ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Technische Universität Berlin

Faculty IV - Electrical Engineering and Computer Science
Information Systems Engineering

Assignment

Enterprise Computing
Winter Semester 2016/2017

Mobile Backend Serverless Reference Architecture
and
Real-time File Processing Serverless Reference Architecture

Group 1	:	Christof Schubert	344450
		Daniel Andika Steinhaus	342563
		Diogenis Lazaridis	341484
		Luis Orta Hernández	380016

Version's Date : 12th February 2017

Contents

1 Serverless Architectures and AWS Lambda Functions

This chapter gives a short introduction into what is meant while talking about serverless architectures and especially *Function as a Service*.

1.1 Serverless Architectures in General

To understand the difference between serverless architectures and traditional architectures a short flashback to the traditional Three-Tier-Architecture which can describe well a lot of web applications, is given. The simple Three-Tier-Architecture consists accordingly to the name of three tiers, the presentation tier, the logic tier and the data tier (see 1).

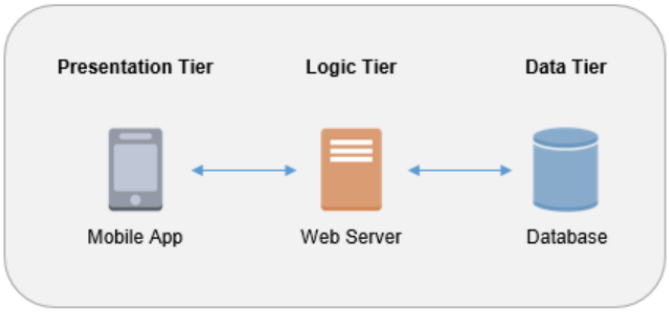

Figure 1: Architectural pattern for a simple three-tier application (source: [BBNN15])

The presentation tier represents the component that the users interacts directly with (such as web pages, mobile app user interfaces and so on). The logic tier translates the user actions to the functionality which determines the application's behavior (usually executed on a referring web server). The data tier provides the storage media that holds the relevant data for the application (for example databases, object stores, caches and so on).With a *serverless* architecture in this context the application's developer outsources the logic tier or the web server to a third-party vendor whereby the three-tier structure in its core stays the same.

For the term *serverless* it is not clearly defined in general what it should mean. There are mainly two different (but also strongly related) areas of what it can mean (see [Rob16]):

- First it was originally used to describe applications which significantly rely on third party applications and services (in the cloud) to manage server-side logic and state. These types of third party services can be described as **Backend as a Service**.
- Second *serverless* can also refer to applications where some amount of server-side logic is still written by the application's developer and is run in stateless compute containers that are event-triggered, ephemal (only last for one invocation), and fully managed by third party applications. This approach can be described as **Function as a Service (FaaS)**.

The further considerations refer mainly to the second meaning when the term *serverless* is used. The following serverless architectures are therefore structured to outsource the logic tier's execution and server provisioning or management while providing the developer to implement mostly the high level part of the functionality by himself. The leading vendor for *FaaS* is *Amazon* at the moment. *Amazon* provides the *AWS* platform (*Amazon Web Services*) with different products which aim to enable the implementation of a serverless architecture very easily. One of the provided services is *AWS Lambda* which can be used as current standard example for *FaaS*.

1.2 Function as a Service with AWS Lambda

The *AWS Lambda* service allows the developer to run the code without provisioning or managing the needed servers. It performs almost the complete administration of the resources including server and operating system maintenance, capacity provisioning and automatic scaling, code monitoring and logging. Basically *Faas* is about running back end code without managing an own server or own server applications. A provisioned server or an all-time running application is not needed. The code is not required to specific framework or library. Currently *AWS Lambda* supports *JavaScript* or *Node.js*, *Java*, *C#* and *Python*. However the *Lambda* function can execute another process that is bundled with its deployment artifact, so it can actually consist of any language that can compile down to a Unix process. Since no server application is needed to run, the deployment is different to traditional systems: The code is uploaded to the *Faas* provider typically as a ne definition (e.g. in a zip or JAR file), and then calling a proprietary API to initiate the update. The service does everything else by itself. *AWS Lambda* executes the code when needed and scales automatically from a few requests per day to thousands per sercond while only causing costs for the actual execution time. The 'compute containers' executing the functions are ephemeral with the *FaaS* provider provisioning and destroying them driven by runtime need. The *Lmabda* functions are event-triggered by event types which are predefined by the provider. Example types can be (file) updates, time schedule tasks or messages added to a message bus. Further the most providers allow triggers as responses to inbound `http` requests typically in some kind of API gateway (also does *AWS* with *Amazon API Gateway*).
Understanding *AWS Lambda* as *FaaS*, it is also useful to consider following points amongst others (see [Rob16]):

- **Stateless Architecture**: *FaaS* functions are restricted when it comes to local (machine / instance / bound) states. It should be always assumed that a given function invocation's in-process or host state is *not* available to *any* subsequent function invocation. States written to RAM or to a local discs are included. This is is called a *stateless architecture* and *serverless architectures* are typically *stateless architectures*. This leads for the functions itself to an either naturally stateless implementation (only transforming given input to referring output) or to using a database, a cross-application cache or network file store to store states across requests or further input to handle the requests.

- **Execution Duration**: *FaaS* functions are usually limited in the execution duration (runtime of an invocation). A *AWS Lambda* function for example is not allowed to run longer than five minutes, otherwise it is terminated automatically. Therefore long lived tasks have to be designed within a suited architecture approach.

- **Startup Latency**: At the moment there are many factors which determine the startup latency of a function's invocation. They latencies can vary from some miliseconds up to several minutes depending amongst other things on the function's code size and the used framework. For many application this is not an issue. But if for example the target is to set up an low latency trading application, the *FaaS* approach might currently not be the system of choice. This should be kept in mind.

- **API Gateway**: The API Gateway (e.g. *Amazon API Gateway*) is a `http` server where routes or endpoints are predefined and mapped to a *FaaS* function. For received `http` requests the server extracts information about the relevant function and function parameters, calls the function and after function execution transforms the result into a `http` response and returns it to the original caller. Further API gateway servers also can perform authentication, input validation, response code mapping and so on. One use case for API gateway and *FaaS* servers can be for creating

`http`-fronted microservices in a serverless way with all the scaling, management and other benefits that come with *FaaS*.

This chapter has given a short explanation of terms and functionality of serverless architectures and *FaaS* which is considered the base for the review of the *AWS Lambda* reference serverless architectures in the next chapter.

2 AWS Lambda Serverless Reference Architectures

Amazon web services (AWS) published five simple examples of how to realize serverless architectures with AWS lambda (see [Vog16]). In the following sections 2.1 and 2.2 the first architecture (*Mobile Backend Serverless Reference Architecture*) and the second one (*Real-time File Processing Serverless Reference Architecture*) are investigated. In the next chapter 3 alternative design approaches not using the *AWS* platform and services are introduced for both of them.

2.1 Mobile Backend Serverless Reference Architecture

The *Mobile Backend serverless reference architecture* (see figure 2) is a demonstration on how to use *AWS Lambda* along with other *AWS* services to set up a serverless backend for a mobile application. The provided example application enables users to upload photos and notes using *Amazon Simple Storage System* (*Amazon S3*), *Amazon DynamoDB* and *Amazon API Gateway* amongst other services next to *AWS Lambda*.

2.1.1 Architecture

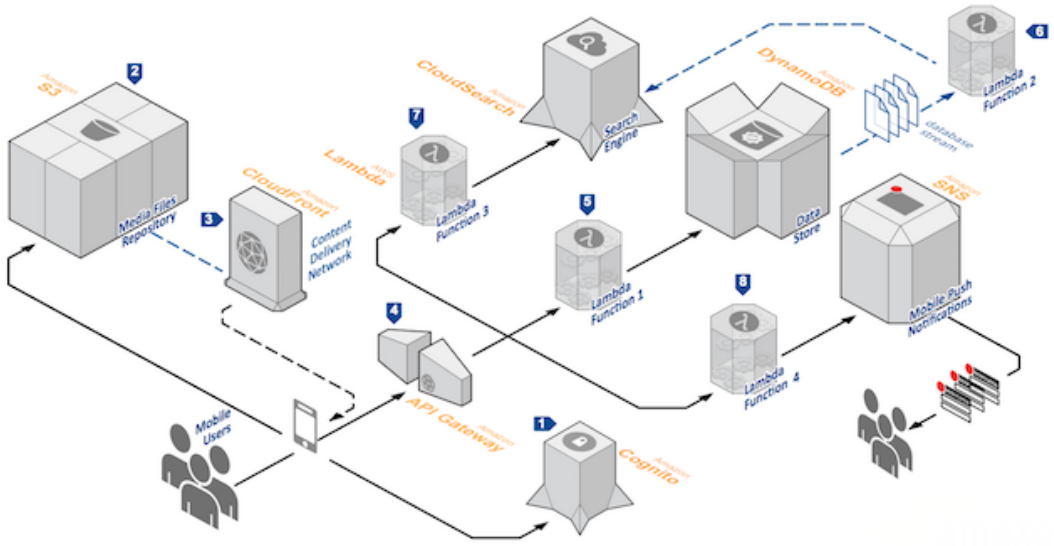

Figure 2: Web applications serverless reference architecture[1]

Referring to figure 2 it follows a short explanation of the architecture's nodes and their functionality:[2]

1. Users retrieve an identity from *Amazon Cognito*. (*Amazon Cognito* provides mobile identity management and data synchronization across mobile devices). Once the user has retrieved an identity, the user can access other *AWS* services.
2. User-generated files (photos) can be stored in an *Amazon S3* file repository.
3. Users can access their uploaded files (photos) through *Amazon CloudFont*, a low latency content delivery network.
4. Users can send requests to the *Amazon API Gateway* to access application logic and dynamic data. The *Amazon API Gateway* is an entry point for the application to access code running on *AWS Lambda*.

[1] Source: [Vog16]
[2] Source: `https://s3.amazonaws.com/awslambda-reference-architectures/mobile-backend/lambda-refarch-mobilebackend.pdf`

5. *Lambda* function 1 provides a synchronous end-point for users to store and retrieve unstructured data from the *Amazon DynamoDB* database.
6. *Lambda* function 2 retrieves changes made by users from *DynamoDB* streams to create search-able documents. The documents are inserted into *Amazon CloudSearch* (a service to set-up, manage and scale search solutions for websites or applications).
7. *Lambda* function 3 provides a synchronous interface for users to search data from cloud search.
8. *Lambda* function 4 provides an asynchronous end-point for users to communicate with other users within the mobile application. The function formats each communication request and sends a push notification to specific users with *Amazon SNS* (a push notification service).

2.1.2 Implementation

The actual implementation can be found in a public GitHub repository.[3] After taking a first look at the implementation it attracts attention that *Lambda* function 4 and the according *Amazon SNS* service is not implemented or used yet. So the following investigations only refer to points 1 to 7 of the architecture's node functionality list (see section 2.1.1).

Set-Up AWS Services: The realization of the project relies on different AWS services next to the *Lambda* functions which are discussed below. *Amazon Cognito Identity Pool*, *Amazon S3*, *Amazon API Gateway*, *Amazon DynamoDB* and *Amazon CloudSearch* are used, not considering the push notification use case. This services have to be configured and launched. To ensure an easy way to start the application, *Amazon* offers an service called *AWS CloudFormation*.

- *Amazon CloudFormation*: By a given template file *AWS CloudFormation* creates and manages a collection of related *AWS* resources. The event triggering of the *Lambda* functions (*event source mappings*) is defined here for example together with several policies, the configuration table of the Dynamo DB which contains the endpoint informations for the *CloudSearch* service, and so on. Except some further information which has to be passed manually during the launching of the services on the *AWS* platform, almost every configuration information for the service collection is stored here.
- *Amazon API Gateway*: The API gateway is the access point to the application logic referring to the notes handling. It is configured to handle `http` requests, to call the relevant *Lambda* functions for searching or creating a note below and to return the results or responses. E.g. the requests to create a note are expected to have the following content ...

