AMAZON ECHO APPROACH

1. Design

The picture below shows the core system architecture and the way the components communicate with each other.

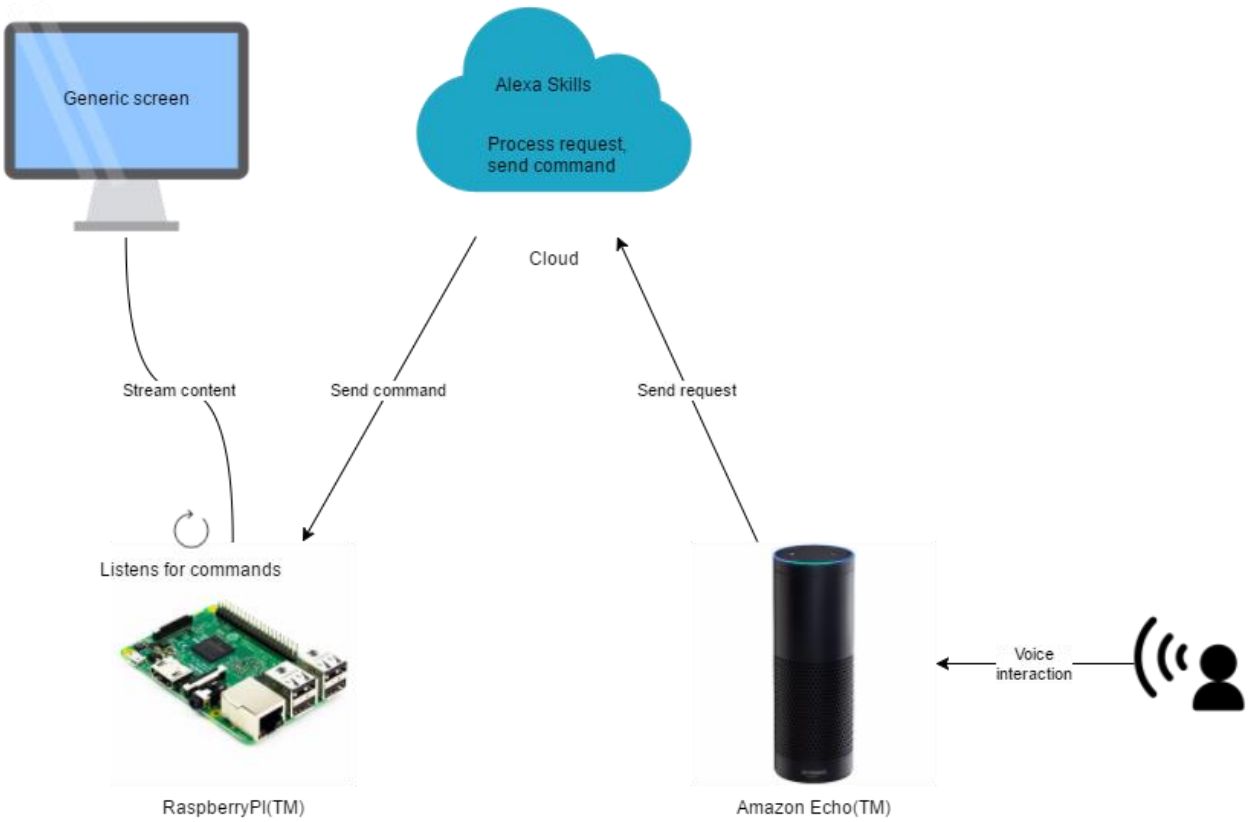

Figure 1. Core system design

2. Core System Implementation

The core system consists of three main hardware components:

- Amazon Echo
- RaspberryPI
- An output console (the monitor)

These components are made to work together to give the desired result.

2.1. Alexa skill

A *Skill* is the only way to add functionality to the Amazon Echo, thus, in order to make the Amazon Echo have video output, a *Skill* needed to be created.

The purpose of the *Skill* is simple – after *Alexa* is asked to say "hello", it will send an *HTTP* request to the specially deployed *Node.js* [7] server on the RaspberryPI.

