Hochschule Worms
Fachbereich Informatik
Studiengang Angewandte Informatik B.Sc

Dokumentation zum Alexa Skill "DSA Telegram"

vorgelegt von
Chiara Police und Nico Vogel

Gutachter:	Prof. Dr. Stephan Kurpjuweit
Bearbeitungszeitraum:	Wintersemester 2020/21
Abgabedatum:	28. Februar 2021

Inhaltsverzeichnis

Skill Idee / Beschreibung

Mit dem Ende der ersten Vorlesung haben wir mit dem Brainstorming für eine Skill-Idee begonnen und konnten uns relativ schnell auf eine Art Verbindung zwischen Alexa und dem Messenger Telegram einigen.

Abgesehen von den negativen Schlagzeilen über manche Nutzergruppen des anonymen Nachrichtendienstes bietet Telegram mit seiner API vielerlei Möglichkeiten - die wir dementsprechend nutzen konnten.

Somit war unsere Idee also mittels Alexa Nachrichten in Telegram versenden zu können und gleichzeitig aber auch Antworten zu empfangen. Wenn man gerade keine Möglichkeit hat sein Handy zu nutzen, zum Beispiel weil der Akku leer ist oder man es gerade nicht finden kann ist man auf diesem Wege immer noch erreichbar.

Erstellung erster Grundlagen

Zur Einfindung in die Alexa Entwicklungsumgebung war es einerseits wichtig die Vorlesung aktiv zu verfolgen, andererseits aber auch nötig sich selbst in gewisse Themenbereiche einzuarbeiten. Gleiches gilt dementsprechend auch für die API von Telegram, wobei da bereits Grundkenntnisse vorhanden waren.

Zu Beginn fingen wir an mit dem Beispiel aus der Vorlesung erste Tests durchzuführen, um sich näher an Node.js (bzw. JavaScript allgemein) als Programmiersprache und Lambda als Entwicklungsumgebung heranzutasten.

Im Laufe der Zeit sind wir dazu übergegangen die eigentliche Skill-Funktionalität, hinter den Intents, durch erste Züge unserer Idee auszutauschen. Jedoch ist man dabei relativ schnell an seine Grenzen gestoßen, da ein Aufruf der API sich innerhalb von Node.js nicht ohne Weiteres umsetzen lässt.

Die erste Telegram Nachricht durch Alexa

Der Post-Request zum Aufruf der API ließ sich dank bereits bekannter Syntax des API Calls und der letztendlichen Dokumentation von Node.js zu entsprechenden Requests (Link) relativ schnell umsetzen.
Somit war die erste Funktionalität in ihrer Grundform geschaffen und es war möglich festgelegte Nachrichteninhalte durch den Funktionsaufruf in einem Intent zu versenden:

```
 async handle(handlerInput) {

    const msg = "Testnachricht von Alexa";
    sendTelegramMessageFunction(msg);

 },

--------------------------------------------------------------------------------

function sendTelegramMessageFunction(msg) {

  const options = {
      host: 'api.telegram.org',
      path: '/bot{botID}/sendMessage?chat_id={chatID}&text=' +
encodeURIComponent(msg),
      method: 'POST',
  };

  const req = https.request(options, res => {
  console.log(`statusCode sendToTelegramFunction: ${res.statusCode}`)

[...]
```

Im späteren Verlauf haben wir den Wert von “msg” zu dem Wert der sprachlichen Eingabe angepasst, so dass auch bestimmte Inhalte und nicht eine Testnachricht verschickt werden können:

```
const slots = handlerInput.requestEnvelope.request.intent.slots;
const msg = slots.messageToTelegram.value;

sendTelegramMessageFunction(msg);     //call the message sending function

const speechOutput = "Nachricht mit dem Inhalt [" + msg + "] gesendet.";
```

Einarbeitung in Lex

Nachdem die Möglichkeit des Versendens von Nachrichten in den ersten Grundzügen funktionierte gingen wir weiter dazu über uns schon mal zu informieren wie man von Telegram entsprechende Nachrichten auf die Alexa übermittelt bekommt. Dabei stießen wir auf einen weiteren Bestandteil der Amazon Services - Lex.

