AI in action
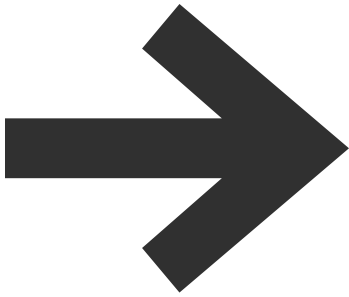
Three technical case studies show how developers are using Azure AI to create the next generation of apps

Microsoft Azure

Developers around the world are using AI to build apps that are more **useful, responsive, and intuitive**. This collection of technical case studies highlights three **real-world examples** of how developers have used Azure AI Services to build applications that lower the barriers between people and technology.

Examples of Azure AI in action

→ Telecommunications provider **Three** built an AI-driven bot with Cognitive Services APIs that provides better self-service for customer tasks such as activating SIM cards, porting their numbers, or upgrading service.

Go to Three Case Study

→ Municipal-services provider **Powel** helps technicians working on hazardous electrical infrastructure stay hands-free—and safer—with a chatbot named André, who helps manage the work checklist while providing useful information.

Go to Powel Case Study

→ Travel-app maker **NAVITIME** developed a chatbot that travelers can use to find local attractions or look up appealing restaurants—all while overcoming local language barriers in anticipation of thousands of visitors attending the Tokyo Olympics in 2020.

Go to NAVITIME Case Study

Each technical case study shows how developers built their apps using Azure AI, and the resulting user experience. The studies also demonstrate how any developer can build their own AI-enhanced applications quickly.

Technical Case Study: Three

Technical case study

Internet provider Three improves customer assistance with self-service chatbot

Lilian Kasem
Jun 13, 2017

Telecommunications and Internet service provider Three has experimented informally with several chatbot services on a variety of platforms but is keen to ensure as much code reusability as possible and reduce maintenance overhead. Its existing webchat has proven very successful but there are still unresolved user queries via this existing interface. To solve this problem, Three worked with Microsoft to build a bot that guides users through several self-service scenarios (that don't require a human agent) such as activating a SIM card and answering general questions with a view to moving this quickly into production.

The final solution was a chatbot that answers general questions at any point during a conversation. It also provides two self-service flows that enable users to activate their SIM cards and to port their numbers using the bot—without ever having to leave the chat interface. Finally, the bot also gave users information on how to cancel or upgrade their contract with Three. Underlying the whole bot, we used Azure Application Insights to track telemetry of all the dialogs that we hope to later use as feedback into the bot for conversational flow/UX improvements.

Customer profile:

Three.co.uk

Three is a telecommunications and Internet service provider operating in the United Kingdom. The company launched in March 2003 as the UK's first commercial video mobile network. It provides 3G and 4G services through its own network infrastructure.

Three wants to make life easier for its customers by helping them get the most out of their mobile devices, offering real value from the services it provides and by removing the barriers that frustrate them.

Key technologies

- Microsoft Bot Framework (Node.js)
- Microsoft QnA Maker
- Language Understanding Intelligent Service (LUIS)
- Web Apps feature of Azure App Service
- Azure Application Insights

Problem statement

For the past 18 months, Three's existing webchat functionality has proven very successful but there are still unresolved customer queries via this existing interface. These include

- Cancel or upgrade a contract.
- New customer to Three.
- Purchase a new phone/contract.
- Report a stolen or lost device/SIM card.
- Add/remove services and contract add-ons.
- Report no Internet service or no/poor mobile coverage.
- Top up service usage.
- Amend user details.
- Account balance queries.

Solutions and steps

Prerequisites

- Install Visual Studio Code
- Install Node.js
- Obtain Cognitive Services keys
- Obtain Azure subscription

Three's success measurements

- Reduced number of webchats related to topics for which there are adequate online self-help resources.
- Increased number of visits to self-service journeys selected to solve customer queries.
- Internal validation within the bot; for example, the number of people who say the bot session was helpful and meant they did not have to call/chat and so on.
- Ascertain effort required by Three resources to set up, maintain, and optimize user cases and responses from the bot.

Solution and architecture

The final solution in the hackfest was a chatbot that:

- Answers frequently asked questions (FAQs) using the Microsoft QnA Maker service.
- Enables users to activate their Three SIM card.
  - Users fill in a form through a conversation with the bot.
- Enables users to port their existing phone number over to Three.
- Enables users to upgrade or cancel their contract with Three.
- Tracks telemetry of all the dialogs using Azure Application Insights, which will later be used for feedback into the bot for conversational flow/UX improvements.

DevBot
Online

Today

Welcome to Three Chatbot.

Hi, I'm Three Chatbot, I'm here to help. I can't answer all your questions, but I can help you activate a SIM, cancel or upgrade a contract, or bring your number to Three.

Activate SIM

Port my number

Upgrade or cancel

via Skype

Type a message here

Activate SIM 11:08

OK which of these best describes you? 11:08

11:08

I ordered a replacement for a missing or bro...

I ordered a different size SIM

I have just upgraded

I am a new customer

I am a new customer 11:08

Good news. Your SIM card has been pre-activated. Just pop it in your phone and away you go. 11:08

We hope you enjoy your time with Three.   Please take a moment to moment to Register on My3 and download the My3 App

Thanks for using ThreeBot. Have you completed what you needed to do?

Yes   No 11:08

Yes 11:09

Great, could you please tell me if I was helpful in you achieving your goal? 11:09

11:09

Yes, you saved me a call

No, it wasn't really a great experience

I want to leave some more detailed feedback

Yes, you saved me a call 11:09

Thanks for taking the time to share your thoughts. Bye! 11:09

Port my number   11:10

OK, I just need to ask you a few questions.   11:10

What is the existing number you want to keep?

07794988675   11:10

What is your temporary new Three number?   11:10

07794988675   11:11

What is your PAC number?   11:11

123465767589097   11:11

What is your email address?   11:11

lilian@email.copm   11:11

What is your full name?   11:11

Lilian   11:11

Upgrade or cancel   11:12

Would you like to cancel or upgrade?   11:12

Cancel   Upgrade   11:12

Upgrade   11:12

What's your contract type?

Handset   11:13

Mobile Broadband

Tablet

SIM only

SIM only   11:13

We are sorry, you cannot currently upgrade online. Please give us a call on 0333 338 1001 or visit a store. Find you nearest store   11:13

Store locator - Supp...
Find your nearest Three store today, just enter...
three.co.uk

Upgrade or cancel   11:14

Would you like to cancel or upgrade?   11:14

Cancel   Upgrade   11:14

Cancel   11:14

Oh, I'm sorry to hear that. OK let me see how I can help you. I can help you to: 11:14

Request a PAC code   11:14

Find out contract end date

Request a PAC code   11:14

If you are a phone customer please call us on 0333 338 1001   11:14

If you are a mobile broadband customer please call us on 0333 338 1003

Thanks for using ThreeBot. Have you completed what you needed to do?
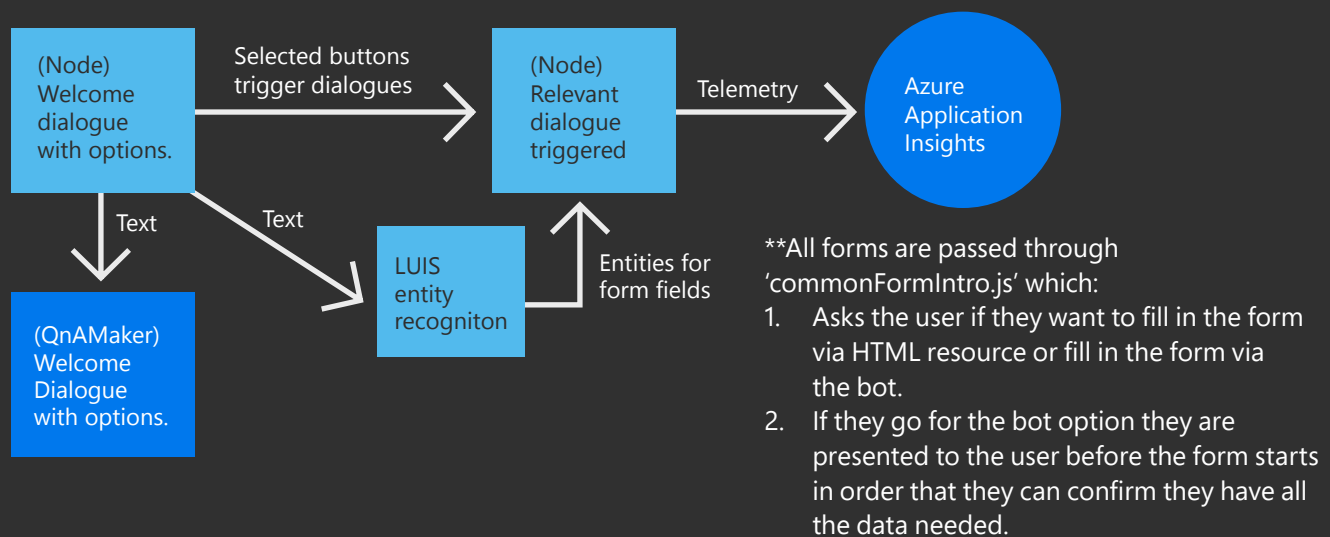
Yes   No   11:14

The bot starts off with a welcome message that is triggered when a user adds a bot to their contacts or opens a conversation with the bot. From here the user can:

- Use free text to ask a question (which gets handled by the QnA Maker).
  - For example: "What is 3G?"
- Use free text to tell the bot to do something (which triggers the relevant dialog).
  - For example: "I want to port my number."
- Just click a button to trigger the relevant dialog.
  - For example: Clicking the "Activate SIM" button.

All the dialogs within the bot incorporate telemetry (using Azure Application Insights) to track where the users are navigating within the bot's conversational flow—what sort of questions they're asking and how long it takes to complete a request.

All the technical implementation for the above can be found in the "Technical delivery" section of this document.

(Node)
Welcome
dialogue
with options.

Selected buttons
trigger dialogues

(Node)
Relevant
dialogue
triggered

Telemetry

Azure
Application
Insights

Text

Text

(QnAMaker)
Welcome
Dialogue
with options.

LUIS
entity
recogniton

Entities for
form fields

**All forms are passed through 'commonFormIntro.js' which:
1. Asks the user if they want to fill in the form via HTML resource or fill in the form via the bot.
2. If they go for the bot option they are presented to the user before the form starts in order that they can confirm they have all the data needed.

** Starting a node form builder activateSIMForm.js which will recognize any entities entered upfront by the user and use them in the form (after being revalidated by the user)

The main flows that Three wanted to build out during the hack for the initial bot solution were:

- Activate SIM
- Port a number
- Cancel or upgrade a contract
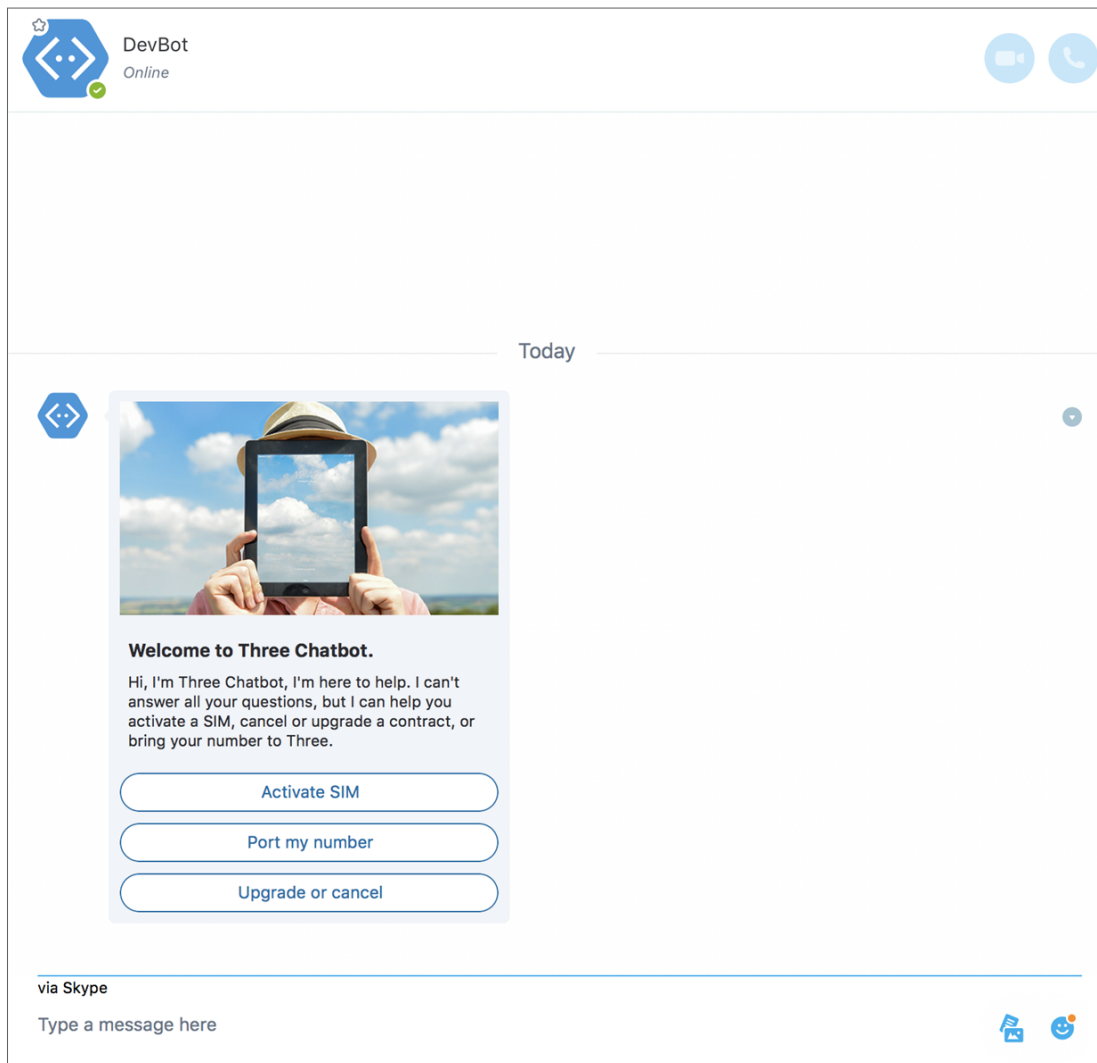
Technical delivery

This section describes the solution implementation details.

