Services

v1.26

Sirenia

September 23, 2020

Contents

A service module is like a built-in api module. But unlike api modules, service modules are configurable. For instance the mail service can be configured with the information necessary to connect to mail servers. This means a flow that needs to send or receive emails does not need to contain mail credentials and other sensitive information.

Services are accessed via the `Service` module. getting access to a mail service with the configured key `myMail` is done as follows:

```
var m = Service.get('myMail', Service.MAIL);
```

Now the variable `m` holds a reference to a mail-service ready to use.

Services can usually also be used to set up flow triggers. In the mail service example, flows can be triggered when emails are received from the mail server configured in the mail service.

1 Services

The following services are available:

- Mail
- Serial port

1.1 Mail

The mail-service module has functionality to list and send mails.

1.1.1 Send

Sending a mail requires at least a list of recipients, a subject and a body and has optional support for defining the from address, list of cc-addresses and attachments.

Arguments

In a call like `send(to, subject, body, options)` the arguments are as follows:

- `to` is a list of email-addresses which will receive the message
- `subject` is the subject of the mail
- `body` is the main text of the mail
- `options` is an *optional* object which may contain the following properties
- `from` the sender address (default is "manatee@sirenia.eu")
- `cc` is a list of cc-addresses
- `attachments` is a list of files to attach (each item is a path to a file)

Examples

```
var m = Service.get('myMail', Service.MAIL);
// Simple send
m.send(['jonathan@sirenia.eu'], 'Hello from a bot', 'Manatee says hello').wait(1
// With all options
m.send(
  ['jonathan@sirenia.eu'],
  'Hello from a bot',
  'Manatee says hello',
  {
    'from': 'bot@somewhere.com',
    'cc': ['martin@sirenia.eu', 'los@sirenia.eu', 'lykke@sirenia.eu'],
    'attachments': ['C:/Users/SomeUser/SomeFile.txt']
  }
).wait(10000);
```

1.1.2 List

List is used to list mails in a given mail-box. The return value is a list of mail objects.

Arguments

- `mailbox` is the mailbox to list (default is the `INBOX`)

Examples

```
// List mails in inbox
var inbox =  m.list();
var mailsElsewhere = m.list('someothermailbox');
```

1.1.3 Mail Object

The mail object returned from list has the following properties:

- `id` id of the mail
- `to` list of recipients
- `cc` list of cc-addresses
- `from` sender address
- `subject` subject
- `body` mail text
- `attachments` a list of attachments

And the following methods:

- `delete(mailbox)` to delete the mail from the given mailbox (default is `INBOX`)
- `move(to, from)` to move the mail between two mailboxes
- `writeAttachmentsToDisk` to write the attachments to disk (access them then via the `attachments` property)

1.2 Serial port

The serial port service module lets flows and triggers communicate via serial ports on the local machine.

Data can be sent and received as either binary data or text. Receiving data can be done synchronously or asynchronously as shown in the examples below.

1.2.1 Sending data

Sending data is simple - obtain the service and give it a string or an array of bytes (numbers) to send. Sending is always synchronous (returns after data has been sent):

```
var myDevice = Service.get('my-device', Service.SERIALPORT);
// Send text
myDevice.send('scan barcode, please');
// Send binary data
myDevice.send([0x09, 0x0A, 0x0B, 0x0C]);
```

1.2.2 Receiving data

All methods for receiving data come in synchronous and asynchronous forms such as `receiveOne` and `receiveOneAsync`. They also all accept as their last argument an options object which can have the following properties: - `timeout` is an optional override of how long to wait for data to arrive in milliseconds. The default is 3000. - `binary` is an optional override of the format in which the data is returned. `true` means a byte array is returned, `false` means a text string is returned. Receiving data as text requires the device to encode text by the same encoding configured in the serial port service. The default is `false`.

