTML and TypeScript. Angular is written in TypeScript. It implements core and optional functionality as a set of TypeScript libraries that are imported into apps. The basic building blocks of an Angular application are NgModules, which provide a compilation context for components. NgModules collect related code into functional sets; an Angular app is defined by a set of NgModules. An app always has at least a root module that enables bootstrapping, and typically has many more feature modules.

- Components define views, which are sets of screen elements that Angular can choose among and modify according to the program logic and data;
- Components use services, which provide specific functionality not directly related to views. Service providers can be injected into components as dependencies, making code modular, reusable, and efficient.

Both components and services are simply classes, with decorators that mark their type and provide metadata that tells Angular how to use them.

- The metadata for a component class associates it with a template that defines a view. A template combines ordinary HTML with Angular directives and binding markup that allow Angular to modify the HTML before rendering it for display;
- The metadata for a service class provides the information Angular needs to make it available to components through dependency injection (DI).

An app's components typically define many views, arranged hierarchically. Angular provides the Router service to help define navigation paths among views. The router provides sophisticated in-browser navigational capabilities.

Chapter 4

Architecture design

In this section the requirement analysis is done starting from the business requirements: it is then followed by a cloud infrastructure comparison, with a final serverless adoption. At the end, cloud and database architectures are explained, giving motivation to the design choices, and APIs are defined.

4.1 Business requirements

The needs of this application are born after the adoption of a centralized system for authentication. There are distinct applications that after authentication of users needs a standardized way to manage roles and grant profiles: here is the need of a centralized proliferation of profiles. Without a centralized proliferation of profiles, the manage of profiles is passed to each application level and can be treated in different ways giving redundancy of functionalities and data, increasing the complexity and having more management logics for the same requirements. The creation and the management of fields for all the application or for a designed one aims to application level. Two level of usage are requested: globally where all the features are passed among all applications or locally for a single application. The user profile must be created from an administrator, not from the user himself, because of finance environment constraints. In this application Users and Applications have to be fully manageable; users are organized in hierarchies and a logic to deal with is requested. Moreover, each application needs a logic to deal with roles and functions: most of the time the latter can literally be the url path of an application website.

4.2 Requirement analysis

To satisfy customer needs a system of management is required: this system is called PASS. PASS is able to manage all users and applications attributes, grants and functionalities. It receives an username from a system of authentication, and then it allows an administrator or an application administrator to create the user profile adding its attributes of different kind respectively for a generic or for a selected application. PASS must satisfy those features:

- high reliability;
- security;
- scalability;
- low computational work;
- storing data.

The choice is to use a serverless solution.
With a serverless solution all the constraints are satisfied and there are no configurations to do besides writing the code to manage the data stored and logics. The other solutions are available but FaaS is the easiest and fastest to configure compared to the others; a lot of features are obtained, as explained in the previous chapter, without configuring other services.

4.3 Functional requirements

Business requirements can be translated in these requirements:

- creating, reading, updating and deleting the user;
- creating, reading, updating and deleting the user's attributes;
- creating, reading, updating and deleting the users hierarchy with a tree structure;
- creating, reading, updating and deleting the application;
- creating, reading, updating and deleting the application attributes;
- creating, reading, updating and deleting the application user profiles with a tree structure;

- creating, reading, updating and deleting the application functions with a tree structure;
- authentication for administrator and application administrator;

The user interface has to allow these features:

- login;
- users management;
- applications management;
- roles management;
- functions management;
- hierarchies management.

4.4 Cloud architecture

The proposed architecture is the following:

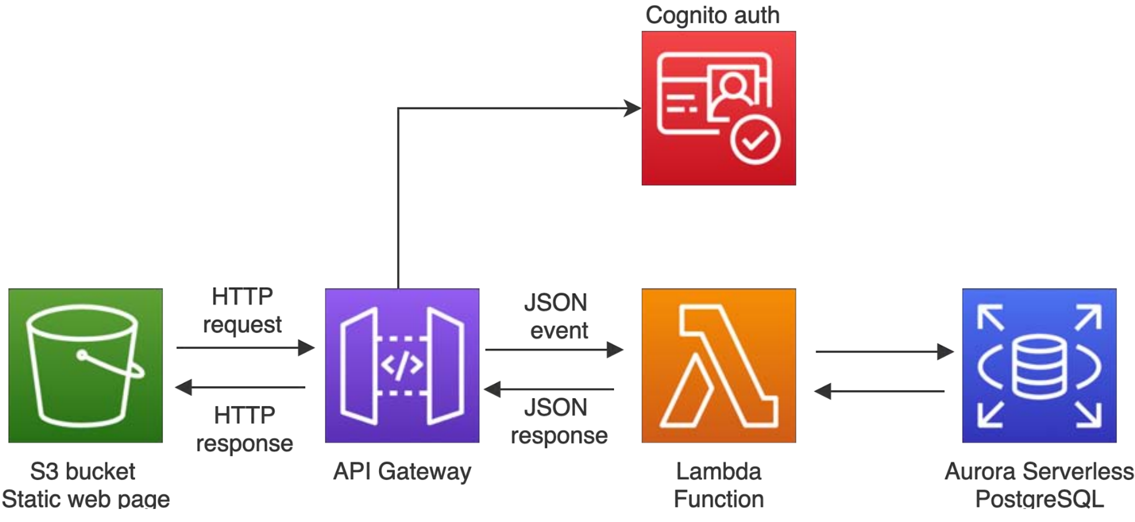

Figure 4.1: Cloud architecture

S3 bucket hosts the static web page, made in angular and publicly accessible. The administrator can authenticate in a login web page through an http request to API gateway that asks Cognito the authorization token, giving it back if the credentials are correct. After the login, the administrator can send his request using the user interface of the

web page. API gateway holds a set of APIs for handle all type of requests: each request passes through API gateway that checks cognito authentication token and lets a lambda be executed. Then, if necessary, lambda can access to the database and send back a JSON response to API gateway, which will pass to the web page that will render the response.
With this architecture the load of requests is satisfied thanks to Lambda, API Gateway and Aurora serverless features, as mentioned above. Therefore, to prevent the access from external networks, a Virtual Private Cloud (VPC) containing Lambda and Aurora: the only way to access to those services is through API gateway, which enables the requests with the authorization token of Cognito.
This architecture is declared in the Serverless Framework configuration file. The lambdas are coded in Node.js using Express and Sequelize.

4.5 Database design

The database proposed is Aurora Serververless Database with postgreSQL. Users and applications are the core entities of the database design; user attributes and application attributes are the generic ways to create custom fields to an user or an application. There is a redundancy of data because the information in User Attributes contain everything regarding an user's applications, so the same data can also be stored in ApplicationAttributes, ApplicationFunctions and ApplicationUserProfiles tables. In this way fast readings are satisfied with a redundancy cost.
For the UserHierarchies, ApplicationFunctions, and ApplicationUserProfiles table, which follow a tree structure, there are two ways of approach in sql databases: the Nested Set Model (NSM) or the adjacency list. With the adjacency list each node knows who is his parent, with a parent identification field on each record. In table 4.1 the pros and cons are compared and the adopted model is the NSM because PASS will query most of the time and rarely edit the trees. NSM is fully implemented in PASS.

