

# SC5

## SERVERLESS MESSENGER BOT WORKSHOP

Mikael Puittinen, Chief Technology Officer

Eetu Tuomala, Principal Cloud Specialist

25.10.2016

# WORKSHOP OBJECTIVES

- Get familiar with AWS Services
- Get familiar with Serverless Framework 1.0
- Get familiar with Messenger Platform & Wit.ai
- Have fun building a weather bot on Messenger

# AGENDA

09:00-10:30	Introduction to AWS Hands-on exercises: AWS Lambda / API Gateway
10:30-11:00	Break
11:00-12:30	Setup Node.js / AWS-SDK Learning Serverless 1.0 by doing Introduction to Messenger bot tech: MessengerPlatform, Wit.ai and our serverless messenger boilerplate
12:30-13:30	Lunch
13:30-15:00	Hand-on-exercises: building a weather bot
15:00-15:30	Break
15:30-17:30	Additional challenges Demos Feedback & discussion

# SC5 BRIEFLY



CLOUD  
SOLUTIONS



BUSINESS  
APPLICATIONS



DIGITAL  
DESIGN

10  
YEARS

60+  
CUSTOMERS

200+  
PROJECTS

85  
HACKERS  
DESIGNERS

HEL  
JKL

~7  
MEUR  
2016



Enegia

Gasum

HAPPYORNOT®



s a n o m a



VISIT OUR WEB SITE FOR MORE INFO: [HTTPS://SC5.IO](https://sc5.io)

This presentation is available for download at:

<http://serverless.fi/docs/londonbots16.pdf>

## JOIN OUR SLACK (HOSTED BY THE SERVERLESS FINLAND COMMUNITY)

- Register at <http://serverless.fi>
- Let's discuss on #serverlessconf-bots during and / or after the workshop

# INTRODUCTION TO AWS

**10 YEARS AGO...**

**MARCH 2006**

Amazon Simple Storage Service (Amazon S3) launch

**AUGUST 2006**

Amazon Elastic Cloud Compute beta (Amazon EC2)  
launch

# SOME AWS SERVICES FOR DEVELOPERS (10 OUT OF 55)

## COMPUTE

 Lambda 2014

 ECS (EC2 Container Service) 2014

## STORAGE

 S3 (Simple Storage Service) 2006

## DATABASE

 DynamoDB 2012

 RDS (Relational Database Service) 2009

## INTERNET OF THINGS

 IoT 2015

## MOBILE SERVICES

 SNS (Simple Notification Service)

## APPLICATION SERVICES

 API Gateway 2015

 SES (Simple Email Service) 2011

 SQS (Simple Queue Service) 2006

*Pick the right cloud*

# DIFFERENT FLAVORS FOR DIFFERENT AUDIENCES

INFRASTRUCTURE-AS-A-SERVICE (IAAS)

USER INTERFACE

APPLICATION LOGIC

DATA

MIDDLEWARE

OS

HARDWARE

NETWORK

CLOUD ENABLED IT

PLATFORM-AS-A-SERVICE (PAAS)

USER INTERFACE

APPLICATION LOGIC

DATA

MIDDLEWARE

OS

HARDWARE

NETWORK

CLOUD NATIVE DEVELOPERS

SOFTWARE-AS-A-SERVICE (SAAS)

USER INTERFACE

APPLICATION LOGIC

DATA

MIDDLEWARE

OS

HARDWARE

NETWORK

CLOUD NATIVE END USERS

*The foundation of the cloud*

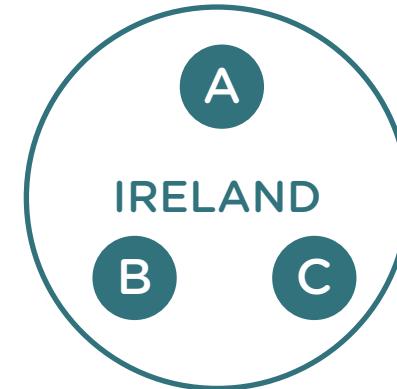
# CLOUD (AWS) INFRASTRUCTURE BASICS

## REGIONS



Each region is independent, scope for running services. No automatic transfer of data / services cross-region.

## DATA CENTERS



A region consists of multiple data centers. Multiple physically separated data centers provide fail-safe capabilities.

# DEVELOPMENT ON DIFFERENT CLOUD FLAVORS

## IAAS

- Think servers
- Work with command line / OS
- Need to plan durability (distribute over multiple data centres)
- Need to plan scalability
- Typically invoiced based on provisioned capacity

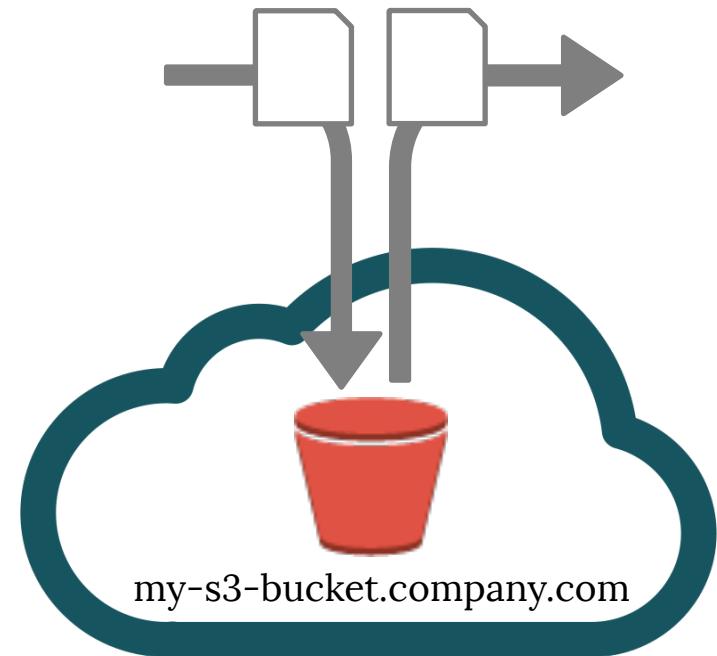
## PAAS / MANAGED SERVICES

- Think services
- Work with APIs
- Durability built into managed services
- Scalability built into managed services
- Typically invoiced per actual use

# DIVE INTO THE (AWS) MANAGED CLOUD SERVICES

# SIMPLE STORAGE SERVICE (S3)

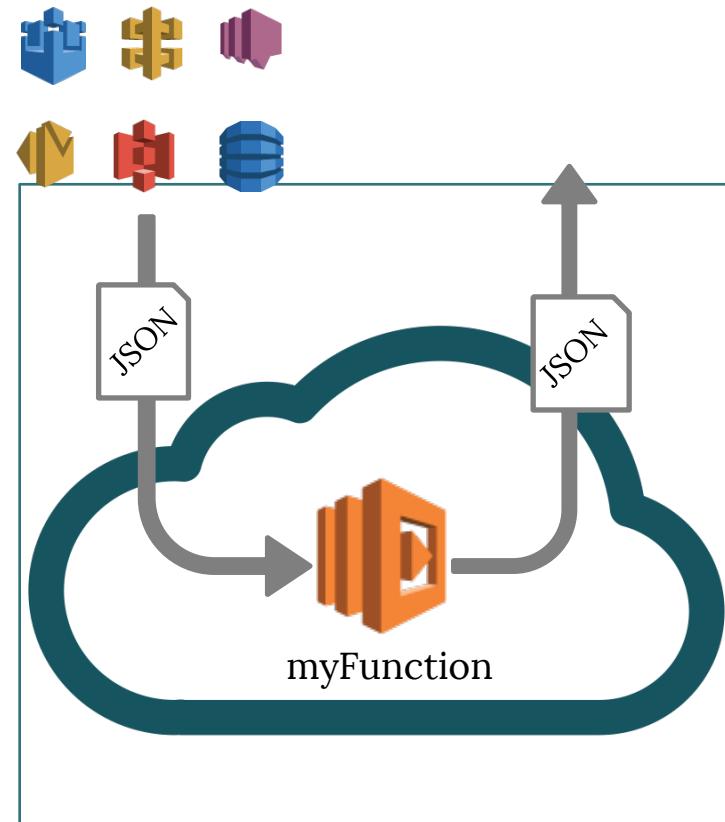
- (Unlimited) file storage service
- Application internal files
- Static web content (e.g. application HTML / CSS / JS / image assets)
- Can be complemented with CloudFront CDN to optimize costs and performance



PRICING: Storage volume + amount of requests

# AWS LAMBDA

- Compute service for running code (functions) in AWS
- Event driven (API Gateway, SNS, SES, S3, DynamoDB, Schedule, ...)
- Provision memory & max time required by single function run
- Additional "instances" spawned automatically



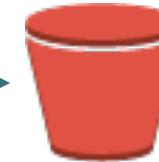
PRICING: Utilized gigabyteseconds (rounded to 100ms)

*Example 1*

# PERISCOPE CONTENT FILTER



1. Periscope app uploads 3s video clips to S3 bucket



2. A lambda function is triggered by uploads to S3 bucket



3. Lambda function filters incoming clips for adult / copyrighted content

# EXERCISE: LAMBDA ECHO

1. Open AWS Console / Lambda
2. Create new function based on the “Hello World” template
3. Name: “Echo”
4. Copy code from the right
5. Role: “Create new role from template”
6. Select policy “Simple Microservice”
7. Once created, test with some JSON input

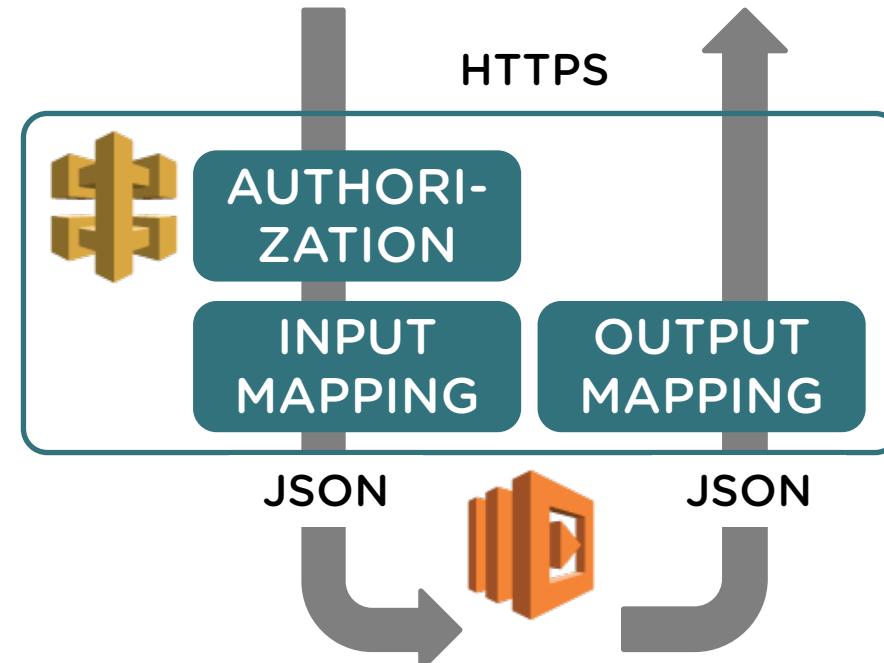
Logs are available in Cloudwatch

```
'use strict';

exports.handler = (event,
context, callback) => {
  event.now = new Date();
  console.log('Received
event:', JSON.stringify(event,
null, 2));
  callback(null, event);
};
```

# API GATEWAY

- AWS Service to implement REST (and other) APIs
- Security via API Keys, customer authorizers (Lambda)
- Connect to e.g. Lambda to publish your functions as REST interfaces
- Input / Output mapping (e.g. URL parameters -> JSON)
- No need for provisioning



PRICING: # of requests + data transfer + cache size

# EXERCISE: API GATEWAY

1. Launch API Gateway from AWS Console
2. Create API "Echo"
3. Create resource "echo" (from Actions)
4. Create "POST" method for resource "echo"
5. Integration type: Lambda Function
6. Deploy API to stage "v1"
7. Copy URL displayed for resource
8. Test API with e.g. Postman / curl

# DYNAMODB

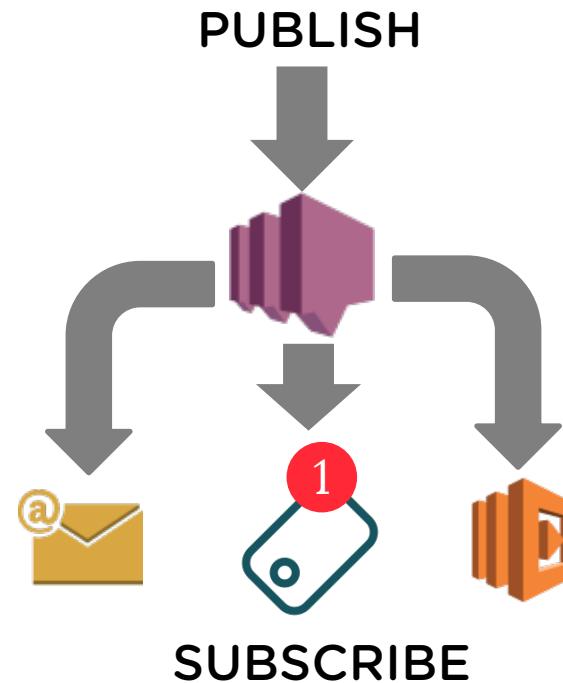
- noSQL database provided by AWS
- noSQL: scalable non-relational database with focus on speed
- Work with tables and indices, no server instances to manage
- Need to provision read / write capacity per table / index