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "CreateNoteRequest",
  "type": "object",
  "properties": {
    "noteId": { "type": "string" },
    "headline": { "type": "string" },
    "text": { "type": "string" }
  }
}
```

Listing 1: http request - create note

... and the following content is returned afterwards:

[3] `https://github.com/awslabs/lambda-refarch-mobilebackend`

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "CreateNoteResponse",
  "type": "object",
  "properties": {
    "success": { "type": "boolean" }
  }
}
```

Listing 2: http response - create note

This requests are mapped to the *upload-note Lambda* function.

- *Amazon DynamoDB*: The *Amazon DynamoDB* instance is running for storing the created and uploaded notes in a relevant table. Further it includes a table with configurations for the *CloudSearch* endpoints. It interacts with *Lambda* function 1 (search) to retrieve new notes and with *Lambda* function 2 (stream-handler) to pass new or updated notes to the *CloudSearch* engine.
- *Amazon CloudSearch*: *Amazon CloudSearch* is a service which sets up, manages and scales up a search solution for websites or applications. It creates a search domain to which search-able data can be added, and executes instances which handle search requests. In the application it manages the search domain for the uploaded notes.
- *Amazon Cognito Identity Pool*: *Amazon Cognito* is a services which handles the user's signing in and signing up to mobile and web apps. It provides several authentication procedures for example through social identity providers (*Facebook*, *Amazon* itself and so on) or by using an own identity system. Here it handles the user pool for the access to the mobile backend.
- *Amazon S3*: *Amazon S3* is a simple object storage service with a simple web interface. In this case it stores the user's uploaded photos.

Lambda Functions: Three *JavaScripts* are provided in the implementation for the *Lambda* functions search (1), stream-handler (2) and upload-note (3). Each of them requires and imports the AWS-sdk which contains JavaScript objects for Amazon S3, DynamoDB and so on. Further an export handler (function) is defined in each JavaScript. This is executed by *AWS Lambda* if the function is invoked. The function's parameters are `event` which passes th event data to the handler, and context which provides the runtime information of the *Lambda* function to the handler.
The functionality of the *Lambda* functions' implementation is described in the following list:

1. **search**: The JavaScript code contains mainly the export handler and two further functions: `handleEvent()` and `processSearchResults()`. First the AWS-sdk is imported and an instance of the `AWS.DynamoDB.DocumentClient` is created. Further a variable `config` is declared. This variable contains the configuration parameters for the DynamoDB.
   - *Export handler*: The export handler calls the `handleEvent()` directly if a valid configuration `config` is set. Otherwise it tries to retrieve the configuration from DynamoDB table *'MobileRefArchConfig'* with key *(Environment: 'demo')*, sets the configuration itself and calls `handleEvent()` afterwards.
   - `handleEvent(event, context)`: An instance of `AWS.CloudSearchDomain()` is created first whereby the needed endpoint is retrieved from the configuration. Further the execution parameters are set. The search term to search for is extracted from the event data passed by the `event` parameter and the size or maximum number of results is set to 10. The Cloud Search Domain instance's search function is called with the specific parameters and if results (a list of

documents that match the specified search criteria), the function `processSearchResults()` is called within a succeeded context. Otherwise an error is stated.

- `processSearchResults(data)`: The function creates a response object for the search results it gets as parameters, and returns it. The function iterates over the list of result documents (`hits`) and parses each document for an ID, headline and text and pushes it into an `searchResult` object. This object is appended afterwards in the `searchResults` list which is returned in the end within the response object next to a *success* tag.

2. **stream-handler**: The JavaScript code contains mainly the export handler and three further functions: `handleEvent()`, `createSearchDocuments()` and `uploadSearchDocuments()`. First the AWS-sdk is imported and an instance of the `AWS.DynamoDB.DocumentClient` is created. Further a variable `config` is declared. This variable contains the configuration parameters for the DynamoDB.
   - *Export handler*: The export handler does basically the same as the one for the *Lambda* function *search*. Certainly the called `eventHandle()` function is another one.
   - `handleEvent(event, context)`: The function retrieves the *records* object from the event data parameter and calls the function `createSearchDocuments()` with it. Afterwards it calls the function `uploadSearchDocuments()` with the *searchDocuments* it got from the first call, and the context parameter.
   - `createSearchDocuments(records)`: The function iterates over the given records list which it gets as parameter. For every record a *searchDocument* object is created which contains an ID, a headline and the note text. This object is pushed into *searchObjects* list which is returned in the end.
   - `uploadSearchDocuments(context, searchDocuments)`: First the *searchDocuments* list from function `createSearchDocuments()` is checked to not be empty. If it is empty the function succeeds the context without further actions. Otherwise an instance of `cloudSearchDomain` is created, whereby the needed endpoint comes from the configuration. Further an object *params* is created which contains the *searchDocuments* list in JSON format. With the `cloudSearchDomain`'s function `uploadDocuments()` the object with the JSON content is uploaded to the cloud search domain service and returns true for success and an error for fail.

3. **upload-note**: The code contains mainly the export handler and an `handleEvent()` function. First the AWS-sdk and the DynamoDB-Doc are imported and an instance of the `AWS.DynamoDB.DocumentClient` is created. Further a configuration as for *Lambda* function search is declared.
   - *Export handler*: The export handler does basically the same as the one for the *Lambda* function *search*. Certainly the called `eventHandle()` function is another one.
   - `handleEvent(event, context)`: First it create an note object which contains the DynamoDB table which should used to store the note and is retrieved from the configuration and ID, headline and text of the note which are retrieved from the event data in the referring parameter. With the `AWS.DynamoDB.DocumentClient push()` function the note object is pushed to the database and an response object including an success entry (*true* for success and an error for fail) is returned.

2.2 Real-time File Processing Serverless Reference Architecture

The *Real-time File Processing Serverless Reference Architecture* (see figure 3) is a general-purpose, event-driven, parallel data processing architecture. It is designed for workloads that need more than one data derivative of an object. It demonstrates a simple markdown conversion application where *Lambda* functions are used to convert markdown files to HTML and plain text.

2.2.1 Architecture

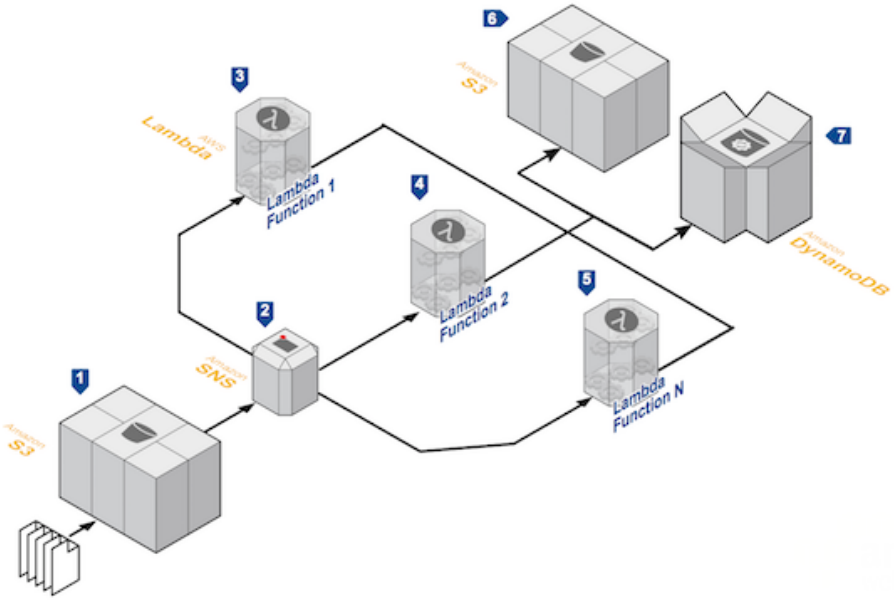

Figure 3: Real-time file processing serverless reference architecture[4]

Referring to figure 3 it follows a short explanation of the architecture's nodes and their functionality:[5]

1. Objects are uploaded into *Amazon S3*. The *S3* bucket publishes an event notification to an *Amazon SNS* topic.
2. *Amazon SNS* is a push notification service. It receives an event notification from the input *S3* bucket and sends messages to the file processing *Lambda* functions.
3. *Lambda* function 1 is used to create a layer of processing for a wide variety of data sets and subsequently sends the results to a post-processing storage layer.
4. *Lambda* function 2 can in parallel process and create another data derivate from the same *S3* objects.
5. Any number of *Lambda* functions can be created to process data.
6. A post-processing storage layer (e.g. S3) can be added to store the results.
7. Alternatively, results can be sent to *DynamoDB* for further processing and/or querying with low-latency.

[4]Source: [Vog16]
[5]Source: `https://s3.amazonaws.com/awslambda-reference-architectures/file-processing/lambda-refarch-fileprocessing.pdf`

2.2.2 Implementation

The actual implementation can be found in a public GitHub repository.[6] After taking a first look at the implementation it can be observed that the stated *DynamoDB* is not integrated yet. Only simple *S3* storages for input and output files are provided. Thus the *DynamoDB* is not considered furthermore. Additionally the *Amazon SNS* service is the only service which is not already described in section 2.1.1:

- *Amazon SNS*: The *Amazon SNS* service is a push notification or messaging service, which supports a publish/subscribe messaging paradigm. For each *SNS* service different *topics* can be created. A *topic* is a named group of events or access points which refer to a certain subject, content or event type. Each topic has an unique identifier which defines an unique endpoint for *publishers* and *subscribers*. *Publishers* send messages to *topics*. *SNS* matches the topic to a list of *subscribers* interested in the topic, and delivers the messages to them. Subscribers are clients (applications, end-users, servers and son on) which want to receive notification messages on specific *topics*.

Lambda Functions: Two *JavaScripts* are provided in the implementation for the *Lambda* functions data-processor (1) and data-processor (2). Each of them imports the AWS-sdk which contains JavaScript objects for Amazon S3, DynamoDB and so on, and creates and *Amazon S3* client. Further they import the *async* package and either the *marked* (for processing a markdown file to a `html` file) package or the *remove-markdown* package (for processing a markdown file to a plain text file). The both scripts are following basically the same scheme. They execute three functions in their *exports.handler* function which have to be executed with the *waterfall* function of the *async* package due to their *callback* structure. The three functions are `download` which downloads the input object (markdown file), `transform` which processes the markdown content either to a `html` or to a plain text content, and `upload` which uploads the processed content in a new object into the output storage bucket.

- `exports.handler`: First the input message part which contains the information for input bucket and file, is converted from an object to a JSON string and modified. Basically the characters `/` and `\` are removed. Afterwards the JSON string is parsed to an object again and the information for input and output are stored in a new created object. Afterwards the waterfall execution of the three methods `upload`, `transform` and `upload` is executed.

- `download`: With the already created *S3* client the input object is downloaded and the client's download function calls the transform function as callback function. It passes the downloaded file's content.

- `transform`: The function transforms the content of the input parameter to either `html` or plain text content and uses the imported package *marked* or *remove-markdown*. It passes the processed content to the `upload` function.

- `upload`: The function defines the new object name from the original file name and the new file extension which is either `"html"` or `txt"`. Afterwards it uploads the new content in a new object into the *S3* output bucket which has basically the same name as the input bucket extended with `_output`. But the bucket is not created here.