-	**Adjacency list**	**Nested Set Model**
Pros	• Simple, easy to understand schema • Easy to query for children • Referential integrity • Easy to update	• Fast to query
Cons	• Deep trees cause problems • Difficult to query subtree • Performance issues	• More complicated • Expensive to update/delete/-move • No referential integrity

Table 4.1: Adjacency list, nested set model comparison

4.6 Entity relation schema

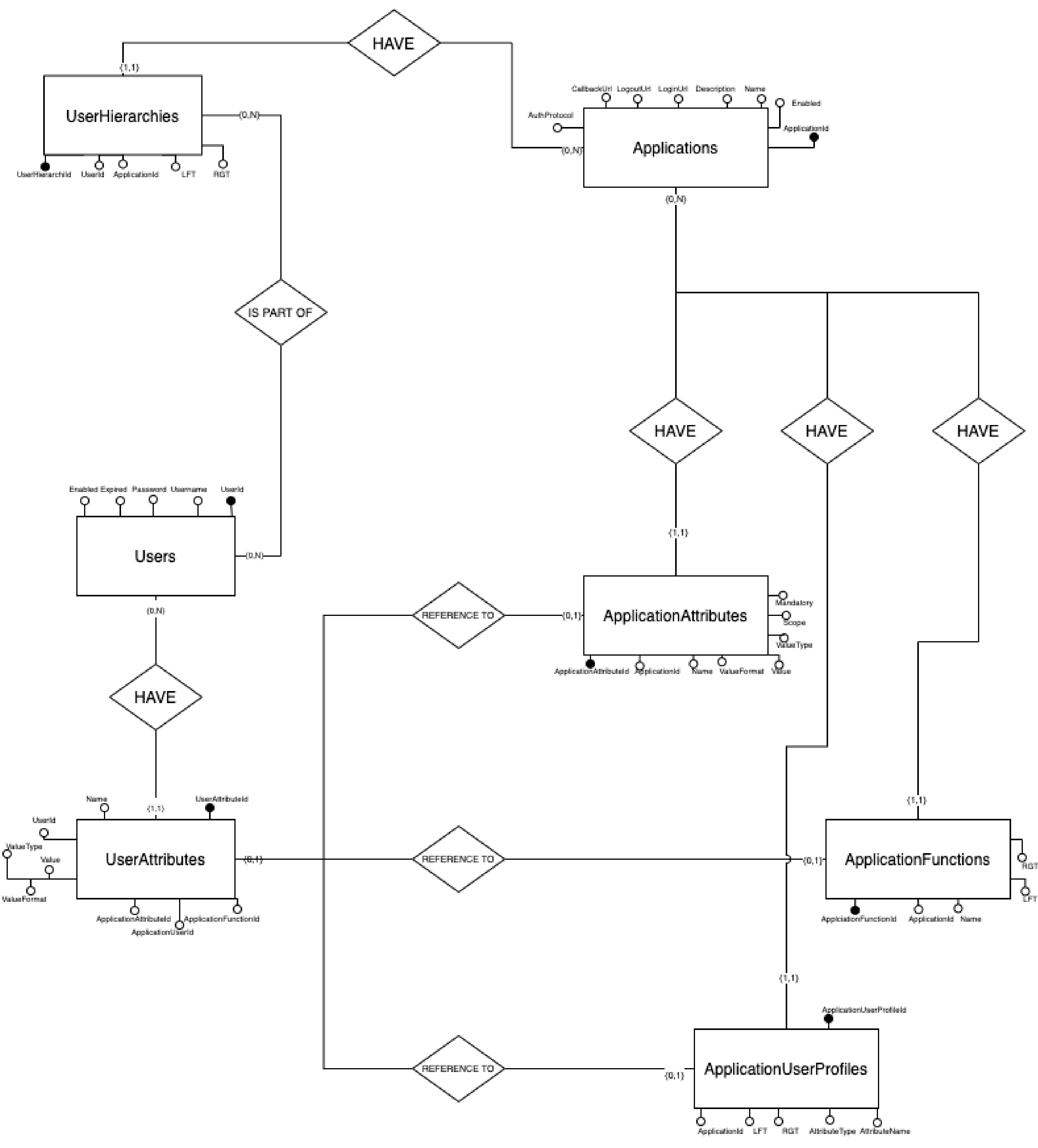

Figure 4.2: Entity relation schema

4.7 Entity table

Entity	**Description**	**Attributes**	**Identifier**
Users	Represents an user of the system with basic details.	• UserId, serial • Username, text • Password, hashed text • Enable, boolean • Expired, datetime	UserId
Applications	Represents an Application with basic details.	• ApplicationId, serial. • Name, text • Enabled, boolean • Description, text • LoginUrl, text • LogoutUrl, text • CallbackUrl, text • AuthProtocol, integer	ApplicationId

UserHierarchies	The hierarchy of an application. Tree structure of users.	• UserHierarchieId, serial • UserId, foreign key • ApplicationId, foreign key • LFT,left indicator of NSM,integer. • RGT, right indicator of NSM, integer	UserHiearchieId
UserAttributes	The attributes of an user. They can refer to the user, to the application, to the functions or to user profiles.	• UserAttributesId, serial • UserId, foreign key • Name, text • ValueType, integer • Value, text • ValueFormat, text • ApplicationAttributeId, foreign key • ApplicationUserId, foreign key • ApplicationFunctionId, foreign key	UserAttributesId

ApplicationAttributes	The attributes of an application.	- ApplicationAttributeId, serial - ApplicationId, foreign key - Name, text - Value, tet - ValueType, integer - ValueFormat, text - Scope, number - Mandatory, boolean	ApplicationAttribute
ApplicationUserProfiles	The application roles and grants, tree structured.	- ApplicationUserProfileId, serial - ApplicationId, foreign key - AttributeName , text - AttributeType , integer - LFT,left indicator of NSM,integer. - RGT, right indicator of NSM, integer	ApplicationUserProfi

ApplicationFunctions	The function of the application, with a tree structure.	- ApplicationFunctionId, serial - ApplicationId, foreign key - Name, text - LFT, left indicator of NSM,integer. - RGT, right indicator of NSM, integer	ApplicationFunctionI

4.8 Relationship table

Relationship	Description	Component Entities
Have	user has user attributes	Users (0,N) - UserAttributes (1,1)
Is Part of	user is part of an user hierarchy	Users (0,N) - UserHierarchies (1,1)
Have	application has user hierarchies	Applications (0,N) - UserHierarchies (1,1)
Have	application has application attributes	Applications (0,N) - ApplicationAttributes(1,1)
Have	application has functions	Applications (0,N) - ApplicationFunctions(1,1)
Have	application has user profiles	Applications (0,N) - ApplicationUserProfiles(1,1)
Reference to	user attribute referred to a function	UserAttributes (0,1) - ApplicationFunctions (0,1)
Reference to	user attribute referred to a user profile	UserAttributes (0,1) - ApplicationUserProfiles (0,1)
Reference to	user attribute referred to a application attribute	UserAttributes (0,1) - ApplicationAttributes (0,1)

Table 4.3: Relationship table

4.9 API design

In this part APIs are defined. Each API has a description, a calling name, that follows the REST architecture. APIs are of different type (GET PUT, POST, DELETE). The input and the output column are both JSON formatted, and in the header of the request there is always the authentication token (except for the login request). The User/Id resource is the most complex one because behind it there are roles, functions and application attributes to deal with.

Table 4.4: API definitions table

Description	Type	Resource Name	Input	Output
Gather all users information	GET	User		All users data
Edit a group of users	PUT	User	User ids with the respective edits	Response message
Create one new user	POST	User	New user's information	Response message
Delete a group of users	DELETE	User	user ids	Response message
Gather an user information	GET	User/{Id}		The designed user data
Edit fields of an user	PUT	User/{Id}	The fields edited	Response message
Delete of user	DELETE	User/{Id}		Response message
Gather all applications information	GET	Application		All applications data
Edit a group of applications	PUT	Application	Applications ids with the respective edits	Response message
Create one new user	POST	Application	New application's information	Response message
Delete a group of application	DELETE	Application	application ids	Response message
Gather an application's datas	GET	Application/{Id}		The designed application data
Edit fields of an application	PUT	Application/{Id}	The fields edited	Response message
Delete an application	DELETE	Application/{Id}		Response message
Gather the function tree	GET	Function/{Id}		The tree data

Edit the function tree	PUT	Function/{Id}	The entire tree structure	Response message
Gather the user hierarchy tree	GET	Hierarchy/{Id}		The tree data
Edit the user hierarchy tree	PUT	Hierarchy/{Id}	The entire tree structure	Response message
Gather the roles tree	GET	Role/{Id}		The tree data
Edit the roles tree	PUT	Role/{Id}	The entire tree structure	Response message

Chapter 5

PASS demo

Practical implementation will be explained in this chapter. Due to secrecy constraints, codes and data are partially or totally hidden. The backend logic will be explained in the first part of the chapter, while in the second there will be the presentation and the explanation of some views of the front-end.