Lex bildet, grob formuliert, eine Kommunikationsschnittstelle zwischen Alexa und z.B. Messenger Diensten, wie in unserem Falle Telegram. Somit wussten wir schon mal, dass wir Lex dazu benötigen werden, um die Nachrichten von Telegram entgegen zu nehmen.

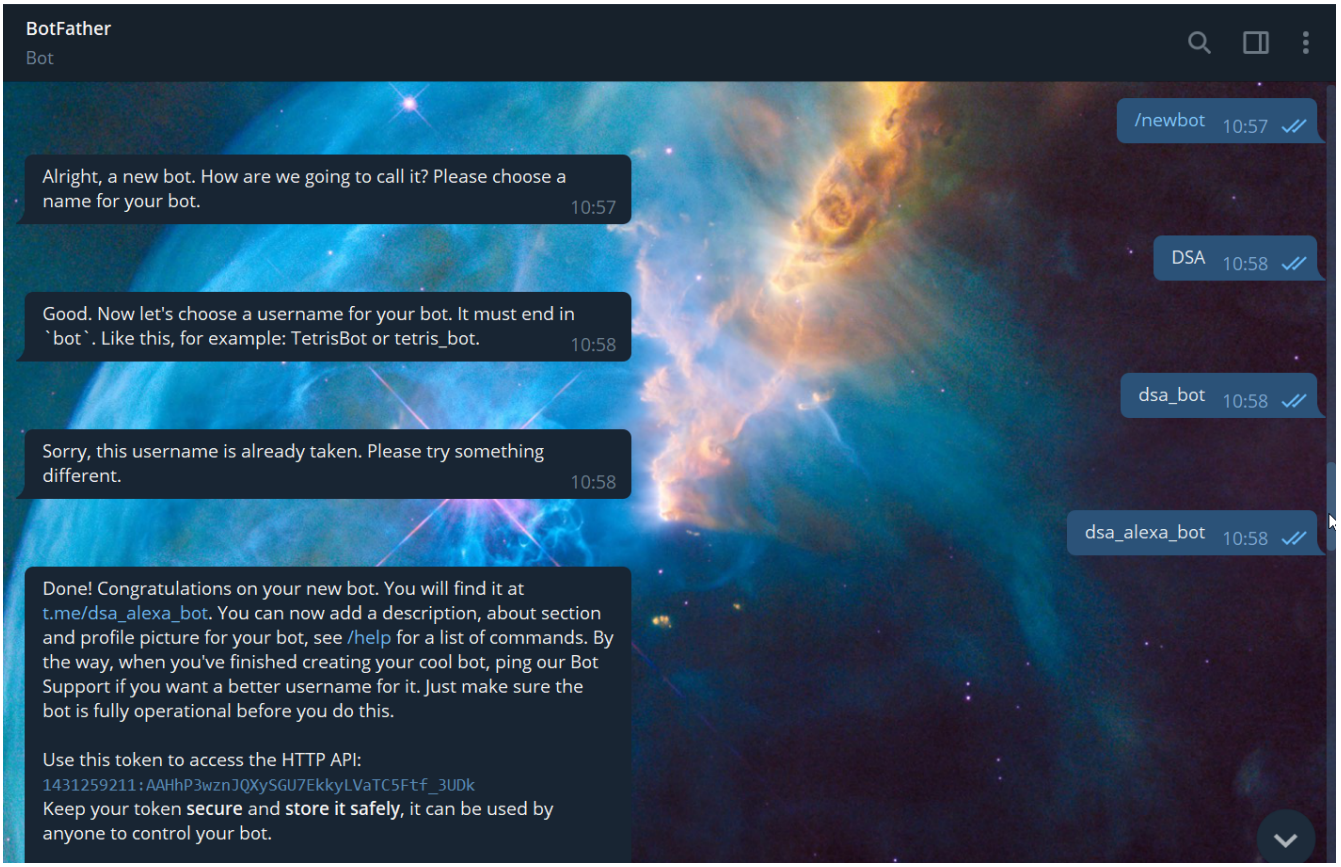
So begann die Einarbeitung in Lex, wofür wir uns diverse Videos, Dokumentationen von AWS usw. angeschaut haben und wir begannen mit ersten Versuchen. Im Laufe dessen gerieten wir relativ schnell an die Grenzen unserer Hochschul “Amazon Developer” Accounts, was letztlich dazu führte einen neuen, privaten Account zu erstellen.

