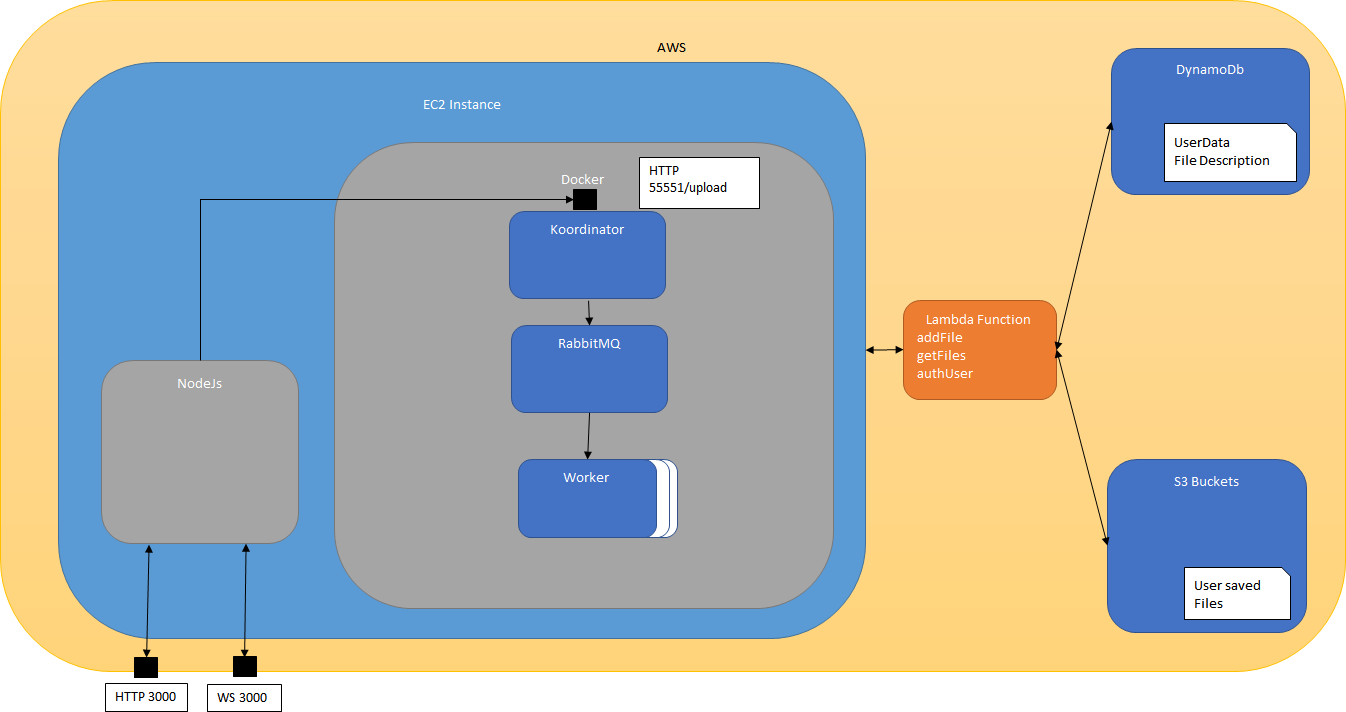
**Architecture Description**



**NodeJS Server:**

Simple Https Server:

* Express for Routing
* Pug for Html Page Templates
* Several NPM Modules (Session, FileUpload, Websocket)
* Running on Port 3000

Simple Secure Websocket Server for Real Time Data display to the Client

The Main Routes are:

* Index: Welcome Page with the options to login yourself. Furthermore, there is a Testing button to path index/test for testing purposes
* Main:
  + Site with all uploaded Files the logged in User has
  + User can upload new Files
  + User can Calculate existing files

**Lambda Functions:**

The used Lambda functions are mainly used to interact with the DynamoDb and the S3 Buckets. Each Lambda Functions uses the Same Role “WildRydesLambda” (We never changed the name) with the Full Access to DynamoDb and S3

