Writing a Local Skill

Current revision: 4 December 2018
Note: Changes since the October 2018 release of this PDF are indicated by highlighting.

Introduction to Local Skills

There are two basic types of skill architecture for Misty:

- **Remote**. Your code runs on an external device (say, in desktop browser or on a Raspberry Pi) and not onboard the robot.
- **Local**. You upload your code to the robot, and it runs internally on Misty. Local skills can interact with external data (cloud calls, non-Misty API calls, etc).

Important! Misty has supported *remote* skills for some time. ***Local* skills are a pre-release, "alpha" technology and are subject to frequent change**. **Any local skills you create for Misty may need updates before release to reflect these changes in the local skill architecture and implementation.**

Note: Because the language currently used for local skills is JavaScript, but because local skills do *not* run in a browser, it's likely that local skill development will differ from standard browser-based JavaScript development. We don't yet know all this ways this might be the case, so we encourage you to let us know if you find issues and limitations with this implementation.

Local Skill Architecture

The following outline can help guide you through this PDF. Once Misty's Local Skill SDK is public, this document will be posted online.

- **Command Syntax.** Local skill and remote skill commands share much of the same functionality, but they are called differently, and local skills sometimes use additional parameters.
- **Data Handling: Events and Callbacks.** Both stored and live data from the robot are made available to a local skill via callback functions.
    - "Get" Data Callbacks
    - Sensor Event Callbacks
    - Timed or Triggered Event Callbacks
- **Data Handling: Variables.** There are two ways to store persistent data with local skills: as a global variable or as "set" data.
- **Command Type Reference.** There are several command types available to local skills, and there are some usage differences among them.
    - Action Commands
    - Get Commands
    - Event Commands
    - Helper Commands

    - Skill Management Commands
- **File Structure & Code Architecture.** This topic discusses the two required file types for a local skill: a “meta” JSON file and a “code” JavaScript file.
    - Meta File
    - Code File
- **Sending an External Request.** The `misty.SendExternalRequest()` command allows your local skill to use data from the Internet.
- **Loading & Running a Local Skill.** There are currently two options for how you load and run skills.
- **Starting & Stopping a Local Skill.** Currently, local skills running on Misty I must be triggered to start/stop from an external request.
- **Using the Local Skill extension for Visual Studio Code.** We provide a simple extension to help when developing local skills.

COMMAND SYNTAX

You may have already used Misty’s REST API if you’ve created a remote skill for her. When writing a local skill, you use similar JavaScript commands (with a few additions), however the syntax differs.

In a local skill, the syntax for using a commands is:

```
misty.<COMMAND>(parameters);
```

So, for example, to change the color of Misty’s logo LED, you would call the `ChangeLED()` command like this:

```
misty.ChangeLED(255, 0, 0);
```

When used with local skills, most of Misty’s commands also allow you to pass in optional “pre-pause” and “post-pause” values, to add delays before the command is run or after. When we show a command, optional parameters are inside brackets. So the `Drive()` command is represented as:

```
misty.Drive(double linearVelocity, double angularVelocity, [int prePause], [int postPause]);
```

If you use a given optional parameter (such as `postPause`), you must place a value in any preceding optional parameter (such as `prePause`). For example:

```
misty.Drive(50, 10, 0, 1000);
```

An example of using the optional pre- and post-pause parameters with the `Drive()` command might be:

```
misty.Drive(50, 10, 500, 1000);
misty.Stop();
```

In this case, the `Drive()` command would wait 500 milliseconds, start the robot driving at the given linear and angular velocities, then pause execution for 1000 milliseconds (this would pause the skill execution but not the driving itself). After the 1000 millisecond delay, the `Stop()` command would run, bringing the movement of the robot itself to a halt.

DATA HANDLING: EVENTS & CALLBACKS

You typically get two kinds of information from your robot:

(i) Stored data, such as the list of audio files currently saved on the robot. This is the type of data you could obtain with one of Misty's "Get" commands, for example.
(ii) Live sensor "event" data, such as distance information, face detection events, etc.

Both types of data are made available to local skills via callbacks, so **you must implement callback methods to be informed when data is ready**.

The syntax and usage of callback commands for these two types of data varies. Additionally, there is a third category of callbacks you can create for Misty: timed or triggered callbacks. Their usage and syntax are described at the end of this section.

"Get" Data Callbacks

For "get" callback functions, the callback name is by default set to be `_<COMMAND>` (you can use your own callback name if desired). So, for example, the default callback function for `GetDeviceInformation` would be `_GetDeviceInformation`.

A "get" callback function must have exactly one parameter. That parameter holds the data returned through the callback. Note that a local skill callback returns data as an `object`, not as a JSON `string` value. An example use of a callback to obtain an audio list and play a random sound is as follows:

```
StartMySkill();
function StartMySkill() {
	misty.GetListOfAudioClips();
}

function _GetListOfAudioClips(callbackData) {
		var audioList = callbackData.Result;
		if (audioList.length > 0) {
			var randomInt = Math.floor(Math.random() * audioList.length);
			var currentSound = audioList[randomInt].Name;
			misty.PlayAudioClip(currentSound);
		}
	}

}
```

“Get” callbacks may be set up with callback rules and a skill to trigger for the callback instead of calling back into the same skill. The available callback rules are `Synchronous`, `Override`, and `Abort`.

- `Synchronous` tells the system to run the new callback thread and to continue running any other threads the skill has started.
- `Override` tells the system to run the new callback thread but to stop running commands on any other threads, including the thread the callback was called within. The system only runs the thread the callback was triggered in, once the callback comes back.
- `Abort` tells the system to ignore the new callback thread if the skill is still doing work on any other threads (including the original thread the callback was called within). For “get” callbacks, using `abort` in this case would mean that the data requested would not be received.

Sample “get” callback with a callback rule:

```
misty.GetListOfAudioClips("synchronous", 500, 1000);
```

Sample “get” callback with a callback rule and including the ID of a skill to trigger:

```
misty.SlamGetMap("override", "9d50efbd-af53-4cd3-9659-c0faf648263d", 500, 10);
```

Sensor Event Callbacks

Changes from October 2018 local skill release:

- If you do not add a property test when you register for an event, the system returns all data relevant to the event.

