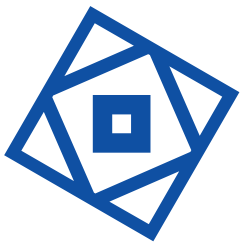


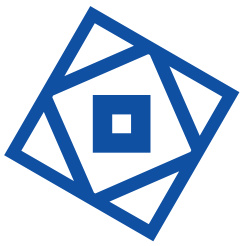
Transilvania
University
of Brasov

FACULTY OF MATHEMATICS
AND COMPUTER SCIENCE

DISSERTATION PAPER

Author: Kedves Szilard, engineer
Supervisor: Associate professor Sasu Lucian Mircea, Ph.D.

Brașov
July 2019



Transilvania
University
of Brasov

FACULTY OF MATHEMATICS
AND COMPUTER SCIENCE

DISSERTATION PAPER

Cloud framework for data streaming

Absolvent: Kedves Szilard, engineer
Coordonator: Associate professor Sasu Lucian Mircea, Ph.D.

Brașov
July 2019

Contents

Chapter 1

Introduction

1.1 Problem statement

In the world of Big Data, data visualization tools and technologies are essential to analyze massive amounts of information and make data-driven decisions. Hence it is not a surprise that applications that tackle this need are in great demand. These applications use graphical representation of information and data by using visual elements like charts and graphs.

1.2 Motivation

The idea started when I first came into contact with a major cloud service provider: Amazon Web Services also known as AWS.

AWS is a subsidiary of Amazon that offers reliable, scalable, and inexpensive cloud computing services. It provides on-demand cloud computing platforms to individuals, companies and governments, on a metered pay-as-you-go basis. In fact, these cloud computing web services provide a set of primitive, abstract technical infrastructure and distributed computing building blocks and tools. One of these services is Amazon Elastic Compute Cloud, which allows users to have at their disposal a virtual cluster of computers, available all the time, through the Internet. AWS's virtual machines emulate most of the attributes of a real computer including hardware (CPU(s) and GPU(s) for processing, local/RAM memory, hard-disk/SSD storage); a choice of operating systems; networking; and pre-loaded application software such as web servers, databases, etc. [1]

At the beginning its API seemed hard to comprehend, but with time I came to realize how easy it was in fact. Hence my determination to work with as many of its services as possible. At first I started with simple services like S3[TODO legend

for S3], but then I got to understand and use services like ECS[TODO legend for ECS]. The more services I used the more Stream Anlaysis began to expand.

1.3 Thesis structure

Chapter 2

Requirements

In this chapter the functional and nonfunctional requirements for the application are presented.

2.1 Functional requirements

2.1.1 Application overview

Stream Analysis is a web application that provides a hosting environment and defines a workflow for streaming applications, leveraged by cloud.

The hosting environment is responsible for user related sensitive information storage, while it paves the ground for an infrastructure that can handle incoming data. Once information pours in, a dashboard is shown at the user's disposal. This dashboard is updated each time new data arrives, which basically makes it a tool for analytical purposes. The user can visually check if a certain threshold was reached, exceeded, etc.

The workflow is predefined which makes it organized and coherent. Data is fetched or produced and packed by a custom application provided by an external user and funneled in through well defined channels. The stream of data arrives in a secure place where it waits to be queried. This place is located in the cloud so even if one can create the data on-premise, one still needs constant access to the internet. Based on the user's choice the packed information is then retrieved in a real-time fashion or stored. Both ways the data is then accessed and charted. 2D plots are shown with the values at the Y axis, where the X axis holds the timestamps.

The web application offers two services:

- Push data into cloud - users have to create their own packed software application that streams any type of data into a broker. The broker supports

5 types of protocols ([TODO]more information in chapter ...). Data is organized into topics and queues:

  - Topics - are used to handle real time messages
  - Queues - are used for historical data. Messages are stored and retrieved later

  Once the image is ready, the user can proceed with the step by step image upload and use it to create containers.

  1. Define topics and queues used in the container
  2. Create a repository for the image
  3. Push image to repository
  4. Add configuration for the image
  5. Run image immediately or create a scheduling rule to run the image multiple times over time

  Once containers are being created the web application subscribes to the topics defined by the user at step 1. The same happens with queues but in this case the stream of data is dequeued into files so that they can later be retrieved. Stream Analysis also offers the user a dashboard to keep track of his created containers and scheduling rules.

- Visualize data from the cloud - since data is split into topics and queues, the web application takes advantage of this and shows real time plots for topic streams and large scale based plots for queue data.

2.1.2 Access

The Stream Analysis web application is accessible from all around the world as long as the user has internet connection, however it is not open-source and its usage is not free.

The application is targeting two types of users:

- Advanced software engineers - they push data into the application. They need to register with an account and they are charged based on time.
- Normal users - they can read and interpret that data from charts in a dashboard. They also need to register with an account and they are charged based on the network traffic they generate during data visualization.

2.2 Nonfunctional requirements

2.2.1 Diversity

One of its main advantages consists in the ability to feed on any type of data. This asset makes it valuable in any kind of business or domain. For example in the field of meteorology users can plot the weather related data. Having Stream Analysis at their disposal they can track real-time the wind changes. In the field of economics, users can map economical growth of a country next to another one and be able compare, draw conclusions, or forecast.

2.2.2 Performance

The application itself was intended to be very fast and responsive to user interactions. However it has no scaling policies set up so there is a limit of users that can use the web application simultaneously. The limit is around 500 reqs/second.

2.2.3 Security

In my views security is pivotal for any well designed and successful application. Users are guaranteed to not have exposed credentials or any kind of sensitive data. If there is a concern of information leakage, this matter is more deeply discussed in [TO DO Chapter ...].

2.2.4 User's Guide

This paper will softly guide users through its features in further chapters. At this moment it has no stand alone public documentation that users can access.

Chapter 3

Framework architecture

At the highest macro level the framework can be put into perspective as shown in figure 3.1.

Figure 3.1: Highest level of architecture view

The roles and components are:

- Data acquisition users - these are external users that get data from any source and have the responsibility to wrap the fetching into an application and use the framework to upload it
- Packed application - the application made on-premise by the data acquisition user
- Framework - a web application hosted in cloud, used for uploading the packed application and also to visualize its output
- Data visualization users - the beneficiaries of the wrapped application, they can visualize information with the help of plots

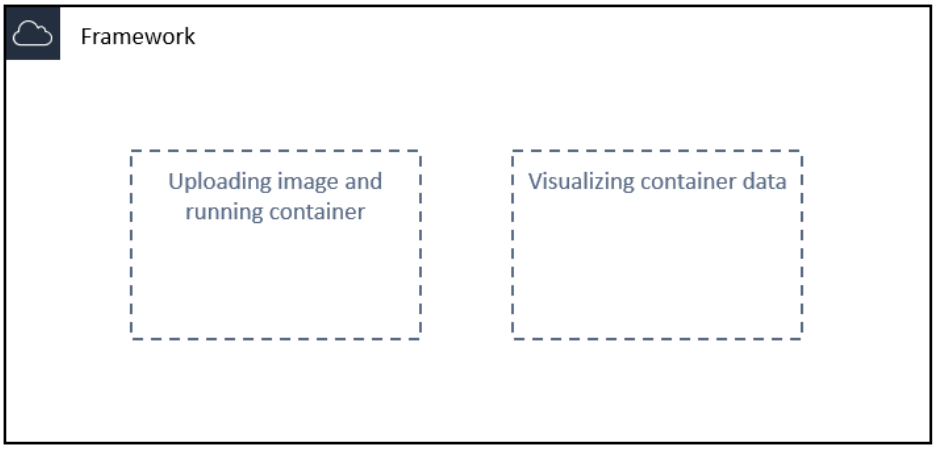


Figure 3.2: Framework main components

Figure 3.2 zooms in the Framework component. The two blocks are as follows:

- Uploading image and running containers - the data acquisition user after wrapping his fetching data application can use this side of the web application to upload and run it. The steps of achieving this goal is further discussed in chapter 4.
- Visualizing container data - this side of the framework is a tool which can be used to select plots filled with data coming from containers. Selection and usage of these plots is discussed in depth at chapter 4.