Each Lambda Function can be invoked with the corresponding Lamda Libs. For Javascript it is the AWS.Lambda Lib. Each Invokation needs params (Javascript)

*var params = {*

*FunctionName: "authUser",*

*InvocationType: "RequestResponse",*

*LogType: "Tail",*

*Payload: '{ "param1Name" : "' + param1 +'", "param2Name" : "' + param2 + '"}'*

*};*

Method: authUser

Params: username, password

Response: on Success: 200

Response on Failure: 500, 403 (Wrong Credentials), 400 (username or password not set

Response Body: None

Description: Takes the username and querys DynamoDb. Checks if the password in the Db is the same as the parameter password if user is found. Returns 200 if it succeeded.

Method: addFile

Params: username, file(file.txt), wordCount (optional = 0), content(file content, optional = undefined), finish (optional = false)

Response on Success: 200

Response on Failure: 500 (Error while query), 400 (Params not set)

Response Body: Error Code or All Clear

Description: Takes the filename and filecontent from a user. If the file doesn’t exist from the user it saves a new map entry for the user and sets the wordcount to it. Furthermore the content is saved into a s3 bucket with the name user-text-files. Filename in Buckets is always username\_filename. If the file exists it gets overwritten.

Method: getFiles

Params: username

Response on Success: 200

Response on Failure: 500 (Error while query), 400 (Params not set)

Response Body: DynamoDb Data Items as a nested JSON Object with the calculated file features. They key of each object is the name of the file

Description: Querys the DynamoDb for all Userfile

Method: getFileByFileName

Params: username, file

Response: on Success: 200

Response on Failure: 500 (Query Failure), 400 (Wrong parameters)

Response Body: Json with Data Buffer of the files content

Description: Returns a Data Buffer from requested File.

Method: getFileCalcData

Params: username, file

Response: on Success: 200

Response on Failure: 500 (Query Failure), 400 (Wrong parameters)

Response Body: Json with Payload.Body with all the calculated fields, for now its

* WordCount
* finished

Description: Retrieves up to date Data from the DynamoDB

**Koordinator**

The coordinator is a docker container which acts as the “control service” of the system.

* Running on Port 55551
* Using RabbitMQ as messaging service
* Written in Python, achieved with flask

The main routes are:

* Upload: It gets a json from the NodeJS Server with the right credentials to use the getFileByFileName Lambda function to get the .txt file it should use. This file gets split line wise and sent to the RabbitMQ queue from which the workers consume.

**Worker:**

The workers are the consumers from the queue and just count the words from the strings they receive.

* Running on port 55553
* Using RabbitMQ as messaging service
* Written in Pyhton

**Fault tolerance**

We split up the whole task to separated applications/services. For each task we have an own service, e.g. the NodeJS server for uploading, coordinator for queueing and so on. The files and credentials we are using are saved to DynamoDb as well in S3 Buckets in case they get lost on the way or other problems.

We don’t have a load balancer since we were not allowed to create one, therefore we don’t have any state in the server. So we have only one instance on which everything runs, and if this one fails everything comes to halt, which is the main weakness of our system.

But still if something gets cancelled we still have our date saved in the buckets and in the DynamoDb, so nothing is lost and the user can continue to work with the already uploaded files. Also, if the user closes the application it continues to run to the end and gets loaded again if he re-joins the session.

So basically, if the instances crashes or some other problem occurs that the system crashes, we have to start the computation anew, but the state of the files is not corrupted while reloading the system.

**Testing the NodeJS and Lambda Functions:**

Use the provided URL of the instance to connect to the Server

Login with one of the following parameters:

* richard, mdk4
* Bernhard, mdk5
* arthur, mdk6

You should now see a list of files. There are 3 files for richard, 1 file for Arthur and 0 files for Bernhard.

Upload a new .txt file by using the “Datei auswählen” Button.

Reload the page after finishing and you see your uploaded file

Press Calculating to start the calculation

If the words get count you will receive new wordCOunts for your file until its finish

After its finish you can go back to the main side and see your new word Count

Optional:

If the Middleware is not counting the words correctly you can navigate to the DynamoDb and change the