
SmartThings Developer Documentation

Release 1.0

SmartThings

September 21, 2015

I	Contents	3
1	September 2015 Release FAQ	5
2	Getting Started	9
2.1	What You Need	9
2.2	What We Will Build	9
2.3	Walkthrough	9
2.3.1	Step 1: Register a developer account	9
2.3.2	Step 2: Go the developer environment page	9
2.3.3	Step 3: Create your SmartApp	10
2.3.4	Step 4: Fill in the preferences block	12
2.3.5	Step 5: Subscribe to events	12
2.3.6	Step 6: Define the event handler	13
2.3.7	Step 7: Run it in the simulator	13
2.3.8	Bonus Step: Publish your SmartApp (for you only)	16
2.4	Next Steps	16
3	Introduction	17
3.1	What is SmartThings?	17
3.1.1	What We Believe	17
3.1.2	Key Concepts	18
	SmartApps	18
	Device Type Handlers	18
	Hub-connected Device Handlers	19
	Cloud or LAN-Connected Device Handlers	19
3.1.3	Supported Protocols	19
3.2	Important Concepts	19
3.2.1	Asynchronous & Eventually Consistent Programming	19
3.2.2	Containers	20
	Accounts	20
	Locations & Users	20
	Groups	20
3.2.3	Capability Taxonomy	22
	Attributes & Events	22
	Commands	22
	Custom Capabilities	22
3.3	Architecture	22
3.3.1	Overview	22

3.3.2	Benefits	24
3.3.3	Big Picture	26
	Devices	26
	SmartThings Devices	26
	Third Party Devices	26
	Hub	27
	Connectivity Management	27
	Device-Type Execution	27
	Subscription Management	27
	SmartApp Execution	27
	Web-UI & IDE	28
3.4	Developing with SmartThings	28
3.4.1	Who can Develop with SmartThings?	28
3.4.2	Create Device Type Handlers	28
3.4.3	Create Event-Handler SmartApps	28
3.4.4	Create Integration SmartApps	29
	Custom SmartApp APIs	29
	Calling Outbound Web Services	29
3.5	Groovy – The SmartThings Programming Language	29
3.5.1	What is Groovy?	29
3.5.2	Why Groovy?	29
3.5.3	Groovy Sandboxing	30
	No Custom Classes or JARs	30
	Class Restrictions	30
	Closure Restrictions	30
	Builder Restrictions	30
	Method Restrictions	30
	Property Restrictions	31
	Other restrictions	31
3.5.4	Tips & Tricks	31
	GStrings	31
	Optional Parentheses	31
	Optional Return Statements	32
	Closures	32
3.5.5	References and Resources	32
4	Tools and IDE	35
4.1	Account Management	35
4.1.1	Locations	35
4.1.2	Hubs	36
4.1.3	Devices	36
4.1.4	SmartApps	37
4.1.5	Device Types	37
4.1.6	Publication Requests	37
4.1.7	Live Logging	37
4.2	Editor and Simulator	37
4.2.1	Creating a New SmartApp	37
4.2.2	Creating a new Device Type Handler	37
4.2.3	Using the Editor	38
4.2.4	Using the Simulator	38
4.2.5	Log Console	39
4.3	Logging	39
4.3.1	Logging Levels	39
4.3.2	Logging Examples	42

4.4	GitHub Integration	44
4.4.1	Overview	44
4.4.2	Setup	45
	Step 1 - Enable GitHub Integration	45
	Step 2 - Connect your GitHub Account to SmartThings	45
	Step 3 - Create a Fork	46
	Step 4 - Clone the Forked Repository	47
	Step 5 - Configure Git to Sync Fork with SmartThings	48
4.4.3	Repository Structure	48
4.4.4	GitHub Integration IDE Tour	49
	Color-Coded Names	49
	GitHub Actions Buttons	49
	Commit Changes	49
	Update from Repo	49
	Settings	50
4.4.5	How-To	50
	Add Files from Repository to the IDE	50
	Get Latest Code from SmartThingsPublic Repository	51
	Commit Changes in the IDE	52
	Keep Your Cloned Repo in Sync with Origin	52
4.4.6	Best Practices	52
	Sync with Upstream Repository Frequently	52
4.4.7	FAQ	53
4.4.8	Getting Help	53
5	SmartApps	55
5.1	Anatomy & Life-Cycle of a SmartApp	55
5.1.1	Types of SmartApps	55
5.1.2	SmartApp Structure	56
5.1.3	SmartApp Execution	57
5.1.4	Device Preferences	57
5.1.5	Event Subscriptions	58
5.1.6	SmartApp Sandboxing	58
5.1.7	Execution Location	58
5.1.8	Rate Limiting	58
5.2	Preferences & Settings	59
5.2.1	Preferences Overview	59
5.2.2	Page Definition	59
5.2.3	Section Definition	60
5.2.4	Single Preferences Page	61
5.2.5	Multiple Preferences Pages	63
5.2.6	Preference Elements & Inputs	64
	paragraph	64
	icon	66
	href	70
	mode	73
	label	75
	app	77
	input	77
5.2.7	Dynamic Preferences	78
5.2.8	Examples	80
5.3	Storing Data	80
5.3.1	Overview	80
5.3.2	How it Works	81

5.3.3	Using State	81
5.3.4	Atomic State	82
5.3.5	Examples	83
5.4	Events and Subscriptions	83
5.4.1	Subscribe to Specific Device Events	83
5.4.2	Subscribe to All Device Events	84
5.4.3	Subscribe to Multiple Devices	84
5.4.4	Subscribe to Location Events	84
5.4.5	The Event Object	85
5.4.6	See Also	85
5.5	Devices	85
5.5.1	Device Overview	85
5.5.2	Preferences - Selecting the Devices	86
5.5.3	Interacting with Devices	86
5.5.4	Device Attributes	86
5.5.5	Device Commands	86
5.5.6	Getting Device Current Values	87
5.5.7	Querying Event History	87
5.5.8	Sending Commands	88
5.5.9	Interacting with Multiple Devices	88
5.5.10	See Also	89
5.6	Modes	89
5.6.1	Overview	89
5.6.2	Getting the Current Mode	90
5.6.3	Getting all Modes	90
5.6.4	Setting the Mode	90
5.6.5	Allowing Users to Select Modes	90
5.6.6	Mode Events	91
5.6.7	Example	91
5.6.8	Further Reading	92
5.7	Routines	92
5.7.1	Overview	94
5.7.2	Get Available Routines	94
5.7.3	Execute Routines	94
5.7.4	Allowing Users to Select Routines	94
5.7.5	Example	95
5.7.6	Further Reading	96
5.8	Scheduling	96
5.8.1	Schedule From Now	97
5.8.2	Run Once in the Future	97
5.8.3	Run on a Schedule	98
5.8.4	Other Scheduling-related Methods	100
5.8.5	Scheduling Limitations, Best Practices, and Things Good to Know	101
5.8.6	Examples	102
5.9	Sunset and Sunrise	102
5.9.1	Sunrise and Sunset Events	102
5.9.2	Looking up Sunrise or Sunset Directly	104
5.9.3	Polling for Sunrise/Sunset	105
5.9.4	Examples	105
5.10	Calling Web Services	105
5.10.1	Configuring The Request	106
5.10.2	Handling The Response	106
5.10.3	Try It Out	107
5.10.4	See Also	108

5.11	Sending Notifications	108
5.11.1	Send Notifications with Contact Book	108
	Selecting Contacts to Notify	108
	Send Notifications to Contacts	109
	Handling Disabled Contact Book	110
	Complete Example	110
5.11.2	Send Push Notifications	111
5.11.3	Send SMS Notifications	112
5.11.4	Send Both Push and SMS Notifications	113
5.11.5	Only Display Message in the Notifications Feed	113
5.11.6	Examples	114
5.11.7	Related API Documentation	114
5.12	Example: Bon Voyage	114
5.13	Submitting SmartApps for Publication	120
5.13.1	Review Process	120
	Functional Review	121
	Code Review	121
	Publication	121
5.13.2	Best Practices	121
	Web Services	121
	Devices and Preferences	121
	Scheduling	122
	Style	122
6	Web Services SmartApps	123
6.1	Web Services SmartApps Overview	123
6.1.1	Introduction	124
6.1.2	Concepts	124
6.1.3	How it Works	124
	OAuth-Integrated App Installation Flow	125
6.1.4	The End-User Journey	126
	Initiate Connection from External System	126
	Authentication & Authorization	126
	Application Configuration	127
6.1.5	Rate Limiting	128
	Rate Limit Headers	128
	Rate Limit HTTP Status Code	128
6.2	Building a Web Services SmartApp - Part 1	128
6.2.1	Overview	129
6.2.2	Create a new SmartApp	129
6.2.3	Define Preferences	129
6.2.4	Specify Endpoints	130
6.2.5	GET Switch Information	131
6.2.6	UPDATE the Switches	131
6.2.7	Self-publish the SmartApp	132
6.2.8	Run the SmartApp in the Simulator	132
6.2.9	Make API Calls to the SmartApp	132
6.2.10	Uninstall the SmartApp	133
6.2.11	Summary	133
6.2.12	Source Code	133
6.3	Building a Web Services SmartApp - Part 2	133
6.3.1	Overview	134
6.3.2	Prerequisites	134
6.3.3	Bootstrap the Sinatra App	134

6.3.4	Get an Authorization Code	136
6.3.5	Get an Access Token	137
6.3.6	Discover the Endpoint	138
6.3.7	Make API Calls	139
6.3.8	Summary	140
6.3.9	Source Code	140
6.3.10	Appendix - Just the URLs, Please	140
	Get the OAuth Authorization Code	140
	Get the API token	140
	Discover the Endpoint URL	141
	Make API Calls	141
7	Device Handlers	143
7.1	Quick Start	143
7.1.1	Go to My Device Types in IDE	143
7.1.2	Create a new Device Handler	144
7.1.3	Make some Changes	146
7.1.4	Install with a Virtual Device	146
7.1.5	Bonus Step - Install on a Real Device	147
7.1.6	Next Steps	147
7.2	Overview	147
7.2.1	Core Concepts	149
	Capabilities	149
	Commands	151
	Attributes	151
	Actuator and Sensor	152
7.2.2	Protocols	152
7.2.3	Execution Location	152
7.2.4	Rate Limiting	152
7.3	Simulator	152
7.3.1	Status	153
7.3.2	Reply	153
7.3.3	Summary	155
7.4	Definition	155
7.4.1	Capabilities	155
7.4.2	Attributes	156
7.4.3	Commands	156
7.4.4	Fingerprinting	156
	ZigBee Fingerprinting	157
	Z-Wave Fingerprinting	157
	Fingerprinting Best Practices and Important Information	157
	Include Manufacturer and Model	157
	Adding Multiple Fingerprints	158
	Device Pairing Process	158
7.5	Tiles	158
7.5.1	Overview	159
	Common Tile Parameters	161
7.5.2	State	162
	State Selection	162
	State Parameters	162
7.5.3	Tile Definitions	163
	standardTile()	163
	controlTile()	163
	valueTile()	164

	carouselTile()	164
	multiAttributeTile()	165
7.5.4	Tile Layouts	169
7.5.5	Examples	169
7.6	Preferences	169
7.7	Parse & Events	170
7.7.1	Parse, Events, and Attributes	170
	Creating Events	171
	Multiple Events	171
	Generating Events Outside of parse	172
7.7.2	Tips	172
7.8	Z-Wave Primer	172
7.8.1	Command Classes	172
7.8.2	Listening and Sleepy Devices	173
7.8.3	Configuration	173
7.8.4	Association	174
7.9	Building Z-Wave Device Handlers	174
7.9.1	Parsing Events	174
7.9.2	Sending Commands	175
7.9.3	Sending commands in response to events	176
7.10	Z-Wave Example	176
7.11	ZigBee Primer	184
7.11.1	Device Network ID	184
7.11.2	Endpoints	184
7.11.3	Clusters	184
7.11.4	Commands	185
7.11.5	Read and Write Attributes	185
7.12	Building ZigBee Device Handlers	185
7.12.1	Read	186
7.12.2	Write	186
7.12.3	Command	186
7.12.4	Zdo Bind	187
7.12.5	ZigBee Utilities	187
7.13	ZigBee Example	187
7.14	Submitting Device Types for Publication	189
7.14.1	Review Guidelines	189
	General	189
	Web Services	190
	Style	190
	Reasons for Rejection	190
8	Cloud and LAN-Connected Devices	191
8.1	Service Manager Design Pattern	191
8.1.1	Basic Overview	191
8.1.2	Cloud-Connected Devices	191
8.1.3	LAN-Connected Devices	191
8.2	Building Cloud-Connected Device Types	191
8.2.1	Division of Labor	192
	Service-Manager Responsibilities	192
	Device Handler Responsibilities	192
	How It All Works	192
8.2.2	Building the Service Manager	193
	Authentication using OAuth	193
	End User Experience	193

	Implementation	194
	Refreshing the OAuth Token	200
	Discovery	201
	Identifying Devices in the Third-Party Device Cloud	201
	Creating Child-Devices	202
	Getting Initial Device State	202
	Handling Adds, Changes, Deletes	202
	Implicit Creation of New Child Devices	202
	Implicit Removal of Child Devices	203
	Changes in Device Name	203
	Explicit Delete Actions	203
8.2.3	Building the Device Handler	204
	The Parse Method	204
	Sending Commands to the Third-Party Cloud	204
	Receiving Events from the Third-Party Cloud	204
	Generating Events at the Request of the Service Manager	205
8.3	Building LAN-Connected Device Types	205
8.3.1	Division of Labor	205
	Service-Manager Responsibilities	206
	Device Handler Responsibilities	206
	How It All Works	206
8.3.2	Building the Service Manager	206
	Discovery	207
	SSDP	207
	mDNS/DNS-SD	207
	Handling Updates (Adds/Changes/Deletes)	208
	Adding Devices	208
	Changing Devices	209
	Deleting Devices	209
	Creating Child Devices	210
8.3.3	Building the Device Type	210
	Making Outbound HTTP Calls with HubAction	210
	Overview	210
	Creating a HubAction Object	211
	Parsing the Response	211
	Getting the Addresses	212
	REST Requests	212
	UPnP/SOAP Requests	213
	Subscribing to Device Events	213
	References and Resources	214
9	Arduino ThingShield	215
9.1	Installing the Library	215
9.2	Pairing the Shield	215
9.3	Changing the Device Type	215
9.4	Arduino Examples	216
10	Capabilities Reference	217
10.1	At a Glance	217
10.2	Acceleration Sensor	221
10.3	Actuator	222
10.4	Alarm	222
10.5	Battery	223
10.6	Beacon	224

10.7	Button	224
10.8	Carbon Monoxide Detector	225
10.9	Color Control	226
10.10	Configuration	227
10.11	Contact Sensor	227
10.12	Door Control	228
10.13	Energy Meter	228
10.14	Illuminance Measurement	229
10.15	Image Capture	230
10.16	Lock	230
10.17	Media Controller	231
10.18	Momentary	231
10.19	Motion Sensor	232
10.20	Music Player	233
10.21	Notification	234
10.22	Polling	234
10.23	Power Meter	234
10.24	Presence Sensor	235
10.25	Refresh	236
10.26	Relative Humidity Measurement	236
10.27	Relay Switch	237
10.28	Sensor	237
10.29	Signal Strength	237
10.30	Sleep Sensor	238
10.31	Smoke Detector	238
10.32	Speech Synthesis	238
10.33	Step Sensor	239
10.34	Switch	239
10.35	Switch Level	240
10.36	Temperature Measurement	240
10.37	Thermostat	241
10.38	Thermostat Cooling Setpoint	242
10.39	Thermostat Fan Mode	242
10.40	Thermostat Heating Setpoint	242
10.41	Thermostat Mode	243
10.42	Thermostat Operating State	243
10.43	Thermostat Setpoint	243
10.44	Three Axis	244
10.45	Tone	244
10.46	Touch Sensor	244
10.47	Valve	245
10.48	Water Sensor	245
11	API Documentation	247
11.1	SmartApp	248
11.1.1	installed()	248
11.1.2	updated()	249
11.1.3	uninstalled()	249
11.1.4	<device or capability preference name>	249
11.1.5	<number or decimal preference name>	250
11.1.6	<text, mode, or time preference name>	250
11.1.7	addChildDevice()	251
11.1.8	apiServerUrl()	251
11.1.9	atomicState	252

11.1.10	canSchedule()	252
11.1.11	deleteChildDevice()	252
11.1.12	getAllChildDevices()	253
11.1.13	getApiServerUrl()	253
11.1.14	getChildDevice()	253
11.1.15	getChildDevices()	253
11.1.16	getSunriseAndSunset()	254
11.1.17	getWeatherFeature()	254
11.1.18	httpDelete()	255
11.1.19	httpGet()	255
11.1.20	httpHead()	256
11.1.21	httpPost()	257
11.1.22	httpPostJson()	257
11.1.23	httpPut()	258
11.1.24	httpPutJson()	259
11.1.25	location	260
11.1.26	now()	260
11.1.27	parseJson()	260
11.1.28	parseXml()	260
11.1.29	parseLanMessage()	260
11.1.30	parseSoapMessage()	261
11.1.31	runIn()	261
11.1.32	runEvery5Minutes()	262
11.1.33	runEvery10Minutes()	262
11.1.34	runEvery15Minutes()	263
11.1.35	runEvery30Minutes()	263
11.1.36	runEvery1Hour()	264
11.1.37	runEvery3Hours()	264
11.1.38	runOnce()	265
11.1.39	schedule()	265
11.1.40	sendEvent()	266
11.1.41	sendLocationEvent()	267
11.1.42	sendNotification()	268
11.1.43	sendNotificationEvent()	268
11.1.44	sendNotificationToContacts()	269
11.1.45	sendPush()	269
11.1.46	sendPushMessage()	270
11.1.47	sendSms()	270
11.1.48	sendSmsMessage()	270
11.1.49	settings	271
11.1.50	state	271
11.1.51	stringToMap()	272
11.1.52	subscribe()	272
11.1.53	subscribeToCommand()	273
11.1.54	timeOfDayIsBetween()	273
11.1.55	timeOffset()	274
11.1.56	timeToday()	274
11.1.57	timeTodayAfter()	275
11.1.58	timeZone()	276
11.1.59	toDateTime()	276
11.1.60	unschedule()	276
11.1.61	unsubscribe()	277
11.2	Device Handler	277
11.2.1	<command name>()	278

11.2.2	parse()	278
11.2.3	apiServerUrl()	279
11.2.4	attribute()	280
11.2.5	capability()	280
11.2.6	carouselTile()	281
11.2.7	command()	282
11.2.8	controlTile()	283
11.2.9	createEvent()	283
11.2.10	definition()	284
11.2.11	details()	285
11.2.12	device	285
11.2.13	fingerprint()	286
11.2.14	getApiServerUrl()	286
11.2.15	httpDelete()	286
11.2.16	httpGet()	287
11.2.17	httpHead()	288
11.2.18	httpPost()	288
11.2.19	httpPostJson()	289
11.2.20	httpPut()	290
11.2.21	httpPutJson()	290
11.2.22	main()	291
11.2.23	metadata()	292
11.2.24	reply()	292
11.2.25	sendEvent()	293
11.2.26	simulator()	293
11.2.27	standardTile()	294
11.2.28	state	295
11.2.29	state()	295
11.2.30	status()	296
11.2.31	tiles()	297
11.2.32	valueTile()	298
11.2.33	zigbee	298
11.2.34	zwave	298
11.3	Attribute	299
11.3.1	data Type	299
11.3.2	name	300
11.3.3	values	300
11.4	Capability	300
11.4.1	name	301
11.4.2	attributes	301
11.4.3	commands	302
11.5	Command	302
11.5.1	arguments	303
11.5.2	name	303
11.6	Device	304
11.6.1	<attribute name>State	304
11.6.2	<command name>()	305
11.6.3	current<Uppercase attribute name>	305
11.6.4	capabilities	306
11.6.5	currentState()	306
11.6.6	currentValue()	307
11.6.7	displayName	307
11.6.8	id	308
11.6.9	events()	308

11.6.10	eventsBetween()	308
11.6.11	eventsSince()	309
11.6.12	hasAttribute()	309
11.6.13	hasCapability()	310
11.6.14	hasCommand()	311
11.6.15	latestState()	311
11.6.16	latestValue()	312
11.6.17	name	312
11.6.18	label	313
11.6.19	statesBetween()	313
11.6.20	statesSince()	313
11.6.21	supportedAttributes	314
11.6.22	supportedCommands	314
11.7	Event	315
11.7.1	date	315
11.7.2	id	316
11.7.3	dateValue	316
11.7.4	description	316
11.7.5	descriptionText	317
11.7.6	device	317
11.7.7	displayName	317
11.7.8	deviceId	317
11.7.9	doubleValue	318
11.7.10	floatValue	318
11.7.11	hubId	318
11.7.12	installedSmartAppId	319
11.7.13	integerValue	319
11.7.14	isDigital()	320
11.7.15	isoDate	320
11.7.16	isPhysical()	320
11.7.17	isStateChange()	320
11.7.18	jsonValue	321
11.7.19	linkText	321
11.7.20	location	321
11.7.21	locationId	322
11.7.22	longValue	322
11.7.23	name	322
11.7.24	numberValue	323
11.7.25	numericValue	323
11.7.26	source	324
11.7.27	stringValue	324
11.7.28	unit	324
11.7.29	value	325
11.7.30	xyzValue	325
11.8	Location	325
11.8.1	contactBookEnabled	326
11.8.2	currentMode	326
11.8.3	id	326
11.8.4	latitude	326
11.8.5	longitude	327
11.8.6	mode	327
11.8.7	modes	327
11.8.8	name	327
11.8.9	setMode()	328

11.8.10	temperatureScale	328
11.8.11	timeZone	328
11.8.12	zipCode	328
11.9	Mode	329
11.9.1	id	329
11.9.2	name	329
11.10	State	329
11.10.1	date	330
11.10.2	dateValue	330
11.10.3	doubleValue	330
11.10.4	floatValue	331
11.10.5	id	331
11.10.6	integerValue	331
11.10.7	isoDate	332
11.10.8	jsonValue	332
11.10.9	longValue	333
11.10.10	name	333
11.10.11	numberValue	333
11.10.12	numericValue	334
11.10.13	stringValue	334
11.10.14	unit	334
11.10.15	value	335
11.10.16	xyzValue	335
11.11	Z-Wave Reference	335

Welcome to the SmartThings developer documentation. The SmartThings platform makes it easy for software developers to build solutions for the connected home.

Developers can do this in two primary ways: First, they can write SmartApps which is code that let users connect devices, actions, and external services to create automations. Then there is Device Type Handlers which parse raw messages from devices to create standardized capabilities for developers to use.

This documentation is a work in progress. As we fill in gaps, add clarifications, and expand content, we will make every effort to not break existing bookmarks.

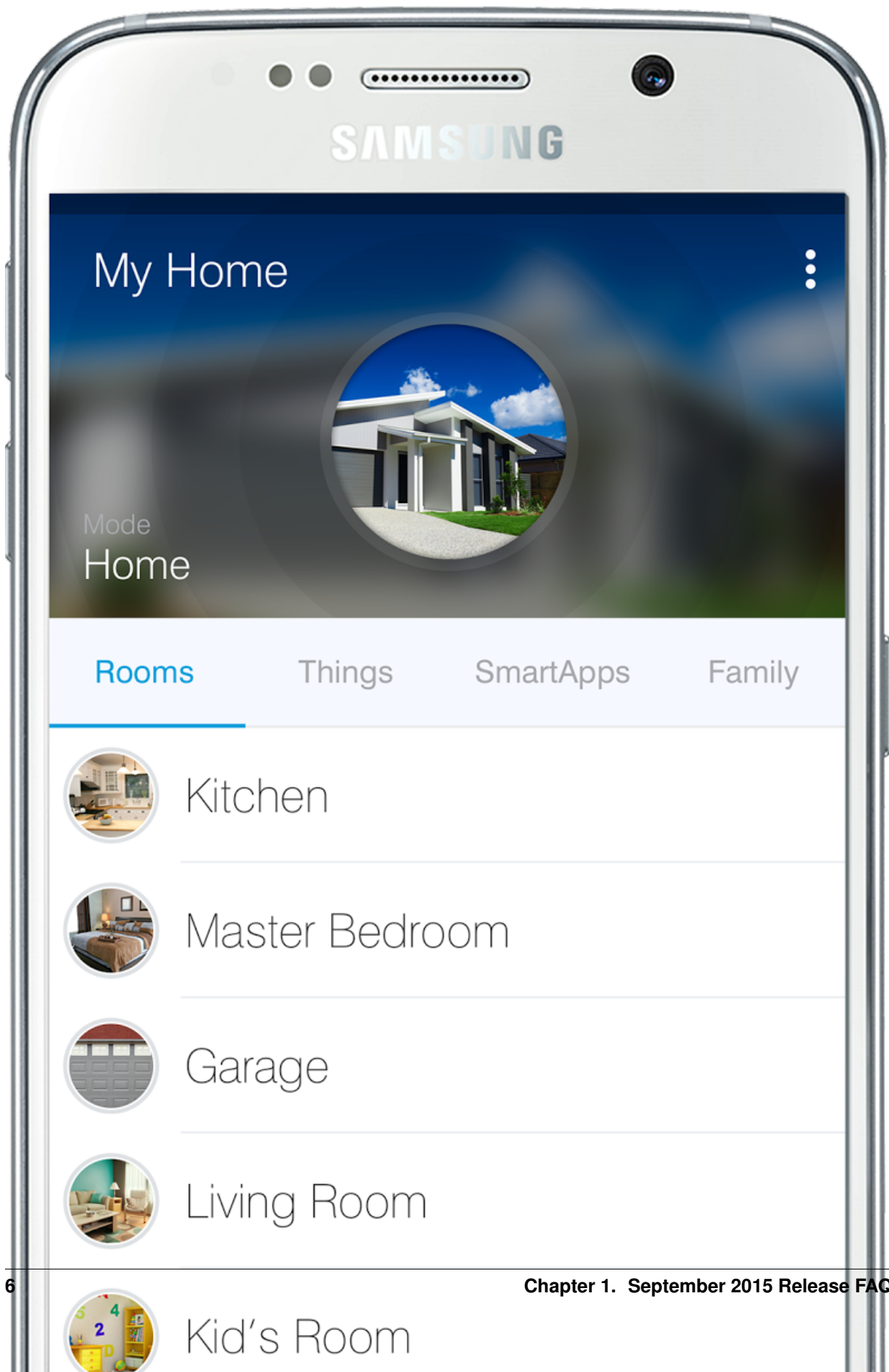
Tip: Find a bug, typo, or just want to make an improvement? This documentation is open source and available on [GitHub](#). We like contributions!

Part I

Contents

September 2015 Release FAQ

A collection of frequently asked questions about the new Samsung SmartThings Hub, and updated SmartThings mobile applications, for SmartThings developers.



What can developers do with the new Samsung SmartThings Hub and updated mobile apps?

Developers can use the new Contact Book feature to easily send notifications to a user's selected contacts, without requesting the user to enter in a phone number for each SmartApp. You can learn more about it in the [Sending Notifications](#) (page 108) documentation.

The new iOS and Android mobile apps also make use of a new Device Tiles layout, that uses a 6 column grid. There is also a new tile available to use for devices - multi-attribute tiles allow a single tile to display information about more than one attribute of a device. You can learn more in the [Tiles](#) (page 158) documentation.

With the new Hub and mobile experience, we've also laid the groundwork for exciting new developer features in the near future. Our developers are what makes SmartThings great, and we're excited to build together!

Do I need to update my SmartApps or Device Type Handlers?

Most custom SmartApps and Device Types will continue to work without the need for code changes. There are some features that you may wish to take advantage of, however, like the new multi-attribute device tile. SmartApps that send notifications should be updated to use the new Contact Book feature, but they will continue to function as they did before without updating your code.

Despite our best intentions and precautions, it is possible that your custom SmartApp or Device Type may not work as it did before. If this is the case, please report the issue to support at support@smarthings.com (include example code, relevant log messages, and screenshots if applicable). The [SmartThings Community Forums](#) are also a good place for developers to help one another. The SmartThings Community Team will be monitoring the forums to identify and help with issues, and incorporating feedback into our documentation.

Hello Home Actions now appear as “Routines” in the mobile application. Do I need to update any of my SmartApps to get or execute Routines?

No. SmartApps that work with Routines still use the methods discussed in the [Routines](#) (page 92) documentation.

At some point in the future, we may create new methods that reflect the terminology change, but we will not do so without advanced notification.

How does local SmartApp or Device Type processing work?

Certain automations can now execute locally on the Samsung SmartThings Hub. The SmartThings internal team specifies which automations are eligible for local execution. This process requires evaluation and testing of the SmartApp and devices, as well as ensuring that the necessary code artifacts are delivered to the Hub.

Any locally executing SmartApps or Device Type Handlers still send events to the SmartThings cloud. This is necessary so that the mobile application can accurately reflect the current state of the devices, as well as perform any cloud-required services (e.g., sending notifications). In the event of an Internet outage, the events will be queued and sent to the SmartThings cloud when Internet is restored.

It is not possible for developers to specify that certain Device Types or SmartApps execute in any particular location (cloud or on the hub). SmartApps or Device Types that have not been reviewed, tested, and delivered to the hub by the SmartThings team will execute in the SmartThings cloud.

What happens when the Internet to the Hub goes out?

Provided there is still power to the hub (wired or battery), any SmartApps that are able to execute locally will still run without an Internet connection. The mobile app will report the hub is offline, and because there are no events being sent to the SmartThings cloud, notifications will not work.

The radios in the hub will still function without Internet. Events to the cloud will be queued, and sent when the Internet is restored.

The mobile app has some new video-related features. How can developers utilize those capabilities?

The APIs for working with the new video features are not yet available, but we are excited to bring them to you soon!

Does the Hub have UDP support?

UDP access for developers is not currently supported, but may come in future updates.

Does the Hub support local file storage?

The Samsung SmartThings Hub stores some information about SmartApps, Device Types, and Locations locally, but this is not publicly accessible.

Can I SSH into the Hub?

No, you cannot SSH into the Samsung SmartThings Hub.

What about Bluetooth?

The Samsung SmartThings Hub ships with BLE to support future expandability, but will be inactive at product launch.

What can I do with the USB ports?

Adding USB ports to the Samsung SmartThings Hub allows for future expandability, but will have no functionality at product launch.

Does the Hub support IPv6?

No. This may come in future updates.

Does the Hub support WebSocket or Telnet for developers?

The Samsung SmartThings Hub does not support WebSocket, Telnet, or raw socket access for developers.

Does the Hub support getting local device status, or controlling local devices, without going through the SmartThings cloud? For example, can I just access the Hub to get device status or control devices?

Currently, no. We know this is a requested feature, and have identified it for future roadmap consideration.

Getting Started

This tutorial is intended to get you up and running with the SmartThings development platform - we'll walk through from setup to running your own SmartApp.

SmartApps are Groovy-based programs that allow a user to tap into the capabilities of their devices to automate their lives. Think of them as the intelligence between our physical devices.

2.1 What You Need

- A SmartThings hub that has been configured. Some devices would be good too, but not required.
- Experience with some form of programming. SmartThings uses the [Groovy programming language](#). You don't need to be a Groovy Ninja, but some familiarity with programming is assumed.

2.2 What We Will Build

We are going to build a SmartApp that turns on a light when a door opens.

Here's the recipe for our app: *When a door opens, turn on a light. When the door closes, turn the light off.*

Let's build it!

2.3 Walkthrough

2.3.1 Step 1: Register a developer account

If you haven't already, [Register for a developer account](#)

2.3.2 Step 2: Go the developer environment page

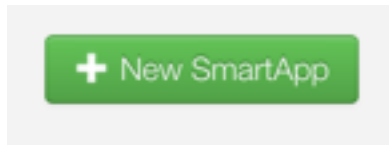
Head over to the [developer environment page](#). This is where you can manage your hubs, devices, view logging, and more. We're going to use the web-based IDE to create a SmartApp.

2.3.3 Step 3: Create your SmartApp

Click on the “My SmartApps” link:



This will take you to your SmartApps page, where you can view and manage your SmartApps. Press the “New SmartApp” button on the right of the page:



Give your app a name, author, and description. Set the category to “My Apps”. Then click the “Create” button.

New SmartApp

From Form

From Code

From Template

Definition

Name:* My First SmartApp

Name of this SmartApp. By convention capitalized with words separated by spaces, e.g. *My First App*

Namespace: Namespace

Used to uniquely identify SmartApps. Optional, but must be set for an app to be published. We suggest using your GitHub username.

Author:* Peter Gregory

Full name of the original author of this SmartApp

Description:* This is my first SmartApp! Woot!

Description of this SmartApp appropriate for display to end users

Category: My Apps

Classification of this SmartApp that is used in determining where it appears in the mobile application UI

Source Code Options

☐ Share source with other SmartThings developers

☒ Include Apache2.0 license in source

Settings

App Settings:		Name	Value
		Setting name	Setting value
		Setting name	Setting value

Optional name/value pairs used to store information needed by the SmartApp that should not be stored in the source code. Typical uses are for API keys and secrets. Accessed in the code using `appSettings.<name>`

Icons


<https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience.png>

Low resolution screen icon


<https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience@2x.png>

Medium resolution screen icon

This will take you to the IDE, where you will see some code has been filled in for you.

There are three core methods that must be defined for SmartApps:

- `preferences` is where we configure what information we need from the user to run this app.
- `installed` is the method that is called when this app is installed. Typically this is where we subscribe to events from configured devices.
- `updated` is the method that is called when the preferences are updated. Typically just unsubscribes and re-subscribes to events, since the preferences have changed.

Our example is going to be pretty simple - we will create an app that triggers a light to come on when a door opens.

At a high level, our app will need to:

1. Gather the devices (door and light) to use for this app
2. Monitor the door device - if it is opened, turn the light on. If it is closed, turn it off.

2.3.4 Step 4: Fill in the preferences block

The first thing we need to do is gather the sensors and switches we want this SmartApp to work with. We do this through the `preferences` definition.

In the IDE, replace the generated preferences block with the following:

```
preferences {
    // What door should this app be configured for?
    section ("When the door opens/closes...") {
        input "contact1", "capability.contactSensor",
            title: "Where?"
    }
    // What light should this app be configured for?
    section ("Turn on/off a light...") {
        input "switch1", "capability.switch"
    }
}
```

Click the “Save” button above the editor.

Note: When interacting with devices, SmartApps should use capabilities to ensure maximum flexibility (that’s the “`capability.contactSensor`” above). The available capabilities can be found on the [Capabilities Reference](#) (page 217) page.

More information about preferences can be found in the Preferences and Settings section of the SmartApp Developer’s Guide.

2.3.5 Step 5: Subscribe to events

In the IDE, note that there is an empty `initialize` method defined for you. This method is called from both the `installed` and `updated` methods.

This is where we will subscribe to the device(s) we want to monitor. In our case, we want to know if the door opens or closes.

Replace the `initialize` method with this:

```
def initialize() {
    subscribe(contact1, "contact", contactHandler)
}
```

Note the arguments to the `subscribe` method. The first argument, “`contact1`”, corresponds to the name in the preferences input for the contact sensor. This tells the SmartApp executor what input we are subscribing to. The second parameter, “`contact`”, is what value of the sensor we want to listen for. In this case, we use “`contact`” to listen to all value changes (open or closed). The third parameter, “`contactHandler`”, is the name of a method to call when the sensor has a state change. Let’s define that next!

(don’t forget to click the “Save” button!)

Note: More information about events and subscriptions can be found in the Events and Subscriptions section of the SmartApp Developer’s Guide.

2.3.6 Step 6: Define the event handler

Add the following code to the bottom of your SmartApp:

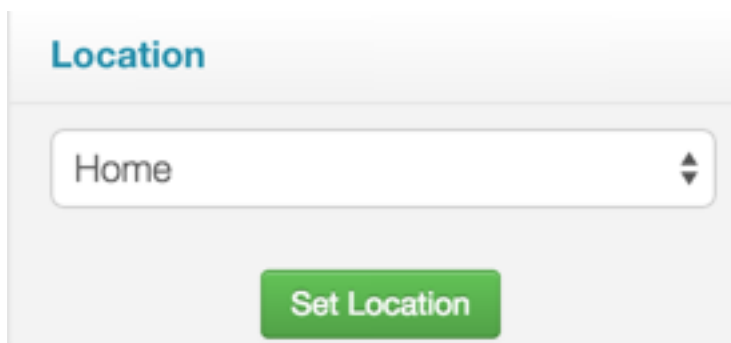
```
// event handlers are passed the event itself
def contactHandler(evt) {
    log.debug "$evt.value"

    // The contactSensor capability can be either "open" or "closed"
    // If it's "open", turn on the light!
    // If it's "closed" turn the light off.
    if (evt.value == "open") {
        switch1.on();
    } else if (evt.value == "closed") {
        switch1.off();
    }
}
```

Click the “Save” button, and let’s try it out!

2.3.7 Step 7: Run it in the simulator

To the right of the editor in the IDE, you should see a “Location” field:

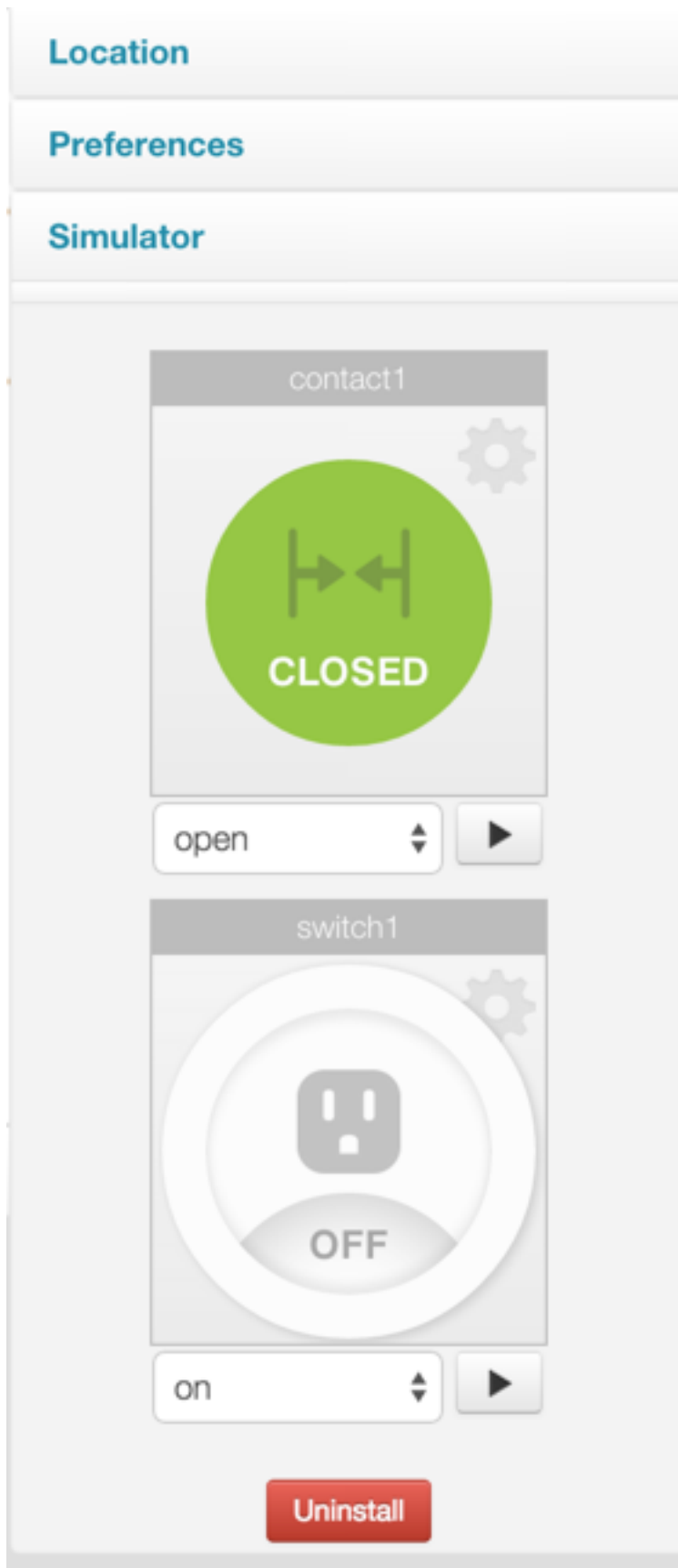


Select the location of your hub (if you have only one hub, it will be selected by default), and click “Set Location”.

Now you can pick some devices if you have them, or create some virtual devices.

The image shows a configuration interface for a SmartThings device. It has a light gray background with rounded corners. At the top, there are two tabs: "Location" and "Preferences", both in blue text. The "Preferences" tab is active. Below the tabs, the title "When the door opens/closes..." is displayed in bold. Under this title is a selection box with the heading "Where?" and an upward-pointing arrow. The box contains two sections: "Virtual Devices" with the item "contact1" and a selected blue radio button, and "Physical Devices" with the item "Kitchen Door" and an unselected white radio button. Below this selection box is another title "Turn on/off a light..." in bold. Under it is another selection box with the heading "Which?" and a downward-pointing arrow. This box contains the item "switch1". At the bottom of the interface are two buttons: a red "Uninstall" button and a green "Install" button.

Once you’ve picked some devices, click “Install” to launch the simulator:



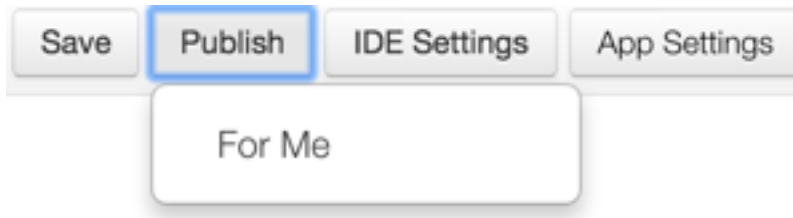
Try changing the contact sensor from closed to open - you should see the switch in the simulator turn on. If you used a real switch, you should see the light actually turn on or off!

Also note the log statements in the log console. Logging is extremely useful for debugging purposes.

2.3.8 Bonus Step: Publish your SmartApp (for you only)

We've run our app in the simulator, which is a great way to test as we develop. But we can also publish our app so we can use it from our smart phone, just like other SmartApps. Let's walk through those steps.

On top of the IDE, there's a "Publish" button right next to the Save button. Click it, and select "For me":



You should see a message indicating your app published successfully.

On your mobile phone, launch the SmartThings app, and go to the Marketplace. Select *SmartApps*, scroll to the bottom, and press *My Apps*. You should see your SmartApp here - select it, and you can install it just any other SmartApp!

2.4 Next Steps

This tutorial has shown you how to set up a developer account, use the IDE to create a simple SmartApp, use the simulator to test your SmartApp, and publish your SmartApp to your mobile phone.

In addition to using this documentation, the best way to learn is by looking at existing code and writing your own. In the IDE, there are several templates that you can review. These are great sources for learning SmartThings development! In fact, the SmartApp we built borrows heavily from (OK, it's a total clone) the "Let There Be Light" SmartApp.

Introduction

SmartThings is the open platform for the Internet of Things, bringing together developers, device makers, and service providers to make the world smarter.

In this guide, you will learn:

- A high-level understanding of the SmartThings architecture and important concepts.
- How you can develop for SmartThings.
- The programming language of SmartThings (hey, it's Groovy!).

Contents:

3.1 What is SmartThings?

SmartThings is the platform for what we call the “Open Physical Graph”. The Physical Graph is the virtual, online representation of the physical world. We believe that virtual representation should be open and easily accessible to consumers, developers, and device makers/manufacturers.

When you interact with the physical graph, it automatically reflects that interaction in the physical world. And when you interact with connected devices in the physical world, it automatically reflects that interaction in the physical graph.

This is what will make the physical world programmable - and when we say programmable, we don't mean by firmware developers with highly specialized skill sets. We mean programmable by anyone with a typical web-developer skill set.

In order to make this vision a reality, we realized early on that we needed to provide an end-to-end solution of hardware, software, a fantastic user experience, and incredible user support.

Each person's usage of the SmartThings platform will be as diverse and varied as are their lives. And to support that diversity, we need a diverse catalog of applications and devices. That is where you, as a developer, can help!

3.1.1 What We Believe

As a starting point in understanding our approach, it is important to recognize that we believe strongly in the separation of *intelligence* from devices. Said another way, we think that most of the value will be created in the *space between the devices*. We believe that the time has come when devices themselves can be limited to their primitive capabilities (open/close, on/off, heat/cool, brew/don't brew), and that the intelligence layer should be kept separate.

By doing this we allow the intelligence (or application) layer to apply flexibly across a wide range of devices, and make it easier to create applications that interact with and across the physical world. In many cases, we also benefit from lower-cost end devices, less maintenance complexity and longer battery life.

3.1.2 Key Concepts

The SmartThings platform provides methods to abstract away the underlying complexity of devices and protocols (using Device Type Handlers) from the application of intelligence (using SmartApps).

Each device in SmartThings has “capabilities”, which define and standardize available attributes and commands for a device. This allows you to develop an application for a device type, regardless of the connection protocol or the manufacturer.

All of the code that developers can write on our platform is written in Groovy, which is a dynamic, object-oriented language built for the Java platform. You can learn more about Groovy on the [Groovy – The SmartThings Programming Language](#) (page 29) page.

SmartApps

SmartApps let users connect devices, actions, and external services to create automations.

“Turn off the lights when I arrive”

“When there’s motion, alert me via SMS”

“When I’m not home, and a window is opened, sound an alarm”

As a SmartApp developer, you define the types of devices you require based on their capabilities, then write logic based on actions, schedules, and events.

A quick example of just how easy it is to write a SmartApp that turns a light on when a door opens:

```
preferences {
    input "theswitch", "capability.switch"
    input "thedoor", "capability.contactSensor"
}

def installed() {
    subscribe(thedoor, "contact.open", doorOpenHandler)
}

def doorOpenHandler(event) {
    theswitch.on()
}
```

You can learn more about SmartApps in the [SmartApps](#) (page 55).

Device Type Handlers

Device Type Handlers parse raw messages from devices to create standardized capabilities for developers to use.

A SmartApp developer who just wants to turn on a light doesn’t, and in fact can’t, know whether that light is a Z-Wave device, a ZigBee device, an IP device, etc. Device Type Handlers parse raw messages from devices to create standardized capabilities for SmartApps to use.

They also define how the devices are visually represented in our mobile apps & IDE/simulator.

Broadly speaking, there are two different types of device type handlers:

Hub-connected Device Handlers

These abstract devices that connect directly to the hub, via ZigBee or Z-Wave. You can learn more about these in the *Device Handlers* (page 143).

Cloud or LAN-Connected Device Handlers

These abstract devices that connect to SmartThings via the Local Area Network (LAN) or through the cloud services of the device manufacturer (e.g., a Sonos player). You can read more about these in the *Cloud and LAN-Connected Devices* (page 191).