PRICING: Provisioned read / write capacity and storage (over 25Gb)

*Serverless push notifications*

# SIMPLE NOTIFICATION SERVICE (SNS)

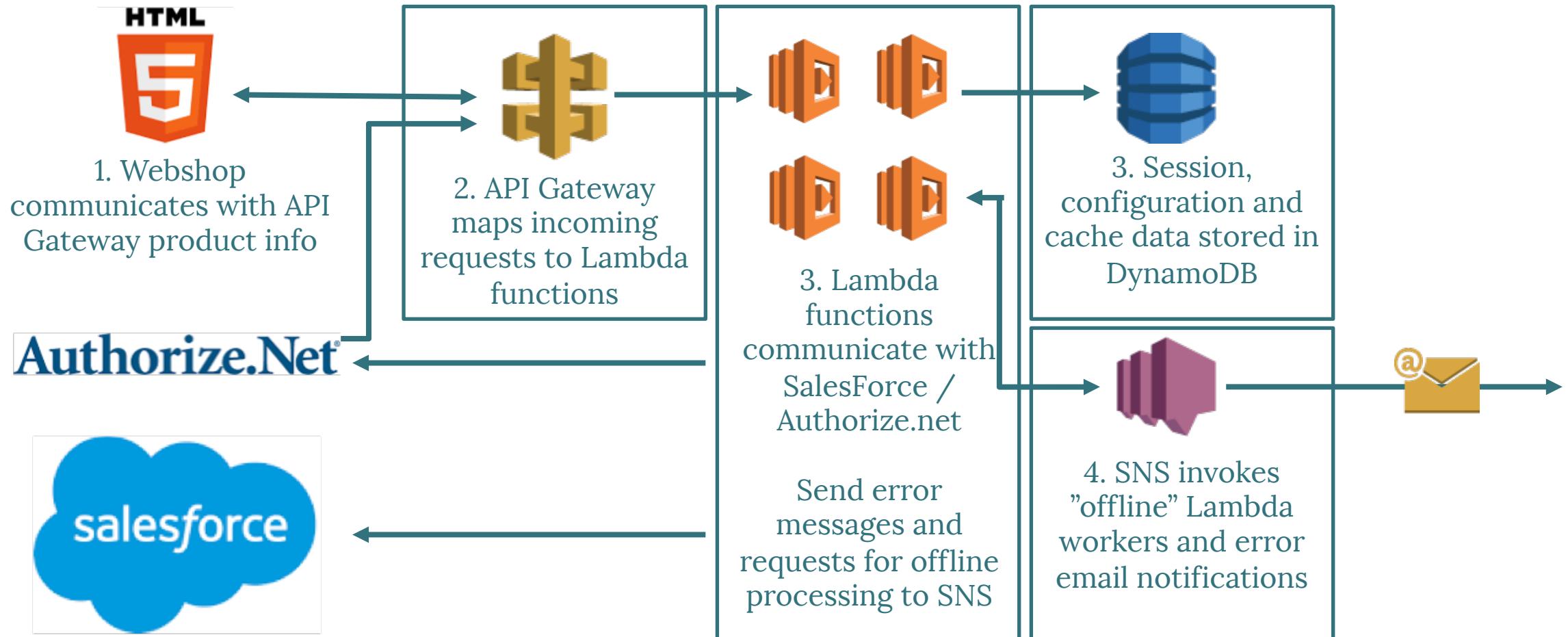
- Push notification service
- Mainly targeted for mobile notifications
- Can also be used for triggering e.g. Lambda functions, mobile, email notifications



PRICING: Amount of messages

Example 3

# HAPPY OR NOT WEBSHOP



# EXERCISE: SNS

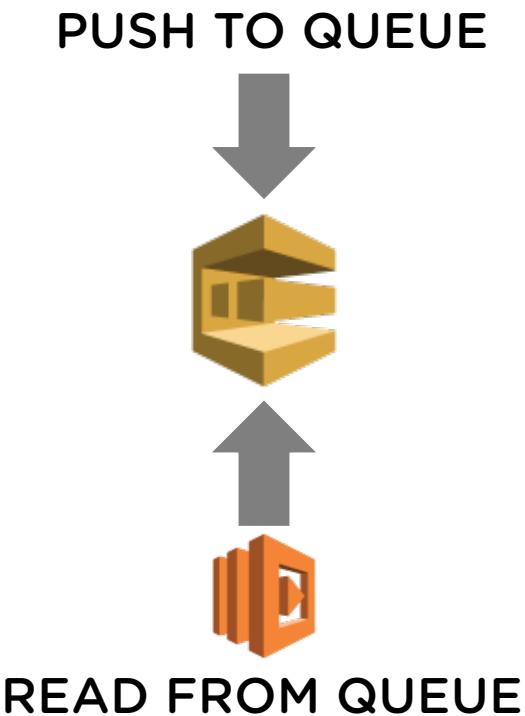
1. In AWS Console, go to Services -> SNS
2. Create new topic "SNSTestTopic"
3. Create a subscription for the Lambda function created earlier
4. Publish something to the topic
5. Go to Services -> CloudWatch
6. Open Logs for the Lambda function

Logs show the SNS message sent above (and the messages from earlier tests)

*Serverless push notifications*

# SIMPLE QUEUE SERVICE (SQS)

- Message queue service (pull)
- Delivery guaranteed
- Each message handled only once



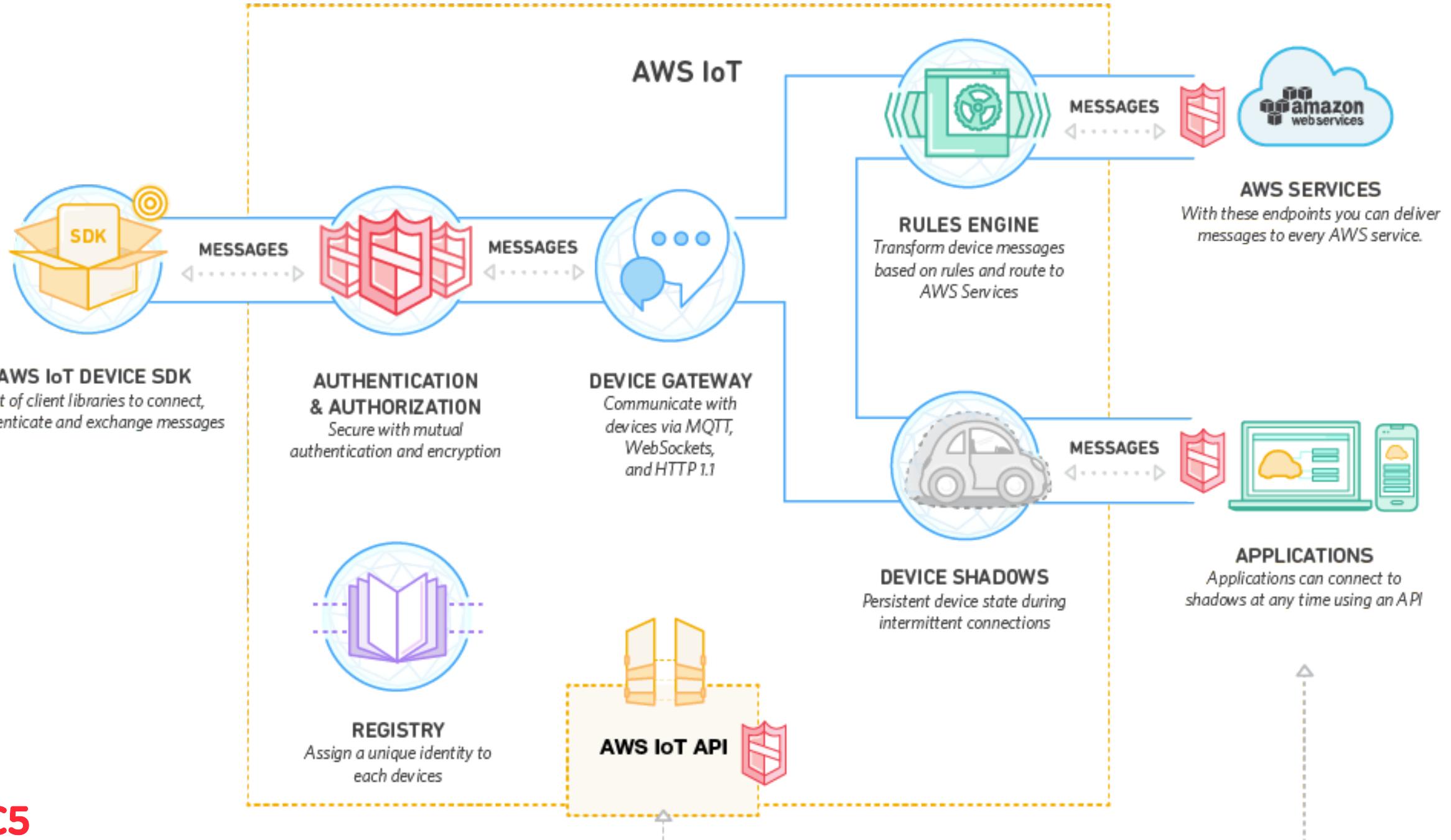
PRICING: Amount of messages

*Push notifications*

## AWS IOT

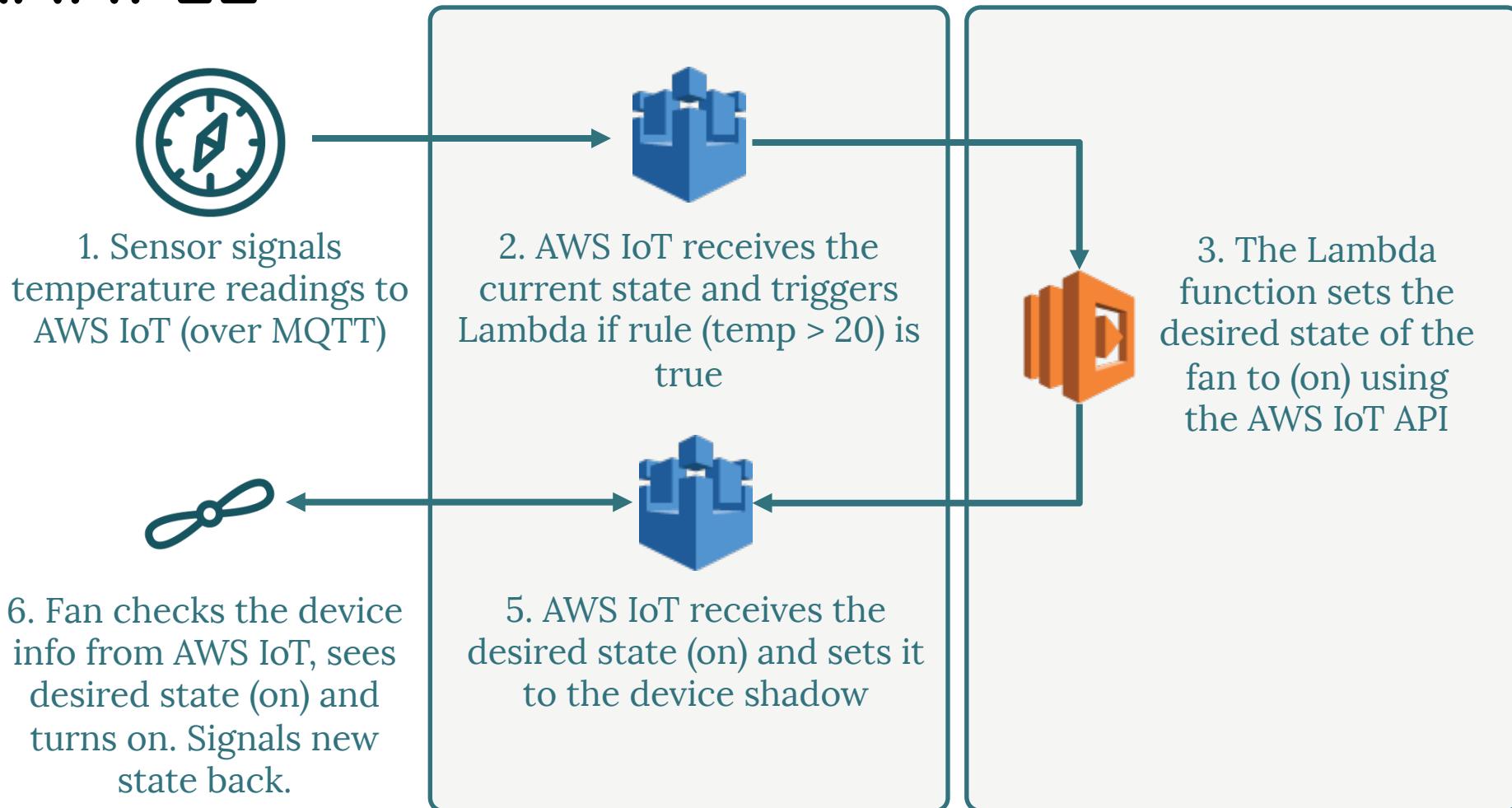
- Device registry + API for communicating with devices + automated actions (Rules)
- Authentication of devices
- Devices can send and retrieve (desired) state over MQTT
- Can perform actions based on rules (e.g. Temperature reading from a specific sensor is out of bounds)