5.1 Back-end

As explained in the architecture design, the back-end core is AWS Lambda. In order to give consistency and easy-to-reproduce infrastructure as well as having benefits on developing and testing, Serverless Framework is adopted.
Serverless framework is used for the following functions:

- deploy in the cloud with CLI;
- local testing;
- declaring lambda functions;
- declaring http event triggers for lambdas;
- declaring VPC;
- declaring cognito auth pool;
- declaring S3 bucket;
- declaring Aurora Serverless DB configuration.

The language used is Node.js: this language has been chosen because Lambda fully supports it. Moreover, Serverless Framework is coded in Node and there is not so much

difference from javascript, used for the front-end. As explained in the introduction chapter, there can be lambdas with different languages in a single application, and this characteristic is really powerful in increasing the development speed, because a copy and paste action of a lambda well coded in any language from another developer can shortcut to the result easily.
The asynchronous language constructs used in Node are async await but since Sequelize is a promise-based ORM there are also promises handled with it.
To manage postgrSQL database, Sequelize is used; all the database structure is declared following the Sequelize documentation, declaring each entity and each relation. What is more, the Nested Set is implemented to handle the tree structure entities as explained in the previous part of this document. Furthermore, conversions logics between NSM and Adjacency list model are implemented due to matching with the visualization format requirements.
Express routes every type of request (GET, PUT, POST, DELETE) to the right portion of code satisfying the lambdas formats with the use of http-serverless wrapper.
In this way, through the use of API Gateway and Lambda, a fully API back-end is easy implemented. API Gateway also has a good versioning of all API, providing a good way of documentation of it.
The good of this back-end approach is that is fully independent from a front-end application, giving a possibility of using those APIs with a different front-end.

5.2 Front-end

The front-end is coded in the Angular framework, with typescript, javascript, css and html. In this section some of the views are explained so as to provide a better understanding of PASS application. It has to be remarked that all following data are clearly fake.
PASS is a single web page application, with an administrator page graphic style based on this github project `https://github.com/akveo/ngx-admin/` that uses Nebular UI library. After a login form, the page view looks like the following. On the left part there is a collapsible navigation sidebar whose central component changes when another page is chosen. Figure 5.1: on the header bar there is a logout button and an application drop menu, useful to choose an application and see the users, roles, functions, hierarchy inheritance to the selected one. There is also the PASS logo and the name of the administrator logged in. Figure 5.2: the application component is a table that allows the end user to see applications, add new ones, see details or delete one. There is a checkbox in front of each

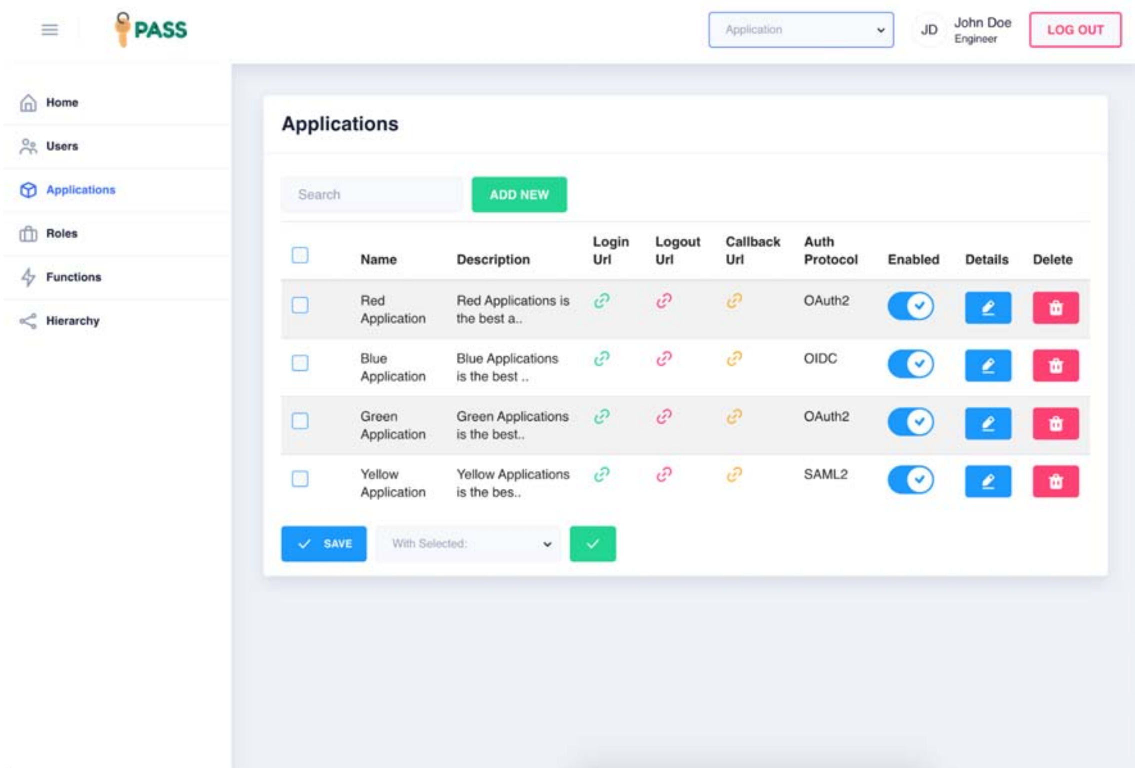

Figure 5.1: Front-end - general view

application row that can be used to simultaneously delete or disable/enable more than one application.

Figure 5.3: when the “add new” button is clicked from the application component, a form dialog component appears and following the angular form logic the user can be helped to insert the right format of each field: when the form respects all fields rules, the “add” button will be enabled from the disabled status.

The save button gathers all the modifications and calls the angular service: following the method explained in the architecture design and back-end sections, it uses APIs to access the database.

Figure 5.4: when the delete button is clicked from the application component, a dialog appears to have the confirmation of the action; when more applications get selected, the delete dialog shows the names that are going to be deleted.

Figure 5.5: when the details button is selected, from the application component, the central component shows details of an application. When using this component it is possible for the administrator to delete or update each field or to manage the application attributes. A brief consideration on the Red application: it has a “Full Name” attribute with scope User; it is also mandatory, which means that in this case if an user is created in this ap-

