Build a home assistant mobile application with Watson and IoT Platform services

Khong Kok Sing  
June 01, 2017

The tutorial shows how a mobile application can use the Watson Assistant, Text to Speech, and Speech to Text services to understand user commands, which are then used to control devices through IBM IoT Platform services. It also shows how to integrate a Raspberry Pi as a home gateway that receives commands from and sends events to the mobile app. Finally, it shows how to store images by using Object Storage Service.

Watson Conversation is now Watson Assistant. Although some illustrations in this tutorial may show the service as Watson Conversation, the steps and processes will still work.

In the past, building a home assistant required a lot of effort and could be technically challenging. However, with IBM Watson and other complementary cloud services, you now have the power to create and innovate right at your fingers.

This tutorial explains how to combine the power of Watson with the simplicity of the IBM IoT platform to create a home assistant to control some basic electronic devices (a light and camera). The framework can be extended to control actual home appliances that are enabled with the connectivity. You can use this tutorial as a guide to building your own application.

Developed in iOS Swift, the Home Assistant mobile app communicates with IBM Watson Assistant by using both key inputs and voice commands; the user interfaces let you send messages or speak. I used the Watson Text to Speech and Speech to Text services to make this conversation possible. Watson Assistant is trained to derive an intent based on voice commands or input messages. Based on the intents, the Home Assistant application sends commands to the devices through the IoT Platform service and a Home Gateway (a Raspberry Pi). The Home Gateway controls the devices and sends device events to the Home Assistant app through the IoT Platform service. The system also uses Object Storage Service to store pictures that are uploaded by the Home Gateway.

You can download the Home Assistant code as a start to building your own app.

Get the code
The basic flow of the application is:

- (1) The user speaks or types in a command from the Home Assistant user interface.
- (2, 3) The voice command is transcribed into text by using the Watson Speech to Text service. This step is skipped if the user types in the command.
- (4, 5) The transcribed text is forwarded to Watson Assistant so that it can derive an intent from the user's speech.
- (6) On receiving the intent, the Home Assistant constructs a command JSON message and sends it to the IoT Platform.
- (7) A Node-RED application is running on Home Gateway, which subscribes to the command topic and receives the command message.
- (8a, 8b) The Home Gateway performs the action, either turning the light on or off or taking a picture.
- (8c) The captured picture is uploaded to Object Storage.
- (9) Upon completion of the task, the Node-RED application sends a device event (containing the status) as a JSON message to the IoT Platform.
- (10) Home Assistant, which subscribes to the device topic, receives the device event.
- (10a) The captured picture is downloaded from Object Storage and displayed on the user interface.
- (11, 12) Home Assistant extracts the status text, and forwards it to the Watson Text to Speech service to be translated to audio.
- (13) Home Assistant then plays the audio to the user.
The following screen captures show the Home Assistant mobile application.

What you'll need to build your application

To build an application like the Home Assistant, you'll need to be familiar with:

- The iOS development environment
- Swift programming
- The Node-RED development environment
- Watson Developer SDKs
- The IoT Platform and MQTT protocol

To build the app, I used the following development environments, hardware, and IBM Cloud services:

**Development environments**

- MacBook
- Xcode V8.3.1
- CocoaPods
- Carthage

**IBM Cloud services**

- IBM IBM Cloud
- Watson Text to Speech
• Watson Speech to Text
• Watson Watson Assistant
• Watson Internet of Things (IoT) Platform
• Object Storage OpenStack Swift

Hardware
• Raspberry Pi 3B Starter Kit, includes SD card and power adapter (with USB cable)
• Adafruit NeoPixel Diffused 8mm Through-Hole LED - 5 Pack
• 5 MP webcam video camera module board 1080 p x720 p fast for Raspberry Pi 3B/2B/B+
• Female-to-female wire jumper cable 1P-1P
• Capacitor, 0.1µF
• Resistor, 560 Ω
• Ethernet cable (needed only for the first time you boot the hardware)
• USB keyboard (needed only for the first time you boot the hardware)
• HDMI display cable (needed only for the first time you boot the hardware)
• Mouse (needed only for the first time you boot the hardware)

1. Prepare Watson services on IBM Cloud

From the IBM Cloud catalog, select and create the following services. Make sure that you save all of the credentials.

1. Log in to IBM Cloud.
2. Click Catalog.
3. Click Watson under the Services list.
4. Click Watson Assistant.
5. After the Service is created, click Service credentials > View credentials and record the Service credential names.
6. Click Create.

Repeat these steps to create instances of the Watson Text to Speech and Watson Speech to Text services.

2. Import the Watson Assistant service

1. After you create the Watson Assistant service instance, click Launch tool to launch it.
2. You can click Create to start creating a new conversation and defining intents, entities, and dialogs or click Import to import a complete workspace. For this tutorial, I import a file.
3. Click Import, select workspace-homeassistant.json (included in the files under Get the code), and click Import.
4. Return to the workspace and click the three vertical dots icon to open the menu. Select View details. This loads the details of the workspace including the workspace ID.
5. Copy the WORKSPACE_ID. You need this when using the Watson Assistant service from your mobile application.