For event callback functions, you set an event name (`eventName`) of your choice at the time you register for the event using the `RegisterEvent()` function. The name of the callback function name is set automatically to be the same as your event name, prefixed with an underscore. The `messageType` value is whatever the predefined `Type` property value is for the data stream as listed here.

The `RegisterEvent()` function has the following form:

```
misty.RegisterEvent(string eventName, string messageType, int debounce, [bool
keepAlive], [string callbackRule], [string skillToCall]);
```

So at its most simple, registering to receive callback data for a `SelfState` event might look like:

```
misty.RegisterEvent("UpdatedAwareness", "SelfState", 1000, true);
```

Note that the `RegisterEvent()` command may *optionally* be set up with callback rules and a skill to trigger for the callback instead of calling back into the same skill. The available callback rules are `Synchronous`, `Override`, and `Abort`.

- `Synchronous` tells the system to run the new callback thread and to continue running any other threads the skill has started.

- `Override` tells the system to run the new callback thread but to stop running commands on any other threads, including the thread the callback was called within. The system only runs the thread the callback was triggered in, once the callback comes back.
- `Abort` tells the system to ignore the new callback thread if the skill is still doing work on the original thread the callback was called within. For event callbacks, the new callback is ignored until all current processes are finished running, then the event is processed the next time it is sent.

Before registering an event callback, you can use the `AddPropertyTest()` command to create one or more property comparison tests to specify the event data that the system sends:

```
misty.AddPropertyTest(string eventName, string property, string inequality, string
valueAsString, string valueType);
```

Note: If you do not specify a property test, the system returns the full data object for the event.

You can also use the `AddReturnProperty()` command to add any additional return property fields you may need:

```
misty.AddReturnProperty(string eventName, string eventProperty);
```

An event callback function must have one parameter to store the data returned through the callback, so an example of an event callback might be:

```
Function _UpdatedAwareness(callbackData) {
      //do work with data
}
```

Putting these together, an example of registering events with property comparison tests (based on the direction of Misty's movement) might look like this:

```
function RegisterEvents(goingForward) {

    if (goingForward) {
        misty.UnregisterEvent("BackTOF");
        misty.AddPropertyTest("FrontTOF", "SensorPosition", "!==", "Back", "string");
        misty.AddPropertyTest("FrontTOF", "DistanceInMeters", "<=", 0.6, "double");
        misty.RegisterEvent("FrontTOF", "TimeOfFlight", 100, false);
    }
    else {
        misty.UnregisterEvent("FrontTOF");
        misty.AddPropertyTest("BackTOF", "SensorPosition", "==", "Back", "string");
        misty.AddPropertyTest("BackTOF", "DistanceInMeters", "<=", 0.6, "double");
        misty.RegisterEvent("BackTOF", "TimeOfFlight", 100, false);
    }
}
```

Finally, when an event callback is triggered, note that **by default it unregisters the callback** to prevent more commands from overriding the initial call, which can become an issue with fast-triggering events. To handle this, you have two choices:

- You can re-register the event in the event callback function.
- To keep the event alive and not unregister on a callback, you can pass `true` for the `keepAlive` parameter when you register the event.

Timed or Triggered Event Callbacks

Using the `RegisterTimerEvent()` function, you can create an event that sends a callback after a certain period of time:

```
misty.RegisterTimerEvent(string eventName, int callbackTimeInMs, bool keepAlive);
```

By default, that event is triggered once and removed, but you can choose to have it call back until unregistered. To do this, you can specify `true` for the `keepAlive` parameter when registering for the timer event. This causes the event to automatically reset when the callback is triggered.

For example, you can set the `callbackTimeInMs` parameter to 5 seconds and specify `keepAlive` to be `true`. Then, after the callback is triggered, the timer resets and the callback will be called again every 5 seconds until the timer event is unregistered with `UnregisterEvent` or is automatically unregistered when the skill ends.

You can also create a triggered event to call back to the skill when a specific command is called. That event is triggered once and removed, but you can immediately re-register as needed in the callback:

```
misty.RegisterUserEvent(string eventName, string callbackMethod);
```

You can also trigger an event by making a REST call to the `event` endpoint with a `POST` command:

```
POST
api/alpha/sdk/skills/event
```

With a JSON body similar to:

```
{
      "UniqueId" : "b307c917-beb8-47e8-9bbf-1c57e8cd4d4b",
      "EventName": "UserEventName",
      "Payload": { "mydata": "two" }
 }
```

The `UniqueId` and `EventName` values are required and must match the ID of the skill to call and the event name you used in that skill. You should place any payload data you wish to send to the skill in the `Payload` field.

DATA HANDLING: VARIABLES

There are two ways to store persistent data with local skills:

- In a global variable, where the data is available (but not updated) across threads in a single skill
- As "set" data, where the data is available (and updated) across threads in a single skill and is shareable among skills

Important! There is no capability at this time to store variable data such that it persists **across a reboot** of the robot.

You can create global variables and use them across all "get" and "event" callbacks within a single skill. Global variables are copied over to new threads as they are created from callbacks. Global variables must be declared at the top of a skill, are prefixed with an underscore, and are **not** declared as `var`, `const`, etc.

Note that the value of a global variable is only preserved going forward. That is, if you have a thread running that spawns a new thread (via a "get" or "event" callback) but then continues to process, the global value will not update for the original thread; only the child thread will update that value going forward.

In this example, `_imageCount` is declared and used as a global variable:

```
_imageCount = 72;

StartSkill();

function StartSkill() {
      misty.GetListOfImages();
}

function _GetListOfImages(response) {
      if(response.Result.length != _imageCount) {
            misty.Debug("Wrong number of expected images!");
            misty.PlayAudioClip("SadSound.wav", 50);
      }
}
```

In cases where you need persistent data that can be (a) validly updated across threads or (b) shared between skills, you need to use the cross-skill `Set()` command:

```
misty.Set(string key, string value);
```

Data saved using `Set()` must be one of these types: `string`, `bool`, `int`, or `double`. Alternately, you can serialize your data into a string using `JSON.stringify()` and parse it out again using `JSON.parse()`.

Additional commands that operate on data across skills are described in the "Helper Commands" section below.