Core bot capabilities

Activate SIM

This bot flow is made up of several dialogs:

- ActivateSIM
- ActivateSIMForm
- CommonFormIntro
- ActivateSIMSubmit

ActivateSIM

Users are first asked which profile best suits them—this will help the Three bot fill in the correct form depending on the type of customer it is interacting with. This is done through a choice prompt. Depending on the response from the user, different dialogs are called such as **CommonFormIntro** or **ActivateSIMForm**. A lot of the following dialogs will use a similar prompt, as shown below, to guide the users.

```
builder.Prompts.choice(
    session,
    'OK\n which of these best describes you?',
    [
        'I ordered a replacement for a missing or broken SIM',
        'I ordered a different size SIM',
        'I have just upgraded',
        'I am a new customer'
    ],
    {listStyle: builder.ListStyle.button}
)
```

Activate SIM   11:08

OK which of these best describes you?   11:08

                                                11:08

I ordered a replacement for a missing or bro...

I ordered a different size SIM

I have just upgraded

I am a new customer

ActivateSIMForm

This dialog asks users a series of questions that are required to complete the form needed to activate their SIM. This is a useful method to easily create a form within the bot framework.

```
function (session, args) {
    // Save entity data to dialogData
    if (args.entityData) {
    session.dialogData.entityData = args.entityData
    }
    session.dialogData.index = args.index ? args.index : 0
    session.dialogData.form = args.form ? args.form : {}

    // Check if entityData exists
    if (session.dialogData.entityData) {
    // If the entityData exists and it possesses the property for this
question, send a confirm prompt
    if
(session.dialogData.entityData.hasOwnProperty(questions[session.dialog
Data.index].field)) {
        var prompt = questions[session.dialogData.index].prompt
        prompt = prompt.replace('{' +
questions[session.dialogData.index].field + '}',

session.dialogData.entityData[questions[session.dialogData.index].fiel
d])
        builder.Prompts.confirm(session, prompt)
    } else {
        // If the entityData exists but the property for this question
doesn't, send a text prompt
        builder.Prompts.text(session,
questions[session.dialogData.index].question)
    }
    } else {
    // If there is no entityData, proceed as normal
    builder.Prompts.text(session,
questions[session.dialogData.index].question)
    }
},
function (session, results, next) {
    // Check if the user responding via a Confirm or Text prompt
    if (results.response === true) {
    // If the confirm prompt is true then we save the entity to the
form object and increment the index
    var field = questions[session.dialogData.index++].field
    session.dialogData.form[field] =
session.dialogData.entityData[field]
    } else if (results.response === false) {
    // If the confirm prompt is false then we delete the entity from
the entityData object but we do NOT increment the index
    field = questions[session.dialogData.index].field
    delete session.dialogData.entityData[field]
```

CommonFormIntro

This dialog introduces the user to the form and describes the information required.

```
var requirements = ['You will need the following information: \n'];
for (var requirement in session.dialogData.formRequirements) {
    requirements.push('\n * ' +
session.dialogData.formRequirements[requirement])
}
requirements = requirements.join('');
session.send(requirements);
```

I want to activate my sim   11:17

OK which of these best describes you?   11:17

I ordered a replacement for a missing or bro…   11:17

I ordered a different size SIM

I have just upgraded

I am a new customer

I ordered a different size SIM   13:38

You can complete this action via a form. Would you like me to help you fill this out?   13:38

yes   no   13:38

yes   13:38

You will need the following information: 13:38
* Your phone number
* The ICCID number found on your SIM card

Do you have this to hand?

yes   no   13:38

yes   13:41

What is your existing phone or mobile broadband number?   13:41

07794988675   13:42

What is your new 19-digit SIM number?   13:42

756489865682908746   13:42

Do you agree with Three's Terms and Conditions?   13:42

yes   13:42

Thanks, your request has been submitted. Your SIM should be activated within the next 24 hours.   13:42

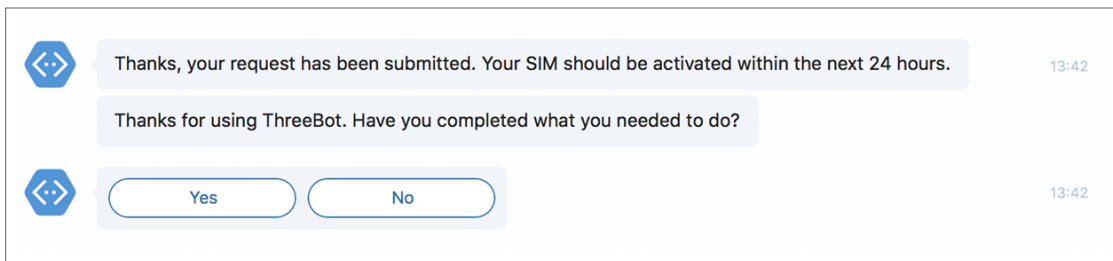Thanks for using ThreeBot. Have you completed what you needed to do?

Yes   No   13:42

ActivateSIMSubmit

This dialog is called once the **ActivateSIMForm** is completed, and submits the form to Three using request to complete a REST POST call to the Three API. The image below displays what the user sees after successfully submitting a form.

Port a number

This dialog is very similar to **ActivateSIMForm** and prompts the user with questions required to submit a form that will port the customer's number. The submission is also made using a POST call to Three.

```
var questions = [
    {field: 'mobileNumber', question: 'What is the
existing number you want to keep?'},
    {field: 'mobileNumber2', question: 'What is your
temporary new Three number?'},
    {field: 'pac', question: 'What is your PAC
number?'},
    {field: 'emailPayMonthly', question: 'What is your
email address?'},
    {field: 'fullName', question: 'What is your full
name?'},
    {field: 'dob', question: 'What is your birthday
(e.g. 01/01/1901)?'},
    {field: 'address1', question: 'What is the first
line of your address?'},
    {field: 'postcode', question: 'What is your
postcode?'}
];
```

```
function (session, args) {
    session.dialogData.index = args ? args.index : 0;
    session.dialogData.form = args ? args.form : {};

    builder.Prompts.text(session,
questions[session.dialogData.index].question);
},
function (session, results) {
    // Save users reply
    var field = questions[session.dialogData.index+
+].field
    session.dialogData.form[field] = results.response
```

	Port my number 11:10
OK, I just need to ask you a few questions. 11:10	
What is the existing number you want to keep?	
	07794988675 11:10
What is your temporary new Three number? 11:10	
	07794988675 11:11
What is your PAC number? 11:11	
	123465767589097 11:11
What is your email address? 11:11	
	lilian@email.copm 11:11
What is your full name? 11:11	
	Lilian 11:11

Cancel or upgrade

This flow contains three dialogs:

- UpgradeOrCancel
- Cancel
- Upgrade

UpgradeOrCancel

This dialog simply asks users whether they want to upgrade or cancel and then calls the relevant dialog depending on the user's choice. This is implemented using a choice prompt.
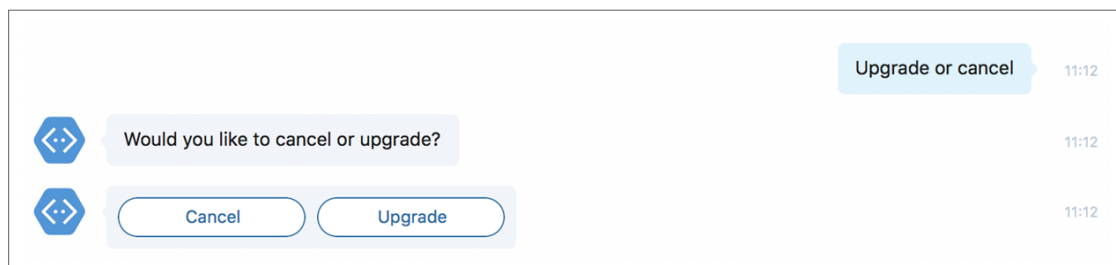
Cancel

This asks the users, using a choice prompt, whether they want a PAC code or to know the end date of their contract. Depending on the response, it replies with the phone number to call or the website they need to navigate to.

Request a PAC code

Find out contract end date

11:14

Find out contract end date   11:14

Ok the best place to do this is in My3. Head over to My3 and you should see your contract end date   11:14
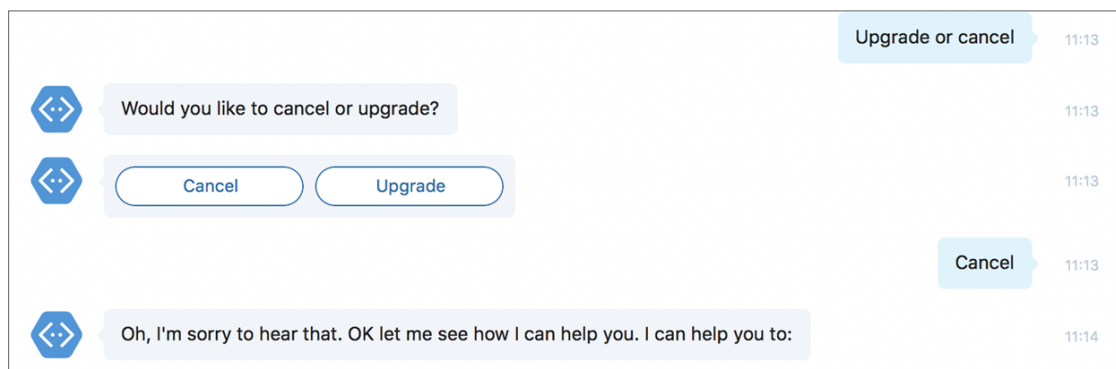
Another great way to do this is via the My3 app. If you don't already have this installed please have a look at how it can make your life easier!

Thanks for using ThreeBot. Have you completed what you needed to do?

Yes   No   11:14

Upgrade or cancel   11:14

Would you like to cancel or upgrade?   11:14

Cancel   Upgrade   11:14

Cancel   11:14

Oh, I'm sorry to hear that. OK let me see how I can help you. I can help you to: 11:14

Request a PAC code   11:14

Find out contract end date

Request a PAC code   11:14

If you are a phone customer please call us on 0333 338 1001   11:14

If you are a mobile broadband customer please call us on 0333 338 1003

Thanks for using ThreeBot. Have you completed what you needed to do?

Yes   No   11:14

Upgrade

This dialog does the same as the 'cancel' dialog. It asks for the type of contract and then, depending on the answer, it gives the user the website they need in order to upgrade.

Bot Intelligence

LUIS

In this project, we primarily relied on using regex to determine the user's intent. However, we also wanted to use the Microsoft Language Understanding Intelligent Service (LUIS) as a fallback in case the user decided to enter free text queries. We created a LUIS model to handle the intents for the three flows that we had built within this bot (activate SIM, port a number, cancel or upgrade).

Example regex used:

```
matches: /^Activate SIM/i
matches: /^Port my number/i
matches: /^Upgrade or cancel/i
```

LUIS model

First, we configured the LUIS recognizer inside **config.js**:

```
// Import LUIS Model
var recognizer = new
builder.LuisRecognizer(process.env.LUIS_MODEL_URL);
bot.recognizer(recognizer);
```

We then set up trigger actions for each main flow dialog so that if the LUIS model recognized the intent, it would trigger the required dialog.

For example, if the user says: "I want to activate my SIM card," LUIS would pick this up as an **ActivateSIM** intent, which would then trigger the start of the ActivateSIM dialog because the intent matches the triggerAction keyword (see below). We also set the **intentThreshold** for the LUIS intent trigger so that only matches above a 0.5 confidence rating would trigger the dialog.

```
.triggerAction({
    matches: 'ActivateSIM',
    intentThreshold: 0.5
  })
```

You can easily learn how to make your own LUIS model at the LUIS website.

QnA Maker

Three used the Microsoft Cognitive Services QnA Maker service to answer simple customer questions that are available online (Three FAQ Page). This was quick to implement and helped triage simple customer queries away from the direct human assistance.

This is the QnA dialog that handles all FAQs. We call **endDialog** so that the bot returns to the previous dialog the user was in when they asked the question.

```
module.exports = function () {
  bot.dialog('QnA', (session, results) => {
    console.log(process.env)
    var client = restify.createJsonClient('https://
westus.api.cognitive.microsoft.com')
    var options = {
      path: '/qnamaker/v2.0/knowledgebases/' + process.env.QNA_KB_ID +
'/generateAnswer',
      headers: {
        'Ocp-Apim-Subscription-Key': process.env.QNA_SUBSCRIPTION_KEY
      }
    }

    var question = {'question': results.question}

    client.post(options, question, (err, req, res, obj) => {
      if (err == null && obj.answers.length > 0) {
        for (var i in obj.answers) {
          if (parseInt(obj.answers[i].score) > 0.80) {
            session.endDialog(obj.answers[i].answer)
          } else {
            session.endDialog('Sorry, I couldn\'t find an answer in
                              our FAQs. Don\'t forget, you can type
                              \'help\'
                              if you need assistance')
          }
        }
      } else {
        session.endDialog('Sorry, there was an error!')
      }
    })
  })
}
```

This dialog is then called through the UniversalBot. This means that the QnA maker is surfaced globally throughout the bot. If the user query is not recognized by either LUIS or by one of the regex expressions, then it will be sent to the QnA dialog.

```
global.bot = new builder.UniversalBot(connector, function (session) {
    session.send('I\'ll just check that for you...',
session.message.text)
    session.replaceDialog('QnA', { question: session.message.text })
});
```

The QnA Maker functionality is also triggered when the user says **help**, **quit**, **problem**, or **support** and doesn't want the main menu:

```
if (results.response && results.response.entity === 'no') {
    builder.Prompts.text(session, 'Ok, why don\'t you try asking your
query here and I\'ll search our FAQs');
}
.
.
.
function (session, results, next) {
    session.replaceDialog('QnA', { question: session.message.text });
}
```

What is 3G?   11:16

I'll just check that for you...   11:16
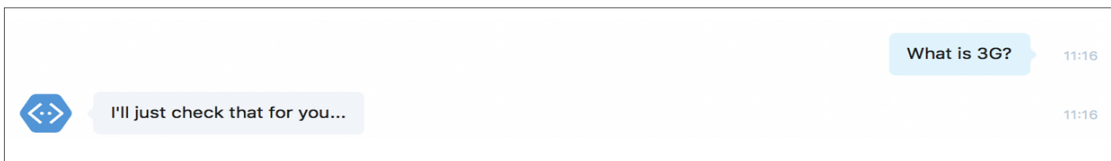
Azure Application Insights

We set up Application Insights to capture telemetry within the bot. This is used to trace the conversational flows that the user has gone through within the bot and to also track metrics on how long it has taken to complete a particular task.