3.1.3 Supported Protocols

The following protocols are supported in the SmartThings Hub:

- ZigBee - A Personal Area Mesh Networking standard for connecting and controlling devices. ZigBee is an open standard supported by the ZigBee Alliance. For more information on ZigBee see <http://en.wikipedia.org/wiki/ZigBee>.
- Z-Wave - A proprietary wireless protocol for Home Automation and Lighting Control. For more information on Z-Wave see <http://en.wikipedia.org/wiki/Z-Wave>.
- IP-Connected Devices - Local Area Network (LAN) connected devices (both hard-wired and WiFi) within the home can be connected to the SmartThings Hub.
- Cloud-Connected Devices - Some device manufacturers have their own Cloud solutions that support their devices and that we can connect to. Most of these devices are actually WiFi connected devices, but they connect to a proprietary set of Cloud services, and therefore we have to go through those services to gain access to the device.

The Samsung SmartThings Hub also ships with a Bluetooth Low Energy (BLE) chip to support future expandability. While not enabled at launch, having the hardware available allows us to support BLE in the future via firmware and software updates.

3.2 Important Concepts

3.2.1 Asynchronous & Eventually Consistent Programming

When dealing with the physical graph there will always be a delay between when you request something to happen and when it actually happens. There is latency in all networks, but it's especially pronounced when dealing with the physical graph.

To deal with this, the SmartApps platform utilizes asynchronous execution. This means that anytime you execute a command, it doesn't stop everything else from running. This helps everyone's code run the most efficiently.

Our basic methodology towards executing a command, such as turning a light switch on, is "fire and forget". This means that you execute a command, and assume it will turn on in due time, without any sort of follow up.

You cannot be guaranteed that your command has been executed, because another SmartApp could interact with your end device, and change its state. For example, you might turn a light switch on, but another app might sneak in and turn it off.

If you needed to know if a command was executed, you can subscribe to an event triggered by the command you executed and check its timestamp to ensure it fired after you told it to. You will, however, still have latency issues to take into consideration, so it's impossible to know the exact current status at any given time.

The SmartApps platform follows eventually consistent programming, meaning that responses to a request for a value in SmartApps will eventually be the same, but in the short term they might differ.

Note: In the future, we'd like to move towards providing levels of consistency for the end user, so you could specify how consistent you need your data to be.

Also, as we move some of our logic into the hub, we may consider allowing blocking methods (synchronous) as they wouldn't weigh down our network as a whole.

3.2.2 Containers

Within the SmartThings platform, there are three different “containers” that are important concepts to understand. These are: accounts, locations, and groups. These containers represent both security boundaries and navigation containers that make it easy for users to browse their devices.

The diagram below shows the hierarchical relationship between these containers. Each type of container is described below in more detail.

Accounts

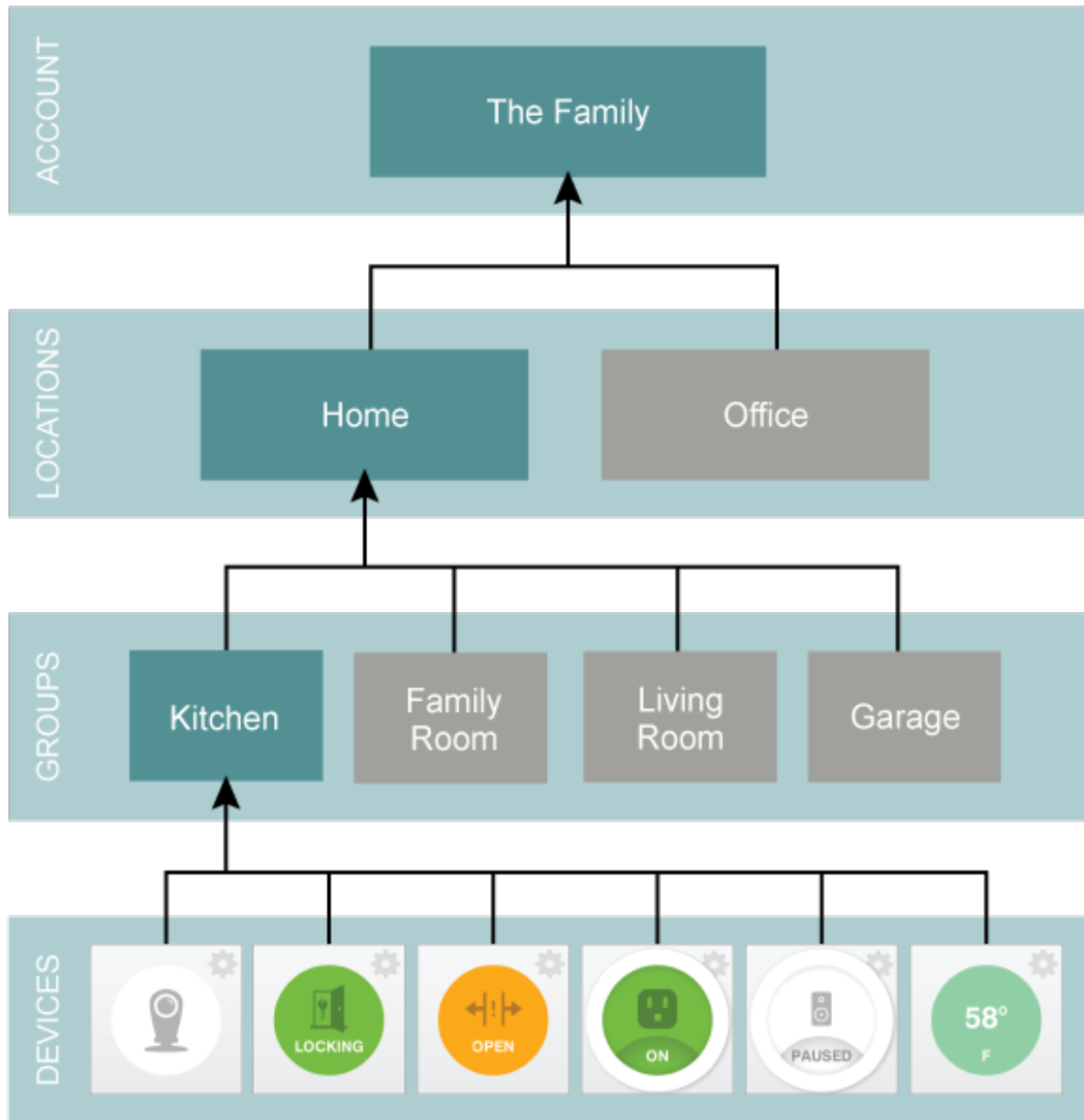
Accounts are the top-level container that represents the SmartThings ‘customer’. Accounts contain only Locations and no other types of objects.

Locations & Users

Locations are meant to represent a geo-location such as “Home” or “Office”. Locations can optionally be tagged with a geo-location (lat/long). In addition, Locations don't have to have a SmartThings Hub, but generally do. Finally, locations contain Groups or Devices.

Groups

Groups are meant to represent a room or other physical space within a location. This allows for devices to be organized into groups making navigation and security easier. A group can contain multiple devices, but devices can only be in a single group. Further, nesting of groups is not currently supported.



3.2.3 Capability Taxonomy

Capabilities represent the common taxonomy that allows us to link SmartApps with Device Handlers. An application interacts with devices based on their capabilities, so once we understand the capabilities that are needed by a SmartApp, and the capabilities that are provided by a device, we can understand which devices (based on their device type and inherent capabilities) are eligible for use within a specific SmartApp.

The *Capabilities Reference* (page 217) is evolving and is heavily influenced by existing standards like ZigBee and Z-Wave.

Capabilities themselves may decompose into both ‘Actions’ or ‘Commands’ (these are synonymous), and Attributes. Actions represent ways in which you can control or actuate the device, whereas Attributes represent state information or properties of the device.

Attributes & Events

Attributes represent the various properties or characteristics of a device. Generally speaking device attributes represent a current device state of some kind. For a temperature sensor, for example, ‘temperature’ might be an attribute. For a door lock, an attribute such as ‘status’ with values of ‘open’ or ‘closed’ might be a typical.

Commands

Commands are ways in which you can control the device. A capability is supported by a specific set of commands. For example, the ‘Switch’ capability has two required commands: ‘On’ and ‘Off’. When a device supports a specific capability, it must generally support all of the commands required of that capability.

Custom Capabilities

We do not currently support creating custom capabilities. You can, however, create a device-type handler that exposes custom commands or attributes.

3.3 Architecture

Our architecture is designed in a way that abstracts away the details of a specific device (ZigBee, Z-Wave, Wifi/IP/UPnP, etc) and allows the developer to focus solely on the capabilities and actions supported by the device (lock/unlock, on/off, etc).

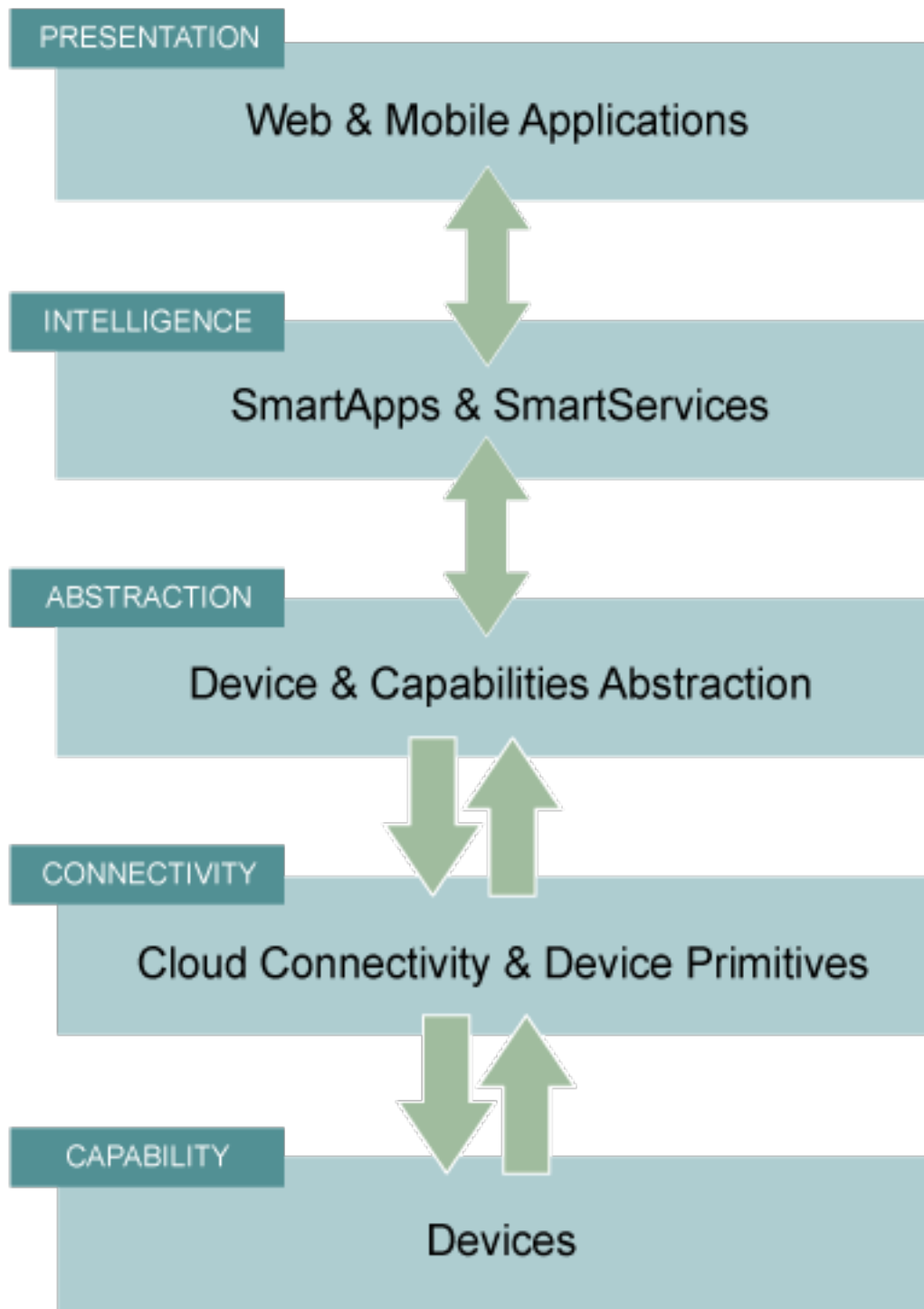
The conceptual architecture looks like this:

3.3.1 Overview

We made the decision at SmartThings to support a “Cloud First” approach for our platform. This means that in our initial release, there is a dependency on the Cloud. This means that your hub will need to be online and connected to the SmartThings cloud.

The second generation of our hub, the Samsung SmartThings Hub, allows for some hub-local capabilities. Certain automations can execute even when disconnected from the SmartThings cloud. This allows us to improve performance and insulate the customer from intermittent internet outages.

This is accomplished by delivering certain automations to the Samsung SmartThings Hub itself, where it can execute locally. The engine that executes these automations are typically referred to as “App Engine”. Events will still be sent



to the SmartThings cloud - this is necessary to ensure that the mobile application reflects the current state of the home, as well as to send any notifications or perform other cloud-based services.

The specific automations that execute locally is managed by the SmartThings internal team.

That said, there are a number of important scenarios where the Cloud is simply required and where we can't reduce or eliminate dependence on the Cloud:

There may not be a hub at all

Many devices are now connected devices, via WiFi/IP.

The advantage of Wifi devices is that they can eliminate the need for a gateway device (hub) and connect directly to the cloud. When you consider the breadth of devices like this that are coming onto the market, it's easy to imagine that there will be customers who want to be able to add intelligence to those devices through SmartApps, but that may not have a SmartThings Hub at all because all of their devices are directly connected to the vendor cloud or the SmartThings Cloud.

Put simply, if there is no Hub, then the SmartApps layer must run in the cloud!

SmartApps May Run Across both Cloud and Hub Connected Devices

As a corollary to the first point above, since there are cases where devices are not hub-connected, SmartApps might be installed to use one device that is hub-connected, and another device that is cloud-connected, all in the same app. In this case, the SmartApp needs to run in the Cloud.

There may be Multiple Hubs

While the mesh network standards for ZigBee and Z-Wave generally eliminate the need for multiple SmartThings Hubs, we didn't want to exclude this as a valid deployment configuration for large homes or even business applications of our technology. In the multi-hub case, SmartApps that use multiple devices that are split across hubs will run in the Cloud in order to simplify the complexity of application deployment.

External Service Integration

SmartApps may call external web services and calling them from our Cloud reduces risk because it allows us to easily monitor for errors and ensure the security and privacy of our customers.

In some cases, the external web services might even use IP white-listing such that they simply can't be called from the Hub running at a user's home or place of business.

Accordingly, SmartApps that use web services will run in the Cloud as well.

Important: Keep in mind that because of the abstraction layer, SmartApp developers never have to understand where or how devices connect to the SmartThings platform. All of that is hidden from the developer so that whether a device (such as a Garage Door opener) is Hub-Connected or Cloud-Connected, all they need to understand is:

```
myGarageDoor.open()
```

3.3.2 Benefits

There are a number of important benefits to the overall SmartThings approach:

Bringing Supercomputing Power to SmartApps and the Physical World

No matter how much computing power we put into the SmartThings Hub, there are scenarios where it simply wouldn't be enough.

Take for example the ability to apply advanced facial recognition algorithms to a photo taken by a connected camera to automatically determine who just walked into your house while you were away. In the Cloud, we can bring all

necessary computing power to solve complex problems, that we would not have if limited to the local processing power in a hub.

The Value of the Network Effect

Our vision is to make your physical world smarter, and we are doing that not just for our Hub and Devices, but for lots of different devices and scenarios. The easier that we make it to create that intelligence (through SmartApps), the bigger that ecosystem of developers and makers will be.

For consumers, that will mean the power of choice and the ability to solve problems with a solution that best fits their needs.

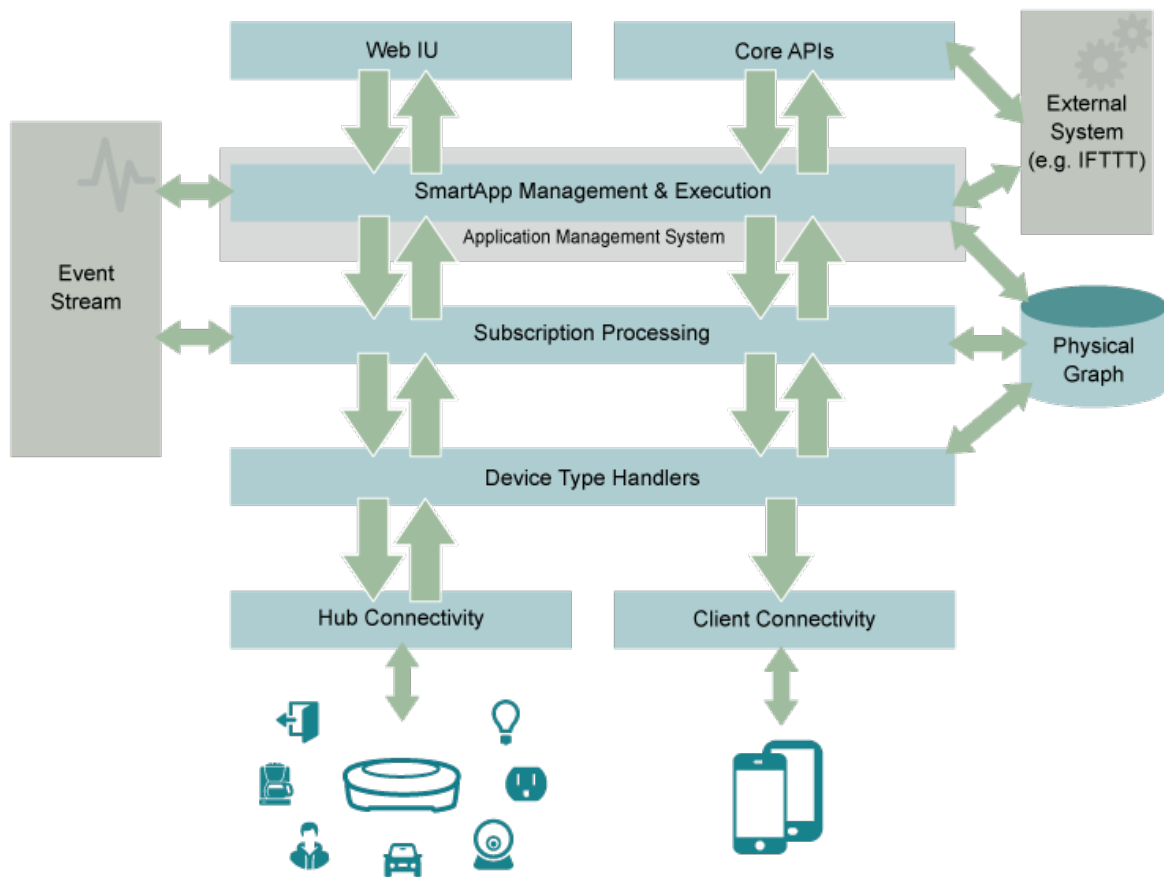
As a developer or maker, it means broad access to consumers and distribution channels for your product.

Increased Ease of Use, Accessibility, Reliability & Availability

By centralizing many capabilities into the SmartThings Cloud, we increase our ability to monitor, manage, and respond to any failures or other issues. More importantly, we can simplify the customer experience and make our solution easier to use than ever before. Further, we ensure that customers have an increased level of access and visibility.

This is not a new trend - there are many examples where on-premise capabilities have migrated to the service provider, because it improved the overall service reliability and customer experience.

3.3.3 Big Picture



Devices

Devices are the building blocks of the SmartThings infrastructure. They are the connection between the SmartThings system and the physical world. There's a huge variety in the devices you can use, some created by SmartThings but most are not.

SmartThings Devices

SmartThings manufactures a variety of devices for you to use with your SmartThings hub. Your initial kit comes with a few devices such as the [SmartSense Multi](#) which reports motion, temperature, and a variety of other sensory updates. SmartThings also manufactures and sells the [SmartSense Motion Sensor](#), [SmartSense Presence Sensor](#), [SmartSense Moisture Sensor](#), and [SmartPower Outlets](#).

Third Party Devices

The real power of SmartThings is that our system works with most home automation devices already on the market. We believe in a fully integrated approach, where you aren't tied into a particular technology or protocol. We offer compatibility with standards such as ZigBee, Z-Wave, and IP/WiFi, so we work with literally hundreds of off the shelf third-party devices. There is [an outlet made by GE](#) that allows you to integrate with your SmartThings system to dim

your lights. There are [sirens](#) for notifying you of happenings in the SmartThings system. We even have solutions for things like [locking your doors](#).

Hub

The SmartThings Hub connects directly to your broadband router and provides communication between all connected Things and the SmartThings cloud and mobile application.

- Connects any SmartThings or SmartThings Ready device to your SmartThings account.
- Simply plug into your Ethernet router and provide power.
- Build your own SmartThings kit by combining with other SmartThings devices.
- Also works with standard ZigBee and Z-Wave devices, such as GE Z-Wave in-wall switches and outlets.

The new Samsung SmartThings Hub (in addition to supporting local execution of automations as discussed above), also comes with four AA batteries. This allows for certain automations to continue, even without power. It also ships with USB ports and is Bluetooth Low Energy capable. While not active at launch, this allows for greater expansion in the future without requiring new hardware.

Connectivity Management

Connectivity Management is the layer that connects your SmartThings hub and client devices (mobile phones) to our servers, and the cloud as a whole. We have two parts of this layer currently:

- Hub Connectivity connects your hub to the cloud.
- Client Connectivity connects your client devices to the cloud.

These are the highways by which your messages are sent to the internet.

Device-Type Execution

The SmartThings system determines what device type you are using based on device type handlers. Once the device type handler is selected, the incoming messages are parsed by that particular device type. The input of the device type handler are device specific messages, and the output is normalized SmartThings events. Note that one message can lead to many SmartThings events.

Subscription Management

When events are created in the SmartThings platform, they don't inherently do anything besides publish that they've happened. Instead of events triggering change, SmartApps are configured with subscriptions that listen for defined events. The purpose of the subscription management layer is to match up events that are triggered by the device type handlers with which SmartApp is using them.

SmartApp Execution

The SmartApp is run when triggered via subscriptions, external calls to SmartApp endpoints, or scheduled methods. It's transient in nature, as it runs and then stops running on completion of its task. Any data that needs to persist throughout SmartApp instances must be stored in a special `state` variable that is discussed in the [Storing Data](#) (page 80) documentation.

Web-UI & IDE

The Web-UI sits on top of all of the other technology and allows you to monitor your devices, hubs, locations and many other aspects of your SmartThings system.

You have full control of the configuration, including editing, adding, removing, and even creating SmartApps. To create, you can write code within the IDE for SmartApps and Device Types. We also have an integrated simulator that allows you to simulate any devices, so it's not required to own the devices you develop for.

3.4 Developing with SmartThings

Developers can create their own SmartApps to create new automations for their homes, or new Device Handlers to integrate a new device.

3.4.1 Who can Develop with SmartThings?

Developing for SmartThings is free and open to all.

Our typical guideline is that anyone with a web developer (back-end) background will be best-equipped to develop with SmartThings.

That said, if you have programmed in any programming language before, you'll be able to learn how to develop with SmartThings.

Never programmed before? You may want to spend some time learning some programming basics first, but you certainly don't need a degree in Computer Science to be successful developing with SmartThings.

SmartThings uses Groovy as its development programming language. You can learn more about Groovy, and its use in SmartThings, in the *Groovy – The SmartThings Programming Language* (page 29) chapter.

3.4.2 Create Device Type Handlers

If you have a new device that we don't already support, you can actually write a new Device Handler for the SmartThings platform that integrates the device.

Building support for new device types means that you also get to design the device detail screens for how the device will appear in our mobile experience.

For information about developing for hub-connected devices, see the *Device Handlers* (page 143).

For information about developing for Cloud or LAN-Connected devices, see the *Cloud and LAN-Connected Devices* (page 191).

3.4.3 Create Event-Handler SmartApps

You can write SmartApps that provide unique functionality across devices.

These are SmartApps that don't have a UI except for configuring their behavior. They generally run in the background and handle events from devices and then issue commands back to other devices to control them.

For information about developing Event-Handler SmartApps, see the *SmartApps* (page 55).

3.4.4 Create Integration SmartApps

SmartApps can call external web services and they can expose web services for external systems to call.

Custom SmartApp APIs

Within SmartApps, you can expose API endpoints that allow you (and other users) to interact with your SmartApp through OAuth. Through these endpoints, you can initiate methods within your SmartApp that can interact with your devices, all from the web.

You can learn more about building SmartApps that expose endpoints in the [Web Services SmartApps](#) (page 123).

Calling Outbound Web Services

There a variety of helper methods you can use within a SmartApp to integrate with a third party API.

For APIs that require authentication, this involves initiating an OAuth connection and setting up endpoints to handle authentication responses. From there, you can make any type of request (POST,GET,PUT, etc) that you need and parse through the response.

For information on calling external web services, see the [Calling Web Services](#) (page 105) chapter of the *SmartApps* (page 55).

3.5 Groovy – The SmartThings Programming Language

The SmartApp programming language is a [domain-specific language](#) (DSL) based on the [Groovy programming language](#).

3.5.1 What is Groovy?

Groovy is an [object-oriented](#) programming language for the [Java platform](#). It is a [dynamic language](#) with features similar to those of [Python](#), [Ruby](#), [Perl](#), and [Smalltalk](#).

It can be used as a [scripting language](#) for the Java Platform, is dynamically compiled to [Java Virtual Machine](#) (JVM) [bytecode](#), and interoperates with other Java code and libraries.

Groovy uses a Java-like [bracket syntax](#). Most Java code is also syntactically valid Groovy.

3.5.2 Why Groovy?

By basing our first programming language on Groovy, we provide the benefits of a dynamic language with the scalability and performance of Java. Longer term, we hope to enable other programming languages in the SmartThings Cloud so that you can ultimately chose from several supported languages. Likely candidates (no promises) for additional languages include Ruby, JavaScript, Clojure, and Python.

For more documentation on the syntax, structure, and capabilities of Groovy, visit the [Groovy Documentation](#). There you will find information for getting started with Groovy, and comprehensive language documentation.

Note, however, that because of the application “sandboxing” that we do in the SmartThings Cloud, some features of Groovy are disabled for security reasons. We will discuss this more in the [Groovy Sandboxing](#) (page 30) topic below.

3.5.3 Groovy Sandboxing

SmartThings runs with a sandboxed environment. This means that not all features of the Groovy programming language are available to SmartThings developers. This is done for reasons of security and simplicity.

Here are some of the restrictions:

No Custom Classes or JARs

You cannot upload or import your own classes or JARs.

Class Restrictions

You cannot define your own classes within the SmartThings platform, or include any of your own classes. You cannot do this:

```
class Foo {  
    def list  
}
```

SmartThings allows only certain classes to be used in the sandbox. A `SecurityException` will be thrown if using a class that is not allowed.

Closure Restrictions

In SmartThings, you cannot define closures outside of methods. For example, you cannot do this:

```
def squareItClosure = {it * it}
```

More information about closures can be found in the Tips & Tricks section below.

Builder Restrictions

If you're familiar with Groovy, you likely know about the [Groovy builder pattern](#). Builders offer a nice way to build a hierarchical data structure.

Due to the way builders are implemented using closures, they will not work in SmartThings. This means things like `XMLBuilder` and `JSONBuilder` are not available to use.

Method Restrictions

Some of the methods you cannot use in SmartThings:

- `getClass`
- `getMetaClass`
- `setMetaClass`
- `propertyMissing`
- `methodMissing`
- `invokeMethod`
- `println`

- sleep

Property Restrictions

You cannot use any of the following properties in SmartThings:

- class
- metaClass

Other restrictions

A few other things you cannot do in SmartThings:

- Create and use new threads
 - Use System methods, like System.out
-

3.5.4 Tips & Tricks

To get comfortable with Groovy, it's recommended you install it and try it out. You can find information about installing Groovy [here](#).

You can also use this handy [Groovy web console](#) if you don't have Groovy installed locally. Some features may not be available, but it's a handy way to try things out quick.

A full discussion of Groovy is obviously beyond the scope of this document, but there are a few key language features that you'll see often in the SmartThings platform that are worth brief discussion here.

GStrings

Groovy Strings. What were you thinking?

GStrings are declared inside double-quotes, and may include expressions. Among other things, this allows us to build strings dynamically without having to worry about concatenation.

Expressions are defined using the `${...}` syntax.

```
def currentDateString = "The current date is ${new Date()}"
```

Properties can be referenced directly without the brackets:

```
def awesomePlatform = "SmartThings"
def newString = "Programming with $awesomePlatform is fun!"
```

Optional Parentheses

Method invocations with arguments in Groovy do not always require the arguments to be enclosed in parentheses.

These are equivalent:

```
"SmartThings".contains "Smart"
"SmartThings".contains("Smart")
```

Optional Return Statements

The return statement may be omitted from a method. The value of the last statement in a method will be the returned value, if the return keyword is not present.

These two methods are equivalent:

```
def yell() {  
    return "all caps".toUpperCase()  
}  
  
def yellAgain() {  
    "all caps".toUpperCase()  
}
```

Closures

One of the more powerful features of Groovy is its support for closures. We'll leave the exact definition of closures to computer scientists (See the Google machine if you're interested), but for our purposes, think of closures as a way to pass a function to another function.

Why would you want to do that? It allows us to be more expressive in our code, and focus on the *what*, not the *how*.

The Groovy Collections APIs make heavy use of closures. Consider this example:

```
def names = ['Erich', 'Richard', 'Gilfoyle', 'Dinesh', 'Big Head']  
def programmers = names.findAll {  
    it != 'Erich'  
}  
// programmers => ['Richard', 'Gilfoyle', 'Dinesh', 'Big Head']
```

If you're new to Groovy or functional-style programming, the above code block may look pretty strange. We'll break it down a bit.

The `findAll` method accepts a closure as an argument. The closure is defined between the brackets. `findAll` will call the closure (`it != 'Erich'`) on each element in `names`. If the item does not equal 'Erich', it will be added to the returned list (remember the optional return statement).

`it` is the default variable name for each item the closure will be called with. We can specify a different name if we wish by providing a name followed by `->`:

```
def names = ['Erich', 'Richard', 'Gilfoyle', 'Dinesh', 'Big Head']  
def programmers = names.findAll {dude ->  
    dude != 'Erich'  
}
```

3.5.5 References and Resources

Groovy is simple enough to be able to jump in and start writing code quickly, but powerful enough to get yourself stuck pretty quickly.

Here are a few resources you can use to sharpen your Groovy skills:

- [Groovy Documentation Portal](#)
- [Groovy Closures](#)
- [Groovy Collections](#)

- [Groovy Web Console](#)
- [Learn Groovy in 5 Minutes](#)

Tools and IDE

The **SmartThings IDE** (Integrated Development Environment) provides SmartThings developers with a set of tools to manage their SmartThings account, and build and publish custom SmartApps and Device Type Handlers.

In this guide, you will learn:

- The various tools that allow you to manage your Locations, Hubs, Groups, and Devices.
- How to use the IDE and simulator to build and test SmartApps and Device Type Handlers.

Contents:

4.1 Account Managment

The SmartThings IDE allows you to view and edit information about your Locations, Hubs, Devices, custom SmartApps and Device Type Handlers, as well as view a live log for all your SmartThings devices and apps.

In this chapter, you will learn:

- An overview of the various account management tools in the IDE.

4.1.1 Locations

The screenshot shows the SmartThings IDE interface. At the top, there is a navigation bar with a 'Show Navigation' link and a user greeting 'Welcome back, mpis@smartthings.com Logout'. Below the navigation bar are tabs for 'My Locations', 'My Hubs', 'My Devices', 'My SmartApps', 'My Device Types', 'Logs', and 'Documentation'. The 'Locations' section is active, displaying a table with columns for Name, Account, Groups, and three empty columns. A '+ New Location' button is in the top right corner.

Name	Account	Groups			
Minneapolis Office	mpis@smartthings.com's Account	<ul style="list-style-type: none"> Lounge Hallway & Stairs Small Conference Room SmartBlock People Maker Lab Unassigned Main Suite Music Kitchen Bean Bag Room 	events	notifications	smartapps

My Locations will show all locations registered to your account. Choosing a particular location will allow you to see more in depth information on that location, including the groups created under that location. You can also see all events, notifications, and SmartApps under a particular location.

4.1.2 Hubs

Hubs

Name	Zigbee Id	Status	Last Activity At
Minneapolis Office Hub		ACTIVE	28 Mar 2014 17:09:20

My Hubs will show all hubs registered to your account. Choosing a particular hub will give a comprehensive look at all of the attributes of your hub, with the opportunity to observe all events that have taken place, by clicking on *List Events*. You can also view all of the devices that are registered to your hub.

4.1.3 Devices

Devices — List | Tile

Display Name	Type	Location	Hub	Group	Zigbee Id	Device Network Id	Status	Last Activity
Adam's Android	Mobile Presence	Minneapolis Office		People			INACTIVE	3 weeks ago
Remote	Aeon Minimote	Minneapolis Office	Minneapolis Office Hub	Maker Lab			INACTIVE	2 weeks ago
Lounge Aeon Multisensor	Aeon Multisensor	Minneapolis Office	Minneapolis Office Hub	Lounge			INACTIVE	3 weeks ago
Andrew's Android	Mobile Presence	Minneapolis Office		People			INACTIVE	37 minutes ago
Beanbag Lamp	Dimmer Switch	Minneapolis Office	Minneapolis Office Hub	Bean Bag Room			INACTIVE	1 week ago
Main Suite	Dimmer Switch	Minneapolis Office	Minneapolis Office Hub	Main Suite			ACTIVE	2 hours ago
Kegerator Dropcam	Dropcam	Minneapolis Office		Lounge			INACTIVE	27 minutes ago
Kitchen Flood Detector	Everspring Flood Sensor	Minneapolis Office	Minneapolis Office Hub	Kitchen			INACTIVE	25 minutes ago
Hue Bridge	Hue Bridge	Minneapolis Office	Minneapolis Office Hub	Main Suite			INACTIVE	25 minutes ago
Hue Lamp 2	Hue Bulb	Minneapolis Office	Minneapolis Office Hub	Main Suite			ACTIVE	3 hours ago
Hue Lamp 1	testHue	Minneapolis Office	Minneapolis Office Hub	Main Suite			INACTIVE	18 minutes ago

My Devices will show all devices attached to any of your hubs. Choosing a particular device will give a comprehensive look at all of the attributes of your device, with the opportunity to observe all events that have taken place, by clicking on *List Events*.

4.1.4 SmartApps

My SmartApps will show all your custom (written or edited by you) SmartApps. You can view the SmartApp status, category, and locations from this list, as well as edit SmartApp metadata. You can click the SmartApp name to be taken to the editor where you can view and modify the code.

4.1.5 Device Types

My Device Types will show all your custom (written or edited by you) Device Type Handlers. You can view the status, supported capabilities, and sessions from this list, as well as edit the metadata associated with this Device Type Handler. You can click on the name to be taken to the editor, where you can view and modify the code.

4.1.6 Publication Requests

My Publication Requests will show all your publication requests for submissions to the SmartThings catalog, along with the publication request status.

4.1.7 Live Logging

Live Logging will show a live logging view for your SmartThings account. Here you will find logs for all your installed SmartApps and Device Type Handlers. You can also filter the logs by a specific SmartApp.

4.2 Editor and Simulator

The SmartThings editor and simulator allows you to create, edit, and test SmartApps and Device Type Handlers.

In this chapter, you will learn:

- How to create new SmartApps or Device Type Handlers from scratch or from an existing template.
- How to use the simulator to test your custom code - without requiring actual physical devices!

4.2.1 Creating a New SmartApp

To create a new SmartApp, click the *New SmartApp* button from the *My SmartApps* page.

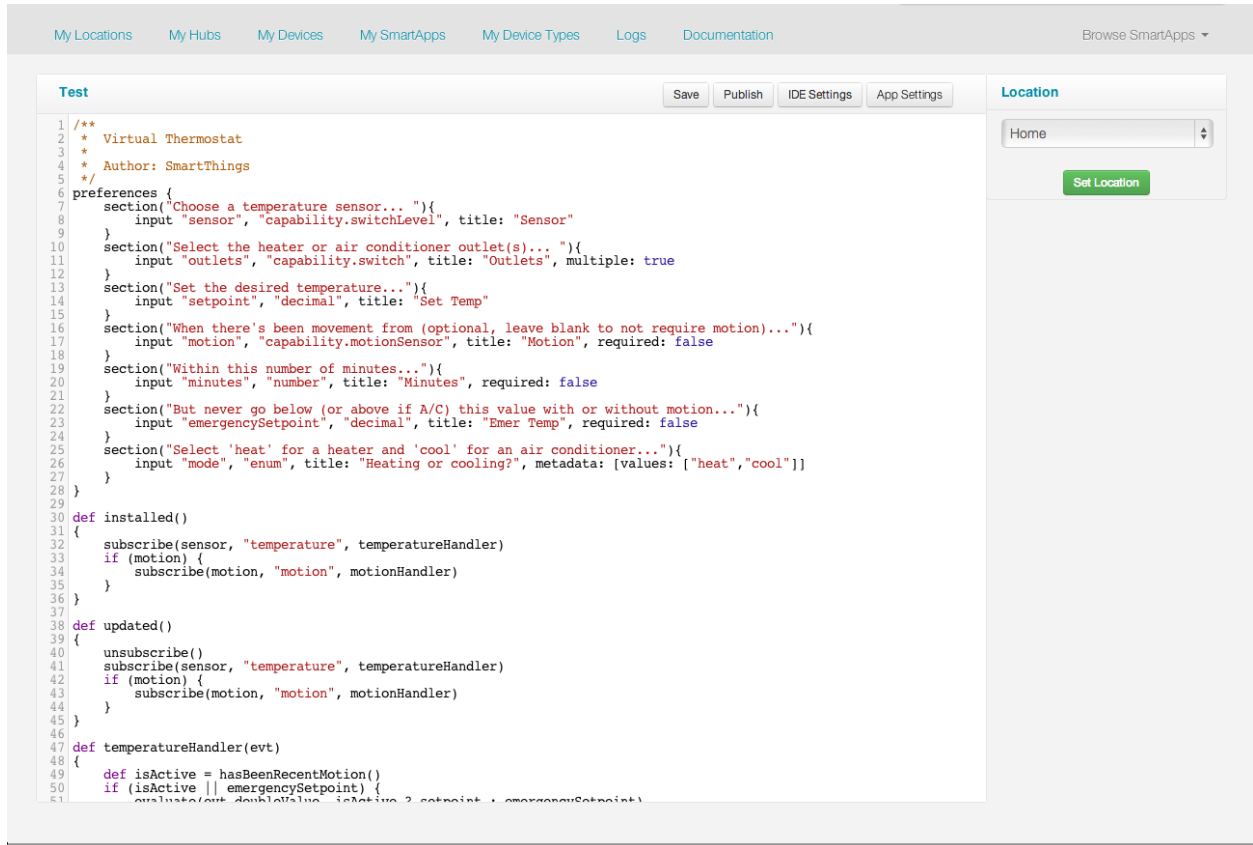
There are three different tabs on the *New SmartApp* page that allow you to create a new SmartApp in different ways:

- *From Form* allows you to create a new SmartApp based on the some metadata you can enter into the form.
- *From Code* allows you to create a new SmartApp directly from existing code. This is useful if you receive the code for a SmartApp - just paste it in to the page and a new SmartApp will be created from it.
- *From Template* allows you to create a new SmartApp based upon existing SmartApps. This is especially useful if you are new to SmartThings development, since you can start from an existing SmartApp.

4.2.2 Creating a new Device Type Handler

To create a new Device Type, click the *New SmartDevice* button from the *My Device Types* page.

There are three different tabs on the *New SmartDevice* page that allow you to create a new Device Type Handler in different ways:



- *From Form* allows you to create a new Device Type Handler based on the some metadata you can enter into the form.
- *From Code* allows you to create a new Device Type Handler directly from existing code. This is useful if you receive the code for a Device Type Handler - just paste it in to the page and a new Device Type Handler will be created from it.
- *From Template* allows you to create a new Device Type Handler based upon existing Device Type Handlers. This is especially useful if you are new to SmartThings development, since you can start from an existing Device Type Handlers.

4.2.3 Using the Editor

The SmartThings web editor allows you to edit code, and provides syntax highlighting for easy code readability.

You can choose from a variety of themes, key maps, and font sizes to suit your preferences by clicking on the *IDE Settings* button above the editor frame.

Tip: Save often! To avoid losing unsaved changes when your session login to the IDE expires, get in the habit of saving often using *Save* button.

4.2.4 Using the Simulator

The simulator allows you to test your SmartApps or Device Type Handlers within the IDE, and without requiring you to have the actual physical devices.

When you run your application in the IDE, it is always running in the simulation framework. The IDE simulator does two very important things to support simulation:

- It acts as a “Virtual Hub” that has virtual devices connected to it
- It acts as if it was the SmartThings Mobile application to receive and process status updates and support direct user actions on devices through a simulated mobile app control.

The IDE simulation environment also allows you to run the simulator attached to any of the “Locations” defined within your account.

When editing a SmartApp or Device Type Handler, you can see the simulator on the right of the page. You can choose a location and click the *Set Location* button, and then input any preferences required by the SmartApp or Device Type Handler. Click the *Install* button to run the simulator.

When simulating a SmartApp, any selected devices will appear in the IDE, along with controls to actuate the devices:

Note: Not all of the capabilities we support will work properly in the simulator. We are actively working to close those gaps.

4.2.5 Log Console

Once installed in the simulator, you will see a log console on the bottom of the page. This is where any logging statements in your code will appear.

The most recent logging statements will appear at the top of the logging console.

For more information about logging, refer to the [Logging](#) (page 39) chapter of this guide.

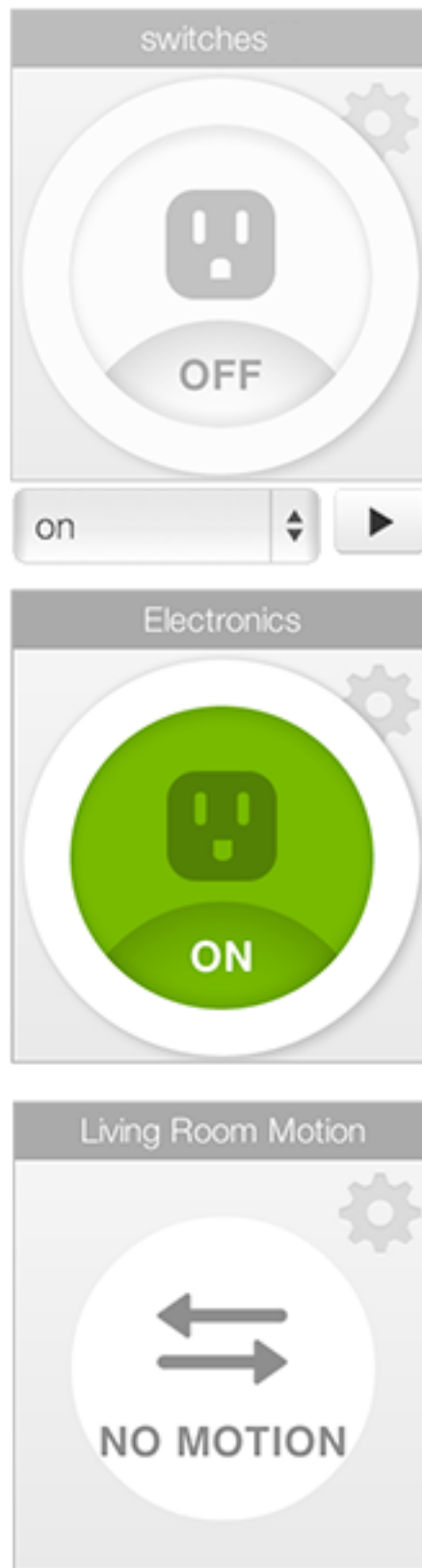
4.3 Logging

Let’s take a minute to talk about logging in SmartApps and Device Handlers. There is an instance of a logger (`log`) injected into each SmartApp/Device Handler and available for you use. Currently, we do not support a debugger environment for stepping through code. Logging however works to this end in enabling you to log messages to the console in the IDE. When you save your code, and start the simulation, a console panel will appear at the bottom of the IDE. This is where the log messages from your SmartApp/Device Handler will appear.

Note: The ‘Clear’ button will clear all of the messages currently in the console.

4.3.1 Logging Levels

The log instance currently supports these log levels, in decreasing order of severity:



Console

Clear

12:30:33 PM: **trace** temperature from sensor was provided with temperatureHandler...creating subscription

12:30:33 PM: **trace** Test is attempting to unsubscribe from all events

12:30:32 PM: **trace** temperature from sensor was provided with temperatureHandler...creating subscription

12:30:32 PM: **trace** Test is attempting to unsubscribe from all events

12:30:32 PM: **trace** temperature from sensor was provided with temperatureHandler...creating subscription

12:30:32 PM: **trace** Test is attempting to unsubscribe from all events

12:30:32 PM: **trace** temperature from sensor was provided with temperatureHandler...creating subscription

12:30:32 PM: **trace** Test is attempting to unsubscribe from all events

12:30:32 PM: **trace** temperature from sensor was provided with temperatureHandler...creating subscription

12:30:32 PM: **trace** Test is attempting to unsubscribe from all events

Console

Clear

12:30:33 PM: **trace** temperature from sensor was provided with temperatureHandler...creating subscription

12:30:33 PM: **trace** Test is attempting to unsubscribe from all events

12:30:32 PM: **trace** temperature from sensor was provided with temperatureHandler...creating subscription

12:30:32 PM: **trace** Test is attempting to unsubscribe from all events

12:30:32 PM: **trace** temperature from sensor was provided with temperatureHandler...creating subscription

12:30:32 PM: **trace** Test is attempting to unsubscribe from all events

12:30:32 PM: **trace** temperature from sensor was provided with temperatureHandler...creating subscription

12:30:32 PM: **trace** Test is attempting to unsubscribe from all events

12:30:32 PM: **trace** temperature from sensor was provided with temperatureHandler...creating subscription

12:30:32 PM: **trace** Test is attempting to unsubscribe from all events

Level	Usage	Description
ERROR	<code>log.error(string)</code>	Runtime errors or unexpected conditions.
WARN	<code>log.warn(string)</code>	Runtime situations that are unexpected, but not wrong. Can also be used to log use of deprecated APIs.
INFO	<code>log.info(string)</code>	Interesting runtime events. For example, turning a switch on or off.
DEBUG	<code>log.debug(string)</code>	Detailed information about the flow of the SmartApp.
TRACE	<code>log.trace(string)</code>	Most detailed information.