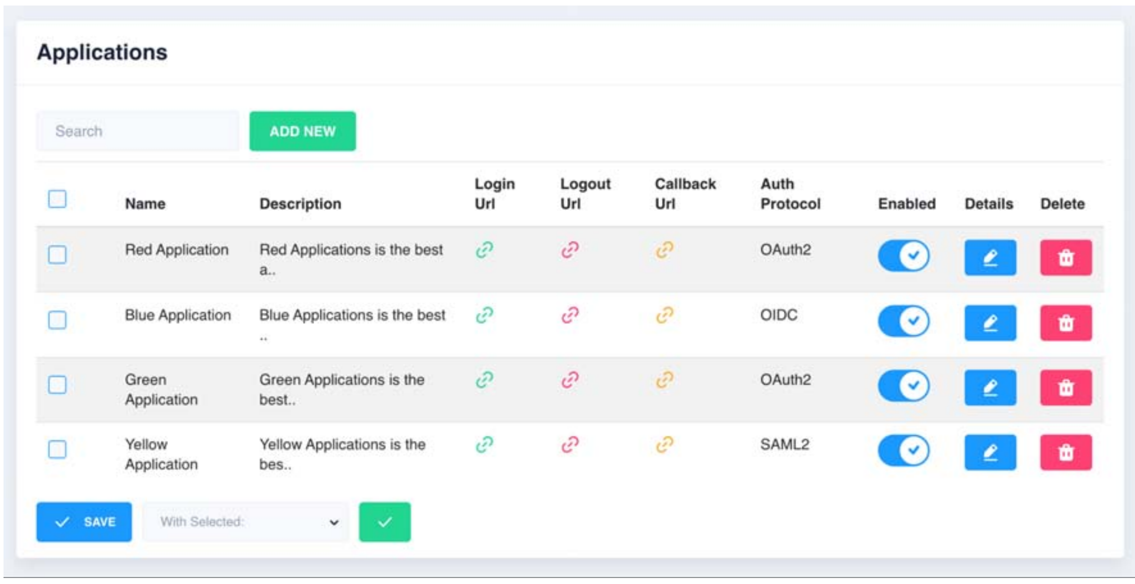

Figure 5.2: Front-end - application component view

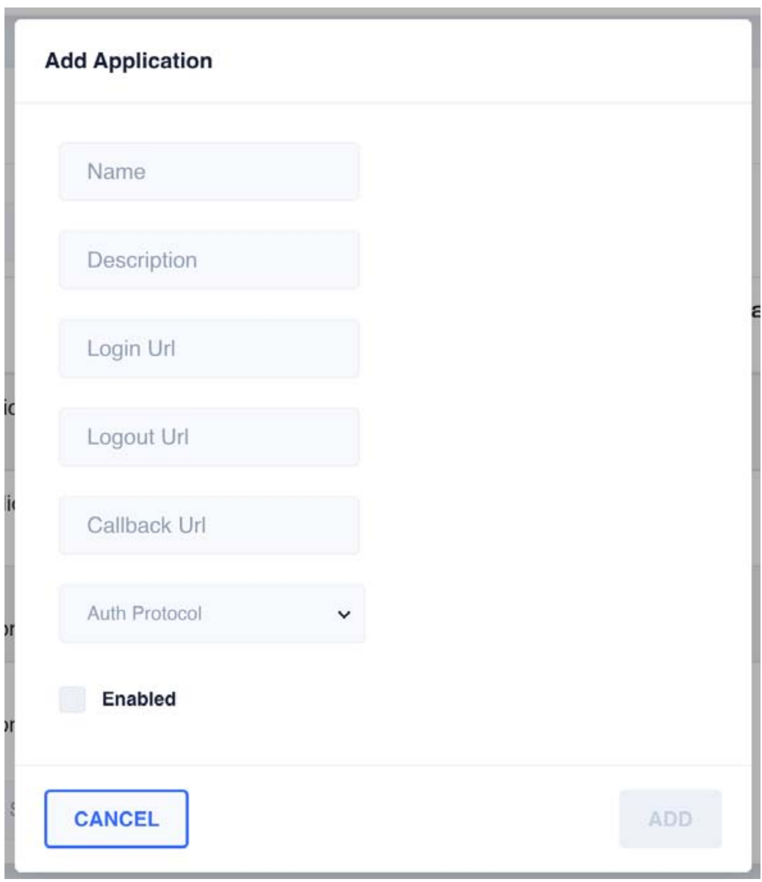

Figure 5.3: Front-end - application add form view

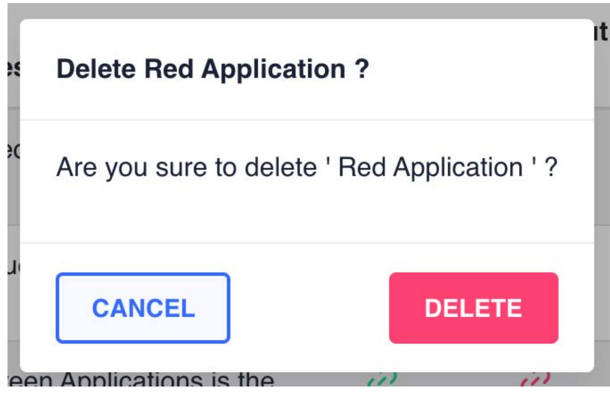

Figure 5.4: Front-end - application delete form view

plication it will have the user attribute required.

The figure 5.6: illustrates the user component; the management of the users is similar to the application.

Figure 5.7: when the details button is clicked, from the user component, there is a page, as the application detail component, allowing the administrator to manage the user profile. In the profile section there are the user attributes without reference. In the application section there are the attributes referred to an application, in this case it is supposed to be in the red app administration, so there is the full name field. In the roles and functions sections there are two checkbox trees that allow to gives the right roles, grants and permission to access the functionalities respectively.

Figure 5.8: this is the interface of management of roles, it is the same for hierarchy and functions, it's an intuitive user interface to manage a tree.
This tree is fully editable, it has drag and drop feature to move tree nodes, the green blue and red button allow to add a children to the selected node, edit the name of the selected node, delete the selected node respectively. The tree can be restored or be permanently saved with the two buttons in the left. When save button is clicked an angular service is called and the right API is called following the PASS back-end logic.