PRICING: Amount of messages



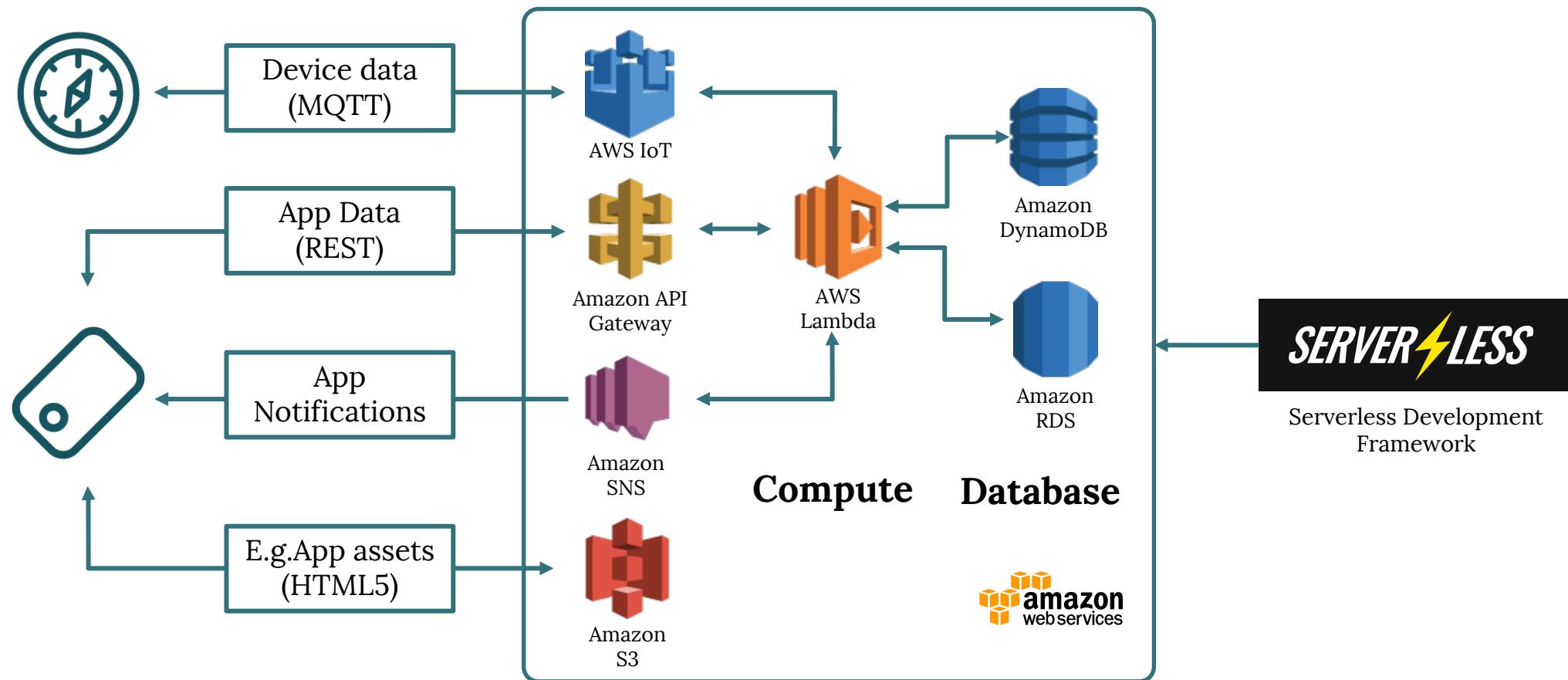
Example 4

# SIMPLE AWS IOT HOME AUTOMATION EXAMPLE



Reference Architecture

# CLOUD NATIVE APPLICATION ARCHITECTURE TOOLS À LA SC5



# OTHER CLOUD VENDORS

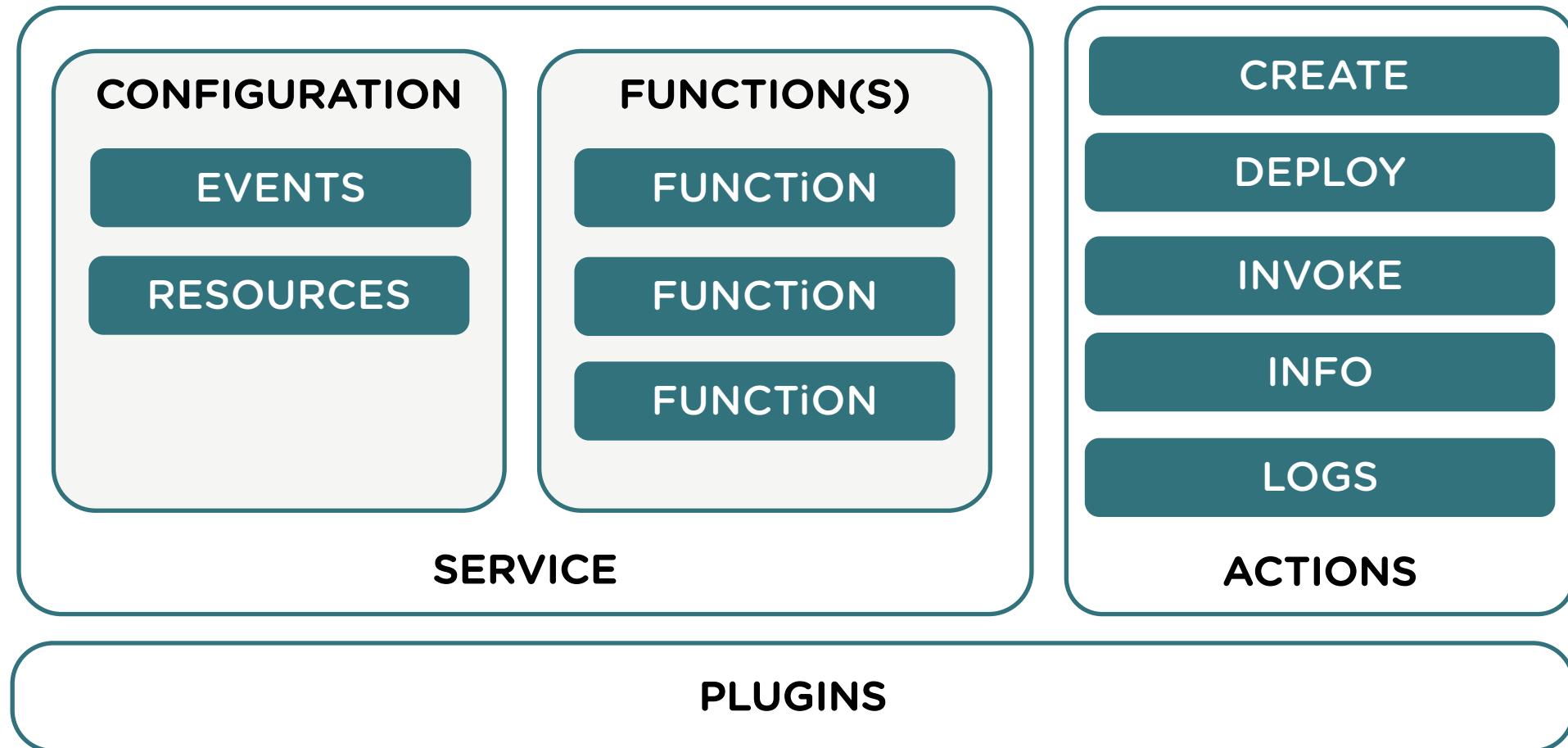
*How about other cloud vendors?*

## SIMILAR OFFERING ALSO FROM OTHER CLOUD VENDORS

AWS	MICROSOFT AZURE	GOOGLE CLOUD	IBM BLUEMIX
S3	Azure Storage	Google Cloud Storage	Object Storage
DynamoDB	DocumentDB	Google Cloud Datastore	Cloudant
Lambda	Azure Functions (preview)	Google Cloud Functions (alpha)	OpenWhisk (experimental)
SNS	Azure Notification Hub	Google Cloud Pub/Sub	IBM Push Notifications
SQS	Azure Queue Storage	Google Cloud Pub/Sub	IBM MQ
IoT	Azure IoT Suite	Use GC building blocks	Watson IoT

# INTRODUCTION TO SERVERLESS FRAMEWORK 1.0

# SERVERLESS FRAMEWORK 1.0



# GETTING SET UP

Instructions for getting set up:

<http://serverless.fi/docs/workshop-preps>

# LEARNING SERVERLESS BY DOING

# CREATING A SERVICE

- We'll start by creating a service "slsIntro"

```
> sls create -t aws-nodejs -n slsIntro -p slsIntro
```

- Parameters explained:
  - **-t aws-nodejs** : use template for create Node.js based AWS service
  - **-n slsIntro** : name the service "slsIntro"
  - **-p slsIntro** : create the service into directory slsIntro

# AWS-NODEJS STRUCTURE

event.json	Input file for invoke action
handler.js	(Default) source file for handler.js
serverless.yml	Service configuration (functions, endpoints, resources)

Lets's check contents for each file!

# TUNE THE FUNCTION

- Dump the response also with `console.log` for debugging purposes
- Add e.g. `date: new Date()` to the body of your response.

# ADD SERVERLESS-MOCHA-PLUGIN TO YOUR SERVICE (FOR LOCAL DEBUGGING)

- Initialize your package.json

```
> npm init
```

- Install serverless-mocha-plugin

```
> npm install --save-dev serverless-mocha-plugin
```

Register the plugin to `serverless.yml` by adding the following:

```
plugins:
```

```
  - serverless-mocha-plugin
```

# CREATE AND INVOKE SERVERLESS-MOCHA-PLUGIN TESTS

- Create a test for function hello with

```
> sls create test -f hello
```

- Check the test contents from test/hello.js

- Invoke the test with

```
> sls invoke test -f hello
```

- Fix test to pass

# DEFINE HTTP ENDPOINTS FOR FUNCTION

- In serverless.yml, add the event definitions (below in red)
- Note: rooted path (e.g. /hello) do not currently work

```
functions:  
  hello:  
    handler: handler.hello  
    events:  
      - http:  
          path: hello  
          method: get  
      - http:  
          path: hello  
          method: post
```

# DEPLOY THE FUNCTIONS AND ENDPOINTS

- Deploy the function with

```
> sls deploy
```

- You could use `-s <stage>` to deploy to other stages (e.g. prod)
- Use Postman / curl to test the endpoints. The endpoint URL is available after deployment or with

```
> sls info
```

- Log output with

```
> sls logs -f hello -t
```

- See what happens to the event with the default API Gateway mapping in Serverless 1.0
- Try adding `integration: lambda` to your endpoints and redeploy

# ADD SERVERLESS-OFFLINE PLUGIN TO YOUR SERVICE (FOR LOCAL DEBUGGING)

- Install the serverless-offline plugin

```
> npm install --save-dev serverless-offline
```

- Register the plugin to `serverless.yml` by adding the following to `plugins`:

```
- serverless-offline
```