4.3.2 Logging Examples

Consider the following simple SmartApp which sets up some switch devices and has an event handler method that will log how many switches are currently turned on.

```
preferences {
    section {
        input "switches", "capability.switch", multiple: true
    }
}

def installed() {
    log.debug "Installed with settings: ${settings}"
    initialize()
}

def updated() {
    log.debug "Updated with settings: ${settings}"
    unsubscribe()
    initialize()
}

def initialize() {
    subscribe(switches, "switch", someEventHandler)
}

def someEventHandler(evt) {
    // returns a list of the values for all switches
    def currSwitches = switches.currentSwitch
```

```
def onSwitches = currSwitches.findAll { switchVal ->
    switchVal == "on" ? true : false
}

log.debug "${onSwitches.size()} out of ${switches.size()} switches are on"
}
```

Let's start the above SmartApp execution in the IDE. The first thing that we can see are messages like this:

```
3:31:01 PM: trace switch from switches[2] was provided with someEventHandler...creating subscription
3:31:01 PM: trace switch from switches[0] was provided with someEventHandler...creating subscription
3:31:01 PM: trace switch from switches[1] was provided with someEventHandler...creating subscription
3:31:00 PM: trace test is attempting to unsubscribe from all events
3:31:00 PM: debug Updated with settings: [switches:[switches[1], switches[0], switches[2]]]
```

It is easy to see that the *debug* message came from the `updated()` method

```
def updated() {
    log.debug "Updated with settings: ${settings}"
}
```

But where did the other *trace* messages come from? These messages are coming from the SmartApp framework. The SmartApp framework automatically will provide certain information like this during the execution of a SmartApp. Try turning one of the switches on in the IDE. You will see some more of these trace messages coming from the SmartApp framework. You will also see the *debug* message in the `someEventHandler()` method.

```
log.debug "${onSwitches.size()} out of ${switches.size()} switches are on"
```

You should expect to see something like this in the console.

Note: The newest messages appear at the top of the console output. Not the bottom.

```
3:39:48 PM: debug 2 out of 3 switches are on
3:39:48 PM: trace Replied with 'switch:on'
3:39:48 PM: trace Received command 'on' for device 'Switch Capability switches[1]'
```

Lets see an example of how each one of the log levels look when output to the console. In the `someEventHandler()` method, I've added the following log messages for this example.

```
log.error "${onSwitches.size()} out of ${switches.size()} switches are on"
log.warn "${onSwitches.size()} out of ${switches.size()} switches are on"
log.info "${onSwitches.size()} out of ${switches.size()} switches are on"
log.debug "${onSwitches.size()} out of ${switches.size()} switches are on"
log.trace "${onSwitches.size()} out of ${switches.size()} switches are on"
```

The output is nice and color coordinated so we can visually see the severity of the various levels.

Finally, an example of how the logger can be used in a try/catch block instead of getting the exception.

```
try {
    def x = "some string"
    x.somethingThatDoesNotExist
} catch (all) {
```

```
3:56:12 PM: trace 2 out of 3 switches are on
3:56:12 PM: debug 2 out of 3 switches are on
3:56:12 PM: info 2 out of 3 switches are on
3:56:12 PM: warn 2 out of 3 switches are on
3:56:12 PM: error 2 out of 3 switches are on
```

```
log.error "Something went horribly wrong!\n${all}"
}
```

```
11:32:37 AM: error Something went horribly wrong!
groovy.lang.MissingPropertyException: No such property: somethingThatDoesNotExist for class: java.lang.String
```

4.4 GitHub Integration

As an open platform, we recognize that giving our community developers access to the repository housing our SmartApps and Device Handlers is extremely important. While you can browse the code in the IDE, not having access to the repository itself is limiting. The [SmartThingsCommunity/SmartThingsPublic](#) GitHub repository is now public, allowing you to browse the source code in a more traditional format.

We have also provided an integration with the GitHub repository into the IDE. This will allow SmartThings developers to integrate their forked SmartThingsPublic repository with the IDE, including the ability to make commits to the forked repository using the IDE.

If you just want to browse the source in GitHub, you can do that using the tools you are most comfortable with.

If you want to take advantage of the GitHub integration with the IDE, read on for more information.

Note: A working knowledge of Git and GitHub is assumed in this guide. If you are new to Git and GitHub, we recommend checking out the [GitHub Bootcamp](#) to help you learn the basics. We will walk you through some specific Git steps, but a full discussion/explanation of Git is beyond the scope of this guide.

4.4.1 Overview

The GitHub IDE integration allows you to integrate your forked SmartThingsPublic repository with the IDE. This allows you to easily view and work with SmartApps or Device Types already in the repository, as well as update the versions in your IDE with upstream repository changes, and make commits to your forked repository right from the IDE.

When you setup GitHub integration in the IDE, you will create a fork of the SmartThingsPublic repository in GitHub. This will then be the repository that the IDE will be connected to. When you add files from the repository to the IDE, this is the repository it will look at to get the available files. When you commit changes in the IDE, you are making commits in your remote forked repository.

You will need to manage the syncing of your forked repository with the original SmartThingsPublic repository, just as you would with any forked repository in GitHub.

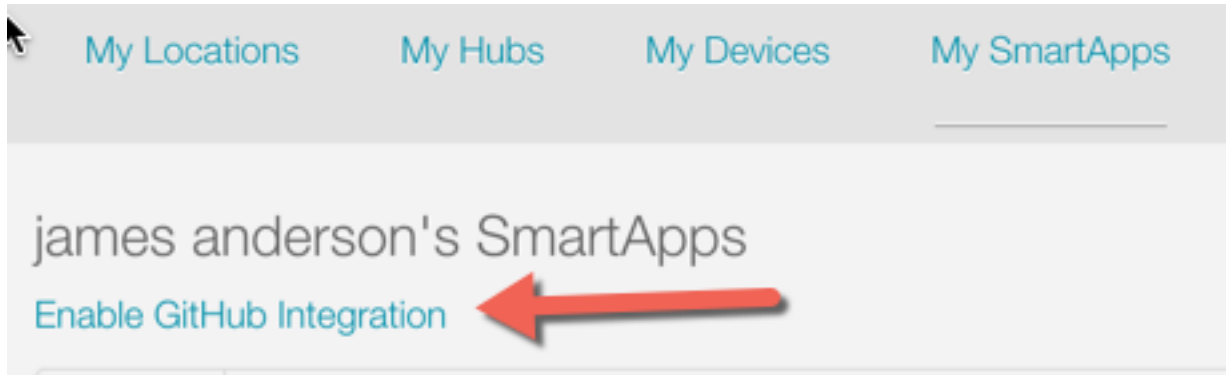
Important: Remember that the IDE is connected to your *remote forked repository in GitHub*. If you create a local clone of your repository, you will need to keep that in sync with the remote repository.

4.4.2 Setup

To connect your GitHub account with the SmartThingsPublic repository in the IDE, follow these steps.

Step 1 - Enable GitHub Integration

Click the *Enable GitHub Integration* link on the *My SmartApps* or *My Device Types* page. This will launch a wizard that will guide you through the process.



Step 2 - Connect your GitHub Account to SmartThings

On Step 1 of the wizard, follow the instructions to authorize SmartThings to integrate with your GitHub account. Click the *Next* button after you have done this.

Step 1

Connect your GitHub account to SmartThings

Authorize application

SmartThings (localhost) by @SmartThingsCommunity
would like permission to access your account

Review permissions



Repositories

Public and private



Authorize application



Click the above link and then click *Authorize application* on the GitHub page to connect SmartThings to your GitHub account. This connection allows you to use the IDE to commit changes to and pull down changes from the repositories you add to your SmartThings account. It is also used to create pull requests into the main SmartThingsCommunity repositories when you submit a SmartApp or device handler for publication.

Cancel

Next

Step 3 - Create a Fork

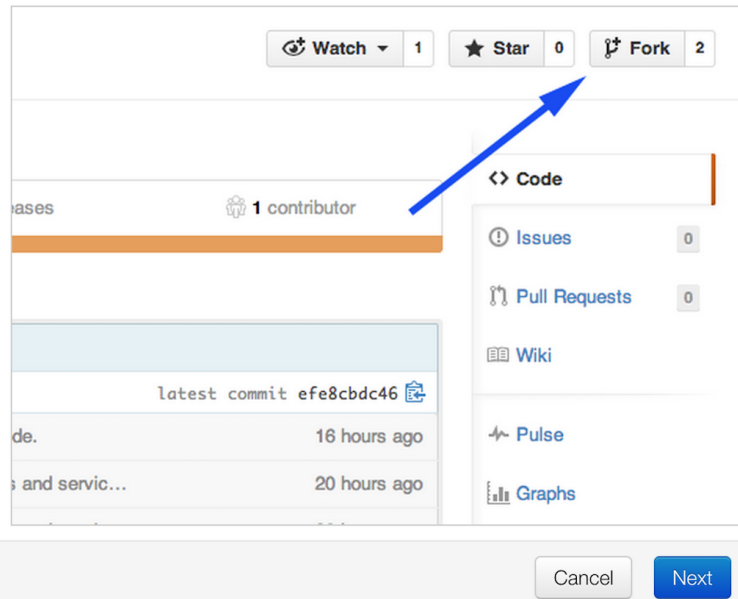
Follow the instructions to fork the SmartThingsCommunity/SmartThingsPublic repository, and then click the *Next* button.

Step 2

Fork the *SmartThingsCommunity/SmartThingsPublic* GitHub Repository

Click the above link and then click *Fork* on the GitHub repository page to fork your own copy of the repository. After you've done that and verified that the forked repository has been created, return to this page and click *Next*.

You be able to commit changes made in the IDE into this repository and update SmartApps and device types in the IDE from changes merged into this repository from other sources.



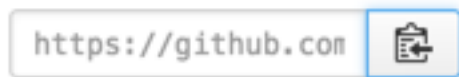
Step 4 - Clone the Forked Repository

Tip: While not required to for submitting changes, this is useful so that you have a local copy of the source code (useful for grepping the source locally, using your favorite editor, etc.), and *is* required to update your fork from the main SmartThingsPublic repository.

Follow these steps to clone your forked repository to your local machine (it is assumed that you have installed and configured Git on your local machine):

On the main page of your forked repository in GitHub, copy the HTTPS clone URL link:

HTTPS clone URL



You can clone with **HTTPS**, **SSH**, or **Subversion**.

In a terminal or command prompt, type:

```
git clone <clone URL copied as above>
```

Press Enter. This will create a local clone of your forked repository.

Step 5 - Configure Git to Sync Fork with SmartThings

If you chose to create a local clone of your forked repository, you should configure it get upstream changes from the original SmartThings repository.

On GitHub, navigate to the SmartThingsCommunity/SmartThingsPublic repository. On the right sidebar of the repository page, copy the clone URL:

Important: This is the clone URL for the main SmartThingsPublic repository, not your fork!



In a terminal or command prompt, change directories to the location of your cloned fork, and type:

```
git remote add upstream <remote URL as copied above>
```

It should look like this:

```
git remote add upstream https://github.com/SmartThingsCommunity/SmartThingsPublic.git
```

Press Enter.

In a terminal or command prompt, type:

```
git remote -v
```

This will show all the configured remotes. You should see an upstream remote configured for the SmartThingsPublic repository.

That's it! You now have connected your GitHub account with the SmartThings IDE. You will now be able to commit changes made in the IDE to this repository, and update SmartApps and Device Types in the IDE from changes merged into this repository from other sources.

4.4.3 Repository Structure

The repository is organized by type (SmartApps or Device Types) and namespace.

Each SmartApp and Device Type should be in its own directory, named the same as the SmartApp or Device Type, and appended with ".src".

For SmartApps:

```
smartapps/<namespace>/<smartapp-name>.src/<smartapp file>.groovy
```

For Device Types:

```
devicetypes/<namespace>/<device-type-name>.src/<device type file>.groovy
```

The namespace is typically your GitHub user name. When you create a SmartApp or Device Type in the IDE, you provide a namespace, which is then populated in the definition method. This namespace will be used in the directory structure as shown above.

4.4.4 GitHub Integration IDE Tour

Color-Coded Names

The first thing you may notice after enabling GitHub integration is that various SmartApps or Device Types are color-coded differently in the IDE. Each name will be color-coded differently depending on its state in the GitHub repository

Hint: Hover your mouse cursor over the name to display a tooltip to give more information.

Black Indicates that the file is unchanged between your forked GitHub repository and the IDE.

Green Indicates that the file is in the IDE only, and not in any repository.

Blue Indicates that the file exists in your GitHub repository, and has been modified in the IDE but not committed to the repository.

Magenta Indicates that the file has been updated in the repository, but not in the IDE. To resolve this, you should click the Update from Repo button, where you will see the file appear in the Obsolete column. More information about the Update from Repo button can be found below.

Red Both the IDE version and repository version have been updated, and are in need of a conflict resolution. To resolve this, you should click the Update from Repo button and follow the steps there (more information about the Update from Repo action can be found below).

Brown Indicates that the SmartApp or Device Type is unattached to the repository version. Typically this happens when a new SmartApp or Device Type is created from a template, and the name or namespace hasn't been changed. If you update from the repo without changing the name or namespace, the IDE version will be replaced with the repo version. Typically in this case you would change the name and namespace to be unique for your code.

GitHub Actions Buttons

When you enable GitHub integration, you will see a few buttons added to the My SmartApps and My DeviceTypes pages in the IDE:



Commit Changes

Clicking the Commit Changes button will first prompt you to select what repository you want to commit to, and then launch a wizard that allows you to commit any new or modified code to your forked repository. You can (and should) also add a commit message as you would normally do when making commits in Git.

Update from Repo

Clicking the Update from Repo button will first prompt you to select what repository you'd like to update from, and then launch a wizard that allows you to update your IDE code from your forked repository.

The wizard will display three columns, each of which is described below:

Tip: The files considered for this action will depend on if you are on the My SmartApps or My DeviceTypes page in the IDE. Only SmartApps will be considered if launched from My SmartApps, and only Device Types if launched from My DeviceTypes

Obsolete (updated in GitHub) Entries showing in the Obsolete column represent files that you have included in the IDE, but have since been updated in your forked repository (with no conflicts existing). To update your IDE version, select the files you wish to update, and click the Execute Update button.

Conflicted (updated locally and in GitHub) Entries showing in the Conflicted column represent files that have been modified both in the IDE and in your forked repository. To resolve these conflicts, select the files and click the Execute Update button.

New (only in GitHub) Entries showing in the New column are any files found in your forked repository that are not currently in the IDE. To bring these files into your IDE, select the files and click the Execute Update button.

Note: When updating from the repo, you also have the ability to publish any updates (either for yourself or all) by checking the Publish check box.

Settings

This is where you can find information about the repository and branch integrated with the IDE, as well as actions to update, remove, or add new repositories.

4.4.5 How-To

Add Files from Repository to the IDE

To add files from your forked SmartThingsPublic repository into the IDE, follow these steps:

Step 1 - Navigate to the **My SmartApps** or **My Device Types** page in the IDE

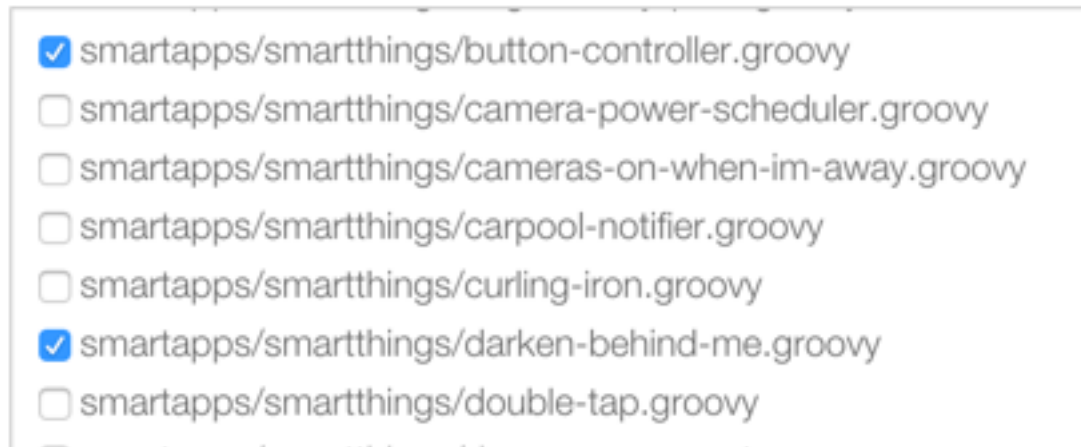
The files available to add to the IDE vary depending upon the context. If you want to add SmartApps to your IDE, navigate to the *My SmartApps* page. If you want to add Device Types, navigate to the *My Device Types*.

Step 2 - Update from Repo

Click the *Update from Repo* button (above the list of SmartApps or Device Types), and select the repo you want to update from.

In the resulting wizard, select the files you want to add to the IDE in the *New (only in GitHub)* column.

New (only in GitHub)



- ☒ smartapps/smartthings/button-controller.groovy
- ☐ smartapps/smartthings/camera-power-scheduler.groovy
- ☐ smartapps/smartthings/cameras-on-when-im-away.groovy
- ☐ smartapps/smartthings/carpool-notifier.groovy
- ☐ smartapps/smartthings/curling-iron.groovy
- ☒ smartapps/smartthings/darken-behind-me.groovy
- ☐ smartapps/smartthings/double-tap.groovy

Click the *Execute Update* button in the wizard.

The IDE will now have the files you selected.

Get Latest Code from SmartThingsPublic Repository

Note: To get the latest code from the SmartThingsPublic repository, you need to have cloned your forked repository and configured it to fetch changes from the main (upstream) SmartThingsPublic repository.

See [Step 4 - Clone the Forked Repository](#) (page 47) and [Step 5 - Configure Git to Sync Fork with SmartThings](#) (page 48) in the [Setup](#) (page 45) section for more information.

To get the latest code from the SmartThingsPublic repository, follow these steps:

Step 1 - Fetch upstream changes

Open a terminal or command prompt and change directory to the root of your forked repository.

Type `git fetch upstream` and press Enter. This will fetch the branches and their commits from the SmartThingsPublic repository.

Step 2 - Checkout your local master branch

Type `git checkout master` and press Enter.

Step 3 - Merge the changes from upstream/master to your local master branch

Type `git merge upstream/master` and press Enter. This will bring your fork's local master branch up to date with the changes in the SmartThingsPublic master branch.

Step 4 - Push changes to your remote fork

Now that we have our local repository updated synced with the latest SmartThingsPublic repository, we need to push those changes to our remote fork. Remember, this is where the IDE looks for changes (not your local clone!).

Type `git push origin master` and press Enter. This will push all commits in your local repository on the master branch, to the remote (origin) master branch.

Step 5 - Update the IDE version

Now, to update the IDE versions with your updated forked repository, click the *Update from Repo* button on the *My SmartApps* or *My Device Types* page, and select the repo you want to update from.

In the resulting wizard, check the box next to any of the files you want to update in the IDE, and click the *Execute Update* button.

The files you chose to update are now updated in the IDE.

Commit Changes in the IDE

To commit changes to a SmartApp or Device Type, whether it is a new file or already exists in the repository, Click on the *Commit Changes* button on the *My SmartApps* or *My Device Types* and select the repository you want to commit to.

In the resulting wizard, check the box next to the file you want to commit, add a commit message, and press the *Commit Changes* button.

This will make a commit in your fork.

Keep Your Cloned Repo in Sync with Origin

If you cloned your forked repository to your local machine, you will want to keep it in sync with your remote forked repository in GitHub.

When you make commits in the IDE, you are making a commit and pushing those changes to your forked repository. To sync your cloned repository with the remote forked repository, follow these steps:

Step 1 - Fetch origin changes

Open a terminal or command prompt and change directory to the root of your forked repository.

Type `git fetch origin` and press Enter. This will fetch the branches and their commits from your forked SmartThingsPublic repository.

Step 2 - Checkout your local branch

Type `git checkout master` (substitute `master` for a different branch, if you choose) and press Enter.

Step 3 - Merge the changes from origin/master to your local branch

Type `git merge origin/master` (substitute `master` for a different branch, if you want to merge from a different branch) and press Enter. This will bring your cloned repository's local branch up to date with the changes in your forked SmartThingsPublic branch.

4.4.6 Best Practices

Sync with Upstream Repository Frequently

If you have cloned your forked repository locally, you should merge changes from the upstream SmartThingsPublic repository frequently. This will help prevent your fork from becoming out-of-date with the SmartThingsPublic repository, and minimize the potential for difficult merging of conflicts.

See *Get Latest Code from SmartThingsPublic Repository* (page 51) for instructions on syncing from the upstream SmartThingsPublic repository.

4.4.7 FAQ

I don't want to grant SmartThings access to my GitHub account. Is there a way around this? Integrating the GitHub repositories with the IDE requires that you grant SmartThings read and write access to your GitHub repositories. If you would rather not grant SmartThings this level of access to your GitHub account, we recommend that you create a new GitHub user to use for SmartThings development. That will allow you to keep your primary GitHub account separate from the SmartThings account.

Do I have to use the GitHub integration? No. The GitHub integration is optional.

Does this change the process for submitting SmartApps or Device Types to SmartThings ? The process for submitting a publication request is essentially the same. The result is slightly different, in that the requests themselves become pull requests in the main SmartThingsPublic repository. This is similar to how it was working previously, but now the pull requests will be visible in the repository since the repository is public.

Can I just make a pull request to the SmartThingsPublic repository, without using the GitHub IDE Integration?

If you make a pull request to the SmartThingsPublic repository, but have not enabled GitHub integration in the IDE, your pull request will not be reviewed or merged in to the SmartThingsPublic repository. Enabling GitHub integration is what allows us to connect your GitHub account with your SmartThings account. If you have enabled the GitHub integration, and then would rather make a pull request to the SmartThingsPublic repository (using the GitHub account you enabled in the IDE) instead of publishing through the IDE, you can. We think it's more efficient to use the tools in the IDE, but nothing prevents you from making a pull request directly in this case.

Where can I find more information about working with Git? See the [Getting Help](#) (page 53) section.

I made a commit to my local GitHub fork (not using the IDE), but don't see it when I try to Update from Repo in the IDE.

Did you push your changes to your forked GitHub repository and branch associated with the IDE? Only changes pushed to your forked repository are visible to the IDE - committing changes to your local repository only, without pushing them to the repository and branch associated with the IDE, will not be visible.

I made a commit through the IDE, but I don't see it in my cloned forked repository. Did you merge the latest changes into your local repository? Remember, when you make a commit in the IDE, you are making a commit to your forked version of the SmartThingsPublic repository. If you cloned the repository locally, you need to sync your local repository with the remote repository. See [Keep Your Cloned Repo in Sync with Origin](#) (page 52) for more information.

I think I found a bug. How do I report it? First, check out the [Getting Help](#) (page 53) section below to see if any of the links may answer your questions. If you're confident you've found a bug, and it's not already discussed on the community forums, email support@smarthings.com. For the fastest response, be sure to include your SmartThings user name, your GitHub account name, and specific steps that caused the issue.

4.4.8 Getting Help

Here are some links for getting help working with Git and GitHub:

- [GitHub](#)
- [GitHub Help Page](#)
- [GitHub Bootcamp](#) - useful for getting started with Git.
- [Fork a Repo](#) - documentation on how to fork a repo in GitHub.
- [Sync a Repo](#) - documentation on how to sync a fork to the upstream repository.
- [Pushing to a Remote](#) - documentation on how to push to a remote repository.

If your questions are about the IDE integration, and aren't answered in this documentation, the [SmartThings Community Forums](#) is a great place to leverage the power of our active community developers to help.

Finally, if you have ideas to help improve this documentation, feel free to contact docs@smarththings.com.

SmartApps

SmartApps are Groovy-based programs that allow a user to tap into the capabilities of their devices to automate their lives.

If you haven't written a SmartApp yet, there are a few resources you should check out:

- [How to Build a SmartApp Demo Video](#) (2 minutes)
- Getting Started Guide - a tutorial for creating your first SmartApp.

After that, you should review the *Anatomy & Life-Cycle of a SmartApp* (page 55) to understand the overall structure and lifecycle of a SmartApp.

The contents of this guide are below:

5.1 Anatomy & Life-Cycle of a SmartApp

SmartApps are applications that allow users to tap into the capabilities of their devices to automate their lives. Most SmartApps are installed by the user via the SmartThings mobile client application, though some come pre-installed. Generally speaking, there are three different kinds of SmartApps: *Event-Handlers*, *Solution Modules*, and *Service Managers*.

5.1.1 Types of SmartApps

Event-Handler SmartApps

Event Handler SmartApps are the most common apps developed by our community. They allow you to subscribe to events from devices and call a handler method upon their firing. This method can then do a variety of things, most commonly invoking a command on another device. We're confident that if you are familiar with back end development of web sites, then you will be more than capable of developing SmartApps.

A very simple example of a SmartApp would involve you walking through a door and having the lights turn on automatically.

Solution Module SmartApps

These apps exist within the dashboard of the SmartThings app interface, and are containers for other SmartApps. The idea behind Solution Module SmartApps is to combine SmartApps that, in the real world, intuitively go together. One example of this would be the "Home & Family" section of the dashboard which allows you to see the comings and goings of your family.

Solution Module SmartApps have traditionally been built by our internal team, but we will be opening them up for external development in the near future.

Service Manager SmartApps

Service Manager SmartApps are used to connect to LAN or cloud devices, such as the Sonos or WeMo. They are the connecting glue between the unique protocols of your external devices and a device type handler you'd create for those devices. They discover devices and then continue to maintain the connection for those devices.

The Service Manager SmartApp must be installed when a user utilizes a device using LAN or the cloud, so for example, there is a Sonos Service Manager SmartApp that is installed when pairing with a Sonos.

5.1.2 SmartApp Structure

SmartApps take the form of a single [Groovy](#) script. A typical SmartApp script is composed of four sections: *Definition*, *Preferences*, *Predefined Callbacks*, and *Event Handlers*. There is also a *Mappings* section that is required for cloud-connected SmartApps that will be described later.

```

definition(
    name: "Simple Demo Application",
    namespace: "demo",
    author: "Demo User",
    description: "Turn a light on when a door opens and off when it closes.",
    category: "",
    iconUrl: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience.png",
    iconX2Url: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience@2x.png",
    oauth: true)

preferences {
    section("Select devices") {
        input "contact1", "capability.contactSensor", title: "Select contact sensor"
        input "light1", "capability.switch", title: "Select a light"
        input "lock1", "capability.lock", title: "Select a lock"
    }
}

def installed() {
    log.debug "Installed with settings: ${settings}"
    initialize()
}

def updated() {
    log.debug "Updated with settings: ${settings}"
    unsubscribe()
    initialize()
}

def initialize() {
    subscribe contact1, "contact.open", openHandler
    subscribe contact1, "contact.closed", closedHandler
}

def openHandler(evt) {
    light1.on()
    lock1.unlock()
}

def closedHandler(evt) {
    light1.off()
}

```

Definition: Metadata that determines how the app is described in the mobile app UI along with other options

Preferences: Defines what devices and other options are required to install the app. Drive the installation screens in the mobile app UI

Pre-defined methods: Pre-defined methods that are called during SmartApp installation, updating, and deletion

Event handlers: Event handlers specified in event subscriptions and other methods required to implement the SmartApp

Definition

The *definition* section of the SmartApp specifies the name of the app along with other information that identifies and describes it.

Preferences

The *preferences* section is responsible for defining the screens that appear in the mobile app when a SmartApp is installed or updated. These screens allow the user to specify which devices the SmartApp interacts with along with other configuration options that affect its behavior.

Pre-defined Callbacks

The following methods, if present, are automatically called at various times during the lifecycle of a SmartApp:

1. **installed()** - Called when a SmartApp is first installed

2. **updated()** - Called when the preferences of an installed smart app are updated
3. **uninstalled()** - Called when a SmartApp is uninstalled.
4. **childUninstalled()** - Called for the parent app when a child app is uninstalled

The installed and updated methods are commonly found in all apps. Since the selected devices may have changed when an app is updated, both of these methods typically set up the same event subscriptions, so it is common practice to put those calls in an *initialize()* method and call it from both the installed and updated methods.

The uninstalled method is typically not needed since the system automatically removes subscriptions and schedules when a SmartApp is uninstalled. However, they can be necessary in apps that integrate with other systems and need to perform cleanup on those systems.

Event Handlers

The remainder of the SmartApp contains the event handler methods specified in the event subscriptions and any other methods necessary for implementing the SmartApp. Event handler methods must have a single argument, which contains the Event object.

5.1.3 SmartApp Execution

SmartApps aren't always running. Their various methods are executed when external events occur. SmartApps execute when any of the following types of events occur:

1. **Pre-defined callback** - Any of the predefined lifecycle events described above occur.
2. **Device state change** - An attribute changes on a device, which creates an event, which triggers a subscription, which calls a handler method within your SmartApp.
3. **Location state change** - A location attribute such as *mode* changes. *Sunrise* and *sunset* are other examples of location events
4. **User action on the app** - The user taps a SmartApp icon or shortcut in the mobile app UI
5. **Scheduled event** - Using a method like *runIn()*, you call a method within your SmartApp at a particular time .
6. **Web services call** Using our web services API, you create an endpoint accessible over the web that calls a method within your SmartApp.

5.1.4 Device Preferences

The most common type of input in the preferences section specifies what kind of devices a SmartApp works with. For example, to specify that an app requires one contact sensor:

```
input "contact1", "capability.contactSensor"
```

This will generate an input element in the mobile UI that prompts for the selection of a single contact sensor (*capability.contactSensor*). *contact1* is the name of a variable that provides access to the device in the SmartApp.

Device inputs can also prompt for more than one device, so to ask for the selection of one or more switches:

```
input "switch1", "capability.switch", multiple: true
```

You can find more information about SmartApp preferences [here](#)

5.1.5 Event Subscriptions

Subscriptions allow a SmartApp to listen for events from devices, the location, and the SmartApp tile in the mobile UI. Device subscriptions are the most common and take the form:

subscribe (device , “attribute[.value]” , handlerMethod)

For example, to subscribe to all events from a contact sensor you would write:

```
subscribe(contact1, "contact", contactHandler)
```

The `contactHandler` method would then be called whenever the sensor opened or closed. You can also subscribe to specific event values, so to call a handler only when the contact sensor opens write:

```
subscribe(contact1, "contact.open", contactOpenHandler)
```

The *subscribe* method call accepts either a device or a list of devices, so you don’t need to explicitly iterate over each device in a list when you specify *multiple: true* in an input preference.

You can learn more about subscribing to device events in the [Events and Subscriptions](#) (page 83) section.

5.1.6 SmartApp Sandboxing

SmartApps are developed in a sandboxed environment. The sandbox is a way to limit developers to a specific subset of the Groovy language for performance and security. We have documented the main ways this should affect you.

5.1.7 Execution Location

With the original SmartThings Hub, all SmartApps execute in the SmartThings cloud. With the new Samsung SmartThings Hub, certain SmartApps may run locally on hub or in the SmartThings cloud. Execution location varies depending on a variety of factors, and is managed by the SmartThings internal team.

As a SmartThings developer, you should write your SmartApps to satisfy their specific use cases, regardless of where the app executes. There is currently no way to specify or force a certain execution location.

5.1.8 Rate Limiting

SmartApps that execute in the cloud are monitored for excessive resource utilization. Rate limiting ensures that no single SmartApp can consume too many shared cloud resources.

All rate limiting is based on an execution limit within a particular time window for an installed SmartApp or Device Handler. When the execution limit has been reached within the time window, no further executions will occur until the next time window. There will be an entry in the logs that will show the SmartApp or Device Type has been rate limited.

SmartApps are limited to executing 250 times in 60 seconds.

The common cause for exceeding this limit is excessive subscriptions. This may be an infinite loop of events (for example, subscribing to an “on” and “off” event, and the “on” command actually triggers the “off” event and vice versa - leading to a never-ending chain of event handlers being called). It’s also possible that a SmartApp that subscribes to a very large number of particularly “chatty” devices may run into this limit.

Additional rate limiting restrictions apply to SmartApps or Device Handlers that expose endpoints via the `mappings` definitions. You can learn about those in the SmartApp Web Services Guide.

5.2 Preferences & Settings

Note: This topic discusses preferences and settings as it pertains to SmartApps. Information about device type preferences can be found in the Device Type Developer's Guide

The preferences section of a SmartApp specifies what kinds of devices and other information is needed in order for the application to run. Inputs for each of these are presented to the user during installation of the SmartApp from the mobile UI. You can present all of these inputs on a single page, or break them up into multiple pages.

As usual, the best way to become comfortable with something is through trying it yourself. So, fire up the [web IDE](#) and try things out!

5.2.1 Preferences Overview

Preferences are made up of one or more pages, which contain one or more sections, which in turn contain one more elements. The general form of creating preferences looks like:

```
preferences {
    page() {
        section() {
            paragraph "some text"
            input "motionSensors", "capability.motionSensor",
                title: "Motions sensors?", multiple: true
        }
        section() {
            ...
        }
    }
    page() {
        ...
    }
}
```

All inputs from the user are stored in a read-only map called `settings`. You can access the value entered by the user by indexing into the map using the name as the key (`settings.someName`)

5.2.2 Page Definition

Pages can be defined a couple different ways:

page(String pageName, String pageTitle) {}

```
preferences {
    // page with name and title
    page("page name", "page title") {
        // sections go here
    }
}
```

page(options) {}

This form takes a comma-separated list of name-value arguments.

Note: this is a common Groovy pattern that allows for named arguments to be passed to a method. More info can be found [here](#).

```
preferences {
    page(name: "pageName", title: "page title",
        nextPage: "nameOfSomeOtherPage", uninstall: true) {
        // sections go here
    }
}
```

The valid options are:

name (required) String - Identifier for this page.

title String - The display title of this page

nextPage String - Used on multi-page preferences only. Should be the name of the page to navigate to next.

install Boolean - Set to `true` to allow the user to install this app from this page. Defaults to `false`. Not necessary for single-page preferences.

uninstall Boolean - Set to `true` to allow the user to uninstall from this page. Defaults to `false`. Not necessary for single-page preferences.

We will see more in-depth examples of pages in the following sections.

5.2.3 Section Definition

Pages can have one or more sections. Think of sections as way to group the input you want to gather from the user.

Sections can be created in a few different ways:

section{}

```
preferences {
    // section with no title
    section {
        // elements go here
    }
}
```

section(String sectionTitle){}

```
preferences {
    // section with title
    section("section title") {
        // elements go here
    }
}
```

section(options, String sectionTitle) {}

```
preferences {
    // section will not display in IDE
    section(mobileOnly: true, "section title")
}
```

The valid options are:

hideable Boolean - Pass `true` to allow the section to be collapsed. Defaults to `false`.

hidden Boolean - Pass `true` to specify the section is collapsed by default. Used in conjunction with `hideable`. Defaults to `false`.

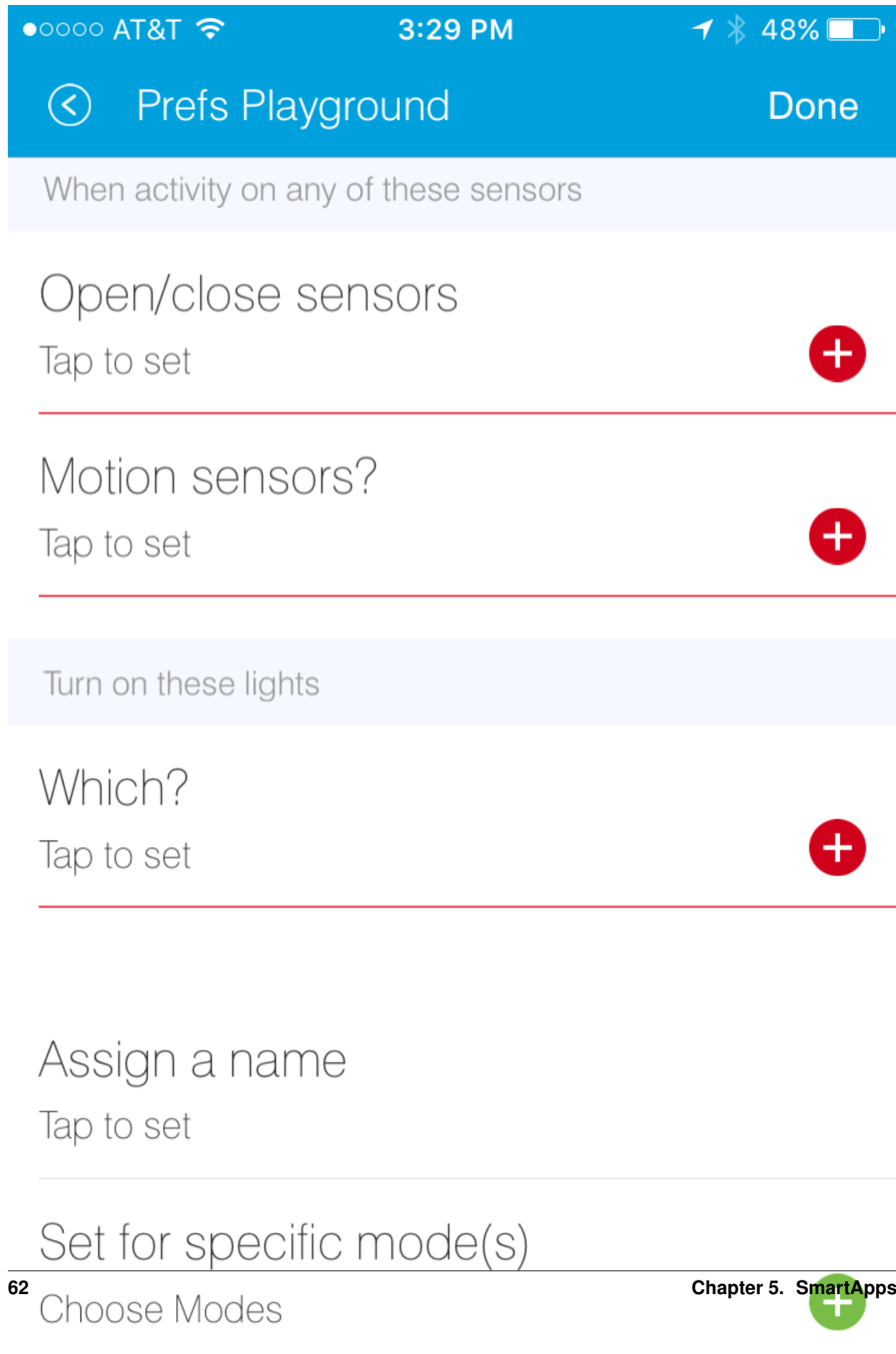
mobileOnly Boolean - Pass `true` to suppress this section from the IDE simulator. Defaults to `false`.

5.2.4 Single Preferences Page

A single page preferences declaration is composed of one or more *section* elements, which in turn contain one or more *elements*. Note that there is no *page* defined in the example below. When creating a single-page preferences app, there's no need to define the page explicitly - it's implied. Here's an example:

```
preferences {
  section("When activity on any of these sensors") {
    input "contactSensors", "capability.contactSensor",
      title: "Open/close sensors", multiple: true
    input "motionSensors", "capability.motionSensor",
      title: "Motion sensors?", multiple: true
  }
  section("Turn on these lights") {
    input "switches", "capability.switch", multiple: true
  }
}
```

Which would be rendered in the mobile app UI as:



Note that in the above example, we did not specify the name or mode input, yet they appeared on our preferences page. When defining single-page preferences, name and mode are automatically added. Also note that inputs that are marked as `required: true` are displayed differently by the mobile application, so that the user knows they are required. The mobile application will prevent the user from going to the next page or installing the SmartApp without entering required inputs.

5.2.5 Multiple Preferences Pages

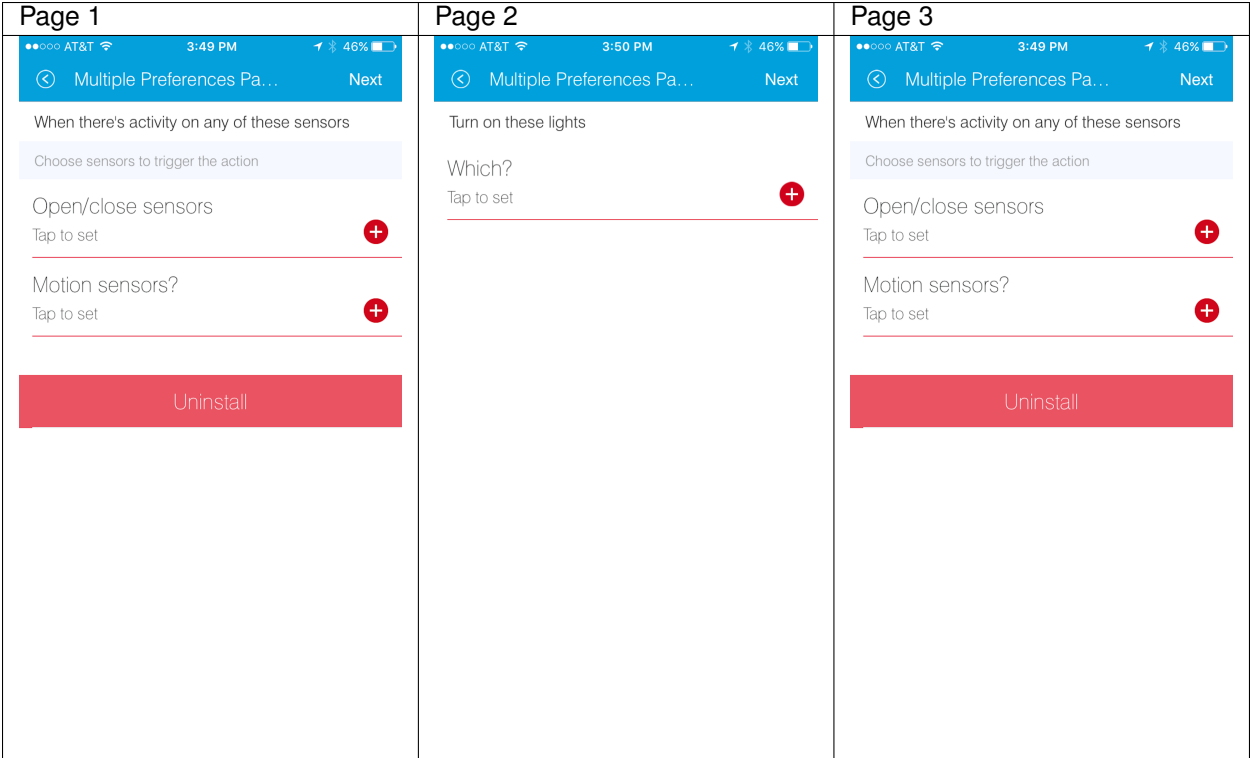
Preferences can also be broken up into multiple pages. Each page must contain one or more *section* elements. Each page specifies a *name* property that is referenced by the *nextPage* property. The *nextPage* property is used to define the flow of the pages. Unlike single page preferences, the app name and mode control fields are not automatically added, and must be specified on the desired page or pages.

Here's an example that defines three pages:

```
preferences {
  page(name: "pageOne", title: "When there's activity on any of these sensors", nextPage: "pageTwo") {
    section("Choose sensors to trigger the action") {
      input "contactSensors", "capability.contactSensor",
        title: "Open/close sensors", multiple: true

      input "motionSensors", "capability.motionSensor",
        title: "Motion sensors?", multiple: true
    }
  }
  page(name: "pageTwo", title: "Turn on these lights", nextPage: "pageThree") {
    section {
      input "switches", "capability.switch", multiple: true
    }
  }
  page(name: "pageThree", title: "Name app and configure modes", install: true, uninstall: true) {
    section([mobileOnly:true]) {
      label title: "Assign a name", required: false
      mode title: "Set for specific mode(s)", required: false
    }
  }
}
```

The resulting pages in the mobile app would show the name and mode control fields only on the third page, and the uninstall button on the first and third pages:



5.2.6 Preference Elements & Inputs

Preference pages (single or multiple) are composed of one or more sections, each of which contains one or more of the following elements:

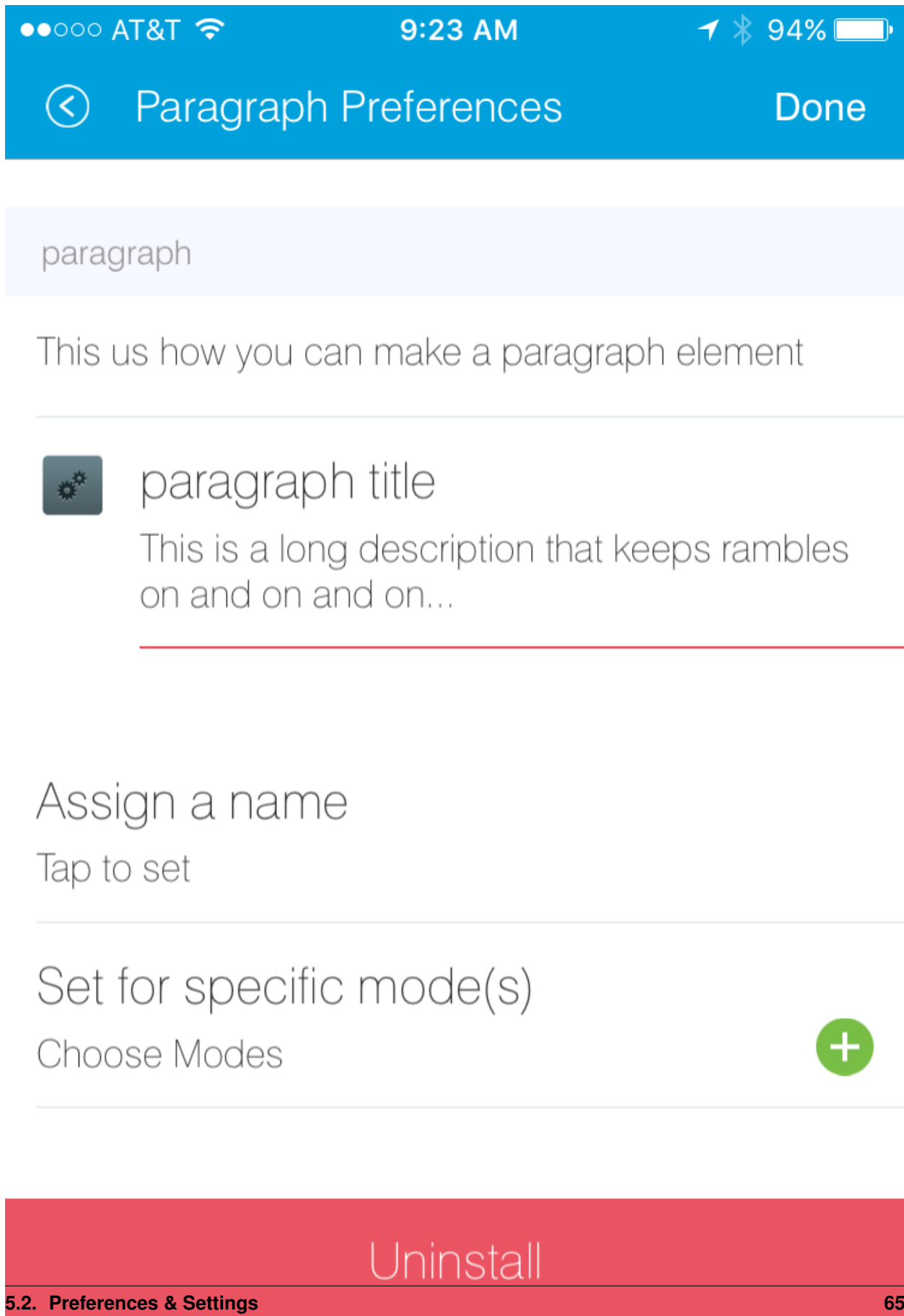
paragraph

Text that's displayed on the page for messaging and instructional purposes.

Example:

```
preferences {
  section("paragraph") {
    paragraph "This us how you can make a paragraph element"
    paragraph image: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience.png",
      title: "paragraph title",
      required: true,
      "This is a long description that rambles on and on and on..."
  }
}
```

The above preferences definition would render as:



Valid options:

title String - The title of the paragraph

image String - URL of image to use, if desired

required Boolean - `true` or `false` to specify this input is required. Defaults to `false`.