COMMAND TYPE REFERENCE

The following lists of commands include those that are specific to local skills, as well as those for which a version is also available to remote skills. Each section also notes any differences in usage for a specific type of command.

Descriptions of those commands that are **only** available for local skills are in the sections below. For descriptions of those commands for which similar versions are available to **both** local and remote skills, see the documentation here.

Action Commands

Changes from October 2018 local skill release:

- The `Halt()` command has been added and can be used to stop all of Misty's motor controllers, including drive motor, head/neck, and arm.
- `SlamStartStreaming()` and `SlamStopStreaming()` can now be used to open and close the data stream from the Occipital Structure Core depth sensor, so you can obtain image and depth data independently from mapping or tracking activities.
- The `TakePicture()` and `SaveImageAssetToRobot()` commands can now make images smaller. To reduce the size of an image you must supply values for both the `Width` and `Height` arguments. Note that if you supply disproportionate values for `Width` and `Height`, the system uses the proportionately smaller of the two values to resize the image.

Action commands tell the robot to do something, but do not return data, so they do not require you to implement a callback.

```
misty.CancelFaceTraining([int prePause], [int postPause]);
misty.ChangeDisplayImage(string fileName, [double timeoutInSeconds], [double alpha],
[int prePause], [int postPause]);
misty.ChangeLED(int red, int green, int blue, [int prePause], [int postPause]);
misty.ClearDisplayText ([int prePause], [int postPause]);
misty.ClearLearnedFaces([int prePause], [int postPause]);
misty.DeleteAudioAssetFromRobot(string filename, [int prePause], [int postPause]);
misty.DeleteImageAssetFromRobot(string filename, [int prePause], [int postPause]);
misty.Drive(double linearVelocity, double angularVelocity, [int prePause], [int
postPause]);
misty.DriveTime(double linearVelocity, double angularVelocity, int timeInMs, [double
degree], [int prePause], [int postPause]);
misty.FollowPath(string pathString, [int prePause], [int postPause]);
misty.Halt([int prePause], [int postPause]);
misty.LocomotionTrack(double leftTrackSpeed, double rightTrackSpeed, [int prePause],
[int postPause]);
misty.MoveHeadDegrees(double pitch, double roll, double yaw, [double velocity], [int
prePause], [int postPause]);
misty.MoveHeadPosition(double pitch, double roll, double yaw, [double velocity], [int
prePause], [int postPause]);
misty.MoveHeadRadians(double pitch, double roll, double yaw, [double velocity], [int
prePause], [int postPause]);
misty.PlayAudioClip(string fileName, [int prePause], [int postPause]);
```

```
misty.SaveAudioAssetToRobot(string filename, string dataAsByteArrayString, bool immediatelyApply = false, bool OverwriteExisting = false,  [int prePause], [int postPause]);
misty.SaveImageAssetToRobot(string filename, string dataAsByteArrayString, int width, int height, [bool immediatelyApply = false], [bool OverwriteExisting = false],  [int prePause], [int postPause]);
misty.SetDefaultVolume(int volume, [int prePause], [int postPause]);
misty.SetHeadDegrees(string axis, double position, [double velocity,] [int prePause], [int postPause]);
misty.SetHeadPosition(string axis, double position, double velocity, [int prePause], [int postPause]);
misty.SetHeadRadians(string axis, double position, [double velocity,] [int prePause], [int postPause]);
misty.SetLogLevel(string level,[int prePause], [int postPause]);
misty.SlamReset([int prePause], [int postPause]);
misty.SlamStartMapping([int prePause], [int postPause]);
misty.SlamStartStreaming([int prePause], [int postPause]);
misty.SlamStartTracking([int prePause], [int postPause]);
misty.SlamStopMapping([int prePause], [int postPause]);
misty.SlamStopStreaming([int prePause], [int postPause]);
misty.SlamStopTracking([int prePause], [int postPause]);
misty.StartFaceDetection([int prePause], [int postPause]);
misty.StartFaceRecognition([int prePause], [int postPause]);
misty.StartFaceTraining(string faceId, [int prePause], [int postPause]);
misty.StartRecordingAudio(string filename, [int prePause], [int postPause]);
misty.Stop([int prePause], [int postPause]);
misty.StopFaceDetection([int prePause], [int postPause]);
misty.StopFaceRecognition([int prePause], [int postPause]);
misty.StopRecordingAudio([string callbackRule = "synchronous"], [string skillToCallUniqueId],[int prePause], [int postPause]);
misty.TakePicture([bool base64], [string fileName], [int width], [int height], [bool DisplayOnScreen], [bool OverwiteExisting], [string callbackRule = "synchronous"], [string skillToCallUniqueId], [int prePause], [int postPause]);
```

Get Commands

Changes from October 2018 local skill release:

- The `GetAudioFile()` command has been added and can be used to obtain an audio file saved on your robot.