The following steps were required to create the most basic *Skill*:

- Register an Amazon developer account on https://developer.amazon.com
- After the registration it will be possible to access *Alexa Skill* kit, where the "Add new skill" button needs to be pressed.
- A newly appeared window will require some user input:

- **Skill information tab**

  Skill type: **Custom Interaction Model**

  Name: **HelloSkill**

  Invocation name: **hi**

- **Interaction model tab**

In this tab "Intent Schema" and "Sample Utterances" fields are defining the behavior of a *skill*. Intent is a hardcoded option of what *Alexa* can answer to a "sample utterance". The "Intent Schema" is written in a form of a *JSON* object. In addition to that each intent can also have a slot for a variable. This is very practical for a continuous talk with *Alexa*, like in the following case:

-*User*: "Alexa, Show me the news"

-*Alexa*: "What kind of news would you like to see?"

-*User*: "Economics"

Since in this project it was decided to establish a one-way communication channel (*Alexa* -> PI -> Screen), there was no need to include slots for variables in the "intent schema". The interaction model used in a system can be visible in the picture below:

```
{
  "intents": [
    {
      "intent": "sayHello"
    },
    {
      "intent": "AMAZON.HelpIntent"
    }
  ]
}
```

Figure 2. System interaction model

The “sample utterances” for the intent are given below:

```
1 sayHello say hello
2 sayHello say hello world
3 sayHello hello
4 sayHello say hi
5 sayHello say hi world
6 sayHello hi
7 sayHello how are you
```

Figure 3. Sample utterances example

A format of utterance is the following:

{INTENT_NAME} {PHRASE_TRIGGERING_INTENT}

- **Configuration tab**

Choosing the end-point defines where *Alexa* will delegate the request after the *Skill* invocation. A common choice of *Alexa Skill* development would be *AWS Lambda*, a service provided by Amazon which supports various server applications like *Node.js*.

In this project an *HTTPS* endpoint was chosen with an IP address of a Raspberry PI giving a possibility to trigger methods on a server required for system functionality.

- **Testing tab**

At this point the *Skill* is ready to be used. Using the service simulator tool it is possible to see if *Alexa* is sending data to the URL defined earlier, and if any data is coming back. More about the “Alexa Service Simulator” can be found in testing section.

2.2. RaspberryPI

As it was mentioned in the *Alexa Skill* implementation chapter, Alexa will be sending requests to the URL defined in a configuration tab. Requests from *Alexa* travel directly to the RaspberryPI device, where they get processed, and if conditions are met, information is displayed on a connected screen.

RaspberryPI has Ubuntu running as an operating system. Interaction and configuration of a device can be done in two ways:

- *SSH* connection
- Interact using user input hardware connected to the RaspberryPI

At this stage *SSH* connection is all that is needed to run the *HTTP* server.

2.2.3. Preparing Node.js

To be able to receive and manipulate data received from *Alexa* it was decided to use *Node.js* server due to its scalability and variety of modules for different purposes.

As soon as the *SSH* connection with RaspberryPI is established, *Node.js* can be installed by typing the following into the terminal window:

```
sudo apt-get install nodejs
```

Figure 4. Node.js installation command

To be able to add additional modules to the server package it is also necessary to install the Node Package Manager (a.k.a. *npm*). Just like the *Node.js* server, *npm* can be installed by typing the following into terminal window:

```
sudo apt-get install npm
```

Figure 5. Node package manager installation command

At this point *Node.js* should be fully ready. Checking versions of *npm* and *Node.js*:

```
$ node -v
v0.10.25
$ npm -v
1.3.10
```

Figure 6. Node.js and npm version check command

2.2.4. Preparing the HTTP server

The *Node.js* framework doesn't include a default module for operating with *HTTP* requests/responses, therefore the server requires an additional module called *Express JS* [8]. Installation is simple:

```
$npm install express
```

Figure 7. Npm install command

2.2.5. Implementing server logic

Node.js is required to be acting as a middleware between *Alexa* and the generic screen, meaning that it has to be able to receive *HTTP* request from the Amazon Echo and be able to communicate with an output screen.

2.2.5.1. Alexa to Server

The *Node.js* server consists of one main file, where using *Express JS* installed in a previous step an *HTTP* server is deployed:

```
var app = require('express')();
var http = require('http').Server(app);


http.listen(3000, "0.0.0.0"); //listen to incoming connections
console.log('listening on *:3000');
```

Figure 8. HTTP server deployment

At first packages "http" and "express" are being imported, then values are passed to the "http.listen()" function. The values indicate on which port and IP address the requests are expected. Now the *Node.js* server is ready to catch incoming *HTTP* requests, yet will not do anything with them.