Now, I'll discuss the artifacts that are defined in the Watson Assistant service. The dialogs are simple because the focus is more on the integration between the various IBM Cloud services, mobile application, and the Node-RED running on Raspberry Pi.
The Watson Assistant service is used mainly to deduce the intent based on the user's interaction by message or voice. The Dialogs are used to drive the conversation.

<table>
<thead>
<tr>
<th>Dialog</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>This is the conversation start dialog.</td>
</tr>
<tr>
<td>Greetings</td>
<td>Handles greetings from the user.</td>
</tr>
<tr>
<td>On Light</td>
<td>Converts user commands to an &quot;On Light&quot; intent.</td>
</tr>
<tr>
<td>Off Light</td>
<td>Converts user commands to an &quot;Off Light&quot; intent.</td>
</tr>
<tr>
<td>Take Picture</td>
<td>Converts user commands to a “Take a Picture” intent.</td>
</tr>
<tr>
<td>Else</td>
<td>This is a catchall that handles all other cases.</td>
</tr>
</tbody>
</table>

3. Prepare Object Storage services on IBM Cloud

1. From the IBM Cloud Catalog, select **Storage > Object Storage** and create an instance. Remember to select the Free tier.
2. After you create the Object Storage, copy the Credentials. Save the `OS_PROJECTID`, `OS_USERID`, `OS_USERNAME`, and `OS_PASSWORD` needed to connect to Object Storage from the Node-RED and mobile applications.

```json
{
    "auth_url": "https://identity.open.softlayer.com",
    "project": "object_storage_750f36e0_f61b_42c0_9458_0876db9f3e36",
    "projectId": [OS_PROJECTID],
    "region": "dallas",
    "userId": [OS_USERID],
    "username": [OS_USERNAME],
    "password": [OS_PASSWORD],
    "domainId": "598e3dcdff9a4d7eb7246a6852bdcc16",
    "domainName": "1307809",
    "role": "admin"
}
```
4. Prepare Watson IoT Platform service

To prepare the Watson IoT Platform service, you'll need to connect the Raspberry Pi as a 'Device' and the mobile application as an 'Application' to the IoT Platform.

Connect Raspberry Pi as a device to the IoT Platform

1. From the IBM Cloud catalog, create an instance of the IoT Platform service by clicking Internet of Things > Internet of Things Platform.
2. Select the Lite plan, and click Create.
3. Click Launch on the Internet of Things Platform page.
4. From the left navigation bar, click Devices.
5. Select the Device Type tab, and click Create Type to create the device type.
6. In the Create Device Type window, click Create device type, and specify the Name and Description. Save the DEV_TYPE name.

7. Click Next until your device type is created.
8. Click Devices in the left navigation bar again, and then click Add device.
9. In the Add Device window, select your device from the Choose Device Type drop-down list and click Next.
10. In the **Add Device > Define Info** window, specify the Device ID and Serial Number (used in Node-RED to connect). Save the `DEV_ID` to be used in the Node-RED Credentials configuration.

11. Accept the defaults for the other fields, and click **Next** until the device is added. An authentication token is shown. Save `ORG_ID`, `DEV_ID`, and `AUTH_TOKEN` to be used in Node-RED Credentials configuration.

You can also get more details from this developerWorks recipe.

**Connect the mobile application to the IoT Platform**

1. From the left navigation bar, click **Apps** to go to the Apps page.
2. Click **Generate API Key** to generate an API key for your mobile application.
3. In the Generate API Key window, select **Standard Application** for the API Role(s) and save the `API_KEY` and `APP_AUTH_TOKEN` to be used later in your mobile application. Click **Generate**.
5. Prepare Raspberry Pi

You can follow these detailed directions to set up the Raspberry Pi. I used Raspbian Jessie with Pixel because it comes installed with Node-RED, which will be required later.

By default, the wifi connection is not set up. You need to connect your Raspberry Pi to a router through an Ethernet cable. You also need to connect your USB keyboard and HDMI display cable.

1. Connect to your wifi network by clicking the wifi button on the upper right corner.
2. Select Pi > Preferences > Raspberry Pi Configuration to enable the required interfaces: SSH and VNC for you to access the Raspberry Pi without a keyboard and screen and Remote GPIO and Camera.

3. Connect the Raspberry Pi with the LED and camera according to the following figure.
   - The GPIO 3 V PIN is wired to the 5 V PIN of the LED.
   - The GPIO GND PIN is wired to the GND PIN of the LED.
   - The GPIO18 PIN is wired to the resistor (560 Ω), before going to the DIN PIN of the LED.
   - The Capacitor (0.1µF) is wired in parallel to the 5 V PIN and the GND.
   - The Camera is wired to the Camera CSI on the Raspberry Pi.
6. Develop Node-RED on Raspberry Pi

Next, you'll need to develop and prepare the Node-RED environment on your Raspberry Pi.

**Prepare Node-RED environment on Raspberry Pi**

1. From the terminal, ssh to the Raspberry Pi and navigate to the Node-RED folder. You might need to check for the IP address by logging in to your router.
   