Chapter 4

The Stream Analysis Web Application

In this chapter the making of the web application is presented in details. The level of description should encourage other developers in using the technologies that were used in this application.

4.1 Front-end

The Stream Analysis Web Application actually consists of two web applications, since Front-end is decoupled from the Back-end and both of them run on different ports. The Front-end/Back-end pattern is very popular nowadays, because the role of each application is well defined. By loose coupling the two parts means that the Front-end can be reused with another Back-end. Moreover this Back-end can be written in any programming language and any web framework.

4.1.1 Angular

The Front-end of Stream Analysis Web Application was written with 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.[3]

Angular supports Single Page Applications which is used by Stream Analysis Web Application as well. Single Page Applications are a type of web applications that load a single HTML page, and the page is updated dynamically according to the interaction of the user with the web app. Single Page Applications, also known as SPAs, can communicate with the back-end servers without refreshing the full webpage, for loading data in the application. SPAs provide better user experience as no one likes to wait too long for reloading of the full webpage.[4]

Angular takes advantage of modularity. One can think of modularity in Angular as if the code is organized into “buckets”. These buckets are known as “modules” in Angular. The application’s code is divided into several reusable modules. A module has related components, directives, pipes, and services grouped together. These modules can be combined with one another to create an application. Modules also offer several benefits. One of them is lazy-loading, that is, one or more application features can be loaded on demand. If properly used, lazy-loading can increase the efficiency of an application a lot.[4]

To run Angular the developer needs to install NodeJs and Angular CLI. Once these two components are installed the developer is three steps away from running a brand new Angular application.

npm new ProjectName

It creates a folder named ProjectName and downloads in it from official Angular repository the web application boilerplate files. From this point on the developer just needs to add additional to build its web site.

npm install

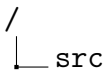
In Angular the dependencies are located in a .json file: package.json. Based on this file the command will download and install all the dependencies in the newly created root folder. The newly installed dependencies can be found in the folder ”nodemodules”.

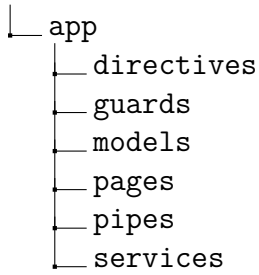
ng serve -o

Once the dependencies are successfully installed the developer can run the server and see web site in the browser. The command will build web application and open the default browser automatically to the default Angular url: http://localhost:4200.

4.1.2 File structure

In case of Stream Analysis Web Application the file structure is pretty simple. Every Angular entity type has its own folder.





As seen in the tree folder structure above there are six different types of Angular entities used. Thanks to the Angular CLI these can be easily created as shown below.

Directives

ng g c fileName

Directives represent an extended HTML syntax. Developers using the native HTML tags may want to create a customized tag that englobes the application's menu. For example in Stream Analysis Web Application's case the menu directive is used with the following HTML syntax: <app-menu></app-menu>. Other examples are: <app-live-chart></app-live-chart>, <app-zoom-chart></app-zoom-chart>, <app-login-form></app-login-form> and <app-register-form></app-register-form>. Once declared and defined, directives can be reused. Moreover they can also be parameterized. This feature is used to make charts plot different data types.

Guards

ng g g fileName

Guards can be used as an internal security for routes. In the application there is a file called auth.guard.ts, which guards the menu routes: it checks of the user was marked as logged in. In case the user did not log in he is redirected to the login page. If the user checks the 'Remember me' checkbox in the login page, then he can refresh the page and he is will still be marked as logged in. The application saves the current user to the browser's local storage and retrieves it with the guard once the user tries to access the page.

Models

ng g i fileName

Models are various class declarations with the properties. These are used throughout the application. This is possible because TypeScript is strongly typed, opposite to plain JavaScript.

Pages

ng g c fileName

Pages in the terms of Angular entities are in fact components. In this application they are the end result of routes. Once the user is routed to a certain page, he sees the HTML declared in these components.

Pipes

ng g p fileName

Pipes change the variable values used in components. They receive the current value of the variables and mutate them as the developer wishes on page rendering. Stream Analysis takes advantage of this to trim down chart dashcard card header names, since they contain as a prefix the user id. Hence on page rendering only the topic/queue names are shown, but in fact the variables used inside the charts are a concatenation of user id and topic/queue name.

Services

ng g s fileName

Services have a certain responsibility assigned to them. Usually they are used to communicate with the Back-end. However they can also persist variable values inside Angular. This feature is needed in case the user makes modifications in a certain page, then routes to another page and then routes back to the previous page. In this case the altered information on the first page will be lost. To persist the state across routings services are a great candidate. Services use the so called dependency injection. Developers need to inject them into components and make use of their declared methods.

4.2 Back-end

4.2.1 ASP .NET Core

The Back-end of Stream Analysis Web Application is written in ASP. NET Core, hence the language used is C-Sharp.

ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-based, Internet-connected applications. Its benefits are:[6]:

- A unified story for building web UI and web APIs.
- Architected for testability.
- Ability to develop and run on Windows, macOS, and Linux.
- Open-source and community-focused.
- Integration of modern, client-side frameworks and development workflows.
- A cloud-ready, environment-based configuration system.
- Built-in dependency injection.
- A lightweight, high-performance, and modular HTTP request pipeline.
- Ability to host on IIS, Nginx, Apache, Docker, or self-host in your own process.
- Tooling that simplifies modern web development.

4.2.2 SignalR

SignalR is a software library for Microsoft ASP.NET that allows server code to send asynchronous notifications to client-side web applications. The library includes server-side and client-side JavaScript components.[14]

Real-time web functionality is the ability to have server-side code push content to the connected clients as it happens, in real-time.[14]

SignalR takes advantage of several transports, automatically selecting the best available transport given the client's and server's best available transport. SignalR takes advantage of WebSocket, an HTML5 API that enables bi-directional communication between the browser and server. SignalR will use WebSockets under the

covers when it's available, and gracefully fall back to other techniques and technologies when it isn't, while the application code remains the same.[14]

In Stream Analysis Web Application SignalR is used to get data from topics in the Back-end and then send in real-time to the Front-end.

4.2.3 File structure

The application is structured as follows:

```
/
├── Controllers
├── Exceptions
├── Hubs
├── Models
├── Interfaces
├── Services
└── Logs
```

Controllers

Controllers are directly responsible for handling incoming requests from the Front-end. They are declared as simple C-Sharp classes, but are decorated with attributes. These attributes indicate the routing paths they are responsible for. In ASP .NET Core the 'Route' attribute can be parameterized:

- controller - At class level it interpolates the path string with the controller class name. In case the class it is sufixed with the word 'Controller' then this trimmed.
- action - At method level it interpolates the path string with the method name.

Stream Analysis Web Application's Back-end has the following routes and api (comments are provided for where needed):

1. AuthController

```
/**
* Route path is "api/Auth"
**/
[Route("api/[controller]")]
[ApiController]
```

```
public class AuthController : ControllerBase
{
        /**
        * Route path is "api/Auth/SignIn"
        **/
        [Route("[action]")]
        /**
        * Expects a POST type request
        **/
        [HttpPost]
        /**
        * Checks user credentials on sign in.
           The FromBody attribute means it gets
           the body content from the request and
           deserialize it into an object
           automatically.
        **/
        public async Task<ActionResult<bool>>
           SignIn([FromBody] SignInUserInfo user)

        /**
        * Route path is "api/Auth/Register"
        **/
        [Route("[action]")]
        [HttpPost]
        /**
        * Saves user credentials in the database
        **/
        public async Task<ActionResult<bool>>
           Register([FromBody] RegisterUserInfo
           user)

        /**
        * Route path is
           "api/Auth/GetCurrentUserId"
        **/
        [Route("[action]")]
        [HttpPost]
        /**
        * Returns current user id from the
           database
```

```
        **/
        public async Task<ActionResult<string>>
           GetCurrentUserId([FromBody] string
           username)
}
```