First, we set up the Application Insights client in the **config** file:

```
global.telemetryModule = require('./telemetry-module')
const appInsights = require('applicationinsights')
appInsights.setup(process.env.APPINSIGHTS_INSTRUMENTATION_KEY).start()
global.appInsightsClient = appInsights.getClient()
```

Then we created a telemetry module to handle the telemetry work:

```
exports.createTelemetry = function (session, properties) {
  var data = {
    conversationData: JSON.stringify(session.conversationData),
    privateConversationData: JSON.stringify(session.privateConversationData),
    userData: JSON.stringify(session.userData),
    conversationId: session.message.address.conversation.id,
    userId: session.message.address.user.id
  }
  if (properties) {
    for (var property in properties) {
      data[property] = properties[property]
    }
  }
  return data
}
```

Finally, we created different telemetry objects in each dialog to trace that the user visited the dialog. For example:

```
// Store entity data in dialogData
session.dialogData.entities = data;
// Create a new telemetry module with session data
session.dialogData.telemetry =
telemetryModule.createTelemetry(session, { setDefault: false });
// Track that the user has been to the activate sim dialog
appInsightsClient.trackTrace('activateSIM',
session.dialogData.telemetry);
```

We can then retrieve the trace results within the Analytics portal in Azure Application Insights:

Completed. Showing results from the last 24 hours.

TABLE **CHART**

Drag a column header and drop it here to group by that column

timestamp	message	severityLevel	application_Version	client_Type	client_OS	client_IP	client_CountryOrRegion
2017-04-06T12:38:32.557Z	activateSIM12	1	1.0.0	PC	16.4.0	194.69.102.0	United Kingdom
2017-04-06T10:10:05.883Z	welcome	1	1.0.0	PC	16.4.0	194.69.102.0	United Kingdom
2017-04-06T10:12:41.416Z	welcome	1	1.0.0	PC	16.4.0	194.69.102.0	United Kingdom
2017-04-06T10:13:49.982Z	welcome	1	1.0.0	PC	16.4.0	194.69.102.0	United Kingdom
2017-04-06T10:14:28.153Z	welcome	1	1.0.0	PC	16.4.0	194.69.102.0	United Kingdom
2017-04-06T10:16:18.847Z	welcome	1	1.0.0	PC	16.4.0	194.69.102.0	United Kingdom
2017-04-06T10:17:13.691Z	welcome	1	1.0.0	PC	16.4.0	194.69.102.0	United Kingdom
2017-04-06T10:17:24.405Z	activateSIM	1	1.0.0	PC	16.4.0	194.69.102.0	United Kingdom
2017-04-06T10:07:41.308Z	welcome	1	1.0.0	PC	16.4.0	194.69.102.0	United Kingdom
2017-04-06T10:08:13.983Z	welcome	1	1.0.0	PC	16.4.0	194.69.102.0	United Kingdom
2017-04-06T10:08:34.412Z	activateSIM	1	1.0.0	PC	16.4.0	194.69.102.0	United Kingdom
2017-04-06T10:08:54.333Z	activateSIM3	1	1.0.0	PC	16.4.0	194.69.102.0	United Kingdom
2017-04-06T12:40:39.242Z	welcome	1	1.0.0	PC	16.4.0	194.69.102.0	United Kingdom
2017-04-06T12:40:44.423Z	activateSIM	1	1.0.0	PC	16.4.0	194.69.102.0	United Kingdom
2017-04-06T12:40:50.934Z	activateSIM12	1	1.0.0	PC	16.4.0	194.69.102.0	United Kingdom

To trace how long it took for a form to be submitted, the following code was used:

```
// Setup a telemetry module
session.dialogData.measuredEventTelemetry =
telemetryModule.createTelemetry(session);
// Start timer. We want to track how long it takes for us to submit a
SIM activation request
session.dialogData.timerStart = process.hrtime();
.
.
.
// Submittion has been made, calculate how long it took.
var timerEnd = process.hrtime(session.dialogData.timerStart);
// Save the time it took to 'metrics' within the
measuredEventTelemetry module
session.dialogData.measuredEventTelemetry.metrics = (timerEnd[0],
timerEnd[1] / 1000000);
// Track the above metric as 'timeTaken'
appInsightsClient.trackEvent('timeTaken',
session.dialogData.measuredEventTelemetry);
```

∧ 2017-03-29T10:23:04.038Z   timeTaken   {"userId":"default-user","userData":"{}","conversati…   1.0.0   PC

      timestamp   2017-03-29T10:23:04.038Z

      name   timeTaken

   ∧ customDimensions   {"userId":"default-user","userData":"{}","conversationId":"e7i55i930e6a6f7l","metrics":"614.

         userId   default-user

         userData   {}

         conversationId   e7i55i930e6a6f7l

         metrics   614.445885

         conversationData {}

     ˅ privateConversationData {"simForm":{"MSISDN":"07794988675","ICCID":"12345678901§23456789"}}

Conclusion

We began this project with the aim of building a bot that can guide users through several self-service scenarios such as activating a SIM card and answering general FAQs with a view to moving this quickly into production.

As a result, we have developed a compelling bot that should save Three significant time and money. It will make the job easier for customer service agents because they will no longer have to deal with commonly asked simpler/general questions, therefore freeing up their time to deal with more specific, trickier queries. The bot should also provide customers with a simpler, faster method to get their queries answered through a guided conversation experience. Three thinks this will change the behavior of customers and agents alike, making them more productive.

Challenges

- The documentation for the Node.js SDK can be tricky to follow in its current form and feels limited in comparison to the C# SDK documentation. This sometimes made it difficult to find the information needed to put the bot together. However, a lot of useful code samples are available.

- Error feedback from LUIS and QnA services is limited, or non-existent in the case of QnA Maker. This made it difficult to debug and figure out why things didn't work as expected.

- Clear guidance on development best practices for the bot framework is still evolving. When developing a bot that is intended for production, it would be useful to see the best practices for doing certain things such as:

  ◦ Determining project structure.
  ◦ Asking the user for feedback at the end of a dialog.
  ◦ Using LUIS versus regex.

Learnings

Welcome messages don't work with Slack

Unlike other channels, **onConversationUpdate** is not triggered when a bot is added to Slack. Also, **contactRelationUpdate** is not triggered when a user is added to Slack. This means there is currently no way to welcome a Slack user and introduce them to the bot, without them interacting with the bot first. We did discover that the native Slack channel does deal with this. If using a web socket, the Slack framework sends a bot_added event to let you know when a user has added a bot to their Slack channel. However, the Microsoft Bot Framework is not using web sockets with Slack and doesn't get notification of this event.

The pre-built QnA Maker package is 'semi-permanent'

If you start the QnA dialog, there is no obvious way out of this dialog flow. The user would be stuck using the QnA service and won't be able to continue using the bot's other dialogs, unless you use the global **restart** command. The solution is to call the QnA API directly and end the dialog after the question is answered. This way, users can ask a question in the middle of any other dialog, get their question answer by the QnA service, and then continue with the dialog they were in last. The implementation for this can be found in the "Technical delivery" section of this document (under QnA Maker) or in this sample GitHub repo.

How to implement safe words

Using regex and actions, we are able to set up global commands that users can use at any point in the bot. This allows users to return out of a dialog when they become stuck in a conversational flow.

```
bot.endConversationAction('goodbye', 'Goodbye :)', { matches: /^bye/
i });
bot.beginDialogAction('home', '/start', { matches: /^home/i });
bot.beginDialogAction('help', '/help', { matches: /^help/i });
```

Using LUIS intents through `triggerActions` **and setting the** `intentThreshold`

Instead of using the usual method of matching LUIS intents:

```
intents = new builder.IntentDialog({recognizers: [recognizer]});

intents
.matches('LUIS_Intent', '/dialog')
.matches('LUIS_Intent', '/dialog')
.onDefault(builder.DialogAction.send("I'm sorry. I didn't
understand."));
```

We found it was a lot cleaner to match LUIS intents to dialogs through **triggerActions** (which you simply add at the end of a dialog). This also led us to discover you can set a threshold for which LUIS intents should meet before they get triggered using **intentThreshold**. The default is to trigger actions even if the confidence of the match is less than **0.1**.

```
.triggerAction({
    matches: 'ActivateSIM',
    intentThreshold: 0.5
  })
```

You also can add an **intentThreshold** at a global level, so that it applies to all LUIS matches:

```
intents = new builder.IntentDialog({ recognizers: [recognizer],
intentThreshold: 0.5 });
```

An easy and clean way to fill out forms

We developed a clean way of asking users questions to fill out a form using dialog recursion.

First, create an array of questions:

```
var questions = [
    {field: 'mobileNumber', question: 'What is the existing number you
want to keep?'},
    {field: 'mobileNumber2', question: 'What is your temporary new
Three number?'},
    {field: 'pac', question: 'What is your PAC number?'},
    {field: 'emailPayMonthly', question: 'What is your email
address?'},
    {field: 'fullName', question: 'What is your full name?'},
    {field: 'dob', question: 'What is your birthday (e.g.
01/01/1901)?'},
    {field: 'address1', question: 'What is the first line of your
address?'},
    {field: 'postcode', question: 'What is your postcode?'}
]
```

Then, loop through each question by calling the same dialog and passing in an index counter, saving the user's response after each question is asked:

```
bot.dialog('FillOutForm', [
function (session, args) {
    session.dialogData.index = args ? args.index : 0
    session.dialogData.form = args ? args.form : {}

    builder.Prompts.text(session,
questions[session.dialogData.index].question)
},
function (session, results) {
    // Save users reply
    var field = questions[session.dialogData.index++].field
    session.dialogData.form[field] = results.response

    // Check for end of form
    if (session.dialogData.index >= questions.length) {
    session.privateConversationData.portForm = session.dialogData.form
    session.beginDialog('EndofForm')
    } else {
    session.replaceDialog('FillOutForm', session.dialogData)
    }
}
```

You can find a follow code sample of this here: Simple Form Sample for Node SDK

Plans for production

The plan for this bot is to roll it out into A/B testing in the coming weeks so that it can be tested with real customers. Following the one-week A/B trial, the plan is to release this bot into production within months.

Looking further forward, Three recognizes it can provide a richer experience by delivering on the following:

- Rather than linking out to resources on the Three website, the user can ask the bot questions based on the topic of the page and receive answers back based on page content.

- Integration with APIs—there are network-based APIs around coverage and outage problems and reporting an issue, which could be worked into a bot experience.

- Posting to forms—for example, requesting a SIM or porting a number to Three can be facilitated through the bot.

- Handing off to a call center/live agents.

- Personality and personalization—addressing the user by name and personalizing content to suit the user.

"**Thanks all for inviting us to this awesome week. It's been really fun and we have more than the bones of a real product to take to market. We're all gutted to be back to real work tomorrow!**"

Justin Beasley

Lead Digital Development Manager

"**Thanks @microsoft-simon and all other MS folk. We had a great time and learned a lot!**"

Stuart Brown

Digital Development Manager

Additional resources

→

Documentation:
Microsoft Bot Framework
QnA Maker
LUIS

Code:
QnA Single Prompt Sample for the Node.js SDK
Simple Form Filling Sample for the Node.js SDK

Team

→

Microsoft:
Lilian Kasem – Technical Evangelist
Simon Michael – Senior Technical Evangelist
Bill Barnes – Principal Software Development Engineer

Three:
Justin Beasley – Lead Digital Development Manager
Nick Bishop – Digital Development Manager
Stuart Brown – Digital Development Manager
Thomas Barton – Scrum Master
Dimos Fountoukos – Software Developer

Technical Case Study: Powel

Technical case study

Using CaaP and André, the voice-driven assistant bot, to enhance on-site inspection

Pedro Dias and Anders Gill
Jun 19, 2017

During March 2017 at Microsoft offices in Lysaker, Norway, Microsoft teamed up with Powel to create a brand-new bot named André. The aim of the hackfest was to create a voice-driven chatbot that could assist field engineers within the space of electrical grids to answer questions and provide reporting capabilities during inspections to free up their hands. The solution was built using tools from Microsoft leveraging chatbots and conversations as a platform (CaaP) technologies.

In this hackfest, we used the following key technologies:

- Microsoft Bot Framework
- Microsoft Cognitive Services
- Language Understanding Intelligent Services (LUIS)
- Xamarin.Forms
- Bing Speech API
- GitHub
- Visual Studio Team Services
- Visual Studio 2017

Additionally, the team expressed a need for getting more hands-on experience with

- Test automation
- Continuous integration/continuous delivery (CI/CD) scenarios

The hackfest occurred after multiple calls in which the problem area was demystified and ambitions for the hackfest were scoped.

Core Team

Powel

- Damian Plaza, Software Engineer

- Jakub Hiszczyn, Software Engineer

- Karol Stosik, Software Engineer

- Maksymilian Jastrzebski, Software Engineer

- Tor Hovland, Software Engineer

- Simen Karlsen, Smart Grid-Enabler

- Øystein Askeland, Interaction Designer

Microsoft

- Pedro Dias, Sr. Technical Evangelist

- Anders Gill, Technical Evangelist

Solution overview

Powel envisions a future in which on-site engineers can use natural voice as the main driver for computer-assisted support during inspections of power facilities. During an inspection of an electrical station or power line, using André, a Xamarin-based phone application, will free the engineers hands, providing a more fluent process than having to fill out a paper form or record data in a device application. Using the capabilities of a wearable device, such as a mobile phone, the application will automatically understand which facility is being inspected by obtaining the user's GPS location and provide the necessary checkpoint lists and dialogs appropriate to that facility.

The hackfest focused on two distinct inspection targets:

- Power stations
- Power lines

The goal was to be able to design two distinctly different conversation dialogs with enough functionality to be tested in the field. Based on these dialogs, Powel would then have amassed enough experience with CaaP to enrich André and devise similar conversational applications for use in other areas.

We used Xamarin.Forms, Microsoft Bot Framework, LUIS, and Cognitive Services to develop a bot that would allow the engineer to talk to the Xamarin.Forms Android app. The app uses the Google Speech recognition engine to transform audio into text; the text would then be sent to LUIS to determine user intent at any point during the conversation. Based on the detected intent and entities, the Bot Framework would send the appropriate response back to the client app, which would read the response to the engineer, helping him stay safe by keeping him hands-free.

One feature in particular that really extends the usefulness of this bot was the integration of the Bot Framework into the Powel back-end system. We successfully integrated the bot to the systems that would, for instance, allow engineers to extract valuable information, such as

- history of inspections at specific facilities
- specific parts prone to constant maintenance
- extended information about facilities
- the quickest route to the next critical prioritized facility that needs remediation
- a checklist of possible rectification methods

"Microsoft Bot Framework and Cognitive Services will help us in Powel to empower our customers by making their working day simpler, safer, and more efficient."

Simen Karlsen

Smart Grid-Enabler, Powel

Customer profile:

Powel spans Europe with a broad and sustainable customer base and a long history as a trusted supplier of software solutions for cities, municipalities, counties, the energy industry, and the contracting sector. Powel generates around US$50M revenue each year and was founded in Norway in 1996. Powel has grown to be an international corporation with staff numbering 460 with offices in these six countries in addition to Norway: Sweden, Denmark, Switzerland, Chile, Turkey, and Poland.