Get commands obtain data from the robot, so they require you to implement a callback to be notified when they return. The callback should contain exactly one parameter, to hold the data being returned. See the "Get Data Callbacks" section above for more usage details.

```
misty.GetAvailableWifiNetworks([string callbackRule = "synchronous"], [string skillToCallUniqueId], [int prePause], [int postPause]);
misty.GetAudioFile(string fileName, [string callbackRule = "synchronous"], [string skillToCallUniqueId], [int prePause], [int postPause])
misty.GetBatteryLevel([string callbackRule = "synchronous"], [string skillToCallUniqueId], [int prePause], [int postPause]);
misty.GetBetaHelp([string endpointName],[string callbackRule = "synchronous"], [string skillToCallUniqueId], [int prePause], [int postPause]);
misty.GetCameraData([string callbackRule = "synchronous"], [string skillToCallUniqueId], [int prePause], [int postPause]);
```

```
misty.GetDeviceInformation([string callbackRule = "synchronous"], [string
skillToCallUniqueId], [int prePause], [int postPause]);
misty.GetHelp([string endpointName],[string callbackRule = "synchronous"], [string
skillToCallUniqueId], [int prePause], [int postPause]);
misty.GetImage(string imageName, [string callbackRule = "synchronous"], [string
skillToCallUniqueId], [int prePause], [int postPause]);
misty.GetLearnedFaces([string callbackRule = "synchronous"], [string
skillToCallUniqueId], [int prePause], [int postPause]);
misty.GetListOfAudioClips([string callbackRule = "synchronous"], [string
skillToCallUniqueId], [int prePause], [int postPause]);
misty.GetListOfAudioFiles([string callbackRule = "synchronous"], [string
skillToCallUniqueId], [int prePause], [int postPause]);
misty.GetListOfImages([string callbackRule = "synchronous"], [string
skillToCallUniqueId], [int prePause], [int postPause]);
misty.GetLogFile(string date, [string callbackRule = "synchronous"], [string
skillToCallUniqueId], [int prePause], [int postPause]);
misty.GetLogLevel([string callbackRule = "synchronous"], [string
skillToCallUniqueId], [int prePause], [int postPause]);
misty.GetWebsocketHelp(string websocketClass,[string differentCallbackName], [string
callbackRule = "synchronous"], [string skillToCallUniqueId], [int prePause], [int
postPause]);
misty.SlamGetDepthImage([string callbackRule = "synchronous"], [string
skillToCallUniqueId], [int prePause], [int postPause]);
misty.SlamGetMap([string callbackRule = "synchronous"], [string skillToCallUniqueId],
[int prePause], [int postPause]);
misty.SlamGetPath([string callbackRule = "synchronous"], [string
skillToCallUniqueId], [int prePause], [int postPause]);
misty.SlamGetStatus([string callbackRule = "synchronous"], [string
skillToCallUniqueId], [int prePause], [int postPause]);
misty.SlamGetVisibleImage([bool base64], [string callbackRule = "synchronous"],
[string skillToCallUniqueId], [int prePause], [int postPause]);
```

Event Commands

Event commands allow you to be notified when sensor data that meets your criteria is available from Misty. See the "Sensor Event Callbacks" section above for more usage details.

```
misty.RegisterEvent(string eventName, string messageType, int debounce, [bool
keepAlive = false], [string callbackRule = "synchronous"], [string
skillToCallUniqueId]);
misty.UnregisterEvent(string eventName);
misty.AddPropertyTest(string eventName, string property, string inequality, string
valueAsString, string valueType);
misty.AddReturnProperty(string eventName, string eventProperty);
```

You can create a timed event, to call back to the skill after a certain period of time.

```
misty.RegisterTimerEvent(string eventName, int callbackTimeInMs, bool
keepAlive=false);
```

You can also create a triggered event to call back to the skill when an specific command is called. See the "Timed or Triggered Event Callbacks" section above for more details on using user-defined event callbacks.

```
misty.RegisterUserEvent(string eventName, bool keepAlive, string callbackRule, string skillToCall)
```

Helper Commands

The system supplies the following commands to assist you in creating a local skill for Misty.

Saving Persistent Data

To create data that persists across skills, you must use one of the following helper commands to save (set) that data to the robot or to get that data from the robot:

```
misty.Set(string key, string value);
misty.Get(string key);
misty.Remove(string key);
misty.Keys();
```

Important! Data stored via the `Set()` command **does not persist across a reboot** of the robot at this time.

Persistent data must be saved as one of these types: `string`, `bool`, `int`, or `double`. Alternately, you can serialize your data into a string using `JSON.stringify()` and parse it out again using `JSON.parse()`.

Currently, any data saved to the robot this way is **not** automatically deleted and by default may be used across multiple skills.

Note that calling `Keys()` provides a list of all the available persistent data stored on the robot.

Accessing External Data

Another type of "helper" command allows your local skill to use data from the Internet. The `misty.SendExternalRequest()` command takes 6 required parameters along with two optional timing parameters at the end. See the section "Sending an External Request" below for an example of using this functionality.

```
misty.SendExternalRequest(string method, string host, string resource, string authorization, string token, JSON args, [int prePause], [int postPause]);
```

Pausing and Debugging

Changes from October 2018 local skill release:

- New `CancelSkill()` command allows you to cancel a specified skill.
- New `Publish()` command writes a debug message to the log, even when the value of `WriteToLog` is set to `False` in the meta file.

There are a few additional helper commands you can use to help with your local skill:

```
misty.Pause(int delay);
misty.RandomPause(int minimumDelay, int maimumDelay)
misty.Debug(string debugInfo);
misty.CancelSkill(string skillName)
misty.Publish(string debugInfo)
```

Note: A local skill must have `BroadcastMode` set to `Verbose`, `Debug`, or `All` in the meta file for debug statements to be broadcast. By default, debug statements are set to `Off`.

Skill Management Commands

Changes from October 2018 local skill release:

- The `RunSkill` command now accepts either the `UniqueID` or the `Name` of a skill for the `Skill` parameter.

The following REST commands are intended to be used from a remote device to control local skills on the robot.

SaveSkillToRobot

Uploads a skill to the robot and makes it immediately available for the robot to run.

Endpoint: POST{robot-ip-address}/api/alpha/sdk/skill/deploy

```
{
"Skill" : "SkillName",
"ImmediatelyApply": false,
"OverwriteExisting": true
}
```

Parameters

- Skill (byte array) - A zipped file containing the two skill files. Both these files (one JSON meta file and one JavaScript code file) should have the same name. For more details, see the “File Structure & Code Architecture” section below.
- ImmediatelyApply (boolean) - True or false. Specifies whether Misty immediately runs the uploaded skill.
- OverwriteExisting (boolean) - True or false. Indicates whether the file should overwrite a file with the same name, if one currently exists on Misty .

LoadSkill

Makes a previously uploaded skill available for the robot to run and updates the skill for any changes that have been made.

Endpoint: POST{robot-ip-address}/api/alpha/sdk/skills/load

```
{
```

```
	"Skill": "SkillName"
}
```

Parameters

- Skill (string) - The name of the skill to load.

ReloadSkills

Makes all previously uploaded skills available for the robot to run and updates any skills that have been edited. **Note:** The `ReloadSkills` command runs immediately, but there may be a significant delay after the call completes before all skills are fully loaded onto the robot if there are many to load.

Endpoint: POST{robot-ip-address}/api/sdk/reload

Parameters

- (None)

Return Values

- Result (array) - An array containing error strings for any errors related to this command.