- Start the and try the endpoint

```
> sls offline start
```

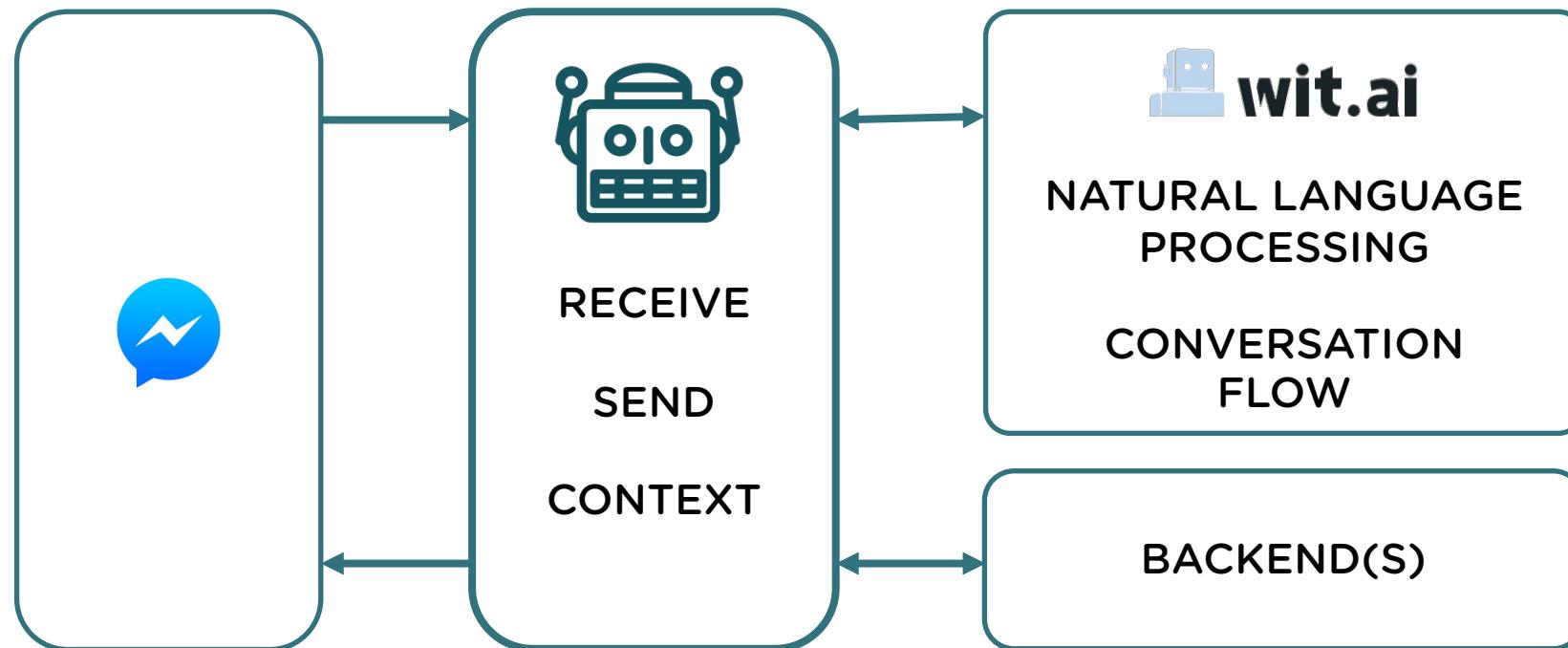
```
> curl http://localhost:3000/dev/hello?foo=bar
```

# SERVERLESS-WEBPACK PLUGIN

- Optimize package size and ensure that all 3rd party modules are included in your deployment
- We'll check that with during the bot exercise

# INTRODUCTION TO MESSENGER BOT COMPONENTS

# MESSENGER BOT ARCHITECTURE



# MESSENGER PLATFORM

Messenger Platform Beta Launched April 2016

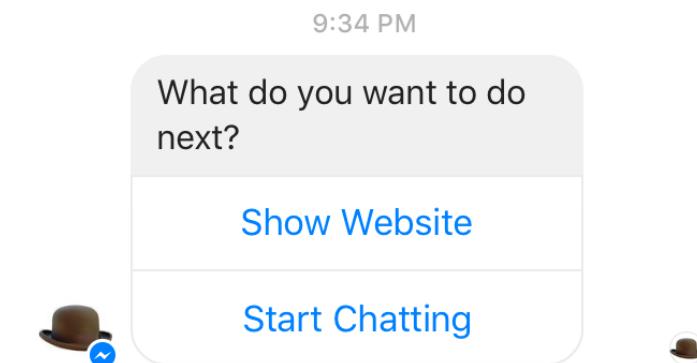
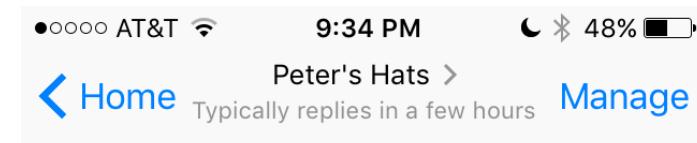
Over 1Bn monthly users

Over 33k bots

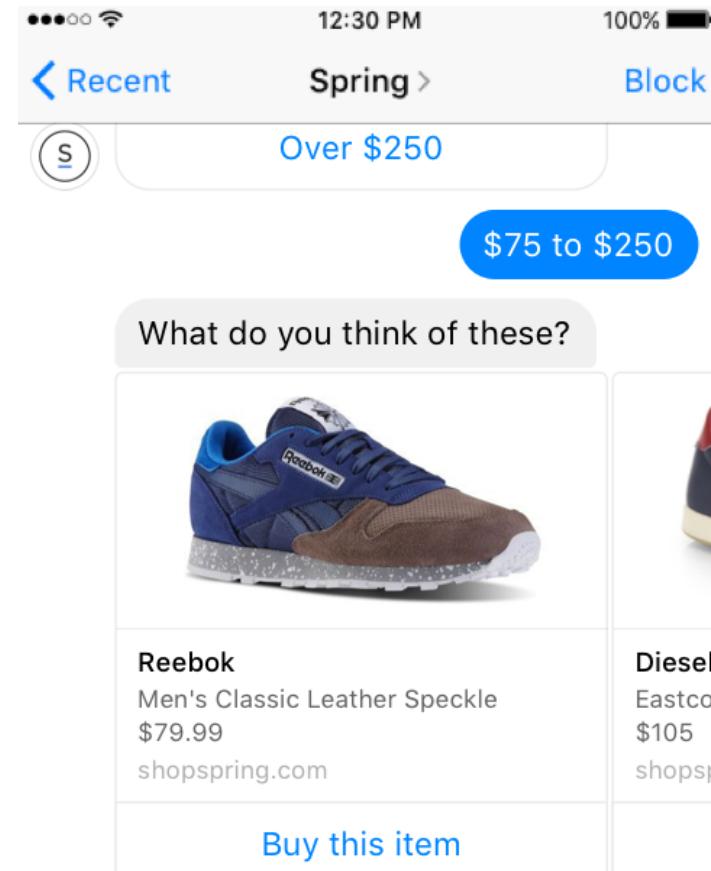
# RICH UI ELEMENTS: TEXT



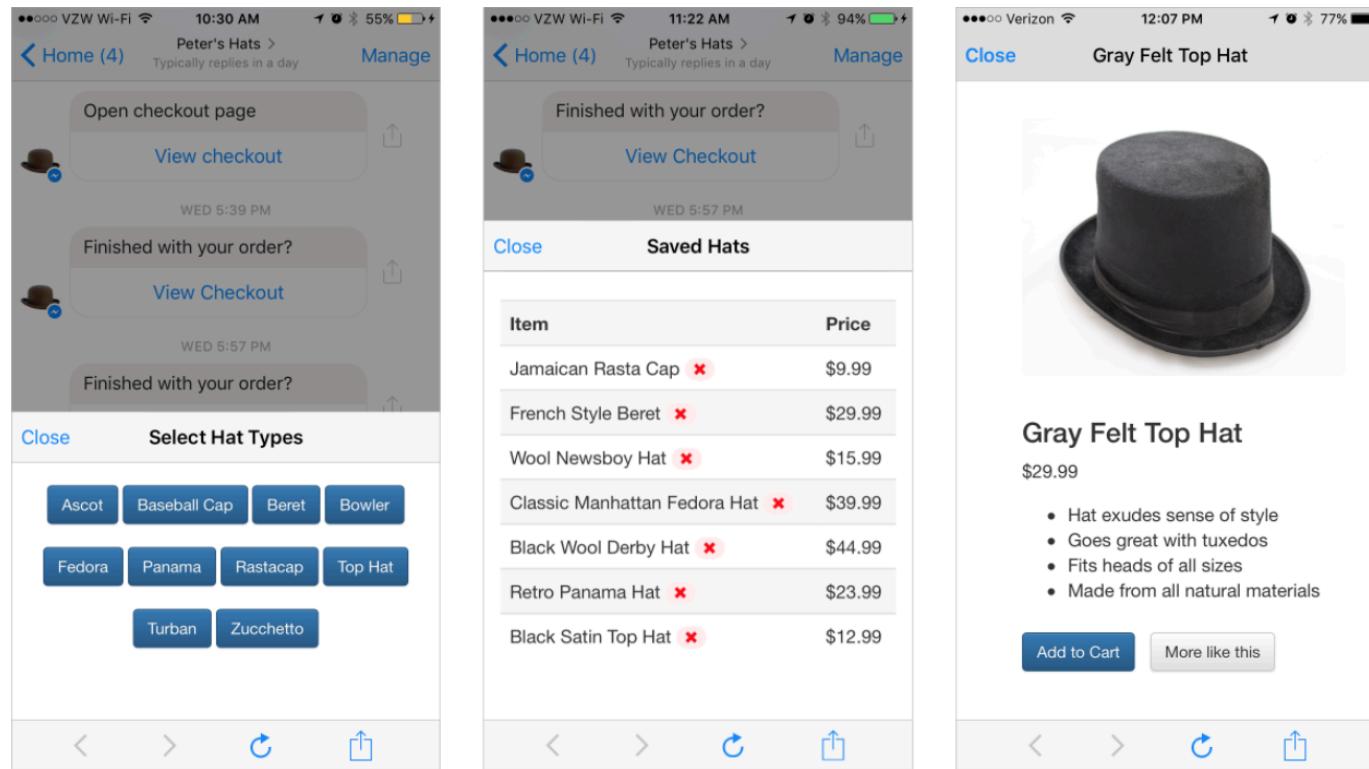
# RICH UI ELEMENTS: BUTTONS



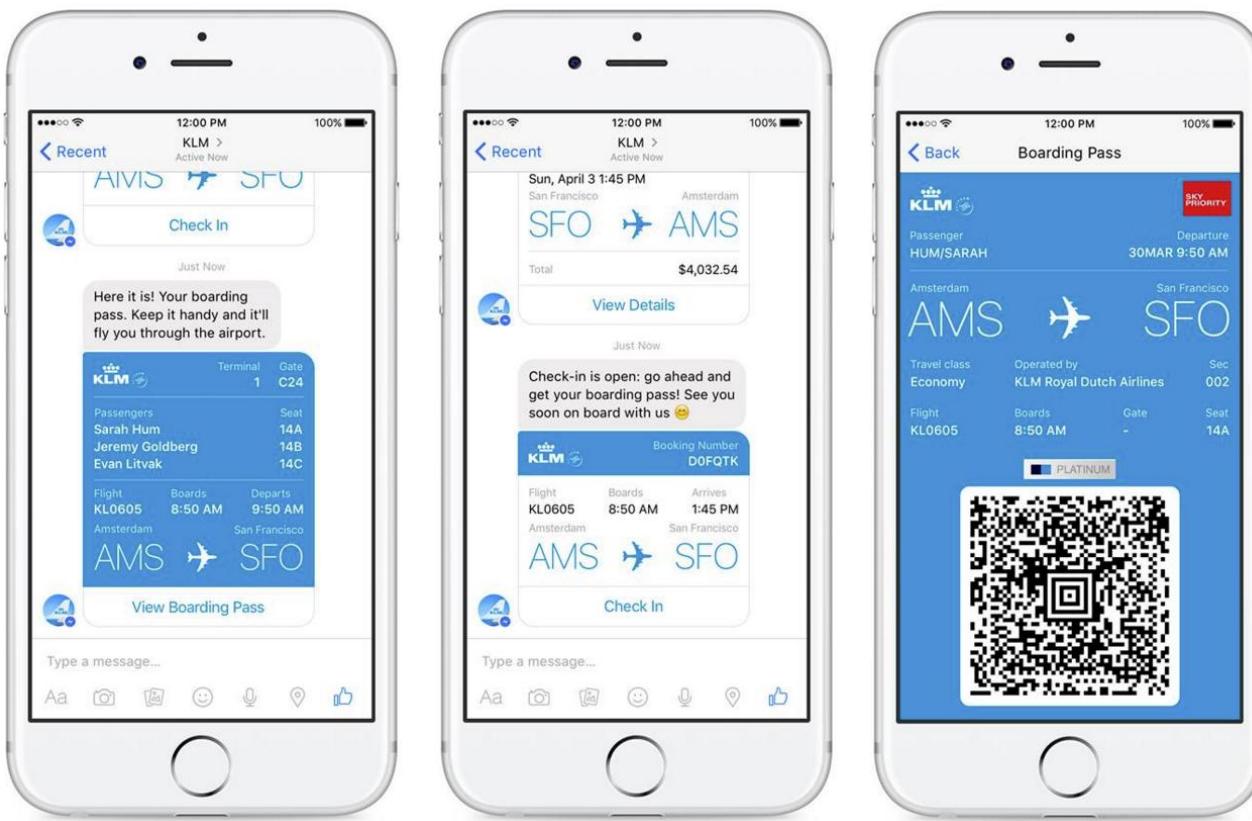
# RICH UI ELEMENTS: CAROUSEL



# RICH UI ELEMENTS: WEB VIEWS



# EXAMPLE: KLM MESSENGER



# MESSENGER BOT SETUP QUICK GUIDE

1. Create a Facebook page
2. Register a Facebook application
3. Create an endpoint for your bot and hook it to the Facebook app
4. Listen to messages from Messenger
5. Post back responses to the Messenger Platform
6. Get your Facebook application approved to reach the public