Nach entsprechender Umstellung ging es relativ schnell zurück zum alten Stand und wir konnten den ersten Bot in Lex ohne Einschränkungen erstellen.

Der Bot sollte zu Beginn lediglich die Funktion erfüllen eine bestimmte Antwort auf egal welche Nachricht zu geben - um zu sehen ob es überhaupt funktioniert. Um das entsprechend umzusetzen war es nötig einerseits den Lex-Bot einzurichten, aber auch zeitgleich den Telegram-Bot.

Das Entgegennehmen von Nachrichten

Dank diverser Kenntnisse zur Telegram API und dem zugehörigen Bot System war es ein Leichtes einen Telegram Bot zu erstellen:



Mittels dem Token (siehe Bild) bekommt man Zugang zu dem erstellten Bot via Telegram API und kann somit z.B. Nachrichten versenden oder auch Nachrichten entgegennehmen und diese weiter verarbeiten.

Seitens Lex gestaltete sich das Ganze anfangs schwerer als gedacht, bis wir auf den Service “Serverless” von AWS stießen, mit dem sich ganz einfach lokal in Visual Studio Code eine JavaScript Datei schreiben und mittels “sls deploy” (+ entsprechender serverless.yml Datei) hochladen lässt.

Alternativ dazu fanden wir im Laufe der Entwicklung auch heraus, dass es noch die Möglichkeit gibt mit dem AWS Addon von Visual Studio Code Lambda Funktionen direkt hochzuladen, was wir dann für den Code des Alexa Skills genutzt haben.

So gelang es also durch Serverless den Endpoint für Lex<>Telegram herzustellen und innerhalb der .yml Datei auch den Link zum passenden Amazon Service Endpoint ausgeben zu lassen:

```
Inside .yml:

functions:
    telegramToLex:
        handler: telegramToLex.handler
        events:
            - http:
                  path: telegram-webhook
                  method: POST
                  cors: true

-----------------------------------

Path: .../DSALexTelegram>sls deploy
Output =>

endpoints:
  POST - {amazon-endpoint}
```

Dieser Endpoint ist in unserem Falle unser sog. Webhook, mit dem der Telegram Bot verbunden wird, um ankommende Nachrichten zu empfangen und in Lex zu verarbeiten.
Das Setzen des Webhooks funktioniert durch folgenden API Aufruf:

```
https://api.telegram.org/bot{botID}/setWebhook?url={webhookURL}
```

Einige, jedoch nicht alle, Quellen zu Lex und Serverless:
https://www.youtube.com/watch?v=rLw2oXEKJG8
https://www.youtube.com/watch?v=KTa1T14nkbw
https://docs.aws.amazon.com/lex/latest/dg/what-is.html
https://aws.amazon.com/de/serverless/
https://www.serverless.com/framework/docs/providers/aws/guide/serverless.yml/
https://www.youtube.com/watch?v=P-QPUdQnc3E

Speichern und Vorlesen von Nachrichten

Der wohl schwierigste und langwierigste Teil des ganzen Projekts ist das Speichern und Vorlesen der empfangenen Nachrichten geworden.

Begonnen haben wir dabei mit der DynamoDB, also einer nicht-relationalen Datenbank - dies ging über mehrere Arbeitsstunden, letztlich haben wir uns aber dann doch für eine relationale Datenbank entschieden, da die Arbeit mit Dynamo nahezu gar nicht funktionierte und daher eher für Motivationslosigkeit sorgte.

Für die relationale Datenbank nutzen wir den Datenbank Service RDS (Link) von Amazon. Damit lässt sich relativ einfach und mehr oder minder kostenlos eine Datenbank Instanz erstellen und verwalten.

Nach der Erstellung der Datenbank ging es an die erste Einrichtung, welche wir mit Tools wie bspw. HeidiSQL umsetzen wollten - dies war jedoch nicht ohne Weiteres möglich, da man für die jeweilige Datenbank Instanz, ähnlich wie auch bei Lex, Lambda, Dynamo und den weiteren Services, passende Sicherheitsgruppen erstellen muss, die auch die zugehörigen IP Adresse Freigaben und vieles mehr beinhalten. Zusätzlich war es dann nötig für externe Zugriffe auf die Datenbank einen SSH Tunnel einzurichten, welcher mit einem Private/Public Key System geschützt ist - anhand einer Anleitung (Link) konnten wir dies zumindest ansatzweise umsetzen.