2. ContainerController

```
/**
* Route path is "api/Container"
**/
[Route("api/[controller]")]
[ApiController]
public class ContainerController : ControllerBase
{
        /**
        * Route path is
           "api/Container/CheckImage"
        **/
        [Route("[action]")]
        [HttpPost]
        /**
        * Checks if the image was succesfully
           uploaded in the cloud
        **/
        public async Task<ActionResult<bool>>
           CheckImage([FromBody]
           Models.Repository repository)

        /**
        * Route path is
           "api/Container/CreateRepository"
        **/
        [Route("[action]")]
        [HttpPost]
        /**
        * Creates an image repository in the
           cloud
        **/
        public async Task<ActionResult>
           CreateRepository([FromBody]
```

```
    Models.Repository repository)
}

/**
* Route path is
    "api/Container/CreateConfiguration"
**/
[Route("[action]")]
[HttpPost]
/**
* Creates an image configuration in the
    cloud
**/
public async Task<ActionResult>
    CreateConfiguration([FromBody]
    ImageConfiguration config)

/**
* Route path is "api/Container/RunImage"
**/
[Route("[action]")]
[HttpPost]
/**
* Runs the image in the cloud and
    creates a container
**/
public async Task<ActionResult>
    RunImage([FromBody]
    RunImageConfiguration runImageConfig)

/**
* Route path is
    "api/Container/ScheduleImageFixedRate"
**/
[Route("[action]")]
[HttpPost]
/**
* Creates a scheduler in the cloud for
    the container. Scheduler is at a fixed
    rate.
**/
```

```
public async Task<ActionResult>
   ScheduleImageFixedRate([FromBody]
   ScheduledImageFixedRate
   scheduledImageFixedRate)

/**
* Route path is
   "api/Container/ScheduleImageCronExp"
**/
[Route("[action]")]
[HttpPost]
/**
* Creates a scheduler in the cloud for
   the container. Scheduler is expressed
   as a Cron Expression. I.e. "0 0 12 * *
   ?" means every day at 12 PM.
**/
public async Task<ActionResult>
   ScheduleImageCronExp([FromBody]
   ScheduledImageCronExpression
   scheduledImageCronExpression)

/**
* Route path is
   "api/Container/StopScheduledImage"
**/
[Route("[action]")]
[HttpPost]
/**
* Deletes the scheduler rule.
**/
public async Task<ActionResult>
   StopScheduledImage([FromBody] string
   configName)

/**
* Route path is
   "api/Container/ListContainers"
**/
[Route("[action]")]
[HttpPost]
```

```
/**
* Lists all containers from the cloud.
**/
public async
   Task<ActionResult<List<Container>>>
   ListContainers([FromBody] string
   tasksGroupName)

/**
* Route path is "api/Container/StopTask"
**/
[Route("[action]")]
[HttpPost]
/**
* Stops a container.
**/
public async Task<ActionResult>
   StopTask([FromBody] string taskId)

/**
* Route path is
   "api/Container/StartFlushingQueues"
**/
[Route("[action]")]
[HttpPost]
/**
* Starts flushing the queues for a
   certain user.
**/
public async Task<ActionResult>
   StartFlushingQueues([FromBody]
   UserQueues userQueues)

/**
* Route path is
   "api/Container/GetUserQueues"
**/
[Route("[action]")]
[HttpPost]
/**
* Returns the queues for a certain user.
```

```
**/
public async
   Task<ActionResult<List<string>>>
   GetUserQueues([FromBody] string userId)

/**
* Route path is
   "api/Container/GetUserTopics"
**/
[Route("[action]")]
[HttpPost]
/**
* Returns the topics for a certain user.
**/
public async
   Task<ActionResult<List<string>>>
   GetUserTopics([FromBody] string userId)

/**
* Route path is
   "api/Container/UpdateUserChannels"
**/
[Route("[action]")]
[HttpPost]
/**
* Updates in the database the
   topics/queues for a certain user.
**/
public async Task<ActionResult<bool>>
   UpdateUserChannels([FromBody]
   UserChannels channels)

/**
* Route path is
   "api/Container/ListSchedulerRules"
**/
[Route("[action]")]
[HttpPost]
/**
* Returns all scheduler rules.
**/
```

```
public async
   Task<ActionResult<List<SchedulerRule>>>
   ListSchedulerRules([FromBody] string
   ruleNamePrefix)

/**
* Route path is
   "api/Container/DeleteSchedulerRule"
**/
[Route("[action]")]
[HttpPost]
/**
* Deletes a scheduler rule.
**/
public async Task<ActionResult>
   DeleteSchedulerRule([FromBody] string
   ruleName)
}

/**
* Route path is
   "api/Container/EnableSchedulerRule"
**/
[Route("[action]")]
[HttpPost]
/**
* Enables a scheduler rule.
**/
public async Task<ActionResult>
   EnableSchedulerRule([FromBody] string
   ruleName)

/**
* Route path is
   "api/Container/DisableSchedulerRule"
**/
[Route("[action]")]
[HttpPost]
/**
* Disables a scheduler rule.
**/
```

```
        public async Task<ActionResult>
            DisableSchedulerRule([FromBody] string
            ruleName)
}
```

3. DashboardController

```
/**
* Route path is "api/Dashboard"
**/
[Route("api/[controller]")]
[ApiController]
public class DashboardController : ControllerBase
{
        /**
        * Route path is
            "api/Dashboard/GetAllTopics"
        **/
        [Route("[action]")]
        /**
        * Expects a GET type request.
        **/
        [HttpGet]
        /**
        * Returns all topics from the database.
        **/
        public async
            Task<ActionResult<List<string>>>
            GetAllTopics()

        /**
        * Route path is
            "api/Dashboard/StartRealTimeMessagesFromTopic"
        **/
        [Route("[action]")]
        [HttpPost]
        /**
        * Starts the real time messages from a
            certain topic.
        **/
        public async Task<ActionResult>
```

```
        StartRealTimeMessagesFromTopic([FromBody]
            string topic)

        /**
        * Route path is
            "api/Dashboard/StopRealTimeMessagesFromTopic"
        **/
        [Route("[action]")]
        [HttpPost]
        /**
        * Stops the real time messages from a
            certain topic.
        **/
        public async Task<ActionResult>
            StopRealTimeMessagesFromTopic([FromBody]
            string topic)
}
```

4. HistoryController

```
/**
* Route path is "api/History"
**/
[Route("api/[controller]")]
[ApiController]
public class HistoryController : ControllerBase
{
        /**
        * Route path is
            "api/History/GetAllQueues"
        **/
        [Route("[action]")]
        [HttpGet]
        /**
        * Returns all the queues from the
            database.
        **/
        public async
            Task<ActionResult<List<string>>>
            GetAllQueues()
```

```
            /**
            * Route path is
               "api/History/GetHistoricalData"
            **/
            [Route("[action]")]
            [HttpPost]
            /**
            * Reads from the cloud the historical
               data from a certain queue.
            **/
            public async
               Task<ActionResult<List<QueueMessage>>>
               GetHistoricalData([FromBody] string
               queue)
}
```

Exceptions

Besides the common exceptions the application uses custom exceptions as well:

1. InexistentCluster
2. InexistentTaskDefinition
3. InvalidRepositoryName

Hubs

The SignalR Hubs API enables developers to call methods on connected clients from the server. In the server code, developers define methods that are called by client. In the client code, they define methods that are called from the server. SignalR takes care of everything behind the scenes that makes real-time client-to-server and server-to-client communications possible.

Stream Analysis has only one hub named: TopicsHub.