[6] `https://github.com/awslabs/lambda-refarch-fileprocessing`

3 Design Approaches

3.1 Mobile Backend Serverless Reference Architecture

The Amazon *Web Services Mobile Backend Reference Architecture* can be duplicated on the *Google Cloud Platform* by using similar and available microservices. For almost all AWS services, there are equivalent services. But for the most important microservice, *AWS Lambda Function*, Google provides *Google Cloud Function*, which is still an Alpha version and not recommended for production use. First the equivalent *Google Cloud Services* are described[7]:

- **Google Cloud Identity & Access Management (IAM)**: *Google Cloud Identity & Access Management (IAM)* authentication can be performed with an email and password, as well as federated identity provider integration like Google. In comparison to the *AWS Cognito* service, it only provides authentication with a Google account, a Service account (which belongs to the application itself), a Google group (collection of Google accounts and Service accounts) and a Gsuite domain. Facebook, Twitter or Amazon accounts are not supported as in *AWS Cognito* (which also supports Google accounts). In other hand, *Google Cloud Identity & Access Management (IAM)* is offered at no additional charge.

- **Google Cloud Storage**: *Google Cloud Storage* supports chunked encoding and resumables uploads that could be valuable in streaming services. It supports multiregional locations in United States, Europe and Asia, whereas AWS S3 offers more options region-wise. The Google Storage price is the same per GB per month in Multi-Regional Storage while the *ASW S3* price varies by region.

- **Google Cloud Functions**: *Google Cloud Functions* is an Alpha version not recommended for production use, whilst *AWS Lambda* is completely functionality permitting to deploy serverless architectures. Google Cloud Functions goal is a lightweight compute solution for stand-alone functions that respond to Cloud events without the need of a server or runtime environment as *AWS Lambda*. *Google Cloud Functions* has, at the moment, some limitations on as a maximum of 20 functions per project and only supporting Javascript.

- **Google Cloud Content Delivery Network CDN** : *Cloud Content Delivery Network (CDN)* has lower network latency, offloads origins, and reduces serving costs. According network performance metrics based on tests performed with RIPE Atlas,[8] the *Cloud Content Delivery Network (CDN)* has a lower latency in every region as *AWS CloudFront*.[9] The *Cloud Content Delivery Network (CDN)* supports the `HTTP/2` protocol in addition to the `HTTP/1.0` and the `HTTP/1.1` protocol and allows to use `SSL/TLS` certificates to secure the content using a domain name as *AWS CloudFront*.

- **Google Cloud API** : *Google Cloud API* is a gateway to connect to the microservices via REST calls or client libraries programming languages, allowing to use as well third-party clients. The access via SDK command line tools is also available. However, the use of this services is not necessary due to the Alpha version of *Google Functions*, because all of the HTTP invocations are made via HTTP triggers. There is an option to integrate *Cloud Functions* with any other Google service that supports Cloud Pub/Sub, a messaging service that allows you to send and receive messages between independent applications, or any service that provides HTTP callbacks. In a final version of *Cloud Functions*, using *Google Cloud APIs* could be the best way for calling this functions.

[7] For further information refer also to: `https://cloud.google.com/solutions/mobile/mobile-app-backend-services`
[8] RIPE Atlas is a global network of probes that measure Internet connectivity and reachability, providing an unprecedented understanding of the state of the Internet in real time. Source: `https://atlas.ripe.net/`
[9] `https://cloudharmony.com/network-3m-for-aws:cloudfront-and-google:cdn-in-global-us_west-eu_west-eu_central-eu_east-and-eu-from-ripe`

- **Google Cloud Jobs API**: *Google Cloud Jobs API* is in an Alpha closed version, which means that the design issues are resolved and all the functionality is being verified. *Cloud Jobs API* objective is providing intuitive job search with relevant search results and adding relevant information and recommendations. *CloudSearch* is fully functional and supports a wide range of data types, enabling it to be applied to a variety of search applications. In addition, *CloudSearch* supports 35 languages, autocomplete suggestions, geospatial search, and others features.
- **Google BigTable**: *Google Cloud Bigtable* has offers a relatively fast, because of its low latency, completely managed, scalable NoSQL database service. Moreover, it offers security due to all data is encrypted (characteristics not present on AWS Dyanamo), and the replication strategy based on the *Google File System (GFS)*. Google offers also a NoSQL document database for web and mobile applications, *Google Cloud Datastore*. It supports a different data types for property values as *AWS Dyanamo*. This services could be also suitable for this architecture.
- **Firebase Cloud Messaging**: *Firebase Cloud Messaging* delivers notifications to Android, iOS, or Web apps whereas *AWS Simple Notification Service (SNS)* allows, in addition, to distribute notification to Fire OS and Windows devices, and SMS messages to mobile device users worldwide. With Firebase Cloud Messaging notification messages are showed to the user or can be data messages for controling the application code. It has a distribution of messages feature in three ways: single devices, groups of devices, and devices subscribed to topics.

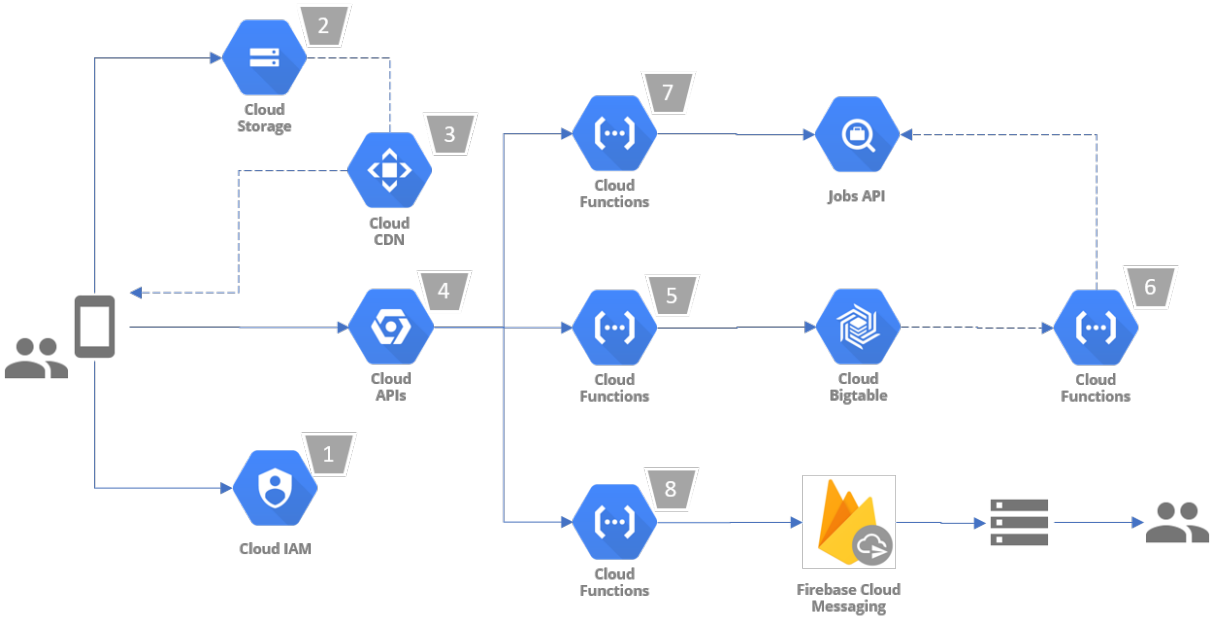

Figure 4: Serverless Mobile Backend with Google Cloud Services

The design of the equivalent architecture on the Google Cloud Platform is described below:

1. Mobile users are authenticated by using *Google Cloud Identity & Access Management (IAM)*. Users can use an email and password or standard Google accounts.
2. Media files are stored in Google Cloud Storage.
3. The *Cloud Content Delivery Network CDN* allow users to access the uploaded data with a low network latency.
4. All the requests are sent by the users through *Cloud APIs* with rest calls to perform the app functionality from the codes in the *Google Cloud Functions*.

5. *Google Cloud Function* is part of this serverless architecture. As with *AWS*, *Google Cloud Function* 1 provides a synchronous endpoint for users to store and retrieve unstructured data from *Google Cloud Bigtable* (instead of *DynamoDB*).

6. *Google Cloud Function* 2 uses a *Dataflow Connector* for *Google Cloud Bigtable* to retrieve changes made by users, creates a search-able document, and inserts it into *Cloud Jobs API*.

7. *Google Cloud Function* 3 offers a synchronous interface for users to search for data from *Cloud Jobs API*.

8. *Google Cloud Function* 4 provides an asynchronous endpoint for mobile users to communicate with each other within a mobile application. The function formats each message request and sends a push notification in three ways: single devices, groups of devices, and devices subscribed to "topics" with *Firebase Cloud Messaging*. *Firebase* is not part of Google Cloud Platform though Google Cloud Messaging users are strongly recommended to upgrade to this solution.

3.2 Real-time File Processing Serverless Reference Architecture

The following chapter is a discussion about the decisions which Microsoft Azure services were used and which ones were rejected as a substitution of the services occurring in the lambda reference architecture. Referring to figure 3 in chapter 2.2 there are several Amazon services used by the reference architecture namely *Amazon S3*, *Amazon SNS*, *AWS Lambda* and *Amazon DynamoDB*. The overall pattern of these services can be mapped through functional interaction of several similar Microsoft Azure Services:

- **Azure Storage**: *Amazon S3* is basically a pure storage service for storing large amounts of unstructured data. The equivalent to this service in the Azure Infrastructure is called *Azure Storage*. It is not a pure object storage like *S3* but rather consists of several independent and specialized services. Figure 5 shows the overview of a General-purpose Storage Account composed of the services Blob Storage, Table Storage, Queue Storage and File Storage (Share).

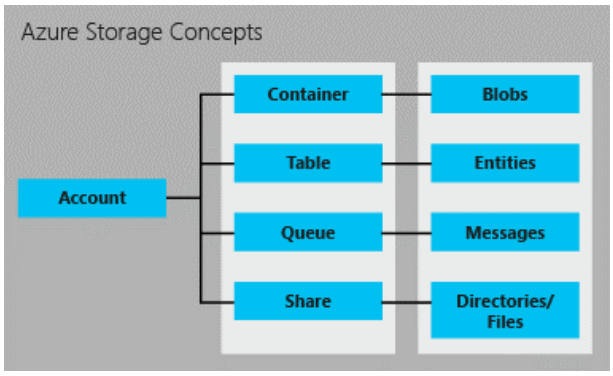

Figure 5: Azure Storage Environment[10]

Our scenario requires a storage for markdown files which are typically not large in size but also have no cap in file size as well. The most appropriate storage services for this task could be a Blob Storage as well as a Table Storage. Since the main use case for Queue Storage is about messaging and for File Storage sharing files, they are not helpful to realize a service to *Amazon S3*. Blobs in a Blob Storage have to be stored in a container inside the storage account and can be used for very large files (up to 4.77 Terabyte[11]) whereas the NoSQL Table Storage might be used for files

[10]Source: `https://docs.microsoft.com/en-us/azure/storage/media/storage-introduction/storage-concepts.png` (Last access: 30.01.2017))
[11]Source: `https://docs.microsoft.com/en-us/azure/storage/storage-introduction (Last access: 30.01.2017)`

in the order of magnitude of typical markdown file sizes. The decision for using blobs was made of three reasons. First, although storing files in the Table Storage would be possible, a clean design approach would include a segregation of the database layer and the storage layer. We would have to add both storage types what basically means adding a complexity layer on top of the architecture. Second bearing in mind that in extreme cases larger markdown files than usual could slow down a Table Storage having the files stored internally, and third, Azure already delivers a template for blob triggers.

- **Azure Functions**: Azure Functions is the direct equivalent as *AWS Lambda Functions* from Amazon. As both are the FaaS part of the Platforms the basic concept behind them is to process custom written code in several programming languages. The specific thing about it is that the processing resources needed for both the number of invocations as well as the complexity of one function are scaled automatically in the background. Functions from both services can be triggered by several options, these can be e.g. `http` requests or events incoming from other services.