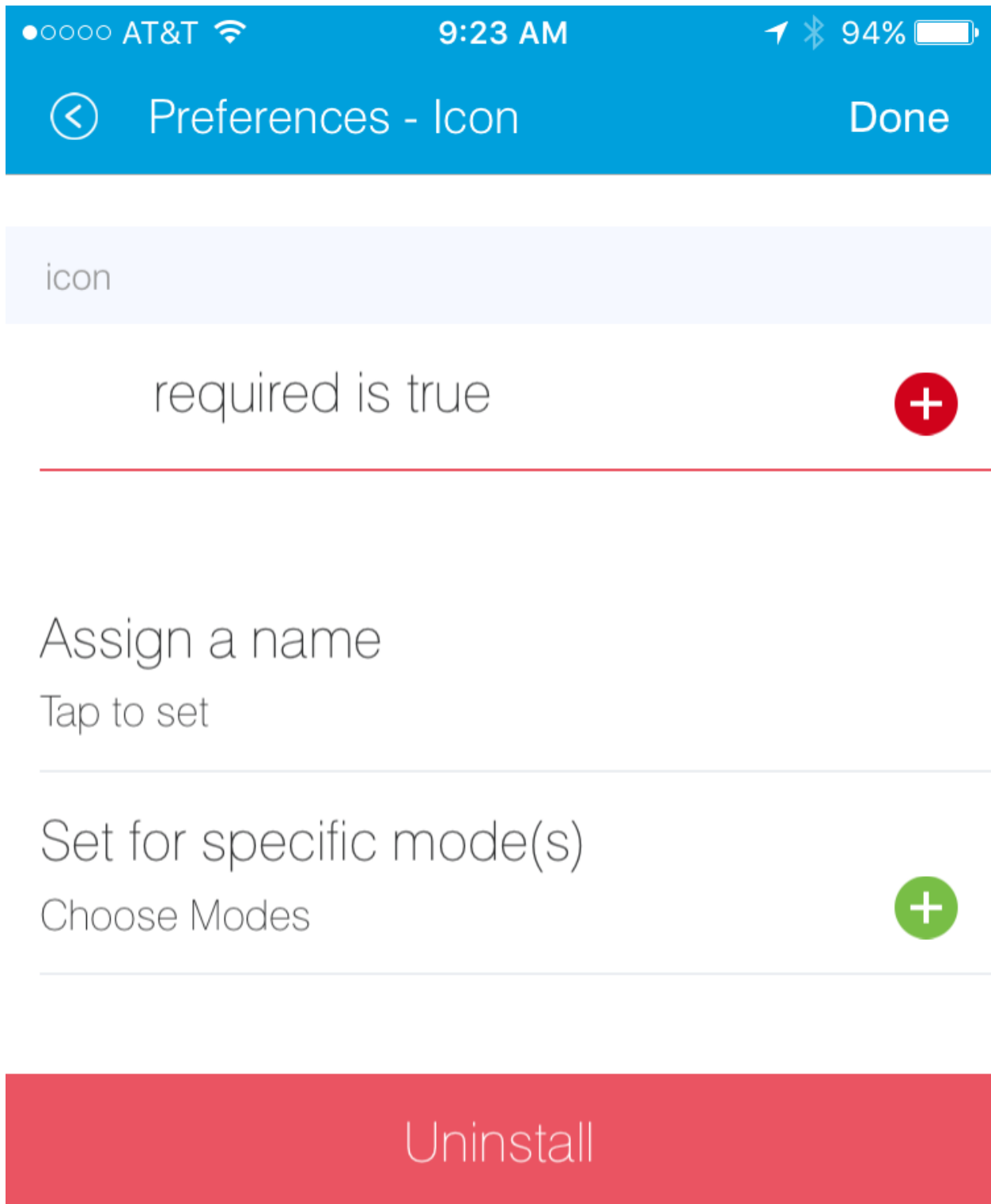
icon

Allows the user to select an icon to be used when displaying the app in the mobile UI

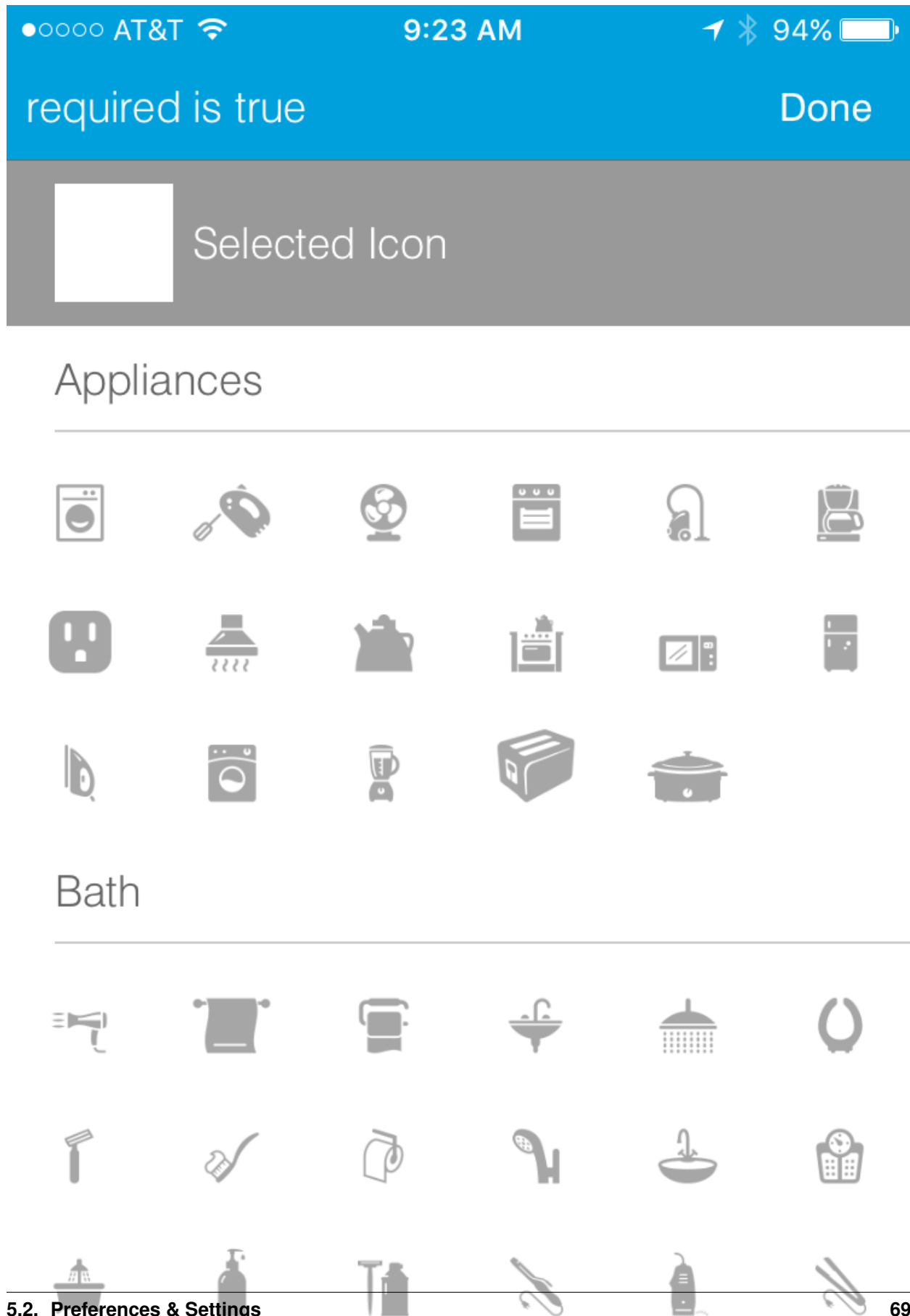
Example:

```
preferences {
  section("paragraph") {
    icon(title: "required:true",
         required: true)
  }
}
```

The above preferences definition would render as:



Tapping the element would then allow the user to choose an icon:



Valid options:

title String - The title of the icon

required Boolean - true or false to specify this input is required. Defaults to false.

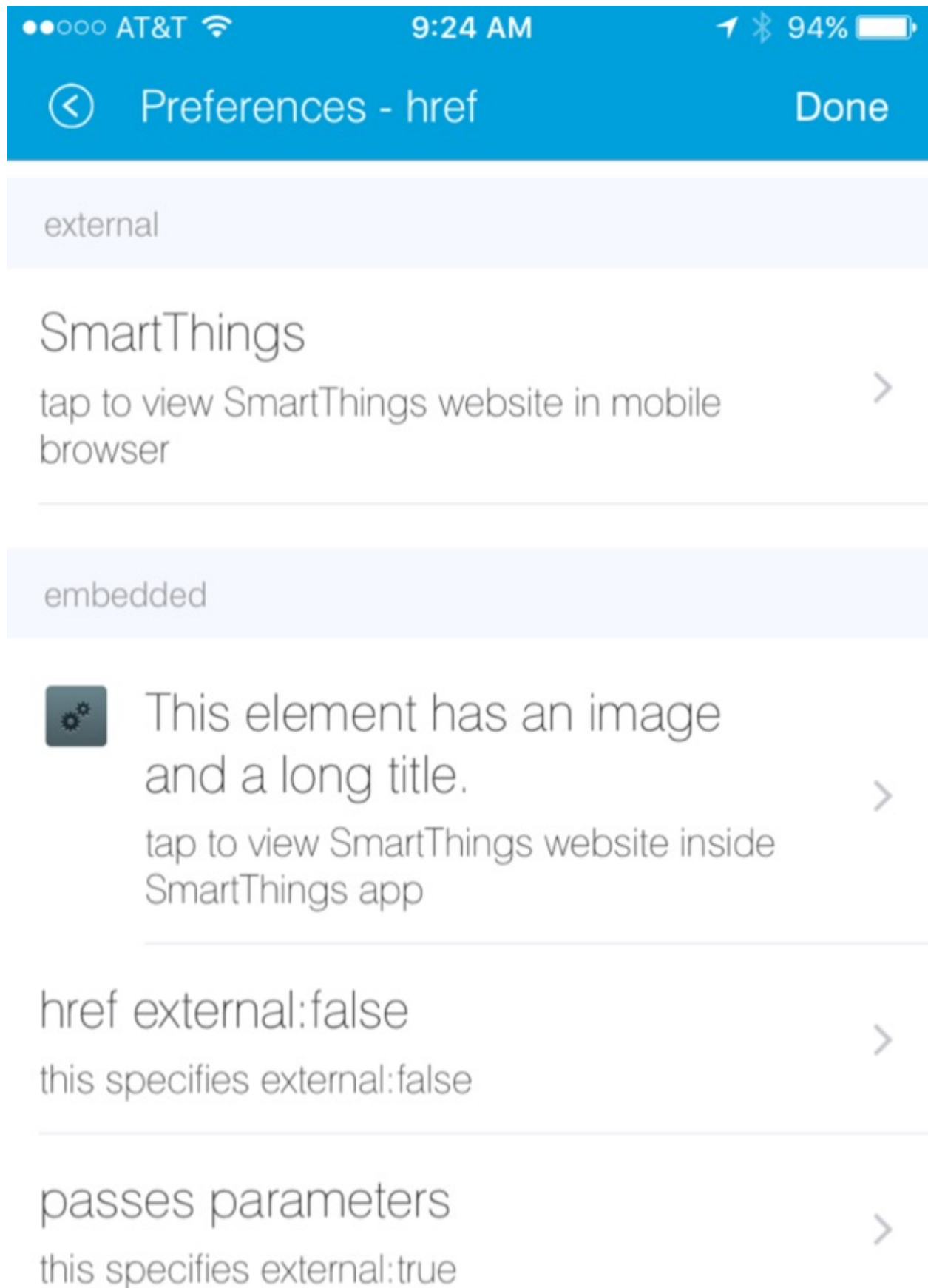
href

A control that selects another preference page or external HTML page.

Example of using href to visit a URL:

```
preferences {
    section("external") {
        href(name: "hrefNotRequired",
            title: "SmartThings",
            required: false,
            style: "external",
            url: "http://smarththings.com/",
            description: "tap to view SmartThings website in mobile browser")
    }
    section("embedded") {
        href(name: "hrefWithImage", title: "This element has an image and a long title.",
            description: "tap to view SmartThings website inside SmartThings app",
            required: false,
            image: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience.png",
            url: "http://smarththings.com/")
    }
}
```

The above preferences would render as:



Example of using href to link to another preference page (dynamic pages are discussed later in this section):

```
preferences {
    page(name: "hrefPage")
    page(name: "deadEnd")
}

def hrefPage() {
    dynamicPage(name: "hrefPage", title: "href example page", uninstall: true) {
        section("page") {
            href(name: "href",
                title: "dead end page",
                required: false,
                page: "deadEnd")
        }
    }
}

def deadEnd() {
    dynamicPage(name: "deadEnd", title: "dead end page") {
        section("dead end") {
            paragraph "this is a simple paragraph element."
        }
    }
}
```

You can use the params option to pass data to dynamic pages:

```
preferences {
    page(name: "firstPage")
    page(name: "secondPage")
}

def firstPage() {
    def hrefParams = [
        foo: "bar",
        someKey: "someVal"
    ]

    dynamicPage(name: "firstPage", uninstall: true) {
        section {
            href(name: "toSecondPage",
                page: "secondPage",
                params: hrefParams,
                description: "includes params: ${hrefParams}")
        }
    }
}

// page def must include a parameter for the params map!
def secondPage(params) {
    log.debug "params: ${params}"
    dynamicPage(name: "secondPage", uninstall: true, install: true) {
        section {
            paragraph "params.foo = ${params?.foo}"
        }
    }
}
```

Valid options:

title String - the title of the element

required Boolean - `true` or `false` to specify this input is required. Defaults to `false`.

description String - the secondary text of the element

external (deprecated - use style instead) Boolean - `true` to open URL in mobile browser application, `false` to open URL within the SmartThings app. Defaults to `false`

style String - Controls how the link will be handled. Specify “external” to launch the link in the mobile device’s browser. Specify “embedded” to launch the link within the SmartThings mobile application. Specify “page” to indicate this is a preferences page.

If `style` is not specified, but `page` is, then `style: "page"` is assumed. If `style` is not specified, but `url` is, then `style: "embedded"` is assumed.

Currently, Android does not support the “external” style option.

url String - The URL of the page to visit. You can use query parameters to pass additional information to the URL (For example, `http://someurl.com?param1=value1¶m2=value1`)

params Map - Use this to pass parameters to other preference pages. If doing this, make sure your page definition method accepts a single parameter (that will be this params map). See the `page-params-by-href` example at the end of this document for more information.

page String - Used to link to another preferences page. Not compatible with the `external` option.

image String - URL of an image to use, if desired.

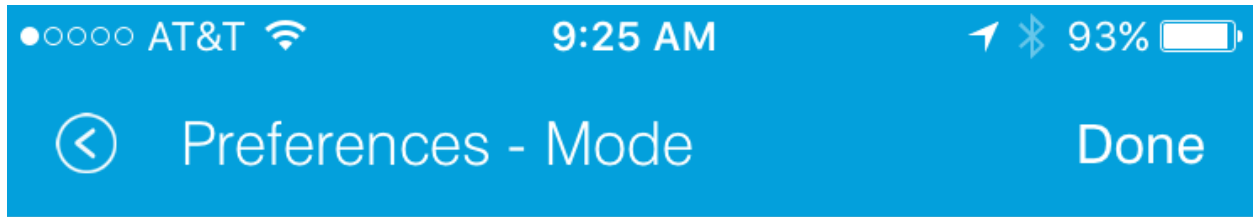
mode

Allows the user to select which modes the app executes in. Automatically generated by single-page preferences.

Example:

```
preferences {
    page(name: "pageOne", title: "page one", nextPage: "pageTwo", uninstall: true) {
        section("section one") {
            paragraph "just some text"
        }
    }
    page(name: "pageTwo", title: "page two") {
        section("page two section one") {
            mode(name: "modeMultiple",
                title: "pick some modes",
                required: false)
            mode(name: "modeWithImage",
                title: "This element has an image and a long title.",
                required: false,
                multiple: false,
                image: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience.png")
        }
    }
}
```

The second page of the above example would render as:



Valid options:

title String - the title of the mode field

required Boolean - `true` or `false` to specify this input is required. Defaults to `false`.

multiple Boolean - `true` or `false` to specify this input allows selection of multiple values. Defaults to `true`.

image String - URL of an image to use, if desired.

Note: There are a couple of different ways to use modes that are worth pointing out. The first way is to use modes as a type of enum input like this:

```
input "modes", "mode", title: "only when mode is", multiple: true, required: false
```

This method will automatically list the defined modes as the options. Keep in mind when using modes in this way that the modes are just data and can be accessed in the SmartApp as such. This does not effect SmartApp execution. In this scenario, it is up to the SmartApp itself to react to the mode changes.

The second example actually controls whether the app is executed based on the modes selected:

```
mode(title: "set for specific mode(s)")
```

Both of these methods of using modes are valid. The impact on SmartApp execution is different in each scenario and it is up to the SmartApp developer to properly label whichever form is used and code the app accordingly.

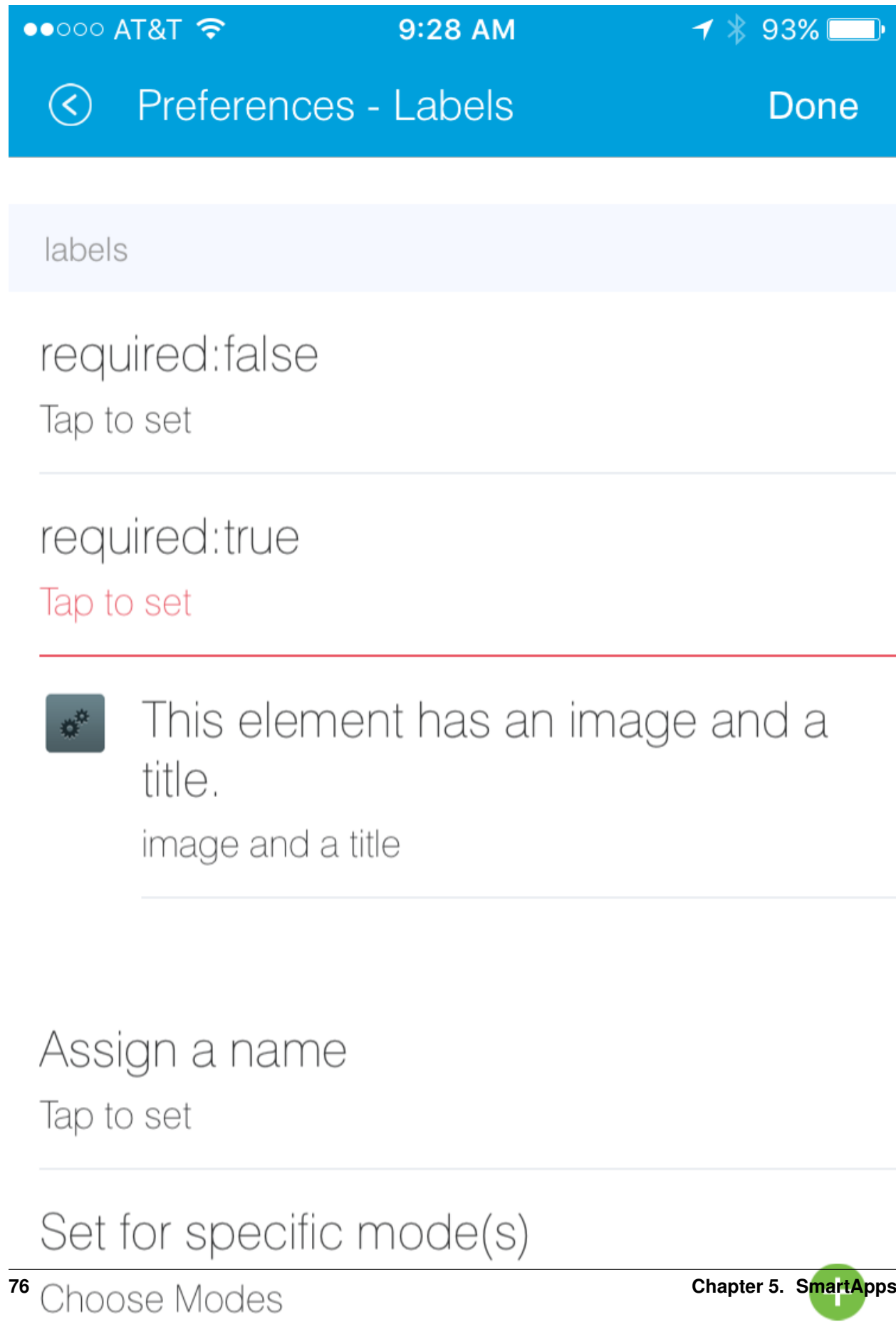
label

Allows the user to name the app installation. Automatically generated by single-page preferences.

Example:

```
preferences {
    section("labels") {
        label(name: "label",
            title: "required:false",
            required: false,
            multiple: false)
        label(name: "labelRequired",
            title: "required:true",
            required: true,
            multiple: false)
        label(name: "labelWithImage",
            title: "This element has an image and a title.",
            description: "image and a title",
            required: false,
            image: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience.png")
    }
}
```

The above preferences definition would render as:



Valid options:

title String - the title of the label field

description String - the default text in the input field

required Boolean - `true` or `false` to specify this input is required. Defaults to `false`. Defaults to `true`.

image String - URL to an image to use, if desired

app

Provides user-initiated installation of child apps. Typically used in dashboard solution SmartApps, which are currently not supported for community development.

input

Allows the user to select devices or enter values to be used during execution of the smart app.

Inputs are the most commonly used preference elements. They can be used to prompt the user to select devices that provide a certain capability, devices of a specific type, or constants of various kinds. Input element method calls take two forms. The “shorthand” form passes in the name and type unnamed as the required first two parameters, and any other arguments as named options:

```
preferences {
    section("section title") {
        // name is "temperature1", type is "number"
        input "temperature1", "number", title: "Temperature"
    }
}
```

The second form explicitly specifies the name of each argument:

```
preferences {
    section("section title") {
        input(name: "color", type: "enum", title: "Color", options: ["Red", "Green", "Blue", "Yellow"])
    }
}
```

Valid input options:

name String - name of variable that will be created in this SmartApp to reference this input

title String - title text of this element.

description String - default value of the input element

multiple Boolean - `true` to allow multiple values or `false` to allow only one value. Not valid for all input types.

required Boolean - `true` to require the selection of a device for this input or `false` to not require selection.

submitOnChange Boolean - `true` to force a page refresh after input selection or `false` to not refresh the page. This is useful when creating a dynamic input page.

options List - used in conjunction with the enum input type to specify the values the user can choose from. Example:
options: ["choice 1", "choice 2", "choice 3"]

type String - one of the names from the following table:

Name	Comment
capabilityName	Prompts for all the devices that match the specified capability. See the <i>Preferences Reference</i> column of the <i>Capabilities Reference</i> (page 217) table for possible values.
deviceName	Prompts for all devices of the specified type.
bool	A true or false value (value returned as a boolean).
boolean	A "true" or "false" value (value returned as a string). It's recommended that you use the "bool" input instead, since the simulator and mobile support for this type may not be consistent, and using "bool" will return you a boolean (instead of a string). The "boolean" input type may be removed in the near future.
decimal	A floating point number, i.e. one that can contain a decimal point
email	An email address
enum	One of a set of possible values. Use the <i>options</i> element to define the possible values.
hub	Prompts for the selection of a hub
icon	Prompts for the selection of an icon image
number	An integer number, i.e. one without decimal point
password	A password string. The value is obscured in the UI and encrypted before storage
phone	A phone number
time	A time of day. The value will be stored as a string in the Java <code>SimpleDateFormat</code> (e.g., "2015-01-09T15:50:32.000-0600")
text	A text value

5.2.7 Dynamic Preferences

One of the most powerful features of multi-page preferences is the ability to dynamically generate the content of a page based on previous selections or external inputs, such as the data elements returned from a web services call. The following example shows how to create a two-page preferences SmartApp where the content of the second page depends on the selections made on the first page.

```

preferences {
    page(name: "page1", title: "Select sensor and actuator types", nextPage: "page2", uninstall: true) {
        section {
            input("sensorType", "enum", options: [
                "contactSensor": "Open/Closed Sensor",
                "motionSensor": "Motion Sensor",
                "switch": "Switch",
                "moistureSensor": "Moisture Sensor"])

            input("actuatorType", "enum", options: [
                "switch": "Light or Switch",
                "lock": "Lock"])
        }
    }

    page(name: "page2", title: "Select devices and action", install: true, uninstall: true) {
}

def page2() {
    dynamicPage(name: "page2") {
        section {

```

```

        input(name: "sensor", type: "capability.$sensorType", title: "If the $sensorType device")
        input(name: "action", type: "enum", title: "is", options: attributeValues(sensorType))
    }
    section {
        input(name: "actuator", type: "capability.$actuatorType", title: "Set the $actuatorType")
        input(name: "action", type: "enum", title: "to", options: actions(actuatorType))
    }
}

private attributeValues(attributeName) {
    switch(attributeName) {
        case "switch":
            return ["on", "off"]
        case "contactSensor":
            return ["open", "closed"]
        case "motionSensor":
            return ["active", "inactive"]
        case "moistureSensor":
            return ["wet", "dry"]
        default:
            return ["UNDEFINED"]
    }
}

private actions(attributeName) {
    switch(attributeName) {
        case "switch":
            return ["on", "off"]
        case "lock":
            return ["lock", "unlock"]
        default:
            return ["UNDEFINED"]
    }
}

```

The previous example shows how you can achieve dynamic behavior between pages. With the `submitOnChange` input attribute you can also have dynamic behavior in a single page.

```

preferences {
    page(name: "examplePage")
}

def examplePage() {
    dynamicPage(name: "examplePage", title: "", install: true, uninstall: true) {

        section {
            input(name: "dimmers", type: "capability.switchLevel", title: "Dimmers",
                description: null, multiple: true, required: false, submitOnChange: true)
        }

        if (dimmers) {
            // Do something here like update a message on the screen,
            // or introduce more inputs. submitOnChange will refresh
            // the page and allow the user to see the changes immediately.
            // For example, you could prompt for the level of the dimmers
            // if dimmers have been selected:

```

```
        section {
            input (name: "dimmerLevel", type: "number", title: "Level to dim lights to...", required: true)
        }
    }
}
```

Note: When a `submitOnChange` input is changed, the whole page will be saved. Then a refresh is triggered with the saved page state. This means that all of the methods will execute each time you change a `submitOnChange` input.

5.2.8 Examples

[page-params-by-href.groovy](#) shows how to pass parameters to dynamic pages using the `href` element.

Almost every SmartApp makes use of preferences to some degree. You can browse them in the IDE under the “Browse SmartApp Templates” menu.

5.3 Storing Data

SmartApps and Device Handlers are all provided a `state` variable that will allow you to store data across executions.

In this guide, you will learn:

- How to store data across executions using the `state` property.
- A basic understanding of how `state` works.
- When `state` may not be the best solution, and what to use instead.

Contents

- *Storing Data* (page 80)
 - *Overview* (page 80)
 - *How it Works* (page 81)
 - *Using State* (page 81)
 - *Atomic State* (page 82)
 - *Examples* (page 83)

5.3.1 Overview

Recall that SmartApps (and Device Handlers) are not always running, but rather execute according to a certain schedule or in response to events being triggered. But what if we need our application to retain some information between executions? We can use `state` to persist data across executions.

Here’s a quick example showing how to work with state:

```
state.myData = "some data"
log.debug "state.myData = ${state.myData}"

state.myCounter = state.myCounter + 1
```


5.3.2 How it Works

Each executing instance of a SmartApp or Device Handler has access to a simple, map-like storage mechanism through the `state` property. When an application executes, it populates the `state` property from the backing store. The application can then interact with it, through simple map-like operations.

When the application is finished executing, the values in `state` are written back to persistent storage.

Important: When an application stores data in `state`, or reads from it, it is only modifying (or querying) the local `state` instance variable within the running SmartApp or Device Handler. Only when the application is done executing are the values written to persistent storage.

The contents of `state` are stored as a string, in JSON format. This means that anything stored in `state` must be serializable to JSON.

Tip: State is stored in JSON format; for most data types this works fine, but for more complex object types this may cause issues.

This is particularly worth noting when working with dates. If you need to store time information, consider using an epoch time stamp, conveniently available via the `now()` method:

```
def installed() {
    state.installedAt = now()
}

def someEventHandler(evt) {
    def millisSinceInstalled = now() - state.installedAt
    log.debug "this app was installed ${millisSinceInstalled / 1000} seconds ago"

    // you can also create a Date object back from epoch time:
    log.debug "this app was installed at ${new Date(state.installedAt)}"
}
```

5.3.3 Using State

You can use `state` to store strings, numbers, lists, booleans, maps, etc. To use `state`, simply use the `state` variable that is injected into every SmartApp and Device Handler. You can think of it as a map that will persist its value across executions.

As usual, the best way to describe code is by showing code itself.

```
def installed() {
    // simple number to keep track of executions
    state.count = 0

    // we can store maps in state
    state.myMap = [foo: "bar", baz: "fee"]

    // booleans are ok of course
    state.myBoolean = true

    // we can use array index notation if we want
    state['key'] = 'value'

    // we can store lists and maps, so we can make some interesting structures
    state.myListOfMaps = [[key1: "vall", bool1: true],
```

```
        [otherKey: ["string 1", "string 2"]]]
    }

    def someEventHandler(evt) {

        // increment by 1
        state.count = state.count + 1

        log.debug "this event handler has been called ${state.count} times since installed"

        log.debug "state.myMap.foo: ${state.myMap.foo}" // => prints "bar"

        // we can access state value using array notation if we wish
        log.debug "state['myBoolean']: ${state['myBoolean']}"

        // we can navigate our list of maps
        state.myListOfMaps.each { map ->
            log.debug "entry: $map"
            map.each {
                log.debug "key: ${it.key}, value: ${it.value}"
            }
        }
    }
}
```

5.3.4 Atomic State

Note: Atomic State is currently only available for SmartApps. Device Handlers do not support Atomic State.

Since `state` is initialized from persistent storage when a SmartApp executes, and is written to storage only when the application is done executing, there is the possibility that another execution *could* happen within that time window, and cause the values stored in `state` to appear inconsistent.

Consider the scenario of a SmartApp that keeps a counter of executions. Each time the SmartApp executes, it increments the counter by 1. Assume that the initial value of `state.counter` is 0.

1. An execution (“Execution 1”) occurs, and increments `state.counter` by one:

```
state.counter = state.counter + 1 // counter == 1
```

2. Another execution (“Execution 2”) occurs *before* “Execution 1” has finished. It reads `state.counter` and increments it by one.

```
state.counter = state.counter + 1 // counter == 1!!!
```

Because “Execution 1” hasn’t finished executing by the time that “Execution 2” begins, the value of `counter` is still 0!

Additionally, because the contents of `state` are only persisted when execution is complete, it’s also possible to inadvertently overwrite values (last finished execution “wins”).

To avoid this type of scenario, you can use `atomicState`. `atomicState` writes to the data store when a value is *set*, and reads from the data store when a value is *read* - not just when the application execution initializes and completes. You use it just as you would use `state`:

```
atomicState.counter = atomicState.counter + 1.
```

Important: Using `atomicState` instead of `state` incurs a higher performance cost, since external storage is

touched on read and write operations, not just when the application is initialized or done executing.

Use `atomicState` only if you are sure that using `state` will cause problems.

It's also worth noting that you should **not** use both `state` and `atomicState` in the same SmartApp. Doing so will likely cause inconsistencies in state values.

5.3.5 Examples

Here are some SmartApps that make use of state. You can find them in the IDE along with the other example SmartApps.

- “Smart Nightlight” - shows using state to store time information.
- “Laundry Monitor” - uses state to store boolean state and time information.
- “Good Night” - shows using state to store time information, including constructing a `Date` object from a value stored in state.

5.4 Events and Subscriptions

Turn on a light when a door opens. Turn the lights off at sunrise. Send a message if a door opens when you're not home. These are all examples of event-handler SmartApps. They follow a common pattern - subscribe to some event, and take action when the event happens.

This section will discuss events and how you can subscribe to them in your SmartApp.

5.4.1 Subscribe to Specific Device Events

The most common use case for event subscriptions is for device events:

subscribe(deviceName, “eventToSubscribeTo”, handlerMethodName)

```
preferences {
    section {
        input "theSwitch", "capability.switch"
    }
}

def install() {
    subscribe(theSwitch, "switch.on", switchOnHandler)
}

def switchOnHandler(evt) {
    log.debug "switch turned on!"
}
```

The handler method must accept an event parameter.

Refer to the [Event](#) (page 315) API documentation for more information about the Event object.

You can find the possible events to subscribe to by referring to the Attributes column for a capability in the [Capabilities Reference](#) (page 217). The general form we use is “<attributeName>.<attributeValue>”. If the attribute does not have any possible values (for example, “battery”), you would just use the attribute name.

In the example above, the switch capability has the attribute “switch”, with possible values “on” and “off”. Putting these together, we use “switch.on”.

5.4.2 Subscribe to All Device Events

You can also subscribe to all states by just specifying the attribute name:

```
subscribe(theSwitch, "switch", switchHandler)

def switchHandler(evt) {
    if (evt.value == "on") {
        log.debug "switch turned on!"
    } else if (evt.value == "off") {
        log.debug "switch turned off!"
    }
}
```

In this case, the `switchHandler` method will be called for both the “on” and “off” events.

5.4.3 Subscribe to Multiple Devices

If your SmartApp allows multiple devices, you can subscribe to events for all the devices:

```
preferences {
    section {
        input "switches", "capability.switch", multiple: true
    }
}

def installed() {
    subscribe(switches, "switch", switchesHandler)
}

def switchesHandler(evt) {
    log.debug "one of the configured switches changed states"
}
```

5.4.4 Subscribe to Location Events

In addition to subscribing to device events, you can also subscribe to events for the user’s location.

You can subscribe to the following location events:

mode Triggered when the mode changes.

position Triggered when the geofence position changes for this location. Does not get triggered when the fence is widened or narrowed - only fired when the position changes.

sunset Triggered at sunset for this location.

sunrise Triggered at sunrise for this location.

sunriseTime Triggered around sunrise time. Used to get the time of the next sunrise for this location.

sunsetTime Triggered around sunset time. Used to get the time of the next sunset for this location.

Pass in the location property automatically injected into every SmartApp as the first parameter to the subscribe method.

```
subscribe(location, "mode", modeChangeHandler)

// shortcut for mode change handler
subscribe(location, modeChangeHandler)
```

```

subscribe(location, "position", positionChange)
subscribe(location, "sunset", sunsetHandler)
subscribe(location, "sunrise", sunriseHandler)
subscribe(location, "sunsetTime", sunsetTimeHandler)
subscribe(location, "sunriseTime", sunriseTimeHandler)

```

Refer to the [Sunset](#) and [Sunrise](#) section for more information about sunrise and sunset.

5.4.5 The Event Object

Event-handler methods must accept a single parameter, the event itself.

Refer to the [Event](#) (page 315) API documentation for more information.

A few of the common ways of using the event:

```

def eventHandler(evt) {
    // get the event name, e.g., "switch"
    log.debug "This event name is ${evt.name}"

    // get the value of this event, e.g., "on" or "off"
    log.debug "The value of this event is ${evt.value}"

    // get the Date this event happened at
    log.debug "This event happened at ${evt.date}"

    // did the value of this event change from its previous state?
    log.debug "The value of this event is different from its previous value: ${evt.isStateChange()}"
}

```

Note: The contents of each Event instance will vary depending on the exact event. If you refer to the Event reference documentation, you will see different value methods, like “floatValue” or “dateValue”. These may or may not be populated depending on the specific event, and may even throw exceptions if not applicable.

5.4.6 See Also

- [Sunset and Sunrise](#)
- [Event](#) (page 315) API Documentation
- [Location](#) (page 325) API Documentation
- [Interacting with Devices](#)

5.5 Devices

SmartApps almost always interact with devices. We often need to get information about a specific device (is this switch on?), or send a device a command (turn this switch off).

5.5.1 Device Overview

Devices are the “things” that SmartApps interact with. Devices may support one or many capabilities.

Capabilities represent the things a device knows (attributes) and the things they can do (commands). They are an abstraction that allows us to work with many different manufacturer's devices transparently.

To build a flexible SmartApp, we should write our SmartApp to work with any device that supports a given capability. We don't want to write a SmartApp that only works with a specific manufacturer's switch for example. We want to write an app that works with any device that supports the switch capability.

5.5.2 Preferences - Selecting the Devices

To allow the user to select devices that support a given capability, we use the preferences input element:

```
preferences {
    section {
        input "presenceSensors", "capability.presenceSensor"
    }
}
```

The above example will allow the user to select any device that supports the presence sensor capability. This could be a mobile phone, or a [SmartSense presence sensor](#). We don't care about the specific device - we just declare we want a device that supports the presence sensor capability.

You can refer to the [Capabilities Reference](#) (page 217) for information on all the supported capabilities. The “Preferences Reference” column tells you what to use in your preferences for a given capability.

5.5.3 Interacting with Devices

After you have declared the devices your SmartApp needs to interact with, a Device object instance will be available in your SmartApp, with the name that you provided.

```
preferences {
    section {
        input "theSwitch", "capability.switch"
    }
}

def someEventHandler(evt) {
    theSwitch.on()
}
```

5.5.4 Device Attributes

Attributes represent the state of a device. A device that supports the “temperatureMeasurement” capability has a “temperature” attribute, for example.

Attributes have state - the “temperature” attribute has an associated [State](#) (page 329) object that contains information about the temperature (its value, the date it was recorded, etc.)

5.5.5 Device Commands

Devices may expose one or many commands. Commands are the things that devices can do. A switch supports the “on” and “off” commands, that turn the switch “on” and “off”, respectively.

Not all devices have commands. Commands typically perform some sort of physical actuation (turn a switch on, or unlock a lock, for example). A humidity sensor has nothing to physically actuate, for example.

5.5.6 Getting Device Current Values

You can retrieve information about a device's current state in a few ways.

The `currentState` method and `<attributeName>State` properties both return a *State* (page 329) object representing the current state of this device.

```
preferences {
    section {
        input "tempSensor", "capability.temperatureMeasurement"
    }
}

def someEventHandler(evt) {

    def currentState = tempSensor.currentState("temperature")
    log.debug "temperature value as a string: ${currentState.value}"
    log.debug "time this temperature record was created: ${currentState.date}"

    // shortcut notation - temperature measurement capability supports
    // a "temperature" attribute. We then append "State" to it.
    def anotherCurrentState = tempSensor.temperatureState
    log.debug "temperature value as an integer: ${anotherCurrentState.integerValue}"
}
```

You can get the current value directly by using the `currentValue(attributeName)` and its shortcut, `current<Uppercase attribute name>`:

```
preferences {
    section {
        input "myLock", "capability.lock"
    }
}

def someEventHandler(evt) {
    def currentValue = myLock.currentValue("lock")
    log.debug "the current value of myLock is $currentValue"

    // Lock capability has "lock" attribute.
    // <deviceName>.current<uppercase attribute name>:
    def anotherCurrentValue = myLock.currentLock
    log.debug "the current value of myLock using shortcut is: $anotherCurrentValue"
}
```

5.5.7 Querying Event History

To get a list of events in reverse chronological order (newest first), use the `events()` method:

```
// returns the last 10 by default
myDevice.events()

// use the max option to get more results
myDevice.events(max: 30)
```

To get a list of events in reverse chronological order (newest first) since a given date, use the `eventsSince` method:

```
// get all events for this device since yesterday (maximum of 1000 events)
myDevice.eventsSince(new Date() - 1)

// get the most recent 20 events since yesterday
myDevice.eventsSince(new Date() - 1, [max: 20])
```

To get a list of events between two dates, use the `eventsBetween` method:

```
// get all events between two days ago and yesterday (up to 1000 events)
// returned events sorted in inverse chronological order (newest first)
myDevice.eventsBetween(new Date() - 2, new Date() - 1)

// get the most recent 50 events in the last week
myDevice.eventsBetween(new Date() - 7, new Date(), [max: 50])
```

Similar date-constrained methods exist for getting State information for a device.

Refer to the full *Device* (page 304) API documentation for more information.

5.5.8 Sending Commands

SmartApps often need to send commands to a device - tell a switch to turn on, or a lock to unlock, for example.

The commands available to your device will vary by device. You can refer to the *Capabilities Reference* (page 217) to see the available commands for a given capability.

Sending a command is as simple as calling the command method on the device:

```
myLock.lock()
myLock.unlock()
```

Some commands may expect parameters. All commands can take an optional map parameter, as the last argument, to specify delay time in milliseconds to wait before the command is sent to the device:

```
// wait two seconds before sending on command
mySwitch.on([delay: 2000])
```

Note: Because specific devices *can* provide more commands than its supported capabilities, it is possible to have more available commands than the capability declares. As a best practice, you should write your SmartApp to the capabilities specification, and not to any specific device. If, however, you are writing a SmartApp for a very specific case, and are willing to forgo the flexibility, you may make use of this ability.

5.5.9 Interacting with Multiple Devices

If you specified `multiple:true` in your device preferences, the user may have selected more than one device. Your device instance will refer to a list of objects if this is the case.

You can send commands to all the devices without needing to iterate over each one:

```
preferences {
    section {
        input "switches", "capability.switch", multiple: true
    }
}
```



```
def someEventHandler(evt) {
    log.debug "will send the on() command to ${switches.size()} switches"
    switches.on()
}
```

You can also retrieve state and event history for multiple devices, using the methods discussed above. Instead of single values or objects, they will return a list of values or objects.

Here's a simple example of getting all switch state values and logging the switches that are on:

```
preferences {
    section {
        input "switches", "capability.switch", multiple: true
    }
}

def someEventHandler(evt) {
    // returns a list of the values for all switches
    def currSwitches = switches.currentSwitch

    def onSwitches = currSwitches.findAll { switchVal ->
        switchVal == "on" ? true : false
    }

    log.debug "${onSwitches.size()} out of ${switches.size()} switches are on"
}
```

5.5.10 See Also

- [Capabilities Reference](#) (page 217)
- Preferences and Settings
- Events and Subscriptions
- [Device](#) (page 304) API Documentation
- [Event](#) (page 315) API Documentation
- [State](#) (page 329) API Documentation

5.6 Modes

SmartThings allows users to specify that SmartApps only execute when in certain *modes*.

In this chapter, you will learn:

- What modes are, and how they are used
- How to get and set the current mode
- How to configure your SmartApp to allow a user to select a mode

5.6.1 Overview

Modes can be thought of as behavior filters for the smart home. Users can change how things act or behave based on the mode you're in. For example:

- When in “Home” mode, motion should turn on a light.
- When in “Away” mode, motion should send a text message and turn on an alarm.

SmartThings comes with a few pre-configured modes, such as “Home”, “Away”, and “Night”. Users can also create their own modes for each location.

5.6.2 Getting the Current Mode

You can get the current mode by using the `mode` or `currentMode` property on the `location` in a SmartApp:

```
def currMode = location.mode // "Home", "Away", etc.
log.debug "current mode is $currMode"

def anotherWay = location.currentMode
log.debug "current mode is $anotherWay"
```

5.6.3 Getting all Modes

You can get a list of all the modes for the location the SmartApp is installed into:

```
def allModes = location.modes // ex: [Home, Away, Night]
log.debug "all modes for this location: $allModes"
```

5.6.4 Setting the Mode

You can use `setLocationMode()` or `location.setMode()` to set the mode for the location:

```
setLocationMode("Away")
```

```
location.setMode("Away")
```

These methods will raise an error if the mode specified does not exist for the location.

5.6.5 Allowing Users to Select Modes

In the SmartApp preferences block, you can specify that the user select a mode by using the “mode” input type:

```
input "modes", "mode", title: "select a mode(s)", multiple: true
```

This will allow the user to select a mode (or multiple modes), and the SmartApp can then vary its behavior based upon the mode(s) selected.

You can also use the `mode()` method to allow a user to select a mode that this SmartApp will execute for:

```
mode(title: "Set for specific mode(s)")
```

The SmartApp will then only execute when in the selected mode, without any action needed by the developer to determine the correct mode.

You can learn more about the various ways to allow a user to select a mode [here](#) (page 73).

5.6.6 Mode Events

You can listen for a mode change by subscribing to the "mode" on the location object:

```
def installed() {
    subscribe(location, "mode", modeChangeHandler)
}

def modeChangeHandler(evt) {
    log.debug "mode changed to ${evt.value}"
}
```

In the example above `modeChangeHandler()` will be called whenever the mode changes for the location this SmartApp is installed into.

5.6.7 Example

The following example is a simplified version of the “Scheduled Mode Change” SmartApp. You can view the SmartApp in the IDE templates for the full example.

This example shows how to use the "mode" input type to ask the user to select a mode, and then (based on the user-defined schedule), changes the mode as specified.

```
preferences {
    section("At this time every day") {
        input "time", "time", title: "Time of Day"
    }
    section("Change to this mode") {
        input "newMode", "mode", title: "Mode?"
    }
}

def installed() {
    initialize()
}

def updated() {
    unschedule()
    initialize()
}

def initialize() {
    schedule(time, changeMode)
}

def changeMode() {
    log.debug "changeMode, location.mode = $location.mode, newMode = $newMode, location.modes = $location.modes"

    if (location.mode != newMode) {
        if (location.modes?.find{it.name == newMode}) {
            setLocationMode(newMode)
        } else {
            log.warn "Tried to change to undefined mode '${newMode}'"
        }
    }
}
```

In the `changeMode()` method above, there are a few things worth calling out.

First, notice we first check if we are already in the mode specified - if we are, we don't do anything:

```
if (location.mode != newMode)
```

If we do need to change the mode, we first verify that the mode actually exists. This ensures that we don't try and set the mode to one that does not exist for the location.

```
if (location.modes?.find{it.name == newMode})
```

Tip: Note the use of the `?` operator above. This Groovy operator is referred to as the *Safe Navigation* operator, and it allows us to avoid a `NullPointerException` that might occur if `location.modes` returned null, in the example above. If `location.modes` did evaluate to null, the rest of the statement simply wouldn't execute.

If you come from a Java background, you might be used to writing the above with an `if` check for `null`. The `?` operator allows you to accomplish the same task, without cluttering your code with such boilerplate instructions.