   ```
   ssh pi@ipaddr-raspberrypi
   ```

2. Install `node-red-node-pi-neopixel` to control the NeoPixel LED.
   
   ```
   curl -sS get.pimoroni.com/unicornhat | bash
   npm install node-red-node-pi-neopixel
   ```

3. Install `node-red-contrib-camerapi` to control the camera.
   
   ```
   npm install node-red-contrib-camerapi
   ```

4. Install `node-red-contrib-objectstore` to access the IBM Cloud Object Store.
   
   ```
   npm install node-red-contrib-objectstore
   ```

   Other required nodes, the Watson IoT input and output nodes, are preinstalled.

5. Run the following command to configure Node-RED to autostart when the system is booted.
   
   ```
   sudo systemctl enable nodered.service
   ```

6. Open a browser and point to http://ipaddr-raspberrypi:1880 to access Node-RED. You'll find the following nodes installed in the palletes.

   - camerapi
   - takephoto
   - rpi neopixels
   - os get
   - os put
   - os del
Develop flow to control LED

Before you follow this step, make sure that you completed the previous step to prepare the Node-RED environment in your Raspberry Pi. Then, you'll need to navigate to the Node-RED application from your browser.

1. Connect the flow as follows. It receives a command, decides the intent, executes the intent (turn the LED on or off), and sends an event to the mobile application.

The following table explains what each portion of the flow controls.

<table>
<thead>
<tr>
<th>Node name</th>
<th>Node type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Light cmd</td>
<td>Watson IoT input</td>
<td>Used to receive commands from the mobile application through the IoT platform. The format is iot-2/cmd/[CMD_TYPE]/fmt/json, where [CMD_TYPE] is defined in the Command field (for example, “light”).</td>
</tr>
<tr>
<td>Decide Intent</td>
<td>Function node</td>
<td>Gets the intent from the payload, and reformats the payload to turn the LED on or off.</td>
</tr>
</tbody>
</table>

![Diagram of the flow](image-url)
### Action Switch Node

```javascript
action = msg.payload.action
object = msg.payload.object
intent = msg.payload.intent
if (intent == "OnLight") {
    msg.payload = "#ffffff"
    return [msg, null]
} else if (intent == "OffLight") {
    msg.payload = "#000000"
    return [null, msg];
}
return msg;
```

### NeoPixel LED
NeoPixel LED is configured as follows:

**Edit rpi-neopixels node**

- **LEDs**: 1
- **Mode**: Bar - Percent of length
- **Background**: HTML colour name or rgggbb
- **Foreground**: HTML colour name or rgggbb
- **Wipe time**: 40 mS
- **Name**: NeoPixel LED

### Format Status On

**Function node**

Construct a status 'on' message in JSON
```javascript
var jsonObj = { "dev":"light", "status": "on"};
msg.payload = JSON.stringify(jsonObj)
return msg;
```

### Format Status Off

**Function node**

Construct a status 'off' message in JSON
```javascript
var jsonObj = { "dev":"light", "status": "off"};
msg.payload = JSON.stringify(jsonObj)
return msg;
```

### Light Event

**Watson IoT output node**

Used to send the device event from the mobile application through the IoT platform. The format is
```
iot-2/type/[DEV_TYPE]/id/[DEV_ID]/evt/[EVT_TYPE]/fmt/json
```
where:
- `[DEV_TYPE]` is defined in the IoT Platform service.
- `[DEV_ID]` is defined in the IoT Platform service.
- `[EVT_TYPE]` is configured in the field Event type (for example, light).
There are Inject nodes that are used to test the On and Off functions of the LED. The Debug nodes are used to send the message payload to the Debug window on the right side.

The Credentials configuration objects obtained from the IoT Platform and used by the Watson IoT nodes are:

- **Organization:** ORG_ID
- **Server-Name:** ORG_ID.messaging.internetofthings.ibmcloud.com
- **Device Type:** DEV_TYPE
- **Device ID:** DEV_ID
- **Auth Token:** AUTH_TOKEN
- **Name:** raspberrypi
Develop a flow to take a picture and upload to Object Storage

Before you follow this step, make sure that you completed the step to prepare the Node-RED environment in your Raspberry Pi. Then, you'll need to navigate to the Node-RED application from your browser.

1. Connect the flow as shown in the following image. The flow receives a command to take a photo and after capturing the photo, it uploads the photo to Object Storage and generates an event.

![Flow diagram](image)

The following table explains what each portion of the flow controls.