To be able to react to the requests, the server needs a simple route handling mechanism. For the sake of simplicity, it was decided that the methods on a server can be invoked by only accessing a link, independently from the parameters passed in a request body. The snippet below shows the implementation of the function which is triggered when the POST request arrives to "0.0.0.0:3000/sayHello:".

```
app.post('/sayHello', function(req, res){ // on sayHello link
      console.log('someone said hello!');
      res.end("thank you for the request!"); // sending a response back to Alexa

});
```

Figure 9. HTTP server route handling

The method will print some text in the console and reply with a response back to the sender.

2.2.5.2. Server to Screen

As the server communicates with *Alexa*, it is now possible to output the result of a request to the screen or graphical user interface.

It was decided to use the web browser as a solution to this problem. This approach simplifies the program to a single web page application.

For the communication between the browser window and the *Node.js* server an extra *JavaScript* library called *socket.io* [9] was used. In simple words, *socket.io* works as a broadcasting station on a server side and as a receiver on the client side (browser window).

Just like *Express JS*, *socket.io* is not a part of *Node.js*, therefore it needs to be installed first:

```
$npm install socket.io
```

Figure 10. Socket.io installation command

Now, by adding one more line of code to the server file, *socket.io* is ready to be used:

```
var io = require('socket.io')(http);
```

Figure 11. Socket.io enabling

A code snippet below is a connection handler implemented using socket.io library.

```
io.on('connection', function(socket){ //////on connection success
      console.log('web interface connected!');
      socket.on('disconnect', function(){ /////on disconnect
      console.log('web interface disconnected');
      });
      socket.emit('connection', 'client and server connected!');
});
```

Figure 12. Connection handler using socket.io

The first parameter in the function “on()” gives the ability to define a condition on which the function in the second parameter is called, in this case, on client connection. Another function named “emit()” is used to send a signal to the client which has connected.

After the client is connected, the server can use the same socket to invoke methods on the client side, in particular the "/sayHello" function:

```
app.post('/sayHello', function(req, res){ // on sayHello link
      console.log('someone said hello!');
      res.end("thank you for the request!"); // sending a response back to Alexa
      io.emit('Hello');
});
```

Figure 13. sayHello method invocation

In the above snippet a modified "/sayHello" function is shown, where "io.emit()" is used to let the client know when the server function is called by *Alexa*.

2.2.6. Browser client implementation

The client side implementation consists of a few *HTML* elements controlled by *JavaScript* in order to omit the necessity of refreshing the page. The *JavaScript* part has a *socket.io* receiver, configured to listen on port 3000 on *localhost*, since the socket communication is going to take part on the same device (*RaspberryPI*).

```
<script>
    var socket = io('http://localhost:3000');

    socket.on('connection', function(){
        $("#con").text("Connected to the server!");
        console.log("connection established!");
    });
```

Figure 14. socket.io receiver

The code above shows the client socket configuration and the function which executes on a successful connection, which uses basic *JQuery* function to write the text to browser window.

In the same manner as above, a function responding to a "Hello" string emitted from the server is implemented as such:

```
socket.on('Hello', function(){
    console.log("hello has been received! :)");
    $("#p1").text("Hello to you sir");
});
```

Figure 15. Function responding to "Hello"

This will change the contents of the *HTML* element, confirming that the signal was received.

2.2.7. Providing the user with control on a system

The basic *Alexa Skill* implementation model mentioned earlier only calls a specific *URL*, where *Node.js* reacts to any request which it comes in contact with. This approach is not secure and it does not provide the user an option of adding parameters to the request, therefore it was implemented for test purposes only. Following the project requirement list it was decided to create another *Alexa Skill*, which will allow the user to choose between different types of content displayed on the screen.

Fortunately, the *Amazon Echo API* includes a method of detecting and passing parameters to a defined end-point in an *HTTP* request body, where certain functions on *Node.js* server will be triggered as expected parameters are detected.

2.2.8. Alexa "show" skill configuration

According to the implementation steps of the basic *Alexa Skill*, the creation of the "show" *Skill* starts from entering the *Skill* information.