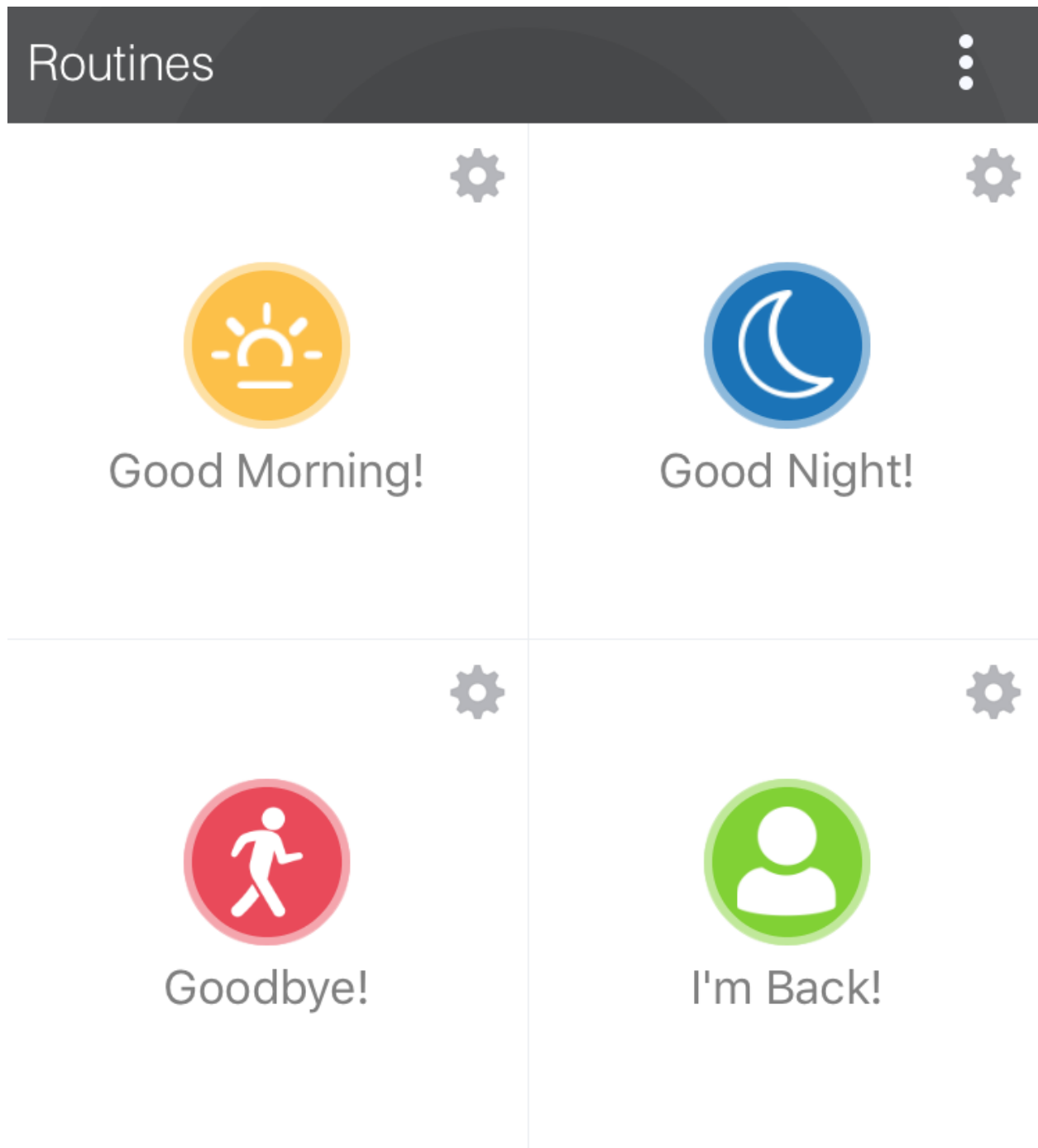
You can read more about this operator [here](#).

5.6.8 Further Reading

- *Mode Input* (page 73)
- *Location Object* (page 325)
- *Mode Object* (page 329)

5.7 Routines

Routines (or *Hello Home Actions* in older mobile apps) allow certain things to happen when the routine is invoked.



In this chapter, you will learn:

- What Routines are
- How to get the available Routines for a location
- How to execute Routines in a SmartApp

5.7.1 Overview

Routines allow for certain things to happen whenever it executes. SmartThings comes with a few routines already installed:

- Good Morning! - You or the house is waking up
- Good Night! - You or the house is going to sleep
- Goodbye! - You're leaving the house
- I'm Back! - You've returned to the house

Each routine can be configured to do certain things. For example, when "I'm Back!" executes, you can set the mode to "Home", unlock doors, adjust the thermostat, etc.

Routines exist for each location in a SmartThings account.

5.7.2 Get Available Routines

You can get the routines for the location the SmartApp is installed into by accessing the `helloHome` object on the `location`:

```
def actions = location.helloHome?.getPhrases()*.label
```

Tip: If the above code example, with the `?` and `*` operator looks foreign to you, read on.

The `?` operator allows us to safely avoid a `NullPointerException` should `helloHome` be `null`. It's one of Groovy's niceties that allows us to avoid wrapping calls in `if(something != null)` blocks. Read more about it [here](#).

The `*` operator is called the *spread operator*, and it invokes the specified action (get the label, in the example above) on all items in a collection, and collects the result into a list. Read more about it [here](#).

5.7.3 Execute Routines

To execute a Routine, you can call the `execute()` method on `helloHome`:

```
location.helloHome?.execute("Good Night!")
```

5.7.4 Allowing Users to Select Routines

A SmartApp may want to allow a user to execute certain Routines in a SmartApp. Since the routines for each location will vary, we need to get the available routines, and use them as options for an `enum` input type.

This needs to be done in a dynamic preferences page, since we need to execute some code to populate the available actions:

```

preferences {
    page(name: "selectActions")
}

def selectActions() {
    dynamicPage(name: "selectActions", title: "Select Hello Home Action to Execute", install: true,

        // get the available actions
        def actions = location.helloHome?.getPhrases()*.label
        if (actions) {
            // sort them alphabetically
            actions.sort()
            section("Hello Home Actions") {
                log.trace actions
                // use the actions as the options for an enum input
                input "action", "enum", title: "Select an action to execute", options: actions
            }
        }
    }
}

```

You can read more about the enum input type and dynamic pages [here](#) (page 59).

You can then access the selected phrase like so:

```
def selectedAction = settings.action
```

5.7.5 Example

This example simply shows executing a selected routine when a switch turns on, and another action when a switch turns off:

```

preferences {
    page(name: "configure")
}

def configure() {
    dynamicPage(name: "configure", title: "Configure Switch and Phrase", install: true, uninstall: true,

        section("Select your switch") {
            input "theswitch", "capability.switch", required: true
        }

        def actions = location.helloHome?.getPhrases()*.label
        if (actions) {
            actions.sort()
            section("Hello Home Actions") {
                log.trace actions
                input "onAction", "enum", title: "Action to execute when turned on", options: actions
                input "offAction", "enum", title: "Action to execute when turned off", options: actions
            }
        }
    }
}

def installed() {
    log.debug "Installed with settings: ${settings}"
}

```

```
    initialize()
}

def updated() {
    log.debug "Updated with settings: ${settings}"
    unsubscribe()
    initialize()
}

def initialize() {
    subscribe(theswitch, "switch", handler)
    log.debug "selected on action $onAction"
    log.debug "selected off action $offAction"
}

def handler(evt) {
    if (evt.value == "on") {
        log.debug "switch turned on, will execute action ${settings.onAction}"
        location.helloHome?.execute(settings.onAction)
    } else {
        log.debug "switch turned off, will execute action ${settings.offAction}"
        location.helloHome?.execute(settings.offAction)
    }
}
```

5.7.6 Further Reading

- *Preferences and Settings Guide* (page 59)

5.8 Scheduling

Topics

- *Scheduling* (page 96)
 - *Schedule From Now* (page 97)
 - *Run Once in the Future* (page 97)
 - *Run on a Schedule* (page 98)
 - *Other Scheduling-related Methods* (page 100)
 - *Scheduling Limitations, Best Practices, and Things Good to Know* (page 101)
 - *Examples* (page 102)

SmartApps often have the need to schedule certain actions to take place at a given point in time. For example, an app may want to turn off the lights five minutes after someone leaves. Or maybe an app wants to turn the lights on every day at a certain time.

Broadly speaking, there are a few different ways we might want to schedule something to happen:

- Do something after a certain time amount from now.
- Do something once at certain time in the future.
- Do something on a recurring schedule.

We'll look at each scenario in detail, and what methods SmartThings makes available to address these requirements.

5.8.1 Schedule From Now

A SmartApp may want to take some action in a certain amount of time after some event has occurred. Consider a few examples:

- When a door closes, turn a light off after two minutes.
- When everyone leaves, adjust the thermostat after ten minutes.
- If a door opens and is not shut after five minutes, send a notification.

All these scenarios follow a common pattern: when a certain event happens, take some action after a given amount of time. This can be accomplished this by using the `runIn` method.

runIn(seconds, method, options)

Executes the specified method in the specified number of seconds from now:

```
def someEventHandler(evt) {
    // execute handler in five minutes from now
    runIn(60*5, handler)
}

def handler() {
    switch.off()
}
```

options Optional. Map of options. Valid options:

[overwrite: true or false]

By default, if a method is scheduled to run in the future, and then another call to `runIn` with the same method is made, the last one overwrites the previously scheduled method. This is usually preferable. Consider the situation if we have a switch scheduled to turn off after five minutes of a door closing. First, the door closes at 2:50 and we schedule the switch to turn off after five minutes (2:55). Then two minutes later (2:52), the door opens and closes again - another call to `runIn` will be made to schedule the switch to turn off in five minutes from now (2:57). If we specified [overwrite: false], we'd now have two schedules to turn off the switch - one at 2:55, and one at 2:57. So, if you do specify [overwrite: false], be sure to write your handler so that it can handle multiple calls.

```
def someEventHandler(evt) {
    runIn(300, handler, [overwrite: false])
}

def handler() {
    // need to handle multiple calls since overwrite:false specified
}
```

Note: It is important to note that you should not rely on the method being called in *exactly* the specified number of seconds. SmartThings will *attempt* to execute the method within a minute of the time specified, but cannot guarantee it. See the [Scheduling Limitations, Best Practices, and Things Good to Know](#) (page 101) topic below for more information.

5.8.2 Run Once in the Future

Some SmartApps may need to schedule certain actions to happen *once* at a specific time and date. *runOnce* handles this case.

Note: You may notice that some of the scheduling APIs accept a string to represent the the date/time to be executed. This is a result of when you define a preference input of the “time” type, it uses a String representation of the value entered. When using this value later to set up a schedule, the APIs need to be able to handle this type of argument.

When simply using the input from preferences, you don’t need to know the details of the specific date format being used. But, if you wish to use the APIs with string inputs directly, you will need to understand their expected format.

SmartThings uses the Java standard format of “yyyy-MM-dd’T’HH:mm:ss.SSSZ”. More technical readers may recognize this format as ISO-8601 (Java does not fully conform to this format, but it is very similar). Full discussion of this format is beyond the scope of this documentation, but a few examples may help:

January 09, 2015 3:50:32 GMT-6 (Central Standard Time): “2015-01-09T15:50:32.000-0600”

February 09, 2015 3:50:32:254 GMT-6 (Central Standard Time): “2015-02-09T15:50:32.254-0600”

For more information about date formatting, you can review the [SimpleDateFormat JavaDoc](#).

runOnce(dateTime, handlerMethod, options)

Executes the handlerMethod once at the specified date and time. The dateTime argument can be either a Date object or a date string.

options Optional. Map of options. Valid options:

[overwrite: true or false]

Specify [overwrite: false] if you do not want the most recently created job for the handlerMethod to overwrite an existing job. See the discussion in the runIn documentation above for more information.

```
def someEventHandler(evt) {  
    // execute handler tomorrow, at the current time  
    runOnce(new Date() + 1, handler)  
}  
  
def handler() {  
    switch.off()  
}
```

```
def someEventHandler(evt) {  
    // execute handler at 4 PM CST on October 21, 2015 (e.g., Back to the Future 2 Day!)  
    runOnce("2015-10-21T16:00:00.000-0600", handler)  
}  
  
def handler() {  
    // do something awesome, like ride a hovercraft  
}
```

5.8.3 Run on a Schedule

Oftentimes, there is a need to schedule a job to run on a specific schedule. For example, maybe you want to turn the lights off at 11 PM every night. SmartThings provides the *schedule* method to allow you to create recurring schedules.

The various *schedule* methods follow a similar form - they take an argument representing the desired schedule, and the method to be called on this schedule. Each SmartApp or device-type handler can only have one handler method

scheduled at any time. This means that, unlike *runIn* or *runOnce*, a job created with *schedule* must either execute or be canceled with the *unschedule* method before you can schedule another job with the same method. The *schedule* method does not accept the overwrite option like *runOnce* and *runIn*.

schedule(dateTime, handlerMethod)

Creates a scheduled job that calls the handlerMethod every day at the time specified by the dateTime argument. The dateTime argument can be a String, Date, or number (to schedule based on Unix epoch time).

Only the time information will be used to derive the recurring schedule.

Here's how you might use a preference to set up a daily scheduled job:

```
preferences {
    section("Time to run") {
        input "time1", "time"
    }
}

...

def someEventHandler(evt) {
    schedule(time1, handlerMethod)
}

def handlerMethod() {
    ...
}
```

Of course, you can create and pass the dateTime string explicitly:

```
def someEventHandler(evt) {
    // call handlerMethod every day at 3:36 PM CST
    schedule("2015-01-09T15:36:00.000-0600", handlerMethod)
}

def handlerMethod() {
    ...
}
```

You can also pass a Groovy Date object:

```
def someEventHandler(evt) {
    // call handlerMethod every day at the current time
    schedule(new Date(), handlerMethod)
}

def handlerMethod() {
    ...
}
```

Finally, you can pass a Long representing the desired time in milliseconds (using [Unix time](#)) to schedule:

```
def someEventHandler(evt) {
    // call handlerMethod every day, at two minutes from the current time
    schedule(now() + 120000, handlerMethod)
}

def handlerMethod() {
    ...
}
```

Scheduling jobs to execute at a particular time is useful, but what if we want to execute a job at some other interval? What if, for example, we want a method to execute at fifteen minutes past the hour, every hour?

SmartThings allows you to pass a cron expression to the `schedule` method to accomplish this. A cron expression is based on the cron UNIX tool, and is a way to specify a recurring schedule. They are extremely powerful, but can be pretty confusing.

`schedule(cronExpression, handlerMethod)`

Creates a scheduled job that calls the `handlerMethod` according to the specified `cronExpression`.

Note: Full documentation of the cron expression format can be found in the [Quartz Cron Trigger Tutorial](#).

```
def someEventHandler(evt) {
    // execute handlerMethod every hour on the half hour.
    schedule("0 30 * * * ?", handlerMethod)
}

def handlerMethod() {
    ...
}
```

Scheduled jobs are limited to running no more often than once per minute.

In addition to creating schedules using cron expressions, SmartThings also provides some convenience methods to set up the schedule for you. They are in the form of *`runEveryXMinutes(handlerMethod)`* or *`runEveryXHours(handlerMethod)`*.

These methods work by creating a random start time in the X minutes or hours, and then every X minutes or hours after that. For example, `runEvery5Minutes(handlerMethod)` will execute `handlerMethod` at a random time in the next five minutes, and then run every five minutes from then.

These methods have the advantage of randomizing the start time for schedules, which can reduce the load on the SmartThings cloud. As such, they should be preferred over cron expressions when available.

The currently available methods are:

`runEvery5Minutes(handlerMethod)`

`runEvery10Minutes(handlerMethod)`

`runEvery15Minutes(handlerMethod)`

`runEvery30Minutes(handlerMethod)`

`runEvery1Hour(handlerMethod)`

`runEvery3Hours(handlerMethod)`

5.8.4 Other Scheduling-related Methods

`canSchedule()`

returns `true` if a job can be scheduled, `false` otherwise. Only four jobs may be scheduled for the future at any time.

```
def someEventHandler(evt) {
    runIn(300, someHandlerMethod1)
    runIn(300, someHandlerMethod2)
    runIn(300, someHandlerMethod3)
}
```

```
runIn(300, someHandlerMethod4)

// false, since we already have four jobs scheduled
canSchedule()
}
```

unschedule(nameOfMethod = “)

Removes the method from the schedule queue, if specified. Note that this is not specific to jobs created with any of the `schedule` methods - any job scheduled for the future (using `runIn`, `runOnce`, or any of other scheduling methods) may be canceled using `unschedule`.

Note: Due to the way that the scheduling service is currently implemented, `unschedule` is a fairly expensive operation, and may take many seconds to execute.

We plan to address this in the future, but until then, you should be aware of the potential performance impacts.

```
// unschedule the someHandlerMethod
unschedule("someHandlerMethod")
```

`unschedule` can also be called with no arguments to unschedule all jobs.

```
// unschedule all jobs
unschedule()
```

5.8.5 Scheduling Limitations, Best Practices, and Things Good to Know

When using any of the scheduling APIs, it's important to understand some limitations and best practices. These limitations are due in part to the fact that execution occurs in the cloud, and are thus subject to limiting factors like load, network connectivity, etc.

Do not expect exact execution time in scheduled jobs

SmartThings will *try* to execute your scheduled job at the specified time, but cannot guarantee it will execute at that exact moment. As a general rule of thumb, you should expect that your job will be called within the minute of scheduled execution. For example, if you schedule a job at 5:30:20 (20 seconds past 5:30) to execute in five minutes, we expect it to be executed at some point in the 5:35 minute.

When using `runIn` with values less than one minute, your mileage will vary.

Do not use `runIn` to set up a recurring schedule of less than sixty seconds

You may have noticed that none of the `schedule` APIs allow you to schedule jobs for less than sixty second intervals. You may be tempted to work around this limitation by using `runIn` to create such a schedule (i.e., a handler method that reschedules itself). This is discouraged, and at some point may be prevented by the SmartThings framework.

The primary reason for discouraging jobs that run more often than every sixty seconds is overall system resource utilization. Using `runIn` to circumvent this is problematic because any failure to execute, even once, will cause the scheduled event to stop triggering.

Only four jobs may be scheduled at any time

To prevent any one SmartApp or device-type handler from using too many resources, only four jobs may be scheduled for future execution at any time.

Do not excessively schedule/poll

While there are some limitations in place to prevent excessive scheduling, it's important to note that excessive polling or scheduling is discouraged. It is one of the items we look for when reviewing community-developed SmartApps or device-type handlers.

Missed job executions will not accumulate

Due to a variety of issues (perhaps the local Internet connection has been dropped, or there is heavy load on the SmartThings server, or some other extreme circumstance), it's possible that a scheduled job could be missed. For example, say you have set up a job to execute every minute, and for some reason, it doesn't execute for three minutes.

When the job does execute again, it will resume its schedule (once every minute) - your handler won't suddenly be called three times, for example.

unschedule can take several seconds to execute

As discussed above, `unschedule` is currently a potentially expensive operation.

We plan to address this in the near future. Until we do, be aware of the potential performance impacts of calling `unschedule`.

5.8.6 Examples

These SmartApps can be viewed in the IDE using the “Browse Templates” button:

- “Once a Day” uses `schedule` to turn switches on and off every day at a specified time.
- “Turn It On For 5 Minutes” uses `runIn` to turn a switch off after five minutes.
- “Left It Open” uses `runIn` to see if a door has been left open for a specified number of minutes.
- “Medicine Reminder” uses `schedule` to check if a medicine door has been opened at a certain time.

5.9 Sunset and Sunrise

SmartApps often need to take some action at or around the local sunrise or sunset time. The SmartThings cloud provides access to this type of rich data, and even generates events for the location (if the geofence is set). We can also get access to sunrise and sunset times using a ZIP code.

5.9.1 Sunrise and Sunset Events

Using the sunrise and sunset events is the preferred (and simpler) way to take some action at (or around) sunrise or sunset. It is required that the location has set up a geofence.

Taking action at sunrise or sunset

If you wish to have certain actions take place at sunrise or sunset, you can use the `sunrise` and `sunset` events. These events will be fired at (gasp!) sunrise and sunset times for the user's location.

You can subscribe to the events by passing in the location (automatically injected into every SmartApp), the event (“sunrise” or “sunset”), and your handler method:

```
def installed() {
    subscribe(location, "sunset", sunsetHandler)
    subscribe(location, "sunrise", sunriseHandler)
}

def sunsetHandler(evt) {
    log.debug "Sun has set!"
    ...
}

def sunriseHandler(evt) {
    log.debug "Sun has risen!"
    ...
}
```

Taking action before or after sunrise/sunset times

If you want to take some action a certain amount of time before or after sunset or sunrise, you can use the “sunriseTime” and “sunsetTime” events. These events are fired every day around the time of sunset or sunrise, and their value is the *next* sunrise or sunset. You can use this information to calculate an offset so that some action happens a certain amount of time before or after sunrise or sunset.

To use, you can subscribe to the events by passing the location, the event (“sunriseTime” or “sunsetTime”), and the handler method.

Consider the following example that turns on lights a specified number of minutes before sunset for the user’s location:

```
preferences {
    section("Lights") {
        input "switches", "capability.switch", title: "Which lights to turn on?"
        input "offset", "number", title: "Turn on this many minutes before sunset"
    }
}

def installed() {
    initialize()
}

def updated() {
    unsubscribe()
    initialize()
}

def initialize() {
    subscribe(location, "sunsetTime", sunsetTimeHandler)

    //schedule it to run today too
    scheduleTurnOn(location.currentValue("sunsetTime"))
}

def sunsetTimeHandler(evt) {
    //when I find out the sunset time, schedule the lights to turn on with an offset
    scheduleTurnOn(evt.value)
}

def scheduleTurnOn(sunsetString) {
    //get the Date value for the string
```

```
def sunsetTime = Date.parse("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'", sunsetString)

//calculate the offset
def timeBeforeSunset = new Date(sunsetTime.time - (offset * 60 * 1000))

log.debug "Scheduling for: $timeBeforeSunset (sunset is $sunsetTime)"

//schedule this to run one time
runOnce(timeBeforeSunset, turnOn)
}

def turnOn() {
    log.debug "turning on lights"
    switches.on()
}
```

Because the `sunriseTime` and `sunsetTime` events are fired every day for the *next* sunrise/sunset event, we use `runOnce` to schedule one execution. Sunrise and sunset times change, so the next time the events are fired, we will create another scheduled execution using the `runOnce` method for that time.

We want it to run today too, so we use the `sunsetTime` value of the user's location to schedule the lights to turn on today.

Note: If a user changes their location's geofence, it could change the sunrise and sunset times. You can listen for position change events and reschedule accordingly: `subscribe(location, "position", locationPositionChangeHandler)`

5.9.2 Looking up Sunrise or Sunset Directly

SmartApps can use the provided `getSunriseAndSunset` methods to get the sunrise and sunset time. You can pass in a ZIP code, which can be useful if the user has not set a geofence for their location.

`getSunriseAndSunset(Map options)`

The supported options are:

zipCode Optional. The ZIP code to get the sunrise and sunset data for. Defaults to the user's location if not provided.

sunsetOffset Optional. A string in the format of "HH:MM" to specify how long or after sunset to return. Use the "-" symbol to indicate time should be before sunset ("-00:30" for 30 minutes prior to sunset)

sunriseOffset Optional. A string in the format of "HH:MM" to specify how long or after sunrise to return. Use the "-" symbol to indicate time should be before sunrise ("-00:30" for 30 minutes prior to sunrise)

date Optional. If you want to find the sunrise or sunset time for a date other than today, you can specify a `Date` object.

The return value is a map in the following form:

```
[sunrise: Date, sunset: Date]
```

```
def initialize() {
    def noParams = getSunriseAndSunset()
    def beverlyHills = getSunriseAndSunset(zipCode: "90210")
    def thirtyMinsBeforeSunset = getSunriseAndSunset(sunsetOffset: "-00:30")

    log.debug "sunrise with no parameters: ${noParams.sunrise}"
    log.debug "sunset with no parameters: ${noParams.sunset}"
    log.debug "sunrise and sunset in 90210: $beverlyHills"
    log.debug "thirty minutes before sunset at current location: ${thirtyMinsBeforeSunset.sunset}"
}
```



```
}

```

5.9.3 Polling for Sunrise/Sunset

You may have seen some SmartApp code that runs a task sometime after midnight (usually in a method called “astroCheck”) and calls a third party weather API to get the sunrise/sunset times. This is strongly discouraged now; it is much more efficient to use location events as they do not rely on third party services.

5.9.4 Examples

You can refer to these example SmartApps in the IDE to see how sunrise and sunset can be used:

- Smart Nightlight
- Sunrise/Sunset

You can also refer to the following examples in Github:

- [Sunset Event Example](#)
- [Sunset Offset Example](#)
- [Sunset by ZIP Code Example](#)

5.10 Calling Web Services

SmartApps or device handlers may need to make calls to external web services. There are several APIs available to you to handle making these requests.

The various APIs are named for the underlying HTTP method they will use. `httpGet()` makes an HTTP GET request, for example.

Important: Requests are limited to ten seconds. If the request takes longer than that, it will timeout.

The following methods are available for making HTTP requests. You can read more about each of them in the [SmartApp](#) (page 248) API documentation.

Method	Description
<code>httpDelete()</code>	Executes an HTTP DELETE request
<code>httpGet()</code>	Executes an HTTP GET request
<code>httpHead()</code>	Executes an HTTP HEAD request
<code>httpPost()</code>	Executes an HTTP POST request
<code>httpPostJson()</code>	Executes an HTTP POST request with JSON Content-Type
<code>httpPutJson()</code>	Executes an HTTP PUT request with JSON Content-Type

Here’s a simple example of making an HTTP GET request:

```
def params = [
    uri: "http://httpbin.org",
    path: "/get"
]

try {
    httpGet(params) { resp ->
```

```
resp.headers.each {
  log.debug "${it.name} : ${it.value}"
}
log.debug "response contentType: ${resp.contentType}"
log.debug "response data: ${resp.data}"
}
} catch (e) {
  log.error "something went wrong: $e"
}
```

5.10.1 Configuring The Request

The various APIs for making HTTP requests all accept a map of parameters that define various information about the request:

Parameter	Description
uri	Either a URI or URL of of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
contentType	Request content type and Accept header.
requestContentType	Content type for the request, if it is different from the expected response content-type.
body	Request body that will be encoded based on the given contentType.

Note: Specifying a `requestContentType` may override the default behavior of the various http API you are calling. For example, `httpPostJson()` sets the `requestContentType` to "application/json" by default.

5.10.2 Handling The Response

The HTTP APIs accept a closure that will be called with the response information from the request.

The closure is passed an instance of a [HttpResponseDecorator](#). You can inspect this object to get information about the response.

Here's an example of getting various response information:

```
def params = [
  uri: "http://httpbin.org",
  path: "/get"
]

try {
  httpGet(params) { resp ->
    // iterate all the headers
    // each header has a name and a value
    resp.headers.each {
      log.debug "${it.name} : ${it.value}"
    }

    // get an array of all headers with the specified key
    def theHeaders = resp.getHeaders("Content-Length")

    // get the contentType of the response
    log.debug "response contentType: ${resp.contentType}"
  }
}
```

```
// get the status code of the response
log.debug "response status code: ${resp.status}"

// get the data from the response body
log.debug "response data: ${resp.data}"
}
} catch (e) {
    log.error "something went wrong: $e"
}
```

Tip: Any ‘failed’ response will generate an exception, so you should wrap your calls in a try/catch block.

If the response returns JSON, data will be in a map-like structure that allows you to easily access the response data:

```
def makeJSONWeatherRequest() {
    def params = [
        uri: 'http://api.openweathermap.org/data/2.5/',
        path: 'weather',
        contentType: 'application/json',
        query: [q: 'Minneapolis', mode: 'json']
    ]
    try {
        httpGet(params) {resp ->
            log.debug "resp data: ${resp.data}"
            log.debug "humidity: ${resp.data.main.humidity}"
        }
    } catch (e) {
        log.error "error: $e"
    }
}
```

The `resp.data` from the request above would look like this (indented for readability):

```
resp data: [id:5037649, dt:1432752405, clouds:[all:0],
  coord:[lon:-93.26, lat:44.98], wind:[speed:4.26, deg:233.507],
  cod:200, sys:[message:0.012, sunset:1432777690, sunrise:1432722741,
    country:US],
  name:Minneapolis, base:stations,
  weather:[id:800, icon:01d, description:Sky is Clear, main:Clear]],
  main:[humidity:73, pressure:993.79, temp_max:298.696, sea_level:1026.82,
    temp_min:298.696, temp:298.696, grnd_level:993.79]]
```

We can easily get the humidity from this data structure as shown above:

```
resp.data.main.humidity
```

5.10.3 Try It Out

If you’re interested in experimenting with the various HTTP APIs, there are a few tools you can use to try out the APIs without signing up for any API keys.

You can use htpbin.org to test making simple requests. The `httpGet()` example above uses it.

For testing POST requests, you can use [PostCatcher](#). You can generate a target URL and then inspect the contents of the request. Here’s an example using `httpPostJson()`:

```
def params = [
  uri: "http://postcatcher.in/catchers/<yourUniquePath>",
  body: [
    param1: [subparam1: "subparam 1 value",
              subparam2: "subparam2 value"],
    param2: "param2 value"
  ]
]

try {
  httpPostJson(params) { resp ->
    resp.headers.each {
      log.debug "${it.name} : ${it.value}"
    }
    log.debug "response contentType: ${resp.contentType}"
  }
} catch (e) {
  log.debug "something went wrong: $e"
}
```

5.10.4 See Also

A simple example using `httpGet` that connects a SmartSense Temp/Humidity to your Weather Underground personal weather station can be found [here](#).

You can browse some templates in the IDE that use the various HTTP APIs. The Ecobee Service Manager is an example that uses both `httpGet()` and `httpPost()`.

5.11 Sending Notifications

SmartApps can send notifications, either as a push notification in the mobile app, or as SMS messages to designated recipients. This allows SmartApps to notify people when important events happen in their home.

In this guide, you will learn:

- How to send notifications to contacts in a user's Contact Book
- How to send push notifications to the mobile app
- How to send SMS notifications
- How to display messages in the Notifications feed of the mobile app

5.11.1 Send Notifications with Contact Book

If a user has added contacts to their Contact Book, SmartApps can prompt a user to select contacts to send notifications to. This allows a user's contacts to be managed independently through the Contact Book, and SmartApps can tap into that feature. This has the advantage that a user does not have to enter in phone numbers for every SmartApp.

Sending notifications by using the Contact Book feature is the preferred way for sending notifications in a SmartApp.

Selecting Contacts to Notify

To allow a user to select from a list of their contacts, use the `"contact"` input type:

```


preferences {
    section("Send Notifications?") {
        input("recipients", "contact", title: "Send notifications to")
    }
}

```

When the user configures this SmartApp, they can then select which contacts they want to notify, and how they should be notified (SMS or push):


Send notifications to Done

+ Create New Contact

 Richard Hendricks

iPhone ☐

Push ☐

 The One Who Knocks

Work ☐

In the example above, the users selected will be stored in a variable named `recipients`. This is just a simple list that we can pass into the `sendNotificationToContacts()` method.

Note: When creating contacts, the user can enter an email address. Emails are *not* currently sent by SmartThings, though they are used to identify SmartThings users, and enable them to receive push notifications.

Send Notifications to Contacts

Use the `sendNotificationToContacts()` method to send a notification to the users (and the specified mode of contact) selected.

`sendNotificationToContacts()` accepts three parameters - the message to send, the contacts selected, and an optional map of additional parameters. The valid option for the additional parameters is `[event: false]`, which will suppress the message from appearing in the Notifications feed.

Assuming the "contact" input named "recipients" above, you would use:

```
sendNotificationToContacts("something you care about!", recipients)
```

If you don't want the message to appear in the Notifications feed, specify `event: false`:

```
sendNotificationToContacts("something you care about!", recipients, [event: false])
```

Handling Disabled Contact Book

A user may not have created any contacts, and SmartApps should be written to handle this.

The "contact" input element takes an optional closure, where you can define additional input elements that will be displayed if the user has no contacts. If the user has contacts, these input elements won't be seen when installing or configuring the SmartApp.

Modifying our preferences definition from above, to handle the case of a user having no contacts, would look like:

```
preferences {
    section("Send Notifications?") {
        input("recipients", "contact", title: "Send notifications to") {
            input "phone", "phone", title: "Warn with text message (optional)",
                description: "Phone Number", required: false
        }
    }
}
```

If the user configuring this SmartApp does have contacts defined, they will only see the input to select from those contacts. If they don't have any contacts defined, they will see the input to enter a phone number.

When attempting to send notifications, we should also check to see if the user has enabled the Contact Book and selected contacts. You can check the `contactBookEnabled` property on `location` to find out if Contact Book has been enabled. It's a good idea to also check if any contacts have been selected.

```
// check that contact book is enabled and recipients selected
if (location.contactBookEnabled && recipients) {
    sendNotificationToContacts("your message here", recipients)
} else if (phone) { // check that the user did select a phone number
    sendSms(phone, "your message here")
}
```

Complete Example

The example SmartApp below sends a notification to selected contacts when a door opens. If the user has no contacts, they can enter in a number to receive an SMS notification.

```
definition(
    name: "Contact Book Example",
    namespace: "smarththings",
```

```

author: "SmartThings",
description: "Example using Contact Book",
category: "My Apps",
iconUrl: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience.png",
iconX2Url: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience@2x.png",
iconX3Url: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience@2x.png")

preferences {
    section("Which Door?") {
        input "door", "capability.contactSensor", required: true,
            title: "Which Door?"
    }

    section("Send Notifications?") {
        input("recipients", "contact", title: "Send notifications to") {
            input "phone", "phone", title: "Warn with text message (optional)",
                description: "Phone Number", required: false
        }
    }
}

def installed() {
    initialize()
}

def updated() {
    initialize()
}

def initialize() {
    subscribe(door, "contact.open", doorOpenHandler)
}

def doorOpenHandler(evt) {
    log.debug "recipients configured: $recipients"

    def message = "The ${door.displayName} is open!"
    if (location.contactBookEnabled && recipients) {
        log.debug "contact book enabled!"
        sendNotificationToContacts(message, recipients)
    } else {
        log.debug "contact book not enabled"
        if (phone) {
            sendSms(phone, message)
        }
    }
}

```

Note: The rest of this guide discusses alternative ways to send notifications (push, SMS, Notifications Feed). SmartApps should use Contact Book, and use the methods described below as a precaution in case the user does not have Contact Book enabled.

5.11.2 Send Push Notifications

To send a push notification through the SmartThings mobile app, you can use the `sendPush()` or `sendPushMessage()` methods. Both methods simply take the message to display. `sendPush()` will display the message in the Notifications feed; `sendPushMessage()` will not.

A simple example below shows (optionally) sending a push message when a door opens:

```
preferences {
    section("Which door?") {
        input "door", "capability.contactSensor", required: true,
            title: "Which door?"
    }
    section("Send Push Notification?") {
        input "sendPush", "bool", required: false,
            title: "Send Push Notification when Opened?"
    }
}

def installed() {
    initialize()
}

def updated() {
    initialize()
}

def initialize() {
    subscribe(door, "contact.open", doorOpenHandler)
}

def doorOpenHandler(evt) {
    if (sendPush) {
        sendPush("The ${door.displayName} is open!")
    }
}
```

Push notifications will be sent to all users with the SmartThings mobile app installed, for the account the SmartApp is installed into.

5.11.3 Send SMS Notifications

In addition to sending push notifications through the SmartThings mobile app, you can also send SMS messages to specified numbers using the `sendSms()` and `sendSmsMessage()` methods.

Both methods take a phone number (as a string) and a message to send. The message can be no longer than 140 characters. `sendSms()` will display the message in the Notifications feed; `sendSmsMessage()` will not.

Extending the example above, let's add the ability for a user to (optionally) send an SMS message to a specified number:

```
preferences {
    section("Which door?") {
        input "door", "capability.contactSensor", required: true,
            title: "Which door?"
    }
}
```



```

    section("Send Push Notification?") {
        input "sendPush", "bool", required: false,
            title: "Send Push Notification when Opened?"
    }
    section("Send a text message to this number (optional)") {
        input "phone", "phone", required: false
    }
}

def installed() {
    initialize()
}

def updated() {
    initialize()
}

def initialize() {
    subscribe(door, "contact.open", doorOpenHandler)
}

def doorOpenHandler(evt) {
    def message = "The ${door.displayName} is open!"
    if (sendPush) {
        sendPush(message)
    }
    if (phone) {
        sendSms(phone, message)
    }
}

```

SMS notifications will be sent from the number 844647 (“THINGS”).

5.11.4 Send Both Push and SMS Notifications

The `sendNotification()` method allows you to send both push and/or SMS messages, in one convenient method call. It can also optionally display the message in the Notifications feed.

`sendNotification()` takes a message parameter, and a map of options that control how the message should be sent, if the message should be displayed in the Notifications feed, and a phone number to send an SMS to (if specified):

```

// sends a push notification, and displays it in the Notifications feed
sendNotification("test notification - no params")

// same as above, but explicitly specifies the push method (default is push)
sendNotification("test notification - push", [method: "push"])

// sends an SMS notification, and displays it in the Notifications feed
sendNotification("test notification - sms", [method: "phone", phone: "1234567890"])

// Sends a push and SMS message, and displays it in the Notifications feed
sendNotification("test notification - both", [method: "both", phone: "1234567890"])

// Sends a push message, and does not display it in the Notifications feed
sendNotification("test notification - no event", [event: false])

```

5.11.5 Only Display Message in the Notifications Feed

Use the `sendNotificationEvent()` method to display a message in the Notifications feed, without sending a push notification or SMS message:

```
sendNotificationEvent("Your home talks!")
```

5.11.6 Examples

Several examples exist in the SmartApp templates that send notifications. Here are a few you can look at to learn more:

- “Notify Me When” sends push or text messages in response to a variety of events.
 - “Presence Change Push” and “Presence Change Text” send notifications when people arrive or depart.
-

5.11.7 Related API Documentation

- [*sendNotificationToContacts\(\)*](#) (page 269)
- [*contactBookEnabled*](#) (page 326)
- [*sendPush\(\)*](#) (page 269)
- [*sendPushMessage\(\)*](#) (page 270)
- [*sendSms\(\)*](#) (page 270)
- [*sendSmsMessage\(\)*](#) (page 270)
- [*sendNotification\(\)*](#) (page 268)
- [*sendNotificationEvent\(\)*](#) (page 268)

5.12 Example: Bon Voyage

To help illustrate some of the important concepts in writing a SmartApp, let’s walk through an example.

Bon Voyage

Our example SmartApp is fairly simple - it will monitor a set of presence detectors, and trigger a mode change when everyone has left.

To accomplish this, our app will need to do the following:

- Gather the necessary input from the user, including which sensors to monitor, what mode to trigger, and other app preferences.
- Subscribe to the appropriate events, and take action when they are triggered.

Let’s begin with configuring the preferences.

Bon Voyage Configurations

To configure the Bon Voyage app, we will want to gather the following information from the user:

- Which sensors to monitor
- The mode to trigger when everyone is away
- A false alarm threshold
- Who should be notified, and how

Each of these inputs corresponds into a preferences section:

```
preferences {
  section("When all of these people leave home") {
    input "people", "capability.presenceSensor", multiple: true
  }
  section("Change to this mode") {
    input "newMode", "mode", title: "Mode?"
  }
  section("False alarm threshold (defaults to 10 min)") {
    input "falseAlarmThreshold", "decimal", title: "Number of minutes", required: false
  }
  section("Notifications") {
    input "sendPushMessage", "enum", title: "Send a push notification?",
      options: ["Yes", "No"], required: false
    input "phone", "phone", title: "Send a Text Message?", required: false
  }
}
```

Let's look at each section in a bit more detail.

The *When all of these people leave home* section allows the user to configure what sensors to use for this app. The user will see a section with the main title “When all of these people leave home.” A dropdown will be populated with all the devices that have the `presenceSensor` capability (*capability.presenceSensor*) for them to select the sensor(s) they'd like to use. *Multiple: true* allows them to add as many sensors as they'd like. Their choice(s) are then stored in a variable named *people*.

The *Change to this mode* section allows the user to specify what mode should be triggered when everyone is away. The input type of *mode* is used, so a dropdown will be populated with all the modes the user has set up. The title property is used to show the title “Mode?” above the field. The selection is stored in the variable named *newMode*.

The section *False alarm threshold (defaults to 10 min)* allows the user to specify a false alarm threshold. These types of thresholds are common in our SmartApps. A section is shown titled “False alarm threshold (defaults to 10 min)”. The input fieldtype of *decimal* is used, to allow the user to input a numeric value that represents minutes. The title “Number of minutes” is specified, and we set the *required* property to *false*. By default, all fields are required, so you must explicitly state if it is not required. We store the user's input in the variable named *falseAlarmThreshold* for later use.

Finally, a section is shown labeled as “Notifications”. This is where the user can configure how they want to be notified when everyone is away. An input with the field type of *enum* is created. With *enum* you must define values for it, so they are defined via *options:["Yes","No"]*. This field is not required as specified by *required:false*, and what the user selects will be stored in *sendPushMessage*. There is also an optional field called “Send a Text Message?”. It uses the field type of *phone* to provide a formatted input for phone numbers. The values input by the user are stored in the *phone* variable.

Monitor and React

Now that we have gathered the input we need from the user, we need to listen to the appropriate events, and take action when they are triggered.

We do this through the required *installed* method:

```
def installed() {
    log.debug "Installed with settings: ${settings}"
    log.debug "Current mode = ${location.mode}, people = ${people.collect{it.label + ': ' + it.currentPresence}}
    subscribe(people, "presence", presence)
}
```

Upon installation, we want to keep track of the status of our people. We use the *subscribe* method to “listen” to the *presence* attribute of the predefined group of presence sensors, *people*. When the presence status changes of any of our people, the method *presence* (the last parameter above) will be called.

(Also note the log statements. We won’t discuss log statements in detail, but providing good logging is a habit you will want to get into as a SmartApps developer. Good logging is invaluable when trying to debug/troubleshoot your app!)

Let’s define our presence method.

```
def presence(evt) {
    log.debug "evt.name: $evt.value"
    if (evt.value == "not present") {
        if (location.mode != newMode) {
            log.debug "checking if everyone is away"
            if (everyoneIsAway()) {
                log.debug "starting sequence"
                runIn(findFalseAlarmThreshold() * 60, "takeAction", [overwrite: false])
            }
        }
        else {
            log.debug "mode is the same, not evaluating"
        }
    }
    else {
        log.debug "present; doing nothing"
    }
}

// returns true if all configured sensors are not present,
// false otherwise.
private everyoneIsAway() {
    def result = true
    // iterate over our people variable that we defined
    // in the preferences method
    for (person in people) {
        if (person.currentPresence == "present") {
            // someone is present, so set our our result
            // variable to false and terminate the loop.
            result = false
            break
        }
    }
    log.debug "everyoneIsAway: $result"
    return result
}

// gets the false alarm threshold, in minutes. Defaults to
// 10 minutes if the preference is not defined.
private findFalseAlarmThreshold() {
    // In Groovy, the return statement is implied, and not required.
    // We check to see if the variable we set in the preferences
    // is defined and non-empty, and if it is, return it. Otherwise,
```

```
// return our default value of 10
(falseAlarmThreshold != null && falseAlarmThreshold != "") ? falseAlarmThreshold : 10
}
```

Let's break that down a bit.

The first thing we need to do is see what event was triggered. We do this by inspecting the *evt* variable that is passed to our event handler. The presence capability can be either “present” or “not present”.

Next, we check that the current mode isn't already set to the mode we want to trigger. If we're already in our desired mode, there's nothing else for us to do!

Now it starts to get fun! If everyone is away, we call the built-in *runIn* method, which runs the method *takeAction* in a specified amount of time (we'll define that method shortly). We use a helper method *findFalseAlarmThreshold()* multiplied by 60 to convert minutes to seconds, which is what the *runIn* method requires. We specify *overwrite: false* so that it won't overwrite previously scheduled *takeAction* calls. In the context of this SmartApp, it means that if one user leaves, and then another user leaves within the *falseAlarmThreshold* time, *takeAction* will still be called twice. By default, *overwrite* is true, meaning that if you scheduled *takeAction* to run previously, it would be cancelled and replaced by your current call.

We also have defined two helper methods above, *everyoneIsAway*, and *findFalseAlarmThreshold*.

everyoneIsAway returns true if all configured sensors are not present, and false otherwise. It iterates over all the sensors configured and stored in the *people* variable, and inspects the *currentPresence* property. If the *currentPresence* is “present”, we set the result to false, and terminate the loop. We then return the value of the result variable.

findFalseAlarmThreshold gets the false alarm threshold, in minutes, as configured by the user. If the threshold preference has not been set, it returns ten minutes as the default.

Now we need to define our *takeAction* method:

```
def takeAction() {
    if (everyoneIsAway()) {
        def threshold = 1000 * 60 * findFalseAlarmThreshold() - 1000
        def awayLongEnough = people.findAll { person ->
            def presenceState = person.currentState("presence")
            def elapsed = now() - presenceState.rawDateCreated.time
            elapsed >= threshold
        }
        log.debug "Found ${awayLongEnough.size()} out of ${people.size()} person(s) who were away long enough"
        if (awayLongEnough.size() == people.size()) {
            //def message = "${app.label} changed your mode to '${newMode}' because everyone left home"
            def message = "SmartThings changed your mode to '${newMode}' because everyone left home"
            log.info message
            send(message)
            setLocationMode(newMode)
        } else {
            log.debug "not everyone has been away long enough; doing nothing"
        }
    } else {
        log.debug "not everyone is away; doing nothing"
    }
}

private send(msg) {
    if ( sendPushMessage != "No" ) {
        log.debug( "sending push message" )
        sendPush( msg )
    }
}
```

```
    if ( phone ) {
        log.debug( "sending text message" )
        sendSms( phone, msg )
    }

    log.debug msg
}
```

There's a lot going on here, so we'll look at some of the more interesting parts.

The first thing we do is check again if everyone is away. This is necessary since something may have changed since it was already called, because of the *falseAlarmThreshold*.

If everyone is away, we need to find out how many people have been away for long enough, using our false alarm threshold. We create a variable, *awayLongEnough* and set it through the Groovy *findAll* method. The *findAll* method returns a subset of the collection based on the logic of the passed-in closure. For each person, we use the *currentState* method available to us, and use that to get the time elapsed since the event was triggered. If the time elapsed since this event exceeds our threshold, we add it to the *awayLongEnough* collection by returning true in our closure (note that we could omit the "return" keyword, as it is implied in Groovy). For more information about the *findAll* method, or how Groovy utilizes closures, consult the Groovy documentation at <http://www.groovy-lang.org/documentation.html>

If the number of people away long enough equals the total number of people configured for this app, we send a message (we'll look at that method next), and then call the *setLocationMode* method with the desired mode. This is what will cause a mode change.

The *send* method takes a String parameter, *msg*, and if the user has configured the app to send a push notification, calls the *sendPush* method. It then checks to see if the user has chosen to send a text message, by checking if the *phone* variable has been set. If it has, it calls the *sendSms(phone, msg)* method.