<table>
<thead>
<tr>
<th>Node name</th>
<th>Node type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camera Cmd</td>
<td>Watson IoT input node</td>
<td>Used to receive commands from the mobile application through the IoT platform. The format is <code>iot-2/cmd/[CMD_TYPE]/fmt/json</code>, where <code>[CMD_TYPE]</code> is defined in the Command field (for example, camera). It uses credentials that are defined in the <code>raspberrypi</code> configuration object.</td>
</tr>
<tr>
<td>Take Photo</td>
<td>camerapi take photo node</td>
<td>Takes a picture by using the Raspberry Pi camera. The file mode &quot;Generate&quot; creates a file in the folder. The file name, folder name, and format are found in <code>msg.filename</code>, <code>msg.filepath</code>, and <code>msg.fileformat</code>, respectively.</td>
</tr>
</tbody>
</table>
### ObjectStorage Upload

**os-put node**

Gets the captured image file and uploads the file to Object Storage in IBM Cloud. The command picks up the file that is specified in the input message (msg.filename, msg.filepath, and msg.fileformat), and uploads the file into the container specified in the node. Upon successful upload, it returns the URL in msg.url and the object name in msg.objectname. It gets the service credentials from the object storage configuration.
**Format Event**

**Function node**

Constructs an event message in JSON that contains the URL, the object name, and the container name so that the mobile application knows where to download the file.

```javascript
var json = {
    "url": msg.url,
    "objectname": msg.objectname,
    "containername": "visual-recognition-images"
};

msg.payload = JSON.stringify(json);
return msg;
```

**Camera Event**

**Watson IoT output node**

Used to send the device event from the mobile app through the IoT platform. The format is:

```
iot-2/type/[DEV_TYPE]/id/[DEV_ID]/
evt/[EVT_TYPE]/fmt/json
```
• [DEV_TYPE] is defined in the IoT Platform service.
• [DEV_ID] is defined in the IoT Platform service.
• [EVT_TYPE] is configured in the field Event type (for example, camera).

Edit Watson IoT node

Connect as
- Device
- ○ Quickestart
- ○ Registered

Credentials: raspberrypi

Event type: camera

Format: json

QoS

Name: Camera Event

‘none’

os-config node

Defines the login credentials to the Object Storage service. These are obtained from the Object Storage Credentials that you saved previously.
- Configuration Information: API Based
- Region: Dallas
- Tenant Id: OS_PROJECTID
- User Id: OS_USERID
- User Name: OS_USERNAME
- Password: OS_PASSWORD

Edit os-config node
7. Develop the mobile application

Because this app is similar to a chat messaging interface, I use a popular iOS user interface widget called JSQMessagesViewController. There is a similar sample code that I could use but it is written in Objective C. However, it's still a good reference to use.

Develop the user interface

Prepare Xcode project

1. From Xcode, create a Single View Application by clicking File > Project. Select Single View Application and specify the Product Name: Home Assistant.
2. Go to the project folder and initialize the CocoaPods. A Podfile is generated.
   pod init
3. Add this line to the Podfile to install the JSQMessagesViewController widget.
   pod 'JSQMessagesViewController'
4. Run the command to install JSQMessagesViewController dependencies. An Xcode workspace is generated. You should re-open Xcode using the Workspace (*.wcworkspace) instead of the Project (*.xcodeproj).
   pod install
5. In the ViewController.swift file, import the JSQMessagesViewController module and change the ViewController class to inherit from the JSQMessagesViewController class. Declare the array of messages to hold the chat messages.
   import JSQMessagesViewController
   class ViewController: JSQMessagesViewController {
       var messages = [JSQMessage]()
   }
6. Finally, enter the following code in Info.plist to enable the microphone.
   <key>NSMicrophoneUsageDescription</key>
   <string>Need microphone to talk to Watson</string>

Set up the user interface and create an extension

To set up the user interface and create an extension:

1. Create a file called UIExt.swift to contain all of the UI-related logic as a Swift extension. Then, declare an extension.
   extension ViewController {
   }
2. Create a setupUI() function that:
   - Initializes the title, senderId, and senderDisplayName.
   - Creates a microphone button and registers touchDown and touchUpInside events to callback functions.
   - Registers a menu item (synthesizes to callback function).

   func setupUI() {
       self.title = "Watson Chat"
       self.senderId = UIDevice.current.identifierForVendor?.uuidString
       self.senderDisplayName = UIDevice.current.identifierForVendor?.uuidString
       JSQMessagesCollectionViewCell.registerMenuAction(#selector(synthesize(sender:)))
   }
3. Add the microphone icon as Assets. Click **Assets.xcassets**, and drag the icon files to the workspace to create the image sets.

![Microphone icons](image)

**Implement handler functions for the user interface**

In the **UIExt.swift** file:

1. Add the following functions to handle the microphone button press event. When you press this button to speak a command, it starts to stream to the Watson Speech to Text service.

```swift
func didPressMicrophoneButton(sender: UIButton) {
    let microphonePressedImage = UIImage(named:"microphone_pressed")!
    sender.setImage(microphonePressedImage, for: .normal)
    AudioServicesPlayAlertSound(SystemSoundID(kSystemSoundID_Vibrate))
    // Clear the input text
    self.inputToolbar.contentView.textView.text = ""
    // speech-to-text start streaming
    sttStartStreaming()
}
```

2. Add the following functions to handle the microphone button release event, which stops the streaming to the Watson Speech to Text service.

```swift
func didReleaseMicrophoneButton(sender: UIButton){
    let microphoneImage = UIImage(named:"microphone")!
    sender.setImage(microphoneImage, for: .normal)
    // speech-to-text stop streaming
    self.sttStopStreaming()
}
```