Powel business segments:

- Smart Energy
- Asset Performance
- Metering
- Water & Community
- Construction

Problem statement

The number-one aspect that every company enforces is worker safety. Powel is no exception. Some jobs, though, are more dangerous than others. Powel has some of the largest customers in Norway that maintain and develop the electrical grids. These inspections must be done once per year for each facility. Other more in-depth technical inspections have to be done once every 10 years for all the components in the electrical grid. These inspection periods are enforced by law.

To perform an inspection, an engineer might have to climb tall masts that can create dangerous situations depending on weather and other factors. These inspections take place especially in wintertime when frost on the ground hinders the development of new grids. This forces the companies to focus on maintenance, which is also an important job. For this reason, it would be very inconvenient to keep a cell phone or a notebook at hand when you could potentially be hanging many meters above the ground.

As a result, an engineer has to remember what to check for, what potential issues might exist, whether other maintenance has been performed on this particular mast or station, what types of remedies counter any issues discovered, and so on. Based on this data, the engineer has to travel back to the main office to hand the information to his colleagues or to digitize it himself. During this transit, valuable information and specific context gathered during the inspection could be lost.

Powel requires a solution that allows the engineers to stay hands-free while still adding value to the conversation. A chatbot triggered by a simple "Hello, André" could potentially relieve the engineer of having to remember all the tasks that he is to execute during his inspection. The great thing about such a chatbot is that it could be connected to all other systems that are relevant and could provide information that the engineer needs. This was one of our goals for the hackfest: being able to develop a simple chatbot that can converse dialogues to external systems and return valuable insight that can increase efficiency during the inspection.

Powel requires a solution that allows the engineers to stay hands-free while still

adding value to the conversation. A chatbot triggered by a simple "Hello, André" could potentially relieve the engineer of having to remember all the tasks that he is to execute during his inspection. The great thing about such a chatbot is that it could be connected to all other systems that are relevant and could provide information that the engineer needs. This was one of our goals for the hackfest: being able to develop a simple chatbot that can converse dialogues to external systems and return valuable insight that can increase efficiency during the inspection.

An inspection could consist of anything from assessing facilities and returning data regarding the types of issues discovered (presently based on a checklist) to actually fixing the issues. Common issues range from woodpeckers pecking the masts or graffiti to more-serious problems like high-voltage electrical components failing.

In summary, the chatbot will

- avoid security issues by staying hands-free
- provide specific insight related to particular facilities

- manage handover by taking notes of current inspection so that no data is lost during the inspection
- support the engineer with other relevant information that might be significant to know about or that he might have forgotten about during the inspection

Solution and steps

The hackfest was initiated by Pedro, who introduced the team to the Bot Framework, showing examples of a working bot, explaining features, and ensuring that the whole team was up to speed before we started developing.

The goal of the three-day hackfest was to create a client app in Xamarin.Forms for Android that could visualize the conversation that the engineer would be having using the Bot Framework using LUIS and the cognitive services. We created a user story in Visual Studio Team Services that consisted of multiple tasks.

The user story was as follows:

As a field worker, I want to be able to submit inspections that are OK to a bot so that I don't have to fill out a form for that.

We defined features and tasks to make it easier to distribute work across the team. We created two repositories: one for the ASP.NET bot application, and one for the Xamarin.Forms application. We also configured CI on the Bot Framework, so that it builds and deploys to Microsoft Azure after each commit.
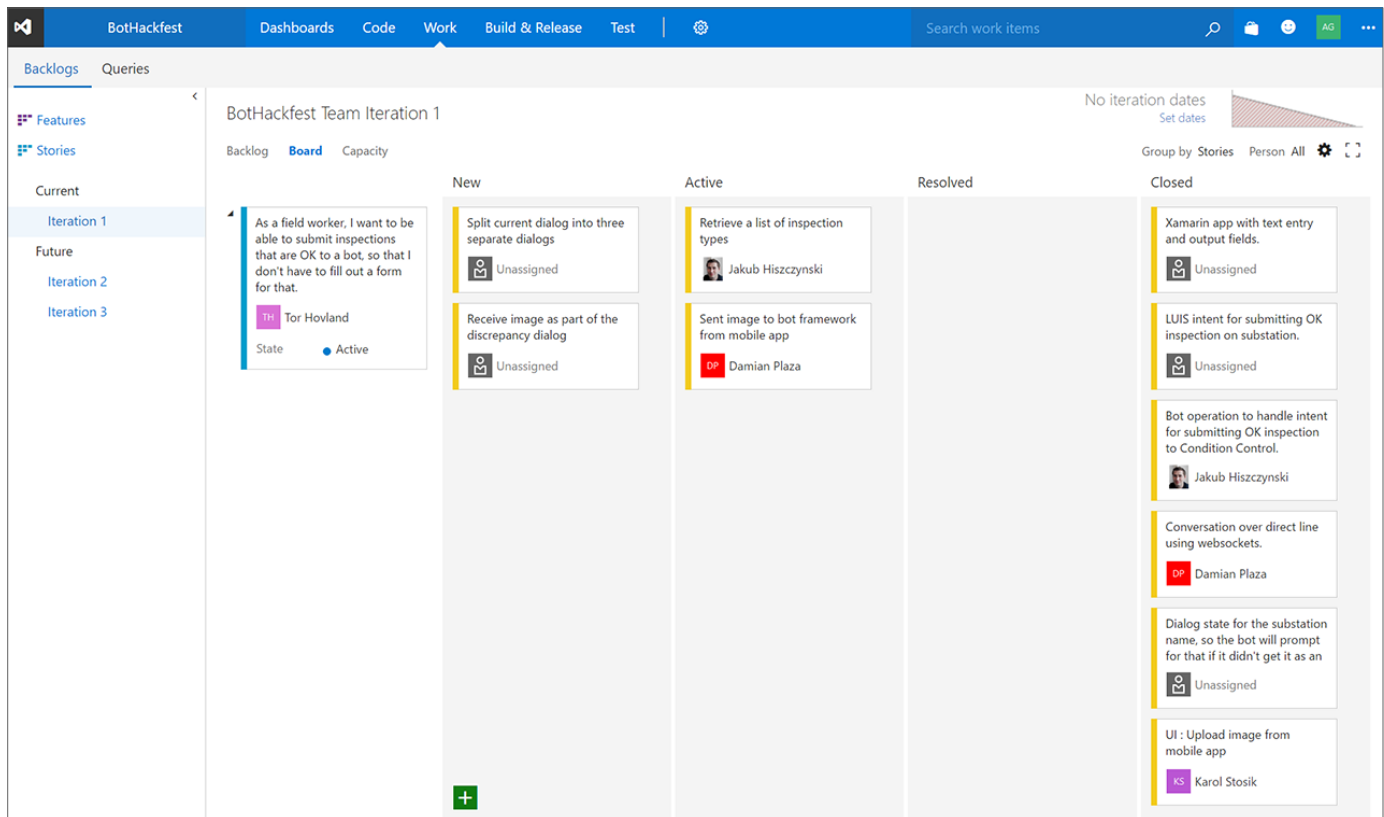
Kanban board

We decided to use Android for the visualization part because the Powel developers all had Android devices. Besides developing a working app, the aim was also to educate the Powel developers on Bot Framework and CaaP, so that they could bring their knowledge back to the drawing board when developing and extending their apps for scenarios that could leverage the Bot Framework and Cognitive Services.

The proof-of-concept (PoC) bot developed during the hackfest would be used as a foundation that would help Powel move their own bot into production after further development.
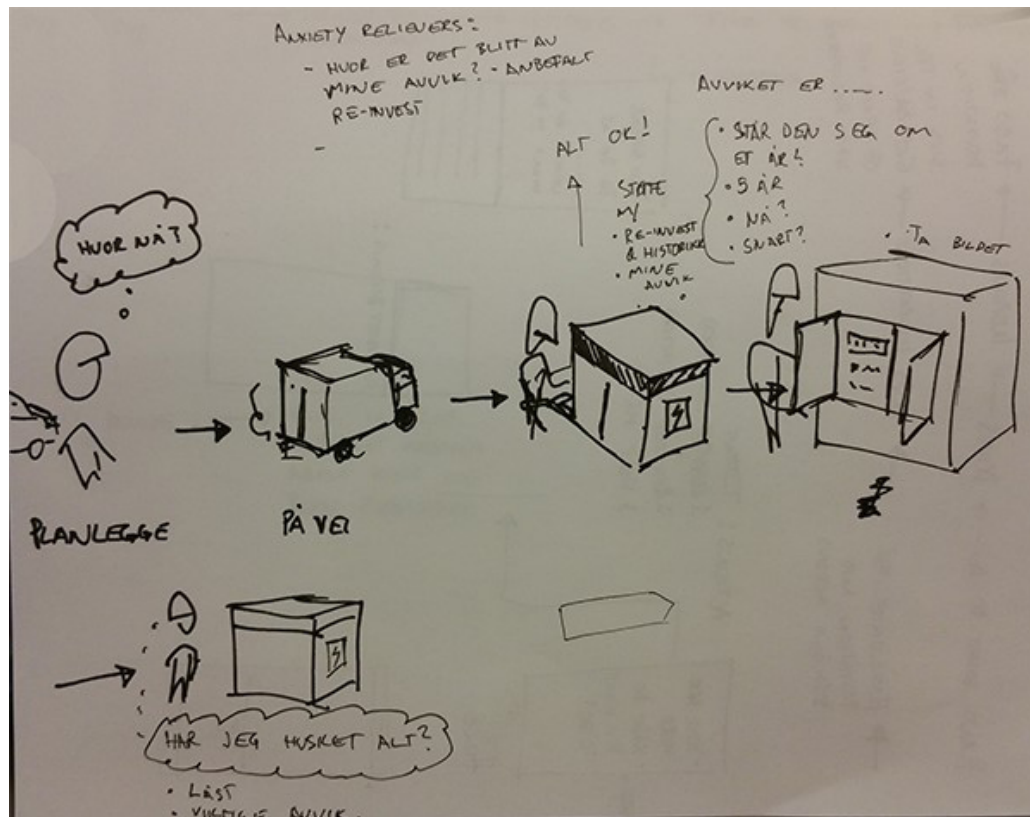
*Sketch of how a field engineer usually would inspect a power station*

The following was achieved by the end of the hackfest:

- Power station inspection
  - Checkpoint lists and dialogs appropriate to a specific facility
  - Selection of substations
  - Selection of discrepancy type
  - Registration of discrepancies with comments, integrated with Powel back end

The following was not implemented but was in scope for the hackfest:

- Power line inspection
- Translation of text from Norwegian to English
- Upload image from mobile app
- Triggering of conversation through voice without the use of a button

The current PoC chatbot allows the field engineer to execute an inspection by initiating a conversation through natural voice, ask for its capabilities, select a substation, and register a discrepancy of a specific type with a comment. This fulfills the feedback loop by sending the data directly to the Powel back end so that the field engineer no longer has to physically be present at the offices to register the same information.
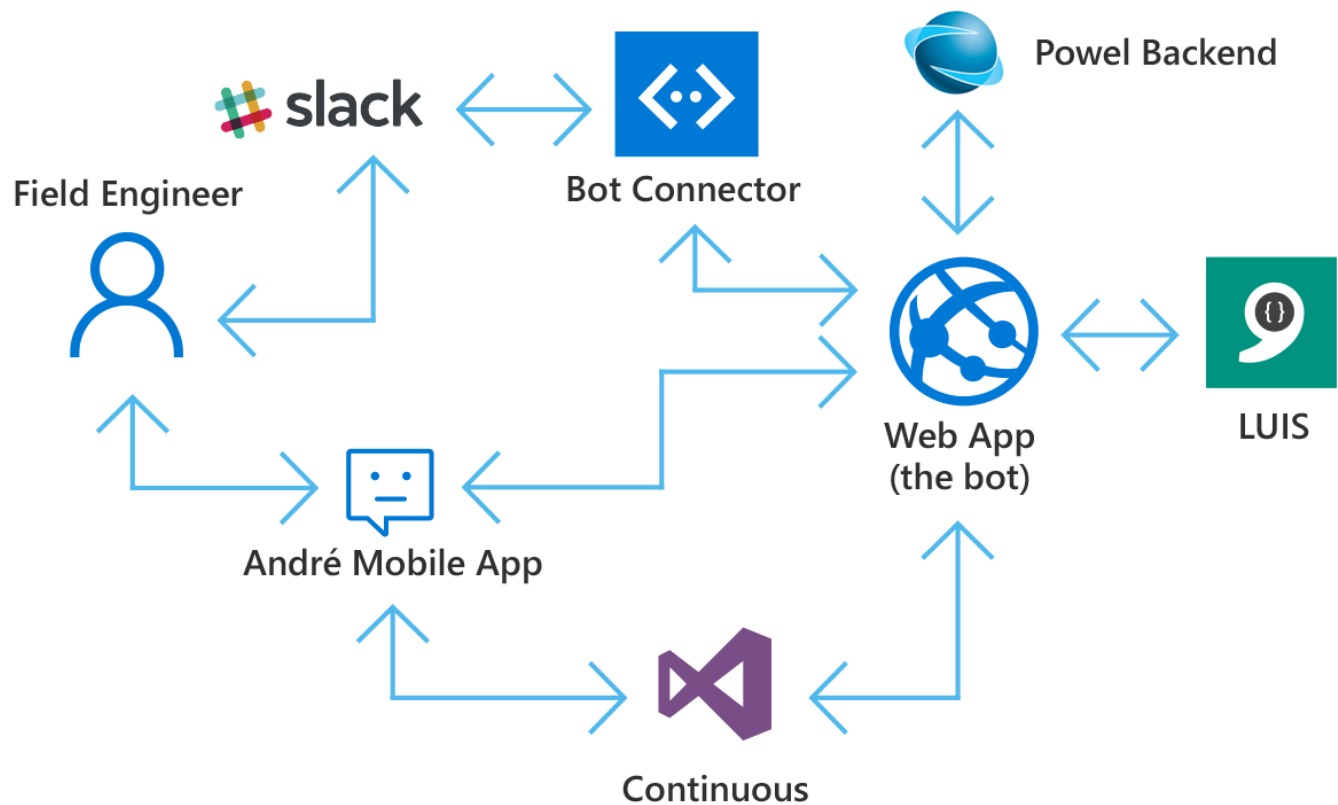
Architecture diagram

Technical delivery

Setting up the development environment

To get started with the project, we had to install and configure the following:

Visual Studio

- Install Visual Studio 2017 (Visual Studio 2015 also works)

  Note: Ensure that Xamarin is included in your Visual Studio installation for developing the André app

- Download and install the Bot Framework emulator

- Download and install the Bot template for Visual Studio for C#

Language Understanding Intelligent Services (LUIS)

To use LUIS, go to the LUIS homepage and create an account. You can try it free of

charge for a generous number of messages per day.