- **Azure Storage Queue**: The Amazon platform only supports one binding from an S3 storage to a lambda function to trigger it.[12] Thus the intermediate component *Amazon SNS* introduced in section 2.2, is necessary. The whole functionality *SNS* provides including the publish/subscribe mechanism are covered by different components at the Azure platform. The mobile push notifications could be realized by using *Azure Notification Hubs*. The processing of messages to other services could be realized either by using *Azure Storage Queues*, *Azure Service Bus Queues/Topics* or an *Azure Event Hub*. Since the reference architecture does not include mobile push notifications there is only the need to distinguish between the three last-named services.

  Queues in general are characterized by a one-way/two-way communication where the sender places a message on the queue and a receiver (consumer) will collect the message either soon or at some time in the future[13] eventually performing actions e.g. acknowledging the reception, locking or deleting the message in the queue.

  Azure Storage queues have a non-blocking receive mechanism also no possibility to push messages forward in a time-critical[14] way. Theoretically, there are unlimited concurrent clients supported[15] but there is no option to split the clients into consumer groups which receive the same message concurrently to process it. A consumer could peek and lease on a message to see if it was processed before, but the consumers would need to know in how many different ways a message should be processed (in our case two times, one transformation to `html` and one to plain text) before deleting it. Since that is an impractical way of dealing with messages in our scenario as well as the time-critical aspect, we rejected this service being an equivalent to *SNS*.

- **Azure Service Bus**: *Azure Service Bus Queues* support queues in a similar way, with the difference of having a non-blocking receiving behavior and a push API which are only accessible by using .NET. The negative aspect is that only one consumer can process a message at a time.

  A *Service Bus Topic*, however, uses a one-to-many publish/subscribe pattern like *SNS*. Thus messages sent to a topic can have a copy of that message forwarded to multiple subscriptions. Every

[12] Source: `https://forums.aws.amazon.com/thread.jspa?threadID=169678` (Last access: 29.01.2017)
[13] Source: `http://microsoftintegration.guru/2015/03/03/azure-event-hubs-vs-azure-messaging` (Last Access: 01.02.2017)
[14] Source: `https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-azure-and-service-bus-queues-compared-contrasted` (Last Access: 30.01.2017)
[15] Source: `https://www.todaysoftmag.com/article/1260/what-messaging-queue-should-i-use-in-azure` (Last Access: 29.01.2017)

consumer entity receives an individual copy of every message from the topic it subscribed to. Reviewing the services the functionality of this component seems to be the most similar one the one Amazon SNS provides in the reference architecture. Nevertheless, we decided due to several reasons to use an *Azure Event Hub* and not a *Service Bus Topic*.

- **Azure Event Hub**: Basically an Event Hub is designed to receive events from many different senders and pass them on to one or several consumer. In this case, the different consumers can be segmented into consumer groups where eventually several competing consumers can exist. Figure 6 illustrates the issue of an Event hub and especially points out the feature of partitions and consumer groups.

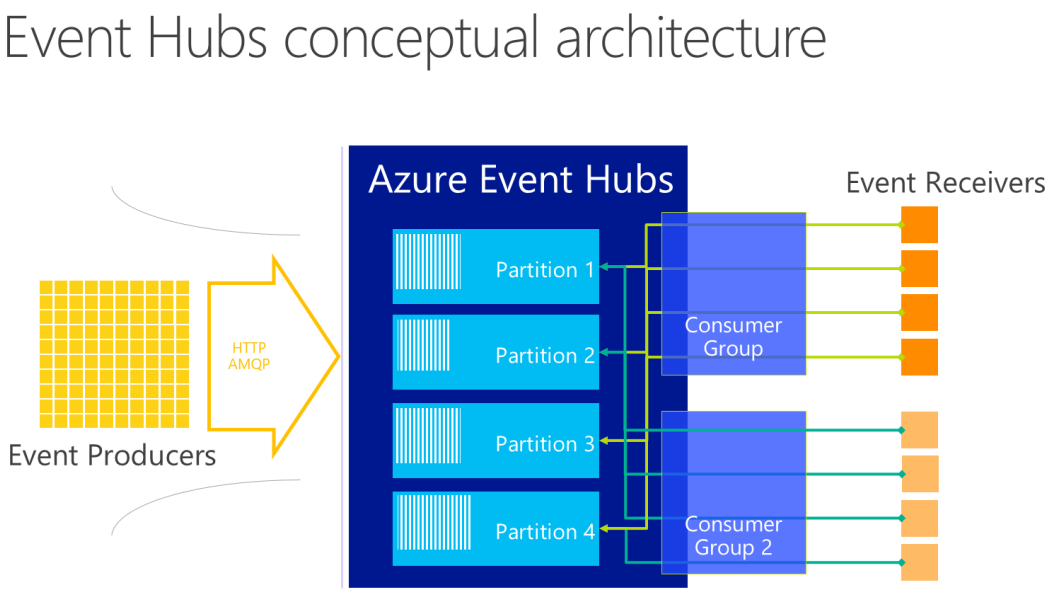

Figure 6: Basic Azure Event Hub Functionality[16]

A partition is an ordered sequence of events where new events are appended on. Partitions are provided for consumers in a consumer group to read on. Typically (but not necessarily) there is a partition for every consumer group which can be read from independently. The message payload of the Event Hub side is provided as a stream which can be read from the consumers using an offset. The aspects below highlight the technical decisions towards using an Event Hub instead of *Service Bus Topic* as a more suitable component to Amazon SNS from the lambda reference architecture. In comparison to the components mentioned above, an event hub can operate on a much higher scale. Since the processing pattern is event-driven rather than polling the messages, it can be delivered at a lower latency. Even messages in a *Service Bus Topic* are sent the same way as to a queue but not received from the topic directly. They are instead received from a topic subscription which is similar to a virtual queue that receives copies of the message meant for the topic.[17]

The stream characteristic of an event hub enables a lower messaging overhead. Since the messages in a *Service Bus Topic* have to be sent to every subscriber, the resulting processing of these messages in such a topic is lower. If there are e.g. 100.000 Messages incoming and 10 subscribers to the topic then the Service Bus needs to create 1 million messages overall. In opposite there is only one event hub stream that can be read from all consumers.

[16] Source: `https://docs.microsoft.com/en-us/azure/event-hubs/media/event-hubs-what-is-event-hubs/event_hubs_architecture.png` (Last Access: 01.02.2017)

[17] Source: `https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-queues-topics-subscriptions` (Last Access: 01.02.2017)

For an event hub there are two logical performance limitations. First an event hub can provide 32 partitions at the maximum. So only 32 consumer groups per event hub can be served since the partitions relate to the number of concurrent subscribers (or subscriber groups). Second every event hub's scaling also refers to "Throughput Units" (TUs) as payment and scaling model. Each TU can process 1MB of ingress, 1000 events or 2MB of egress *per second* and will stop if one of the limitations is met.[18] The TUs can be varied between 1 to 20 units which leads to a limitation to 20000 input events, 20 MB ingress or 40MB egress per second. In our scenario we chose the typical pattern of two partitions, one for every consumer group/processing function (the equivalent to the lambda functions in the lambda reference architecture). Since the lambda reference architecture states the possibility of having any count of lambda functions, one event hub could be easily scaled to a maximum of 32 partitions or in other words managing the events of 32 processing functions with an overall ingress of 20MB per second. The ingress limitation is of interest for deciding the content of the input event messages.

The only negative aspect of an event hub is the consumer having to deal with failure or managing the offset value for reading the stream and detect whether it has already read a message or not on its own.

Regarding the described decisions, the following architectural design is developed:

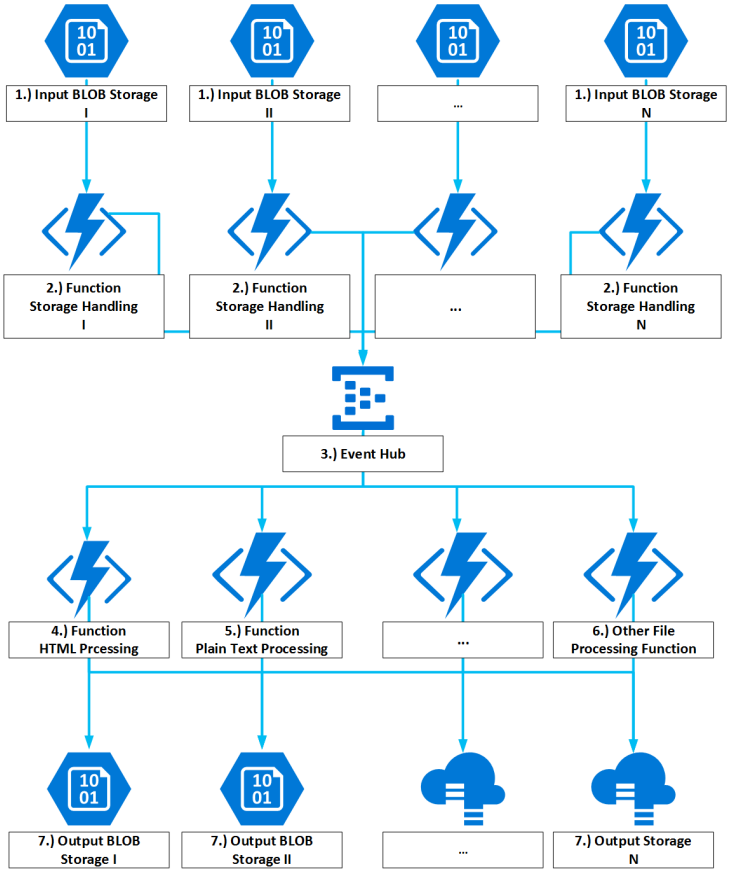

Figure 7: Real-time File Processing with Azure

1. Input files are stored in *Azure Blob Storage*.

[18] Source: `https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-what-is-event-hubs` (Last Access: 01.02.2017)

2. *Azure Functions* are managing the connectivity between the *Azure Blob Storage Container* and the *Azure Event Hub*. Additionally they create an output storage container at an early stage.

3. *Azure Event Hub* uses a many-to-many communication approach capable of managing connections to other devices or Azure services and processing messages with a high throughput.

4. *Azure Function* - HTML Processing: The function is triggered by the event hub, downloads a markdown file from the input *Azure Blob Storage* container, transforms it into an HTML file and stores it in the output *Azure Blob Storage* container.

5. *Azure Function* - Plain Text Processing: It is similar to the HTML Processing function but transforms a markdown file into a plain text file.

6. *Azure Function* - Other File Processing: It is possible to add other functions which implement additional functionality and act similar to the above mentioned processing functions.

7. Processed files are stored in Azure Blob Storage or maybe in another storage (such as a *Azure Table Storage*).

4 Implementation and Set-Up Instructions

The following chapter will briefly explain how to set up the *Real-Time File Processing Architecture* on *Microsoft Azure*. Certainly a *Microsoft Azure* account is needed. *Microsoft* provides a free trial account for 30 days after signing up. To set up the whole structure just follow the instructions stated below. The last part of the chapter describes also a short and easy way how to test the functionality.

4.1 Storage Accounts

Considering figure 7 the implemented design needs two input blob containers. To create these two containers, at least one *Azure Storage Account* is needed which holds the containers. The containers can also be divided into different accounts. So first create an *Azure Storage Account* in your *Azure* environment. Name it at will. The account kind can be set to *General purpose*, but since only blobs can be used for triggers, *Blob storage* would fit better. The encryption can be disabled, the performance can be set either to *standard* or *premium* at will. For the resource group the one has to be chosen, to which the other services will assigned to. After the storage account is created a container has to be created. Name it at will and set the public access level to *container*. The first container should be created by now.
Since the current implementation provides two different storage handling functions, two container should be provided which can trigger the functions. The second container can be added to the same account or into a new one. Finally, the processed output of the file processing functions has to be stored somewhere. Since the both storage handling functions will create a new container for the output automatically, the information for the destination storage accounts is needed. This can be the first one for all functions again, or different ones. Create as many accounts as wished.
After all at least on storage account with two containers should exist. For the current implementation, four accounts are provided for testing purposes: two input accounts named `input1` and `input2` which both hold one container called `input`, and two output accounts named `outputstorage1` and `outputstorag2`. But feel free to perform everything on one account.