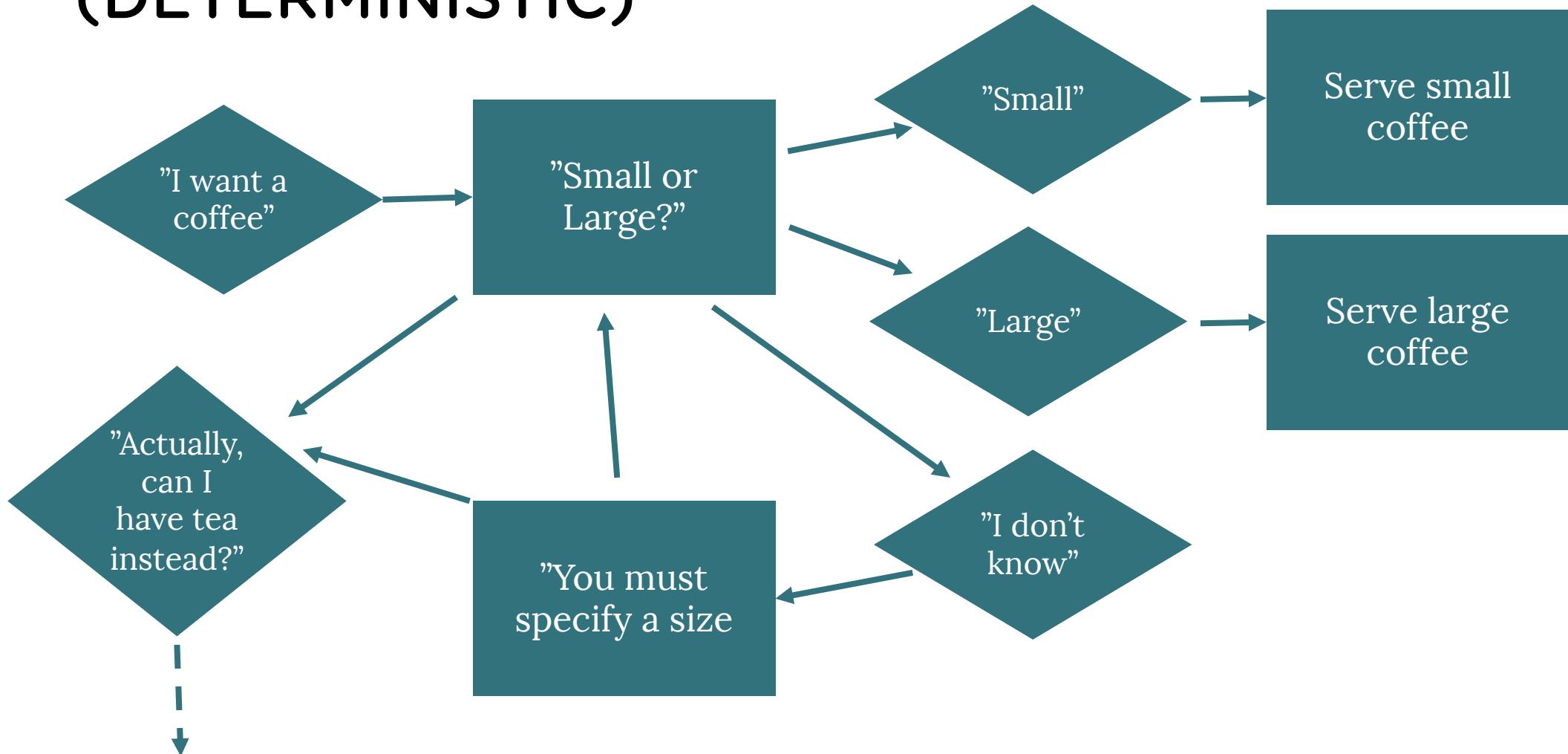
Messenger Platform API documentation available at:

<https://developers.facebook.com/docs/messenger-platform/>

WIT.AI

SC5

# CONVERSATION WORKFLOW (DETERMINISTIC)



# WIT.AI APPROACH

- Stories to define "rules"/"heuristics" for the conversation
  - Context to define current state
- => No need for hard coded flows. A combination of stories with rules based on context allow to define complex conversations

# CONTEXTS

"I want a coffee"

```
{  
  "user_id": 123435,  
  "name": "John Smith",  
  "intent": "purchase",  
  "order": {  
    "type": "coffee"  
    "qty": 1  
  }  
}
```

"I want a large coffee"

```
{  
  "user_id": 123435,  
  "name": "John Smith",  
  "intent": "purchase",  
  "order": {  
    "type": "coffee"  
    "size": "large"  
    "qty": 1  
  }  
}
```

# NATURAL LANGUAGE PROCESSING



```
{  
  "user_id": 123435,  
  "name": "John Smith",  
  "intent": "weather",  
  "date": "2016-10-27T08:11:42.222Z"  
}
```



```
{  
  "user_id": 123435,  
  "name": "John Smith",  
  "intent": "weather",  
  "location": "London"  
}
```

# SAMPLE WIT.AI CONVERSATION (1)

## REQUEST

What's the weather in Brussels?

## RESPONSE

```
{  
  "type": "merge",  
  "entities": {  
    "location": [{  
      "body": "Brussels",  
      "value": {  
        "type": "value",  
        "value": "Brussels",  
        "suggested": true},  
      "start": 11,  
      "end": 19,  
      "entity": "location"  
    }]  
  "confidence": 1  
}
```

# SAMPLE WIT.AI CONVERSATION (2)

## REQUEST

```
{  
  "loc": "Brussels"  
}
```

## RESPONSE

```
{  
  "type": "action",  
  "action": "fetch-forecast"  
}
```

# SAMPLE WIT.AI CONVERSATION (3)

## REQUEST

```
{  
  "loc": "Brussels",  
  "forecast": "Sunny"  
}
```

## RESPONSE

```
{  
  "type": "msg",  
  "msg": "It's gonna be sunny in Brussels"  
}
```

The *node-wit* module provides higher level method *runActions()* that hides the logic of iterating the contexts with Wit.ai.

Wit.ai HTTP API documentation available at:

<https://wit.ai/docs/http>

Node.js SDK available at

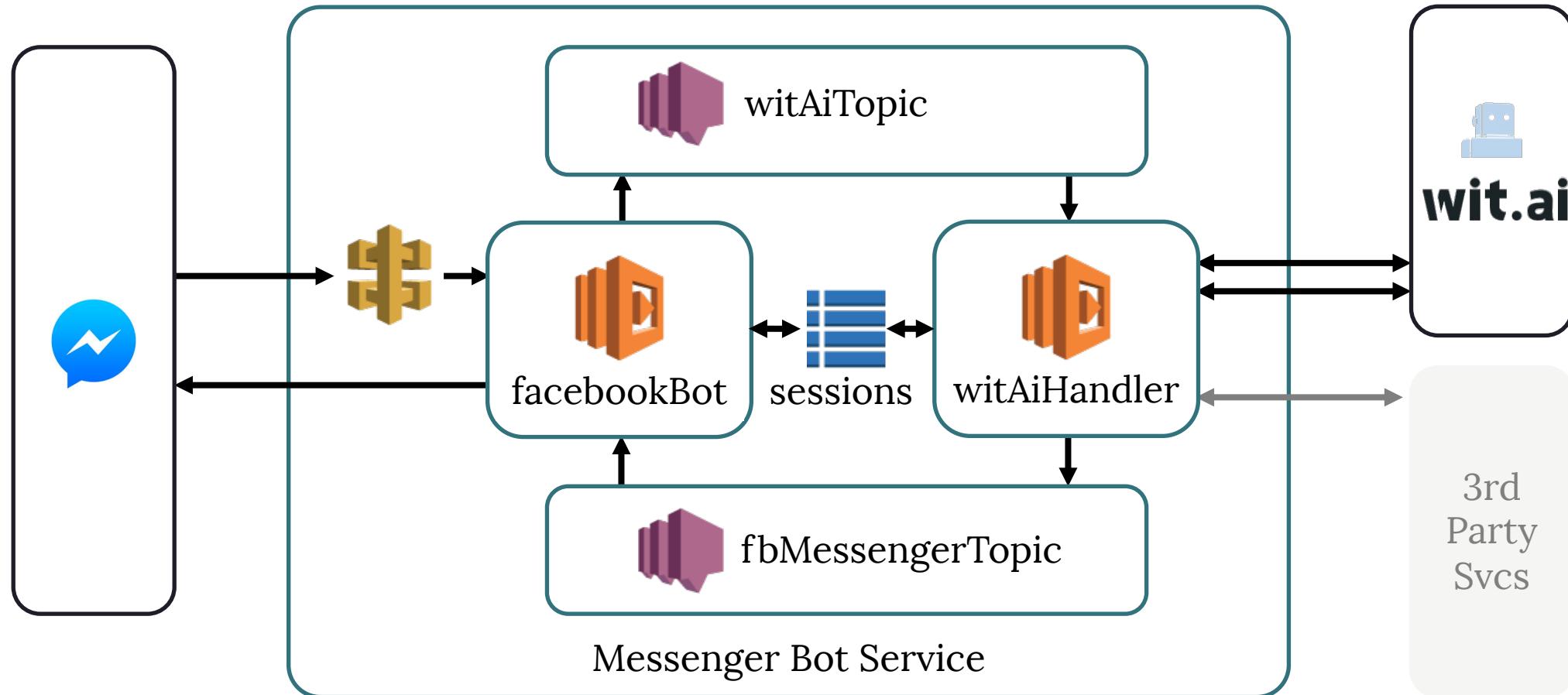
<https://github.com/wit-ai/node-wit>

# OUR BOILERPLATE

serverless-messenger-boilerplate and its documentation  
available at

<https://github.com/SC5/serverless-messenger-boilerplate>

# BOILERPLATE ARCHITECTURE



# BOILERPLATE STRUCTURE

example.env

Template for .env (Facebook / Wit.ai secrets) <= COPY TO .env

facebook-bot/

facebookBot function

fb-messenger.js

Facebook Messenger interop

handler.js

Function entrypoint

lib/

Common libraries used by both functions

messageQueue.js

Library for reading / publishing to SNS

session.js

Library for reading / updating user session (DynamoDB)

package.json

ADD NODE MODULES REQUIRED BY YOUR BOT HERE

serverless.yml

Service configuration (functions, endpoints, resources)

test/

Serverless-mocha-plugin tests

facebookBot.js

Tests for facebookBot function

witAiHandler.js

Tests for witAiHandler function

webpack.config

Configuration for Webpack deployment (serverless-webpack plugin)

wit-ai/

witAiHandler function

handler.js

Entrypoint for function

my-wit-actions.js

Your bot logic (Wit.ai actions) <= IMPLEMENT YOUR BOT LOGIC HERE

wit-ai.js

Bot logic built interop on Wit.ai

# .ENV FILE (FROM EXAMPLE.ENV)

FACEBOOK\_BOT\_VERIFY\_TOKEN

User defined verification token for the Facebook App

FACEBOOK\_BOT\_PAGE\_ACCESS\_TOKEN

Facebook-generated access token for the page

WIT\_AI\_TOKEN

API token for Wit.ai

FACEBOOK\_ID\_FOR\_TESTS

Facebook ID used to post messages in tests. Available from the sessions table

SERVERLESS\_PROJECT

Service name (keep in sync with service name in serverless.yml)

# DECOUPLING WITH SNS

- Lambda SNS subscriptions: see [serverless.yml](#)
- Posting and decoding messages done with [messageQueue.js](#)
- Sample endpoint at [facebookBot/handler.js](#)
- In production environments, SNS could be strengthened e.g. with SQS to guarantee delivery

# LOCAL DEVELOPMENT AND TESTING (SERVERLESS-MOCHA-PLUGIN)

- serverless-mocha-plugin included in boilerplate with predefined tests for facebookBot and witAiHandler functions in the test/ directory
- Run all tests : *sls invoke test*
- Run tests for facebookBot : *sls invoke test -f facebookBot*
- Run tests for witAiHandler: *sls invoke test -f witAiHandler*
- (Create new tests with *sls create test -f functionName*)
- Comment out line that sets *process.env.SILENT* in *test/\*.js* if you want messages to be sent during testing

## OPTIMIZED DEPLOYMENT (SERVERLESS-WEBPACK)

- serverless-webpack plugin included to streamline deployment package and accelerate cold start of Lambda functions
- Pre-configured in webpack.config