Nach vielen erfolglosen Versuchen eine Verbindung herzustellen haben wir es nochmal mit Alternativen wie DBeaver und MySQL Workbench ausprobiert - mit MySQL Workbench funktionierte es dann nach ein wenig ausprobieren auch. Somit konnten wir nach vielen, langwierigen Stunden endlich mit dem Einrichten der Datenbank beginnen.

Das letztendliche Schema der Tabelle sah dann wie folgt aus:
“ID” als Primärschlüssel zur eindeutigen Identifikation;
“name” als Absender einer Nachricht;
“read” als quasi boolescher Integerwert zur Repräsentation, ob Alexa die Nachricht bereits vorgelesen hat;
“message” entspricht der gesendeten Nachricht;
“messageID” dient(e) dem Debugging

Column	Type	Default Value	Nullable	Character Set	Collation
id	int		NO		
name	varchar(45)		YES	utf8	utf8_general_ci
read	int	0	YES		
message	varchar(5000)		YES	utf8	utf8_general_ci
messageID	varchar(50)		YES	utf8	utf8_general_ci

Nach erfolgreicher Einrichtung der Datenbank und des entsprechenden Schemas ging es damit weiter die ankommenden Nachrichten, also die, die dem Telegram Bot innerhalb des Chats und somit also Lex geschickt werden, auch in der Datenbank zu speichern. Durch die Einbindung des “mysql”-Moduls war es einfach eine Verbindung zur Datenbank herzustellen und dank relativ gutem Vorwissen zu SQL war es auch keine Schwierigkeit die Daten in die Datenbank zu schreiben.

```
var conn = mysql.createConnection({
    host : '{amazon-endpoint}',
    user : '{user}',
    password : '{pw}',
    database : '{db}'
});
```

Die Funktion zum Inserieren der Daten wird innerhalb einer anderen Funktion, welche die Daten für Lex vorbereitet, aufgerufen:

```
const TelegramToLex = body => {
//extract needed information of message body
    const chatID = String(body.message.chat.id);
    const message = body.message.text;
    const name = body.message.chat.first_name;
    const messageID = String(body.update_id);

    insertIntoDB(name, message, messageID);

--------------------------------------------------------------------------------

function insertIntoDB(name, message, messageID){
//insert incoming messages into db

    conn.connect();

    let sql = mysql.format("INSERT INTO dsa (message, messageID, name) VALUES
(?, ?, ?)", [message, messageID, name]);

    conn.query(sql, function(error){
        if(error) throw error;
    });

    conn.end();
}
```

Um die Reaktion von Lex, auf ankommende Nachrichten, an Telegram zu senden gibt es, ähnlich wie bereits oben gezeigt, einen Aufruf der API mit der übergebenen Nachricht:

```
const sendToTelegram = message => {
    const token = '{botID}';
    const telegramURL = `https://api.telegram.org/bot${token}/sendMessage`;

    return Axios.post(telegramURL, message);   //posts the https API request
};
```

Somit war es also möglich Nachrichten per Alexa zu senden, Nachrichten in Telegram an Lex zu senden und entsprechend in die Datenbank speichern zu lassen - jedoch nicht gespeicherte Nachrichten auch von Alexa vorlesen zu lassen.

Somit begann also der schwierigste Teil des ganzen Projekts, da technische Hürden und Unwissenheit einen großen Problemfaktor dargestellt haben.

Zur Einbindung der Datenbank haben wir wieder das “mysql”-Modul eingebunden und die Verbindung zur Datenbank aufgebaut. Dabei war es erneut nötig die entsprechenden Sicherheitsgruppen und passende Richtlinien zu verteilen.

Mit der funktionieren Verbindung war erstmal unser Ziel die Daten aus der Datenbank zu ziehen, um zu prüfen, ob alles wie geplant funktioniert - das war auch direkt von Beginn an der Fall. Die geladenen Daten haben vorerst in der Konsole anzeigen lassen, um zu sehen in welchem Format sie vorliegen und wie wir darauf zugreifen können. Zur Vereinfachung haben wir sie in ein JSON Objekt umgewandelt und erste Tests damit durchgeführt.