You now have everything you need to create a bot application. For a complete reference on getting started with Bot Builder, see Bot Builder SDK for .NET.

Achievements day 1

André is a Xamarin.Forms application, so we needed to implement native services to have the mobile app listen and speak through the Google speech-recognition interface. We also started work on the chat components for the app.

With LUIS, we had 10 intents described and trained with utterances. We started writing imaginary dialogs between a user and the app to get a better understanding of how the conversation would flow. We also started to experiment with LUIS entities and having intents recognize and use these entities.

For the Bot Framework, we created a bot application in the dev.framework.com space and a site in Azure for the web API.

Setting up the bot in Azure

The Bot Framework essentially operates like any ASP.NET app service. It has a controller that handles the conversation messages, so in essence, all you need to do is the following:

- Create or register a new bot on the Bot Framework developer site. This will give you a bot handle (GUID) and an app secret for your bot.

- Take a note of the app ID and password in the same portal.

- Deploy the bot code to Azure App Service.

- Configure your settings to include the app ID and password so that you can connect the web API to the Bot Framework.

(For a complete reference to this process, see Deploy a bot to the cloud.)

We opted to used the **CloudConfigurationManager** class to read the settings directly from App Service.

We set up CI in Visual Studio Team Services to build and deploy the bot to Azure on every commit. We then added LUIS integration to start working with the intents. At this stage, we had two intents implemented.

Achievements day 2

We finalized the direct connection through the use of the WebSockets portable class library (PCL) but ended up going for HTTP because we could not make the PCL work. This was largely because of the challenge of getting WebSockets to work on Android: We weren't sure at this stage whether we had a quick fix, so, in the interest of time, we opted out and focused our efforts on getting everything to work.

We successfully connected the bot application to the Powel back end (an API named Condition Monitoring) through the use of APIs.

A note about the back-end APIs

We connected the back-end APIs of Powel through some NuGet packages internal to Powel, and called the methods on the objects within.

Accessing these NuGet packages through Visual Studio Team Services for CI was a breeze. Because of the sensitivity of the code, we cannot share specific code examples, but in essence we used the APIs to get installation IDs by GPS location (longitude/ latitude) and for submitting the report data.

We connected the mobile app to the bot and made it work. We also spent some time designing the Xamarin.Forms app and making sure that the interface and user experience were good.

The following image is a rough initial sketch of the design interface, made by using PowerPoint Storyboarding.
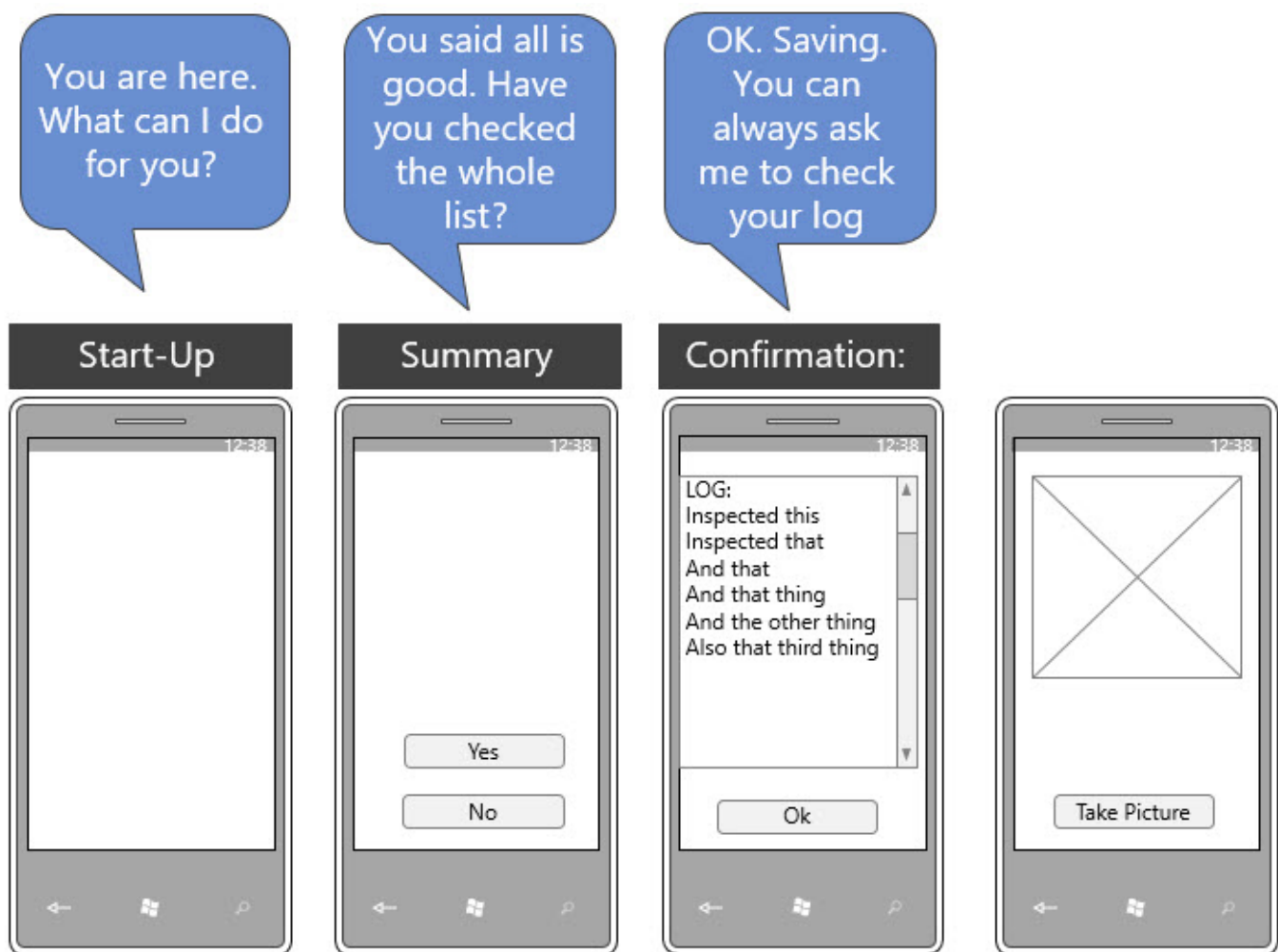
We further worked on getting NuGet packages from the Powel core APIs (specifically, the Condition Monitoring API) to be included in the build for the bot. We worked on some conversation flow plans for later implementation, and spent some more time on working with the intents and training LUIS to recognize them.

We spent a lot of time on the PromptDialog objects because we forgot to set the dialog implementation as **[Serializable]**.

Other pain points we experienced:

Some issues with the .NET Framework version not matching the Bot Framework WebApiConfig from the bot template made all parameter names in JSON camel-cased. This was not accepted by the internal Powel API, which took some time to figure out.
The configuration initially looked like this.

```
JsonConvert.DefaultSettings = () => new JsonSerializerSettings()
{
    ContractResolver = new CamelCasePropertyNamesContractResolver(),
    Formatting = Newtonsoft.Json.Formatting.Indented,
    NullValueHandling = NullValueHandling.Ignore,
};
```

We simply removed the **ContractResolver** line, and that seemed to fix the issue.

We ended the day by testing the bot outside to see how it handled background noise and whether it would be able to capture the intent of the field engineer. The location chosen was a noisy environment (a highway) that could very much be an actual inspection site where a field engineer would have to inspect an overhead line or a power mast. The testing succeeded with positive results.

Achievements day 3

The final part of the hackfest setting up build automation for the Xamarin.Forms project using Visual Studio Mobile Center, as well as finding possibilities for making a conversation that flows through different dialogs.

The following diagram show the final flow of the PoC chatbot. (Sadly, we did not implement text translation because of time constraints, but we included that part in the following image).
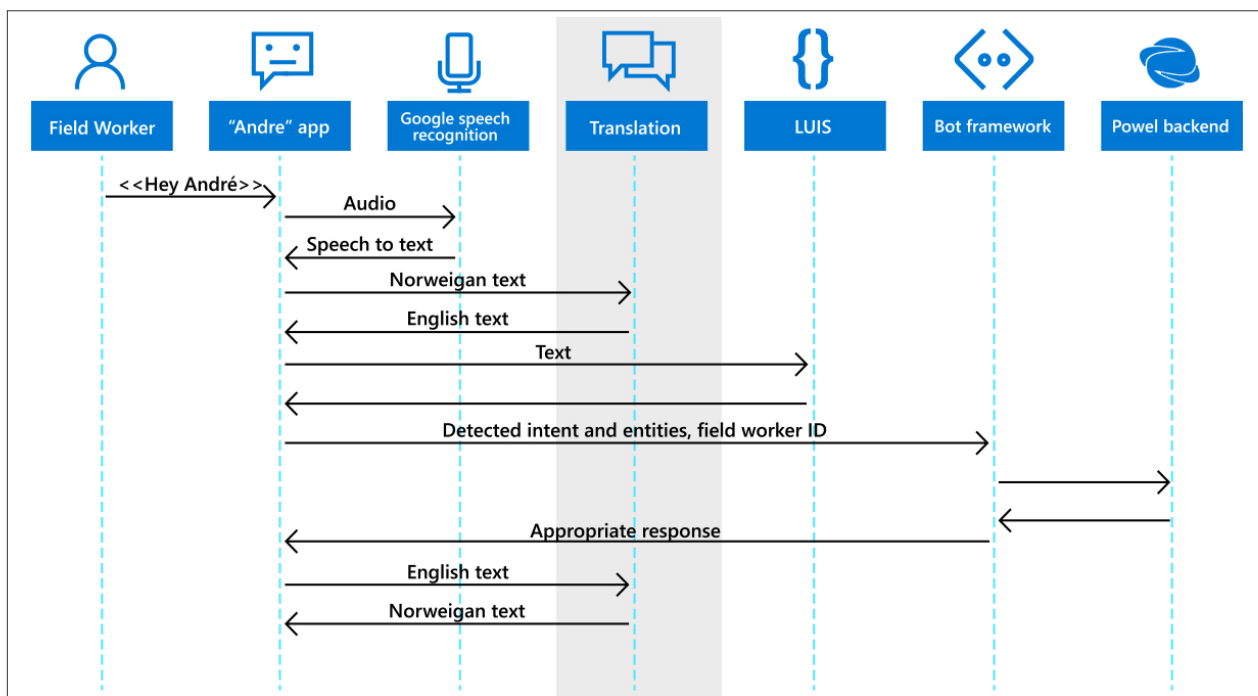
A field worker talks to André (the mobile app) in his natural language. The speech is transformed into text by the Google speech recognition engine and returned to the mobile app. The app sends the text for translation and receives fully translated text in return. The app then passes the translated text to LUIS, which captures the intent and entities and sends the response back to the mobile app. The mobile app sends the detected intent and entities (including the field worker ID) to the Bot Framework, which sends a request to the Powel back end. The appropriate response is returned to the mobile app, which translates the text into the field worker's language and speaks the text aloud.

Handling dialogs

In the Bot Framework, a dialog works as the "brains" of the chatbot.

A dialog provides a couple of neat features that manage the state of the conversation and session management, as well as persistence that relies on an implementation of an interface named **IDialog**. This interface manages the state transition using **IDialogContext**. These principals have been included in the way we worked with the bot during the hackfest and form an integral part of the bot, enabling it to steer a conversation using the underlying stack, where it saves data to the session object, which can be accessed throughout the course of the conversation.

An example: When LUIS acknowledges the user's intent to be happy, the Bot Framework will proceed to that particular method with the intent and respond with the appropriate response.

```
[LuisIntent("happy")]
public async Task Happy(IDialogContext context, LuisResult result)
{
    await context.PostAsync("That's so kind of you!");
    context.Wait(MessageReceived);
}
```

Another example is when LUIS acknowledges that the user wants to select a substation.

```csharp
[LuisIntent("selectSubstation")]
public async Task SelectSubstation(IDialogContext context, LuisResult
result)
{
    AssetService assetService = new AssetService(new
AssetRepository());
    string extractedName;
    string message = "I'm sorry, I didn't catch the name of the
substation.";

    if (TryExtractingSubstationName(result, out extractedName))
    {
        var asset = await assetService.GetByName(extractedName);
        substationName = asset != null ? extractedName : null;
        message = asset != null ? $"The substation {substationName}
has been chosen." : $"Sorry, I didn't find a substation called
'{extractedName}'.";
    }

    await context.PostAsync(message);
    context.Wait(MessageReceived);
}
```

More-complex dialogs were also developed in which the Bot Framework communicates with the appropriate Powel back end and lists the possible discrepancy types that the field engineer can choose from.

```
[LuisIntent("getDiscrepancyType")]
public async Task SelectControlPoints(IDialogContext context,
LuisResult result)
{
    if (String.IsNullOrEmpty(substationName))
    {
        await context.PostAsync($"You need to tell what substation you
want to select.");
        context.Wait(MessageReceived);
    }
    else
    {
        DiscrepancyTypeService discrepancyService = new
DiscrepancyTypeService(new DiscrepancyTypeRepository(new
InspectionRepository(new AssetRepository()), new AssetRepository())));
        AssetService assetService = new AssetService(new
AssetRepository());
        InspectionService inspectionService = new
InspectionService(new InspectionRepository(new AssetRepository()));
        var asset = await assetService.GetByName(substationName);
        var inspection = await
inspectionService.GetNotPerformedByAssetId(asset.ObjectId);
        var discrepancyTypes = await
discrepancyService.Get(asset.ObjectId, inspection.Id);

        await context.PostAsync($"You have your discrepancy types
here:\n {JsonConvert.SerializeObject(discrepancyTypes)}");
        context.Wait(MessageReceived);
    }
}
```

Prompt dialog

The Bot Framework includes several built-in prompts that can collect input from a user. This type of prompt was used throughout the bot project, such as when André asks the field engineer whether he is satisfied with the inspection and wants to finish. This is based on the recognition of a LUIS intent of **assetAllGood**.

```csharp
[LuisIntent("assetAllGood")]
public async Task AssetAllGood(IDialogContext context, LuisResult
result)
{
    if (String.IsNullOrEmpty(substationName))
    {
        await context.PostAsync($"You need to tell what substation you
want to select.");
        context.Wait(MessageReceived);
    }
    else
    {
        PromptDialog.Confirm(context, AfterConfirmAsync, "Are you sure
that you want to finish this inspection?");
        await Task.FromResult<Object>(null);
    }
}
```

In the preceding scenario, the field engineer is asked to confirm with either "Yes" or "No" (by either tapping the appropriate button or using natural voice to respond), but many other types of data-collection inputs can be chosen:

- **Prompts.text**   Asks the user to enter a string of text
- **Prompts.confirm**   Asks the user to confirm an action
- **Prompts.number**   Asks the user to enter a number
- **Prompts.time**   Asks the user for a time or date
- **Prompts.choice**   Asks the user to choose from a list of choices
- **Prompts.attachment**   Asks the user to upload a picture or video

FormFlow

As we have seen, dialogs work as a powerful mechanism to guide a conversation, but they can get complex when the puzzle has many small pieces, such as ordering a sandwich. To simplify a guided conversation, the Bot Framework offers a much easier way of building a dialog, named FormFlow. A FormFlow dialog guides the user through filling in the form while providing help and guidance along the way. The form dialog was used when the field engineer needs to submit a discrepancy, for which he has to fill in the remaining fields, including the type of discrepancy and the comment.

```csharp
[Serializable]
public class DiscrepancyFormDialogBuilder
{
    public string DiscrepancyType;
    public string Comment;

    public static IForm<DiscrepancyFormDialogBuilder> BuildForm()
    {
        return new FormBuilder<DiscrepancyFormDialogBuilder>()
            .AddRemainingFields()
            .Build();
    }

    public static IFormDialog<DiscrepancyFormDialogBuilder>
Build(FormOptions options = FormOptions.PromptInStart)
    {
        // Generated a new FormDialog<T> based on IForm<BasicForm>
        return FormDialog.FromForm(BuildForm, options);
    }
};
```

Connector

The Bot Framework Connector is a component that provides a single API for the bot to communicate across multiple client services, such as Skype, email, and Slack. Because we were already using Slack for communication throughout the hackfest, we decided to set up integration with Slack so that we could talk to the bot efficiently (read about the steps for configuring bot channels in the "Configuring Channels" section of Getting Started with the Connector.)