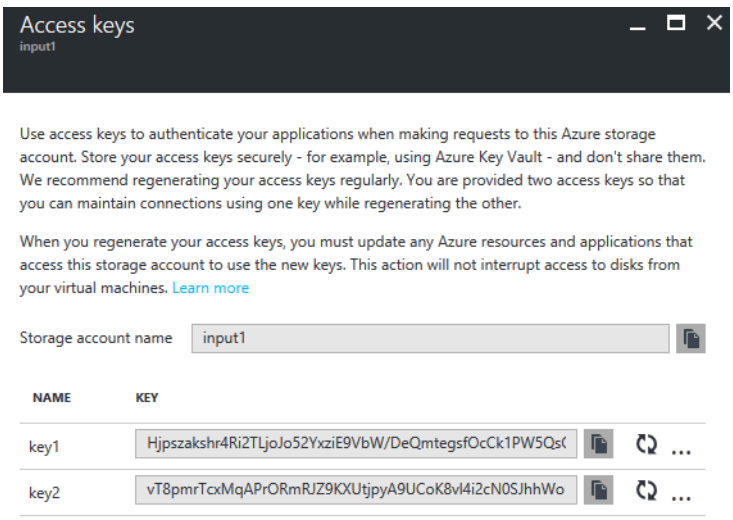

Figure 8: Storage Account Access Keys

If the accounts are created, the key(s) to access the accounts should be checked. They are needed in the functions to access the storage account(s). The key can be found in the storage account's settings (look for a key symbol) and should look like figure 8. By default two keys are provided. One is supposed to be used for internal services of the environment. The second one is supposed for external access. So if the

key for external access is changed to prevent further access, the internal services are not affected. So it is recommended to use the first key for every storage account.

4.2 Event Hub

It is recommended to create the *Azure Event Hub* before creating the functions because all the functions have to be bound to that hub. Create a namespace for the event hub and name it at will (e.g. *event-handling*). Afterwards set up one event hub and name it at will (e.g. *inputevent-hub*). Remember to use the same resource group as for the storage account(s) is used. The event hub will create an event for every message it receives. The file processing functions will be triggered by the events. Every event hub defines consumer groups. The subscribers to one consumer group are concurrent. If one subscriber is triggered by an event, the other subscribers of the same group will not be triggered. Since every file processing function should be executed if an event is created, each file processing function needs its one consumer group. So create two consumer and name it at will (e.g. *processing-1* and *processing-2*, see figure 9).

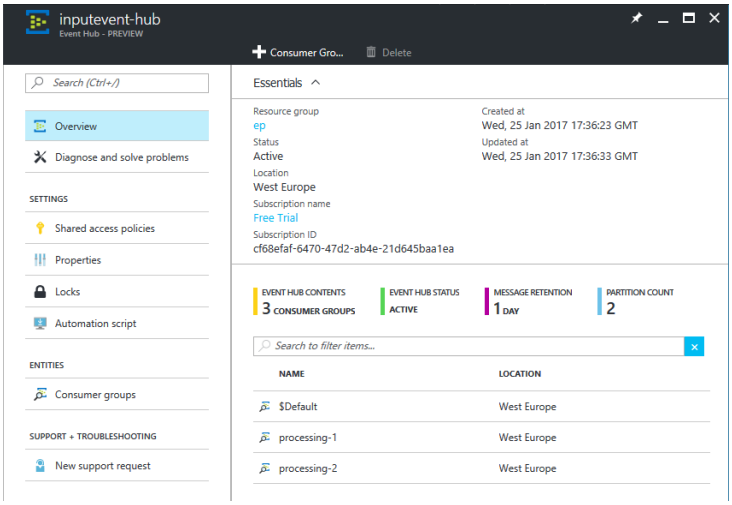

Figure 9: Event Hub

By now an event hub with two consumer groups should be created. At least check the connection string for the event hub which can be found in the event hub's namespace (look for the key symbol again). It is needed in the functions and should look like figure 10.

4.3 Storage Handling Functions

The *Storage Handling Functions* are the functions which are triggered, if a file is added to the input containers. One function for each container has to be set up:

1. **Create Function App**: A name for the function app has to be stated (e.g. *storage-handling*). Further the resource group, consumption plan and the location have to be chosen. This can be done at will but remember to use the same resource group as used for the storage accounts. Now a function app should be running.

2. **Create Function**: Open the function app and click on + *New Function*. The *BlobTrigger-JavaScipt* template is needed now. After choosing the template, a name (e.g. *Storagehandling1*), a path to the blob file which triggers this function and a connection to the storage account have to be stated. For the path `<container name>/{blobname}.{blobextension}` (e.g. `input/{blobname}.`

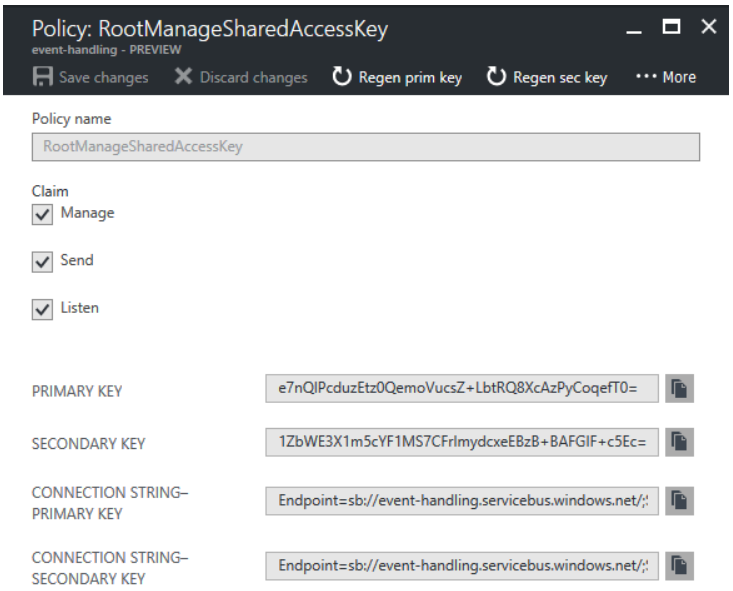

Figure 10: Connection String to Event Hub's Namespace

`{blobextension}`) should be taken. Taking the patterns `input/{blobname}.{blobextension}` allows the *JavaScript* code to access the blob's file name and the blob's file extension. For the connection click on the *new* button and select your first input storage account which has been created in the first step (section 4.1). The connection can be named at will. At this point a simple function is created which triggers for every blob file which is added to the bounded input container, and logs the blob's content.

3. **Binding the Event Hub**: Now the created event hub has to be bound in as output. Click on the new function in the function app and select *Integrate* afterwards. Click on *+ New Output* and select *Azure Event Hub*. For the template a parameter name for the message which will be sent to the event hub, a connection and an event hub have to be provided. The parameter name should be `outputEventHubMessage` to match the provided code below (see listing 4 line 38). The event hub's name is the name which has been given to the hub in the section 4.1 before. The connection can be named at will again, the connection string has to be the one of the event hub's namespace (see last section 4.1). (If this is the second function, just use the existing connection which is used for the first one). Now the function can send a message to the event hub after execution.

4. **Adjusting the Template**: Before integrating the code, a short look at the `function.json` file which contains the binding's template for the function, should be taken. It can be accessed by clicking on *Develop* in the function's menu. Afterwards, click on *View Files* in the upper right corner. The function's folder's content should be shown right-handed now. Open the `function.json` file. It should be look like the listing 3. Change the blob trigger's name to `inputblob` from `myBlob` since it is referenced that way in the code.

5. **Package Installation**: Before the provided function's code can work, some packages are needed. To install the packages for the function, open *Function app settings* in the bottom left corner of the function's menu. Afterwards select *Go to Kudu* which is a command line tool. Change the directory to `/site/wwwroot/<function name>` which should contain the `index.js` file with the function's code and the `function.json` file with the template. Here execute the command `npm install azure-storage` and wait, until the installation is finished. The package includes a library to access *Azure* blob storages which is needed.

```
{
  "bindings": [
    {
      "name": "inputblob",
      "type": "blobTrigger",
      "direction": "in",
      "path": "input/{blobname}.{blobextension}",
      "connection": "input1_STORAGE"
    },
    {
      "type": "eventHub",
      "name": "outputEventHubMessage",
      "path": "inputevent-hub",
      "connection": "event-handling_TRIGGER",
      "direction": "out"
    }
  ],
  "disabled": false
}
```

Listing 3: Azure Function - Storage Handling - Template

6. **Providing the Code**: At least click on *Develop* again, open the `index.js` file and copy the provided code below (see listing 4) into that file after everything else is deleted.
   As every *Azure Function* code, it provides the function `module.exports` which is executed if an instance of the *Azure Function* is invoked. The function gets `context` and `inputblob` as input parameters (see line 3). The first one is an object which ensures access to environment variables and resources and provides the logging. The second one is the object which contains the content of the blob file, which triggered this function. First, the code provides the information which is needed to download the input blob from its storage and to upload the processed data into the destination storage after file processing. The information is stored in the object `blobData` (see line 9), which has to be adjusted now (refer to the section 4.1 for the needed data):

   - `sourceStorageAccount`: This entry contains the name of the input storage account (e.g. *input1*).
   - `sourceKey`: This entry contains the access key to the input storage, see figure 8.
   - `sourceContainer`: This entry contains the input container name (e.g. *input*).
   - `blobName`: This entry contains the name of the blob file without file extension. With `context.bindingData.blobname` it is automatically set to the pattern described in point 2. above. Nothing has to be done here.
   - `blobExtension`: This entry contains blob file's extension. With `context.bindingData.blobextension` it is automatically set to the pattern described in point 2. above. Nothing has to be done here.
   - `destinationStorageAccount`: This entry contains the name of the output storage account (e.g. *outputstorage1*).
   - `destinationKey`: This entry contains the access key to the output storage.
   - `destinationContainer`: This entry contains the input container name. With `context.bindingData.blobname.toLowerCase()`, the container is named after the blob file's name without file extension and in lowercase letters since container names are not allowed to contain uppercase letters. If another container naming policy is whished, change it.

```
1  var azureStorage = require("azure-storage");
2
3  module.exports = function (context, inputblob) {
4    context.log("JavaScript blob trigger function processed blob
5                     \nName: ", context.bindingData.blobname,
6                    "\n Blob Data Type: ", context.bindingData.blobextension,
7                    "Blob Size: ", inputblob.length, "Bytes");
8
9    var blobData = {
10     sourceStorageAccount: "input1",
11     sourceKey: "<some key>",
12     sourceContainer:"input",
13     blobName: context.bindingData.blobname,
14     blobExtension: context.bindingData.blobextension,
15     destinationStorageAccount: "outputstorage1",
16     destinationKey: "<some key>",
17     destinationContainer: context.bindingData.blobname.toLowerCase()
18   }
19
20 var blobService = azureStorage.createBlobService(
21         blobData.destinationStorageAccount, blobData.destinationKey);
22   context.log("Create new container", blobData.destinationContainer,
23                    "if it does not exist")
24   blobService.createContainerIfNotExists(blobData.destinationContainer,
25         {publicAccessLevel : "container"},
26         function(err, result, response) {
27           if (err) {
28             context.log("Could not create container", containerName,
29                           "\nError:", err);
30             next("Could not create container " + containerName +
31                    "\nError: " + err);
32           }else{
33             context.log("Result: ", result);
34             context.log("Response: ", response);
35          }
36   });
37
38   var message = JSON.stringify(blobData);
39   context.log("Created message:\n", message);
40   context.bindings.outputEventHubMessage = message;
41   context.done();
42 };
```

Listing 4: Azure Function - Storage Handling

Afterwards the code creates an blob client instance to connect to the destination storage account (see lines 20 and 21) and creates the output container if it does not exist (see lines 24 to 36). In the end, the `blobData` object is converted into a JSON string and set as output message for the event hub (see line 38). The message is sent automatically afterwards since the event hub is integrated in the environment template `function.js`.