Nachdem wir die Konsolenausgabe so angepasst hatten, dass wir die Daten an den eigentlichen Ausgabepunkt weitergeben wollten, stießen wir auf die ersten Probleme. Der Aufruf der Datenbank Abfrage fand asynchron statt und wir haben versucht mittels des “Callback” den eigentlichen “Return” Wert aus der Funktion zu bekommen - jedoch war dieser Wert, wenn man ihn versucht hat außerhalb der Funktion auszugeben, immer “undefined”, d.h. die Funktion lief schneller in den “Callback” hinein, als die Datenbankabfrage entsprechende Ergebnisse liefern konnte.

So probierten wir also stundenlang verschiedenste Methoden, haben uns versucht durch Stackoverflow und sonstige Hilfeforen eine mögliche Lösung zu erarbeiten, jedoch ohne Erfolg.

Irgendwann stießen wir auf “Promise”, welches die finale Beendigung einer asynchronen Operation darstellt - mit vielen weiteren Versuchen, basierend auf

nicht vorhandenen JavaScript Kenntnissen, haben wir es letztendlich geschafft eine erfolgreiche Übergabe des “Query Result” zu realisieren:

```
function getUnreadMessages() {
//get unread messages from db (inserted by lex)
  return new Promise(function(resolve){
    conn.connect();

      conn.query('SELECT id, name, message, `read`, (SELECT COUNT(id) FROM dsa WHERE `read` = 0)
AS rowcount FROM dsa WHERE `read` = 0', function (error, results) {

          if (error) throw error;

          JSON.stringify(results);

          if(results.length < 1){
//result length = 0 if the query cant return *any results*

            resolve("Es wurde keine neue Nachricht gefunden!", null);

          }else{

            let text = "";

            for(i=0; i<results[0]["rowcount"]; i++){
//concatenate the output string

              text = text + "Nachricht ["+(i+1)+"] von " + results[i]["name"] + " mit dem Inhalt
" + results[i]["message"] + " [ ] ";

            }

            for(i=0; i<results[0]["rowcount"]; i++){
//update read in db to prevent multiple output

              input = results[i]["id"];

              sql = mysql.format("UPDATE dsa SET `read` = 1 WHERE id = ?", [input]);

              conn.query(sql, function (error) {
                if (error) throw error;
              });

            }

            resolve(text, null);
          }
      });
  });
}
```

Innerhalb des Request Handlers sah es dann wie folgt aus:

```
const ReadMessageHandler = {
  canHandle(handlerInput) {
    const request = handlerInput.requestEnvelope.request;
    return(request.type === 'IntentRequest'
        && request.intent.name === 'ReadMessageIntent');
  },
  async handle(handlerInput) {

    return new Promise(function(resolve) {getUnreadMessages().then(result => {
//get messages from db

        conn.end();

        const speechOutput = result;

        resolve(handlerInput.responseBuilder
          .speak(speechOutput)
          .reprompt(HELP_REPROMPT)
          .getResponse()
        );

      }, error => {
        if(error) throw error;
      });

    });
  },
};
```

HINWEIS

Eine Gruppe von Kommilitonen hatte ein ähnliches Problem, bei der Verarbeitung ihrer Daten aus einer asynchronen Funktion, so haben wir uns bereit erklärt zu helfen.
Dies scheiterte anfangs, wie auch bei uns, sehr kläglich, bis wir uns letztendlich dazu entschieden haben unsere Lösung des Problems weiterzugeben, unter der Prämisse, dass die andere Gruppe einen Verweis auf unsere Mithilfe innerhalb des Codes und auch der Dokumentation hinterlegt, um Missverständnisse wegen des “gleichen” Codes zu vermeiden und die Transparenz durch unsere Mithilfe zu wahren!

Testen, testen, testen…

Zum Ende der Entwicklung haben wir sehr viele Tests durchgeführt, alle möglichen Varianten und Formen von Nachrichten ausprobiert, um zu testen ob alles ordnungsgemäß funktioniert.

Dabei ist uns der Fehler unterlaufen Emojis in die Nachrichten zu packen, da Lex sich daran aufhängt und dementsprechend jegliche weitere Kommunikation verweigert, was dazu führte, dass wir den Lex Bot und auch den Telegram Bot nochmal neu erstellen mussten.