3. Override the **didPressSend()** function to handle when the send button is pressed. The code appends the message to the message array, then sends a request and receives a response from the Watson Assistant service.

```swift
override func didPressSend(_ button: UIButton!, withMessageText text: String!, senderId: String!, senderDisplayName: String!, date: Date!) {
    send(text)
}
```
Override UI callback functions

In the UIExt.swift file, add the following functions:

1. Override the callback `collectionView | numberOfItemsInSection` function to return the number of messages count in the array.
   ```swift
   override func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
       return self.messages.count
   }
   ```

2. Override the callback `collectionView | cellForItemAt` function that sets the text and renders the color for a specific cell at the index path.
   ```swift
   override func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
       let cell = super.collectionView(collectionView, cellForItemAt: indexPath) as! JSQMessagesCollectionViewCell
       let message = self.messages[indexPath.item]
       if !message.isMediaMessage {
           if message.senderId == self.senderId {
               cell.textView.textColor = UIColor(R: 0x72, G: 0x9B, B: 0x79)
           } else {
               cell.textView.textColor = UIColor(R: 0x47, G: 0x5B, B: 0x63)
           }
           let attributes : [String: AnyObject] = [
               NSForegroundColorAttributeName: cell.textView.textColor!, NSUnderlineStyleAttributeName: 1 as AnyObject
           ]
           cell.textView.linkTextAttributes = attributes
       }
       return cell
   }
   ```

3. Override the callback `collectionView | messageBubbleImageDataForItemAt` function to return whether it is an incoming or outgoing bubble for a specific cell at the index path.
   ```swift
   override func collectionView(_ collectionView: JSQMessagesCollectionView!, messageBubbleImageDataForItemAt indexPath: IndexPath!) -> JSQMessageBubbleImageDataSource! {
       let data = messages[indexPath.row]
       switch(data.senderId) {
       case self.senderId:
           return self.outgoingBubble
       default:
           return self.incomingBubble
       }
   }
   ```

4. Override the callback `collectionView | attributedTextForMessageBubbleTopLabelAt` function to set and return the sender name as the label on top of the message bubble, beside the style attributes.
   ```swift
   override func collectionView(_ collectionView: JSQMessagesCollectionView!, attributedTextForMessageBubbleTopLabelAt indexPath: IndexPath!) -> NSAttributedString! {
       if let message = firstMessage(at: indexPath) {
           let paragraphStyle = NSMutableParagraphStyle()
           paragraphStyle.alignment = NSTextAlignment.left
           let attrs = [
               NSParagraphStyleAttributeName: paragraphStyle,
               NSBaselineOffsetAttributeName: NSNumber(value: 0),
               NSForegroundColorAttributeName: UIColor(R: 0x1e, G: 0x90, B: 0xff)
           ]
           return NSAttributedString(string: message.senderDisplayName, attributes: attrs)
       } else {
           return nil
       }
   }
   ```
5. Override the callback `collectionView | heightForMessageBubbleTopLabelAt` function to return the height of the text label.

```swift
override func collectionView(_ collectionView: JSQMessagesCollectionView!,
    layout collectionViewLayout: JSQMessagesCollectionViewFlowLayout!,
    heightForMessageBubbleTopLabelAt indexPath: IndexPath!) -> CGFloat {
    if let _ = firstMessage(at: indexPath) {
        return kJSQMessagesCollectionViewCellLabelHeightDefault
    } else {
        return 0.0
    }
}
```

**Utility functions**

Finally, also in the UIExt.swift file, add the following functions:

1. The `send()` function plays a sound, appends the sent message into the array of messages, calls the `finishSendingMessage()` function, and calls the `conversationRequestResponse()` function that handles the interaction with the Watson Assistant service.

```swift
func send(_ text: String) {
    setAudioPortToSpeaker()
    JSQSystemSoundPlayer.jsq_playMessageSentSound()
    let message = JSQMessage(senderId: self.senderId, senderDisplayName: self.senderDisplayName, date: Date(), text: text)
    self.messages.append(message!)
    self.finishSendingMessage(animated: true)
    self.conversationRequestResponse(text)
}
```

2. The `firstMessage(at:)` function returns the first message that is not previously from the same sender to indicate the sender at the top of the text bubble.

```swift
func firstMessage(at: IndexPath) -> JSQMessage! {
    let message = self.messages[at.item]
    if message.senderId == self.senderId {
        return nil
    } else if at.item - 1 > 0 {
        let previousMessage = self.messages[at.item-1]
        if previousMessage.senderId == message.senderId {
            return nil
        }
    }
    return message
}
```

3. The `didReceiveConversationResponse()` function is called when Watson Assistant returns a response. The function plays a sound, appends the message to the array of messages, and dispatches the response to the Watson Text to Speech service to synthesize the sentence. Finally, it calls the `finishReceiveMessage()` function.
func didReceiveConversationResponse(_ response: [String]) {
    let sentence = response.joined(separator: " ")
    if sentence == "" { return }
    setAudioPortToSpeaker()
    JSQSystemSoundPlayer.jsq_playMessageReceivedSound()
    let message = JSQMessage(senderId: "Home Assistant", senderDisplayName: "Home Assistant", date: Date(), text: sentence)
    self.messages.append(message!)