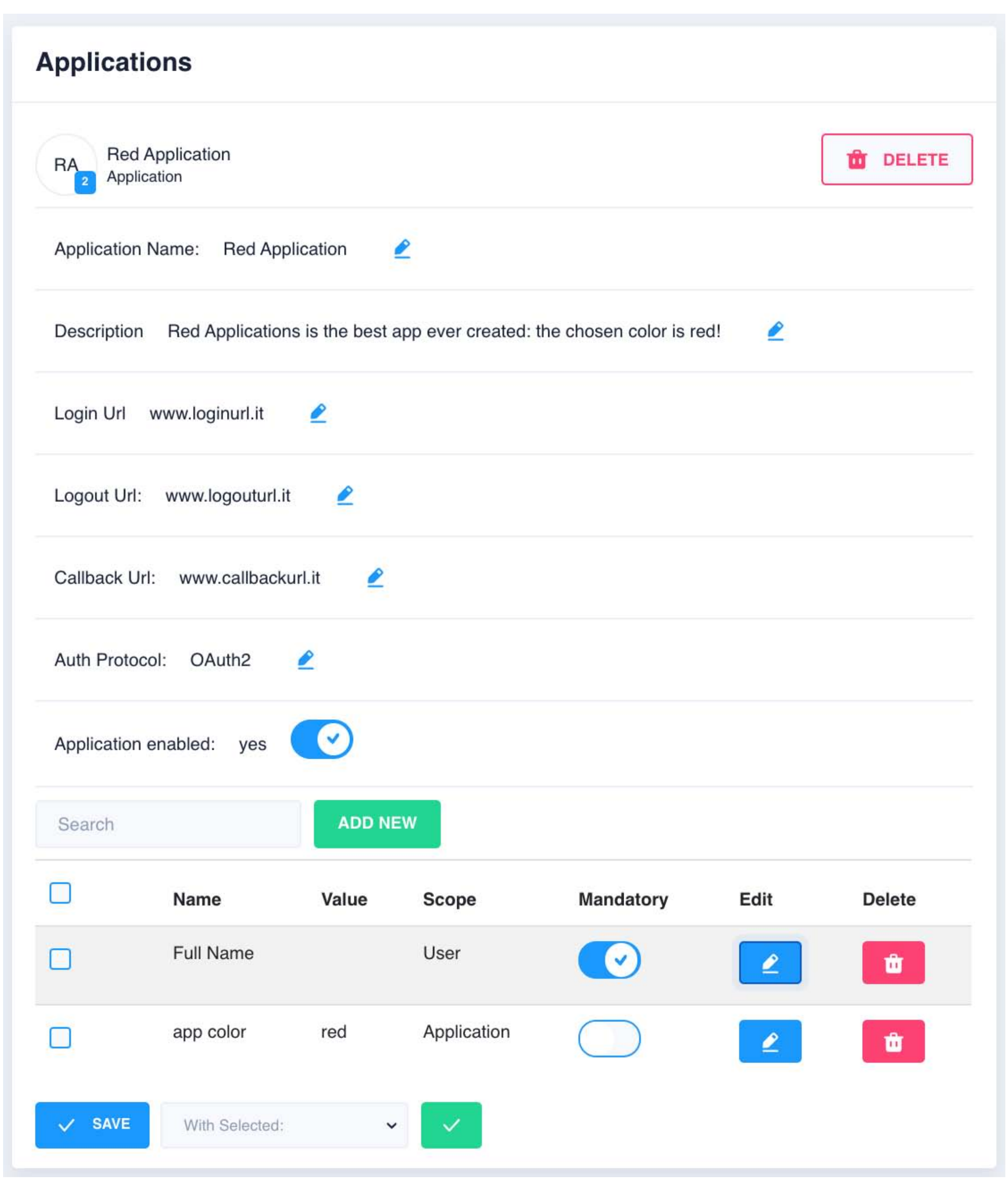

Figure 5.5: Front-end - application detail view

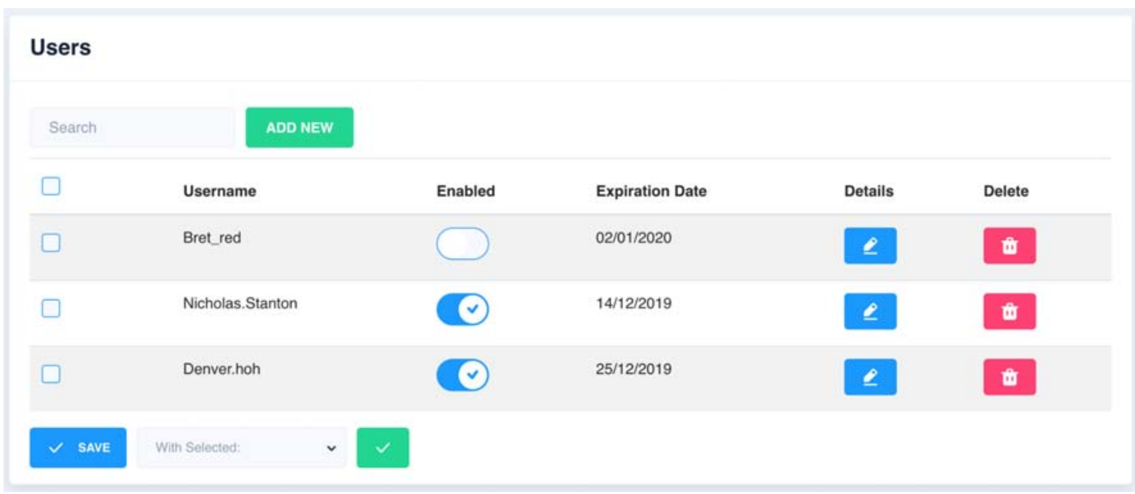

Figure 5.6: Front-end - user component view

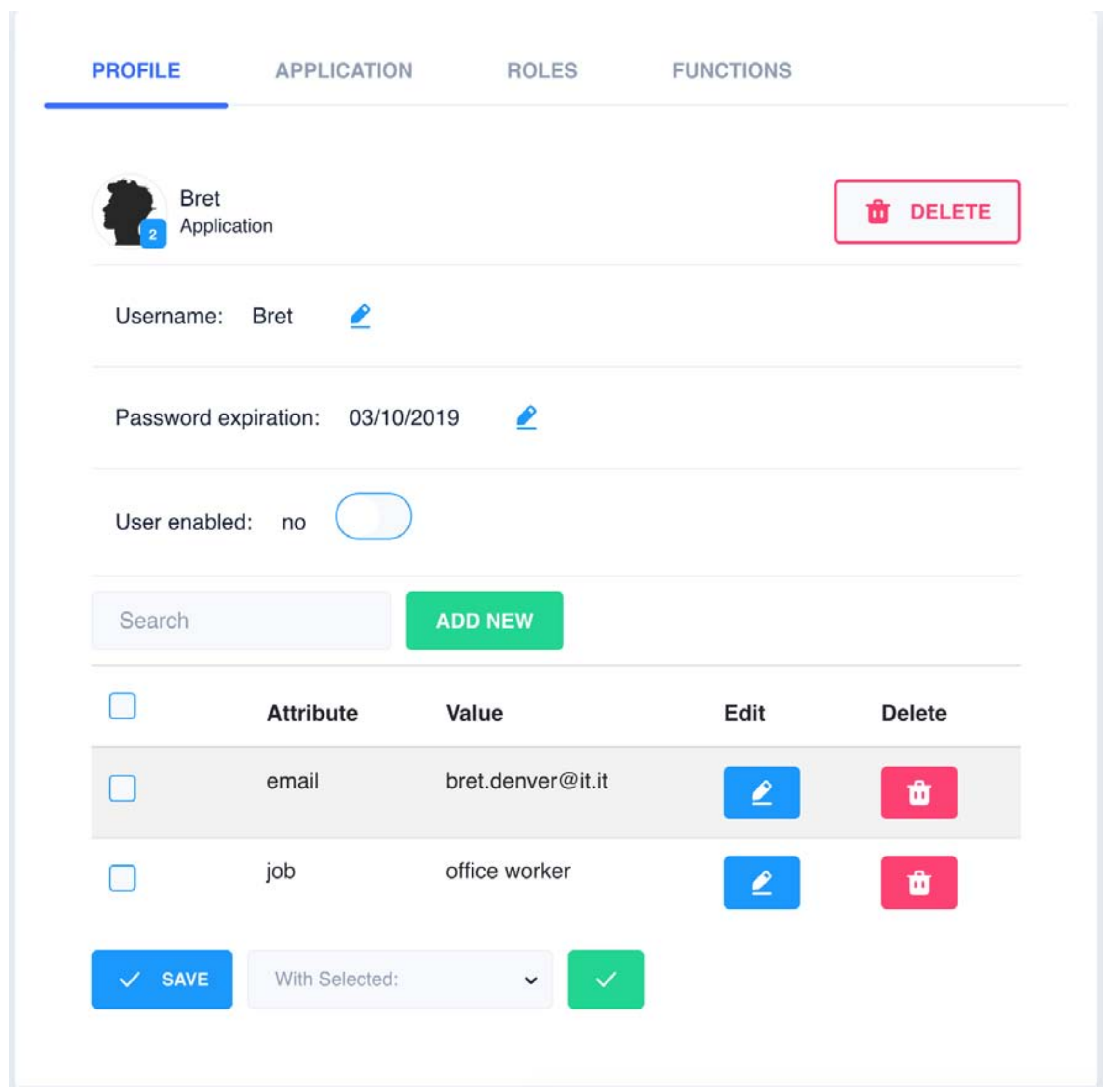

Figure 5.7: Front-end - user detail view

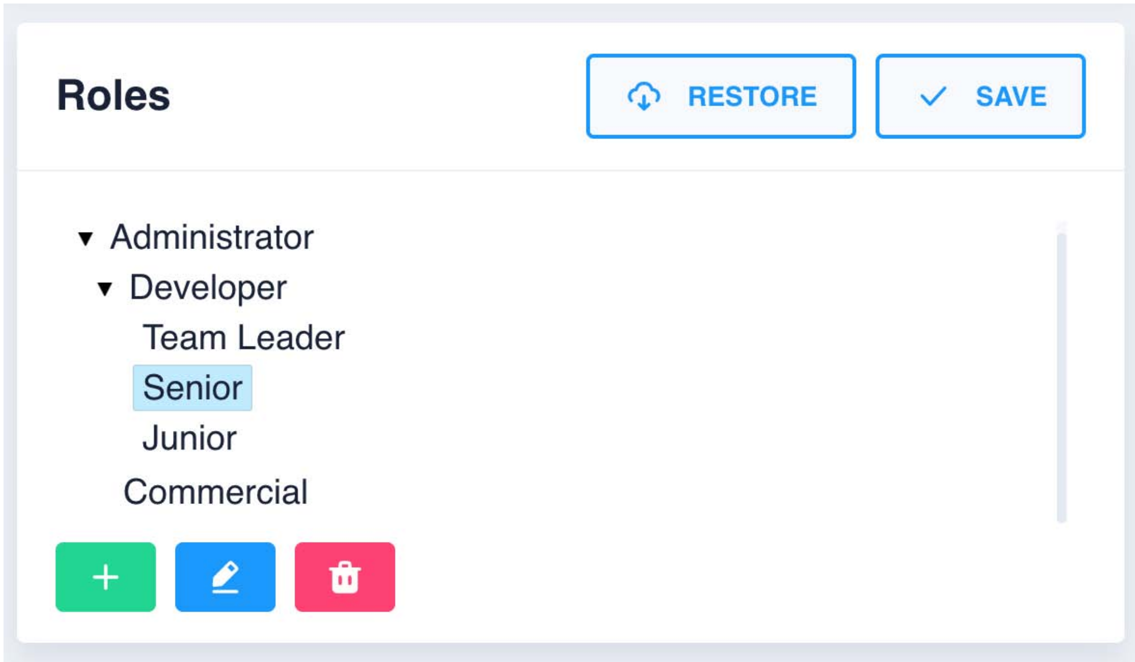

Figure 5.8: Front-end - roles component view

Chapter 6

Conclusion

In the following chapter, final considerations will be presented, providing an overall insight on the internship experience and on the PASS application.

6.1 Internship experience

In this four months internship at SCAI ITEC the student had the possibility of facing a real working project in the financial environment. The technology and services used are many, and the skills achieved in the cloud are consistent enough in order to work with it in the future. Starting objectives are fully satisfied and the working experiences helped reaching the right fundamentals on a working environment.
During the internship the student had the possibility to share with the colleagues all the milestone reached in serverless environment, facing new considerations and questions; he also attended physical or streamed events to meet more serverless and AWS cloud developers, who suggested tips and shortcuts useful to increase knowledge.
The PASS web application is ready to go in the testing and finding issues part and then in the production section.
The PASS needs are satisfied allowing administrators to create and manage standard roles, functions and hierarchies of a group of users, obviously managing users profiles and applications profiles. With the features of a serverless backend a low cost, implementation is done with a pay per use billing; moreover, no maintenance is needed.
Features of scalability, high reliability, security, world coverage and storing of big data are satisfied giving responsibilities partially or totally to the Amazon cloud. What is more, the entire work is easy to reproduce in another AWS account, and with the API structure

of back-end more functionalities can be easily added in the future, taking advantage of the modularity of the system and the multi-language support.

Regarding the structure of the database, a better fit solution would have been with a noSQL database, but due the constraints only a postgreSQL has been adopted.

Future works could be the development of a interface for monitoring the profiles access, the implementation of exporting data in PDF or CSV format and implement a notification system.

Appendix A

Lambda insights

A.0.1 Permissions

To manage access to the Lambda API and resources like functions and layers, AWS Identity and Access Management (IAM) is used. In order to manage permissions, the developer can apply permission policies to IAM users, groups, or roles. To grant permissions to other accounts or AWS services that use a Lambda resources, a policy can be made so that it applies to the resource itself.
A Lambda function also has a policy, called an execution role, that grants it permission to access AWS services and resources. Function configuration A Lambda function consists of code and any associated dependencies. In addition, a Lambda function also has configuration information associated with it.
Function settings:

- Code – The code and dependencies of the function. For scripting languages, the function code can be edited in the embedded editor. To add libraries, or for languages that the editor doesn't support, a deployment package (S3) can be uploaded;
- Runtime – The Lambda runtime that executes the function;
- Handler – The method that the runtime executes when the function is invoked, such as index.handler. The first value is the name of the file or module, and the second is the name of the method;