Ebenfalls war es bis zum Ende der Entwicklung leider nicht im Rahmen unserer Möglichkeiten umsetzbar innerhalb des Skills mehrmals eine Datenbankabfrage zu stellen. Die Funktionen erfordern eine Beendigung der Verbindung:

```
conn.end();
```

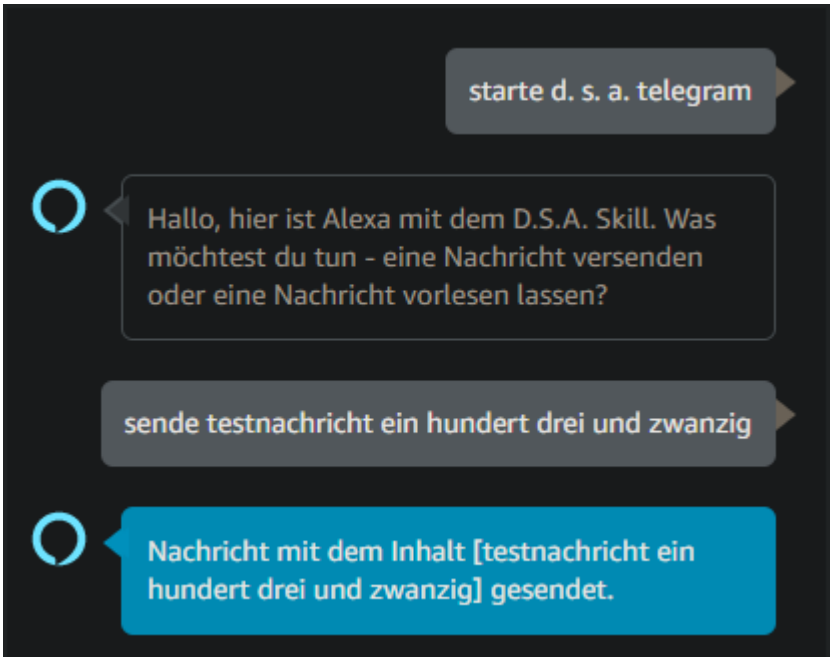
da sie andernfalls in einen “Timeout” laufen und keinerlei Ausgaben zurückliefern.
Ruft man entsprechende Funktion nochmal auf versucht der Skill eine erneute Verbindung herzustellen, was von RDS aber verweigert wird, da es innerhalb der gleichen “Session” geschieht und die Verbindung innerhalb dieser Session bereits beendet wurde.

Das Ganze sieht dann innerhalb der Konsole so aus:

```
"errorType": "Error", "errorMessage": "Cannot enqueue Handshake after invoking quit."
```

Wir haben es mit diversen “Workarounds” probiert, sind daran jedoch gescheitert.

Ein weiterer Punkt steht im Zusammenhang mit dem gewählten “Slot Type” und diversen Zeichenketten, wie bspw. Zahlen - der eigentlich Aufruf war “Sende Testnachricht 123” (siehe Bild)



Wir konnten leider keinen anderen Typ herausfinden, in dem wir längere Strings speichern können und waren daher gezwungen “searchQuery” zu nehmen, welches für die Konvertierung von Zahlen in ausgeschriebene Worte oder auch die Punktsetzung in Abkürzungen sorgt - wie bspw. bei DSA => d. s. a.

Die Zukunft des Skills

Wir möchten den Skill nach Beendigung des Moduls gerne weiterführen und auch ausbauen. Diverse Funktionen die wir bereits geplant haben, den Rahmen der Umsetzung innerhalb des Moduls aber wesentlich überschritten hätten sind:

- Versenden von Nachrichten an bestimmte Personen (aktuell nur an Entwickler möglich)
- Nachrichten auch stumm versenden können
    - (`/sendMessage?chat_id={chatID}"&disable_notification=true`)
- Nachrichten auch mit Formatierung versenden können
    - (`/sendMessage?chat_id={chatID}"&parse_mode=html`)
- Einbindung von Emojis und Stickern
    - (`/sendMessage?chat_id={chatID}"&sticker={stickerID}`)
- Sprachnachrichten (insofern von Alexa unterstützt)