    DispatchQueue.main.async {
        // text-to-speech synthesize
        self.ttsSynthesize(sentence)
        self.reloadMessagesView()
        self.finishReceivingMessage(animated: true)
    }
}

See the [UIExt.swift](#) file for complete implementation (included in the files under Get the code)

### Interface with Watson Services

Now it’s time to implement the interface with Watson Service.

#### Prepare the Watson SDK

1. Install the [Watson Swift SDK](#). You must use Carthage to do this, and one way to install Carthage is by using Homebrew (a package manager for MacOS).
2. Install Homebrew.
   ```bash
   /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
   ```
3. Install Carthage.
   ```bash
   brew update
   brew install carthage
   ```
4. Create a Cartfile in the project folder and add the following line.
   ```text
   github "https://github.com/watson-developer-cloud/swift-sdk"
   ```
5. Install the Swift SDK framework.
   ```bash
   carthage update --platform iOS
   ```
6. If you get an error similar to:
   ```text
   Module compiled with Swift 3.1 cannot be imported in Swift 3.0.2
   ```
   Then recompile the binaries with this command:
   ```bash
   carthage update --platform iOS --no-use-binaries
   ```
7. Add the Swift SDK framework to the project by navigating to General > Linked Frameworks > Libraries. Click the + icon.

8. Click Add Other and navigate to the Carthage/Build/iOS folder. Select the frameworks: TextToSpeechV1.framework, SpeechToTextV1.framework, ConversationV1.framework, and RestKit.framework.
When you are done, you should have the following frameworks added.

<table>
<thead>
<tr>
<th>Name</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConversationV1.framework</td>
<td>Required</td>
</tr>
<tr>
<td>RestKit.framework</td>
<td>Required</td>
</tr>
<tr>
<td>SpeechToTextV1.framework</td>
<td>Required</td>
</tr>
<tr>
<td>TextToSpeechV1.framework</td>
<td>Required</td>
</tr>
<tr>
<td>Pods_Home_Assistant.framework</td>
<td>Required</td>
</tr>
</tbody>
</table>

9. Copy the frameworks into your application to make them accessible at runtime. Go to the **Build Phases** tab, click the + icon, and select **New Run Script Phase**.

10. In the Run Script, specify:

```
/usr/local/bin/carthage copy-frameworks
```

11. Specify the following Input Files by clicking the + icon.

```
$(SRCROOT)/Carthage/Build/iOS/TextToSpeechV1.framework
$(SRCROOT)/Carthage/Build/iOS/SpeechToTextV1.framework
$(SRCROOT)/Carthage/Build/iOS/ConversationV1.framework
$(SRCROOT)/Carthage/Build/iOS/RestKit.framework
```
12. Finally, enter the following lines in the Info.plist file as source code to connect to the Watson IoT platform.

```xml
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSEXceptionDomains</key>
    <dict>
        <key>watsonplatform.net</key>
        <dict>
            <key>NSEXceptionRequiresForwardSecrecy</key>
            <false/>
            <key>NSEXceptionIncludesSubdomains</key>
            <true/>
            <key>NSEXceptionAllowsInsecureHTTPLoads</key>
            <true/>
            <key>NSEXceptionMinimumTLSVersion</key>
            <string>TLSv1.0</string>
        </dict>
    </dict>
</dict>
```

**Initialization**

1. Create a file called WatsonExt.swift to contain the code that interfaces with the Watson Text to Speech, Speech to Text, and Assistant services. Declare an extension for ViewController.

```swift
Initialization

1. Create a file called WatsonExt.swift to contain the code that interfaces with the Watson Text to Speech, Speech to Text, and Assistant services. Declare an extension for ViewController.

```
Build a home assistant mobile application with Watson and IoT Platform services

```swift
conversation = Conversation(username: Credentials.ConversationUsername,
password: Credentials.ConversationPassword,
version: "2017-03-12")
let failure = { (error: Error) in print(error) }
conversation?.message(withWorkspace: Credentials.ConversationWorkspaceID, failure: failure) { 
    response in
    print("output.text: \(response.output.text)"
    self.didReceiveConversationResponse(response.output.text)
    self.context = response.context
}
```

### Watson Speech to Text service

The `sttStartStreaming()` function is used to connect to the Speech to Text service and can start the streaming request. It also starts the microphone input. The function is called when a user presses the microphone button.

```swift
func sttStartStreaming() {
    // define settings
    var settings = RecognitionSettings(contentType: .opus)
    settings.continuous = true
    settings.interimResults = true

    self.speechToTextSession?.connect()
    self.speechToTextSession?.startRequest(settings: settings)
    self.speechToTextSession?.startMicrophone()
}
```