- *Skill* name – **News controller**
- Invocation name – **shows**

In the interaction model tab, the "intent schema" will be dictating the way *Alexa* extracts key words from sentences uttered by the user. The "Intent schema" below is presented in form of a *JSON* object, making it easily serializable and globally recognizable.

```
"intents": [
  {
    "intent": "show",
    "slots": [
      {
        "name": "what",
        "type": "AMAZON.LITERAL"
      }
    ]
  }
```

Figure 16. Intent schema

In the above "intent schema" three key things are defined:

- Intent name (in this case – *show*)
- Number of slots (words which will be extracted by *Alexa*)
- Slot name (in this case – *what*)
- Slot type (AMAZON.LITERAL is Amazons' substitute for type String)

Next step is to fill the “sample utterances” field with all the possible sentences related to this *Skill*, adding the slot type and slot name in place where they are expected. In this project the only choice is between two options.

```
show me {sports|what}
show {sports|what}
show me {news|what}
show {news|what}
```

Figure 17. Sample utterances

Each sentence should start with the name of the intent and if the *Skill* contains slots, they should be defined in between curly brackets as such: **{expected_value | slot_name}**.

The next tab with an end-point configuration was left blank until the proper functionality of *Node.js* server is written. The following tab will require choosing an *SSL certificate* for end-point.

With the help of *Alexa’s* service simulator, it is very easy to see if the skill works as expected. When asked to “show news” the request body will look the same as in the picture below:

Figure 18. "show news" skill request body

In a same way, when asked to “show sports”, the produced output will look in the following way:

Figure 19. "show sports" skill request body

Knowing that this information is successfully sent to the endpoint, it is only a matter of *Node.js* server configuration to process the requests coming from *Alexa* and show appropriate information on a screen.

2.2.9. Receiving and processing requests on Raspberry PI

As the request from *Alexa* is sent, the *Node.js* endpoint where the request arrives needs to be prepared for parsing *JSON* objects and extract values from them. After taking the basic *Alexa Skill* files as a base and installing an additional package on *Node.js* called *body-parser* [10] it became possible to reach *JSON* object values in a received *HTTP* request on a server side.

Installation of the *body-parser* is called using the node packet manager by opening command terminal, changing the directory to the project directory and running a following command:

```
$npm install body-parser
```

Figure 20. Body parser installation command

After the installation is complete the *body-parser* is ready for use. The next step is initializing and connecting the *body-parser* module to an *Express.js HTTP* server. Adding the code below into the top of the *main.js* file will allow the program to use the *body-parser* functions:

```
var bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
```

Figure 21. Body-parser-enabling code

As it was mentioned in the basic *Skill* implementation, *Alexa* sends *HTTP POST* requests, which means that the server is now ready to process them. Like previously, a function listening to requests of type *POST* was built using *Express JS.*

Note: The function below is only triggered when the client side accesses the *SERVER_IP_OR_DOMAIN:3000/show* URL.

```
app.post('/show', function(req, res){

      var json = JSON.stringify(req.body);
      var parsed = JSON.parse(json).request.intent.slots.what.value;

      console.log("showing result:" + parsed);

      if(parsed === "news"){
            io.emit('news');
      }
      else if(parsed === "sports"){
            io.emit('sports');
      }
});
```

Figure 22. Function which processed the POST request

When the function is triggered, "JSON.stringify()*"* extracts the request body, which is a *JSON* object and stores it inside the "json" variable. Afterwards, using the "JSON.parse()" function, the program gets the slot value from the previously parsed object, saving it in the variable called "parsed". An "if" statement is checking if the correct variables arrived, and on detection of the "news" or "sports" strings, different signals are emitted to the client window.

2.2.10. Receiving and processing signals from server

In the "index.html" file from the basic *Alexa Skill* emitted signals from the server should be already reachable, but still requires processing.

Knowing that the server can emit either "news", "sports" or "hello", the client needs to be prepared for those events to occur. Like in the basic *Alexa Skill* implementation, event handling functions are created using *socket.io Node.js* module. The code snippet below shows an example of how the "sports" section is displayed:

```
socket.on('sports', function(){
   $("#p1").removeClass("showing");
   $("#p2").removeClass("showing");
   $("#item-1").removeClass("green-item");
   $("#item-2").removeClass("green-item");

   $("#p3").addClass("showing");
   $("#item-3").addClass("green-item");
});
```

Figure 23. "sports" section display method

The function presented above is triggered after the server emits the “sports” signal. The program makes all the containers invisible besides the “sports” section which features a video, additionally highlighting the selected menu item on the left side of a page.

3. Testing and Results

The system was tested against every Use Case to be sure that everything works as planned. For each new hardware component added to the core system, many tests had also had to be done to each of the components individually so that any bugs or errors which may occur can be spotted in due time and solved as soon as possible.

After this followed testing the core system as a whole from both a hardware and software point of view, to make sure that every component communicates well with each other and give the proper result as output.