# BUILDING A WEATHER BOT

Code snippets for the workshop are available from  
<http://serverless.fi/docs/messenger-workshop/>

# SETUP PROJECT

# 1. CREATE SERVERLESS PROJECT

```
> sls install -u https://github.com/SC5/serverless-messenger-boilerplate  
> mv serverless-messenger-boilerplate weather-bot
```

Change service name to “weather-bot” in `serverless.yml` and `.env`, then install node modules and copy `example.env` to `.env`

```
> npm install  
> cp example.env .env
```

Generate a random verification token to `VERIFY_TOKEN` in `.env`

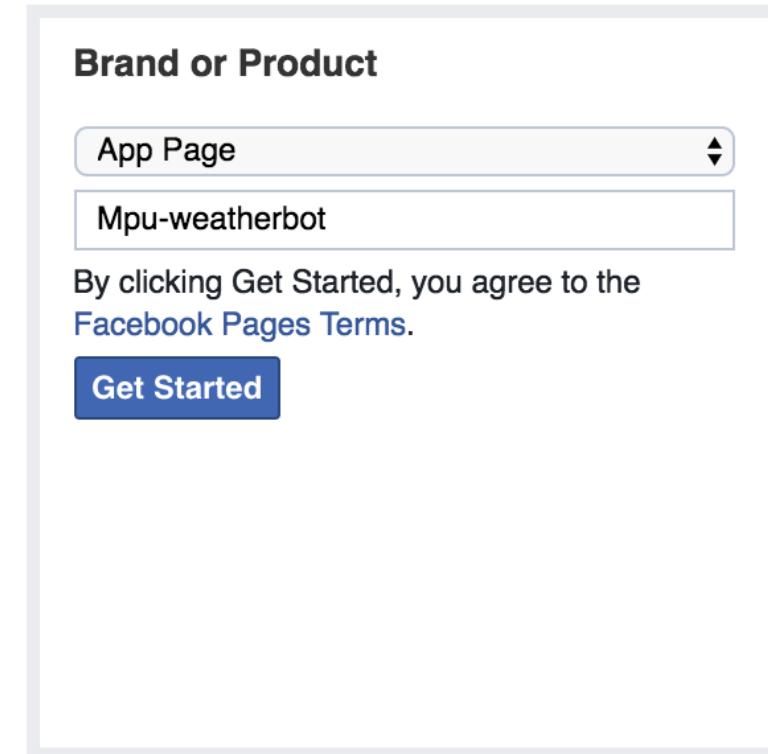
Deploy the service and memorize the URL

```
> sls deploy
```

# SETUP FACEBOOK PAGE AND APPLICATION

# CREATE A FACEBOOK PAGE

- Go to  
<http://facebook.com/pages/create>
- Create new page
- Skip all following steps



# CREATE FACEBOOK APPLICATION

- Go to <https://developers.facebook.com/quickstarts/?platform=web>
- Click "Skip and Create App ID"
- Fill in info
- Create app ID

## Create a New App ID

Get started integrating Facebook into your app or website

Display Name

Contact Email

Category

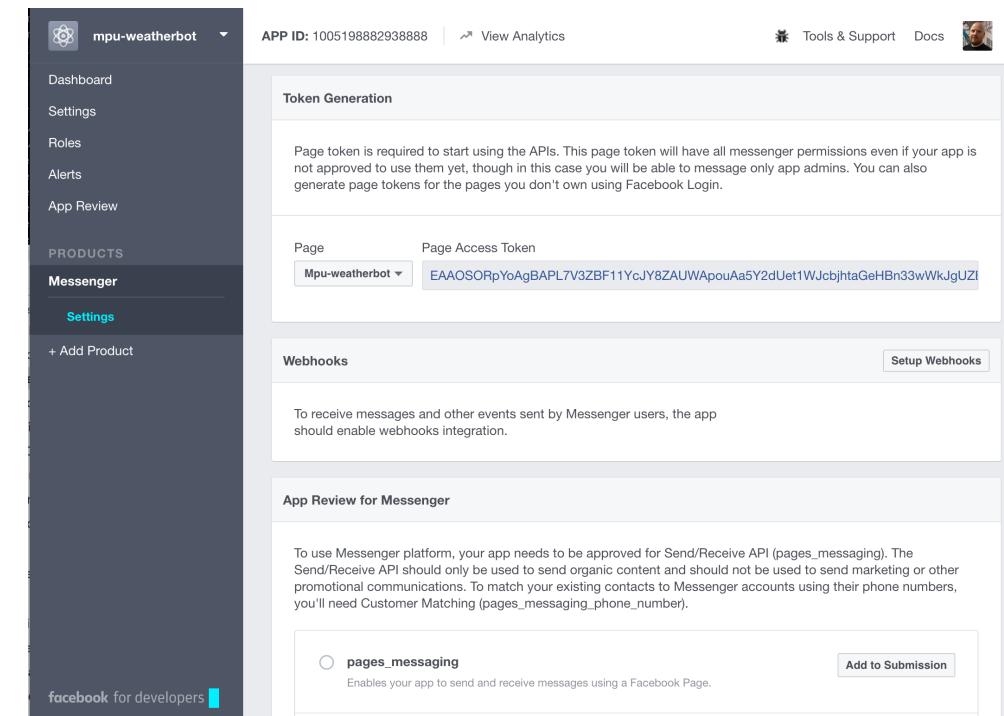
By proceeding, you agree to the [Facebook Platform Policies](#)

[Cancel](#)

[Create App ID](#)

# SETUP MESSENGER TOKEN

- "Get Started" for Messenger
- Generate a token for your page
- Copy token to .env (FACEBOOK\_BOT\_TOKEN)

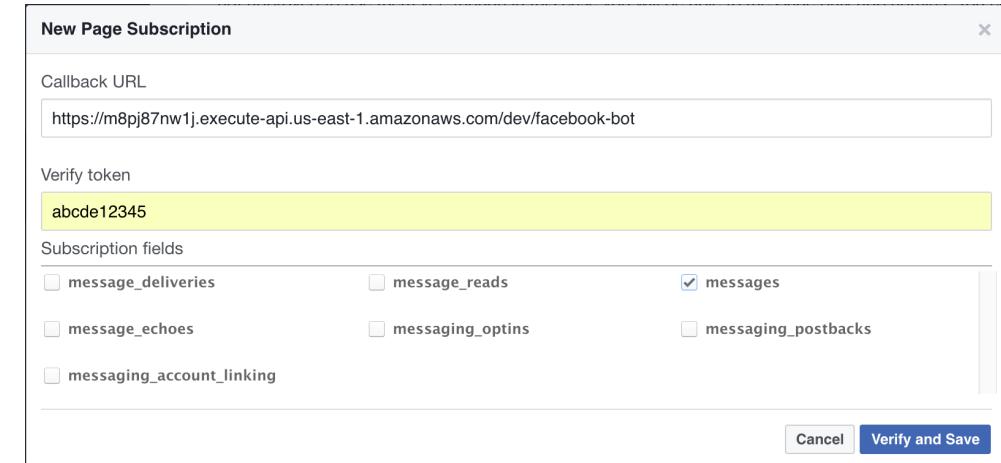


# SETUP MESSENGER WEBHOOK

- Initiate "Setup Webhooks"
- Enter the endpoint URL that you got during deployment
- Enter your verify token from .env (FACEBOOK\_VERIFY)
- Deploy

```
> sls deploy
```

- Verify and Save



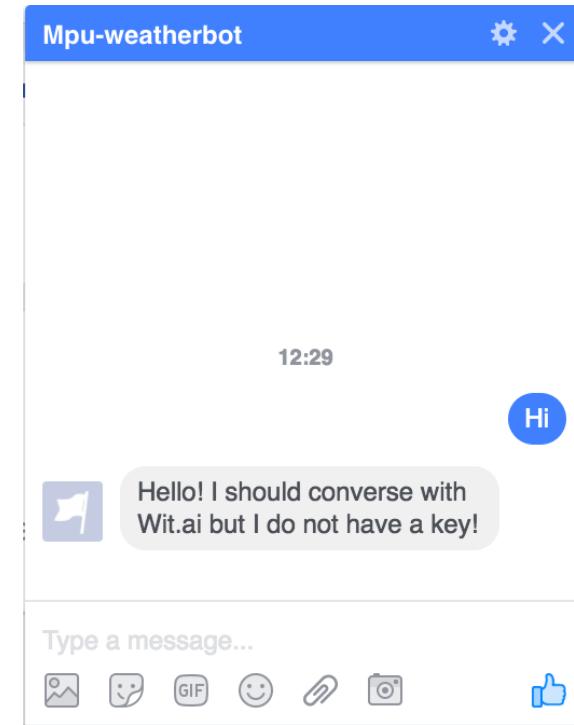
# SUBSCRIBE THE WEBHOOK TO THE FACEBOOK PAGE

1. Under "Webhooks", select your page and subscribe

The screenshot shows the Facebook App Dashboard under the 'Webhooks' tab. At the top, there are fields for 'Page' (with a dropdown menu labeled 'Select a Page') and 'Page Access Token' (with a note: 'You must select a Page to generate an access token'). Below this, the 'Webhooks' section is visible, featuring a summary of selected events and a 'Subscribe' button. The summary includes: 'To receive messages and other events sent by Messenger users, the app should enable webhooks integration.' (with a green checkmark and 'Complete'), 'Selected events: messages', and 'Select a page to subscribe your webhook to the page events' (with a dropdown menu 'Mpu-weatherbot' and a 'Subscribe' button). At the bottom, there is a section for 'App Review for Messenger'.

# TRY IT

- Send message to your Facebook page



# FACEBOOK APPLICATION ACCESS

Access to unapproved applications is limited to developers only.

If you want to give access to other users, you should provide "Developer" / "Testers" roles to them from the "Roles" panel (accessible from the left sidebar)

# SET UP OPENWEATHER API

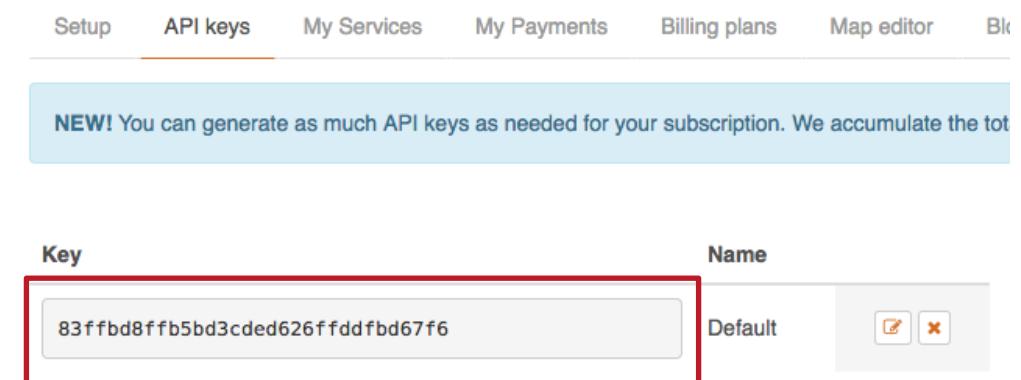
# REGISTER TO OPENWEATHERMAP

1. Go to

<https://openweathermap.org/appid>

2. Sign up

3. Copy API key and set it to  
WEATHER\_API\_TOKEN in .env



# SETUP WIT.AI

# REGISTER + CREATE WIT.AI APP

- Go to <https://wit.ai>
- Register (if not already done)
- Create a new App

The screenshot shows the 'Create a new App' page on the wit.ai website. At the top, there's a header with the wit.ai logo, a user profile icon, and navigation links for 'Apps', 'Docs', 'Explore', 'Help', and a user account. Below the header, the page title 'Create a new App' is displayed. A breadcrumb navigation shows 'mpuittinen / mpu-weatherbot'. The main form area has a text input field containing 'Weather bot for Serverless Messenger Workshop'. A 'Language' dropdown menu is set to 'English'. Below the input field, there are two radio button options: 'Open' (selected) and 'Private'. The 'Open' option is described as 'Your data will be open to the community'. The 'Private' option is described as 'Your data will be private and accessible only by you and the developers you decide to share your app with.' At the bottom of the form, there's a 'Browse...' button for importing a backup and a blue 'Create App' button.

# CONNECT THE ENDPOINT WITH WIT.AI

- Go to Settings
- Copy the Server Access Token ID to .env (WIT\_AI\_TOKEN)
- Deploy your service

```
> sls deploy
```

The screenshot shows the 'Change App Details' and 'API Details' sections of the Wit.ai developer console.