Finally, we need to write our *updated* method, which is called whenever the user changes any of their configurations. When this method is called, we need to call the *unsubscribe* method, and then *subscribe*, to effectively reset our app.

```
def updated() {
    log.debug "Updated with settings: ${settings}"
    log.debug "Current mode = ${location.mode}, people = ${people.collect{it.label + ': ' + it.currentState}}
    unsubscribe()
    subscribe(people, "presence", presence)
}
```

Our SmartApp is now complete! Putting it all together, here's our final Bon Voyage app:

Complete Code Listing

```
/**
 * Bon Voyage
 *
 * Author: SmartThings
 * Date: 2013-03-07
 *
 * Monitors a set of presence detectors and triggers a mode change when everyone has left.
 */

preferences {
    section("When all of these people leave home") {
        input "people", "capability.presenceSensor", multiple: true
    }
    section("Change to this mode") {
        input "newMode", "mode", title: "Mode?"
    }
    section("False alarm threshold (defaults to 10 min)") {

```

```

        input "falseAlarmThreshold", "decimal", title: "Number of minutes", required: false
    }
    section( "Notifications" ) {
        input "sendPushMessage", "enum", title: "Send a push notification?",
            options: ["Yes", "No"], required: false
        input "phone", "phone", title: "Send a Text Message?", required: false
    }
}

def installed() {
    log.debug "Installed with settings: ${settings}"
    log.debug "Current mode = ${location.mode}, people = ${people.collect{it.label + ': ' + it.currentPresence}}"
    subscribe(people, "presence", presence)
}

def updated() {
    log.debug "Updated with settings: ${settings}"
    log.debug "Current mode = ${location.mode}, people = ${people.collect{it.label + ': ' + it.currentPresence}}"
    unsubscribe()
    subscribe(people, "presence", presence)
}

def presence(evt) {
    log.debug "evt.name: $evt.value"

    // The presence capability can either be "present" or "not present".
    // If the user is not present, we want to check if everyone is away
    if (evt.value == "not present") {
        // Check that the desire mode isn't already the same as the current mode.
        if (location.mode != newMode) {
            log.debug "checking if everyone is away"
            // If everyone is away, start the sequence
            if (everyoneIsAway()) {
                log.debug "starting sequence"
                runIn(findFalseAlarmThreshold() * 60, "takeAction", [overwrite: false])
            }
        }
        else {
            log.debug "mode is the same, not evaluating"
        }
    }
    else {
        log.debug "present; doing nothing"
    }
}

// returns true if all configured sensors are not present,
// false otherwise.
private everyoneIsAway() {
    def result = true
    // iterate over our people variable that we defined
    // in the preferences method
    for (person in people) {
        if (person.currentPresence == "present") {
            // someone is present, so set our result
            // variable to false and terminate the loop.
            result = false
            break
        }
    }
}

```

```
    }
}
log.debug "everyoneIsAway: $result"
return result
}

// gets the false alarm threshold, in minutes. Defaults to
// 10 minutes if the preference is not defined.
private findFalseAlarmThreshold() {
    // In Groovy, the return statement is implied, and not required.
    // We check to see if the variable we set in the preferences
    // is defined and non-empty, and if it is, return it. Otherwise,
    // return our default value of 10
    (falseAlarmThreshold != null && falseAlarmThreshold != "") ? falseAlarmThreshold : 10
}

def takeAction() {
    if (everyoneIsAway()) {
        def threshold = 1000 * 60 * findFalseAlarmThreshold() - 1000
        def awayLongEnough = people.findAll { person ->
            def presenceState = person.currentState("presence")
            def elapsed = now() - presenceState.rawDateCreated.time
            elapsed >= threshold
        }
        log.debug "Found ${awayLongEnough.size()} out of ${people.size()} person(s) who were away long enough"
        if (awayLongEnough.size() == people.size()) {
            //def message = "${app.label} changed your mode to '${newMode}' because everyone left home"
            def message = "SmartThings changed your mode to '${newMode}' because everyone left home"
            log.info message
            send(message)
            setLocationMode(newMode)
        } else {
            log.debug "not everyone has been away long enough; doing nothing"
        }
    } else {
        log.debug "not everyone is away; doing nothing"
    }
}

private send(msg) {
    if ( sendPushMessage != "No" ) {
        log.debug( "sending push message" )
        sendPush( msg )
    }

    if ( phone ) {
        log.debug( "sending text message" )
        sendSms( phone, msg )
    }

    log.debug msg
}
```


5.13 Submitting SmartApps for Publication

To submit your SmartApps for consideration for publication to the SmartThings Platform, you can create a Publication Request by clicking on the [My Publication Requests](#) tab in the [SmartThings IDE](#), then clicking on the *New Request* button in the upper-right-hand corner of the screen.

5.13.1 Review Process

Once submitted, your SmartApp will undergo a review. Here is a detailed outline of the approval process:

Functional Review

- Does this SmartApp duplicate an existing SmartApp? If so, does it improve the current SmartApp?
- Does it have a good title, description, and configuration preferences? Will the user understand how it works?
- Does the SmartApp work as expected?

Code Review

- In the `preferences` section, are all inputs used?
- Naming conventions: do variable and function names appropriately describe their use?
- All `subscribe()` calls have corresponding `unsubscribe()` method.
- All `schedule()` calls have corresponding `unschedule()` as needed.
- Scheduling: check for any `runIn()`, `schedule()`, `runDaily()`, etc and make sure they are not running too often.
- When communicating with web services, follow best practices below.
- Avoid redundant code and infinite loops.
- Comment your code!

Publication

Once your app has been approved, it will be published in our mobile app.

5.13.2 Best Practices

For the greatest likelihood of success, follow these guidelines:

- For greatest communication between yourself and the SmartThings reviewer, add your Github username as the namespace.
- Do not use offensive, profane, or libelous language.
- No advertising or sponsorships.

Web Services

- If your SmartApp sends any data from the SmartThings Platform to an external service, include in the description exactly what data is sent to the remote service, how that data will be used, and include a link to the privacy policy of the remote service
- If your SmartApp exposes any Web Service APIs, describe what the APIs will be used for, what data may be accessed by those APIs, and where possible, include a link to the privacy policies of any remote services that may access those APIs.

Devices and Preferences

- Label your SmartApp preferences to match the devices that you are asking for access to.
- Use the *password* input type whenever you are asking a user for a password.

Scheduling

- Do not aggressively loop or schedule.
- If using sunrise and sunset, make sure you are following the guidelines in the [Sunset and Sunrise](#) (page 102) guide.

Style

- Use meaningful variable names.
- Maintain consistent formatting and indentation. We can't review code that we can't easily read.

Web Services SmartApps

SmartApps may themselves be a web service, exposing a URL and any defined endpoints.

This allows external applications to make web API calls to a SmartApp, and get information about, or control, end devices.

To understand how Web Services SmartApps work, including the security measures in place, read the *Web Services SmartApps Overview* (page 123) guide.

For a step-by-step tutorial creating a Web Services SmartApp, read the SmartApp as Web Service tutorial. It's organized into two parts:

- *Building a Web Services SmartApp - Part 1* (page 128), will walk you through a simple SmartApp that exposes endpoints, and use the simulator and curl to test making API calls to your SmartApp.
- *Building a Web Services SmartApp - Part 2* (page 133) will guide you through the steps an external application would take to integrate with SmartThings.

Contents:

6.1 Web Services SmartApps Overview

Integrating with SmartThings using SmartApps Web Services

In this guide, you will learn:

- The overall design of how WebServices SmartApps work.
- Security measures taken to ensure access is only granted to trusted clients, and specific devices as chosen by the user.
- The end user flow for external applications integrating with Web Services SmartApps.

Contents

- *Web Services SmartApps Overview* (page 123)
 - *Introduction* (page 124)
 - *Concepts* (page 124)
 - *How it Works* (page 124)
 - * *OAuth-Integrated App Installation Flow* (page 125)
 - *The End-User Journey* (page 126)
 - * *Initiate Connection from External System* (page 126)
 - * *Authentication & Authorization* (page 126)
 - * *Application Configuration* (page 127)
 - *Rate Limiting* (page 128)
 - * *Rate Limit Headers* (page 128)
 - * *Rate Limit HTTP Status Code* (page 128)

6.1.1 Introduction

Our goal is to become *the* open platform for the consumer Internet of Things. But, that doesn't mean that we can ignore security, even for a minute.

In designing a way to allow external systems API access, we wanted to give developers the flexibility they need, while ensuring that the customer understands why their account is being accessed through an external API, and has specifically authorized that access.

As such, we've designed an architecture and a user experience around external API access that meets the following goals:

- It uses industry best practices such as OAuth2 to authenticate and authorize basic external API access.
- It requires the end user (customer) to specifically authorize the access to specific devices.
- It delivers a user experience that is easy to understand.
- It delivers a developer experience that is easy to understand and implement.

6.1.2 Concepts

There are a couple of important concepts that need to be understood with respect to how SmartApps APIs work.

- All SmartApps APIs are authenticated using OAuth2.
- When we talk about SmartApps APIs, we are referring to APIs that are exposed by SmartApps themselves.
- SmartApps execute in a special security context, where they only have access to devices specifically authorized by the user at installation time. This is no different for SmartApps APIs.

Important: Right now it is not possible for users outside of the U.S. to install SmartApps using the OAuth install flow. Engineering is actively working to support the OAuth flow outside of the U.S. We will follow up when we have a solution or more information.

6.1.3 How it Works

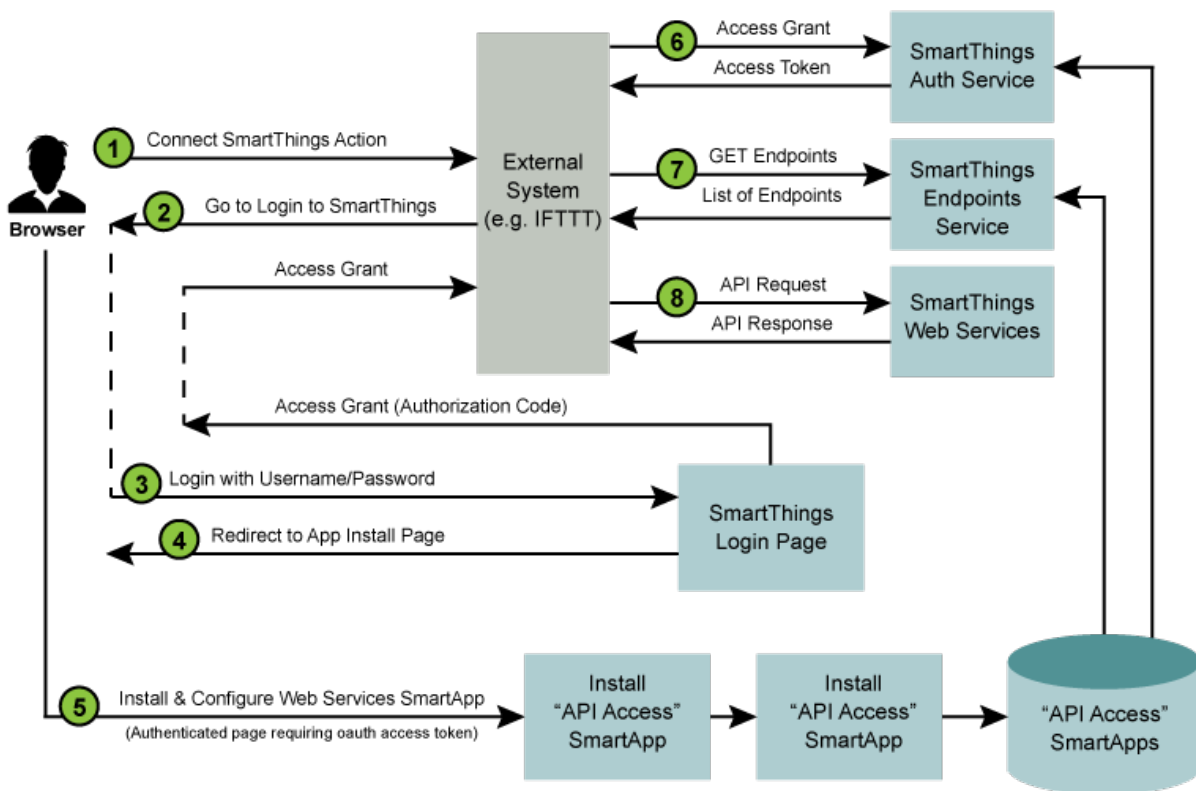
Our overall approach to API access requires the end user to authenticate and authorize the API access in two steps:

1. The installation of a SmartThings Web Services “SmartApp” into the user’s SmartThings Account/Location, along with specific device preferences that specify the devices to which the external system is being granted access.
2. The typical OAuth login flow grants the external system the OAuth access token.

It is important to understand that it is the SmartApp itself that exposes the API endpoints that are then used by the external system to integration with SmartThings.

This approach is designed to ensure that an external system must have explicit access granted to the devices, before it can control those devices, and that the OAuth access token isn’t enough to grant access to the user’s entire smart home.

OAuth-Integrated App Installation Flow



The diagram above outlines the following standard steps in the API Connection and Usage process:

1. A user of the external system takes some action that initiates a “Connect to SmartThings” flow. An example of this is an [IFTT](#) user adding the SmartThings “channel”.
2. The external service will typically redirect to the SmartThings login page. The HTTP request to this page includes the required OAuth client id (more details below), allowing our login page to recognize this as a login request using OAuth.
3. The login page is displayed, and if the login is successful, a subsequent page is displayed that allows the non-authenticated user to install and configure the Web Services SmartApp that is associated with the client id. When this step is complete, an authorization code is returned to the browser.

4. Typically, the authorization code is then given to the external system, and it is used (along with the OAuth client id and client secret), to request an access token. The authorization code takes the place of the user credentials in this case, and is only valid for a single use. Once the external system has the OAuth access token, API requests can be made using this token.
5. The first API call that the external system should make is to the endpoints service. This service exists on a standard URL, and will return the specific URL that the external system should use (for this specific OAuth access token) to make all API requests.
6. Finally, the external system can use the specified endpoint URL and the provided OAuth2 access token to make API calls to the SmartApp providing the web services.

6.1.4 The End-User Journey

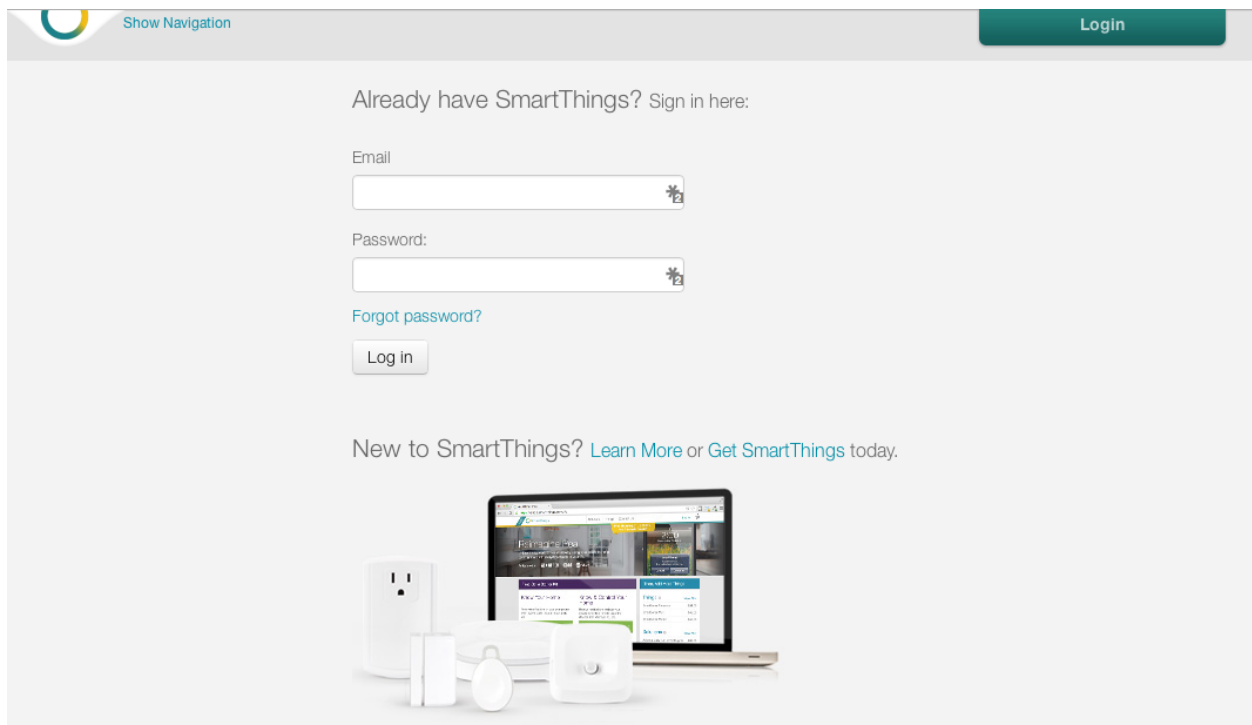
Before discussing the specific steps to building a Web Services SmartApp, you should understand the end user experience.

Initiate Connection from External System

The first step is to initiate the connection with the SmartThings cloud from the external web application. This is different for each web application, but is just a URL.

Authentication & Authorization

The typical OAuth journey is the OAuth2 authorization code flow, initiated from the website of the external system, whereby the user is redirected to the SmartThings website. This is where they enter their SmartThings credentials, as shown below:

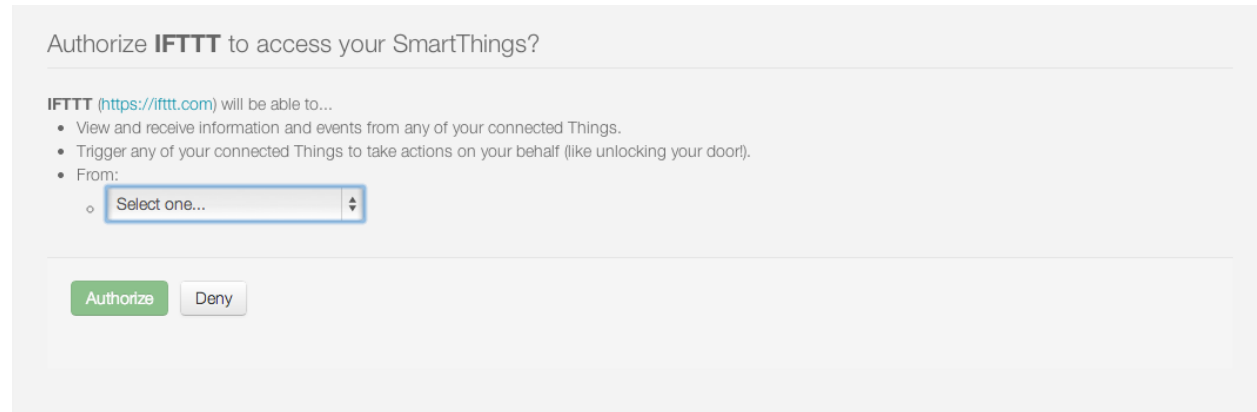


Once authenticated with SmartThings, they will be prompted to specifically authorize access by the application.

Application Configuration

The user is prompted to configure the Web Services SmartApp that will be automatically installed. The user does not have to select the specific SmartApp, because it can be automatically identified by the OAuth client id.

The first step in the application configuration process is to identify the Location in which the SmartApp will be installed.



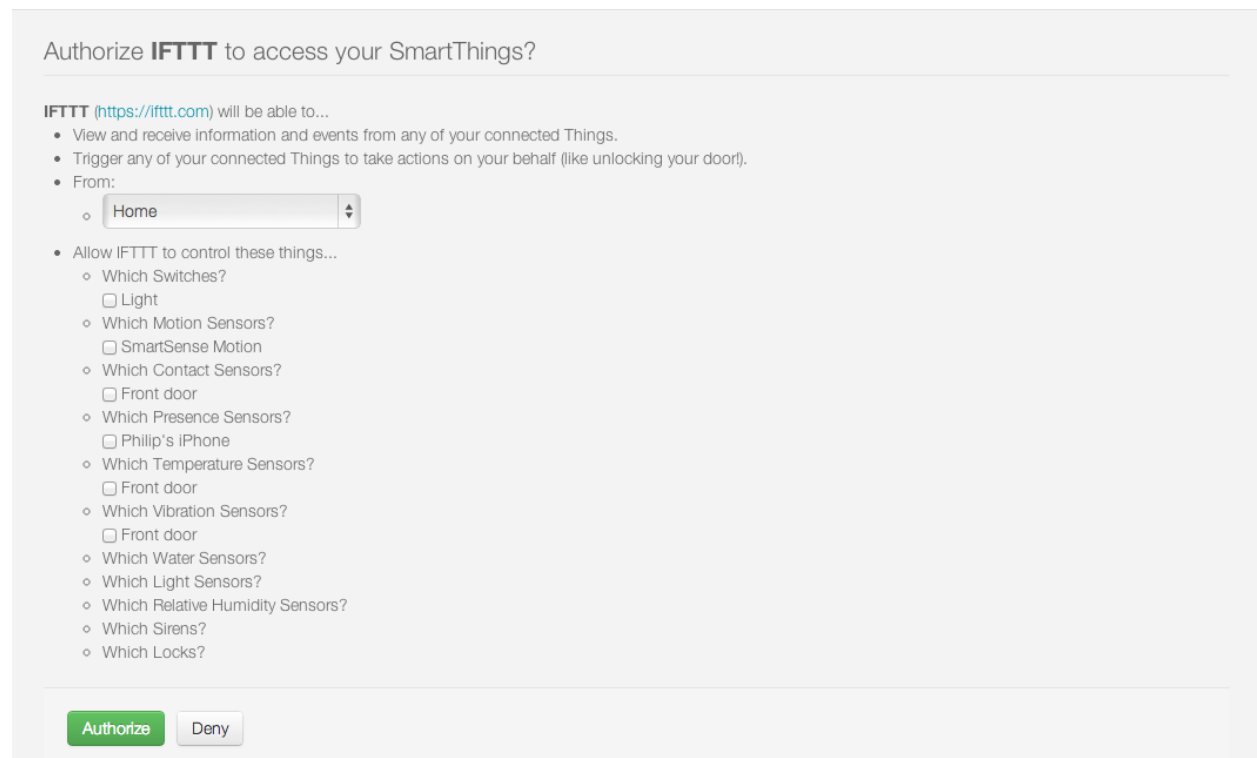
Authorize **IFTTT** to access your SmartThings?

IFTTT (<https://ifttt.com>) will be able to...

- View and receive information and events from any of your connected Things.
- Trigger any of your connected Things to take actions on your behalf (like unlocking your door!).
- From:
 -

The second step is to configure exactly which devices will be accessible through any external web services that are exposed by the SmartApp.

An example of the IFTTT SmartApp device selection options is shown below:



Authorize **IFTTT** to access your SmartThings?

IFTTT (<https://ifttt.com>) will be able to...

- View and receive information and events from any of your connected Things.
- Trigger any of your connected Things to take actions on your behalf (like unlocking your door!).
- From:
 -
- Allow IFTTT to control these things...
 - Which Switches?
 - ☐ Light
 - Which Motion Sensors?
 - ☐ SmartSense Motion
 - Which Contact Sensors?
 - ☐ Front door
 - Which Presence Sensors?
 - ☐ Phillip's iPhone
 - Which Temperature Sensors?
 - ☐ Front door
 - Which Vibration Sensors?
 - ☐ Front door
 - Which Water Sensors?
 - Which Light Sensors?
 - Which Relative Humidity Sensors?
 - Which Sirens?
 - Which Locks?

Finally, the user clicks on “Authorize” to complete both the authorization of the application and the installation of the SmartApp and the connection between the external system and the SmartThings Cloud is now complete.

Once the user authorizes access, the external system is provided with the OAuth authorization code, which is in turn used to request and receive an OAuth access token. Once the external system has the token, it can access the web services provided by the SmartApp.

6.1.5 Rate Limiting

SmartApps or Device Handler's that expose web services are limited in the number of inbound requests they may receive in a time window. This is to ensure that no one SmartApp or Device Handler consumes too many resources in the SmartThings cloud. There are various headers available on every request that provide information about the current rate limit limits for a given installed SmartApp or Device Handler. These are discussed further below.

SmartApps or Device Handlers that expose web APIs are limited to receiving 250 requests in 60 seconds.

Rate Limit Headers

The SmartThings platform will set three HTTP headers on the response for every inbound API call, so that a client may understand the current rate limiting status:

X-RateLimit-Limit: 250 The rate limit - in this example, the limit is 250 requests.

X-RateLimit-Current: 1 The current count of requests for the given time window. In this example, there has been one request within the current rate limit time window.

X-RateLimit-TTL: 58 The time remaining in the current rate limit window. In this example, there is 58 seconds remaining before the current rate limit window resets.

Rate Limit HTTP Status Code

In addition to the three HTTP headers above, when the rate limit has been exceeded, the HTTP status code of 429 will be sent on the response.

6.2 Building a Web Services SmartApp - Part 1

This is the first part of two that will teach you how to build a WebServices SmartApp.

In part 1 of this tutorial, you will learn:

- How to develop a Web Services SmartApp that exposes endpoints.
- How to call the Web Services SmartApp using simple API calls.

Contents

- *Building a Web Services SmartApp - Part 1* (page 128)
 - *Overview* (page 129)
 - *Create a new SmartApp* (page 129)
 - *Define Preferences* (page 129)
 - *Specify Endpoints* (page 130)
 - *GET Switch Information* (page 131)
 - *UPDATE the Switches* (page 131)
 - *Self-publish the SmartApp* (page 132)
 - *Run the SmartApp in the Simulator* (page 132)
 - *Make API Calls to the SmartApp* (page 132)
 - *Uninstall the SmartApp* (page 133)
 - *Summary* (page 133)
 - *Source Code* (page 133)

6.2.1 Overview

Part 1 of this tutorial will build a simple SmartApp that exposes endpoints to get information about and control switches.

6.2.2 Create a new SmartApp

Create a new SmartApp in the IDE. Fill in the required fields, and make sure to click on *Enable OAuth in SmartApp* to receive an auto-generated client ID and secret.

Note the Client ID and secret - they'll be used later (should you forget, you can get them by viewing the "App Settings" in the IDE).

6.2.3 Define Preferences

SmartApps declare preferences metadata that is used at installation and configuration time, to allow the user to control what devices the SmartApp will have access to.

This is a configuration step, but also a security step, whereby the users must explicitly select what devices the SmartApp can control.

Web Services SmartApps are no different, and this is part of the power of this approach. The end user controls exactly what devices the SmartApp will have access to, and therefore what devices the external systems that consume those web services will have access to.

The preferences definition should look like this:

```
preferences {
    section ("Allow external service to control these things...") {
        input "switches", "capability.switch", multiple: true, required: true
    }
}
```

Also ensure that you have an `installed()` and `updated()` method defined (this should be created by default when creating a SmartApp). They can remain empty, since we are not subscribing to any device events in this example.

6.2.4 Specify Endpoints

The `mappings` declaration allows developers to expose HTTP endpoints, and map the various supported HTTP operations to an associated handler.

The handler can process the HTTP request and provide a response, including both the [HTTP status code](#), as well as the response body.

Our SmartApp will expose two endpoints:

- The `/switches` endpoint will support a GET request. A GET request to this endpoint will return state information for the configured switches.
- The `/switches/:command` endpoint will support a PUT request. A PUT request to this endpoint will execute the specified command ("on" or "off") on the configured switches.

Tip: There is no limit to the number of endpoints a SmartApp exposes, but the path level is restricted to four levels deep (i.e., `/level1/level2/level3/level4`).

Here's the code for our mappings definition. This is defined at the top-level in our SmartApp (i.e., not in another method):

```
mappings {
  path("/switches") {
    action: [
      GET: "listSwitches"
    ]
  }
  path("/switches/:command") {
    action: [
      PUT: "updateSwitches"
    ]
  }
}
```

The mappings configuration is made up of one or many `path` definitions. Each `path` defines the endpoint, and also is configured for each HTTP operation using the `action` definition.

`action` is a simple map, where the key is the HTTP operation (e.g., GET, PUT, POST, etc.), and the value is the name of the handler method to be called when this endpoint is called.

Note the use of variable parameters in our PUT endpoint. Use the `:` prefix to specify that the value will be variable. We'll see later how to get this value.

Tip: Endpoints can support multiple REST methods. If we wanted the `/switches` endpoint to also support a PUT request, simply add another entry to the `action` configuration:

```
action: [
  GET: "listSwitches",
  PUT: "putHandlerMethodName"
]
```

Go ahead and add empty methods for the various handlers. We'll fill these in in the next step:

```
def listSwitches() {}

def updateSwitches() {}
```

6.2.5 GET Switch Information

Now that we've defined our endpoints, we need to handle the requests in the handler methods we stubbed in above.

Let's start with the handler for GET requests to the `/switches` endpoint. When a GET request to the `/switches` endpoint is called, we want to return the display name, and the current switch value (e.g., on or off) for the configured switch.

Our handler method returns a list of maps, which is then serialized by the SmartThings platform into JSON:

```
// returns a list like
// [[name: "kitchen lamp", value: "off"], [name: "bathroom", value: "on"]]
def listSwitches() {
    def resp = []
    switches.each {
        resp << [name: it.displayName, value: it.currentValue("switch")]
    }
    return resp
}
```

6.2.6 UPDATE the Switches

We also need to handle a PUT request to the `/switches/:command` endpoint. `/switches/on` will turn the switches on, and `/switches/off` will turn the switches off.

If any of the configured switches does not support the specified command, we'll return a 501 HTTP error.

```
void updateSwitches() {
    // use the built-in request object to get the command parameter
    def command = params.command

    if (command) {

        // check that the switch supports the specified command
        // If not, return an error using httpError, providing a HTTP status code.
        switches.each {
            if (!it.hasCommand(command)) {
                httpError(501, "$command is not a valid command for all switches specified")
            }
        }

        // all switches have the command
        // execute the command on all switches
        // (note we can do this on the array - the command will be invoked on every element
        switches."$command"()
    }
}
```

Tip: Our example uses the endpoint itself to get the command. If you would instead like to pass parameters via the request body, you can retrieve those parameters via the built-in `request` object as well. Assuming the request body looked like `{"command": "on"}`, we can get the specified command parameter like this:

```
// Get the JSON body from the request.
// Safe de-reference using the "?." operator
// to avoid NullPointerException if no JSON is passed.
def command = request.JSON?.command
```

6.2.7 Self-publish the SmartApp

Publish the app for yourself, by clicking on the “Publish” button and selecting “For Me”.

6.2.8 Run the SmartApp in the Simulator

Using the simulator, we can quickly test our Web Services SmartApp.

Click the *Install* button in the simulator, select a Location to install the SmartApp into, and select a switch.

Note that in the lower right of the simulator there is an API token and an API endpoint. We can use these to test making requests to our SmartApp.

6.2.9 Make API Calls to the SmartApp

Using whatever tool you prefer for making web requests (this example will use curl, but [Apigee](#) is a good UI-based tool for making requests), we will call one of our SmartApp endpoints.

From the simulator, grab the API endpoint. It will look something like this:

```
https://graph.api.smarthings.com/api/smartapps/installations/158ef595-3695-49ab-acc1-80e93288c0c8
```

Your installation will have a different, unique URL.

To get information about the switch, we will call the `/switch` endpoint using a GET request. You’ll need to substitute your unique endpoint and API key.

```
curl -H "Authorization: Bearer <api token>" <api endpoint>/switch
```

This should return a JSON response like the following:

```
[{"name": "Kitchen 2", "value": "off"}, {"name": "Living room window", "value": "off"}]
```

To turn the switch on or off, call the `/switch` endpoint using a PUT request, passing the command in the request body. Again, you’ll need to substitute your unique endpoint and API key:

```
curl -H "Authorization: Bearer <api token>" -X PUT <api endpoint>/switch/on
```

Change the command value to `"off"` to turn the switch off. Try turning the switch on and off, and then using `curl` to get the status, to see that it changed.

Tip: You can also pass the API token directly on the URL, via the `access_token` URL parameter, instead of using the Authorization header. This may be useful when you do not have the ability to set request headers.

6.2.10 Uninstall the SmartApp

Finally, uninstall the SmartApp using the *Uninstall* button in the IDE simulator.

6.2.11 Summary

In this tutorial, you learned how to create a SmartApp that exposes endpoints to get information about, and control, a device. You also learned how to install the SmartApp in the simulator, and then make API calls to the endpoint.

In the next part of this tutorial, we'll look at how a external application might interact with SmartThings using the OAuth2 flow (instead of simply using the simulator and its generated access token).

6.2.12 Source Code

The full source code for this tutorial (both parts), can be found [here](#).

6.3 Building a Web Services SmartApp - Part 2

In Part 1 of this tutorial, you learned how to create a simple Web Services SmartApp, and install it in the IDE simulator, and make web requests to it.

In Part 2, we'll build a simple web application that will integrate with SmartThings and the WebServices SmartApp we created in Part 1.

In Part 2 of this tutorial, you will learn:

- How to get the API token.
- How to discover the endpoints of a Web Services SmartApp.
- How to make calls to the Web Services SmartApp.

Contents

- *Building a Web Services SmartApp - Part 2* (page 133)
 - *Overview* (page 134)
 - *Prerequisites* (page 134)
 - *Bootstrap the Sinatra App* (page 134)
 - *Get an Authorization Code* (page 136)
 - *Get an Access Token* (page 137)
 - *Discover the Endpoint* (page 138)
 - *Make API Calls* (page 139)
 - *Summary* (page 140)
 - *Source Code* (page 140)
 - *Appendix - Just the URLs, Please* (page 140)
 - * *Get the OAuth Authorization Code* (page 140)
 - * *Get the API token* (page 140)
 - * *Discover the Endpoint URL* (page 141)
 - * *Make API Calls* (page 141)

6.3.1 Overview

In Part 2 of this tutorial, we will build a simple Sinatra application that will make calls to the Web Services SmartApp we built in Part 1.

If you're not familiar with Sinatra, you are encouraged to try it out. It's not strictly necessary, however, as our application will simply make web requests to get the API token and the endpoint.

If you just want to see the manual steps to make requests to get the access token, discover endpoints, and start making calls, you can see the [Appendix - Just the URLs, Please](#) (page 140).

Note: If Node is more your speed, check out the awesome SmartThings OAuth Node app written by community member John S (@schettj) [here](#). It shows how you can get an access token using the OAuth flow for a WebServices SmartApp using Node.

6.3.2 Prerequisites

Aside from completing Part 1 of this tutorial, you should have Ruby and Sinatra installed.

Visit the [Ruby](#) website to install Ruby, and the [Sinatra Getting Started Page](#) for information about installing Sinatra.

6.3.3 Bootstrap the Sinatra App

Create a new directory for the Sinatra app, and change directories to it:

```
mkdir web-app-tutorial
cd web-app-tutorial
```

In your favorite text editor*, create a new file called `server.rb` and paste the following into it, and save it.

**(If your favorite text editor is vim or emacs, then our hat's off to you. We're impressed - maybe even a bit intimidated. If your favorite editor is notepad, well... we're not as impressed, or intimidated. :@))*

```

require 'bundler/setup'
require 'sinatra'
require 'oauth2'
require 'json'
require "net/http"
require "uri"

# Our client ID and secret, used to get the access token
CLIENT_ID = ENV['ST_CLIENT_ID']
CLIENT_SECRET = ENV['ST_CLIENT_SECRET']

# We'll store the access token in the session
use Rack::Session::Pool, :cookie_only => false

# This is the URI that will be called with our access
# code after we authenticate with our SmartThings account
redirect_uri = 'http://localhost:4567/oauth/callback'

# This is the URI we will use to get the endpoints once we've received our token
endpoints_uri = 'https://graph.api.smarththings.com/api/smartapps/endpoints'

options = {
  site: 'https://graph.api.smarththings.com',
  authorize_url: '/oauth/authorize',
  token_url: '/oauth/token'
}

# use the OAuth2 module to handle OAuth flow
client = OAuth2::Client.new(CLIENT_ID, CLIENT_SECRET, options)

# helper method to know if we have an access token
def authenticated?
  session[:access_token]
end

# handle requests to the application root
get '/' do
  %(<a href="/authorize">Connect with SmartThings</a>)
end

# handle requests to /authorize URL
get '/authorize' do
  'Not Implemented!'
end

# handle requests to /oauth/callback URL. We
# will tell SmartThings to call this URL with our
# authorization code once we've authenticated.
get '/oauth/callback' do
  'Not Implemented!'
end

# handle requests to the /getSwitch URL. This is where
# we will make requests to get information about the configured
# switch.
get '/getswitch' do
  'Not Implemented!'
end
end

```

Create your Gemfile - open a new file in your editor, paste the contents below in, and save it as Gemfile.

```
source 'https://rubygems.org'

gem 'sinatra'
gem 'oauth2'
gem 'json'
```

We'll use bundler to install our app. If you don't have it, you can learn how to get started [here](#).

Back at the command line, run bundle:

```
bundle install
```

You'll also want to set environment variables for your ST_CLIENT_ID and ST_CLIENT_SECRET.

Now, run the app on your local machine:

```
ruby server.rb
```

Visit <http://localhost:4567>. You should see a pretty boring web page with a link to “Connect with SmartThings”.

We're using the [OAuth2 module](#) to handle the OAuth2 flow. We create a new Client, using the `client_id` and `api_key`. We also configure it with the `options` data structure that defines the information about the SmartThings OAuth endpoint.

We've handled the root URL to simply display a link that points to the `/authorize` URL of our server. We'll fill that in next.

6.3.4 Get an Authorization Code

When the user clicks on the “Connect with SmartThings” link, we need to get our OAuth authorization code.

To do this, the user will need to authenticate with SmartThings, and authorize the devices this application can work with. Once that has been done, The user will be directed back to a specified `redirect_url`, with the OAuth authorization code. This will be used (along with the Client ID and secret), to get the access token.

Note: By authorizing the application to work with SmartThings, the SmartApp will be installed into the user's account.

Replace the `/authorize` route with the following:

```
get '/authorize' do
  # Use the OAuth2 module to get the authorize URL.
  # After we authenticate with SmartThings, we will be redirected to the
  # redirect_uri, including our access code used to get the token
  url = client.auth_code.authorize_url(redirect_uri: redirect_uri, scope: 'app')
  redirect url
end
```

Kill the server if it's running (CTRL+C), and start it up again using `ruby server.rb`.

Visit <http://localhost:4567> again, and click the “Connect with SmartThings” link.

This should prompt you to authenticate with your SmartThings account (if you are not already logged in), and bring you to a page where you must authorize this application. It should look something like this:

Click the Authorize button, and you will be redirected back your server.

Authorize IFTTT to access your SmartThings?

IFTTT (<https://ifttt.com>) will be able to...

- View and receive information and events from any of your connected Things.
- Trigger any of your connected Things to take actions on your behalf (like unlocking your door!).
- From:

-

- Allow IFTTT to control these things...

- Which Switches?
 - ☐ Light
- Which Motion Sensors?
 - ☐ SmartSense Motion
- Which Contact Sensors?
 - ☐ Front door
- Which Presence Sensors?
 - ☐ Philip's iPhone
- Which Temperature Sensors?
 - ☐ Front door
- Which Vibration Sensors?
 - ☐ Front door
- Which Water Sensors?
- Which Light Sensors?
- Which Relative Humidity Sensors?
- Which Sirens?
- Which Locks?

You'll notice that we haven't implemented handling this URL yet, so we see "Not Implemented!".

6.3.5 Get an Access Token

When SmartThings redirects back to our application after authorizing, it passes a `code` parameter on the URL. This is the code that we will use to get the API token we need to make requests to our Web Services SmartApp.

We'll store the access token in the session. Towards the top of `server.rb`, we configure our app to use the session, and add a helper method to know if the user has authenticated:

```
# We'll store the access token in the session
use Rack::Session::Pool, :cookie_only => false

def authenticated?
  session[:access_token]
end
```

Replace the `/oauth/callback` route with the following:

```
get '/oauth/callback' do
  # The callback is called with a "code" URL parameter
  # This is the code we can use to get our access token
  code = params[:code]

  # Use the code to get the token.
  response = client.auth_code.get_token(code, redirect_uri: redirect_uri, scope: 'app')

  # now that we have the access token, we will store it in the session
  session[:access_token] = response.token
```

```
# debug - inspect the running console for the
# expires in (seconds from now), and the expires at (in epoch time)
puts 'TOKEN EXPIRES IN ' + response.expires_in.to_s
puts 'TOKEN EXPIRES AT ' + response.expires_at.to_s
redirect '/getswitch'
end
```

We first retrieve the access code from the parameters. We use this to get the token using the OAuth2 module, and store it in the session.

Note: Requesting the token returns JSON which contains information about the token type and the token expiration, in addition to the token itself. The raw response looks something like this:

```
{
  "access_token": "43373fd2871641379ce8b35a9165e803",
  "expires_in": 1576799999,
  "token_type": "bearer"
}
```

The `expires_in` response is the time, in seconds from now, that this token will expire. The time for the token to expire is approximately 50 years from token grant; a refresh token is not sent, but the original token has a very long expiration date.

We then redirect to the `/getswitch` URL of our server. This is where we will retrieve the endpoint to call, and get the status of the configured switch.

Restart your server, and try it out. Once authorized, you should be redirected to the `/getswitch` URL. We'll start implementing that next.

6.3.6 Discover the Endpoint

Now that we have the OAuth token, we can use it to discover the endpoint of our WebServices SmartApp.

Replace the `/getswitch` route with the following:

```
get '/getswitch' do
  # If we get to this URL without having gotten the access token
  # redirect back to root to go through authorization
  if !authenticated?
    redirect '/'
  end

  token = session[:access_token]

  # make a request to the SmartThings endpoint URI, using the token,
  # to get our endpoints
  url = URI.parse(endpoints_uri)
  req = Net::HTTP::Get.new(url.request_uri)

  # we set a HTTP header of "Authorization: Bearer <API Token>"
  req['Authorization'] = 'Bearer ' + token

  http = Net::HTTP.new(url.host, url.port)
  http.use_ssl = (url.scheme == "https")
```

```

response = http.request(req)
json = JSON.parse(response.body)

# debug statement
puts json

# get the endpoint from the JSON:
endpoint = json[0]['url']

'<h3>JSON Response</h3><br/>' + JSON.pretty_generate(json) + '<h3>Endpoint</h3><br/>' + endpoint
end

```

The above code simply makes a GET request to the SmartThings API endpoints service at `https://graph.api.smartthings.com/api/smartapps/endpoints`, setting the "Authorization" HTTP header with the API token.

The response is JSON that contains (among other things), the endpoint of our SmartApp. For this step, we just display the JSON response and endpoint in the page.

By now, you know the drill. Restart your server, refresh the page, and click the link (you'll have to reauthorize). You should then see the JSON response and endpoint displayed on your page.

6.3.7 Make API Calls

Now that we have our token and endpoint, we can (gasp!) make API calls to our SmartApp!

As you may have guessed by the URL path, we're just going to display the name of the switch, and it's current status (on or off).

Remove the line at the end of the `getswitch` route handler that outputs the response HTML, and add the following:

```

# now we can build a URL to our WebServices SmartApp
# we will make a GET request to get information about the switch
switchUrl = 'https://graph.api.smartthings.com' + endpoint + '/switches?access_token=' + token

# debug
puts "SWITCH ENDPOINT: " + switchUrl

getSwitchURL = URI.parse(switchUrl)
getSwitchReq = Net::HTTP::Get.new(getSwitchURL.request_uri)

getSwitchHttp = Net::HTTP.new(url.host, url.port)
getSwitchHttp.use_ssl = true

switchStatus = getSwitchHttp.request(getSwitchReq)

'<h3>Response Code</h3>' + switchStatus.code + '<br/><h3>Response Headers</h3>' + switchStatus.to_hash

```

The above code uses the endpoint for our SmartApp to build a URL, and then makes a GET request to the `/switches` endpoint. It simply displays the the status, headers, and response body returned by our WebServices SmartApp.

Note: Note that we used the `access_token` URL parameter to specify the API key this time, instead of the "Authorization" HTTP header. This is just to illustrate that you can use both methods of passing the API key.

Restart your server and try it out. You should see status of your configured switches displayed!

6.3.8 Summary

In the second part of this tutorial, we learned how an external application can work with SmartThings by getting an access token, discover endpoints, and make API calls to a WebServices SmartApp.

You are encouraged to explore further with this sample, including making different API calls to turn the configured switch on or off.

6.3.9 Source Code

The full source code for this tutorial (both parts), can be found [here](#).

6.3.10 Appendix - Just the URLs, Please

If you want to quickly test getting access to a Web Services SmartApp, without creating an external application, you can use your web browser to make requests to get the API token and endpoint. Most of these steps will not be visible to the end user, but can be useful for testing, or just for reference so you can build your own app.

Here are the steps:

Get the OAuth Authorization Code

In your web browser, paste in the following URL, replacing the CLIENT_ID with your OAuth Client ID:

```
https://graph.api.smarthings.com/oauth/authorize?response_type=code&client_id=CLIENT_ID&scope=app&redirect_uri=
```

Once authenticated, you will be asked to authorize the external application to access your SmartThings. Select some devices to authorize, and click *Authorize*.

This will redirect you to a page that doesn't exist - but that's ok! The important part is the OAuth authorization code, which is the "code" parameter on the URL. Grab this code, and note it somewhere. We'll use it to get our API token.

Get the API token

Important: OAuth Changes

Access token requests have changed to require users to pass their OAuth client ID and secret using HTTP Basic Authentication. This is a security-related improvement, and aligns us closer to the OAuth 2.0 Specification (RFC 6749).

For backwards compatibility, we still support sending the Client ID and secret as POST or GET parameters (outside of the browser context for which the authorization was invoked), but this functionality is deprecated and should be updated as discussed below.

Using the code you just received, and our client ID and secret, we can get our access token. This call must be done **outside of the browser**. The call must include the client ID and secret using HTTP Basic Authentication (we'll use *curl*).

Paste the following into a new terminal window, replacing CLIENT_ID, CLIENT_SECRET, and CODE with the appropriate values:

```
curl -u CLIENT_ID:CLIENT_SECRET 'https://graph.api.smarthings.com/oauth/token?code=CODE&grant_type=a
```

This should return JSON like the following, from which you can get the `access_token`:

```
json {
  "access_token": "43373fd2871641379ce8b35a9165e803",
  "expires_in": 1576799999,
  "token_type": "bearer"
}
```

Discover the Endpoint URL

You can get the endpoint URL for your SmartApp by making a request to the SmartThings endpoints service, specifying your access token.

In your web browser, paste the following into your address bar, replacing `ACCESS_TOKEN` with the access token you retrieved above.

```
https://graph.api.smarthings.com/api/smartapps/endpoints?access_token=ACCESS_TOKEN
```

That should return JSON that contains information about the OAuth client, as well as the endpoint for the SmartApp:

```
[
  {
    "oauthClient": {
      "clientId": "myclient",
      "authorizedGrantTypes": "authorization_code"
    },
    "url": "/api/smartapps/installations/8a2aa0cd3df1a718013df1ca2e3f000c"
  }
]
```

Make API Calls

Now that you have the access token and the endpoint URL, you can make web requests to your SmartApp endpoint using whatever tool you prefer.

Just make sure to preface `http://graph.api.smarthings.com` to the beginning of the URL returned above, and any endpoints your SmartApp exposes (e.g., `/switches`) to the end of the URL.

You can either specify your access token via the `access_token` URL parameter as above, or (preferably) use the Authorization header (`Authorization: Bearer <API TOKEN>`).

Device Handlers

Device Handlers (sometimes also referred to as Device Type Handlers, or SmartDevices), are the virtual representation of a physical device.

If you are new to writing device handlers, start with the [Quick Start](#) (page 143).

After that, read the [Overview](#) (page 147) for a broad discussion about device handlers and where they fit in the SmartThings architecture.

The rest of the guide discusses the various components of Device Handlers primarily targeted for hub-connected (ZigBee or Z-Wave) devices (though the common Device Handler principles and patterns apply to other devices as well).

Note: This guide discusses hub-connected device handlers. For information about LAN and cloud-connected device handlers, see this [guide](#)

Table of Contents:

7.1 Quick Start

Device handlers are the virtual representation of a physical device in the SmartThings platform. They are responsible for communicating between the actual device, and the SmartThings platform.