In parallel to implementation of a core system, each communication channel established between each hardware component was checked separately.

- Amazon Echo -> RaspberryPI
- Raspberry PI -> Output window (monitor)

3.1. Alexa to RaspberryPI

To test if the RaspberryPI is receiving any data from *Alexa* a tool provided by *Amazon* called “Service Simulator” was used, allowing sending *HTTP* requests to the end-link defined in the *Alexa* skill configuration tab. Typing a sample utterance into the field will generate a body for a *HTTP* request in form of a JSON string, containing data like the user ID, request type, attributes (if any were defined in the Interaction model tab), application ID and other metadata relevant for the server to determine the identity of a caller.

The picture below shows an utterance being invoked i.e. – a string that Echo extracts from a phrase the user says, in this case “say hello”, and a service request body.

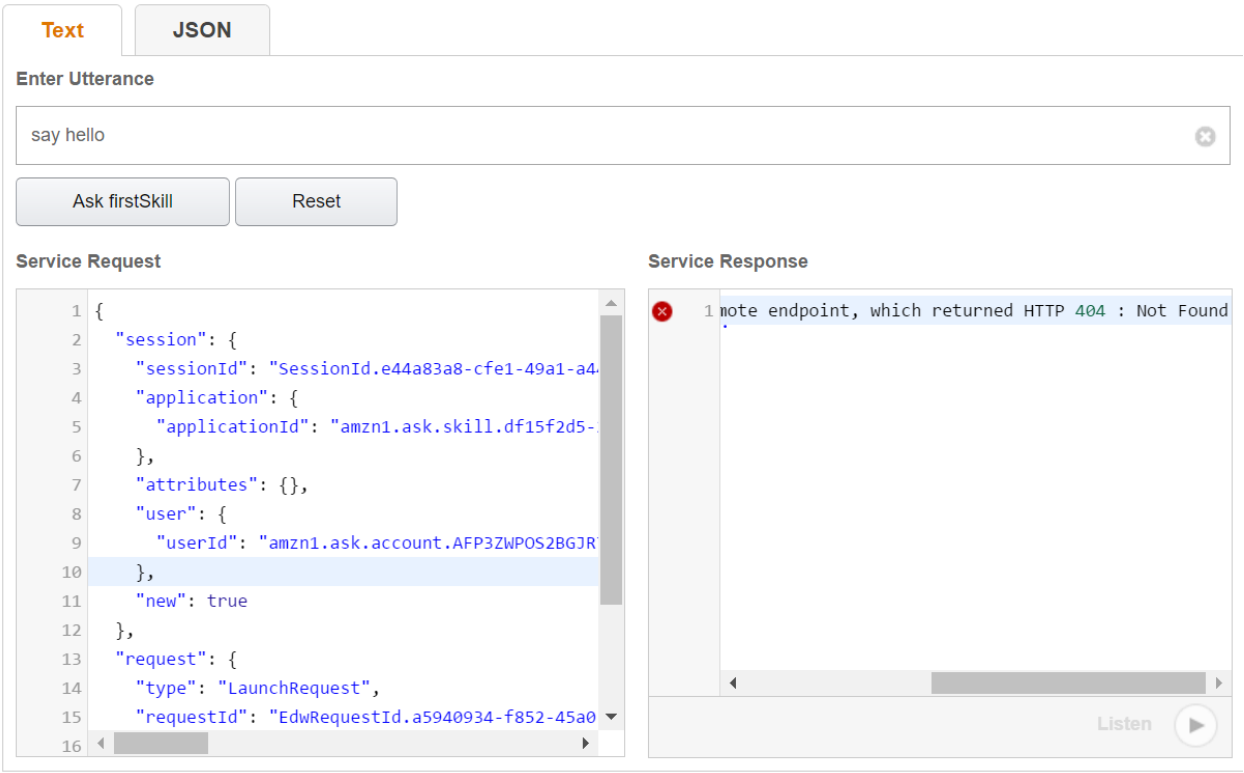

Figure 24. Utterance invocation

While the RaspberryPI server is not active, the response Echo will get is the error message "404: Not found".

Depending on the network the RaspberryPI is connected to, the end-point Amazon Echo is referring to might differ. The problem might occur when exposing the RaspberryPI *Node.js* server on the external IP, since in most cases the network connection goes through a router, requiring port forwarding, which is not always possible to do due to the accessibility of the router device.