In the following image, you can see how testing the conversation using Slack looks.
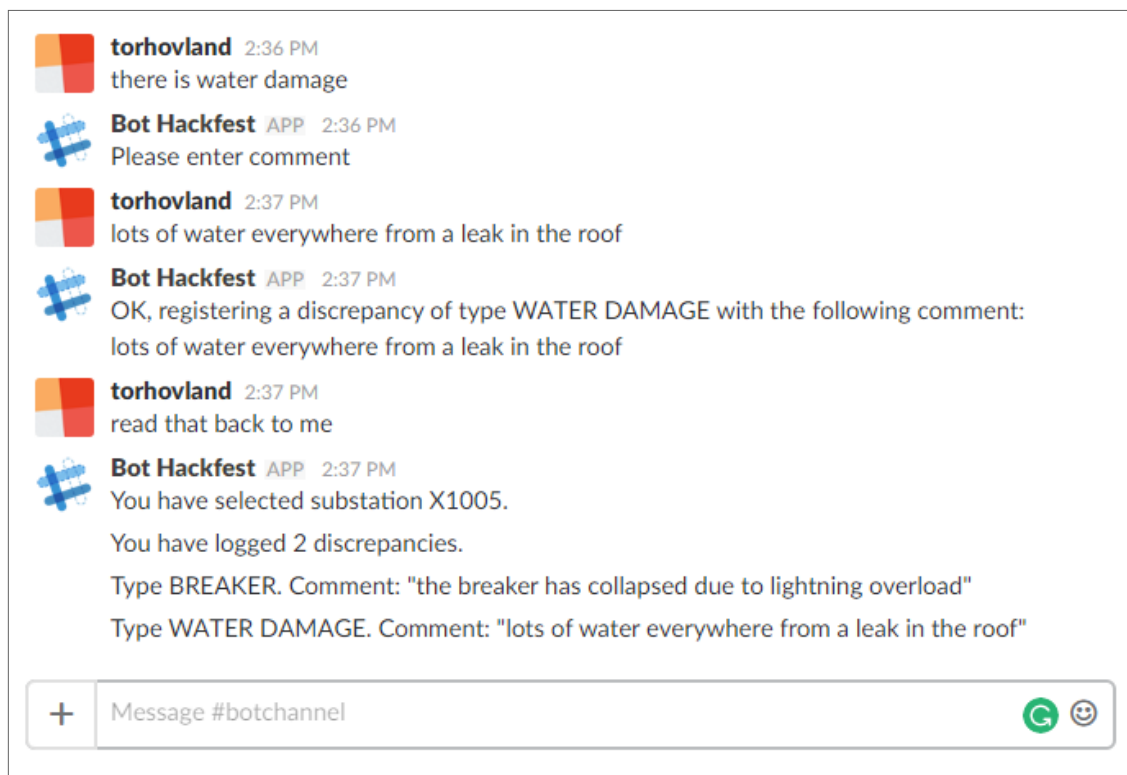
We managed to set up the Slack integration by registering the bot in the dev.botframework space and choosing the connector type without the need of any coding.

Bot Communication

Direct Line is a simple REST API for connecting directly to the bot. The main purpose of this API is to let developers connect custom agents to the bot to enable conversations. As mentioned during earlier, we initially set out to implement the connectivity through WebSockets in the mobile app developed in Xamarin.Forms but decided to go for HTTP because we couldn't make the PCL work. The following code shows a simple send task.

```
public async Task Send<T>(T item)
{
    var client = new HttpClient();
    client.DefaultRequestHeaders.Accept.Clear();
    client.DefaultRequestHeaders.Accept.Add(new
MediaTypeWithQualityHeaderValue("application/json"));
    client.DefaultRequestHeaders.Authorization = new
AuthenticationHeaderValue("Bearer",
_conversationStartedResponse.Token);
    var content = JsonConvert.SerializeObject(item);

    var response = await client.PostAsync($"https://
directline.botframework.com/api/conversations/
{_conversationStartedResponse.ConversationId}/messages", new
StringContent(content, Encoding.UTF8, "application/json"));
}
```

Other measures implemented

In the Bot Framework, you have to directly provide **BotId**, **MicrosoftAppId**, and **MicrosoftAppPassword** in the web.config file to set the appropriate app settings. Because of security issues, we did not want to do this. To provide the settings in a more secure way, we got the settings values from **CloudConfigurationManager** or a local JSON file that has the bot settings.

We wrote the following code for the settings reader.

```csharp
public string this[string index]
{
    get
    {
        var settingValue =
CloudConfigurationManager.GetSetting(index);

        if (string.IsNullOrEmpty(settingValue))
            settingValue = GetByLocalJsonFile(index);

        return settingValue;
    }
}

private string GetByLocalJsonFile(string index)
{
    var path =
Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.UserP
rofile), "FieldWorkerBotSettings.Json");
    if (!File.Exists(path))
        return string.Empty;

    var fileContent = File.ReadAllText(path);
    if (string.IsNullOrEmpty(fileContent))
        return string.Empty;

    var json = JObject.Parse(fileContent);

    if (!json.HasValues)
        return string.Empty;

    return json[index].Value<string>();
}
```

This allows us to create a new **LuisModelAttribute** object through we pass appId and **appKey** in the message controller.

```
private static readonly LuisService _service;

static MessagesController()
{
    var settings = new SettingsReader();
    var appId = settings["LUIS:AppId"];
    var appKey = settings["LUIS:AppKey"];
    var model = new LuisModelAttribute(appId, appKey);
    _service = new LuisService(model);
}
```

Because there is too much code to display and explain in this report, you can find all the source code for the hackfest in the GitHub repository readyforchaos/Powel-power-station-inspection-bot.

Agent UI

The following animated GIF shows the client mobile app on the left developed in Xamarin.Forms, and the Powel Condition Monitoring app on the right displaying the status of a power station.
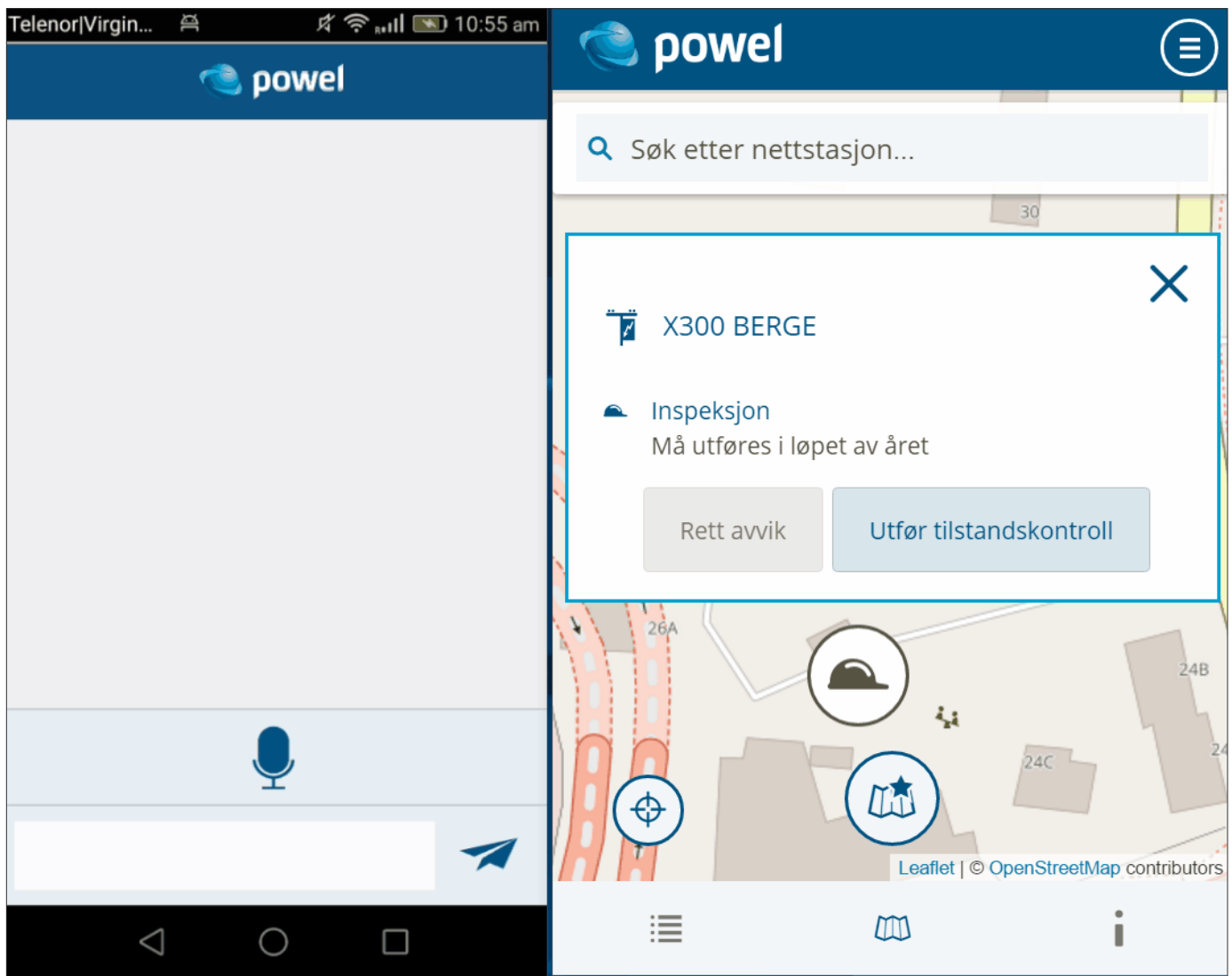
Conclusion

The results of the hackfest were brilliant, and it is mesmerizing to think that the developers from Powel went from knowing nothing about the Bot Framework to successfully building an integrated chatbot that could handle conversations across multiple systems and provide much-needed insight for the field engineers. Powel is dedicated to continuing the development of the chatbot during the next few months to include analytics and historical view of specific parts before going live, but also to extend the concept to areas and application domains of Powel other than just the power-business segment.

From a developer standpoint, all the necessary parts are there, and integrating them to create a solution felt almost too easy! The bumps that we encountered were not directly related to the Bot Framework itself, but more specific to Xamarin-related challenges.

The guidance provided for the Bot Framework and LUIS have the potential to improve, but it is sufficient to get you started, and once you've got the hang of it, you will spend the majority of your time designing conversational flows and less time on code.

Having all the information within the Powel back end accessible through natural speech within an app empowers the engineer to make decisions based on knowledge for which they can simply ask through the app, leading to more accurate inspection reports.

Being able to register discrepancies on the fly, ask for inspection details, and obtain other related information allows engineers to stay paperless during inspections so that they are less likely to forget an inspection point, and don't have the extra overhead of having to return to the office to digitize the findings after the fact.

The cool part about this hackfest is that we managed to build something that actually works and could be put into production in almost no time. With LUIS and the Bot Framework, we can now give the field worker the support he needs when he needs it and without the use of hands.

Going forward

We did not manage to include all the features that we scoped for the three-day hackfest. The scoped features that were left out were the following:

- Translating text
- Uploading images from the client app
- Triggering André by natural voice instead of a button
- Figuring out the WebSockets PCL and fixing it

Powel intends to iron out these issues shortly after the hackfest to produce a working prototype to be tested in the field as soon as possible.

Additional Resources

Want to try out Bot Framework right now? Start here: Microsoft Bot Framework

Want to try out LUIS and Microsoft Cognitive Services? Go to Microsoft Cognitive Services

Source code from the hackfest can be found on GitHub: readyforchaos/Powel-power-station-inspection-bot

Technical Case Study: NAVITIME

Technical case study

NAVITIME adds chatbots to improve the travel-app experience

Masayuki Ota
Jun 28, 2017

Travelers want unique experiences, but finding local food, sightseeing spots, and such can be challenging. With typical travel apps, travelers can find famous restaurants and locations but not local information. So travelers ask local people, which can be difficult given language barriers. For example, many travelers know sushi as Japanese food, but they don't know great local food such as shirasu-don and motsu-yaki.

NAVITIME wants to solve the problem, so they implemented a chatbot in the Kamakura Travel Guide app. Users can use the app to find information about famous attractions. In addition, users can communicate with the bot to find local food and interesting places.

Technologies used:

- Language Understanding Intelligent Service (LUIS)
- Bing Spell Check API
- Bing Images Search API
- Custom Vision Service
- Text Analytics API
- Microsoft Bot Framework
- Direct Line API
- Azure Web Apps
- Azure Functions
- API Management

- Azure Cosmos DB (previously named DocumentDB)
- Azure Search
- Azure Storage
- Power BI

NAVITIME Travel project members:

Ikuo Odanaka – Developer Manager, NAVITIME JAPAN

Shinichi Tanabe – Senior Software Development Engineer, NAVITIME JAPAN

Makoto Yoshihama – Software Development Engineer, NAVITIME JAPAN

Naoya Sasaki – Software Development Engineer, NAVITIME JAPAN

Ayako Omori (@ayako_omori) – Technical Evangelist, Microsoft Japan

Hiroyuki Watanabe (@hiwatan007) – Technical Evangelist, Microsoft Japan

Naoki Sato (@satonaoki) – Senior Technical Evangelist, Microsoft Japan

Daiyu Hatakeyama (@dahatake) – Principal Software Developer, Microsoft Japan

Masayuki Ota (@masota0517) – Technical Evangelist, Microsoft Japan

Customer profile

NAVITIME JAPAN is a leading provider of navigation technology and services. They offer mainly business-to-business (B2B) and business-to-consumer (B2C) navigation applications.