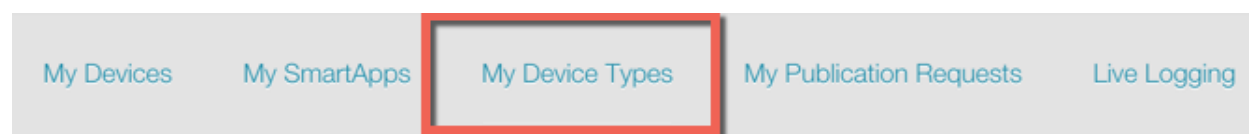
This guide will walk you through getting your first device handler running.

Note: This guide assumes you have created a developer account, and are generally familiar with development in the SmartThings ecosystem.

If you are new to SmartThings development, consider starting with the [Getting Started Guide](#).

7.1.1 Go to My Device Types in IDE

Log in to the [Web IDE](#), and click on the “My Device Types” link on the top menu.



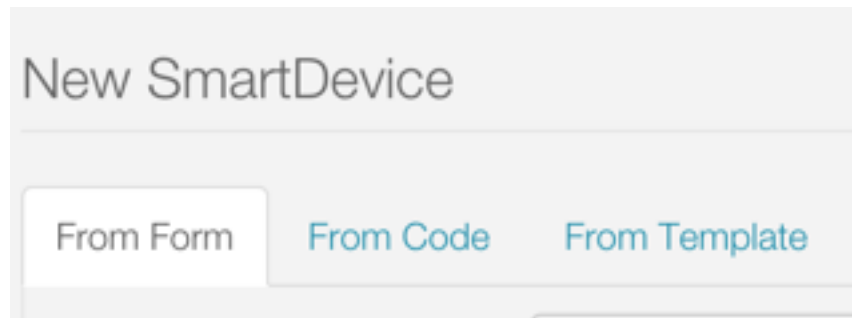
Here you will see all your device handlers, if you have any.

7.1.2 Create a new Device Handler

Create a new device handler by clicking on the “New SmartDevice” button in the upper-right of the page.



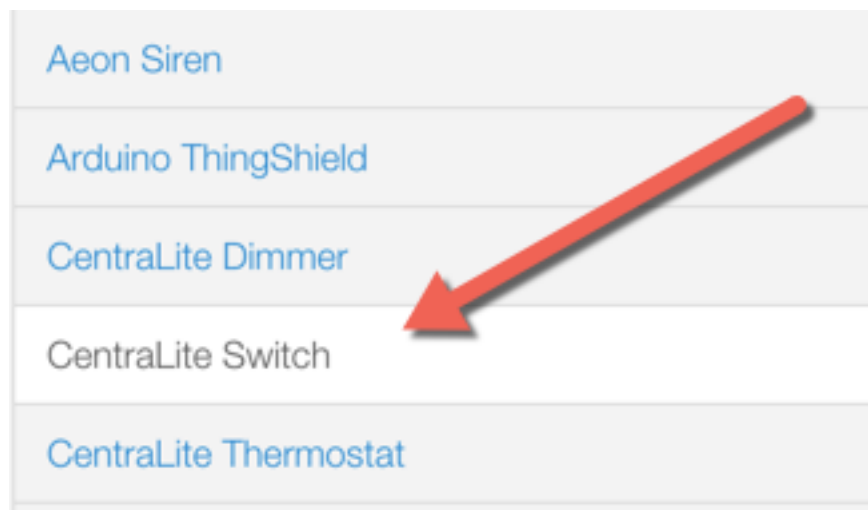
You will see a form for creating a new device type. Note the tabs at the top of the form. You will see a few different options for creating a new device type:



To create a new device handler from a form, use the “From Form” tab. To create a new device handler from some code, use the “From Code” tab. To create a new device handler from a template, use the “From Template” tab.

Go ahead and browse the different forms if you wish, then select the “From Template” tab.

We are going to create a new device handler from the Centralite Switch template. Select the “Centralite Switch” template in the menu on the left, then click the “Create” button.



You will now see the code in the editor of the IDE.

Take a minute to look at the code and its structure. Don’t worry about the details yet - the rest of this guide will address that. For now, just take note of the anatomy of the device handler:


```

metadata {
    // Automatically generated. Make future change here.
    definition (name: "SmartPower Outlet", namespace: "smarththings",
        author: "SmartThings") {
        capability "Actuator"
        capability "Switch"
        capability "Sensor"

        fingerprint profileId: "0104", inClusters: "0000,0003,0006",
            outClusters: "0019"
    }

    // simulator metadata
    simulator {
        // status messages
        status "on": "on/off: 1"
        status "off": "on/off: 0"

        // reply messages
        reply "zcl on-off on": "on/off: 1"
        reply "zcl on-off off": "on/off: 0"
    }

    // UI tile definitions
    tiles {
        standardTile("switch", "device.switch", width: 2, height: 2,
            canChangeIcon: true) {
            state "off", label: '${name}', action: "switch.on",
                icon: "st.switches.switch.off", backgroundColor: "#ffffff"
            state "on", label: '${name}', action: "switch.off",
                icon: "st.switches.switch.on", backgroundColor: "#79b821"
        }

        main "switch"
        details "switch"
    }
}

// Parse incoming device messages to generate events
def parse(String description) {
    if (description?.startsWith("catchall: 0104 000A")) {
        log.debug "Dropping catchall for SmartPower Outlet"
        return []
    } else {
        def name = description?.startsWith("on/off: ") ? "switch" : null
        def value = name == "switch" ? (description?.endsWith(" 1") ?
            "on" : "off") : null
        def result = createEvent(name: name, value: value)
        log.debug "Parse returned ${result?.descriptionText}"
        return result
    }
}

// Commands to device
def on() {
    'zcl on-off on'
}

def off() {
    'zcl on-off off'
}

```

The *definition* metadata is where this device's supported capabilities are listed, along with the device fingerprinting

simulator metadata allows you to simulate device interaction in the IDE.

tiles specify how the device information appears in the mobile app.

The *parse* method is responsible for consuming raw messages from the device, and usually creating SmartThings events from them.

Command definitions. All commands as specified by the declared capabilities must be implemented by the device-type handler.

7.1.3 Make some Changes

Because we installed from a template, we want to change some of the metadata.

In the definition method, change the `name`, to be something like “MY CentraLite Switch”, the `namespace` to be your github user account (or you can leave it blank), and the `author` to be your name.

While we’re here, let’s change some tile names so we see our changes reflected in the simulator (and in the mobile app).

Find the tile definition for the “switch” tile:

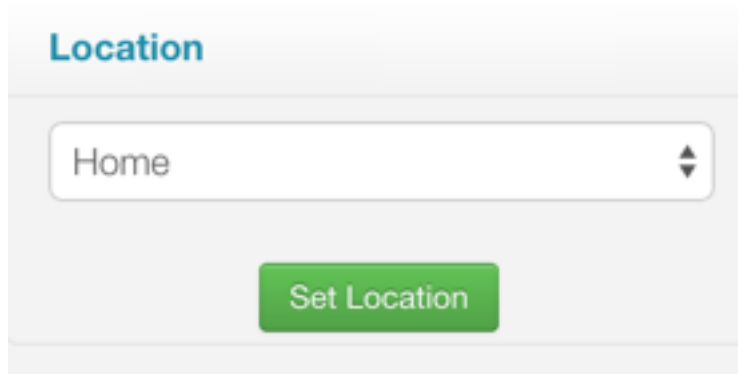
```
standardTile("switch", "device.switch", width: 2, height: 2,
    canChangeIcon: true) {
    state "off", label: '${name}', action: "switch.on",
        icon: "st.switches.switch.off", backgroundColor: "#ffffff"
    state "on", label: 'AM ON', action: "switch.off",
        icon: "st.switches.switch.on", backgroundColor: "#79b821"
}
```

Change the value of the `label` parameters from `'${name}'` to something like “MY ON” or “MY OFF”. Feel free to be more creative than that. :)

Click the “Save” button above the editor.

7.1.4 Install with a Virtual Device

In the right-hand side of the IDE, you will see a drop-down menu where you can choose any of your locations.



Choose a location, and click “Set Location”.

You will now be able to choose a device to test with. For now, select the virtual device (it will likely be selected already).

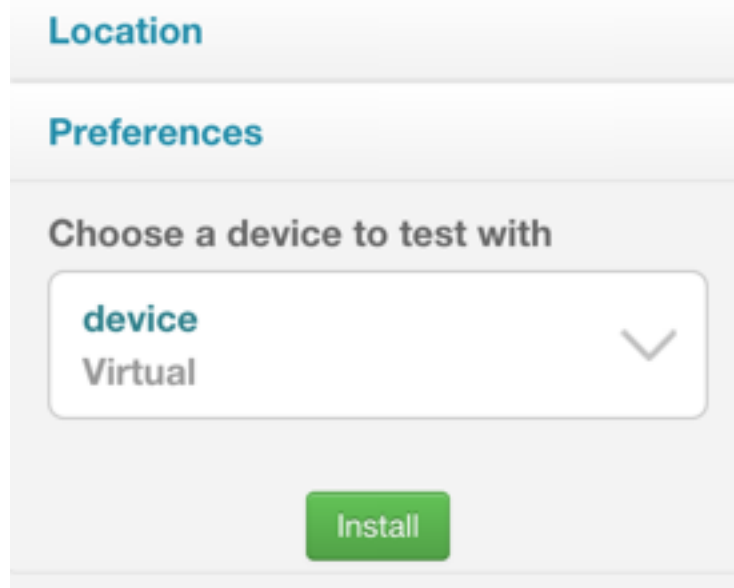
Then click the “Install” button.

You will then see the simulator, with the device tile and Tools in the IDE.

Try clicking on the switch icon in the IDE (notice it should display the label that you changed above). The switch will “turn on”, and you can note the logging in the logging window in the bottom of the IDE.

You can also test sending some messages by selecting a message in the drop-down, and clicking the *play* button.

Towards the bottom of the tools, you’ll see some buttons like “on” and “off”.



These are the commands that your device handler supports. Notice that they are organized by the capability that defines those commands (e.g., “on” and “off” come from the “Switch” capability). You can test sending commands to your device handler. This simulates a SmartApp calling the `on()` command on your device, for example.

Feel free to make some changes, like logging some more information, then saving and re-installing in the simulator.

7.1.5 Bonus Step - Install on a Real Device

If you happen to have a CentraLite switch, you can swap in your new device handler for the default CentraLite switch device-handler.

Go to the [My Devices](#) page in the IDE:

Find your device that has the type “CentraLite Switch”, and click on the display name.

Towards the bottom of the page for the CentraLite Switch, click the “Edit” button. Find the “Type” field, and select your device handler. Click the “Update” button to save your changes.

Your switch is now using your device handler. If you refresh the mobile app (you may need to kill it and restart it), you should see the tile icons updated with whatever label you gave it.

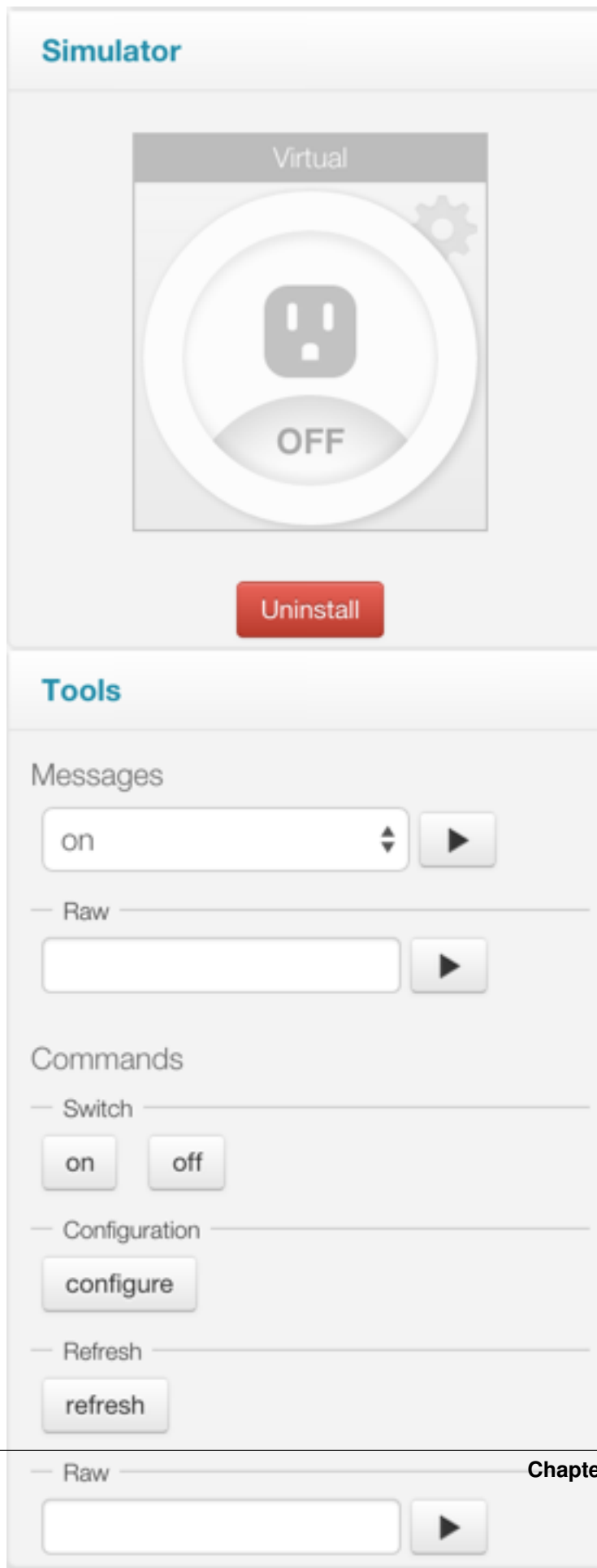
If you make future changes to your device handler, don’t forget to click the “Publish” button after you have saved.

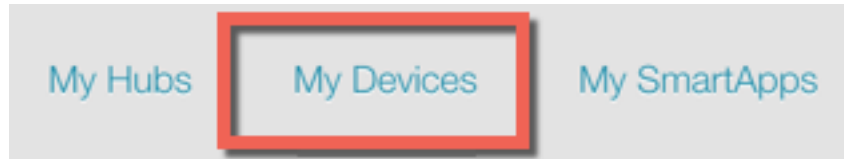
7.1.6 Next Steps

Now that you have created and installed your first device handler, use the rest of this guide to learn more. Start with the [Overview](#) (page 147), and then learn about the various components.

7.2 Overview

The SmartThings architecture provides a unique abstraction of devices from their distinct capabilities and attributes in a way that allows developers to build applications that are insulated from the specifics of which device they are using. For example, there are lots of wirelessly controllable “switches”. A switch is any device that can be turned On or Off.





When a SmartApp interacts with the virtual representation of a device, it knows that the device supports certain actions based on its capabilities. A device that has the “switch” capability must support both the “on” and “off” actions. In this way, all switches are the same, and it doesn’t matter to the SmartApp what kind of switch is actually involved.

This virtual representation of the device is called a device handler, or SmartDevice.

Note: This layer of abstraction is key to the successful function and flexibility of the SmartThings platform. Architecturally, device handlers are the bridge between generic capabilities and the device or protocol specific interface actually used to communicate with the device.

The diagram below depicts where device handlers sit in the SmartThings architecture.

In the example shown above, the job of the device handler (that is implementing the “switch” capability) is to parse incoming, protocol-specific status messages from the device and turn them into normalized “events”. It is also responsible for accepting normalized commands (such as “on” and “off”) and turning those into the protocol-specific commands that can be sent to the device to affect the desired action.

For example, for a Z-Wave compatible on-off switch, the incoming status messages used by the device to report an “on” or “off” state are as shown below:

Device Command	Protocol-Specific Command Message
on	command: 2003, payload: FF
off	command: 2003, payload: 00

Whereas the device status reported to the SmartThings platform for the device is literally just a simple “on” or “off”.

Similarly, when a SmartApp or the mobile app invoked an “on” or “off” command for a switch device, the command that is sent to the device handler is just that simple: “on” or “off”. The device handler must turn that simple command into a protocol-specific message that can be sent down to the device to affect the desired action.

The table below shows the actual Z-Wave commands that are sent to a Z-Wave switch by the device handler.

Device Command	Protocol-Specific Command Message
On	2001FF
Off	200100

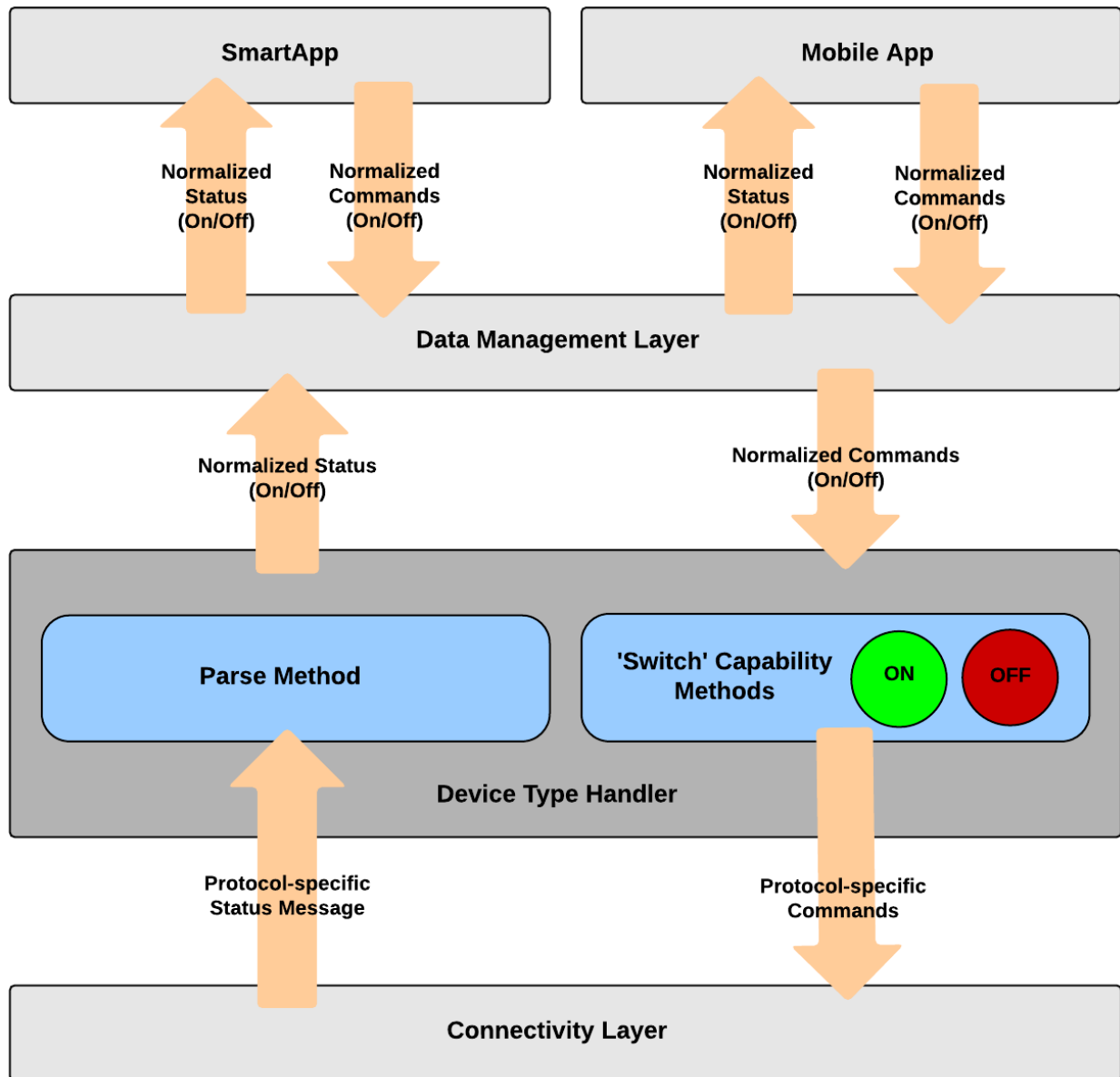
7.2.1 Core Concepts

To understand how device handlers work, a few core concepts need to be discussed.

Capabilities

Capabilities are the interactions that a device allows. They provide an abstraction layer that allows SmartApps to work with devices based on the capabilities they support, and not be tied to a specific manufacturer or model.

Consider the example of the “Switch” capability. In simple terms, a switch is a device that can turn on and off. It may be that a switch in the traditional sense (for example an in-wall light switch), a connected bulb (a Hue or Cree bulb), or even a music player. All of these unique devices have a device handler, and those device handler’s support the “Switch” capability. This allows SmartApps to only require a device that supports the “Switch” capability and thus work with a variety of devices including different manufacturer and model-specific “switches”. The SmartApp



can then interact with the device knowing that it supports the “on” and “off” command (more on commands below), without caring about the specific device being used.

This code illustrates how a SmartApp might interact with a device that supports the “Switch” capability:

```
preferences() {
  section("Control this switch"){
    input "theSwitch", "capability.switch", multiple: false
  }
}

def someEventHandler(evt) {
  if (someCondition) {
    theSwitch.on()
  } else {
    theSwitch.off()
  }

  // logs either "switch is on" or "switch is off"
  log.debug "switch is ${theSwitch.currentSwitch}"
}
```

The above example illustrates how a SmartApp requests a device that supports the “Switch” capability. When installing the SmartApp, the user will be able to select any device that supports the “Switch” capability - be it an in-wall light switch, a connected bulb, a music player, or any other device that supports the “Switch” capability.

The *Capabilities Reference* (page 217) outlines all the supported capabilities.

Device handlers typically support more than one capability. A device handler for a Hue bulb would support the “Switch” capability as well as the “Color Control” capability. This allows SmartApps to be written in a very flexible manner.

Commands and attributes deserve their own discussion - let’s dive in.

Commands

Commands are the actions that your device can do. For example, a switch can turn on or off, a lock can lock or unlock, and a valve can open or close. In the example above, we issue the “on” and “off” command on the switch by invoking the `on()` or `off()` methods.

Commands are implemented as methods on the device handler. When a device supports a capability, it is responsible for implementing all the supported command methods.

Attributes

Attributes represent particular state values for your device. For example, the switch capability defines the attribute “switch”, with possible values of “on” and “off”.

In the example above, we get the value of the “switch” attribute by using the “current<attributeName>” property (`currentSwitch`).

Attribute values are set by creating events where the attribute name is the name of the event, and the attribute value is the value of the event. This is discussed more in the Parse and Events documentation

Like commands, when a device supports a capability, it is responsible for ensuring that all the capability’s attributes are implemented.

Actuator and Sensor

If you look at the [Capabilities Reference](#) (page 217), you'll notice two capabilities that have no attributes or commands - "Actuator" and "Sensor".

These capabilities are "marker" or "tagging" capabilities (if you're familiar with Java, think of the Cloneable interface - it defines no state or behavior).

The "Actuator" capability defines that a device has commands. The "Sensor" capability defines that a device has attributes.

If you are writing a device handler, it is a best practice to support the "Actuator" capability if your device has commands, and the "Sensor" capability if it has attributes. This is why you'll see most device handlers supporting one of, or both, of these capabilities.

The reason for this is convention and forward-looking abilities - it can allow the SmartThings platform to interact with a variety of devices if they *do* something ("Actuator"), or if they report something ("Sensor").

7.2.2 Protocols

SmartThings currently supports both the [Z-Wave](#) and [ZigBee](#) wireless protocols.

Since the device handler is responsible for communicating between the device and the SmartThings platform, it is usually necessary to understand and communicate in whatever protocol the device supports. This guide will discuss both Z-Wave and ZigBee protocols at a high level.

7.2.3 Execution Location

With the original SmartThings hub, all Device handlers execute in the SmartThings cloud. With the new Samsung SmartThings hub, certain Device handlers may run locally on the hub or in the SmartThings cloud. Execution location varies depending on a variety of factors, and is managed by the SmartThings internal team.

As a SmartThings developer, you should write your device handlers to satisfy their specific use cases, regardless of where the handler executes. There is currently no way to specify or force a certain execution location.

7.2.4 Rate Limiting

Like SmartApps, Device Handlers are restricted to executing no more than 250 times in a 60 second window. Execution attempts exceeding this limit will be prevented, and a message will be logged indicating that the limit has been reached. The count will start over when the current time window closes, and the next begins.

Common causes for exceeding this limit are a SmartApp that sends many commands to one device by receiving a large number of event subscriptions (if that doesn't first hit the limit for SmartApps). For example, DLNA players that are extremely chatty or devices that bind to frequently changing energy/power values may also encounter this limit.

7.3 Simulator

Using the IDE simulator, we can model the behavior of the device without actually requiring a physical device.

On the right-hand side of the IDE, after you install a device handler, you'll see the simulator. The image below is the simulator seen after installing the "Z-Wave Switch" device handler (available via the "Browse Device Templates" menu).

Go ahead, try it out. Install the device handler in the IDE, and choose a virtual switch. Modify some of the simulator metadata as you read through this and see what happens.

The purpose of the simulator metadata is to model the behavior of the physical device. Using the simulator, we can test sending messages and commands to our device handler.

There are two types of simulator declarations to define in a device handler - “status” and “reply”.

7.3.1 Status

The “status” declarations specify actions that result in a person physically actuating the device. In the case of the Z-Wave switch, for example, we have:

```
status "on": "command: 2003, payload: FF"
status "off": "command: 2003, payload: 00"
```

status takes a map as an argument. The key (“on” in the example above) is just a name for the action. The value (“command: 2003, payload: FF”) is the message that the device will send to the device handler’s parse (message) method when that action is taken on the physical device.

In the simulator, each status key (“on” or “off” in the example above) will be an available message in the simulator.

7.3.2 Reply

The “reply” declarations specify responses that the physical device will send to the device handler when it receives a certain message from the hub. For a Z-Wave switch, for example, we specify:

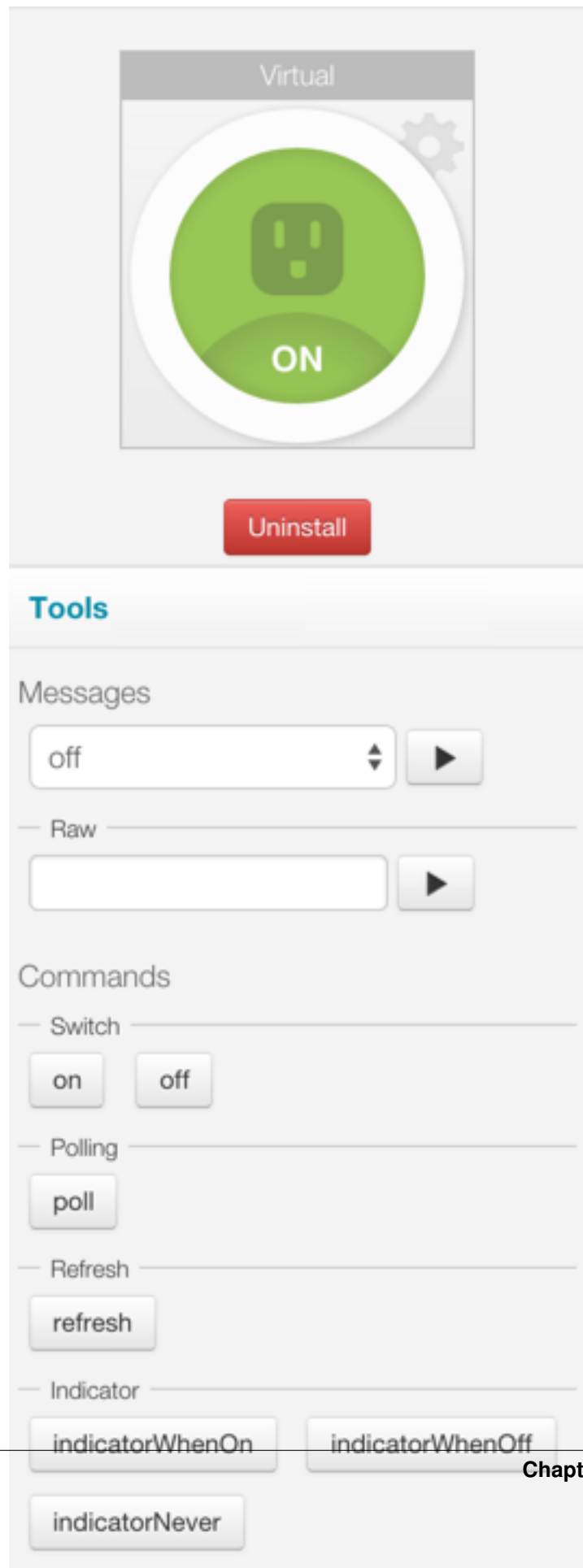
```
reply "2001FF,delay 100,2502": "command: 2503, payload: FF"
reply "200100,delay 100,2502": "command: 2503, payload: 00"
```

Just like status, reply accepts a map as a parameter. The key is a comma-separated list of the raw commands sent to the device, i.e. what’s returned from the device handler’s command methods. For example, the Z-Wave switch commands that send the above methods are:

```
def on() {
    delayBetween([
        zwave.basicV1.basicSet(value: 0xFF).format(),
        zwave.switchBinaryV1.switchBinaryGet().format()
    ])
}

def off() {
    delayBetween([
        zwave.basicV1.basicSet(value: 0x00).format(),
        zwave.switchBinaryV1.switchBinaryGet().format()
    ])
}
```

Those methods will return the values in the first arguments of the reply declarations. The second argument in the reply declarations works the same way as the status declarations - they define messages sent to the parse method. But in this case it’s in response to commands, not physical actuations.



7.3.3 Summary

The purpose of these declarations is to allow a virtual device to function in the IDE simulator, without being attached to a physical device. The `status` method allows us to simulate physical actuation, while the `reply` method allows us to simulate sending messages to the device in response to a command from the hub.

7.4 Definition

The definition metadata defines core information about your device handler. The initial values are set from the values entered when creating your device handler.

Example definition metadata:

```
metadata {
  definition(name: "test device", namespace: "yournamespace", author: "your name") {

    capability "Alarm"
    capability "battery"

    attribute "customAttribute", "string"

    command "customCommand"

    fingerprint profileId: "0104", inClusters: "0000,0003,0006",
                  outClusters: "0019"
  }

  ...
}
```

The definition method takes a map of parameters, and a closure.

The supported parameters are:

name The name of the device handler

namespace The namespace for this device handler. This should be your github user name. This is used when looking up device handlers by name to ensure the correct one is found, even if someone else has used the same name.

author The author of this device handler.

The closure defines the capabilities, attributes, commands, and fingerprint information for your device handler.

7.4.1 Capabilities

To define that your device supports a capability, simply call the `capability` method in the closure passed to `definition`.

The argument to the `capability` method is the Capability name.

```
capability "Actuator"
capability "Power Meter"
capability "Refresh"
capability "Switch"
```

7.4.2 Attributes

If you need to define a custom attribute for your device handler, call the `attribute` method in the closure passed to the definition method:

`attribute(String attributeName, String attributeType, List possibleValues = null)`

attributeName Name of the attribute

attributeType Type of the attribute. Available types are “string”, “number”, and “enum”

possibleValues Optional. The possible values for this attribute. Only valid with the “enum” attributeType.

```
// String attribute with name "someName"
attribute "someName", "string"

// enum attribute with possible values "light" and "dark"
attribute "someOtherName", "enum", ["light", "dark"]
```

7.4.3 Commands

To define a custom command for your device handler, call the `command` method in the closure passed to the definition method:

`command(String commandName, List parameterTypes = [])`

commandName The name of the command. You must also define a method in your device handler with the same name.

parameterTypes Optional. An ordered list of the parameter types for the command method, if needed.

```
// command name "myCommand" with no parameters
command "myCommand"

// comand name myCommandWithParams that takes a string and a number parameter
command "myCommandWithParams", ["string", "number"]

...

// each command specified in the definition must have a corresponding method

def myCommand() {
    // handle command
}

// this command takes parameters as defined in the definition
def myCommandWithParams(stringParam, numberParam) {
    // handle command
}
```

7.4.4 Fingerprinting

When trying to connect your device to the SmartThings hub, we need a way to identify and join a particular device to the hub. This process is known as a “join” process, or “fingerprinting”.

The fingerprinting process is dependent on the type of device you are looking to pair. SmartThings attempts to match devices coming in based on the input and output clusters a device uses, as well as a `profileId` (for ZigBee) or `deviceId`

(for Z-Wave). Basically, by determining what capabilities your device has, SmartThings determines what your device is.

ZigBee Fingerprinting

For ZigBee devices, the main profileIds you will need to use are

- HA: Home Automation (0104)
- SEP: Smart Energy Profile
- ZLL: ZigBee Light Link (C05E)

The input and output clusters are defined specifically by your device and should be available via the device's documentation.

An example of a ZigBee fingerprint definition:

```
fingerprint profileId: "C05E", inClusters: "0000,0003,0004,0005,0006,0008,0300,1000", outClusters: "
```

Z-Wave Fingerprinting

For Z-Wave devices, the fingerprint should include the deviceId of the device and the command classes it supports in the inClusters list. The easiest way to find these values is by adding the actual device to SmartThings and looking for the *Raw Description* in its details view in the SmartThings developer tools. The device class ID is the four-digit hexadecimal number (eg. 0x1001) and the command classes are the two-digit hexadecimal numbers. So if the raw description is

```
0 0 0x1104 0 0 0 8 0x26 0x2B 0x2C 0x27 0x73 0x70 0x86 0x72
```

The fingerprint will be

```
fingerprint deviceId:"0x1104", inClusters:"0x26, 0x2B, 0x2C, 0x27, 0x73, 0x70, 0x86, 0x72"
```

If the raw description has two lists of command classes separated by a single digit 'count' number, the second list is the outClusters. So for the raw description

```
0 0 0x2001 0 8 0x30 0x71 0x72 0x86 0x85 0x84 0x80 0x70 1 0x20
```

The fingerprint will be

```
fingerprint deviceId:"0x2001", inClusters:"0x30, 0x71, 0x72, 0x86, 0x85, 0x84, 0x80, 0x70", outClusters:"0x20"
```

Note that the fingerprint clusters lists are comma separated while the raw description is not.

Fingerprinting Best Practices and Important Information

Include Manufacturer and Model

Try and include the manufacturer and model name to your fingerprint (only supported for ZigBee devices right now - you can add them to Z-Wave devices as well, but they won't be used yet):

```
fingerprint inClusters: "0000,0001,0003,0020,0406,0500", manufacturer: "NYCE", model: "3014"
```

When adding the manufacturer model and name, you'll likely need to add the following raw commands in the `refresh()` command. This is used to report back the manufacturer/model name from the device.

The reply will be hexadecimal that you can convert to ascii using the hex-to-ascii converter (we'll be adding a utility method to do this, but in the meantime you can use an online converter like [this one](#)):

```
def refresh() {
    "st rattr 0x${zigbee.deviceNetworkId} 0x${zigbee.endpointId} 0 4", "delay 200",
    "st rattr 0x${zigbee.deviceNetworkId} 0x${zigbee.endpointId} 0 5"
}
```

Adding Multiple Fingerprints

You can have multiple fingerprints. This is often desirable when a Device Handler should work with multiple versions of a device.

The platform will use the fingerprint with the longest possible match.

Device Pairing Process

The order of the `inClusters` and `outClusters` lists is not important to the pairing process. It is a best practice, however, to list the clusters in ascending order.

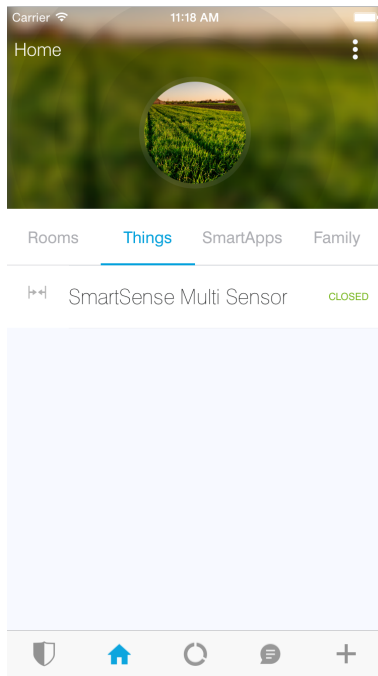
The device can have more clusters than the fingerprint specifies, and it will still pair. If one of the clusters specified in the fingerprint is incorrect, the device will *not* pair.

7.5 Tiles

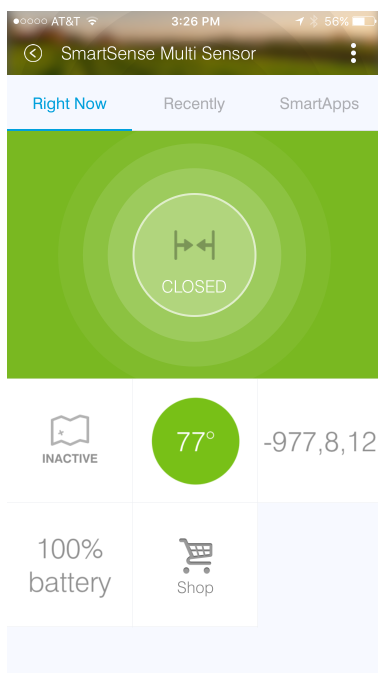
Tiles define how devices are represented in the SmartThings mobile application. There are currently two main areas where devices are viewed.

Note: Be sure to check out `multiAttributeTile()` (page 165) below for new tile layout options.

The Things view is where you can see all of your devices listed.



When tapping on one of the devices in the Things view, you will be presented with the Details view for the device.



When creating a Device Handler, you define how it will appear for the user on their Details screen by defining and configuring different Tiles.

Tiles are defined inside the metadata block of device handlers. Let's take a look at how we can define Tiles in our device handlers.

7.5.1 Overview

Developers have control over the look and feel of the Details view by defining Tiles.

Tiles are defined in the Device Handler by calling the `tiles()` method. The `tiles()` method is composed of tile definitions, and layout information (the `main` and `details` method). There are five types of tiles that you can use within your Device Handler. Each tile serves a different purpose.

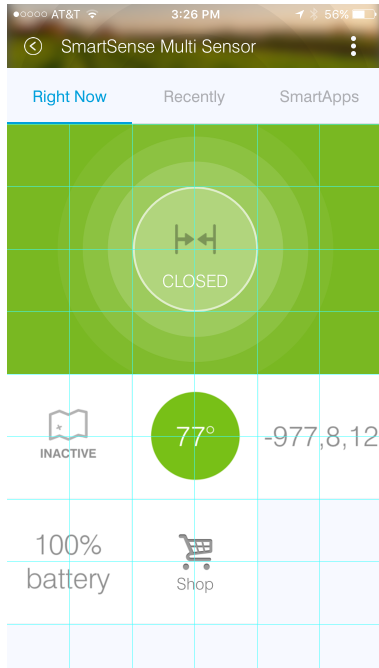
Consider this tiles block for the Multipurpose Sensor from the screenshot above:

```
tiles(scale: 2) {
    multiAttributeTile(name:"status", type: "generic", width: 6, height: 4){
        tileAttribute ("device.status", key: "PRIMARY_CONTROL") {
            attributeState "open", label:'${name}', icon:"st.contact.contact.open", backgroundColor:
            attributeState "closed", label:'${name}', icon:"st.contact.contact.closed", backgroundCo:
            attributeState "garage-open", label:'Open', icon:"st.doors.garage.garage-open", backgroun
            attributeState "garage-closed", label:'Closed', icon:"st.doors.garage.garage-closed", bac
        }
    }
    standardTile("contact", "device.contact", width: 2, height: 2) {
        state("open", label:'${name}', icon:"st.contact.contact.open", backgroundColor:"#ffa81e")
        state("closed", label:'${name}', icon:"st.contact.contact.closed", backgroundColor:"#79b821")
    }
    standardTile("acceleration", "device.acceleration", width: 2, height: 2) {
        state("active", label:'${name}', icon:"st.motion.acceleration.active", backgroundColor:"#53a
        state("inactive", label:'${name}', icon:"st.motion.acceleration.inactive", backgroundColor:"
    }
    valueTile("temperature", "device.temperature", width: 2, height: 2) {
        state("temperature", label:'${currentValue}°',
            backgroundColors:[
                [value: 31, color: "#153591"],
                [value: 44, color: "#1e9cbb"],
                [value: 59, color: "#90d2a7"],
                [value: 74, color: "#44b621"],
                [value: 84, color: "#f1d801"],
                [value: 95, color: "#d04e00"],
                [value: 96, color: "#bc2323"]
            ]
        )
    }
    valueTile("3axis", "device.threeAxis", decoration: "flat", wordWrap: false, width: 2, height: 2) {
        state("threeAxis", label:'${currentValue}', unit:"", backgroundColor:"#ffffff")
    }
    valueTile("battery", "device.battery", decoration: "flat", inactiveLabel: false, width: 2, height: 2) {
        state "battery", label:'${currentValue}% battery', unit:""
    }
    standardTile("refresh", "device.refresh", inactiveLabel: false, decoration: "flat", width: 2, height: 2) {
        state "default", action:"refresh.refresh", icon:"st.secondary.refresh"
    }

    main(["status", "acceleration", "temperature"])
    details(["status", "acceleration", "temperature", "3axis", "battery", "refresh"])
}
```

Tiles are defined with either a `scale: 1` (default) or `scale: 2` argument. The value of 2 will enable the *6 X Unlimited* grid layout. If the `scale` argument is not supplied, it will be set to the default value of 1.

Here you can see how the tiles defined above are laid out using the *6 X Unlimited* grid (using the `scale: 2` option):



Note: The grid layout can be a 3 column, unlimited row, grid system or a *6 X Unlimited* grid to be more visually appealing and to give developers more flexibility when defining layouts. New *6 X Unlimited* tiles will be scaled back on older versions of the SmartThings mobile app that do not support the *6 X Unlimited* grid layout.

The first argument to the tile methods (`standardTile()`, `valueTile()`, etc.) is the name of the tile. This is used to identify the tile when specifying the tile layout.

The second argument is the attribute this tile is associated with. Each tile is associated with an attribute of the device.

In the example above, a `standardTile()` (more on that later) is created with the name "contact", for the "contact" attribute. The convention is to prefix the attribute name with "device" - so the format is "device.<attributeName>".

The contact attribute has two possible values: "open", and "closed". Since we want the display to change depending on if the contact is open or closed, we define a state for each. The `state()` method allows us to specify display information like icon and background color for each state, as well as specify what action should happen when the tile is interacted with in its current state.

The `state()` method is discussed later in this document.

Common Tile Parameters

All tiles support the following parameters:

width number - controls how wide this tile is. Default is 1.

height number - controls how tall this tile is. Default is 1.

canChangeIcon boolean - `true` to allow the user to pick their own icon. Defaults to `false`.

canChangeBackground boolean - `true` to allow a user to choose their own background image for the tile. Defaults to `false`.

decoration String - specify "flat" for the tile to render without a ring.

Note: You may see Device Handlers that use the `inactiveLabel` property. This is deprecated and has no effect.

7.5.2 State

Each tile can have one or more `state()` definitions.

Let's consider a switch tile definition example:

```
standardTile("switchTile", "device.switch", width: 2, height: 2,
    canChangeIcon: true) {
    state "off", label: '${name}', action: "switch.on",
        icon: "st.switches.switch.off", backgroundColor: "#ffffff"
    state "on", label: '${name}', action: "switch.off",
        icon: "st.switches.switch.on", backgroundColor: "#E60000"
}
```

Important: Notice anything strange about the `label` value for state? It appears to be using Groovy's string interpolation syntax (`${}`), but with a **single quote**. In Groovy, String interpolation is only possible for strings defined in double quotes. So, what gives?

When the SmartThings platform executes the `tiles()` method you have defined, it doesn't yet know anything about the actual devices. Only later, when the device details screen is rendered in the mobile client, does the platform know information about the specific devices.

So, we use single quotes for the label (`${name}`) because the platform can then manually substitute the actual value later, when it is available.

Long story short - the above is not a typo. Use single quotes for interpolated string values in the tiles definition.

The "switch" attribute specifies two possible values - "on" and "off". We define a state for each possible value. The first argument to the `state()` method should be the value of the attribute this state applies to (there is an exception to this rule discussed below).

When the switch is off, and the user presses on the tile on their mobile device, we want to turn the switch on. We specify this action using the `action` parameter.

The value of the `action` parameter should be the name of the command to invoke. The convention is to prefix the command name with the capability, so in the example above we have `"switch.on"`.

State Selection

The following algorithm is used to determine which state to display, when there are multiple states:

1. If a state is defined for the attribute's current value, it will render that.
2. If no state exists for the attribute value, it will render a state that has specified `defaultState: true`. Use this in place of the "default" state name that you may see in some device handlers.
3. If no state matches the above rules, it will render the first state declaration.

State Parameters

The valid parameters are:

action String - The action to take when this tile is pressed. The form is `<capabilityReference>.<command>`.

backgroundColor String - A hexadecimal color code to use for the background color. This has no effect if the tile has `decoration: "flat"`.

backgroundColors List - Specify a list of maps of attribute values and colors. The mobile app will match and interpolate between these entries to select a color based on the value of the attribute.

defaultState boolean - Specify `true` if this state should be the active state displayed for this tile. See the [State Selection](#) (page 162) topic above for more information.

icon String - The identifier of the icon to use for this state. You can view the icon options [here](#). iOS devices support specifying a URL to a custom image.

label String - The label for this state.

Note: The example above uses some attributes within our state method. We use the `name` and `currentValue` attributes to make our state definition more dynamic.

7.5.3 Tile Definitions

standardTile()

Use a standard tile to display current state information. For example, to show that a switch is on or off, or that there is or is not motion.

```
standardTile("water", "device.water", width: 2, height: 2) {
    state "dry", icon:"st.alarm.water.dry", backgroundColor:"#ffffff"
    state "wet", icon:"st.alarm.water.wet", backgroundColor:"#53a7c0"
}
```

The above tile definition would render as (when wet):



controlTile()

Use a control tile to display a tile that allows the user to input a value within a range. A common use case for a control tile is a light dimmer.

In addition to name and attribute parameters, `controlTile()` requires a third argument to specify the type of control. The valid arguments are “slider” and “color”.