To enable SignalR Hubs in the application's Startup file the following configurations are made:

```
public void ConfigureServices(IServiceCollection
   services)
{
        services.AddSignalR();
```

```
}

public void Configure(IApplicationBuilder app,
   IHostingEnvironment env)
{
        app.UseSignalR(routes =>
           routes.MapHub<TopicsHub>("/hub/RealTimeMessages"));
}
```

Models

Models are user defined classes with properties that are used throughout the application. Some of the models are used to describe database tables:

```
/**
* Corelates to the table 'DashboardUsersTable' from
   the cloud hosted database.
**/
[DynamoDBTable("DashboardUsersTable")]
public class DashboardUser
{
        [DynamoDBHashKey]
        public string UserId { get; set; }

        public string Email { get; set; }

        public string Username { get; set; }

        public string Password { get; set; }
}

/**
* Corelates to the table 'ContainerUsersTable' from
   the cloud hosted database.
**/
[DynamoDBTable("ContainerUsersTable")]
public class ContainerUser
{
        [DynamoDBHashKey]
        public string UserId { get; set; }
```

```
        public string Email { get; set; }

        public string Username { get; set; }

        public string Password { get; set; }
}

/**
* Corelates to the table 'UserChannelsTable' from the
   cloud hosted database.
**/
[DynamoDBTable("UserChannelsTable")]
public class UserChannels
{
        [DynamoDBHashKey]
        public string UserId { get; set; }

        public List<string> Queues { get; set; }

        public List<string> Topics { get; set; }
}
```

Interfaces

Interfaces are used together with Services. In fact every Service implements a certain Interface, so that later on these Services could be injected through Dependency Injection. In Stream Analysis the Interfaces are named after the AWS service they work with:

1. IActiveMQService
2. ICloudWatchService
3. IContainerService
4. IDynamoDBService
5. IIamService
6. IS3Service
7. ISnsService

Services

In ASP .NET Core the lifetime of services is controlled entirely by the framework, however there a few types of lifetimes. To register services for Dependency Injection the following code needs to be added in the Startup file:

```
public void ConfigureServices(IServiceCollection
   services)
{
        /**
        * 'Scoped' means that one service instance is
           created for every client.
        **/
        services.AddScoped<IDynamoDBService,
           DynamoDBService>();
        services.AddScoped<IContainerService,
           ContainerService>();
        services.AddScoped<ICloudWatchService,
           CloudWatchService>();
        services.AddScoped<ISnsService, SnsService>();
        services.AddScoped<IS3Service, S3Service>();
        services.AddScoped<IIamService, IamService>();
        /**
        * 'Singleton' means that onc service instance
           is created when requested and never after.
        **/
        services.AddSingleton<IActiveMQService,
           ActiveMQService>();
}
```

4.3 Cloud environment

Stream Analysis Web Application is hosted in Amazon Web Services Cloud. Hence it makes use of the Cloud provider services. This thesis will not cover all the infrastructure laid and all the services used, but it will highlight the important ones.

4.3.1 Elastic Container Service (ECS)

Amazon Elastic Container Service (Amazon ECS) is a highly scalable, fast, container management service that makes it easy to run, stop, and manage Docker

containers on a cluster.[12]

Elastic Compute Cloud (EC2)

Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides secure, resizable compute capacity in the cloud. It is designed to make web-scale cloud computing easier for developers.[10] Basically it is a virtual machine in the cloud.

Docker

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. By doing so, thanks to the container, the developer can rest assured that the application will run on any other Linux machine regardless of any customized settings that machine might have that could differ from the machine used for writing and testing the code.[9]

In a way, Docker is a bit like a virtual machine. But unlike a virtual machine, rather than creating a whole virtual operating system, Docker allows applications to use the same Linux kernel as the system that they're running on and only requires applications be shipped with things not already running on the host computer. This gives a significant performance boost and reduces the size of the application.[9]

Since Docker can easily wrap an application into a container it is a great fit for Stream Analysis. The users that want to feed the queues and topics can make their own application without any restriction on the programming language.

ECS resources

An ECS instance is actually an EC2 instance that has Docker preinstalled and a container agent. The container agent runs on each infrastructure resource within an Amazon ECS cluster. It sends information about the resource's current running tasks and resource utilization to Amazon ECS, and starts and stops tasks whenever it receives a request from Amazon ECS.

Stream Analysis makes use of ECS in two ways:

1. The web application is deployed(hosted) on the ECS instance
2. Containers that are started by the user are actually running via Docker on the ECS instance.

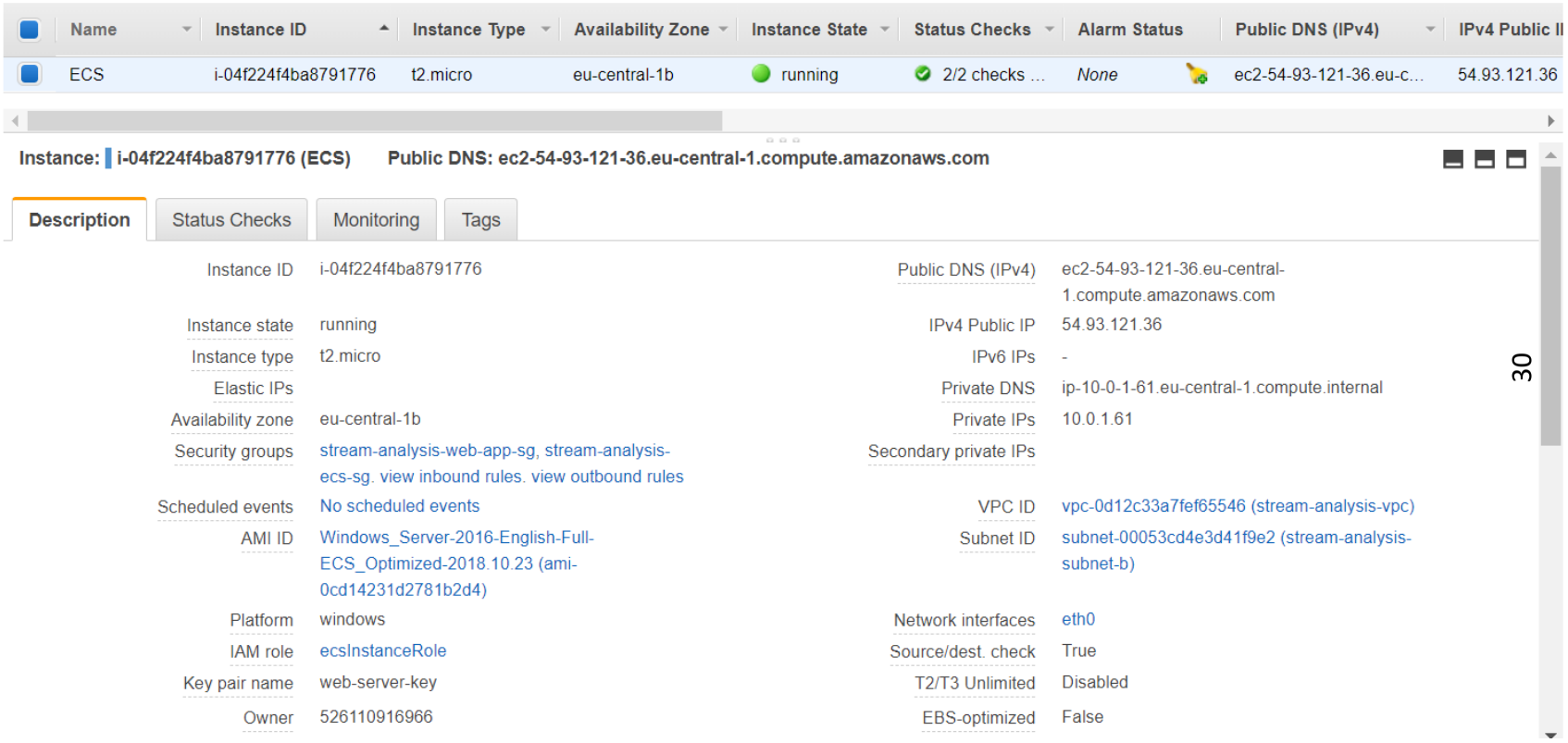


Figure 4.1: ECS instance description

The ECS instance details can be seen in the below figure.
ECS has the following entities:

1. Cluster - logical grouping of tasks
2. Task Definition - configuration of a container once it's launch via an image
3. Task - container

The Web Application owns a default cluster named: stream-analysis-ecs-cluster, as seen in figure 4.2. This one is used for all tasks(containers) the users launch.