The described steps have also to be taken for the second storage handling function which is triggered by the second input container except that the same function app can be used. The values for the `blobData` object have to vary referring to the connection information for input and output. The rest stays basically the same. If further input storages and containers should be added, further storage handling functions can be added following the same scheme.

4.4 File Processing Functions

There are two file processing functions implemented. The first one creates an html-file from the input md-file, the other one creates an plain text txt-file): The file processing functions can be set up with the following steps:

1. **Create Function App**: A name for the function app has to be stated (e.g. *storage-handling*). Further the resource group, consumption plan and the location have to be chosen. This can be done at will but remember to use the same resource group as used for the storage accounts. Now a function app should be running.

2. **Create Function**: Open the function app and click on + *New Function*. The *EventHubTrigger-JavaScipt* template is needed now. After choosing a name (e.g. *ProcessToHTML*), the referring event hub's name and the connection to the event hub's namespace have to be stated. The name of the event hub is the one chosen in section 4.2. To set up the connection, the connection string of the event hub's namespace is needed (see figure 10). The connection can be named at will. If this is already the set up for the second file processing function, choose the connection which is already created.

3. **Adjusting the Template**: Before integrating the code, a short look at the `function.json` file which contains the binding's template for the function, should be taken. It can be accessed by clicking on *Develop* in the function's menu. Afterwards click on *View Files* in the upper right corner. The function's folder's content should be shown right-handed now. Open the `function.json` file. It should be look like the listing 5. Change the event hub trigger's name to `inputEventHubTrigger` since it is referenced that way in the code. Further change the consumer group to the one which has been created in section 4.2 (e.g. *processing-1*). The first file processing function gets the first consumer group, the second function the second group.

```
{
  "bindings": [
    {
      "type": "eventHubTrigger",
      "name": "inputEventHubTrigger",
      "direction": "in",
      "path": "inputevent-hub",
      "connection": "event-handling_TRIGGER",
      "consumerGroup": "processing-1"
    }
  ],
  "disabled": false
}
```

Listing 5: Azure Function - Storage Handling - Template

4. **Package Installation**: The file processing functions also need some packages. To install the packages for the function, open *Function app settings* in the bottom left corner of the function's menu again. Afterwards select *Go to Kudu* and change the directory to `/site/wwwroot/<function name>` For the html processing function install the packages `azure-storage`, `async` and `marked` with the `npm install` command. For the plain text processing function, install `azure-storage`, `async` and `remove-markdown` with the `npm install`.

5. **Providing the Code**: Finally the provided code can be integrated into the `index.js` file after deleting everything else (see figure 6). The code should work without further adjustments.

```
1  var azureStorage = require("azure-storage");
2  var marked = require("marked"); //only for html file processing
3  var removeMD = require("remove-markdown"); //only for plain text file processing
4  var asyn = require("async");
5
6  module.exports = function (context, inputEventHubTrigger) {
7
8    context.log("JavaScript eventhub trigger function processes work item",
9                    inputEventHubTrigger);
10
11   asyn.waterfall([
12
13   function downloadBlob(next){
14     //see listing below
15   },
16
17   function processContent(blobContent, next){
18     //see listing below
19   },
20
21   function uploadBlob(processedContent, next){
22     //see listing below
23   }
24
25   ], function(err){
26        if(err){
27          context.log("Waterfall execution error:", err);
28        }else{
29          context.log("Waterfall execution succeeded.");
30        }
31   });
32   context.done();
33 };
```

Listing 6: Azure Function - File Processing Function

The code contains mainly of three functions `downloadBlob` (see line 13), `processContent` (see line 17) and `uploadBlob` (see line 21) which are executed with `async.waterfall` due to their *callback* structure. The actual function's code can be found in the listings 7 to 10. The main structure from listing 6 is used for both file processing functions. Only the required packages vary for `marked` for the `html` processing and `remove-markdown` for the plain text processing. The content of the event hub's message is passed to the module.exports function already as `inputEventHubTrigger` parameter.

- `downloadBlob`: This function simply downloads the blob from its source container. First a blob client is created which extracts the connection information to the input storage account from the message and connects to the account (see lines 4 to 6 of listing 7). Afterwards the blob's name is extracted from the message (see line 7) and the blob is finally downloaded (see line 11). If the download succeeded, the blob's content is passed to the callback function (see line 17). Otherwise it passes an error message and the waterfall execution is stopped (see line 21).

```
1    function downloadBlob(next){
2      context.log("Create blob service to storage account",
3                      inputEventHubTrigger.sourceStorageAccount);
4      var blobServiceIn = azureStorage.createBlobService(
```

```
5                     inputEventHubTrigger.sourceStorageAccount,
6                     inputEventHubTrigger.sourceKey);
7     var blobName = inputEventHubTrigger.blobName + "." +
8                     inputEventHubTrigger.blobExtension;
9     context.log("Download blob", blobName, "from",
10                    inputEventHubTrigger.sourceContainer);
11    blobServiceIn.getBlobToText(
12                    inputEventHubTrigger.sourceContainer,
13                    blobName,
14                    function(err, blobContent, blob){
15                      if(!err){
16                        context.log("Sucessfully downloaded blob ", blobName);
17                        next(null, blobContent);
18                      }else{
19                        context.log("Could not download blob", blobName,
20                                        "\nError:", err);
21                        next("Could not download blob");
22                     }
23    });
24  }
```

Listing 7: downloadBlob Function

- `processContent`: This function is rather simple. It receives the input blob's content as input parameter (see listings 8 and 9 line 1), processes it to `html` (see figure 8 line 5) or to plain text (see figure 9 line 5) and passes the new content string to the callback function (see line 5).

```
1   function processContent(blobContent, next){
2     context.log("Processing blob");
3     var processedContent = marked(blobContent);
4     context.log("Processed Content:\n", processedContent);
5     next(null, processedContent);
6   }
```

Listing 8: processContent Function - HTML

```
1   function processContent(blobContent, next){
2     context.log("Processing blob");
3     var processedContent = removeMD(blobContent);
4     context.log("Processed Content:\n", processedContent);
5     next(null, processedContent);
6   }
```

Listing 9: processContent Function - Plain Text

- `uploadBlob`: This function uploads the processed content into a new blob into the destination container. First a blob client is created which extracts the connection information from the message and connects to the output storage account (see lines 4 to 6 of listing 10). Next it creates a string for the new blob, which consists of the original blob's name and the new file extension `html` or `txt` (see line 7 for `html` processing, otherwise the line would include `"txt"` instead of `"html"`). Finally the content is uploaded into a new blob into the output container (see lines 10 to 20).

```
1   function uploadBlob(processedContent, next){
2     context.log("Create blob service to storage account",
```

```
                    inputEventHubTrigger.destinationStorageAccount);
    var blobServiceOut = azureStorage.createBlobService(
                    inputEventHubTrigger.destinationStorageAccount,
                    inputEventHubTrigger.destinationKey);
    var newBlobName = inputEventHubTrigger.blobName + "." + "html";
    context.log("Uploading new blob", newBlobName,
                    "to container", inputEventHubTrigger.destinationContainer)
    blobServiceOut.createBlockBlobFromText(
                    inputEventHubTrigger.destinationContainer,
                    newBlobName,
                    processedContent,
                    function(err, result, response){
                      if(err){
                        context.log("Blob could not be created:", err);
                        next("Blob could not be created:" + err);
                      }else{
                        context.log("Blob is uploaded successfully.");
                        next(null);
                      }
    });
  }
```

Listing 10: uploadBlob Function

As already mentioned, both file processing functions follow the same scheme and can set up with the stated steps above. The only differences (next to *Azure Functions'* namings) are the different packages which have to be installed in step 4, and one line of code in each function `processContent` (the actual processing) and `uploadBlob` (the blob's file extension).

4.5 Testing

After taking all the steps until here, the complete architecture should be set up and working. The setup on the *Azure Dashboard* should look like figure 11 (if the services are pinned to the dashboard).

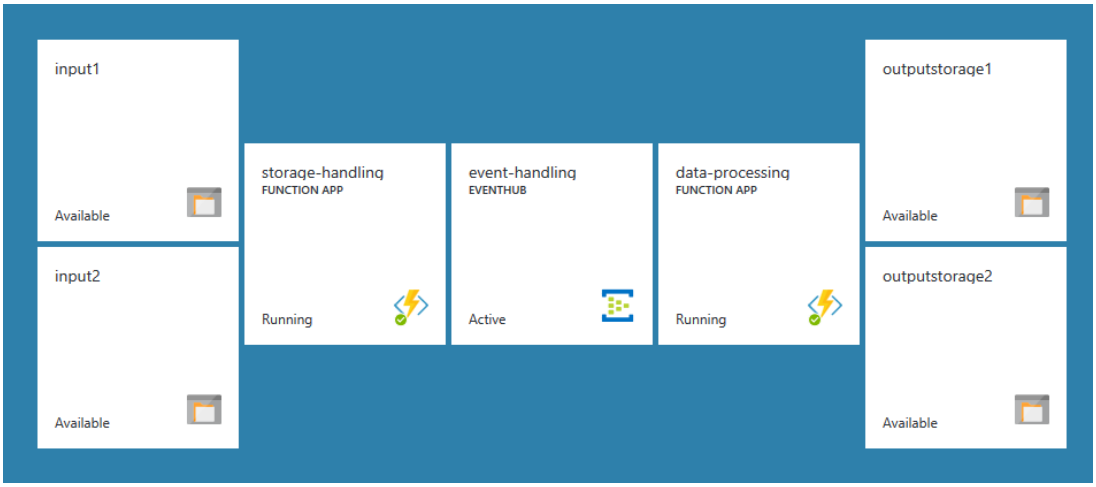

Figure 11: Setup on the Azure Dashboard

4.5.1 Testing Own Setup

To test the whole architecture open your first (or second) input container and upload a file. As input file, one of the README md-files of the *AWS reference architecture* can be used which are provided in the referring GitHub repository[19], e.g. README-DE.md (see figure 12). Use *Block blob* for the *Block type*.

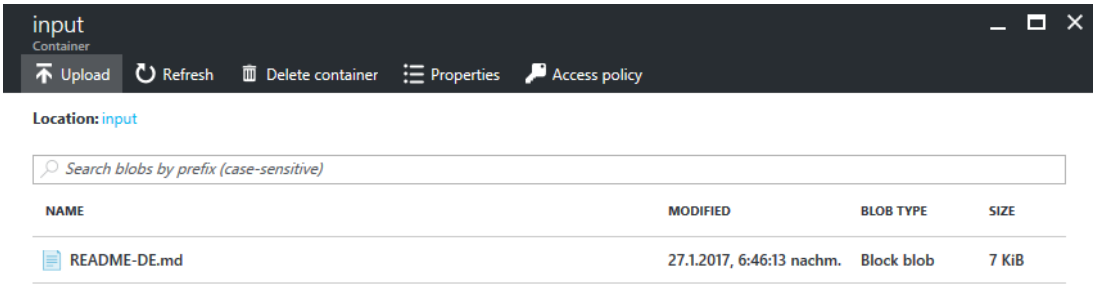

Figure 12: Upload MD-File into Input Container

Now be patient and wait for several minutes. Triggering the storage handling function, sending the message to the event hub, triggering the file processing function and processing the file take some time. Then open the specified output storage account and check for the referring output container. It should be named after the input blob file without the file extension and only using lowercase letters. If it is not there, just be patient and do not forget to refresh. After at most 10 minutes (mostly 5 minutes should be enough), the output should be there. The triggering of the storage handling function by the input storage takes the most time. The outcome should look like figure 13.