One of the ways to expose a *Node.js* server on the internet is using a service called ngrok [11], which is installed on the RaspberryPI using the node package manager over Linux terminal:

```
$npm install ngrok
```

Figure 25. ngrok installation command

After install, the *Node.js HTTP* server has to be launched by changing the directory in a terminal window to the location of the "main.js" file, then running the node command to run the server:

```
$node main.js
```

Figure 26. The "main.js" node command

A resulting window with a success message will look in the following way:

```
>node main.js
listening on *:3000
```

Figure 27. Server running successfully

After that, in another terminal window “ngrok” needs to be launched by typing the command below along with a port number to listen:

```
>ngrok http 3000
```

Figure 28. Launch "ngrok" command

This will open an “ngrok” status window, where a proxy link to the *Node.js HTTP* server is located:

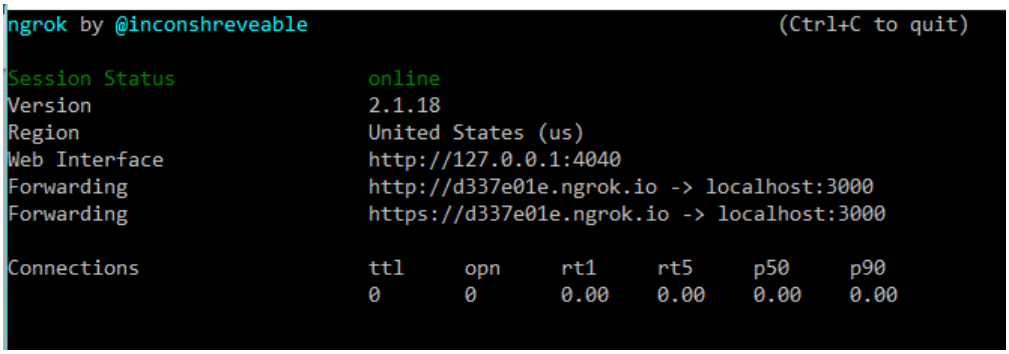

Figure 29. "ngrok" status window

As it is visible in the screenshot above, “ngrok” generates an *HTTPS* and *HTTP* link accessible from the internet and redirects the traffic to the *Node.js* server running on the RaspberryPI.

Note: The proxy URL generated by ngrok is not constant, meaning that every time the ngrok service is restarted the proxy URL will change requiring to update the end-point under the Alexa skill configuration tab.

At this stage the *Node.js* server is accessible from the internet. To check if the *HTTP* server is getting any data, a request tracking tool provided by “ngrok” can be used. The link to the tool can be seen in the “ngrok” status window under the name Web Interface, which always points to localhost port 4040 (or https://127.0.0.1:4040).

Now using the Alexa Service Simulator and “ngrok” web interface it is possible to see if the request sent by the Echo is being received on the *Node.js* server. Inputting an utterance into the Service Simulator in the testing tab of the *Alexa Skill* will trigger the function on the server side producing an output in the server console.

A fragment from the ngrok web interface below confirms that the Echo request was received successfully. Selecting an element from the left column allows to see the selected request body and meta data.