Launching a container is similar to executing the command "run" from the Docker CLI on an image. In fact behind the scenes this is exactly what AWS does. However the Docker "run" command has a lot of options expressed as command line parameters. Stream Analysis provides two of these in the benefit of the user. These two parameters can be checked on the UI when the user configures his image, as seen in the figure 4.3.

The two options are as follows:

1. Interactive (-i) - Keep STDIN open even if not attached
2. Pseudo Terminal (-t) - Allocate a pseudo-TTY

If both are checked the following command will be executed behind the scenes.

docker run -it <imageId>

4.3.2 Elastic Container Registry (ECR)

Amazon Elastic Container Registry (Amazon ECR) is a managed AWS Docker registry service that is secure, scalable, and reliable. Amazon ECR supports private Docker repositories with resource-based permissions using AWS IAM so that specific users or Amazon EC2 instances can access repositories and images. Developers can use the Docker CLI to push, pull, and manage images.[11]

The Web Application uses ECR to store the Docker images that users push. These images represent the containerized application. Figure 4.4 shows an example of such an image store in the cloud registry.

Every image is put in a repository in ECR and user is responsible to give a name in the Front-end. Also the user is responsible to push the image to ECR from the CLI (instruction are given in the UI).

Figure 4.2: ECS Cluster description

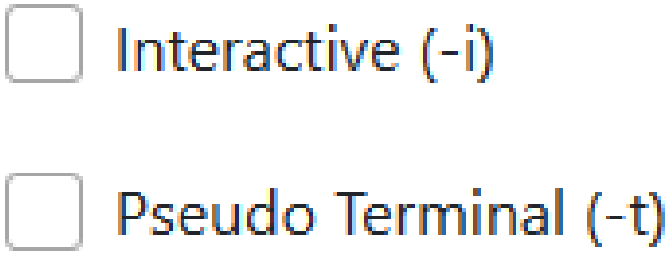


Figure 4.3: Container Scheduling Options

4.3.3 Amazon CloudWatch

Amazon CloudWatch monitors the developer Amazon Web Services (AWS) resources and the applications he is running on AWS in real time. The developer can use CloudWatch to collect and track metrics, which are variables you can measure for your resources and applications.[7]

However CloudWatch has another functionally that is used by the Stream Analysis Web Application. It can schedule events to happen based on time. The two time options are:

1. Fixed rate interval
2. Cron expression

The scheduling system is used by the Web Application to launch scheduled containers. The user must provide the necessary parameters in the Front-end as seen in figure 4.5. The user also needs to provide a scheduling rule name.

4.3.4 Amazon MQ

Amazon MQ is a managed message broker service for Apache ActiveMQ that makes it easy to migrate to a message broker in the cloud. A message broker allows software applications and components to communicate using various programming languages, operating systems, and formal messaging protocols.[2]

Stream Analysis uses such a broker but it only listens to the OpenWire protocol. Hence the users in their containerized application must send information to a defined path and port:

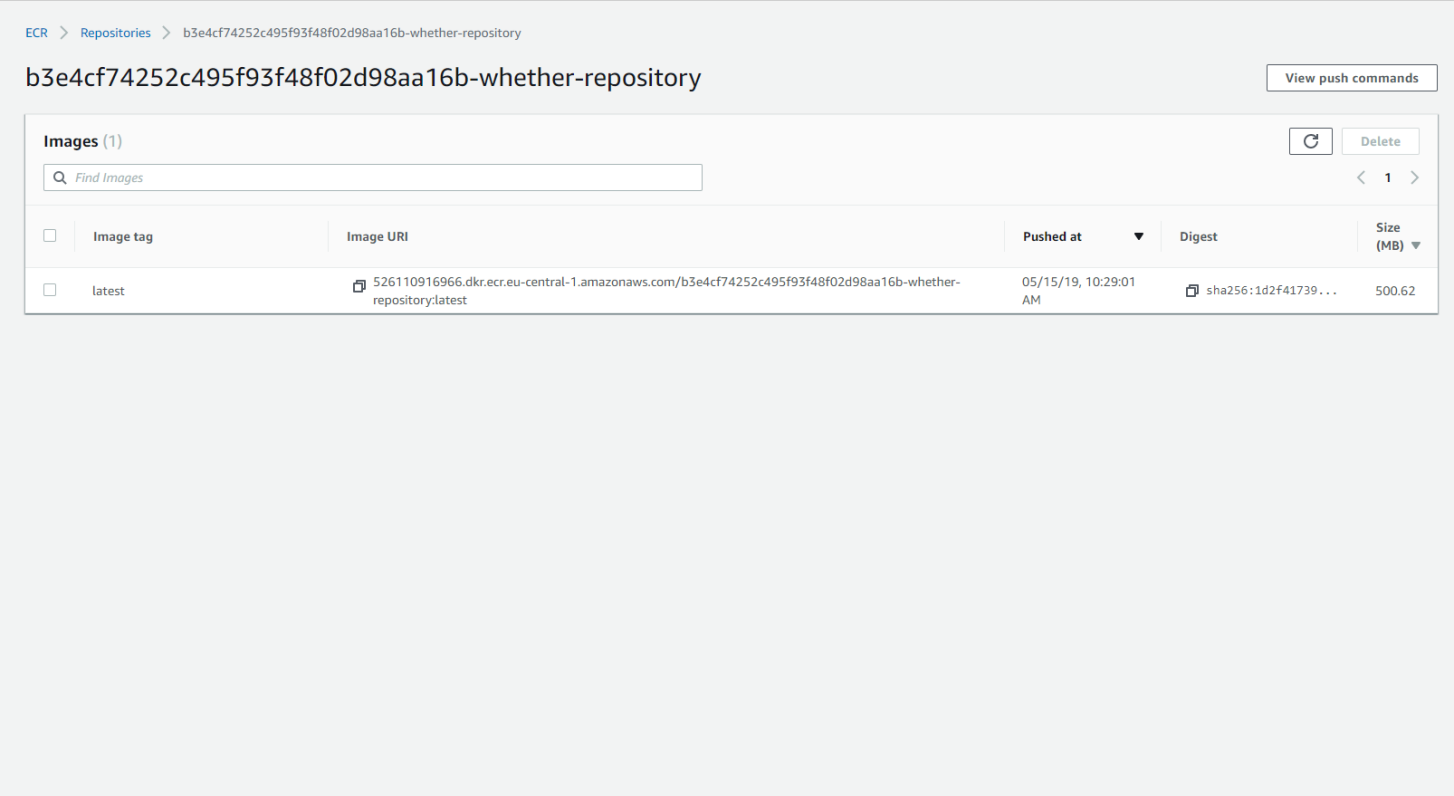


Figure 4.4: Example image from ECR

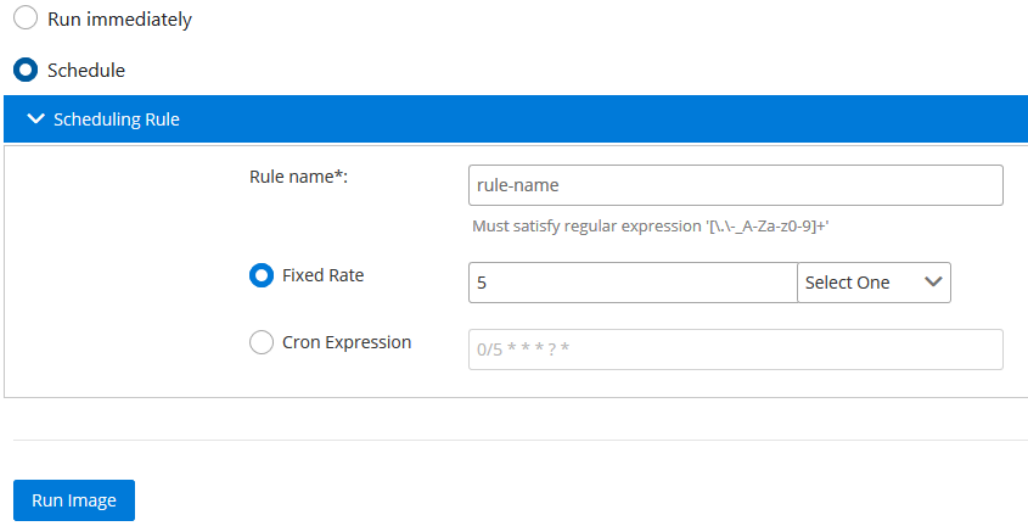


Figure 4.5: Container Scheduling Options

activemq:ssl://b-d517d345-559e-4c9c-b84a-8413f5aedbdd-1.mq.eu-central-1.amazonaws.com:61617

Users are informed about this URI in the Front-end.