- Environment variables – Key-value pairs that Lambda sets in the execution environment. Environment variables can be used to extend the function's configuration outside of code;

- Tags – Key-value pairs that Lambda attaches to the function resource. Tags apply to the entire function, including all versions and aliases;
- Execution role – The IAM role that AWS Lambda assumes when it executes the function;
- Description – A description of the function;
- Memory – The amount of memory available to the function during execution. An amount between 128 MB and 3,008 MB in 64 MB increments can be chosen; Lambda allocates CPU power linearly in proportion to the amount of memory configured.
- Timeout – The amount of time that Lambda allows a function to run before stopping it. The default is 3 seconds. The maximum allowed value is 900 seconds;
- Virtual private cloud (VPC) – If the function needs network access to resources that are not available over the internet;
- Dead letter queue (DLQ) – If the function is invoked asynchronously, choose a queue or topic to receive failed invocations;
- Active tracing – Sample incoming requests and trace sampled requests with AWS X-Ray;
- Concurrency – Reserve concurrency for a function to set the maximum number of simultaneous executions for a function, and reserves capacity for that concurrency level. Reserved concurrency applies to the entire function, including all versions and aliases.

Function settings can only be changed on the unpublished version of a function. When a version is published, code and most settings are locked to ensure a consistent experience for users of that version.

A.0.2 Invoking Functions

Synchronous invocations are the most straightforward way to invoke Lambda functions. In this model, functions execute immediately when a the Lambda is Invoked through API call. This can be accomplished through a variety of options, including using the CLI or any of the supported Software Development Kit (SDK).
Here is an example of a synchronous invoke using the CLI:

```
$ aws lambda invoke -function-name MyLambdaFunction -invocation-type
RequestResponse -payload  ‘‘[JSON string here]’’
```

The Invocation-type flag specifies a value of "RequestResponse". This instructs AWS to execute the Lambda function and wait for the function to complete.
Many AWS services can emit events that trigger Lambda functions. Here is a list of services that invoke Lambda functions synchronously:

- Elastic Load Balancing (Application Load Balancer);
- Amazon Cognito;
- Amazon Lex;
- Amazon Alexa;
- Amazon API Gateway;
- Amazon CloudFront (Lambda@Edge);
- Amazon Kinesis Data Firehose;
- Asynchronous Invokes.

Here is an example of an **asynchronous invoke** using the CLI:

```
$ aws lambda invoke -function-name MyLambdaFunction -invocation-type
Event -payload  ‘‘[JSON string here]’’
```

Notice that the Invocation-type flag specifies "Event": if the function returns an error, AWS will automatically retry the invoke twice, for a total of three invocations.
Here is a list of services that invoke Lambda functions asynchronously:

- Amazon Simple Storage Service;
- Amazon Simple Notification Service;
- Amazon Simple Email Service;
- AWS CloudFormation;
- Amazon CloudWatch Logs;
- Amazon CloudWatch Events;
- AWS CodeCommit;

- AWS Config.

Asynchronous invokes place an invoked request in Lambda service queue and AWS processes the requests as they arrive. **Poll-Based Invokes**
This invocation model is designed to allow the integration with AWS Stream and Queue based services with no code or server management. Lambda will poll the following services retrieve records, and invoke functions. The following are supported services:

- Amazon Kinesis;
- Amazon SQS;
- Amazon DynamoDB Streams.

AWS will manage the poller and perform Synchronous invokes of the function with this type of integration. The retry behavior for this model is based on data expiration in the data source.