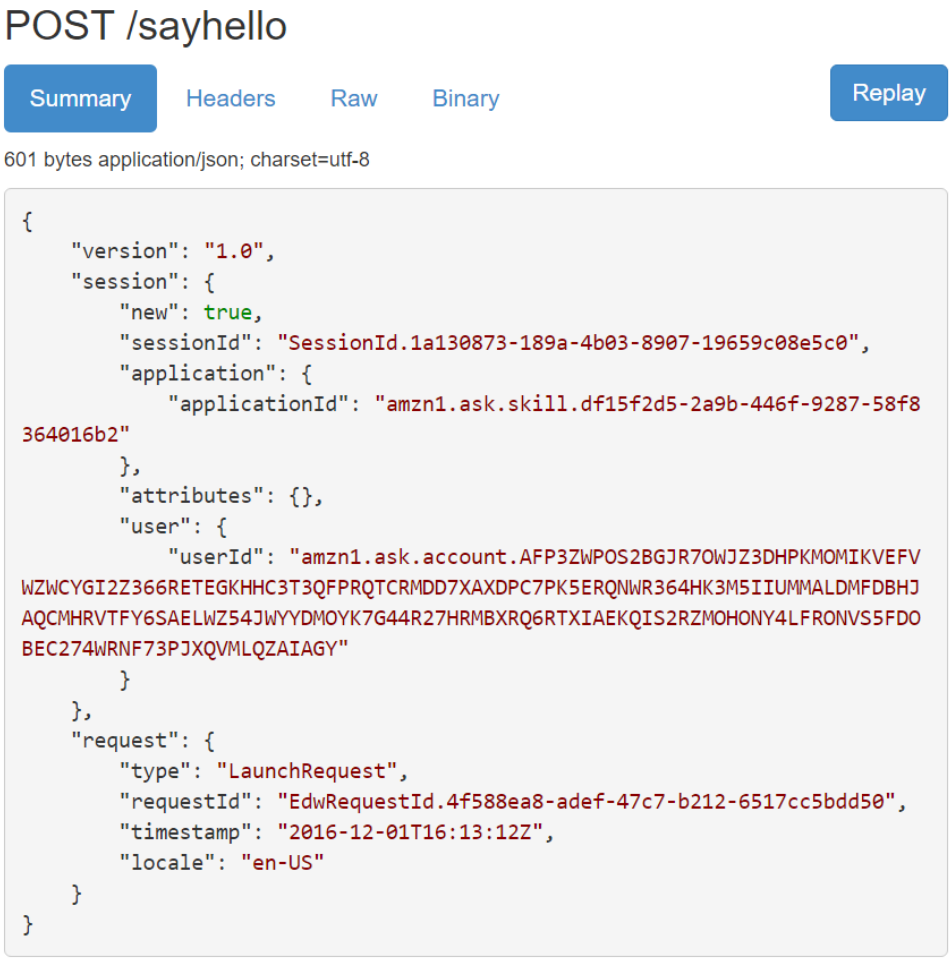

Figure 30. "ngrok" web interface

On the *Node.js* side the request from the Amazon Echo triggers the function which prints a confirmation message to the terminal window:

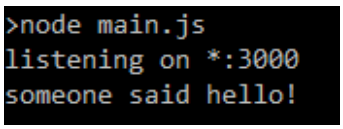

Figure 31. Node.js confirmation message

3.2. HTTP server to screen

Real time output on the screen is implemented using *HTML*, *CSS*, *jQuery* and *socket.io*. The output *HTML* window opened on the screen will automatically connect to the Node.js server. After this both parties can communicate on the common *socket.io* channel.

Now, as all components are ready for use, sending a request from the *Alexa Skill* Test tab will result in printing the next message in the HTML client window:

Figure 32. HTTP server screen output

3.3. Testing “show” skill

To test the “show skill” steps listed below had to be conducted:

- Start the *Node.js* server (*main.js)*

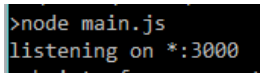

Figure 33. node.js start server command

- Start the “ngrok” service on a server machine

Figure 34. "ngrok" start server command

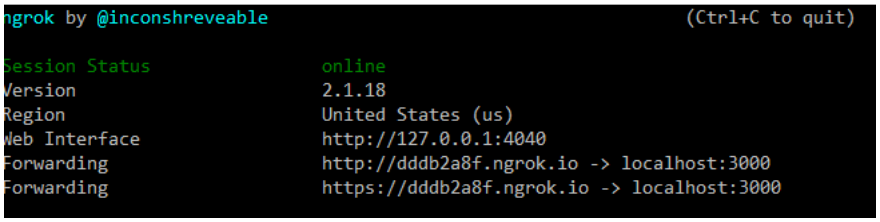

Figure 35. "ngrok" session status

- Open the output *HTML* in the browser on a server machine (*index.html).*

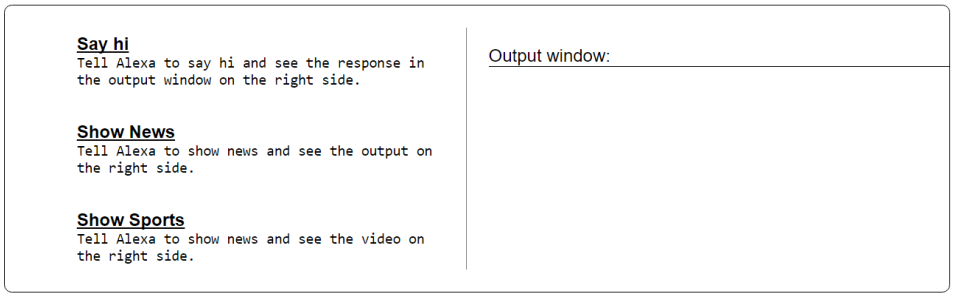

Figure 36. "index.html" file

The output window will remain blank until any of the *Alexa Skills* are invoked.