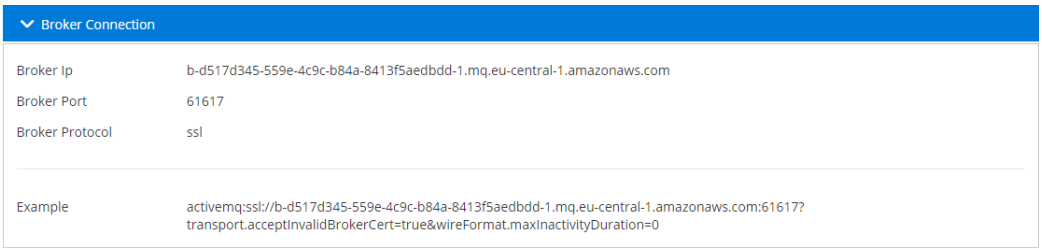


Figure 4.6: Broker URI

The broker from the cloud has the following configurations:

Users are also informed from the Front-end about the topic and queue data model.

The topic data is restricted to the format given in figure 4.8.

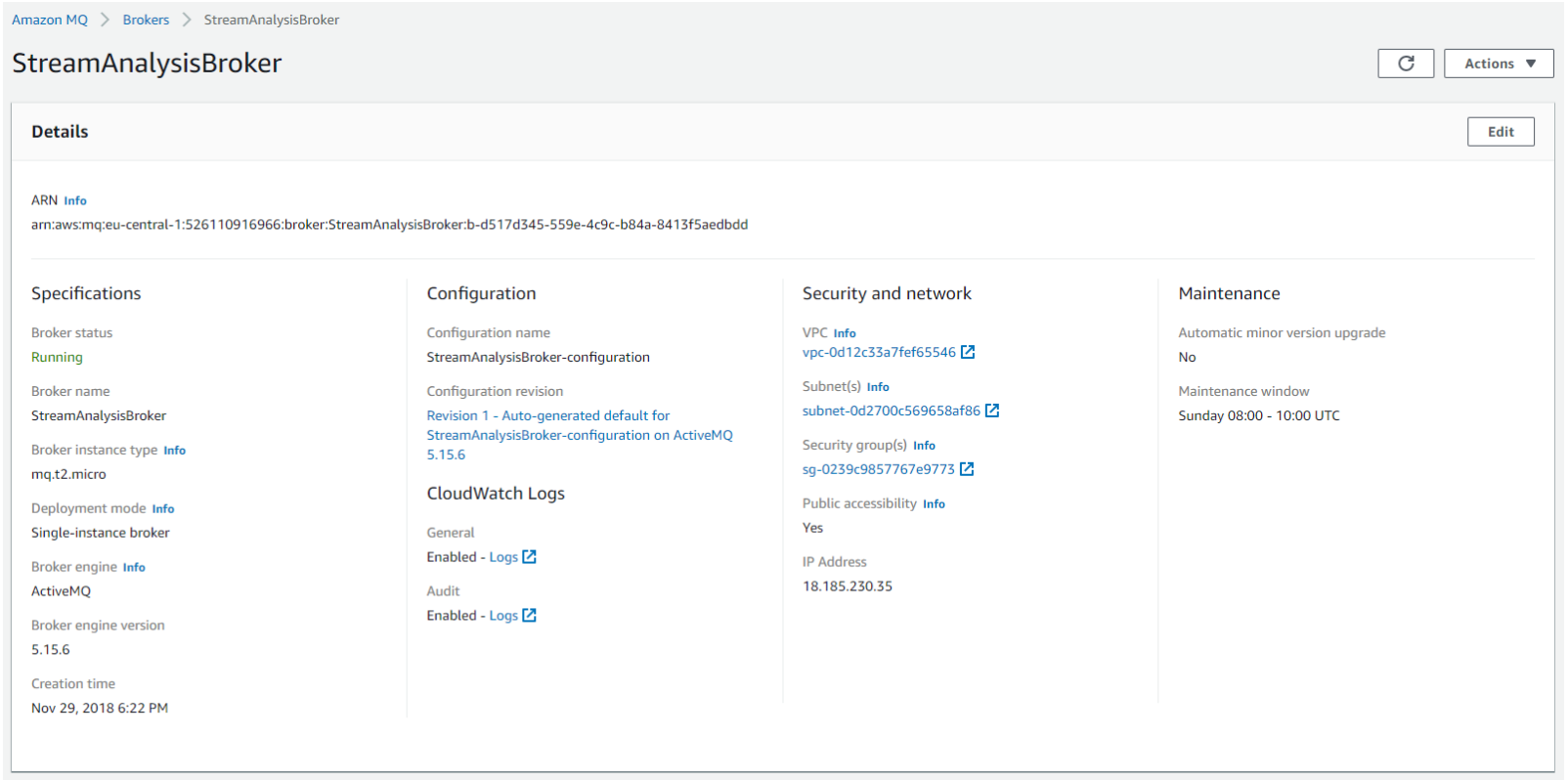


Figure 4.7: Broker description

Figure 4.8: Topic model

The queue data is restricted to the format given in figure 4.9.

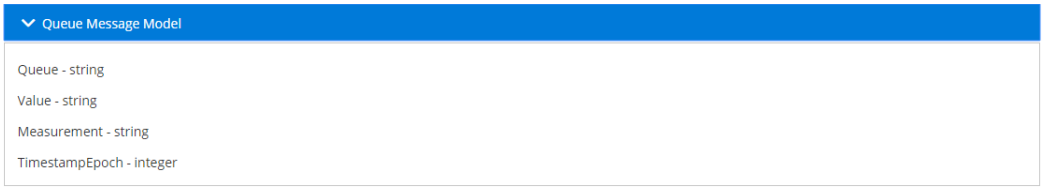


Figure 4.9: Queue model

Apache ActiveMQ

Apache ActiveMQ™ is the most popular open source, multi-protocol, Java-based messaging server. It supports industry standard protocols so users get the benefits of client choices across a broad range of languages and platforms. Connectivity from C, C++, Python, .Net, and more is available. Integrate your multi-platform applications using the ubiquitous AMQP protocol. Exchange messages between your web applications using STOMP over websockets. Manage your IoT devices using MQTT. Support your existing JMS infrastructure and beyond. ActiveMQ offers the power and flexibility to support any messaging use-case.[5]

Stream Analysis makes use of Apache ActiveMQ by storing the queues and topics. These channels can be then used to receive and send data.

4.3.5 Simple Storage Service (S3)

Amazon Simple Storage Service is storage for the Internet. It is designed to make web-scale computing easier for developers.[13]

Amazon S3 has a simple web services interface that you can use to store and retrieve any amount of data, at any time, from anywhere on the web. It gives any developer access to the same highly scalable, reliable, fast, inexpensive data

storage infrastructure that Amazon uses to run its own global network of web sites. The service aims to maximize benefits of scale and to pass those benefits on to developers.[13]

The Web Application flushes queue data into files that are stored in the cloud. More precisely into a bucket. The bucket is called: stream.analysis.bucket.

The files put in separate folders based on the user. Folder are named after the user's id. File names are a concatenation of user id and queue name, as it can be seen below. Each file hold the data of one and only one unique queue per user.