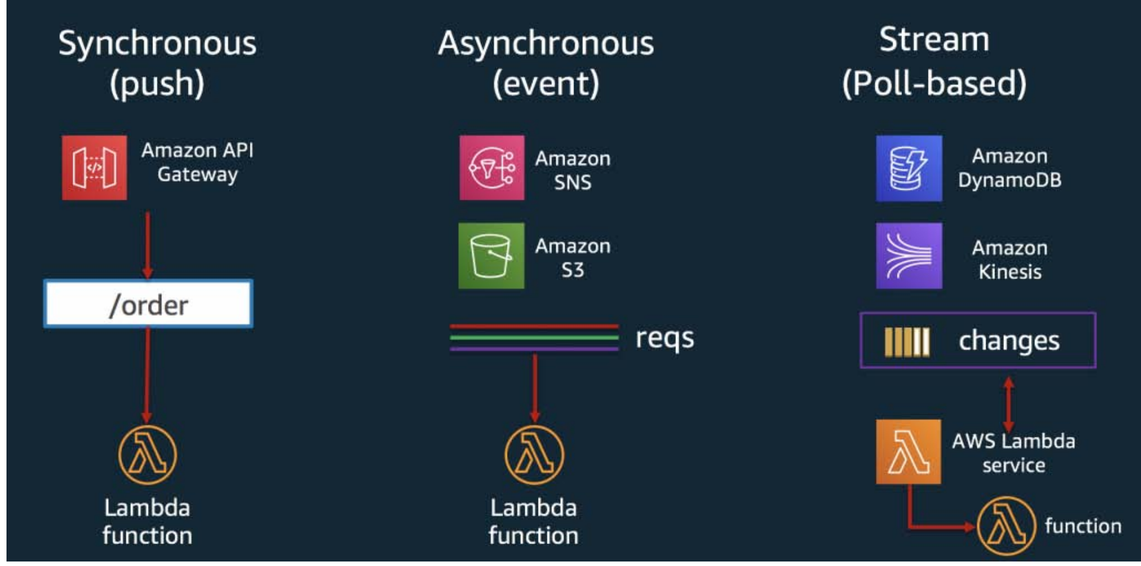

Figure A.1: Synchronous, Asynchronous and Stream invocations for AWS Lambda

A.0.3 Error handling and automatic retries

When a function is invoked, two types of error can occur. Invocation errors occur when the invocation request is rejected before the function receives it. Function errors occur when the function's code or runtime returns an error. Depending on the type of error, the type of invocation, and the client or service that invokes the function, the retry behavior and the strategy for managing errors varies.

Issues with the request, caller, or account can cause invocation errors. Invocation errors include an error type and status code in the response that indicate the cause of the error. Common invocation errors:

- Request - The request event is too large or isn't valid JSON, the function doesn't exist, or a parameter value is the wrong type;
- Caller - The user or service doesn't have permission to invoke the function;
- Account - The maximum number of function instances are already running, or requests are being made too quickly.

Function errors occur when the function code or the runtime that it uses return an error. Common function errors:

- Function - function's code throws an exception or returns an error object;
- Runtime - The runtime terminated the function because it ran out of time, detected a syntax error, or failed to put the response object into JSON. The function exited with an error code.

Unlike invocation errors, function errors don't cause Lambda to return a 400-series or 500-series status code. If the function returns an error, Lambda indicates this by including a header named X-Amz-Function-Error, and a JSON-formatted response with the error message and other details.
When a function is invoked directly, the strategy for handling errors can be chosen. It's possible to retry, send the event to a queue for debugging, or ignore the error. The function's code might have run completely, partially, or not at all. If retry, ensure that the function's code can handle the same event multiple times without causing duplicate transactions or other unwanted side effects.
When a function is invoked indirectly, need to be aware of the retry behavior of the invoker and any service that the request encounters along the way.
To help dealing with errors in Lambda applications, Lambda integrates with services like Amazon CloudWatch and AWS X-Ray.

A.0.4 Function Scaling

The first time a function is invoked, AWS Lambda creates an instance of the function and runs its handler method to process the event. When the function returns a response, it

sticks around to process additional events. If the function is invoked again while the first event is being processed, Lambda creates another instance.

As more events come in, Lambda routes them to available instances and creates new instances as needed. The function's concurrency is the number of instances serving requests at a given time. For an initial burst of traffic, the function's concurrency can reach an initial level of between 500 and 3000, which varies per Region. Initial concurrency burst limits

- 3000 - US West (Oregon), US East (N. Virginia), EU (Ireland);
- 1000 - Asia Pacific (Tokyo), EU (Frankfurt);
- 500 - Other Regions.

After the initial burst, the function's concurrency can scale by an additional 500 instances each minute. This continues until there are enough instances to serve all requests, or a concurrency limit is reached. When the number of requests decreases, Lambda stops unused instances to free up scaling capacity for other functions.
When requests come in faster than the function can scale, or when the function is at maximum concurrency, additional requests fail with a throttling error.

A.0.5 Handler in Node.js

The handler is the method in a Lambda function that processes events. When a function is invoked, the runtime runs the handler method. When the handler exits or returns a response, it becomes available to handle another event.
When configuring a function, the value of the handler setting is the file name and the name of the exported handler module, separated by a dot. The default in the console and for examples in this guide is index.handler. This indicates the handler module that's exported by index.js.
The runtime passes three arguments to the handler method. The first argument is the event object, which contains information from the invoker. The invoker passes this information as a JSON-formatted string when it calls Invoke, and the runtime converts it to an object. When an AWS service invokes a function, the event structure varies by service.
The second argument is the context object [X], which contains information about the invocation, function, and execution environment.
Context methods:

- getRemainingTimeInMillis() – Returns the number of milliseconds left before the execution times out.

Context properties:

- functionName – The name of the Lambda function;
- functionVersion – The version of the function;
- invokedFunctionArn – The Amazon Resource Name (ARN) that's used to invoke the function; Indicates if the invoker specified a version number or alias;
- memoryLimitInMB – The amount of memory that's allocated for the function;
- awsRequestId – The identifier of the invocation request;
- logGroupName – The log group for the function;
- logStreamName – The log stream for the function instance;
- identity – (mobile apps) Information about the Amazon Cognito identity that authorized the request;
- clientContext – (mobile apps) Client context that's provided to Lambda by the client application;
- Custom – Custom values that are set by the mobile application;
- callbackWaitsForEmptyEventLoop – Set to false to send the response right away when the callback executes, instead of waiting for the Node.js event loop to be empty. If this is false, any outstanding events continue to run during the next invocation.

The third argument, callback, is a function that can be called in non-async functions to send a response. The callback function takes two arguments: an Error and a response.
For async functions, return a response, error, or promise to the runtime instead of using callback.

A.0.6 Logging in Node.js

A Lambda function comes with a CloudWatch Logs log group, with a log stream for each instance of the function. The runtime sends details about each invocation to the log stream, and relays logs and other output from the function's code.
To output logs from the function code, methods on the console object, or any logging library that writes to stdout or stderr can be used. In figure A.2 and A.3 there are example logs the values of environment variables and the event object.

```
exports.handler = async function(event, context) {
  console.log("ENVIRONMENT VARIABLES\n" + JSON.stringify(process.env, null, 2))
  console.info("EVENT\n" + JSON.stringify(event, null, 2))
  console.warn("Event not processed.")
  return context.logStreamName
}
```

Figure A.2: Example index.js file logging in AWS Lambda

```
START RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac Version: $LATEST
2019-06-07T19:11:20.562Z    c793869b-ee49-115b-a5b6-4fd21e8dedac    INFO    ENVIRONMENT
VARIABLES
{
  "AWS_LAMBDA_FUNCTION_VERSION": "$LATEST",
  "AWS_LAMBDA_LOG_GROUP_NAME": "/aws/lambda/my-function",
  "AWS_LAMBDA_LOG_STREAM_NAME": "2019/06/07/[$LATEST]e6f4a0c4241adcd70c262d34c0bbc85c",
  "AWS_EXECUTION_ENV": "AWS_Lambda_nodejs12.x",
  "AWS_LAMBDA_FUNCTION_NAME": "my-function",
  "PATH": "/var/lang/bin:/usr/local/bin:/usr/bin/:/bin:/opt/bin",
  "NODE_PATH":
"/opt/nodejs/node10/node_modules:/opt/nodejs/node_modules:/var/runtime/node_modules",
  ...
}
2019-06-07T19:11:20.563Z    c793869b-ee49-115b-a5b6-4fd21e8dedac    INFO    EVENT
{
  "key": "value"
}
2019-06-07T19:11:20.564Z    c793869b-ee49-115b-a5b6-4fd21e8dedac    WARN    Event not processed.
END RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac
REPORT RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac  Duration: 128.83 ms Billed Duration: 200
ms Memory Size: 128 MB Max Memory Used: 74 MB  Init Duration: 166.62 ms    XRAY TraceId: 1-
5d9d007f-0a8c7fd02xmpl480aed55ef0   SegmentId: 3d752xmpl1bbe37e Sampled: true
```

Figure A.3: Example log format in AWS Lambda

The Node.js runtime logs the START, END, and REPORT lines for each invocation. It adds a timestamp, request ID, and log level to each entry logged by the function. The report line provides the following details.