- Set an endpoint for *Alexa* "show" *Skill* to point to "ngrok" generated URL and save.

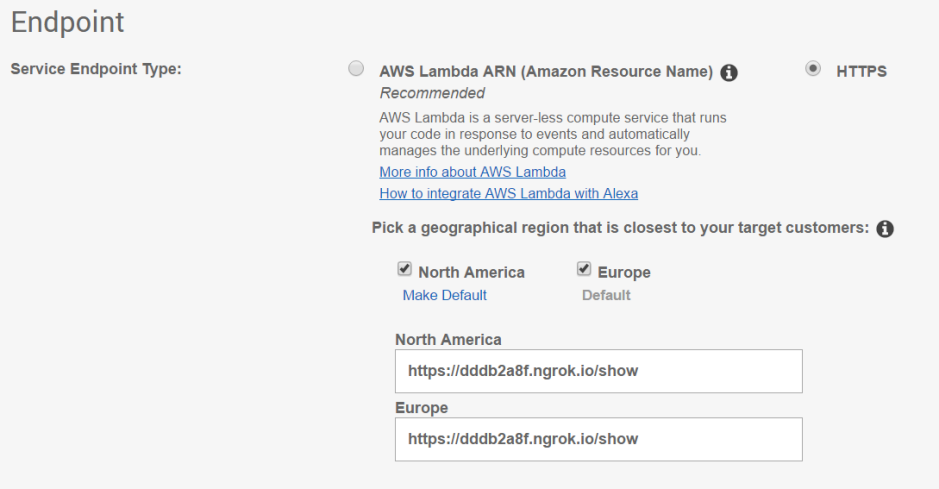

Figure 37. Alexa Skill endpoint

- Send request to *Node.js* server using *Alexa* service simulator in *Alexa Skill* test tab.

Figure 38. Alexa service simulator

- Observe the changes in the browser window of the server machine.

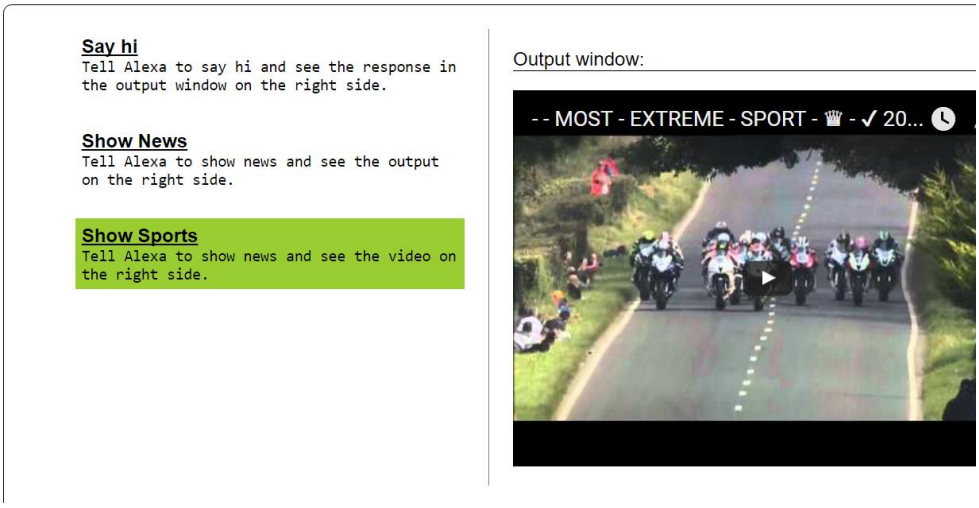

Figure 39. Server machine browser window

This concludes the testing and results area. Testing was made on both individual components and software as well as the core system as a whole. All tests have been a success proving that the system works as expected.

4. References

[1] - https://www.amazon.com/Amazon-Echo-Bluetooth-Speaker-with-WiFi-Alexa/dp/B00X4WHP5E

[2] - https://en.wikipedia.org/wiki/Amazon_Echo

[3] - https://madeby.google.com/home/

[4] - https://www.raspberrypi.org/

[5] - https://beagleboard.org/black

[6] - http://www.intel.com/content/www/us/en/compute-stick/intel-compute-stick.html

[7] - https://www.nodejs.org/

[8] - https://www.expressjs.com/

[9] - https://socket.io/

[10] - https://www.npmjs.com/package/body-parser

[11] - www.ngrok.com