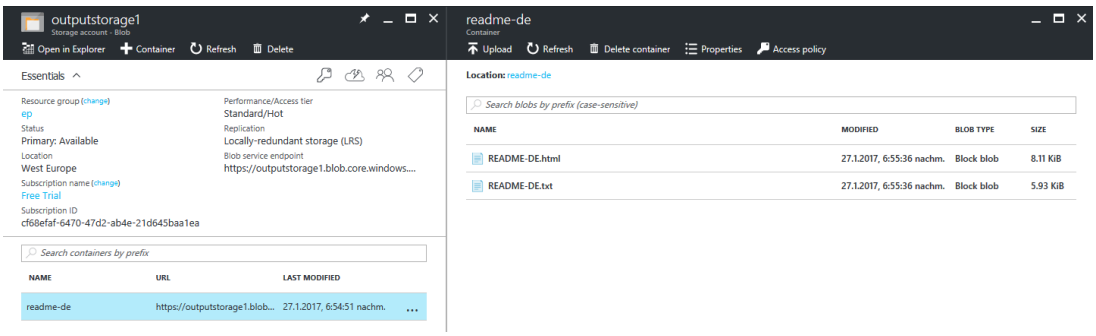

Figure 13: File Processing Output

If the container is up (and did not exist before), the storage function has been executed and the file processing function is still executing or failed. If no container is up, already the storage handling function failed or is still executing. To find the issue, just open the referring function and select *Monitor* left-handed. There should be an entry for an failed or still running function instance. The logs can also be checked here. If the function is still running, select *Develop* left-handed and activate the log stream in the upper right corner. Any issues should be stated here. But the setup is tested and worked for the authors. Finally check the output files after downloading them from the output container. They should contain a `html` and a plain text version of the original md-file.

[19] `https://s3.amazonaws.com/awslambda-reference-architectures/file-processing/lambda-refarch-fileprocessing.pdf`

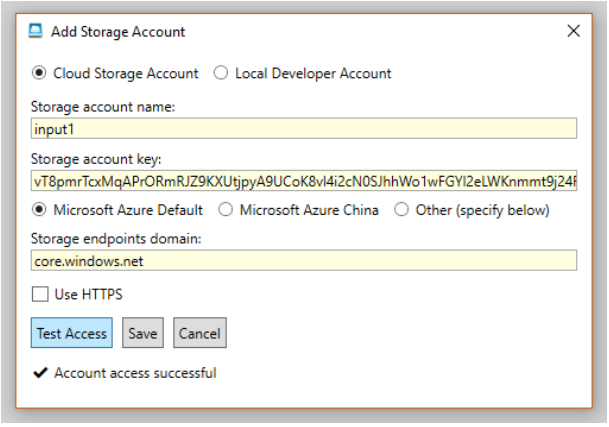

Figure 14: Azure Explorer 6 - Add Storage Account

4.5.2 Testing Authors' Setup

To test the currently running version of the authors, an *Azure Storage Client* can be used, e.g. the *Azure Storage Explorer 6*.[20] Just download the explorer and start it. Afterwards add a new account for the input container and an account for the output container (see figure 14). Storage account names and keys can be found in the provided GitHub repository in the *Access.txt*. The setup should be accessible for some days after the presentation (before it is shut down).

After adding the accounts, upload a md-file into the input container (see figure 15).

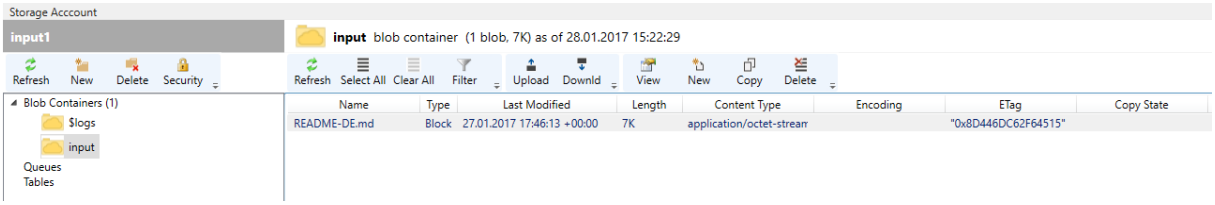

Figure 15: Azure Explorer 6 - Upload File

Afterwards be patient again and wait for several minutes (mostly 5 should be enough again, at maximum 10 minutes are needed). Then open the output container and check for the outcome. The *Azure Storage Explorer* does not refresh automatically the container's content so do not forget to refresh before checking. The outcome should look lie figure 16.

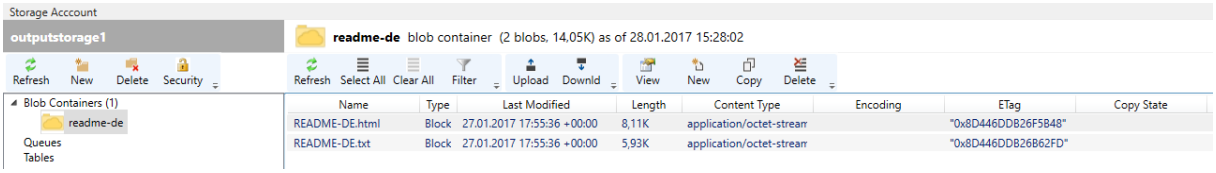

Figure 16: Azure Explorer 6 - Output

[20] `https://azurestorageexplorer.codeplex.com/`

5 Review

This chapter performs a review on both the implementation and the design. For the implementation section, some simple suggestions are made to improve the fault tolerance and the implementation itself of the setup. In the design section, the complete architecture design and used services are discussed for general issues and advantages or disadvantages in comparison to the *AWS reference architecture*.

5.1 Design Approach

In the following it is pointed out what design aspects we chose to borrow and what aspects we chose to revise from the lambda reference regarding the decision for using and connecting the different services and especially the usage of the Event Hub.

Theoretically, we could have designed and implemented a much easier approach for the given scenario. Using a Blob Storage which would trigger the processing functions which would in turn save the output to the same or another Blob Storage. But since we wanted to keep the basic concept of the reference architecture and only change it selectively or eventually even enhance, a more convenient draft was chosen.

- **Input Storages**: The general concept of using just one input storage service in the architecture was transferred to our design. The decision of the possibility of using several storage containers instead of just one (what would also be perfectly fine) was made as a structural scheme for the input data possibly on a per user level. This approach could alternatively be implemented using the container permission "Public read access for blobs only". Since Azure Storages autoscale there would be no performance difference between these two variations expected in a heavy load scenario.

  The advantage of this approach, assuming a viable quantity of containers, is having a better overview of the container management and eventually varying container permissions for each container. A static approach was used but it could also be implemented as a dynamic solution to manage containers on the fly. The disadvantage regarding the static approach however is that every single input storage has to be spawned and bound to a storage handling function individually.

- **Storage Handling Functions**: The reference architecture shows a direct connection between the *S3* and the *SNS* service. Since we could not manage to establish an equivalent connection between the Azure Storage and the Event Hub we managed to create Azure Functions we name storage handling functions. Similar to the static approach regarding the input storages we used a declarative binding of the storage handling function to a particular not already bound input storage. This was necessary because every function can only have one input trigger. The whole scenario could also be implemented dynamically at runtime through imperative binding.

  The second reason for creating storage handling functions was the functionality to create a corresponding output storage container on the fly in an early stage. In our tests the Azure platform took some time to create an output container so after the markdown file conversions were finished the file processing function had to wait for the container to spawn. That way we managed to overcome this issue resulting in less errors. And additionally it ensures that there is only one attempt to create an output container instead of one attempt per file processing function.

  But still there is a problem with the storage handling functions, or to be more precise: with the Blob Storage trigger. The trigger fires the storage handling function within one (or at most two minutes) after a new blob is added to the container. But this happens only as long as the

Function App blade in the Azure portal is opened. As soon as the *Function App* or even the complete *Azure* portal is closed in the browser, the *Function App* is somehow spinned down and it takes much more time until the storage handling function is triggered (up to 20 minutes). This is a problem wich several users have faced and on which Microsoft seems to be currently working on.[21] Since the bug concerns the core WebApp's infrastructure, it may still take several weeks or months to fix it and to integrate it into Azure's environment. Until then the blob trigger cannot fit any real-time scenarios. To bypass this problem a further time-triggered function can be added to the function app which only triggers to keep the function app alive (and to do nothing afterwards). But regarding costs which are calculated for function invocations (and run-time which would not be that important here), this would be unnecessarily expensive. It also does not match the idea of a real-time scenario. If this problem has to be solved and the fix is not delivered in time, the time-triggered function would be a solution next to changing the blob trigger into an storage queue approach and avoiding blob triggers at all.

- **Event Hub**: The most purposely variation of the reference architecture was made utilizing the Azure Event Hub. Figuring out a heavily loaded system we identified the messaging component as the one most probably being the bottleneck of our architecture. Therefore, we integrated the Event Hub as the one messaging service of the Azure platform, which has the best performance for the task but not too many features on top which could result in excessive costs. An alternative would have been the Azure IoT Hub which can additionally communicate bidirectionally. Our design did not need this feature and furthermore in a theoretical workload scenario of 1000 events per second over one month the costs could roughly exceed the costs of an Event Hub by a factor of ten.[22]
  Bearing these reasons in mind as well as the technical advantages of the *Event Hub* over the other Azure services mentioned in chapter 3.2 there was also a design decision element as well. With the decision of using more than one input storage and thus more than one storage handling function the messaging component of our architecture had to be a many-to-many service. This was a conscious amendment and not resulting from something which was not possible or we could not manage to achieve equivalent to the reference architecture as this was the case by introducing the storage handling functions.

- **File Processing Functions**: The two implemented File Processing Functions are equivalent to the lambda functions in the reference architecture. After invoking the function the markdown file is downloaded, transformed (depending on the function into a `html` or plain text file) and uploaded into an Output Storage.

- **Output Storages**: The Output Storages represent the equivalent to the *S3* storage of the reference architecture and are the corresponding element to the Input Storages. In the output storage of the lambda architecture there is moreover a *DynamoDB* which was not used in the implementation. This is the reason we did not map it into our own design. However it would not be difficult to integrate a database as an additional Output Storage into our design.

- **Real-Time Aspects**: Real-time in terms of data processing can be defined in terms of time constraints, concurrency and data flow.[23] Since the time constraint component can vary a lot between different use cases we could not state an explicit time limit to call the architecture convenient for real-time file processing scenarios. The concurrency aspect is fulfilled in both the lambda reference architecture and our design by using a publish/subscribe push model respectively an event driven

[21] Source: `http://martinabbott.azurewebsites.net/2017/01/07/azure-functions-blob-triggers-and-timers/` and `https://github.com/Azure/azure-webjobs-sdk-script/issues/691`
[22] Source: `https://azure.microsoft.com/de-de/pricing/calculator`(Prices can vary) (Last Access: 01.02.2017)
[23] Source: J.A. De La Puente, L. Boullart (1992): Real-Time Programming

pattern. Regarding the data flow component we would see both designs as not real-time architectures. The Amazon architecture indeed uses a push based mechanism to pass messages from *SNS* to the *AWS lambda functions* but the functions then after being invoked download the markdown file on their own. We decided to implement it the same way as the reference architecture does. However, regarding this aspect of a real-time scenario we could have also passed the blob content (instead of only the blob download location) as the message payload along the message chain to the processing functions which would in our opinion match the data flow criterion of a real-time architecture. (The current blob trigger bug is not considered here. As long as the blob trigger bug is not fixed, there is no chance to fit this approach in any real-time scenario.) But then it has to be kept in mind, that the relevant limitation for the event hub could be the 1 MB ingress (or 2MB egress) at the maximum instead of 1000 messages at the maximum per second (and per Throughput Unit) which is the relevant limitation criteria at the moment (see the part about the *Azure Event Hub* in section 3.2).