For B2B, they offer navigation apps for businesspeople, consulting for transportation, and advertising. For B2C, they offer navigation apps for traveling by train, bus, car, bicycle, or on foot and started a travel business named NAVITIME Travel.

Problem statement

Language barriers can challenge even the most experienced travelers. Navitime wanted to address this challenge before the Tokyo Olympics in 2020, when foreign travelers will visit Japan.

Solution and steps

Architecture
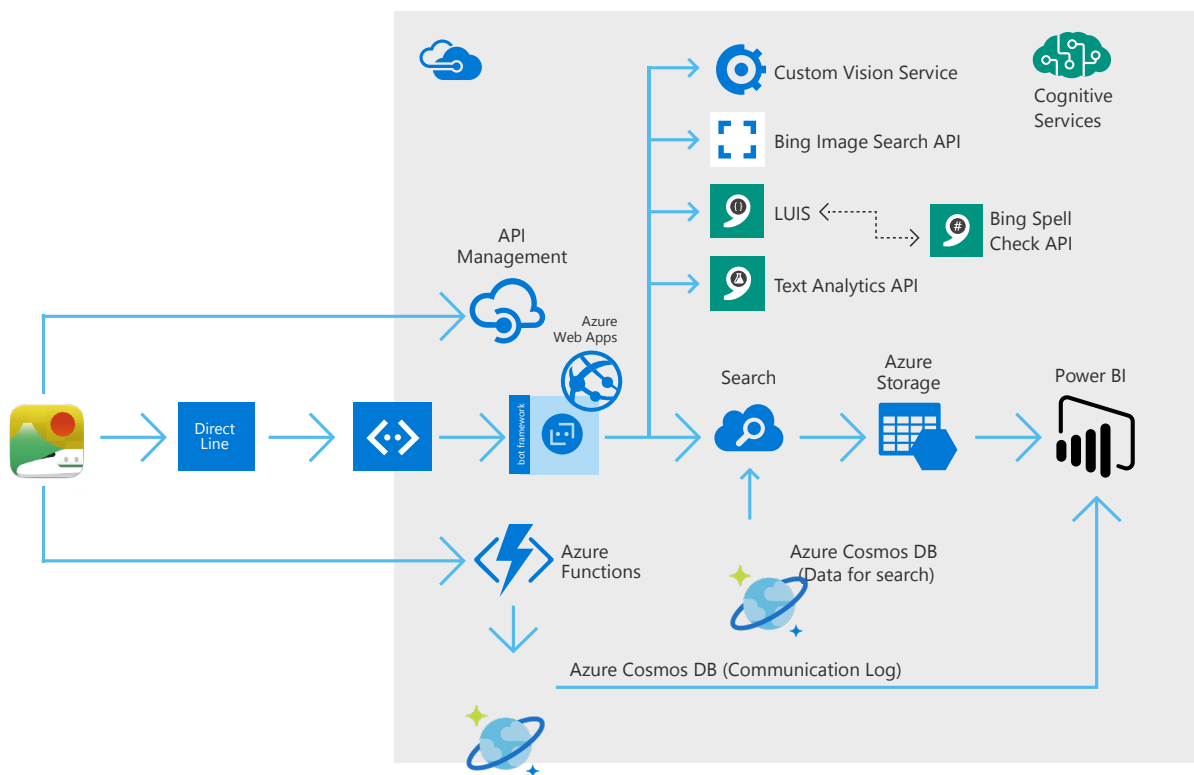
We used the Bot Framework to implement a chatbot, and we also used the Direct Line API to communicate with the chatbot from our iOS app. (Android and Windows apps can also use Direct Line.) For extracting intent and entities from user input, we used LUIS. We supported spelling correction by using the Bing Spell Check API before passing text to LUIS. After extracting intent and entities, we call Azure Search to fetch information about local food and sightseeing spots stored in Cosmos DB.

Users can freely input text, which means that sometimes LUIS can't identify entities. For this reason, we also use the Text Analytics API to get the key phrase. (See Detect sentiment, key phrases, and languages in the Azure Text Analytics Quick Start Guide.) We then pass the key phrase to Azure Search.

Users can also communicate with the bot by using photos. To support this scenario, we implemented image recognition by using the Bing Images Search API and Custom Vision API.

DevOps and continuous improvement are also important when developing bots. For managing bot versions, we used API Management. For storing the user-input log, we call Azure Functions and save data in Cosmos DB. We also use Azure Search, Azure Storage, and Power BI for monitoring search terms. We can get user demands from the text logs and brush up the bot again and again.

User experience by app and chatbot combination

We implemented the bot in an iOS app because we thought that a chatbot could extend the app's user experience. We show a notification message (the green window in the following image) in the app and invite users to use the chatbot in two situations:

1. When a user starts the app for the first time, the notification message says, "Welcome! Feel free to ask me for travel tips and spot information!"

2. If the app detects that the user has seen several recommended articles but has not selected one, it concludes that the user can't find suitable information and says, "Not interested in our recommended articles? You can also…"
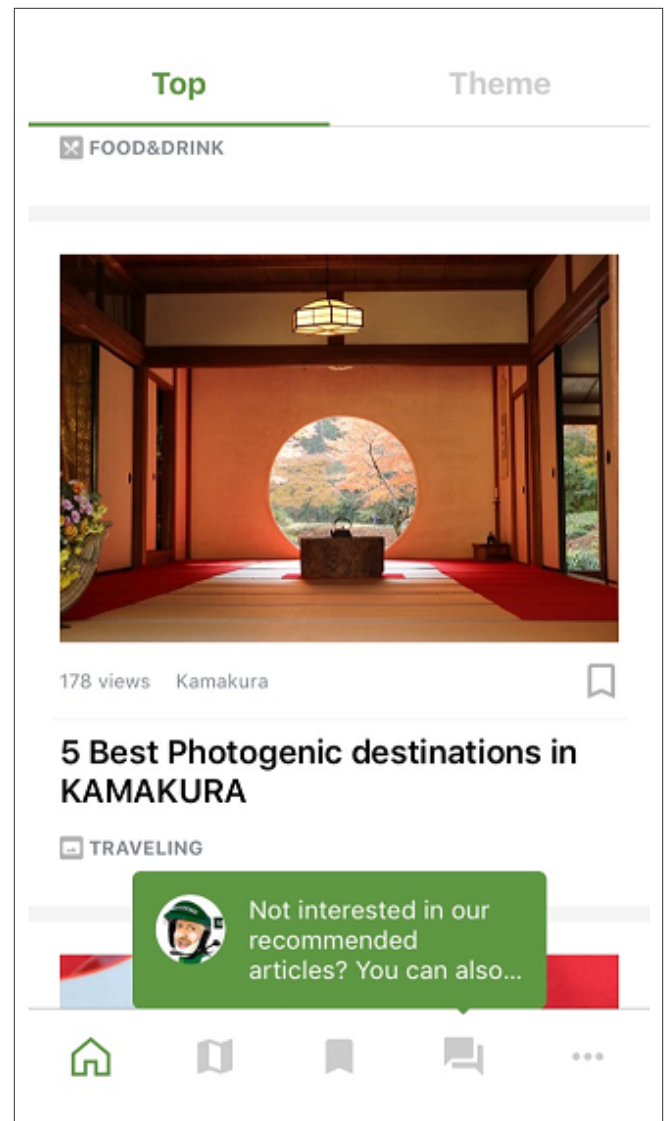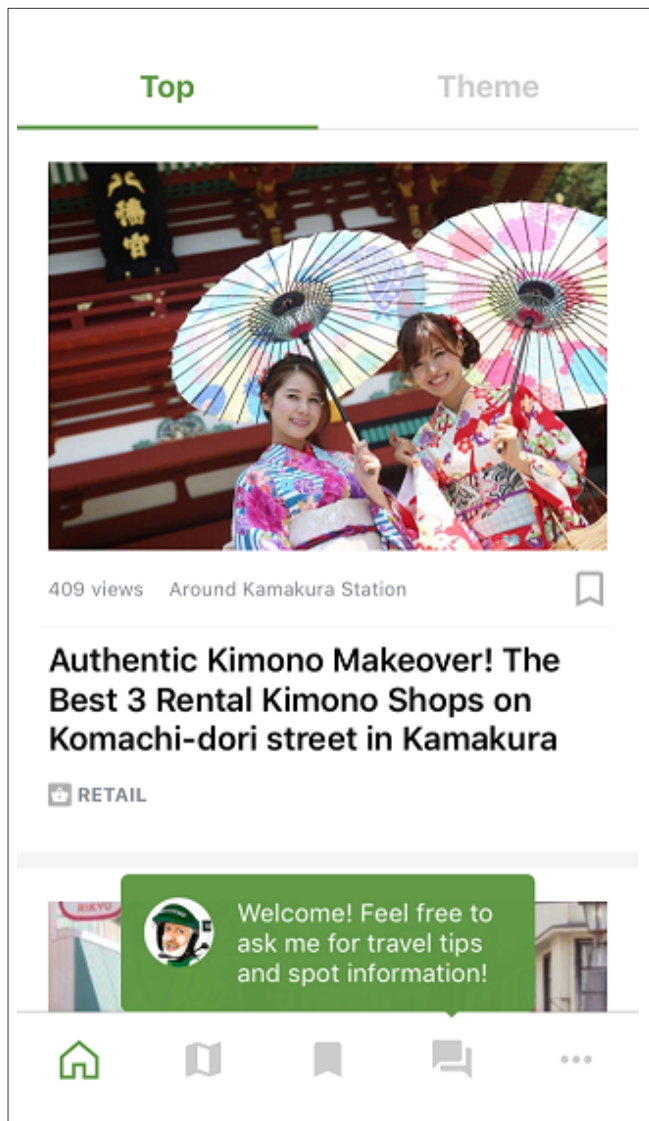
Users can tap the chat button (fourth from left) in the menu list to start communication with the chatbot. Users can input free text in English and Japanese, such as "I want to eat soba." The bot replies "Do you want...?" and recommends a restaurant. The app also shows a menu from which users can select the next action.
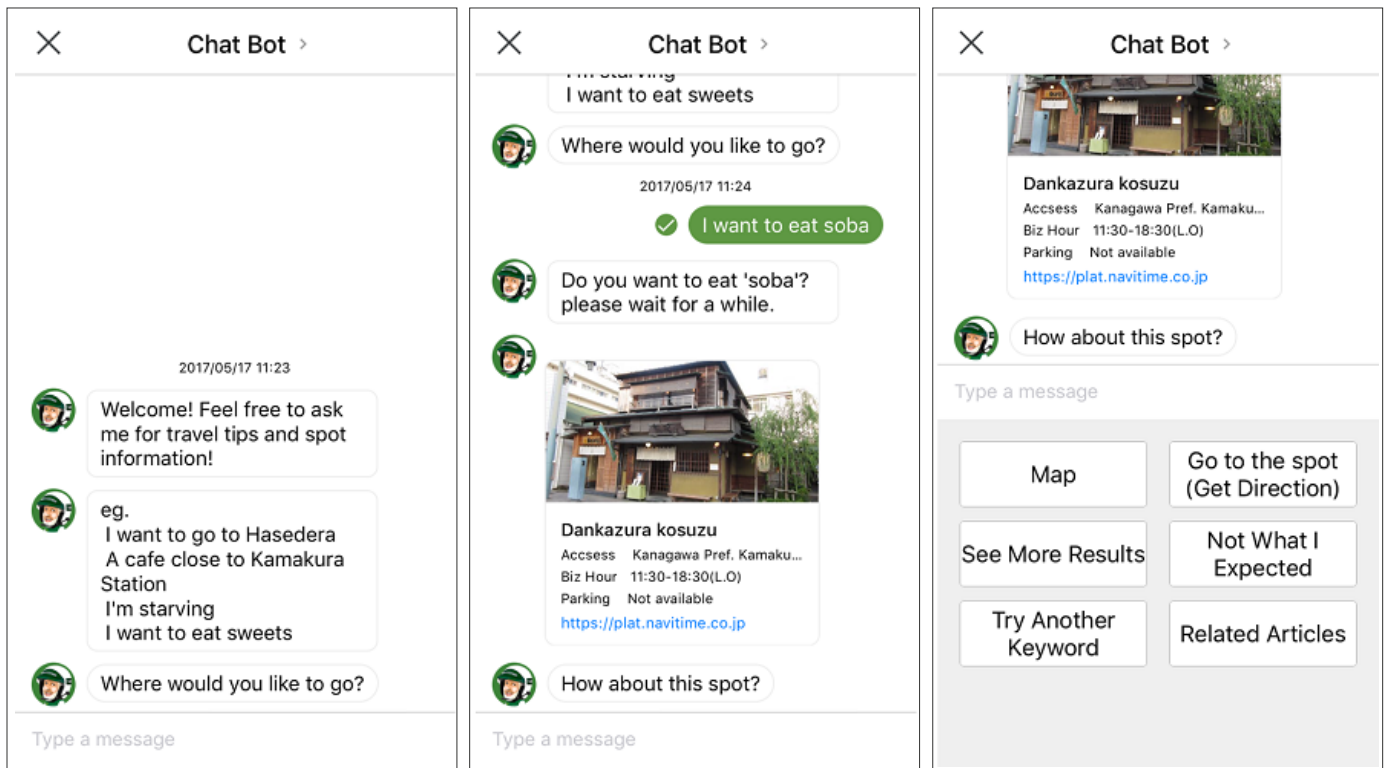
The user can tap "See More Results" to see another recommendation or tap "Try Another Keyword" to enter different keywords. Tapping the Map button displays a map; by tapping "Go to the spot (Get Direction)" a user can both see the map and how to get to the location. Because NAVITIME has own routing technology, the bot redirects the user to the NAVITIME routing web app.
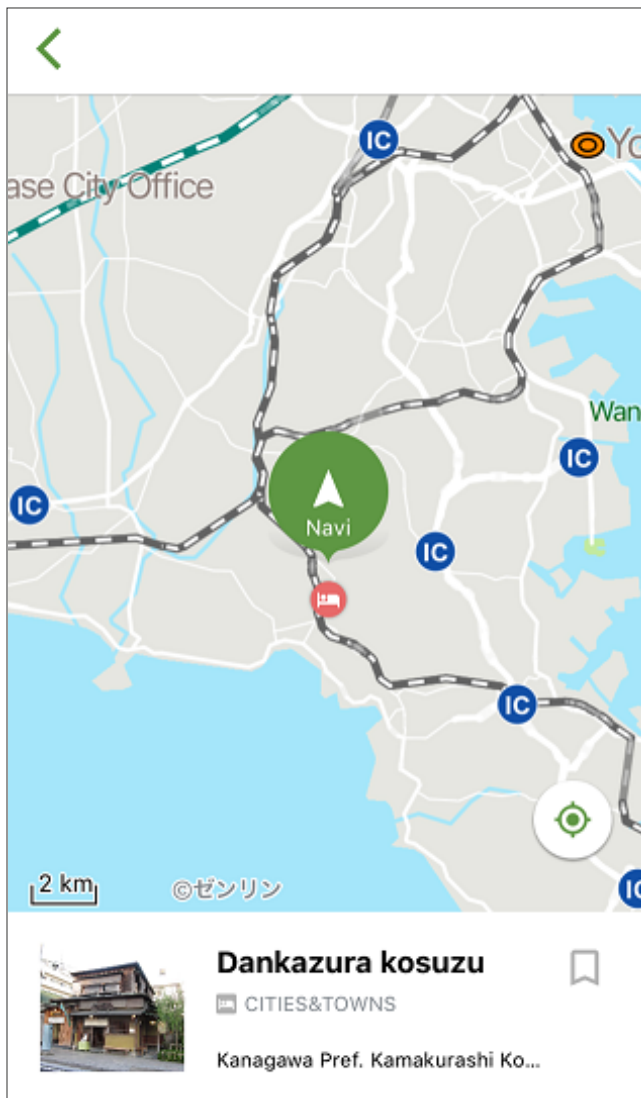
The user can tap "See More Results" to see another recommendation or tap "Try Another Keyword" to enter different keywords. Tapping the Map button displays a map; by tapping "Go to the spot (Get Direction)" a user can both see the map and how to get to the location. Because NAVITIME has own routing technology, the bot redirects the user to the NAVITIME routing web app.