**Change App Details:**

- App name: mpu-weatherbot
- Default Timezone: America/Los\_Angeles
- Language: English

**API Details:**

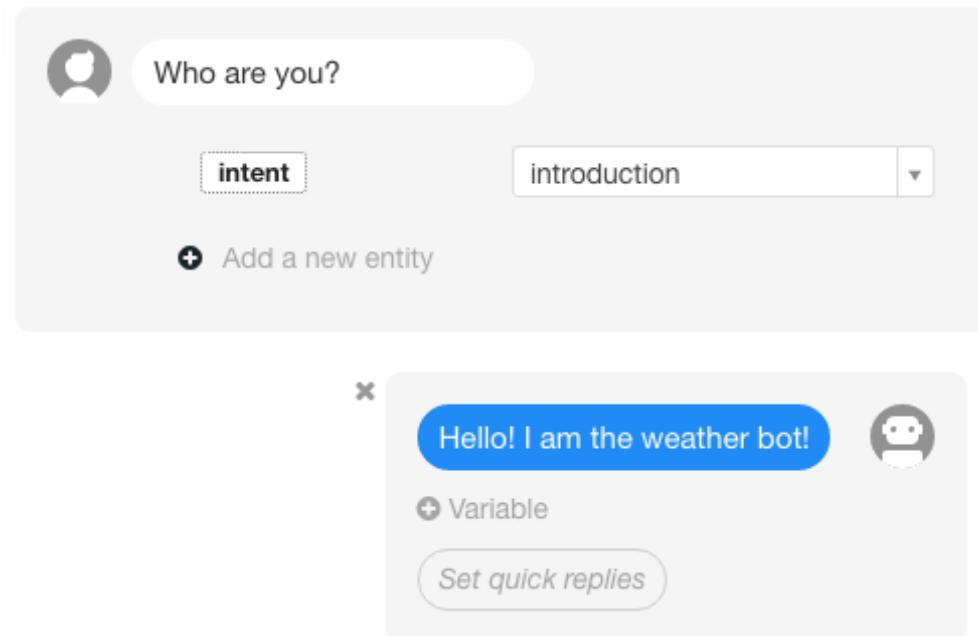
You can use the tokens below to start making API requests from your app. Learn more through the guide, or contact us at anytime. We look forward to what you create :)

App ID	58060c42-102c-4053-91fa-637678afbb50
Server Access Token ⓘ	DPGGWLE5RGUNDPOONJJAPNTS5ZBIJJS
Client Access Token ⓘ	[redacted]
...	[redacted]

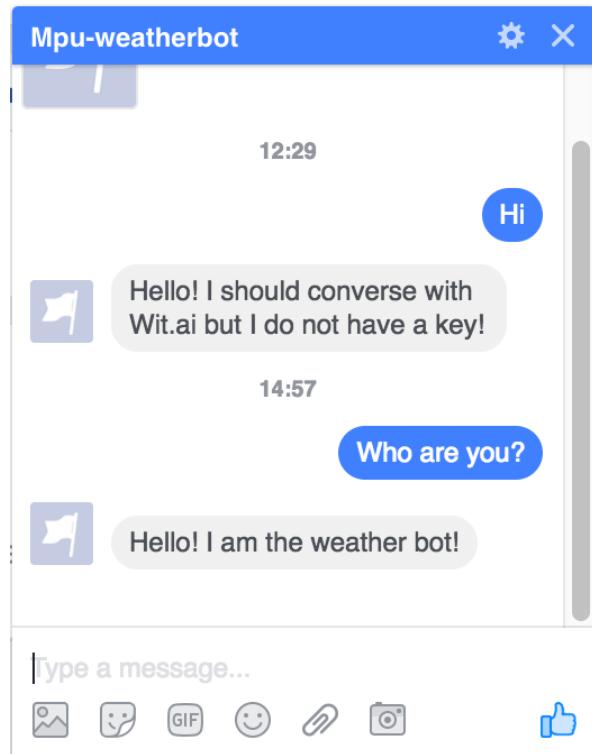
A red box highlights the 'Server Access Token' field, which contains the value 'DPGGWLE5RGUNDPOONJJAPNTS5ZBIJJS'.

# LET THE BOT INTRODUCE ITSELF

- Create a new story
- Add user input and intent
- Add a response with "Bot Sends"
- Remember to save the story

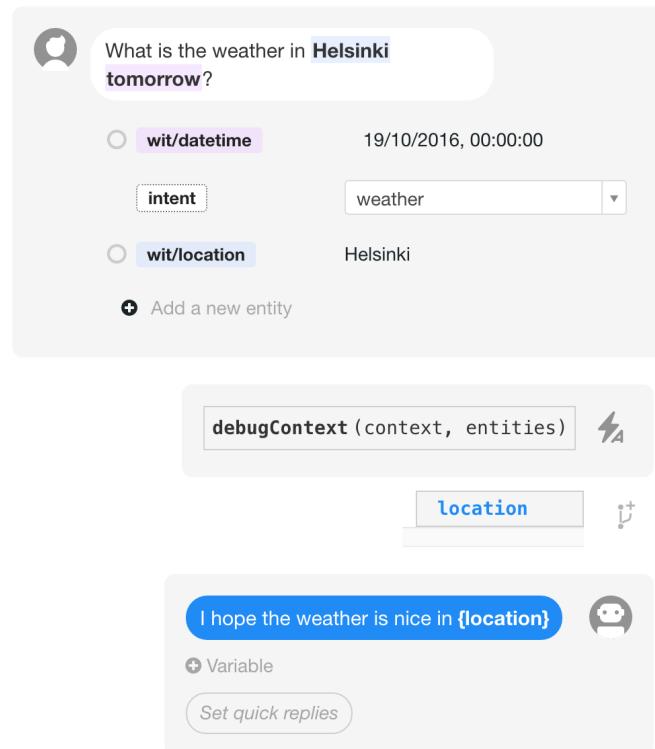


# TRY IT!

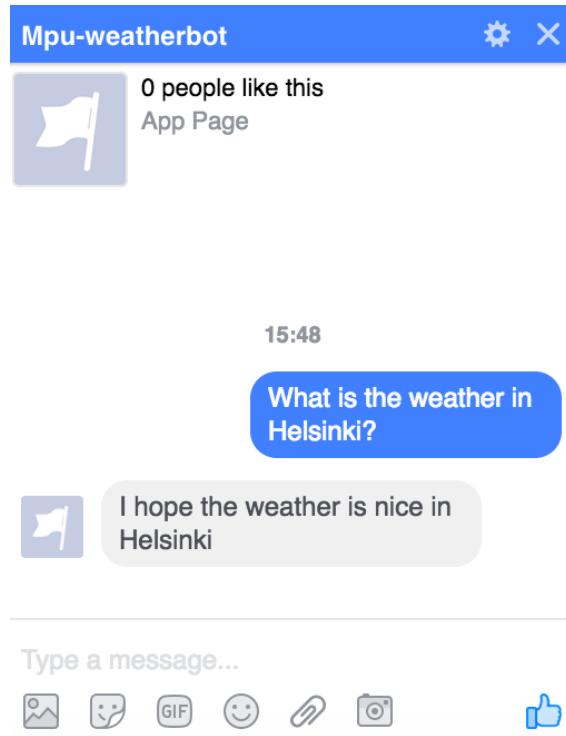


# WEATHER STORY (WITH FIXED RESPONSE)

- Add a new story
- Type in your user input
- Add entity *intent* with value *weather*
- Select the location and add new entity *wit/location*
- Select the date and add new entity *wit/datetime*
- Add action *debugContext* with "Bot Executes"
- Set context key *location*
- Add a response using "Bot sends"
- Save



# TRY IT (WITH DEBUG)!

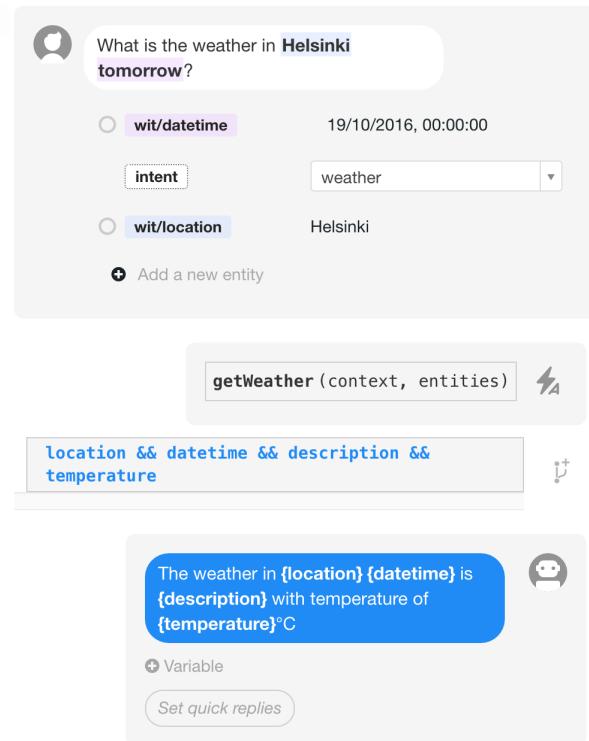


```
> sls deploy function -f facebookBot
```

```
START RequestId: bc745ce1-9530-11e6-8a4c-f7159450ad15 Version: $LATEST
2016-10-18 15:45:23.932 (+03:00) bc745ce1-9530-11e6-8a4c-f7159450ad15 WIT.AI DEBUG:
2016-10-18 15:45:23.932 (+03:00) bc745ce1-9530-11e6-8a4c-f7159450ad15 {
  "sessionId": "948003351971940-1476794723470",
  "context": {},
  "text": "What is the weather in Helsinki?",
  "entities": {
    "location": [
      {
        "confidence": 0.9998750679246946,
        "type": "value",
        "value": "Helsinki",
        "suggested": true
      }
    ],
    "intent": [
      {
        "confidence": 0.9995693812970181,
        "value": "weather"
      }
    ]
  }
}
END RequestId: bc745ce1-9530-11e6-8a4c-f7159450ad15
REPORT RequestId: bc745ce1-9530-11e6-8a4c-f7159450ad15 Duration: 684.66 ms Billed Duration:
700 ms Memory Size: 1024 MB Max Memory Used: 63 MB
```

# ADD LOGIC TO THE STORY

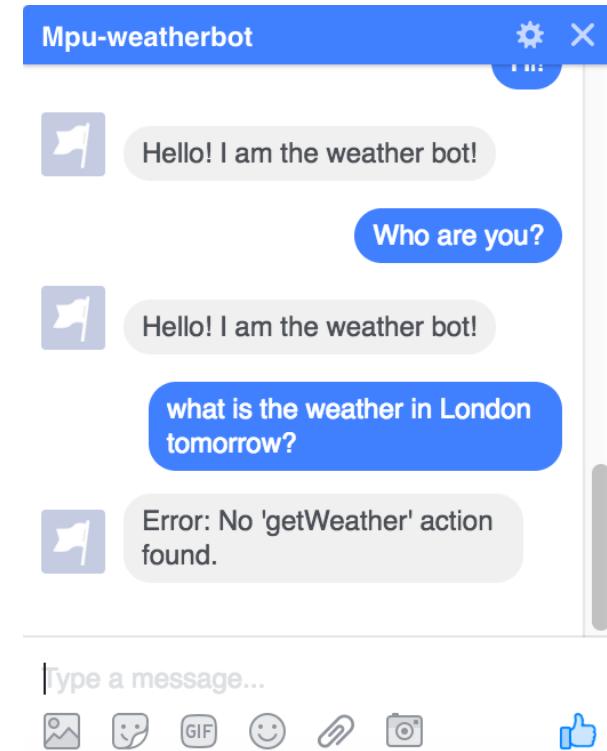
- Remove action *debugContext* and context key *location*
- Add action *getWeather*
- Add context key *location && datetime && description*
- Add response



# TRY IT!

The boilerplate returns an error because the *getWeather* action has not been implemented.

The earlier *debugContext* action is built-in the boilerplate.



# IMPLEMENT LOGIC FOR FETCHING WEATHER

- Copy snippet "getWeather.js" to *my-wit-actions.js*
- Copy file weather.js to the *facebook-bot* directory

```
> sls deploy function -f facebookBot
```

```
1 const moment = require('moment');
2 const weather = require('./weather');
3
4 const actions = {
5   getWeather: (data) => new Promise((resolve, reject) => {
6     const context = data.context;
7     const entities = data.entities;
8
9     const missingLocation = entities.location === undefined;
10    const location = entities.location ? entities.location[0].value : null;
11    const datetime = entities.datetime ? entities.datetime[0].value : null;
12
13    if (missingLocation) {
14      const contextData = Object.assign({}, context, { missingLocation });
15      resolve(contextData);
16    } else if (datetime) {
17      weather.forecastByLocationName(location, datetime)
18        .then((weatherData) => {
19          const contextData = Object.assign({}, context, weatherData);
20          if (datetime) {
21            Object.assign(contextData, { datetime: moment(datetime).calendar().toLowerCase() });
22          }
23          resolve(contextData);
24        })
25        .catch(reject);
26    } else {
27      weather.weatherByLocationName(location)
28        .then((weatherData) => {
29          const contextData = Object.assign({}, context, weatherData);
30          if (datetime) {
31            Object.assign(contextData, { datetime: moment(datetime).calendar().toLowerCase() });
32          }
33          resolve(contextData);
34        })
35        .catch(reject);
36    }
37  });
}
```

# TRY IT!

- An inquiry with date and location information works!
- But if we drop out the date, we do not get the right response.