The `sttStopStreaming()` function is used to stop the Speech-to-Text service. It also stops the microphone input. The function is called when a user releases the microphone button.

```swift
func sttStopStreaming() {
    self.speechToTextSession?.stopMicrophone()
    self.speechToTextSession?.stopRequest()
    // No need to disconnect -- the connection will timeout if the microphone
    // is not used again within 30 seconds. This avoids the overhead of
    // connecting and disconnecting the session with every press of the
    // microphone button.
    //self.speechToTextSession?.disconnect()
    //self.speechToText?.stopRecognizeMicrophone()
}
```

The `onResults()` callback function updates the text view widget based on the best transcript result from the Speech to Text service.

```swift
func onResults(results: SpeechRecognitionResults) {
    self.inputToolbar.contentView.textView.text = results.bestTranscript
    self.inputToolbar.toggleSendButtonEnabled()
}
```

### Watson Text to Speech

The `ttsSynthesize()` function is used to synthesize a speech based on a sentence by using the Text to Speech service. It plays the synthesized audio data.
func ttsSynthesize(_ sentence: String) {
    // Synthesize the text
    let failure = { (error: Error) in print(error) }
    self.textToSpeech?.synthesize(sentence, voice: SynthesisVoice.gb_Kate.rawValue, failure: failure) { data
        in
            self.audioPlayer = try! AVAudioPlayer(data: data)
            self.audioPlayer.prepareToPlay()
            self.audioPlayer.play()
    }
}

**Watson Assistant**

The `conversationRequestResponse()` function handles the interaction with the Watson Assistant service. It sends the request (in text) and receives a response from the Watson Assistant service. It then calls the `didReceiveConversationResponse()` function to handle the response text. Finally, it calls the `issueCommand()` function.

```swift
func conversationRequestResponse(_ text: String) {
    let failure = { (error: Error) in print(error) }
    let request = MessageRequest(text: text, context: self.context)
    self.conversation?.message(withWorkspace: Credentials.ConversationWorkspaceID,
                                request: request,
                                failure: failure) {
        response in
            print(response.output.text)
            self.didReceiveConversationResponse(response.output.text)
            self.context = response.context
            // issue command based on intents and entities
            print("appl_action: \\
                  \\
                "(response.context.json["appl_action"]))
            self.issueCommand(intents: response.intents, entities: response.entities)
    }
}
```

The `issueCommand()` function deciphers that intent from the Watson Assistant service and sends the command to the Raspberry Pi through the Watson IoT Platform service.

```swift
func issueCommand(intents: [Intent], entities: [Entity]) {
    for intent in intents {
        print("intent: \(intent.intent), confidence: \(intent.confidence)")
    }
    for entity in entities {
        print("entity: \(entity.entity), value: \(entity.value)")
    }
    for intent in intents {
        if intent.confidence > 0.9 {
            switch intent.intent {
            case "OnLight":
                let command = Command(action: "On", object: "Light", intent: intent.intent)
                sendToDevice(command, subtopic: "light")
            case "OffLight":
                let command = Command(action: "Off", object: "Light", intent: intent.intent)
                sendToDevice(command, subtopic: "light")
            case "TakePicture":
                let command = Command(action: "Take", object: "Picture", intent: intent.intent)
                sendToDevice(command, subtopic: "camera")
                default:
                    print("No such command")
                    return
            }
        }
    }
}
```
Interface with Watson IoT Platform services

This section explains how you set up the interface with the Watson IoT Platform services.

Prepare Xcode project

1. Add the following code to Podfile to install the CocoaMQTT, which uses a protocol that is called MQTT and provides a nice Swift SDK to communicate with the Watson IoT Platform services. Also, add SwiftyJSON to parse and format JSON messages.

   ```swift
   pod 'CocoaMQTT'
   pod 'SwiftyJSON'
   ```

2. Run the command to install CocoaMQTT and SwiftyJSON dependencies.

   ```bash
   pod install
   ```

Initialization

1. Declare a global variable in the AppDelegate.swift file because the MQTT connection must be handled at the application level, rather than the view level. The MQTT client is initialized with the host name, the port, and the client ID.

   ```swift
   let mqttClient = CocoaMQTT(clientID: Credentials.WatsonIOTClientID, host: Credentials.WatsonIOTHost, port: UInt16(Credentials.WatsonIOTPPort))
   ```

   - host name: `[ORG_ID].messaging.internetofthings.ibmcloud.com`
   - port: 1883
   - client ID: `a:[ORG_ID]:[API_KEY]`

2. Create a file called MqttExt.swift to contain the code that interfaces with the Watson IoT Platform service. Declare an extension for ViewController and implement the CocoaMQTTDelegate.

   ```swift
   extension ViewController : CocoaMQTTDelegate {
   }
   ```

3. Initialize the user name with `[API_KEY]`, the password with `[AUTH_TOKEN]`, and set itself as the delegate.

   ```swift
   mqttClient.username = Credentials.WatsonIOTUsername
   mqttClient.password = Credentials.WatsonIOTPPassword
   mqttClient.keepAlive = 60
   mqttClient.delegate = self
   ```

4. Connect when the application becomes active.

   ```swift
   func applicationDidBecomeActive(_ application: UIApplication) {
     mqttClient.connect()
   }
   ```

5. Disconnect when the application enters the background.

   ```swift
   func applicationDidEnterBackground(_ application: UIApplication) {
     mqttClient.disconnect()
   }
   ```

Implement MQTT callbacks

1. Implement the following callback to receive a connection successful event. When connected, call to subscribe to the event topic, which is in the format `iot-2/type/[DEV_TYPE]/id/[DEV_ID]/evt/+fmt/json`, where + is a wildcard.