- RequestId – The unique request ID for the invocation;
- Duration – The amount of time that the function's handler method spent processing the event;
- Billed Duration – The amount of time billed for the invocation;
- Memory Size – The amount of memory allocated to the function;

- Max Memory Used – The amount of memory used by the function;
- Init Duration – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method;
- XRAY TraceId – For traced requests, the AWS X-Ray trace ID;
- SegmentId – For traced requests, the X-Ray segment ID;
- Sampled – For traced requests, the sampling result.

Logs in the Lambda console can be viewed in the CloudWatch Logs console, or from the command line.

A.0.7 Errors in Node.js

When code raises an error, Lambda generates a JSON representation of the error. This error document appears in the invocation log and, for synchronous invocations, in the output.

```
exports.handler = async function() {
  return x + 10
}
```

Figure A.4: Example index.js file

```
{
  "errorType": "ReferenceError",
  "errorMessage": "x is not defined",
  "trace": [
    "ReferenceError: x is not defined",
    "    at Runtime.exports.handler (/var/task/index.js:2:3)",
    "    at Runtime.handleOnce (/var/runtime/Runtime.js:63:25)",
    "    at process._tickCallback (internal/process/next_tick.js:68:7)"
  ]
}
```

Figure A.5: Example reference error response JSON

The code in figure A.4, results in a reference error (figure A.5). Lambda catches the error and generates a JSON document with fields for the error message, the type, and the stack trace.

A.0.8 Monitoring and troubleshooting

AWS Lambda automatically monitors its functions and reports metrics through Amazon CloudWatch, figure A.6. To help monitoring code as it executes, Lambda automatically tracks the number of requests, the execution duration per request, and the number of requests that result in an error. It also publishes the associated CloudWatch metrics. These metrics can be used to set CloudWatch custom alarms.

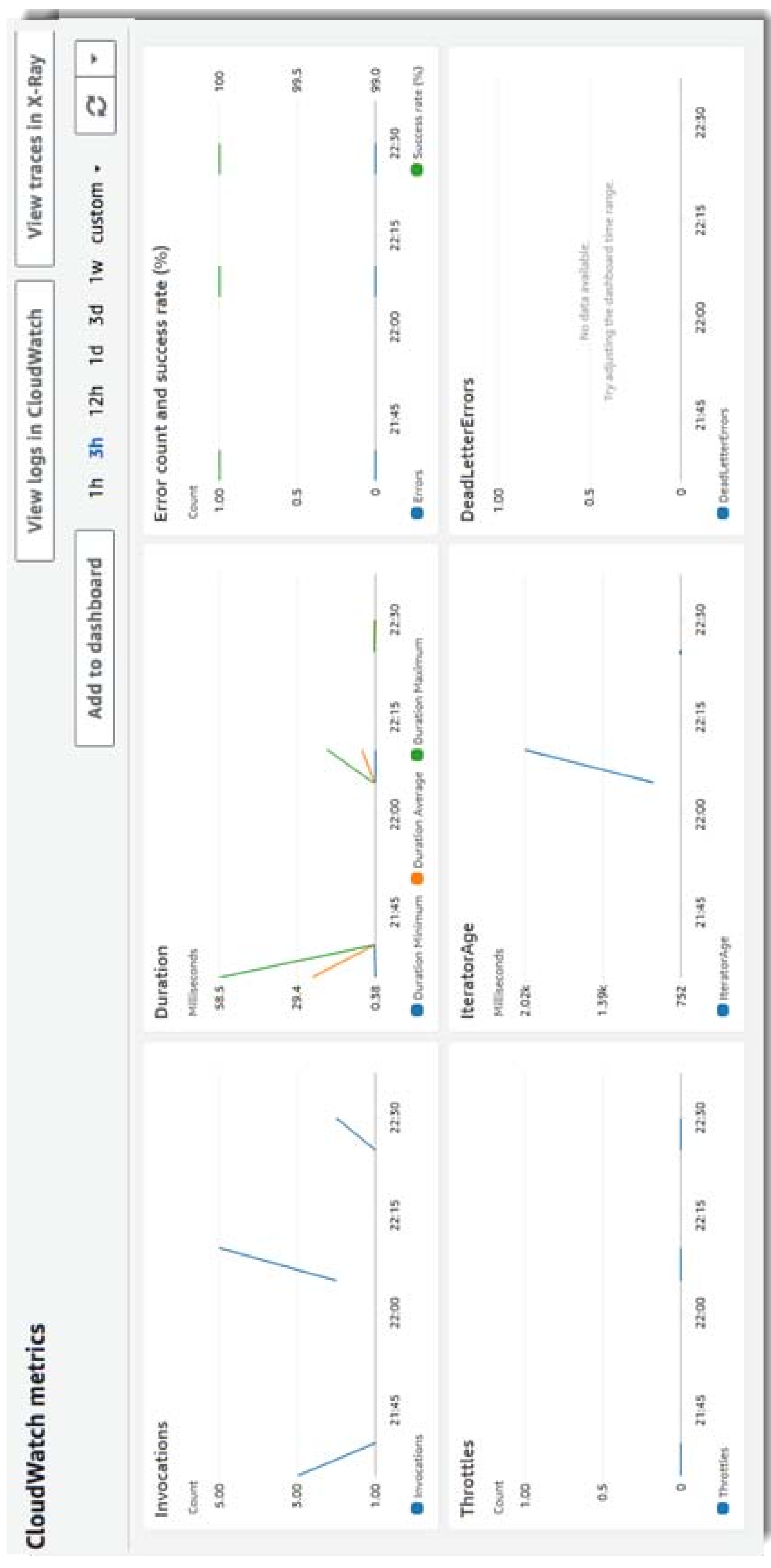

Figure A.6: CloudWatch metrics: view example

Bibliography

[1] Peter Mell and Timothy Grance. The nist definition of cloud computing (draft). *NIST special publication*, 800:145, 2011.

[2] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, May 2010.

[3] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. *Serverless Computing: Current Trends and Open Problems*, pages 1–20. Springer Singapore, Singapore, 2017.

[4] Dan Hazel. Using rational numbers to key nested sets. *CoRR*, abs/0806.3115, 2008.

[5] Margaret Rouse. Single sign-on (sso) definition, 2019. `https://searchsecurity.techtarget.com/definition/single-sign-on`.

[6] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, Irvine, California, 2000.

[7] typescriptlang.org. Typescript language official documentation. 2019. `https://www.typescriptlang.org/`.

[8] Nodejs.org. Node js language official documentation. 2019. `https://nodejs.org/en/docs/`.

[9] Amazon Web Services. Amazon web services official documentation. 2019. `https://docs.aws.amazon.com/`.

[10] Serveless Framework. Serverless framework official documentation. 2019. `https://serverless.com/framework/docs/`.

[11] Express js. Express official documentation. 2019. https://expressjs.com/it/api.html.

[12] Sequelize. Sequelize official documentation. 2019. https://sequelize.org/.