RunSkill

Immediately runs a previously uploaded skill.

Endpoint: POST{robot-ip-address}/api/alpha/sdk/skill

```
{
	"Skill": "SkillName"
}
```

Parameters

- Skill (string) - As specified with the `Name` value in the skill's meta file, the name of the skill to run. You can also pass the `UniqueID` for a skill.
- Method (string) - Optional. A specific method within a skill to run, which can be useful for testing. If no value is specified for the `Method` parameter, `RunSkill()` by default starts running the skill from the beginning.

GetSkills

Obtains a list of the skills currently uploaded onto the robot.

Endpoint: GET{robot-ip-address}/api/alpha/sdk/skills

Parameters

- (None)

Return Values

- Result (array) - An array containing the names of the uploaded skills on the robot.

CancelSkill
Stops a specified running skill (or all running skills if no name is specified).

Endpoint: POST{robot-ip-address}/api/alpha/sdk/skills/cancel

```
{
	"Skill": "SkillName"
}
```

Parameters
- Skill (string) - As specified with the `Name` value in the skill's meta file, the name of the skill to run. Use an empty payload to cancel all running skills.

TriggerSkillEvent
Triggers an event within a skill. The skill must be running already for Misty to trigger the event within the skill.

Endpoint: POST{robot-ip-address}/api/alpha/sdk/skills/event

```
{
"UniqueId" : "b307c917-beb8-47e8-9bbf-1c57e8cd4d4b",
"EventName": "UserEvent",
"Payload": { "test": "two" }
}
```

Parameters
- UniqueId (string) - As specified in the skill's JSON meta file, the 128-bit GUID for the skill that holds the event to trigger.
- EventName (string) - The name of the event to trigger.
- Payload (JSON string) - Any arguments needed for the event.

UnloadSkill
Makes a skill unavailable to be run which is currently onboard the robot, but does not remove the skill from the robot's memory.

Endpoint: POST{robot-ip-address}/api/alpha/sdk/skills/unload

```
{
	"Skill": "SkillName"
}
```

Parameters
- Skill (string) - The name of the skill to unload.

UnloadAllSkills
Makes all skills onboard the robot unavailable to be run, but does not remove the skills from the robot's memory.

Endpoint: POST{robot-ip-address}/api/alpha/sdk/skills/unloadall

Parameters

- (None)

FILE STRUCTURE & CODE ARCHITECTURE

There are two basic file types required for a local skill: a “meta” JSON file and a “code” JavaScript file. On the robot, these files are located in the following directory structure:

```
User Folders\Music\SDKAssets\Misty\Skills\Meta\[filename.json]
User Folders\Music\SDKAssets\Misty\Skills\Code\[filename.js]
```

Each skill MUST have files of the same name in both the `Code` and `Meta` folders. For example:

```
User Folders\Music\SDKAssets\Misty\Skills\Meta\Wander.json
User Folders\Music\SDKAssets\Misty\Skills\Code\Wander.js
```

Meta File

Changes from initial local skill release:

- The JSON key/value pairs used in the meta file have changed.

Every local skill must include a named meta file with a `.json` extension. This brief `meta` file enables Misty to understand what your skill does and how to execute your code. For example:

```
{
      "Name": "sample_skill",
      "UniqueId" : "f34a3aa0-8341-4047-8b54-59d658620ecf",
      "Description": "My skill is amazing!",
      "StartupRules": ["Manual", "Robot"],
      "Language": "javascript",
      "BroadcastMode": "verbose",
      "TimeoutInSeconds": 300,
      "CleanupOnCancel": false,
      "WriteToLog": false,
      "Parameters": {
            "int":10,
            "double":20.5,
            "String":"twenty"
            "foo": "bar"
      }
}
```

The value for `UniqueId` should be a 128-bit GUID, and `Name` can be any value you want. To get up and running quickly with your own skill, you can duplicate the values we’ve provided and simply generate a new GUID for the `UniqueId` value.

Note that the `WriteToLog` value is optional, and that example meta files may include additional key/value pairs that are not currently in active use and may change in the future.

You can use the `Parameters` value(s) in the meta file to define any optional default parameters for the skill. You can then access these values in your code file via the global `_params` variable.

So, in the example below from the code file for a skill, we could create a global variable named `_global` that would hold the value `"bar"`, which was set in the meta sample above:

```
_global = _params.foo;
```

Code File

The `.js` code file contains the running code for your local skill. A valid JavaScript `code` file can be even simpler than a corresponding JSON `meta` file. Here's an example of a complete, very simple `code` file for a local skill:

```
misty.Debug("Hello, World!");
misty.ChangeLED(0, 255, 0);
```

Most local skills include callback functions and registering and unregistering of events. That's because most local skills interact with data in some way. For example, when sensor data is sent from Misty to your code, it triggers an event, and this in turn executes the callback function for the registered event.

In the sample skill code file below, an event callback function is used to handle sensor data from one of Misty's time-of-flight sensors:

```
// Debug message to indicate the skill has started
misty.Debug("Starting my skill");

// Issue POST commands to change LED and start DriveTime
misty.ChangeLED(0, 255, 0); // green, GO!
misty.DriveTime(50, 0, 10000);
j
// Register for TOF and add property tests
misty.AddPropertyTest("FrontTOF", "SensorPosition", "!==", "Back", "string");
misty.AddPropertyTest("FrontTOF", "DistanceInMeters", "<=", 0.2, "double");
misty.RegisterEvent("FrontTOF", "TimeOfFlight", 100, true);

// Define the TOF callback
function _FrontTOF(data) {
      // Grab property test results
      let frontTOF = data.PropertyTestResults[0].PropertyParent;

      // Log distance object was detected and sensor
      misty.Debug(JSON.stringify(frontTOF.DistanceInMeters));
```

```
	misty.Debug(JSON.stringify(frontTOF.SensorPosition));

	misty.Stop();
	misty.ChangeLED(255, 0, 0);
	misty.Debug("Ending my skill")
}
```

Note that when a skill starts, the code within the skill automatically starts running. When a skill has finished executing (or has been cancelled), normal cleanup automatically begins. Normal cleanup drops any pending callbacks, deletes cached code, etc.