4.3.6 Simple Notification Service (SNS)

Amazon Simple Notification Service (Amazon SNS) is a web service that coordinates and manages the delivery or sending of messages to subscribing endpoints or clients. In Amazon SNS, there are two types of clients—publishers and subscribers—also referred to as producers and consumers. Publishers communicate asynchronously with subscribers by producing and sending a message to a topic, which is a logical access point and communication channel. Subscribers (i.e., web servers, email addresses, Amazon SQS queues, AWS Lambda functions) consume or receive the message or notification over one of the supported protocols (i.e., Amazon SQS, HTTP/S, email, SMS, Lambda) when they are subscribed to the topic.[15]

Stream Analysis Web Application uses the Simple Notification Service to flush every 5 seconds the queues. It contains a topic named: stream-analysis-notification. This topic receives information about the queues that need to be flushed and triggers the Lambda.

Figure 4.12 shows the topic details, but also the fact that a Lambda is subscribed to it.

4.3.7 Amazon Lambda

AWS Lambda is a compute service that lets a developer run code without provisioning or managing servers. AWS Lambda executes his code only when needed and scales automatically, from a few requests per day to thousands per second. He pays only for the compute time you consume - there is no charge when your code is not running. With AWS Lambda, he can run code for virtually any type of application or backend service - all with zero administration. AWS Lambda runs his code on a high-availability compute infrastructure and performs all of the administration of the compute resources, including server and operating system maintenance, capacity provisioning and automatic scaling, code monitoring and logging. All he needs to do is supply your code in one of the languages that AWS Lambda supports.[8]

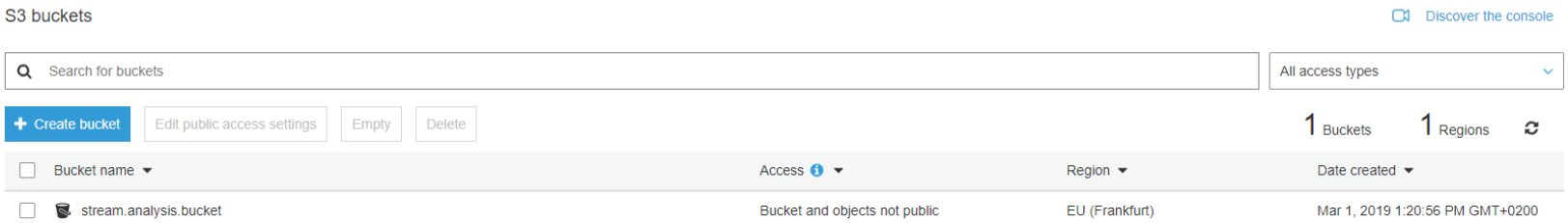


Figure 4.10: Bucket description

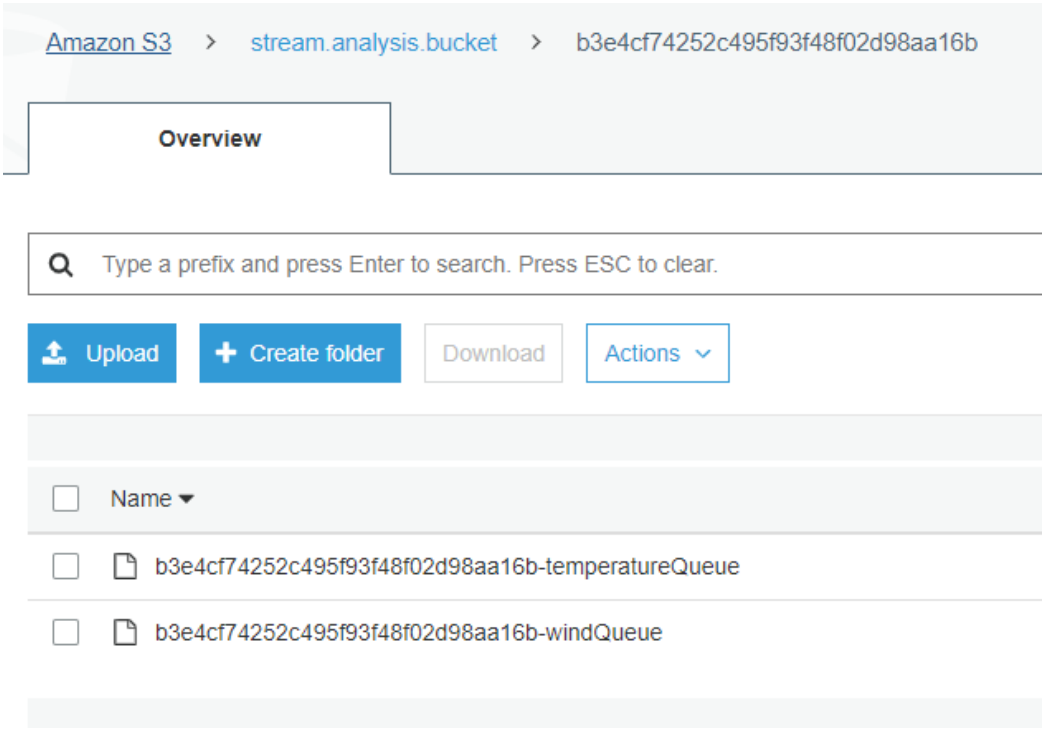


Figure 4.11: File examples

The Web Application owns such a Lambda service and its name is: ActiveMQFlush-Lambda. More details are shown in figure 4.13. Basically Lambda function is a standalone application that can run on command, once it is triggered by an event. Figure 4.13 shows that the application was written in .NET Core 2.1 and that the actual function that handles the event in the code is named ”FunctionHandler”.

This Lambda function is responsible to flush the queues once it is notified to do so.

The ”FunctionHandler” mentioned above has the following signature:

```
public async Task FunctionHandler(SNSEvent snsEvent,
    ILambdaContext context)
{
}
```

One can see that the function is expecting a Simple Notification Service event. This is event is triggered once the topic ”stream-analysis-notification” receives a message with the queue. The SNSEvent class carries records with the messages the topic received. In this case these records are the queue names that need to be dequeued and then saved into the S3 bucket ”stream.analysis.bucket”.