Let's fix that!



**Weather in London in 2 days**

The weather in London  
thursday at 1:00 pm is broken  
clouds with temperature of  
13°C

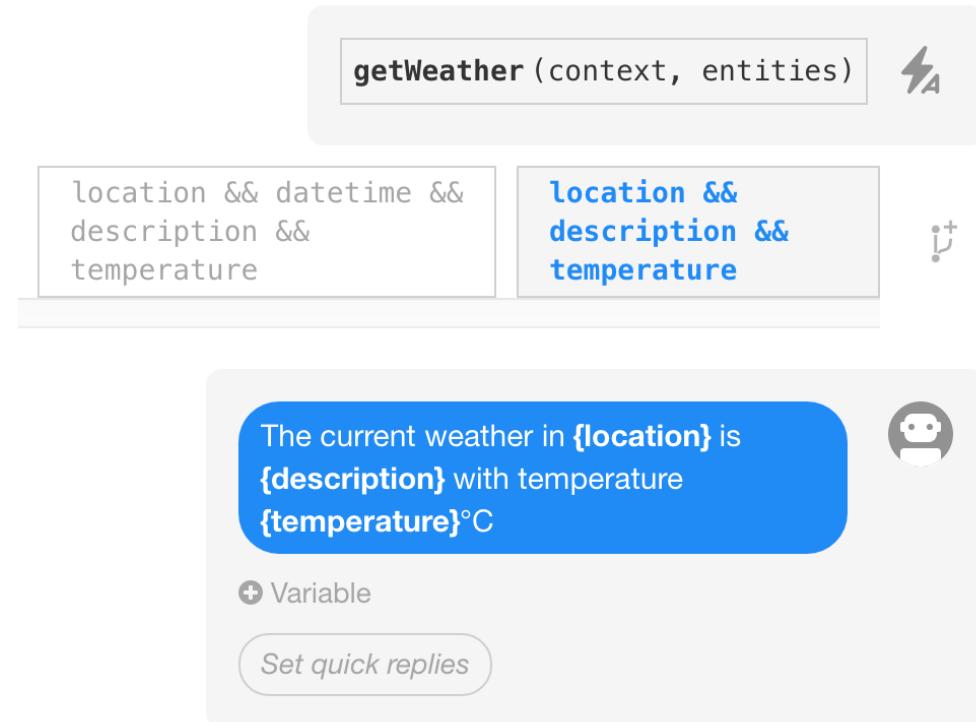


**Weather in London**

Hello! I am the weather bot!

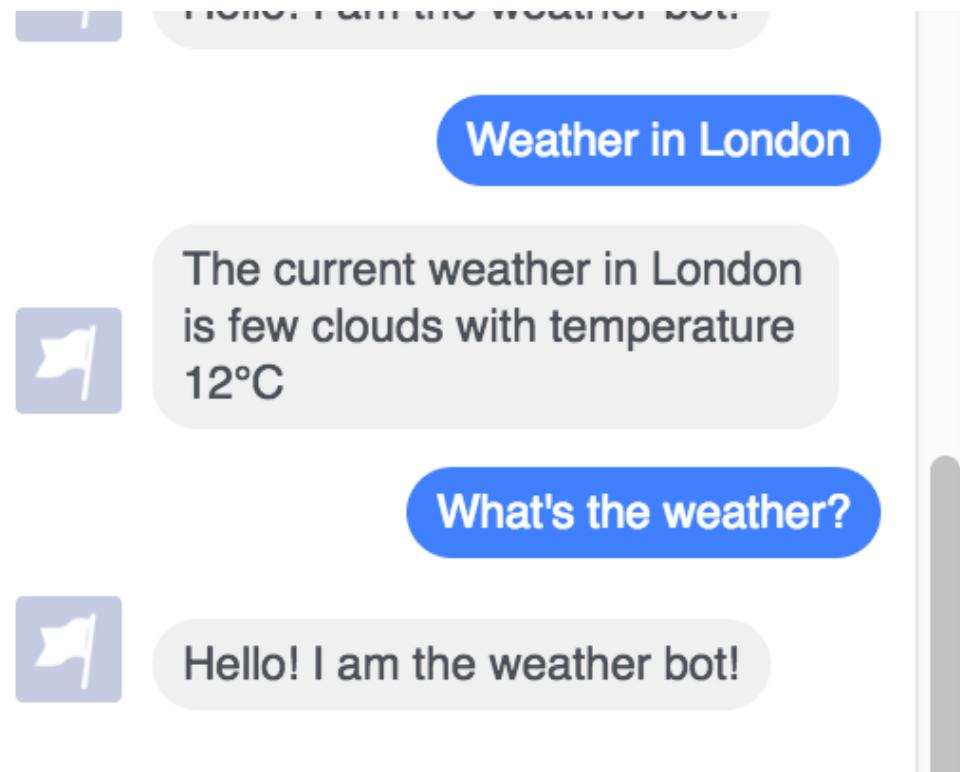
# FORK THE STORY WITHOUT A DATE

- Click the fork icon next to the context keys
- Create new context key *location && description && temperature* (without *datetime*)
- Create new response
- Save



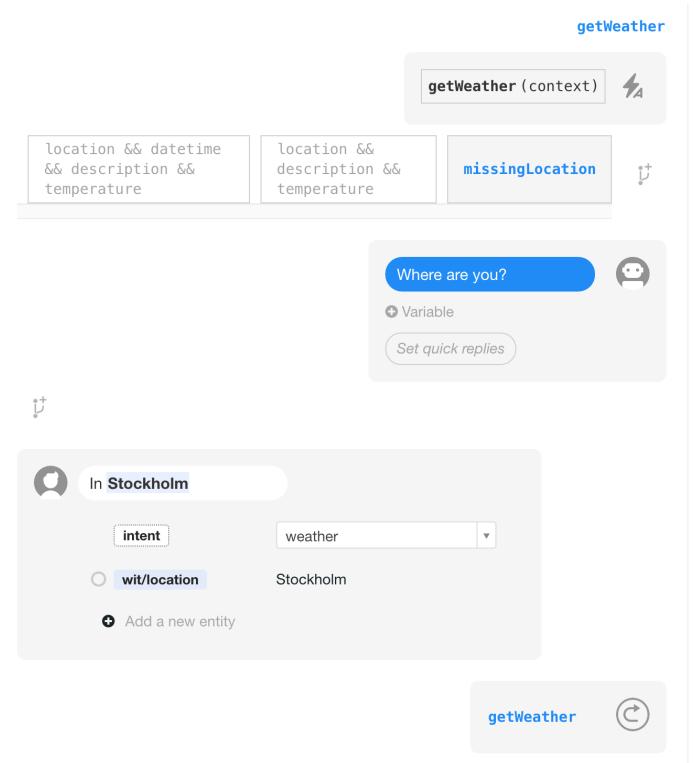
# TRY IT!

- The current weather is now given if no date is provided
- But if we do not provide a location, the bot should probably ask for it!



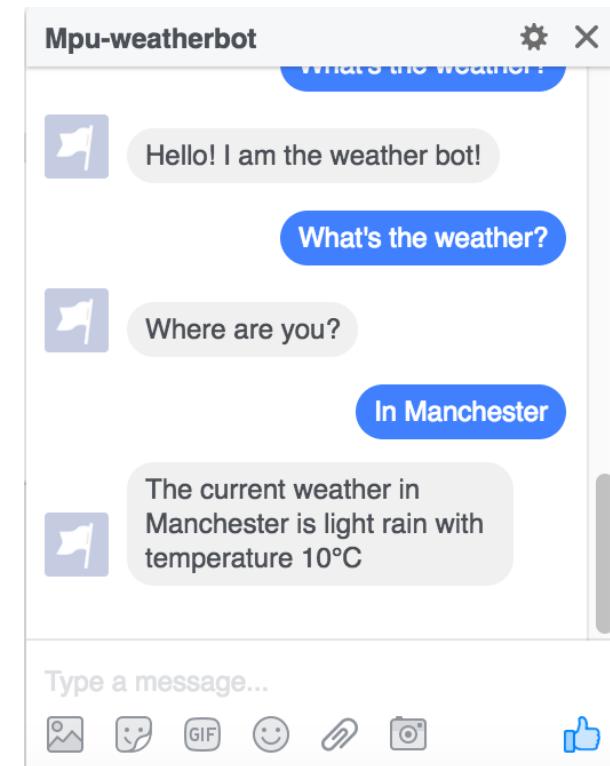
# ASK ADDITIONAL QUESTIONS FROM THE USER

- Click the bookmark icon next to the `getWeather` action and create bookmark `getWeather`
- Create new fork with context key `missingLocation`
- Add a bot response for asking the location
- Add a user response and set `intent + wit/location` entities
- Add Jump to `getWeather` bookmark created earlier



# TRY IT!

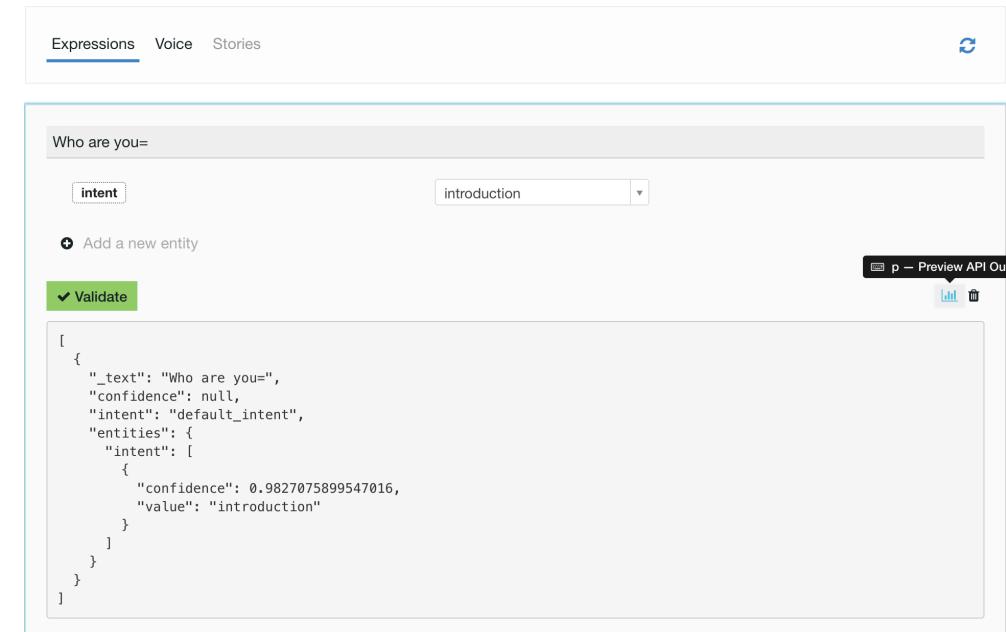
# IT WORKS!



# TRAINING THE BOT

# WIT.AI INBOX

- Use Wit.ai inbox to train model based on incoming messages
- Validate expressions that have been entered by users
- Preview API output for expressions



The screenshot shows the Wit.ai inbox interface. At the top, there are tabs for 'Expressions', 'Voice', and 'Stories'. Below the tabs, a search bar contains the text 'Who are you?'. Underneath the search bar, there is a dropdown menu set to 'introduction' and a button labeled 'intent'. A link 'Add a new entity' is visible. On the right side of the interface, there is a preview area titled 'p - Preview API Output' which contains the following JSON response:

```
[{"text": "Who are you?", "intent": "default_intent", "entities": [{"entity": "introduction", "value": "introduction", "confidence": 0.9827075899547016}]}]
```

# WIT.AI INBOX

- Use Wit.ai understanding to train model
- Enter expressions, set entities and validate to improve accuracy

## Test how your bot understands a sentence

You can train your bot by adding more examples

What is the temperature in **Frankfurt on Sunday?**

<input type="radio"/> wit/datetime	23/10/2016, 00:00:00
<input type="radio"/> intent	weather
<input type="radio"/> wit/location	Frankfurt
<a href="#">Add a new entity</a>	
<a href="#">✓ Validate</a>	

Your app uses 3 entities

Name	Search Strategy ⓘ	Values

Press ⌘ to chat with your bot

# ADDITIONAL CHALLENGES

## ADDITIONAL CHALLENGES (FOR EXAMPLE)

1. Create new story in a different language and hook your bot to it
2. Add SQS to the messageQueue to add robustness (remember to submit a PR to the boilerplate ☺ )
3. Implement flow for "What is the weather tomorrow?", i.e. remember datetime when asking for location
4. Hook voice commands to bot
5. Or anything else / extend bot with something non-weather related

# SC5

THANKS A LOT!

*mikael.puittinen@sc5.io*