If `CleanupOnCancel` is set to `true` in the `meta` file, then when a skill is cancelled, additional commands are automatically issued to stop running processes that may have been started in the skill. These process might include facial detection / recognition / training, SLAM mapping / tracking / recording / streaming, and record audio. If `CleanupOnCancel` is set to `false`, then this additional cleanup does not occur when cancelled (`false` is currently the default value). Currently, this does **not** affect the behavior of the skill if it ends normally. These commands are **not** automatically issued in this case.

SENDING AN EXTERNAL REQUEST

Changes from initial local skill release:

- `SendExternalRequest()` can now return audio and image file data.

Even though your skill is running locally on Misty, it can still access external data from the internet and deliver it back again. This sample skill fetches the current temperature of a designated city, then sends it back through a debug message. To do this, you'll need to use the `SendExternalRequest()` command to send a `GET` request to the APIXU API to obtain the data.

You can set the designated city as a parameter in the JSON meta file:

```
{
	"Name": "local_helloworld4",
	"UniqueId": "523c7187-706e-4313-a657-0fa11d8bbdd4",
	"Description": "Local 'Hello, World!' tutorial series, part 4.",
	"StartupRules": [ "Manual", "Robot" ],
	"Language": "javascript",
	"BroadcastMode": "verbose",
	"TimeoutInSeconds": 300,
	"CleanupOnCancel": false,
	"WriteToLog": false,
	"Parameters": {
		"city": "Denver"
	}
}
```

The code file is simple, focusing on the use of `SendExternalRequest()`. As a reminder, here's the `SendExternalRequest()` prototype:

```
misty.SendExternalRequest(string method, string host, string resource, string
authorization, string token, JSON args, [int prePause], [int postPause]);
```

In this example we're sending a GET request, so we'll use the string `GET` for the first (`method`) parameter.

The second parameter (`host`) should contain the URL for the serving hosting the external API that you want to use. In this example that is `http://api/apixu.com`

The third parameter (`resource`) should contain the rest of the URL required for the endpoint, which in the case of our example is a bit complex. We'll designate the city we want to get weather data for using a template literal, passing the value `_params.city` into the URL string. Using `_params` allows us to use the `Parameters` value from the meta file. In the meta file, we specified the key `city` to hold the city we want to use in the search. Therefore, here in the code file, `_params.city` holds the string `Denver`. In this example with APIXU, the URL will also typically include an API key (which can easily be obtained through APIXU by registering for an account). If you're curious, for further information on constructing the query necessary for this example, see APIXU's documentation here.

For some requests additional authorization may be necessary. This is where the fourth (`authorization`) and fifth (`token`) parameters come in to play. However, in the case of APIXu, the API key provided as part of `ResourceURL` is sufficient so we can enter `null` for the fourth and fifth parameters.

The sixth required parameter (`args`) holds any data you are including in your request. In this case, this can also be set to `null` as this example is a GET request. **Note:** When you have no data to send, some services may require an empty JSON payload instead of `null`. When this is the case, the service returns an error message indicating the value of `args` cannot be `null` or empty.

As with other local skill commands, `prePause` and `postPause` are optional.

The final form of this example command is:

```
misty.SendExternalRequest("GET", "http://api.apixu.com",
`/v1/current.json?key={{APIKEY}}&q=${_params.city}`, null, null, null);
```

Once Misty receives the data back from APIXU, the callback -- which is automatically set to `_SendExternalRequest` -- will run:

```
function _SendExternalRequest(data) {}
```

Once the data comes back from the request, we parse the data to find the searched-for city name and the current temperature of that city. We can assign those to variables as shown below:

```
let currentCity = data.Result.location.name;
let currentTemp = data.Result.current.temp_f;
```

The final step is to have Misty send us the data back through a debug message:

```
misty.Debug(`The current temperature of ${currentCity} is ${currentTemp} degrees
fahrenheit.`);
```

The complete `.js` file is below for reference:

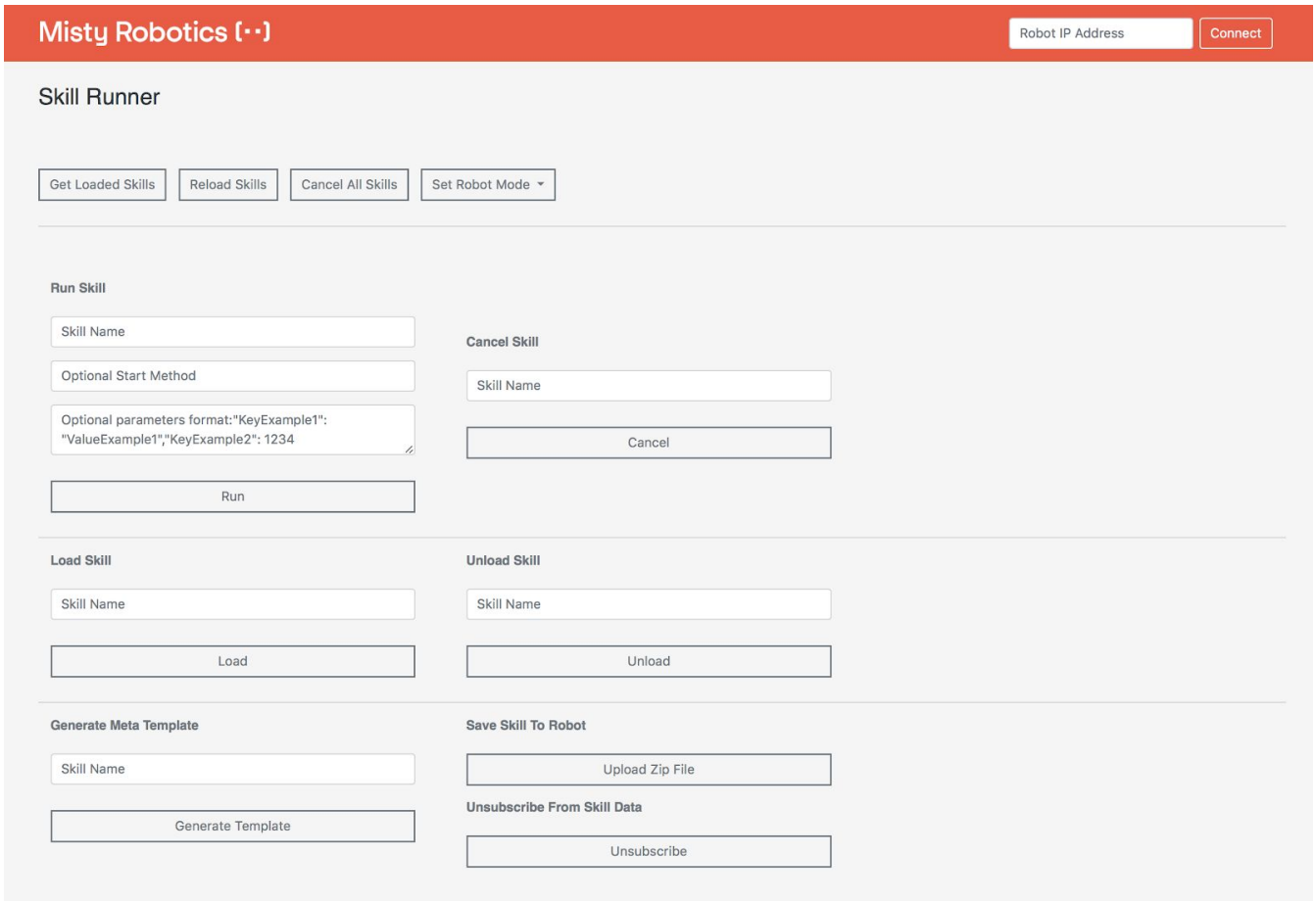
```
// Debug message to indicate the skill has started
misty.Debug("starting weather report skill");

// Send GET request to weather API
misty.SendExternalRequest("GET", "http://api.apixu.com",
`/v1/current.json?key=e1642c939804479ab3c213052182509&q=${_params.city}`, null, null,
null);

// Callback for external data
function _SendExternalRequest(data) {

      // Assign variables to grab the city name and current temperature
    let currentCity = JSON.stringify(data.Result.location.name);
    let currentTemp = JSON.stringify(data.Result.current.temp_f);

      // Log message to display data to the user
    misty.Debug(`The current temperature of ${currentCity} is ${currentTemp} degrees
fahrenheit.`);
}
```

Loading & Running a Local Skill

Once you've created the files for your skill, you must load them onto your robot before you can run them. The two methods for loading skills onto Misty are:

- the Skill Runner tool (available as a ZIP archive; click `SkillRunner.html` to open), which provides a simple upload feature
- a REST tool such as Postman that can send a `POST` request to a dedicated endpoint for skill deployment

Using Skill Runner



1. Compress and save your skill's `Meta` and `Code` files into a .zip file with the same name as your skill.
2. Open SkillRunner.html and connect to Misty using your robot's IP address.
3. Open up your browser's JavaScript console for the Skiller Runner page, so you can see what's happening.
4. Select **Upload Zip File** under "Save Skill to Robot."
5. Select the .zip file you just created. Observe the JavaScript console for a success message confirming that the upload request was received.
6. Once the file has been uploaded to Misty, click **Reload Skills** at the top of the page. This ensures that your robot and latest code changes are in sync. Observe the JavaScript console for a log message verifying the skills have been loaded.
7. To run your skill, enter the skill's name under "Run Skill" and click **Run**. Continue observing the console; as events are triggered, you'll see debug messages in the console.

Note: You can generate useable `meta` file content with the **Generate Meta Template** controls in Skill Runner.

Using Postman

There are many ways to send a `POST` request to the skill deployment endpoint, but here we'll use Postman.

1. Compress and save your skill's `Meta` and `Code` files into a .zip file with the same name as your skill.
2. To attach your skill .zip to the request, first navigate to the Headers section in Postman.
3. For the header key, enter "Content-Type".
4. For the header value, enter "multipart/form-data".
5. In the body section confirm that "form-data" is selected at the top.
6. For the body key, enter "skills", then select "File" from the dropdown menu on the right.
7. In the body value section, click "Choose Files" and select the .zip file for your skill.
8. To add and load a skill onto the robot, send a `POST` request to `http://{your robot's ip address}/api/alpha/sdk/skill/deploy` with the following parameters:
    - `Skill` (byte array) - A zipped file containing the two skill files (`Meta` and `Code`).
    - `ImmediatelyApply` (boolean) - `True` or `False`. Specifies whether the robot immediately runs the uploaded skill.
    - `OverwriteExisting` (boolean) - `True` or `False`. Indicates whether the file should overwrite a file with the same name, if one currently exists on this robot.
9. Look at the response to confirm the request was successful.
10. Open SkillRunner.html and connect to Misty using your robot's IP address.
11. Open up your browser's JavaScript console for the Skiller Runner page, so you can see what's happening.
12. Click "Reload Skills" at the top of the page. This ensures that your robot and latest code changes are in sync. Observe the JavaScript console for a log message verifying the skills have been loaded.
13. To run your skill, enter the skill's name under "Run Skill" and click "Run." Continue observing the console; as events are triggered, you'll see debug messages in the console.

Starting & Stopping a Local Skill

Changes from initial local skill release:

- A default timeout of 5 minutes was added. After 5 minutes the skill will cancel, even if it is performing actions or waiting on callbacks. This duration can be changed by providing a different `TimeoutInSeconds` value in the meta file. In addition, if a skill is not performing any actions nor waiting on any commands, it will automatically cancel after 5 seconds.

Currently, local skills running on Misty I must be triggered to start/stop from an external command. This command can be sent via a REST client or as a JavaScript AJAX request.