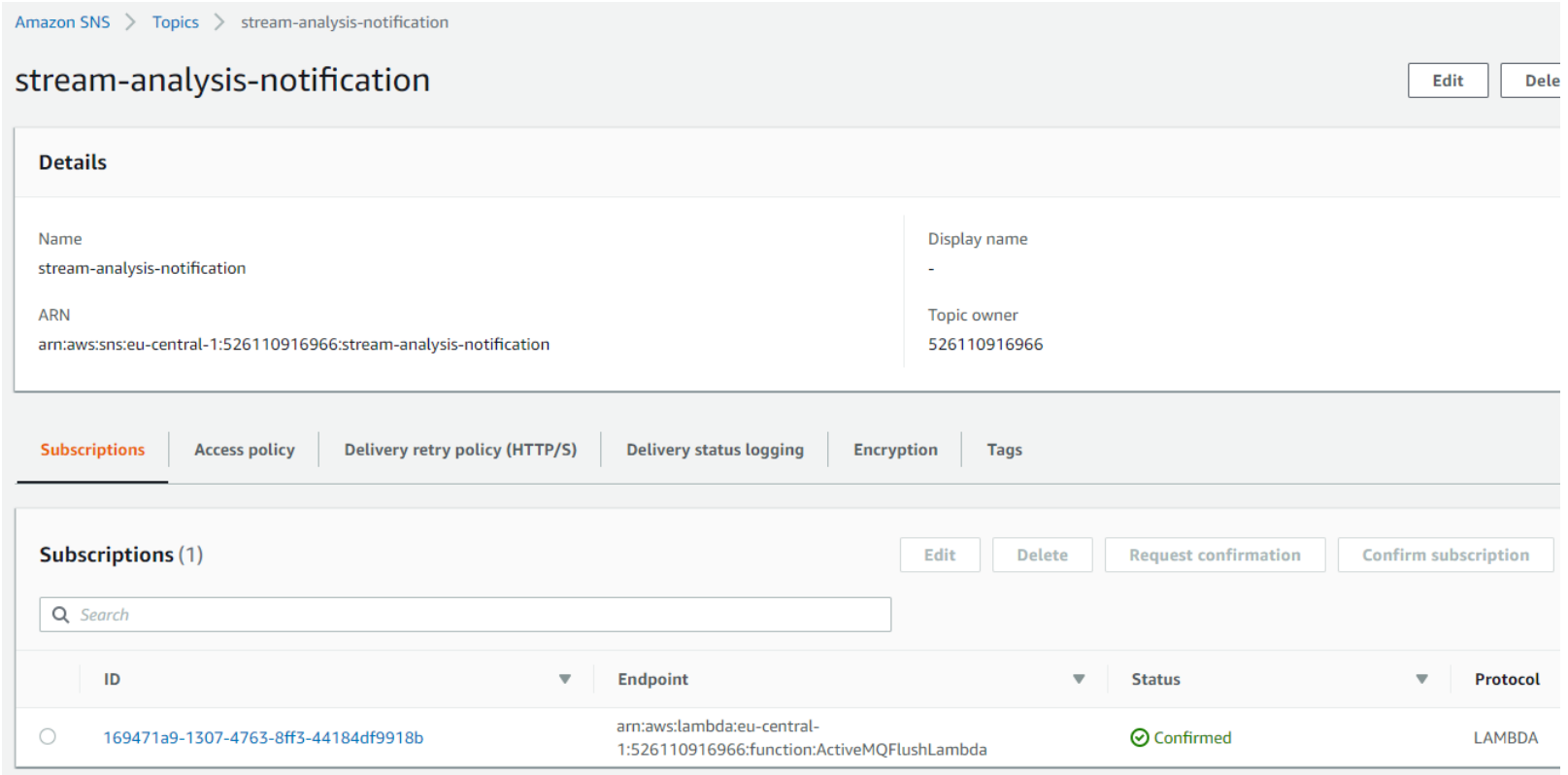


Figure 4.12: Topic description

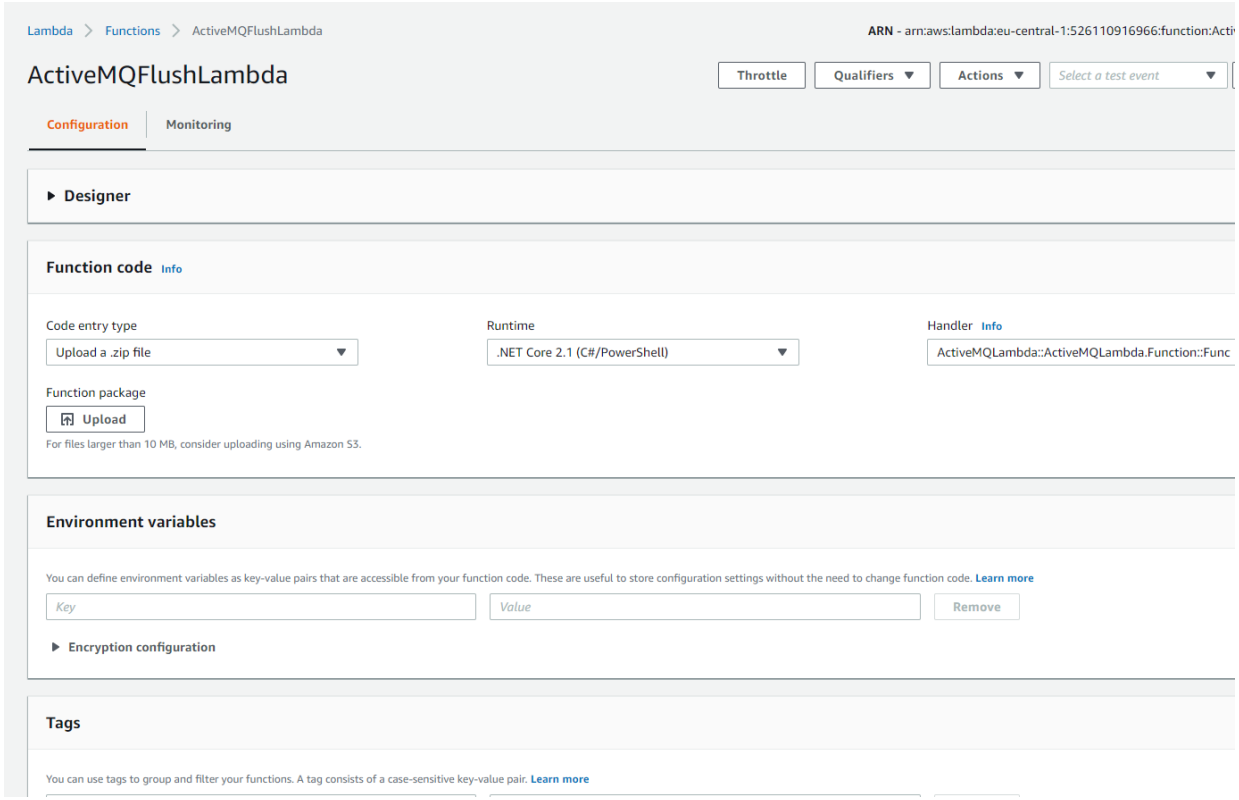


Figure 4.13: Lambda description

Taking the Lambda function code more in-depth we have the following steps:

1. Connect to the broker
2. Get the queue names from SNSEvent
3. Iterate over all the queues
4. Append the queues content the associated files from the S3 bucket

4.4 Containerized application

The containerized application is provided by the users. The purpose of the application is to connect to the given broker and start emitting messages into defined topics and queues.

The application can be made in any programming language as long as it has support from Apache ActiveMQ. An example would be .NET, where the package name is: Apache.NMS.ActiveMQ.

The below example uses fragments of C Sharp code and provides a step by step presentation on what the container users should aim for. To be noted that this example uses a self made library wrapped around Apache.NMS.ActiveMQ, however the steps can still be clearly understood.

1. Connect to the broker

```
using (IStreamAnalysisConnection
   connection =
   factory.CreateConnection())
{
        connection.Start();
}
```

2. Create a streaming session and stream weather data

```
using (IStreamAnalysisSession session =
   connection.CreateStreamingSession(queue1))
{
        var queueMessage =
           Observable.Interval(TimeSpan.FromSeconds(8))
        .Select(x =>
        {
                var weather =
                   ApixuService.GetWeatherDataByAutoIP();
```

```
                return new QueueMessage()
                {
                        Queue =
                           queue1.Split("://")[1],
                        Value =
                           weather.current.temp_c,
                        Measurement =
                           "°C",
                        TimestampEpoch =
                           weather.last_updated_epoch
                }
        });
        session.StreamData(queueMessage);
}
```

Chapter 5

Web Application Usage Guide

Chapter 6

Conclusions

Bibliography

[1] *"Amazon Web Services"*. URL: https://en.wikipedia.org/wiki/Amazon_Web_Services.

[2] *"AmazonMQ"*. URL: https://docs.aws.amazon.com/amazon-mq/latest/developer-guide/welcome.html.

[3] *"Angular"*. URL: https://en.wikipedia.org/wiki/Angular_(web_framework).

[4] *"Angular"*. URL: https://hackr.io/blog/why-should-you-learn-angular-in-2019.

[5] *"ApacheActiveMQ"*. URL: https://activemq.apache.org/.

[6] *"ASPDotNetCore"*. URL: https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.2.

[7] *"AWSCloudWatch"*. URL: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html.

[8] *"AWSLambda"*. URL: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html.

[9] *"docker"*. URL: https://opensource.com/resources/what-docker.

[10] *"EC2"*. URL: https://aws.amazon.com/ec2/?nc2=h_m1.

[11] *"ECR"*. URL: https://docs.aws.amazon.com/AmazonECR/latest/userguide/what-is-ecr.html.

[12] *"ECS"*. URL: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html.

[13] *"S3"*. URL: https://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html.

[14] *"SignalR"*. URL: https://en.wikipedia.org/wiki/SignalR.

[15] *"SNS"*. URL: https://docs.aws.amazon.com/sns/latest/dg/welcome.html.