   ```swift
   mqttClient.onConnect { (result) in
     mqttClient.subscribe("iot-2/type/[DEV_TYPE]/id/[DEV_ID]/evt/+fmt/json")
   }
   ```
func mqtt(_ mqtt: CocoaMQTT, didConnect host: String, port: Int) {
    mqttClient.subscribe(eventTopic)
}

2. Implement the didReceiveMessage() callback function to handle events from the device. If the function receives a status from the camera, download the picture from the Object Storage service. Otherwise, if the function receives a status from the light, append a message to the conversation.

func mqtt(_ mqtt: CocoaMQTT, didReceiveMessage message: CocoaMQTTMessage, id: UInt16) {
    var json : JSON = JSON.null
    if let message = message.string {
        json = JSON.init(parseJSON: message)
    } else {
        return // do nothing
    }
    let cameraTopic = "iot-2/type/\(Credentials.DevType)/id/\(Credentials.DevId)/evt/camera/fmt/json"
    let lightTopic = "iot-2/type/\(Credentials.DevType)/id/\(Credentials.DevId)/evt/light/fmt/json"
    switch message.topic {
    case cameraTopic:
        //ObjectStorage
        if let objectname = json?"d"?"objectname".string {
            if let containername = json?"d"?"containername".string {
                self.downloadPictureFromObjectStorage(containername: containername, objectname: objectname)
            }
        }
    case lightTopic:
        if let status = json?"d"?"status".string {
            switch status {
            case "on":
                self.didReceiveConversationResponse(["Light is on")
            case "off":
                self.didReceiveConversationResponse(["Light is off")
            default:
                break
            }
        }
    default:
        break
    }
}

Send a command to the device

1. Implement this function to send a command to the device. The format of the topic is iot-2/type/[DEV_TYPE]/id/[DEV_ID]/cmd/[light|camera]/fmt/json.

func sendToDevice(_ command: Command, subtopic: String) {
    if let json = command.toJSON() {
        let topic = "iot-2/type/\(Credentials.DevType)/id/\(Credentials.DevId)/cmd/\(subtopic)/fmt/json"
        let message = CocoaMQTTMessage(topic: topic, string: json)
        print("publish message \(json")
        mqttClient.publish(message)
    }
}

Retrieve image from Object Storage services

To retrieve the image from the Object Storage service, you must prepare your Xcode project and then initialize it.

Prepare Xcode project

1. Add the following code to the Podfile to install the BluemixObjectStorage framework.
Build a home assistant mobile application with Watson and IoT
Platform services

pod 'BluemixObjectStorage'

2. Run the command to install BluemixObjectStorage dependencies.
   pod install

Initialization

1. Create a file called ObjectStorageExt.swift to contain the code that interfaces with the Object Storage service and declare an extension for ViewController.
   extension ViewController {
   }

2. Initialize the service with [OS_PROJECTID], [OS_USERNAME], and [OS_PASSWORD], which you saved previously.
   self.objectStorage = ObjectStorage(projectId: Credentials.ObjectStorageProjectId)
   objectStorage.connect(userId: Credentials.ObjectStorageUserId,
   password: Credentials.ObjectStoragePassword, region: ObjectStorage.Region.Dallas) {
   error in
   if let error = error {
   print("objectstorage connect error :: \(error)")
   } else {
   print("objectstorage connect success")
   }
   }

3. Implement the downloadPictureFromObjectStorage() function to download the photo, which is taken and uploaded by the Node-RED application.
   func downloadPictureFromObjectStorage(containername: String, objectname: String) {
   self.objectStorage.retrieve(container: containername) {
   error, container in
   if let error = error {
   print("retrieve container error :: \(error)")
   } else if let container = container {
   container.retrieve(object: objectname) {
   error, object in
   if let error = error {
   print("retrieve object error :: \(error)")
   } else if let object = object {
   print("retrieve object success :: \(object.name)")
   guard let data = object.data else {
   return
   }
   if let image = UIImage(data: data) {
   self.addPicture(image)
   self.didReceiveConversationResponse(["Picture taken"])
   } else {
   print("retrieve object exception")
   }
   } else {
   print("retrieve object exception")
   }
   } else {
   print("retrieve container exception")
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
   }
to integrate a Raspberry Pi as a home gateway to receives commands from and send events to the mobile application. Finally, it shows how to store images by using Object Storage service.

Thanks to Chung Kit Chan for his help in reviewing this tutorial.
Related topics

- IBM Cloud
- Watson Assistant
- Watson Speech to Text
- Watson Text to Speech
- Watson IoT Platform
- Build a voice-activated, serverless home automation hub using natural-language services and OpenWhisk

© Copyright IBM Corporation 2017
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)