Travelers tend to upload photos to social network services such as Instagram and Facebook. When other users see those photos, they want to go to that place or eat that food—but they don't know where it is or how to get it. Users can sent photos to our bot, which helps to recognize images and tell users what it is and how to get it.

Technical delivery

This section describes how to implement a similar bot by using Bot Framework, Cognitive Services, and Azure.

Prerequisites

- Install Visual Studio and the Bot Builder SDK by following the steps in the tutorial Create a bot with the Bot Builder SDK for .NET.
- Install the Bot Framework Emulator; see Debug bots with the Bot Framework Emulator for details.
- Create a free Azure account if you don't already have one.
- Create an app with LUIS to fit your scenario and call it from the bot (see LUIS action binding for web apps).

Develop a bot with Bot Framework

After the prerequisite steps, we started to implement our own bot by using the

MultiDialogSample code on GitHub. Because we wanted to release an iOS app, we used the Direct Line API.

Extract intent and entities with LUIS and the Bing Spell Check API

For extracting intent and entity from user messages, we used LUIS. Because users sometimes mistype and LUIS can't extract intent, we also use the Bing Spell Check API to detect and correct typos before sending messages to LUIS with steps below.

1. From the Azure portal, add the Bing Spell Check API.
2. In the LUIS menu, choose **My keys**.
3. Choose **External Keys** and **Add a new key** to add a Bing Spell Check API key to LUIS.
4. Go to your LUIS app and choose **Publish App** in the left menu.
5. At the bottom, choose the **Add Key Association** button to bind the Bing Spell Check API to LUIS.
6. Select **Enable Bing spell checker**, which changes the endpoint URL. Use this URL to call LUIS with the Bing Spell Check API.

Save NoSQL data in Cosmos DB

We used the fully managed NoSQL database service Cosmos DB to store NAVITIME location information, which is formatted as JSON. We also use Cosmos DB for storing logs of communication between users and bots.

If you want to learn how to initialize Cosmos DB and write code for it, start with the tutorial Azure CosmosDB: Develop with the DocumentDB API in .NET. You need to import data for testing and production by following the instructions in How to import data into Azure Cosmos DB for the DocumentDB API.

Implement Azure Search

Although LUIS can extract intents well, users sometimes input just words, not sentences, so we also use Azure Seach. With the following architecture, we can handle sentences and words and reply with the correct information.

To use Azure Search, we wrote C# code like the following.

```csharp
[Serializable]
public class AzureSearchService
{
    private static readonly string QueryString = $"https://
{WebConfigurationManager.AppSettings["SearchName"]}.search.windows.net
/indexes/{WebConfigurationManager.AppSettings["IndexName"]}/docs?api-
key={WebConfigurationManager.AppSettings["SearchKey"]}&api-
version=2015-02-28&";

    public async Task<SearchResult> SearchByName(string name)
    {
        using (var httpClient = new HttpClient())
        {
            string nameQuey = $"{QueryString}search={name}";
            string response = await
httpClient.GetStringAsync(nameQuey);
            return
JsonConvert.DeserializeObject<SearchResult>(response);
        }
    }

    public async Task<FacetResult> FetchFacets()
    {
        using (var httpClient = new HttpClient())
        {
            string facetQuey = $"{QueryString}facet=Era";
            string response = await
httpClient.GetStringAsync(facetQuey);
            return
JsonConvert.DeserializeObject<FacetResult>(response);
        }
    }

    public async Task<SearchResult> SearchByEra(string era)
    {
        using (var httpClient = new HttpClient())
        {
```

You can use it from the Bot Framework with the following code. If you want to see a sample project, go to ryanvolum/AzureSearchBot on GitHub. This project describes how to set up Azure Search and Cosmos DB and how to call Azure Search from a bot.

```csharp
public virtual async Task MessageRecievedAsync(IDialogContext context,
IAwaitable<IMessageActivity> result)
{
    var message = await result;
    try
    {
        SearchResult searchResult = await
searchService.SearchByName(message.Text);
        if(searchResult.value.Length != 0)
        {
            CardUtil.showHeroCard(message, searchResult);
        }
        else{
            await context.PostAsync($"No musicians by the name
{message.Text} found");
        }
    }
    catch(Exception e)
    {
        Debug.WriteLine($"Error when searching for musician:
{e.Message}");
    }
    context.Done<object>(null);
}
```

Recognize images with Bing Image Search and Custom Vision Service

We can use Bing Image Search to find similar images and "best representative queries." For example, if you send a photo of the temple seen earlier in this article to Bing Image Search, it replies with URLs of similar images and "Tsuruoka Hachimangu Temple, Kamakura" as **bestRepresentativeQuery**. Bing has enormous knowledge of images, and we can use it to recognize famous places and foods with the following code.To use Azure Search, we wrote C# code like the following.

```csharp
public virtual async Task MessageRecievedAsync(IDialogContext context,
IAwaitable<IMessageActivity> result)
{
    var message = await result;
    try
    {
        SearchResult searchResult = await
searchService.SearchByName(message.Text);
        if(searchResult.value.Length != 0)
        {
            CardUtil.showHeroCard(message, searchResult);
        }
        else{
            await context.PostAsync($"No musicians by the name
{message.Text} found");
        }
    }
    catch(Exception e)
    {
        Debug.WriteLine($"Error when searching for musician:
{e.Message}");
    }
    context.Done<object>(null);
}
```

You can find a sample project in the GitHub repo NT-D/suggesttriplocationBot and learn about how to post images to a bot and how to use Bing Image Search to recognize images and get **bestRepresentativeQuery**. You can see more detail in the Bing Image Search API Reference.

Although Bing Image Search helps us to find famous places and foods, it's difficult to recognize local or lesser-known places and foods. Therefore we decided to use Custom Vision Service. To build, test, and use this API, we started with the getting-started document Build a classifier using Custom Vision Service machine learning.

Drive feedback cycles with Power BI, Cosmos DB, and Azure Search

Because it is important to improve the bot logic with real input from users, we want to see user input and search logs easily.

To save user searches, we call Azure Functions and store data in Cosmos DB. Azure Functions natively support Cosmos DB bindings and can save data by using only a little code, such as the following.

```csharp
#r "Newtonsoft.Json"

using System;
using System.Net;
using Newtonsoft.Json;

public static HttpResponseMessage Run(HttpRequestMessage req,
TraceWriter log, out string outputDocument)
{
    log.Info($"Webhook was triggered!");
    string jsonContent = req.Content.ReadAsStringAsync().Result;
    //Store Data in Cosmos DB
    outputDocument = jsonContent;

    dynamic data = JsonConvert.DeserializeObject(jsonContent);

    //Return HTTP Response (BadRequest or OK)
    if (data.first == null || data.last == null) {
        return req.CreateResponse(HttpStatusCode.BadRequest, new {
            error = "Please pass first/last properties in the input
object"
        });
    }

    return req.CreateResponse(HttpStatusCode.OK, new {
        greeting = $"Hello {data.first} {data.last}!"
    });
}
```

You can use Power BI to access the data in Cosmos DB by following the information in Data sources in Power BI Desktop. To analyze search logs, see Analyzing your data with Power BI.

Improve response performance

We deployed the bot in the West US region to minimize response time. (The Bot Framework State service and LUIS are hosted in the West US region, and the Direct Line endpoints are in Eastern Asia, Europe, and North America.)

At first we deployed our bot in the Japan East region, which caused slow response

because when the bot calls LUIS and the State service, the traffic round-trips the Pacific Ocean again and again. If you experience slow response, you can improve the performance by moving your bot to the West US region.
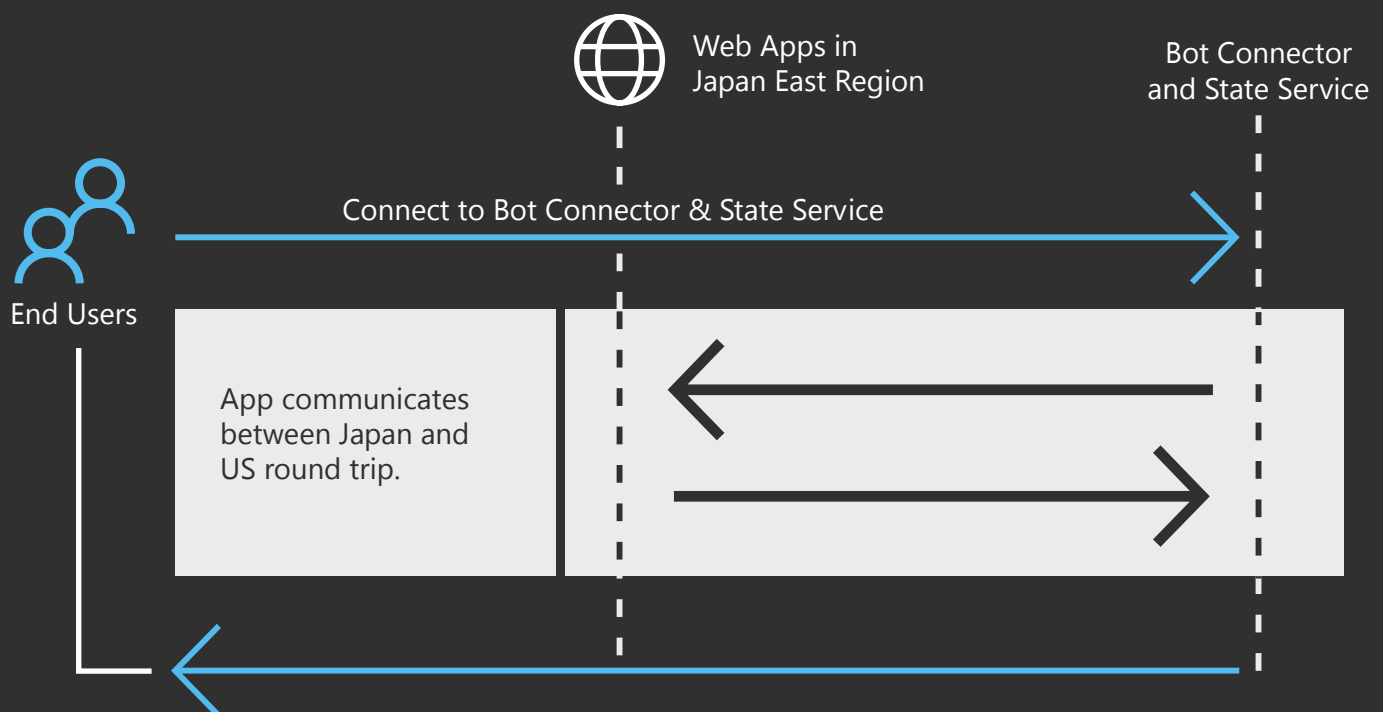
Conclusion

Travelers want to have an unique expeience in trip, but hard to find local food, sightseeing spot and more with normal app. It's also difficult to find it by communicating with local people because of language barrier.

The chatbot is now live in the Kamakura Travel Guide app. Users can find both famous and obscure local information by using both the app and its built-in bot.

"It's difficult to make NLP logic in multi-languages, but LUIS solves this problem. LUIS is really good for getting intents in the message."

Ikuo Odanaka

Developer Manager, NAVITIME JAPAN

"It's [Cosmos DB] a very cool data store. We can save NoSQL data in it and can fetch data with SQL-like queries, so it is very easy to use. Read/ write speed is very quick; integration with Azure Seach is seamless."

Shinichi Tanabe

Senior Software Developer, NAVITIME JAPAN

"It is the smoothest project for me. PaaS such as Cosmos DB and Azure Search can help us make new things rapidly."

Shinichi Tanabe

Senior Software Developer, NAVITIME JAPAN

As these case studies have shown, Azure AI is making a **real difference in how developers work and create.** Ready-to-use AI tools, advanced cloud infrastructure, and a flexible platform offer developers what they need to move their apps into a new world of exciting products, superior customer service, and greater business agility.

Developers have big opportunities as AI becomes increasingly important in a world focused on customer experience and as businesses seek new ways to gain competitive advantage. To make the most of those opportunities, developers are looking for:

1. Powerful and productive tools.

Developers don't want to spend time writing code for core AI features. They need ready-to-go APIs that are easily implemented into apps.

2. Data and AI for every developer.

Developers want to augment their applications with AI in a way that's best for them. To get the most of AI, they want to build it into their apps using a wide range of programming languages, as well as choosing from a range of data sources.

3. An open and flexible platform.

Developers are happiest when they can choose the technology and frameworks that are best suited for their scenarios and skills.

What's next?

We invite you to explore Azure AI Services
via these following links:

→ Learn more about Azure AI

→ Try Cognitive Services APIs for free

→ Create your Azure free account