5.2 Implementation

First of all the implementation works and does what it is supposed to do. Several users can create their own storage accounts and bind it into the structure with a referring storage handling function. Afterwards their files will be processed. But there are certainly improvements for the implementation. Amongst others the following points can be considered:

- **Provided File Types**: At the moment only markdown files (md files) can be handled by the file processing in a useful way. It could be considered to make a file type check in the storage handling functions by checking the file extension for example. This would ensure that invalid file types will not be processed.
- **Input/Output Configuration**: The current implementation provides several input containers. The idea behind this design decision is to allow different users to use the architecture. Each user can access his own input and output storages without having to share the accounts with other users. Since it could be possibly better to not let the users access the storage handling functions if they want to change their output storages (or output container configuration), the configuration for the output storage could be outsourced to the input storage. At the moment the needed information for the output storage is hard-coded in the storage handling functions. Instead a configuration file, e.g. a simple JSON file, could be provided in the input storage container. Then the storage function would download that to get the information for the output storage and the user could modify it without accessing the function's code.
- **Retry policies**: If a functions creates a container via a blob storage client, the request will be responded to be successful handled before the container is actually accessible. It takes a little amount of time before the container is accessible after creation. Due to this some upload request may fail if they are sent too shortly after container creation. To avoid such failures some retry rules could be implemented so that the file processing functions would retry an upload for a defined amount of times with a timeout between the retries. Further retry rules can be considered for downloads also and so on. Further failures can be occur if different functions want to create a same named container in the same storage account. Thus the storage handling functions are creating the needed output containers instead of the file processing functions. But if different storage handling functions refer to the same output storage this can still be a problem. Retry rules could solve this also since a container would not be created in a second attempt, if it exists then.
- **Further Output Storages**: Further output storage types such as databases can be added. Therefor the existing functions have to be modified. First the storage handling function has to know the

output storage type and provide further information in the sent message. An extra entry of storage type should be added and then the necessary connection information. Then a fitting equivalent for the functionality which creates a new container, has to be added. Further, the upload function on the file processing functions as to be extended for the new storage type. How this is done depends on the storage types and has to be checked in the particular case.

List of Figures

Listings

References

[BBNN15] Baird, Andrew ; Buliani, Stefano ; Nagrani, Vyom ; Nair, Ajay: AWS Serverless Multi-Tier Architectures - AWS Whitepaper. (2015). – Online; accessed 02-January-2017

[Rob16] Roberts, Mike: *Serverless Architectures*. `http://martinfowler.com/articles/serverless.html`, 2016. – Online; accessed 02-January-2017

[Vog16] Vogels, Werner: *AWS Serverless Multi-Tier Architectures*. `http://www.allthingsdistributed.com/2016/06/aws-lambda-serverless-reference-architectures.html`, 2016. – Online; accessed 02-January-2017

Content of the GitHub Directory

For this project a *GitHub* page is established which you can reach with:
`https://github.com/ortaluis/EC-Real-time-File-Processing-Microsoft-Azure`
In the repository you can find the following content:

- **The Azure Functions' Code**: For each implemented *Azure Function* (two storage handling functions *Input1Handler* and *Input2Handler* and two file processing functions *ProcessToHtml* and *ProcessToPlainText*), a folder is provided. The folder contains the referring *index.js* and *function.json* files (see sections 4.3 and 4.4).
- **Azure Storage Account Access**: To access an *Azure Storage Account*, the storage account's name and the referring access key(s) have to be used (see chapter 4). The referring access information for the input and output storages of the authors' implementation can be found in the *StorageAccessKeys.txt* file.
- **Documentation**: This documentation can be found as *Documentation.pdf* file.

(Honest) Developer's Diary

First Week of January

- **Group meeting**: On the meeting without Daniel, important decisions are made and a general agenda is set up in a Dropbox document. The definitive goal for this assignment: Getting 9 points. Will it be difficult? Yes, nobody has any clue, what the reference implementations are doing ... Is it achievable? DEFINITELY NOT ENOUGH TIME.
- **Christof Schubert**: Christof is a silent working animal. He does not talk much, but sometimes new resources on the assignment appear in the shared Dropbox folder ...
  Spent Hours: 4 hours on research
- **Daniel Andika Steinhaus**: Daniel is still on vacation after New Year. He refuses to do anything but promises to pay back later. Nobody is sure whether to believe this or not ...
  Spent Hours: 0, not even reachable
- **Diogenis Lazaridis**: Diogenis is somehow the motivator for the group. He always urges to remember the goal (9 points to everybody) and the time which is left (DEFINITELY NOT ENOUGH).
  Spent Hours: 3 hours on research
- **Luis Orta Hernández**: Luis is the group's oasis of tranquility. He does not talk much. But if he does it is relevant all the time.
  Spent Hours: 3 hours on research

Second Week of January

- **Group meeting**: To be honest, everybody is chilling. Meeting is a waste of time. Daniel who has not done any research yet, is arguing about everything.
- **Christof Schubert**: Christof simply does the review for the second architecture with Daniel. He already guesses that this is the one which will be implemented, but does not talk about this ...
  Spent Hours: 4 hours on the review

- **Daniel Andika Steinhaus**: Daniel likes nice Latex papers although everybody would be satisfied with a default MS Word document. It does not make sense to argue about that. Thus Daniel has to set up the documentation layout. Since it was difficult to agree on what *serverless* and *stateless* really means in the given context during the second week's meeting, he also writes down a waste introduction chapter on general terms. There is also no sense in arguing about that with him.
  Spent hours: 30 minutes on the Latex Layout (which is basically the layout of his Bachelor thesis), two hours on the waste introduction chapter, 3 hours on the review of the second reference architecture (with Christof)
- **Diogenis Lazaridis**: Diogenis reviews the first reference architecture with Luis.
  Spent Hours: 3 hours on research, 2 hours on writing down the review
- **Luis Orta Hernández**: Luis reviews the first reference architecture.
  Spent Hours: 3 hours on research, 2 hours on writing down the review

Third Week of January

- **Group meeting**: Diogenis mentions that there is not much time left for the assignment. He is probably right. So the decision for the architecture which should be implemented, has to be taken wisely. After everybody has understood the implementations at that time, Daniel proposes an easy indicator: the number of lines in the *Amazon Cloud Formation* template. Since the functions' Javascript codes can somehow be duplicated on any new platform, the template gives an idea on how complicate it is to set up everything. The second architecture is implemented (template is half the size of the first one). The choice of the platform for the implementation is even more easy. The *AWS Lambda* function's equivalents on BlueMix or Google Cloud Services have beta or even alpha status. There is definitely not enough time to be a beta or even alpha tester. Microsoft Azure is chosen.
- **Christof Schubert**: Christof is the new account manager since he is the only one with the credit card (which is needed to sign up on Microsoft Azure). He stays for a weekend with Daniel to get everything implemented. Daniel gets co-administrative access to the account.
  Spent Hours: 3 hours on signing up on Azure, 18 hours on setting up the architecture on Azure (thereof half the time crying about the terrible documentation of simply everything on Azure ...)
- **Daniel Andika Steinhaus**: Daniel locks up Christof at his home and feeding him (and himself) with (double Espresso) coffee all the time. An all around wasted coding session feeling arises which is actually strange, because setting up everything on Azure is half the time more like clicking buttons (and watching damn load beams) than really coding.
  Spent Hours: 3 hours on laughing about Christof, while he tries to sign up on Azure, 18 hours on setting up the architecture on Azure (thereof half the time crying about the terrible documentation of simply everything on Azure ...)
- **Diogenis Lazaridis**: Diogenis does not want to be useless. So he is allowed to read through the Azure documentation to find solutions for different problems. After some time, he figures out that he is abused by Daniel and Christof, and decides to work on the Documentation instead. Three people coding in two small *JavaScript* files does not make much sense anyway ...
  Spent Hours: about 10 hours on Skype with Daniel and Christof for the implementation to help with researching the documentation, (1 hour of frustrated arguing about not being the documentation idiot), 4 hours on the design chapter for the second architecture
- **Luis Orta Hernández**: Since everybody else is too busy with Azure, brave knight Luis faces *Google Cloud Services* lonely. He wears his shiny armor of tranquility and stands confidently on his

own battlefield, looking at the enemy's alpha services and lacking implementations ...
Spent Hours: 12 hours on designing the first architecture approach

Fourth Week of January

- **Group meeting**: The group meeting is spent to go through the complete implementation and to discuss what is missing. Since the set up already runs on Azure (nobody could really believe that), Diogenis does not mention anything about any time left.

- **Christof Schubert**: Christof is also the MS Visio whisperer. He creates a nice diagram for the implementation's architecture with the original(!) Azure icons which are also used on Microsoft's marketing web pages.
Spent Hours: ?? hours on the diagram (magician's secret)

- **Daniel Andika Steinhaus**: Daniel has decided to not trying to use the *Azure Resource Manager* (which can be considered to be an equivalent for *AWS Cloud Formation*). The documentation is terrible. So he writes down a description on how to set up everything up manually on Azure. He talks with nobody about his true intention of this: Giving an set up description instead of a template minimizes the chance that somebody is actually setting the whole architecture up, especially if the description is too accurate with over ten pages ...
Spent Hours: 5 hours on writing a description on how to set everything up (and rechecking everything described at least five times nonetheless, because Daniel is a pussy)

- **Diogenis Lazaridis**: Since Diogenis has left the coding session early, he has to write down the documentation on the design approach for the implementation and a big part of the implementation's review. He is mourning about something like DEFINITELY NOT ENOUGH TIME and UNFAIR, but nobody is listening.
Spent Hours: 7 hours on the documentation.

- **Luis Orta Hernández**: Luis is writing down the design approach for the *Google Cloud Services*. He also provides a nice diagram with the original(!) Google icons freshly from their marketing pages. But he is no MS Visio whisperer as Christof, only PowerPoint ...
Spent Hours: 3 hours on writing down the documentation, 1 hour on creating the diagram (no magic here)

First Week of February

- **Group meeting**: The group meeting is actually a lot of fun. Since all the real work is done, only one small thing is left: The question about who did what (remember: The goal is 9 points for everyone.) Result of the meeting is this absolutely honest developer's diary. We agree that everybody has accomplished his work.

- **Christof Schubert**: Christof has only reviewed the documentation having a lot of fun.
Spent Hours: 3 hours on reviewing the documentaion

- **Daniel Andika Steinhaus**: Daniel does the presentation slides since there also exists some layout of his Bachelor thesis presentation.
Spent Hours: 5 hours on reviewing and correcting the documentation

- **Diogenis Lazaridis**: In the end, he gives the presentation since his business English is outstanding.
Spent Hours: 3 hours on reviewing the documentation

- **Luis Orta Hernández**: Luis gives the first part of the presentation because he is the knight who killed *Google Cloud Services* on his own. Further he made the absolutely exciting video on which the functionality is shown. Be tuned for 3 minutes of suspense and (refresh) action. Third he has set up the GitHub directory. Nobody ever used it, but it is somehow required.
  Spent Hours: 5 hours on the video (`Exclusive Director's Cut - Blob Trigger Bug Fix Edition`), 3 hours on reviewing the documentation, 30 minutes on the GitHub repository