To start a skill, send the `RunSkill POST` command with the following syntax. Note that the `Skill` value must be the same as the `Name` value for the skill in its JSON `meta` file.

```
http://{robot-ip-address}/api/alpha/sdk/skill
{
      "Skill": "SkillName"
}
```

For example:

```
http://{robot-ip-address}/api/alpha/sdk/skill
{
	"Skill": "Wander"
}
```

To stop a skill, send the `CancelSkill POST` command to the following endpoint, again using the `Name` value for the skill from its JSON `meta` file:

```
http://{robot-ip-address}/api/alpha/sdk/cancel
{
	"Skill": "Wander"
}
```

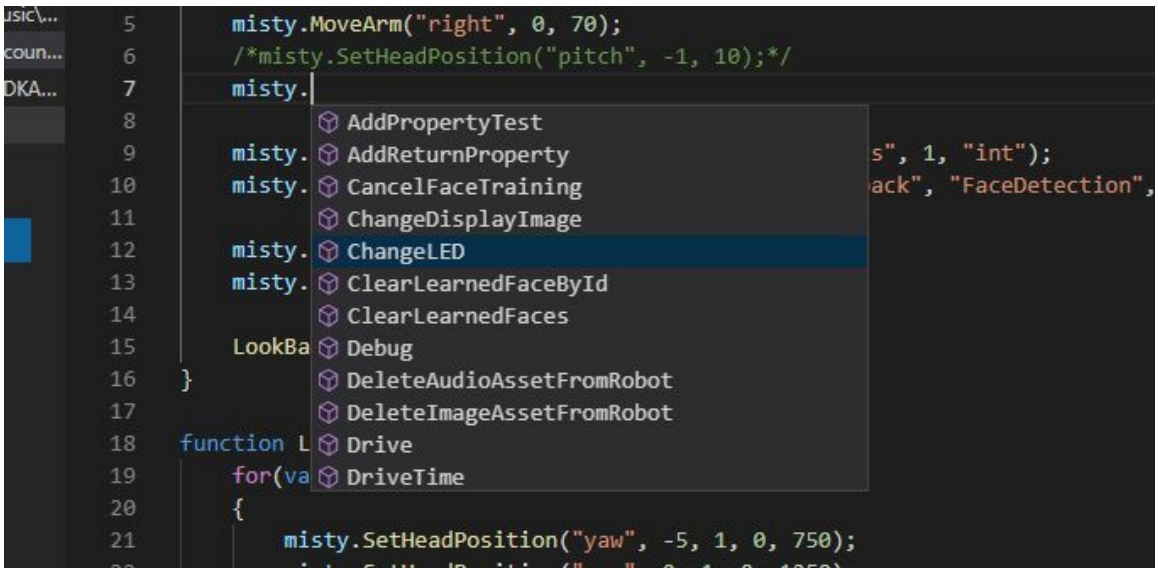
To stop all running skills, send the same `POST` command with an empty payload (no skill name specified):

```
http://{robot-ip-address}/api/alpha/sdk/cancel
{}
```

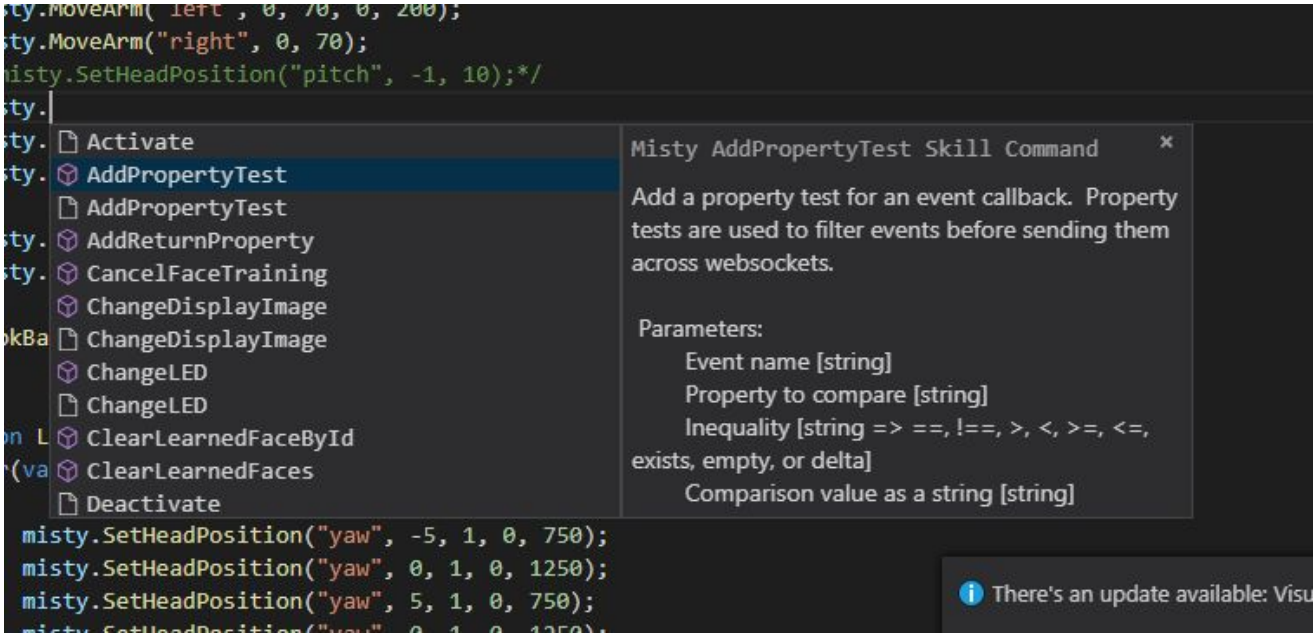
Important! Local skills are subject to a default timeout of 5 minutes. After 5 minutes the skill will cancel, even if it is performing actions or waiting on callbacks. This duration can be changed by providing a different `TimeoutInSeconds` value in the meta file. In addition, if a skill is not performing any actions nor waiting on any commands, it will automatically cancel after 5 seconds.

Using the Local Skill Extension for Visual Studio Code

As part of the Local Skill SDK, we provide the MistySkills extension for [Visual Studio **Code**]. This extension provides a list of the available methods for local skills:



As well as auto-complete, tabbed parameter entry, and information about each method:



To install the MistySkill extension, use the [Visual Studio Code Extensions Manager]. Just select **Install from VSIX** in the dropdown menu.

To activate the MistySkill extension when writing a skill:

On Mac OS - Press **Command+Shift+P** and select MistySkills. Type `misty` to start getting autocomplete and command information.

On Windows - Press **Conrol+Shift+P** and select MistySkills. Type `misty` to start getting autocomplete and command information.