Request / reply

As a convenience for the common task of sending a request and receiving a reply, two `requestReply` methods are available.

```
var myDevice = Service.get('my-device', Service.SERIALPORT);
// Send and receive text synchronously. Here we override the default receive tim
var barcodeText = myDevice.requestReply('scan barcode, please', { timeout: 10000
// When binary data is sent, the data received is also binary data by default -
var replyBytes = myDevice.requestReply([0x09, 0x0A, 0x0B, 0x0C]);
```

The asynchronous form returns a task object. The task object behaves the same as the tasks used in the Http and Task modules.

```
var task = myDevice.requestReplyAsync('scan barcode, please');
// ... Do other things while we wait for the response...
task.wait();
var barcodeText = task.result;
```

Receive one message

If we expect to receive a data message from the device, we can receive it like so:

```
var messageText = myDevice.receiveOne();
// ... or as undecoded bytes
var messageBytes = myDevice.receiveOne({ binary: true });
```

The asynchronous form offers no surprises:

```
var task = myDevice.receiveOneAsync();
// ... Do other things while we wait for the message...
task.wait();
var messageText = task.result;
```

Receive multiple messages

Sometimes we expect a device to send multiple messages. If we use `receiveOne`, there is a risk that a message arrives between invocations and is lost. To address this situation, we can use `receiveMany`. Its syntax is slightly more complex as we must provide the `receiveMany` method with a callback which will be called for each received message.

The callback must return `true` while more messages are expected. This means when we receive what we know to be the last message, we can return `false` and `receiveMany` will return control to the flow immediately without waiting for the timeout to elapse.

Note that Api methods that show dialogs (for instance `Dialog.input` or `Debug.showDialog`) are not supported within the callback. Use the callback to collect the messages - parsing only enough to determine the return value of the callback.

```
var receivedMessages = [];
// Listen for messages until the default timeout elapses and put them in the arr
myDevice.receiveMany(function(message) {
  receivedMessages.push(message);
  return true;
});

// Listen for messages until the 'BYEBYE' message is received (or until the time
myDevice.receiveMany(function(message) {
  receivedMessages.push(message);
  return message !== 'BYEBYE';
});
```

This method also has an asynchronous form to enable parallel processing:

```
var receivedMessages = [];
// Listen for messages until the default timeout elapses and put them in the arr
var task = myDevice.receiveManyAsync(function(message) {
  receivedMessages.push(message);
  return true;
});
// ... do something else while messages are received ...
task.wait();
// Now we can process the messages in the 'receivedMessages' array.
```

Latest inbound messages

This service module keeps track of the most recent messages received from the device. This can be useful if a flow is triggered by the reception of a message and the flow needs to inspect the messages preceding the triggering message. Note that messages are only added to this collection while a receive operation is active on the serial port. An active serial port trigger will cause messages to be added. A flow with a long running receive operation likewise.

The history is returned as an array of entry objects with the following properties: - `time` is a `Date` object indicating when the message was received - `data` is a string or an array of bytes depending on the optional `binary` option

```
var history = myDevice.getLatestInbound();
if (history.length > 0) {
  var lastEntry = history[history.length - 1];
  // ... do something with the last received entry ...
}

// Get binary data in stead
history = myDevice.getLatestInbound({ binary: true });
```

Open / close port

The methods for sending and receiving data will open the port automatically and close it again when they are done. This means it isn't strictly necessary to excplicitly open and close the port. If for any reason it is undesirable for the port to only be open while it is in use, you can open and close the port excplicitly from your flow:

```
myDevice.open();
// ... communicate with device
myDevice.close();
```

Note that manatee manages the state of the physical serial port intelligently. A serial port trigger and a flow can use the same serial port service at the same time. This means that calling `.open()` and `.close()` may not have a direct effect on the physical port if a trigger is already keeping the port open in order to listen for triggering messages. The `open` and `close` methods merely express intent to use the port for more than one operation. As such they guarantee that the port will not automatically close between separate operations. Excplicitly closing the port isn't a strict requirement as it will happen automatically when the flow has completed.

Byte / string conversions

The serial port service module exposes the means to convert back and forth between byte arrays and their string representation under the encoding configured on the serial port service in cuesta.

```
// myDevice service uses us-ascii
var text = 'abc';
var bytes = [ 0x61, 0x62, 0x63 ]
var decodedText = myDevice.bytesToString(bytes);
var encodedBytes = myDevice.stringToBytes(text)
// text and decodedText are now the same
// bytes and encodedBytes are now the same
```