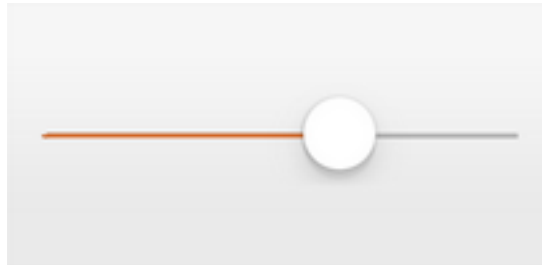
name Name of this tile.

attribute Attribute that this tile displays

type The type of control. Valid types are “slider” and “color”

```
controlTile("levelSliderControl", "device.level", "slider",  
            height: 1, width: 2) {  
    state "level", action:"switch level.setLevel"  
}
```

This renders as:



You can also specify a custom range by using a range parameter. It is a string, and is in the form "<lower bound>..<upper bound>"

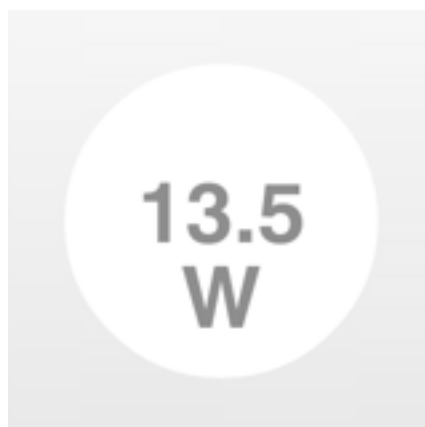
```
controlTile("levelSliderControl", "device.level", "slider", height: 1,  
            width: 2, inactiveLabel: false, range:"(0..100)") {  
    state "level", action:"switch level.setLevel"  
}
```

valueTile()

Use a value tile to display a tile that displays a specific value. Typical examples include temperature, humidity, or power values.

```
valueTile("power", "device.power", decoration: "flat") {  
    state "power", label:'${currentValue} W'  
}
```

This renders as:

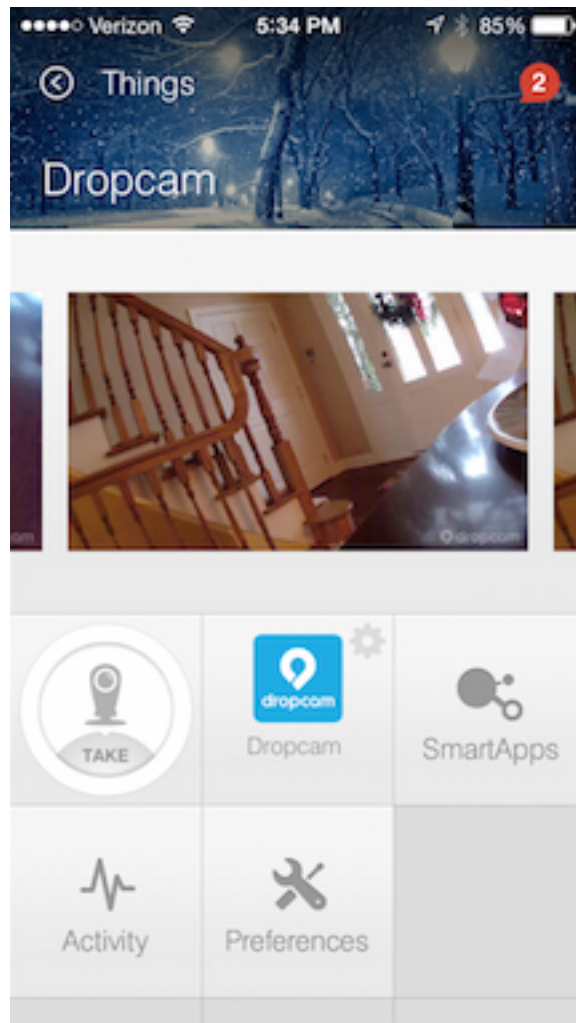


carouselTile()

A carousel tile is often used in conjunction with the Image Capture capability, to allow users to scroll through recent pictures.

Many of the camera Device Handlers will make use of the `carouselTile()`.

```
carouselTile("cameraDetails", "device.image", width: 3, height: 2) { }
```



multiAttributeTile()

Multi-Attribute Tiles are a new kind of tile that incorporate multiple attributes into one tile. They are meant to combine several different attributes into one 6X4 tile. Here are some of the types of tiles that you can create:

Lighting	Thermostat	Video Player	Generic (Default)

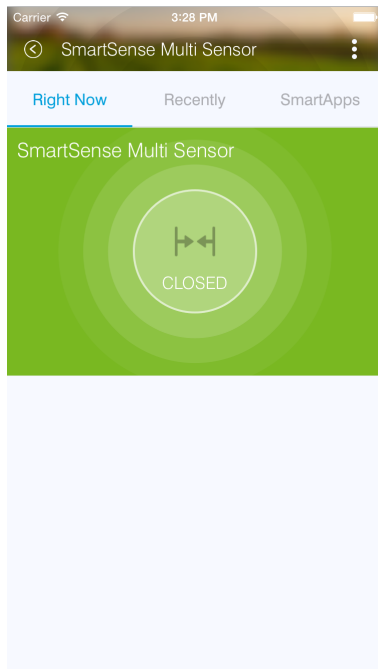
Multi-Attribute Tiles must have a width of 6 and a height of 4. This means that the tiles block of your Device Handler must use the new *6 X Unlimited* grid layout.

```
tiles(scale: 2) {  
    ...  
}
```

The `multiAttributeTile()` method works much like any of the other tile methods currently available. Let's look at an example of a simple generic tile for a contact sensor.

```
tiles(scale: 2) {  
    multiAttributeTile(name:"richcontact", type:"generic", width:6, height:4) {  
        tileAttribute("device.contact", key: "PRIMARY_CONTROL") {  
            attributeState "open", label: '${name}', icon:"st.contact.contact.open", backgroundColor:"#ffa  
            attributeState "closed", label: '${name}', icon:"st.contact.contact.closed", backgroundColor:"#  
        }  
    }  
  
    main "richcontact"  
    details "richcontact"  
}
```

This code should render a device details page that looks like this:



The `multiAttributeTile()` method takes the same parameters as any other tile except for the `type` attribute. Valid options for `type` are "generic", "lighting", "thermostat", and "video".

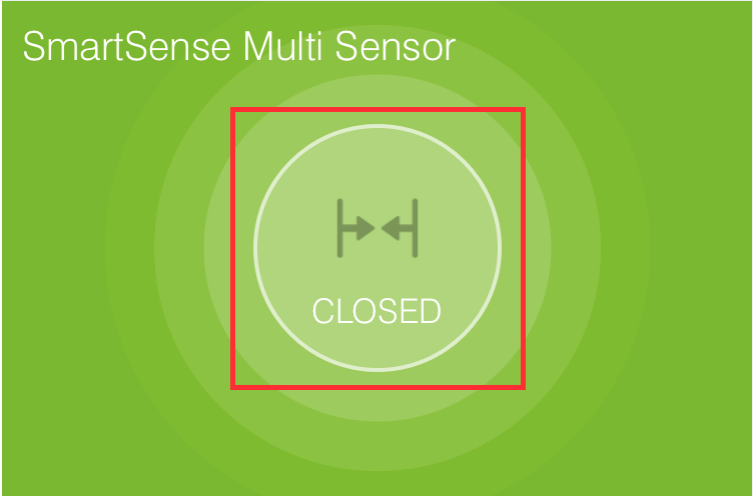
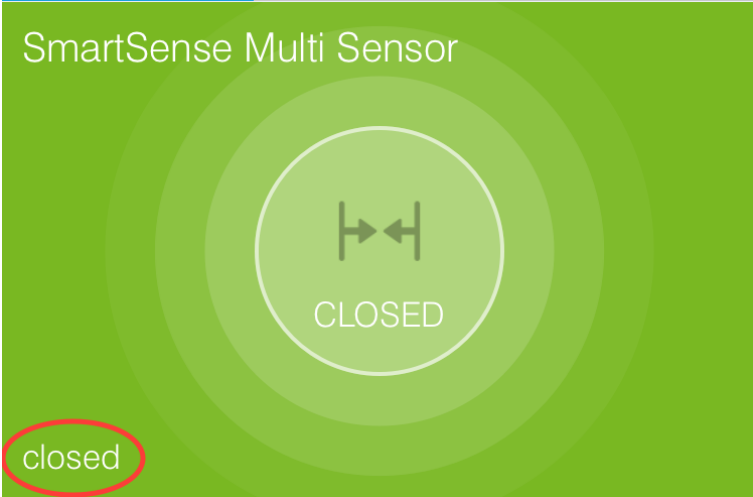
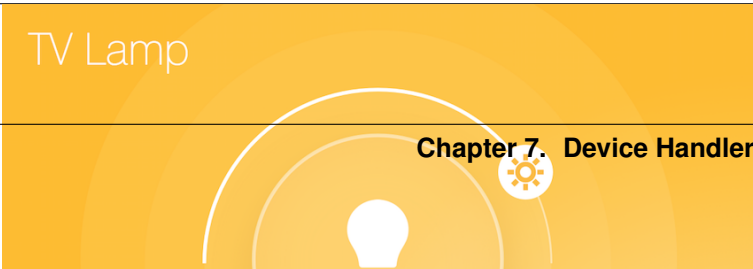
Note: The `multiAttributeTile()` `type` option are currently a placeholder. The specified type does not change how the tile will appear.

Also worth noting is that you may see other types of tiles in existing Device Handlers. Tiles that are not documented here should be considered experimental, and subject to change.

Multi-Attribute Tiles support a new child method parameter called `tileAttribute()`. This is where the real power of multi-attribute tiles comes into play. Each `tileAttribute()` declaration defines an attribute that should be visible on the multi attribute tile. The `tileAttribute()` method currently supports two parameters:

tileAttribute(attribute, key)

- The `attribute` parameter is the device attribute that the tile attribute represents. For example, `device.contact` or `device.level`.
- the `key` parameter can have the following values:

Value	Meaning	Example
PRI-MARY_CONTROL	Main control in middle	 The image shows a green circular UI for a 'SmartSense Multi Sensor'. In the center is a white circle containing a double-headed arrow icon and the word 'CLOSED'. A red rectangle highlights this central circle.
SEC-ONDARY_CONTROL	Textual status message	 The image shows the same green circular UI for a 'SmartSense Multi Sensor'. In the bottom-left corner, the word 'closed' is written in white. A red circle highlights this text.
SLIDER_CONTROL	Slider above primary control	 The image shows an orange circular UI for a 'TV Lamp'. It features a white lightbulb icon in the center with the word 'ON' below it. A red rectangle highlights a slider control consisting of a white arc and a gear icon positioned above the lightbulb.
168		 The image shows the same orange circular UI for a 'TV Lamp'. A red circle highlights a gear icon in the bottom-right area of the UI.

Note: The color of the multi-attribute tile is controlled by the PRIMARY_CONTROL tile attribute. It will default to a light gray color. If the PRIMARY_CONTROL attribute contains states that change the color, the color of the multi attribute tile will also change.

The last piece of the puzzle is *state*. `tileAttribute()` can support *states* just like other tile types. This is done with the new method `attributeState()`. From the contact example above:

```
tileAttribute("device.contact", key: "PRIMARY_CONTROL") {
    attributeState "open", label: '${name}', icon: "st.contact.contact.open", backgroundColor: "#ffa81e"
    attributeState "closed", label: '${name}', icon: "st.contact.contact.closed", backgroundColor: "#79b82c"
}
```

This will render the main control in the middle (because the key is specified as "PRIMARY_CONTROL", with the label either “open” or “closed”, the appropriate icon, and a yellow color for the open state and green for closed. You can also supply actions just as you would for `state()`, to trigger actions when tapping on the control. `attributeState()` is just like `state` but for `tileAttribute()`.

7.5.4 Tile Layouts

To control which tile shows up on the things screen, use the `main` method in the `tiles` closure. The `main` method also supports a list argument just like the `details` method. When given a list, the `main` method will allow the user to choose which tile will be visible on the Things screen. The `details` method defines an ordered list (will render from left-to-right, top-to-bottom) of tiles to display on the tile details screen.

```
tiles {
    // tile definitions. Assume tiles named "tileName1"
    // and "tileName2" created here.

    main "tileName1"
    details(["tileName1", "tileName2"])
}
```

7.5.5 Examples

All Devices define `tiles`, but here are a few examples (ordered from simpler to more complex) that illustrate using many of the tiles discussed above:

- [SmartSense Multi Sensor](#)
- [Centralite Dimmer](#)
- [Hue Bulb](#)

7.6 Preferences

Note: This documentation is incomplete. Until it is expanded, you are encouraged to look at other device-type handlers for example usage. The *SmartSense Multi* and *HomeSeer Multisensor*, available to browse via the “Device Type Templates” menu in the IDE, both use preferences.

Device type handlers can define preferences, similar to how SmartApps do. They are not the same, however.

When you add a device, in addition to the “name your device” field you could show other fields, and they’ll be editable by tapping the “preferences” tile in the device details. This is a fairly uncommon scenario, but would be handled by adding a preferences block to the metadata:

```
metadata {
    ...
    preferences {
        input "sampleInput", "number", title: "Sample Input Title",
            description: "This is the sample input.", defaultValue: 20,
            required: false, displayDuringSetup: true
    }
    ...
}
```

Device preferences are limited to input elements, and do not support multiple pages or sections.

7.7 Parse & Events

The `parse` method is the core method in a typical device handler.

All messages from the device are passed to the `parse` method. It is responsible for turning those messages into something the SmartThings platform can understand.

Because the `parse` method is responsible for handling raw device messages, their implementations vary greatly across different device types. This guide will not discuss all these different scenarios (see the Z-Wave Device Handler Guide or ZibBee Device Handler guide for protocol-specific information).

Consider an example of a simplified `parse` method (modified from the Centralite Switch):

```
def parse(String description) {
    log.debug "parse description: $description"

    def attrName = null
    def attrValue = null

    if (description?.startsWith("on/off:")) {
        log.debug "switch command"
        attrName = "switch"
        attrValue = description?.endsWith("1") ? "on" : "off"
    }

    def result = createEvent(name: attrName, value: attrValue)

    log.debug "Parse returned ${result?.descriptionText}"
    return result
}
```

Our `parse` method inspects the passed-in description, and creates an event with name “switch” and a value of “on” or “off”. It then returns the created event, where the SmartThings platform will handle firing the event and notifying any SmartApps subscribed to that event.

7.7.1 Parse, Events, and Attributes

Recall that the “switch” capability specifies an attribute of “switch”, with possible values “on” and “off”. *The parse method is responsible for creating events for the attributes of that device’s capabilities.*

That is a critical point to understand about device handlers - it is what allows SmartApps to respond to event subscriptions!

Note: Only events that constitute a state change are propagated through the SmartThings platform. A state change is when a particular attribute of the device changes. This is handled automatically by the platform, but should you want to override that behavior, you can do so by specifying the `isStateChange` parameter discussed below.

Creating Events

Use the `createEvent` method to create events in your device handler. It takes a map of parameters as an argument. You should provide the `name` and `value` at a minimum.

Important: The `createEvent` just creates a data structure (a Map) with information about the event. *It does not actually fire an event.*

Only by returning that created map from your `parse` method will an event be fired by the SmartThings platform.

The parameters you can pass to `createEvent` are:

name (required) String - The name of the event. Typically corresponds to an attribute name of the device-handler's capabilities.

value (required) The value of the event. The value is stored as a String, but you can pass in numbers or other objects. SmartApps will be responsible for parsing the event's value into back to its desired form (e.g., parsing a number from a string)

descriptionText String - The description of this event. This appears in the mobile application activity feed for the device. If not specified, this will be created using the event name and value.

displayed boolean - `true` to display this event in the mobile application activity feed. `false` to not display this event. Defaults to `true`.

linkText String - Name of the event to show in the mobile application activity feed, if specified.

isStateChange boolean - `true` if this event caused the device's attribute to change state. `false` otherwise. If not provided, `createEvent` will populate this based on the current state of the device.

unit String - a unit string, if desired. This will be used to create the `descriptionText` if it (the `descriptionText` parameter) is not specified.

Multiple Events

You are not limited to returning a single event map from your `parse` method.

You can return a list of event maps to tell the SmartThings platform to generate multiple events:

```
def parse(String description) {  
    ...  
  
    def evt1 = createEvent(name: "someName", value: "someValue")  
    def evt2 = createEvent(name: "someOtherName", value: "someOtherValue")  
  
    return [evt1, evt2]  
}
```

Generating Events Outside of parse

If you need to generate an event outside of the `parse` method, you can use the `sendEvent` method. It simply calls `createEvent` and fires the event. You pass in the same parameters as you do to `createEvent`.

7.7.2 Tips

When creating a device handler, determining what messages need to be handled by the `parse` method varies by device. A common practice to figure out what messages need to be handled is to simply log the messages in your `parse` method (“`log.debug “description: $description”`”). This allows you to see what the incoming message is for various actuations or states.

7.8 Z-Wave Primer

This document covers some important aspects of the Z-Wave application-level standard that you may come in contact with when developing device handlers for Z-Wave devices. If you are already familiar with Z-Wave development, you can learn how SmartThings integrates with it in [Building Z-Wave Device Handlers](#).

7.8.1 Command Classes

Z-Wave device messages are all called “commands”, even if they are just info reports or other kinds of communications. They are organized into *command classes* which group related functionality together. Some devices list which command classes they support in their manuals.

There is a list of the command classes that SmartThings supports here: [Z-Wave Command Reference](#). Notice some of them have multiple versions. The Z-Wave standard occasionally adds a new version of a command class that may add new commands or add more data fields to existing commands. New versions are backwards-compatible and generally our command parsing system can handle different versions interchangeably, but you may need to specify a specific version in some cases.

Some commonly seen command classes:

- [0x20 Basic](#)
A generalized get/set/report command class that all devices support. It is usually mapped to another more specific command class, like Switch Binary for switches or Sensor Binary for sensors.
- [0x25 Switch Binary](#)
Control of on/off switches.
- [0x26 Switch Multilevel](#)
Control of dimmer switches.
- [0x30 Sensor Binary](#)
Sensors with two states, such as motion detectors and open/closed sensors.
- [0x31 Sensor Multilevel](#)
Sensors that report a numeric value, like temperature or illuminance.
- [0x32 Meter](#)
Outlets and meters that measure energy use.
- [0x71 Alarm/Notification](#)
The *Alarm* command class was renamed to *Notification* in version 3.
Used by sensors and other devices to report events.

- [0x70 Configuration](#)
See *Configuration* section below.
- [0x80 Battery](#)
Battery level reporting for battery powered devices.
- [0x84 Wake Up](#)
See *Listening and Sleepy Devices* section below.
- [0x85 Association](#)
See *Association* section below.
- [0x86 Version](#)
All devices report their Z-Wave framework and firmware version on request.
- [0x72 Manufacturer Specific](#)
All devices report their manufacturer and model (via numeric code).
- [0x98 Security](#)
Commands to and from security-sensitive devices can be sent encrypted by wrapping them in *SecurityMessageEncapsulation* commands.
- [0x60 Multi-Channel/Multi-Instance](#)
The *Multi Instance* command class was renamed to *Multi Channel* in version 3. It is used by devices to distinguish between multiple control or reporting end points.

7.8.2 Listening and Sleepy Devices

Z-Wave devices that are plugged in to power are called **listening** devices because they keep their receiver on all the time. Listening devices act as repeaters and therefore extend the Z-Wave mesh network.

Battery powered Z-Wave devices such as sensors or remote controllers are **sleepy** – they turn off their receivers to save energy, so you can’t send them commands at any time. Instead, they wake up at a regular interval and send a [WakeUpNotification](#) to alert other devices that they will be listening for incoming commands for the next few seconds. The [WakeUpIntervalSet](#) command is used to configure both how often the device will wake up and which controller it will send its *WakeUpNotification* to. When the controller gets the *WakeUpNotification* and has no commands to send to the device, it can send [WakeUpNoMoreInformation](#) to tell the device that it can go back to sleep.

Some battery powered devices like door locks and thermostats have to be able to receive commands at any time. These are known as **beamable** devices, because they wake up for only a tiny slice of time each second or quarter-second and listen for a “beam”. Thus, the sending device must “beam” the receiving device for a full second to wake it up fully before sending a command. This makes communication with these devices take a significantly longer time than with a normal listening device.

7.8.3 Configuration

A Z-Wave device can use the [Configuration](#) command class to allow the user to change its settings. Configuration parameters and their interpretation vary between device models, and are usually detailed in the device’s manual or technical documentation.

The command class includes commands to read and set configuration parameter values. One thing to be careful of is that the [ConfigurationSet](#) command encodes the setting value in a 1, 2, or 4 byte format, and many devices will only properly interpret the value if it is sent in the same byte format. When sending a *ConfigurationSet*, make sure to set the ‘size’ argument to the same value as it has in an incoming [ConfigurationReport](#) from the device for the parameter number in question.

7.8.4 Association

The [Association](#) command class is used to tell a Z-Wave device that it should send updates to another device. It provides the ability to add associated devices to different numbered groups that can have different meanings. This functionality is used in a few different ways, often detailed in the device's manual or technical documentation.

- Some sensors will send reports of the events they detect only to devices that have been added to a specific association group.
- Many sensors will send [BasicSet](#) commands to associated devices, for example to turn a light on when a door opens and off when it closes.
- Some devices have multiple groups for different uses, like group 1 gets sent *BasicSet* commands, group 2 gets sent *SensorBinaryReport* events, and group 3 gets sent *BatteryReport* updates.
- Most door locks will send status updates to associated devices when they are locked or unlocked manually.

The SmartThings hub automatically adds itself to association group 1 when a device that supports association joins the network. If this is inappropriate for your device type, your device handler can use [AssociationRemove](#) to undo it. To associate to a group higher than 1, the device handler can send [AssociationSet](#). The hub's node ID is provided to device handler code in the variable `zwaveHubNodeId`.

7.9 Building Z-Wave Device Handlers

Z-Wave is a proprietary protocol, so we cannot document the full details of the interface. Instead, device handlers use command objects that represent the standard commands and messages that Z-Wave devices use to send and request information.

7.9.1 Parsing Events

When events from Z-Wave devices are passed into your device handler's `parse` method, they are in an encoded string format. The first thing your `parse` method should do is call `zwave.parse` on the description string to convert it to a Z-Wave command object. The object's class is one of the subclasses of `physicalgraph.zwave.Command` that can be found in the [Z-Wave Command Reference](#). If the description string does not represent a valid Z-Wave command, `zwave.parse` will return `null`.

```
def parse(String description) {
    def result = null
    def cmd = zwave.parse(description)
    if (cmd) {
        result = zwaveEvent(cmd)
        log.debug "Parsed ${cmd} to ${result.inspect()}"
    } else {
        log.debug "Non-parsed event: ${description}"
    }
    return result
}
```

Once you have a command object, the recommended way of handling it is to pass it to a overloaded function such as `zwaveEvent` used in this example, with different argument types for the different types of commands you intend to handle:

```
def zwaveEvent(physicalgraph.zwave.commands.basicv1.BasicReport cmd)
{
    def result
    if (cmd.value == 0) {
```

```

        result = createEvent(name: "switch", value: "off")
    } else {
        result = createEvent(name: "switch", value: "on")
    }
    return result
}

def zwaveEvent(physicalgraph.zwave.commands.meterv3.MeterReport cmd) {
    def result
    if (cmd.scale == 0) {
        result = createEvent(name: "energy", value: cmd.scaledMeterValue, unit: "kWh")
    } else if (cmd.scale == 1) {
        result = createEvent(name: "energy", value: cmd.scaledMeterValue, unit: "kVAh")
    } else {
        result = createEvent(name: "power", value: cmd.scaledMeterValue, unit: "W")
    }
    return result
}

def zwaveEvent(physicalgraph.zwave.Command cmd) {
    // This will capture any commands not handled by other instances of zwaveEvent
    // and is recommended for development so you can see every command the device sends
    return createEvent(descriptionText: "${device.displayName}: ${cmd}")
}

```

Remember that when you use `createEvent` to build an event, the resulting map must be returned from `parse` for the event to be sent. For information about `createEvent`, see the [Creating Events](#) section.

As the [Z-Wave Command Reference](#) shows, many Z-Wave command classes have multiple versions. By default, `zwave.parse` will parse a command using the highest version of the command class. If the device is sending an earlier version of the command, some fields may be missing, or the command may fail to parse and return `null`. To fix this, you can pass in a map as the second argument to `zwave.parse` to tell it which version of each command class to use:

```
zwave.parse(description, [0x26: 1, 0x70: 1])
```

This example will use version 1 of `SwitchMultilevel` (0x26) and `Configuration` (0x70) instead of the highest versions.

7.9.2 Sending Commands

To send a Z-Wave command to the device, you must create the command object, call `format` on it to convert it to the encoded string representation, and return it from the command method.

```

def on() {
    return zwave.basicV1.basicSet(value: 0xFF).format()
}

```

There is a shorthand provided to create command objects: `zwave.basicV1.basicSet(value: 0xFF)` is the same as `new physicalgraph.zwave.commands.basicv1.BasicSet(value: 0xFF)`. Note the different capitalization of the command name and the ‘V’ in the command class name.

The value `0xFF` passed in to the command is a hexadecimal number. Many Z-Wave commands use 8-bit integers to represent device state. Generally 0 means “off” or “inactive”, 1-99 are used as percentage values for a variable level attribute, and `0xFF` or 255 (the highest value) means “on” or “detected”.

If you want to send more than one Z-Wave command, you can return a list of formatted command strings. It is often a good idea to add a delay between commands to give the device an opportunity to finish processing each command and possibly send a response before receiving the next command. To add a delay between commands, include a string

of the form "delay N" where N is the number of milliseconds to delay. There is a helper method `delayBetween` that will take a list of commands and insert delay commands between them:

```
def off() {
    delayBetween([
        zwave.basicV1.basicSet(value: 0).format(),
        zwave.switchBinaryV1.switchBinaryGet().format()
    ], 100)
}
```

This example returns the output of `delayBetween`, and thus will send a `BasicSet` command, followed by a 100 ms delay (0.1 seconds), then a `SwitchBinaryGet` command in order to check immediately that the state of the switch was indeed changed by the `set` command.

7.9.3 Sending commands in response to events

In some situations, instead of sending a command in response to a request by the user, you want to automatically send a command to the device on receipt of a Z-Wave command.

If you return a list from the parse method, each item of the list will be evaluated separately. Items that are maps will be processed as events as usual and sent to subscribed SmartApps and mobile clients. Returned items that are `HubAction` items, however, will be sent via the hub to the device, in much the same way as formatted commands returned from command methods. The easiest way to send a command to a device in response to an event is the `response` helper, which takes a Z-Wave command or encoded string and supplies a `HubAction`:

```
def zwaveEvent(physicalgraph.zwave.commands.wakeupv1.WakeUpNotification cmd)
{
    sendEvent(descriptionText: "${device.displayName} woke up", displayed: false)
    def result = []
    result << zwave.batteryV1.batteryGet().format()
    result << "delay 1200"
    result << zwave.wakeupV1.wakeUpNoMoreInformation().format()
    response(result) // returns the result of reponse()
}
```

The above example uses the `response` helper to send Z-Wave commands and delay commands to the device whenever a `WakeUpNotification` event is received. The reception of this event that indicates that the sleepy device is temporarily listening for commands. In addition to creating a hidden event, the handler will send a `BatteryGet` request, wait 1.2 seconds for a response, and then issue a `WakeUpNoMoreInformation` command to tell the device it can go back to sleep to save battery.

7.10 Z-Wave Example

Below is a device handler code sample with examples of many common commands and parsed events.

You can also view this example in [GitHub here](#).

```
metadata {
    definition (name: "Z-Wave Device Reference", author: "SmartThings") {
        capability "Actuator"
        capability "Switch"
        capability "Polling"
        capability "Refresh"
        capability "Temperature Measurement"
        capability "Sensor"
        capability "Battery"
    }
}
```



```

    }

    simulator {
        // These show up in the IDE simulator "messages" drop-down to test
        // sending event messages to your device handler
        status "basic report on":
            zwave.basicV1.basicReport (value:0xFF).incomingMessage()
        status "basic report off":
            zwave.basicV1.basicReport (value:0).incomingMessage()
        status "dimmer switch on at 70%":
            zwave.switchMultilevelV1.switchMultilevelReport (value:70).incomingMessage()
        status "basic set on":
            zwave.basicV1.basicSet (value:0xFF).incomingMessage()
        status "temperature report 70°F":
            zwave.sensorMultilevelV2.sensorMultilevelReport (scaledSensorValue: 70)
        status "low battery alert":
            zwave.batteryV1.batteryReport (batteryLevel:0xFF).incomingMessage()
        status "multichannel sensor":
            zwave.multiChannelV3.multiChannelCmdEncap (sourceEndPoint:1, destinationEndPoint:1)

        // simulate turn on
        reply "2001FF,delay 5000,2002": "command: 2503, payload: FF"

        // simulate turn off
        reply "200100,delay 5000,2002": "command: 2503, payload: 00"
    }

    tiles {
        standardTile("switch", "device.switch", width: 2, height: 2,
            canChangeIcon: true) {
            state "on", label: '${name}', action: "switch.off",
                icon: "st.unknown.zwave.device", backgroundColor: "#79b821"
            state "off", label: '${name}', action: "switch.on",
                icon: "st.unknown.zwave.device", backgroundColor: "#ffffff"
        }
        standardTile("refresh", "command.refresh", inactiveLabel: false,
            decoration: "flat") {
            state "default", label: '', action: "refresh.refresh",
                icon: "st.secondary.refresh"
        }

        valueTile("battery", "device.battery", inactiveLabel: false,
            decoration: "flat") {
            state "battery", label: '${currentValue}% battery', unit: ""
        }

        valueTile("temperature", "device.temperature") {
            state("temperature", label: '${currentValue}°',
                backgroundColors: [
                    [value: 31, color: "#153591"],
                    [value: 44, color: "#1e9cbb"],
                    [value: 59, color: "#90d2a7"],
                    [value: 74, color: "#44b621"],
                    [value: 84, color: "#f1d801"],
                    [value: 95, color: "#d04e00"],
                    [value: 96, color: "#bc2323"]
                ])
        }
    }

```

```

    }

    main(["switch", "temperature"])
    details(["switch", "temperature", "refresh", "battery"])
  }
}

def parse(String description) {
  def result = null
  def cmd = zwave.parse(description, [0x60: 3])
  if (cmd) {
    result = zwaveEvent(cmd)
    log.debug "Parsed ${cmd} to ${result.inspect()}"
  } else {
    log.debug "Non-parsed event: ${description}"
  }
  result
}

def zwaveEvent(physicalgraph.zwave.commands.basicv1.BasicReport cmd)
{
  def result = []
  result << createEvent(name:"switch", value: cmd.value ? "on" : "off")

  // For a multilevel switch, cmd.value can be from 1-99 to represent
  // dimming levels
  result << createEvent(name:"level", value: cmd.value, unit:"%",
    descriptionText:"${device.displayName} dimmed ${cmd.value==255 ? 100 : cmd.value}")

  result
}

def zwaveEvent(physicalgraph.zwave.commands.switchbinaryv1.SwitchBinaryReport cmd) {
  createEvent(name:"switch", value: cmd.value ? "on" : "off")
}

def zwaveEvent(physicalgraph.zwave.commands.switchmultilevelv3.SwitchMultilevelReport cmd) {
  def result = []
  result << createEvent(name:"switch", value: cmd.value ? "on" : "off")
  result << createEvent(name:"level", value: cmd.value, unit:"%",
    descriptionText:"${device.displayName} dimmed ${cmd.value==255 ? 100 : cmd.value}")

  result
}

def zwaveEvent(physicalgraph.zwave.commands.meterv1.MeterReport cmd) {
  def result
  if (cmd.scale == 0) {
    result = createEvent(name: "energy", value: cmd.scaledMeterValue,
      unit: "kWh")
  } else if (cmd.scale == 1) {
    result = createEvent(name: "energy", value: cmd.scaledMeterValue,
      unit: "kVAh")
  } else {
    result = createEvent(name: "power",
      value: Math.round(cmd.scaledMeterValue), unit: "W")
  }

  result
}

```

```

}

def zwaveEvent(physicalgraph.zwave.commands.meterv3.MeterReport cmd) {
    def map = null
    if (cmd.meterType == 1) {
        if (cmd.scale == 0) {
            map = [name: "energy", value: cmd.scaledMeterValue,
                  unit: "kWh"]
        } else if (cmd.scale == 1) {
            map = [name: "energy", value: cmd.scaledMeterValue,
                  unit: "kVAh"]
        } else if (cmd.scale == 2) {
            map = [name: "power", value: cmd.scaledMeterValue, unit: "W"]
        } else {
            map = [name: "electric", value: cmd.scaledMeterValue]
            map.unit = ["pulses", "V", "A", "R/Z", ""][cmd.scale - 3]
        }
    } else if (cmd.meterType == 2) {
        map = [name: "gas", value: cmd.scaledMeterValue]
        map.unit = ["m^3", "ft^3", "", "pulses", ""][cmd.scale]
    } else if (cmd.meterType == 3) {
        map = [name: "water", value: cmd.scaledMeterValue]
        map.unit = ["m^3", "ft^3", "gal"][cmd.scale]
    }
    if (map) {
        if (cmd.previousMeterValue && cmd.previousMeterValue != cmd.meterValue) {
            map.descriptionText = "${device.displayName} ${map.name} is ${map.value} ${map.unit}"
        }
        createEvent(map)
    } else {
        null
    }
}

def zwaveEvent(physicalgraph.zwave.commands.sensorbinaryv2.SensorBinaryReport cmd) {
    def result
    switch (cmd.sensorType) {
        case 2:
            result = createEvent(name: "smoke",
                                value: cmd.sensorValue ? "detected" : "closed")
            break
        case 3:
            result = createEvent(name: "carbonMonoxide",
                                value: cmd.sensorValue ? "detected" : "clear")
            break
        case 4:
            result = createEvent(name: "carbonDioxide",
                                value: cmd.sensorValue ? "detected" : "clear")
            break
        case 5:
            result = createEvent(name: "temperature",
                                value: cmd.sensorValue ? "overheated" : "normal")
            break
        case 6:
            result = createEvent(name: "water",
                                value: cmd.sensorValue ? "wet" : "dry")
            break
        case 7:
    }
}

```

```

        result = createEvent(name:"temperature",
                             value: cmd.sensorValue ? "freezing" : "normal")
        break
    case 8:
        result = createEvent(name:"tamper",
                             value: cmd.sensorValue ? "detected" : "okay")
        break
    case 9:
        result = createEvent(name:"aux",
                             value: cmd.sensorValue ? "active" : "inactive")
        break
    case 0x0A:
        result = createEvent(name:"contact",
                             value: cmd.sensorValue ? "open" : "closed")
        break
    case 0x0B:
        result = createEvent(name:"tilt", value: cmd.sensorValue ? "detected" : "okay")
        break
    case 0x0C:
        result = createEvent(name:"motion",
                             value: cmd.sensorValue ? "active" : "inactive")
        break
    case 0x0D:
        result = createEvent(name:"glassBreak",
                             value: cmd.sensorValue ? "detected" : "okay")
        break
    default:
        result = createEvent(name:"sensor",
                             value: cmd.sensorValue ? "active" : "inactive")
        break
    }
    result
}

def zwaveEvent(physicalgraph.zwave.commands.sensorbinaryv1.SensorBinaryReport cmd)
{
    // Version 1 of SensorBinary doesn't have a sensor type
    createEvent(name:"sensor", value: cmd.sensorValue ? "active" : "inactive")
}

def zwaveEvent(physicalgraph.zwave.commands.sensormultilevelv5.SensorMultilevelReport cmd)
{
    def map = [ displayed: true, value: cmd.scaledSensorValue.toString() ]
    switch (cmd.sensorType) {
        case 1:
            map.name = "temperature"
            map.unit = cmd.scale == 1 ? "F" : "C"
            break;
        case 2:
            map.name = "value"
            map.unit = cmd.scale == 1 ? "%" : ""
            break;
        case 3:
            map.name = "illuminance"
            map.value = cmd.scaledSensorValue.toInteger().toString()
            map.unit = "lux"
            break;
        case 4:

```

```

        map.name = "power"
        map.unit = cmd.scale == 1 ? "Btu/h" : "W"
        break;
    case 5:
        map.name = "humidity"
        map.value = cmd.scaledSensorValue.toInteger().toString()
        map.unit = cmd.scale == 0 ? "%" : ""
        break;
    case 6:
        map.name = "velocity"
        map.unit = cmd.scale == 1 ? "mph" : "m/s"
        break;
    case 8:
    case 9:
        map.name = "pressure"
        map.unit = cmd.scale == 1 ? "inHg" : "kPa"
        break;
    case 0xE:
        map.name = "weight"
        map.unit = cmd.scale == 1 ? "lbs" : "kg"
        break;
    case 0xF:
        map.name = "voltage"
        map.unit = cmd.scale == 1 ? "mV" : "V"
        break;
    case 0x10:
        map.name = "current"
        map.unit = cmd.scale == 1 ? "mA" : "A"
        break;
    case 0x12:
        map.name = "air flow"
        map.unit = cmd.scale == 1 ? "cfm" : "m^3/h"
        break;
    case 0x1E:
        map.name = "loudness"
        map.unit = cmd.scale == 1 ? "dBA" : "dB"
        break;
    }
    createEvent(map)
}

// Many sensors send BasicSet commands to associated devices.
// This is so you can associate them with a switch-type device
// and they can directly turn it on/off when the sensor is triggered.
def zwaveEvent(physicalgraph.zwave.commands.basicv1.BasicSet cmd)
{
    createEvent(name:"sensor", value: cmd.value ? "active" : "inactive")
}

def zwaveEvent(physicalgraph.zwave.commands.batteryv1.BatteryReport cmd) {
    def map = [ name: "battery", unit: "%" ]
    if (cmd.batteryLevel == 0xFF) { // Special value for low battery alert
        map.value = 1
        map.descriptionText = "${device.displayName} has a low battery"
        map.isStateChange = true
    } else {
        map.value = cmd.batteryLevel
    }
}

```

```

    // Store time of last battery update so we don't ask every wakeup, see WakeUpNotification handler
    state.lastbatt = new Date().time
    createEvent(map)
}

// Battery powered devices can be configured to periodically wake up and
// check in. They send this command and stay awake long enough to receive
// commands, or until they get a WakeUpNoMoreInformation command that
// instructs them that there are no more commands to receive and they can
// stop listening.
def zwaveEvent(physicalgraph.zwave.commands.wakeupv2.WakeUpNotification cmd)
{
    def result = [createEvent(descriptionText: "${device.displayName} woke up", isStateChange: false)]

    // Only ask for battery if we haven't had a BatteryReport in a while
    if (!state.lastbatt || (new Date().time) - state.lastbatt > 24*60*60*1000) {
        result << response(zwave.batteryV1.batteryGet())
        result << response("delay 1200") // leave time for device to respond to batteryGet
    }
    result << response(zwave.wakeUpV1.wakeUpNoMoreInformation())
    result
}

def zwaveEvent(physicalgraph.zwave.commands.associationv2.AssociationReport cmd) {
    def result = []
    if (cmd.nodeId.any { it == zwaveHubNodeId }) {
        result << createEvent(descriptionText: "$device.displayName is associated in group $cmd.groupId")
    } else if (cmd.groupingIdentifier == 1) {
        // We're not associated properly to group 1, set association
        result << createEvent(descriptionText: "Associating $device.displayName in group $cmd.groupId")
        result << response(zwave.associationV1.associationSet(groupingIdentifier: cmd.groupingIdentifier))
    }
    result
}

// Devices that support the Security command class can send messages in an
// encrypted form; they arrive wrapped in a SecurityMessageEncapsulation
// command and must be unencapsulated
def zwaveEvent(physicalgraph.zwave.commands.securityv1.SecurityMessageEncapsulation cmd) {
    def encapsulatedCommand = cmd.encapsulatedCommand([0x98: 1, 0x20: 1])

    // can specify command class versions here like in zwave.parse
    if (encapsulatedCommand) {
        return zwaveEvent(encapsulatedCommand)
    }
}

// MultiChannelCmdEncap and MultiInstanceCmdEncap are ways that devices
// can indicate that a message is coming from one of multiple subdevices
// or "endpoints" that would otherwise be indistinguishable
def zwaveEvent(physicalgraph.zwave.commands.multichannelv3.MultiChannelCmdEncap cmd) {
    def encapsulatedCommand = cmd.encapsulatedCommand([0x30: 1, 0x31: 1])

    // can specify command class versions here like in zwave.parse
    log.debug("Command from endpoint ${cmd.sourceEndPoint}: ${encapsulatedCommand}")

    if (encapsulatedCommand) {
        return zwaveEvent(encapsulatedCommand)
    }
}

```

```

    }
}

def zwaveEvent(physicalgraph.zwave.commands.multichannelv3.MultiInstanceCmdEncap cmd) {
    def encapsulatedCommand = cmd.encapsulatedCommand([0x30: 1, 0x31: 1])

    // can specify command class versions here like in zwave.parse
    log.debug ("Command from instance ${cmd.instance}: ${encapsulatedCommand}")

    if (encapsulatedCommand) {
        return zwaveEvent(encapsulatedCommand)
    }
}

def zwaveEvent(physicalgraph.zwave.Command cmd) {
    createEvent(descriptionText: "${device.displayName}: ${cmd}")
}

def on() {
    delayBetween([
        zwave.basicV1.basicSet(value: 0xFF).format(),
        zwave.basicV1.basicGet().format()
    ], 5000) // 5 second delay for dimmers that change gradually, can be left out for immediate
}

def off() {
    delayBetween([
        zwave.basicV1.basicSet(value: 0x00).format(),
        zwave.basicV1.basicGet().format()
    ], 5000) // 5 second delay for dimmers that change gradually, can be left out for immediate
}

def refresh() {
    // Some examples of Get commands
    delayBetween([
        zwave.switchBinaryV1.switchBinaryGet().format(),
        zwave.switchMultilevelV1.switchMultilevelGet().format(),
        zwave.meterV2.meterGet(scale: 0).format(), // get kWh
        zwave.meterV2.meterGet(scale: 2).format(), // get Watts
        zwave.sensorMultilevelV1.sensorMultilevelGet().format(),
        zwave.sensorMultilevelV5.sensorMultilevelGet(sensorType:1, scale:1).format(), // get
        zwave.batteryV1.batteryGet().format(),
        zwave.basicV1.basicGet().format(),
    ], 1200)
}

// If you add the Polling capability to your device type, this command
// will be called approximately every 5 minutes to check the device's state
def poll() {
    zwave.basicV1.basicGet().format()
}

// If you add the Configuration capability to your device type, this
// command will be called right after the device joins to set
// device-specific configuration commands.
def configure() {
    delayBetween([
        // Note that configurationSet.size is 1, 2, or 4 and generally

```

```
// must match the size the device uses in its configurationReport
zwave.configurationV1.configurationSet(parameterNumber:1, size:2, scaledConfigurationValue:1)

// Can use the zwaveHubNodeId variable to add the hub to the
// device's associations:
zwave.associationV1.associationSet(groupingIdentifier:2, nodeId:zwaveHubNodeId).format()

// Make sure sleepy battery-powered sensors send their
// WakeUpNotifications to the hub every 4 hours:
zwave.wakeUpV1.wakeUpIntervalSet(seconds:4 * 3600, nodeId:zwaveHubNodeId).format(),

    ])
}
```

7.11 ZigBee Primer

Before we start, let's take a look at a full ZigBee message as it would look in a SmartThings Device Type Handler. Then we'll break up the message into its parts and dive into what each part means. Make sure you download the ZigBee Cluster Library as a reference for ZigBee message formatting and what is possible for each device.

Here is a full command: "st cmd 0x\${device.deviceNetworkId} \${endpointId} 8 4 {FFFF0000}"

The 3 Main types of ZigBee Messages

- st cmd - SmartThings Command which formats the message as a ZigBee Command
- st rattr - SmartThings Read Attribute which formats the message as a ZigBee Read Attribute
- st wattr - SmartThings Write Attribute which as you guessed formats the message as a ZigBee Write Attribute

7.11.1 Device Network ID

All connected devices have a Device Network ID that is used to route messages correctly to the device. In the loosest terms think of the Network ID as the IP Address. It is a 4 digit hex number that the device gets while pairing. Since the Network ID is different by device, you can reference it dynamically in a Device Handler like this: 0x\${device.deviceNetworkId}

7.11.2 Endpoints

Endpoints are simple. Think of them basically as ports. Different endpoints can support different clusters and a device can have multiple endpoints to do different things. Endpoints can be used to separate functionality when needed. For example a temperature sensor can have the Temperature Measurement Cluster on endpoint 1 and have Over The Air Boot loader Cluster on endpoint 2.

7.11.3 Clusters

Clusters are a group of commands and attributes that define what a device can do. Think of clusters as a group of actions by function. A device can support multiple clusters to do a whole variety of tasks. Majority of clusters are defined by the ZigBee Alliance and listed in the ZigBee Cluster Library. There are also profile specific clusters that are defined by their own ZigBee profile like Home Automation or ZigBee Smart Energy, and Manufacture Specific clusters that are defined by the manufacture of the device. These are typically used when no existing cluster can be used for a device.

Most used clusters are

- 0x0006 - On/Off (Switch)
- 0x0008 - Level Control (Dimmer)
- 0x0201 - Thermostat
- 0x0202 - Fan Control
- 0x0402 - Temperature Measurement
- 0x0406 - Occupancy Sensing

7.11.4 Commands

Commands are basically actions a device can take. It's how we get things to do stuff. We start a command with "st cmd". Commands and whats available are defined by the cluster.

Keeping on the On/Off cluster as an example, the available commands are:

- 0x00 - Off
- 0x01 - On
- 0x02 - Toggle

In a SmartThings Device Type the following line would turn a switch off (look at the last number): "st cmd 0x\${device.deviceNetworkId} \${endpointId} 6 0 {}"

This would turn it on: "st cmd 0x\${device.deviceNetworkId} \${endpointId} 6 1 {}"

This would toggle it: "st cmd 0x\${device.deviceNetworkId} \${endpointId} 6 2 {}"

7.11.5 Read and Write Attributes

Attributes are used to get information from a device and to set preferences on a device. The two main types are Read and Write. The data type and values are specified by cluster.

An example of a Read Attribute that would read the current level of a dimmer and return the value:

```
"st rattr cmd 0x${device.deviceNetworkId} ${endpointId} 8 0 {}"
```

Write Attributes are used to set specific preferences. Write attributes can need specific data type that the payload is in. In this example the 0x21 in the message means Unsigned 16-bit integer.

An example of a Write Attribute that would set the transition time from on to off of a dimmer look like this:

```
"st wattr 0x${device.deviceNetworkId} 1 8 0x10 0x21 {0014}"
```

In this case the payload ({0014}) translates to 2 seconds. Breaking the payload down we see that the hex value of 0x0014 equals the decimal value of 20. $20 * 1/10$ of a second equals 2 seconds.

7.12 Building ZigBee Device Handlers

There are four common ZigBee commands that you will use to integrate SmartThings with your ZigBee Devices.

7.12.1 Read

Read gets the devices current state and is formatted like this:

```
def refresh() {  
    "st rattr 0x${device.deviceNetworkId} 1 0xB04 0x50B"  
}
```

In this example, the device type (from the “CentraLite Switch” device type) is calling the “refresh” function. It is sending a ZigBee Read Attribute request to read the current state (the active power draw). The cluster we are reading here is Electrical Measurement (0xB04) and specifically the Active Power Attribute (0x50B).

Component	Description
st rattr	SmartThings Read Attribute
0x\${device.deviceNetworkId}	Device Network ID
1	Endpoint Id
0xB04	Cluster
0x50B	Attribute

7.12.2 Write

Write sets an attribute of a ZigBee device and is formatted like this:

```
def configure() {  
    "st wattr 0x${device.deviceNetworkId} 1 8 0x10 0x21 {0014}"  
}
```

In this example (from the “ZigBee Dimmer” device type) we are writing to an attribute to set the amount of time it takes for a light to fully dim on and off. Here we are using the Level Control Cluster (8) to write to the attribute that defines on and off transition time (0x10). The value we are using is formatted in an Unsigned 16-bit integer (0x21) with the payload being in 1/10th of a second. In this case the payload ({0014}) translates to 2 seconds. Breaking the payload down we see that the hex value of 0x0014 equals the decimal value of 20. $20 * 1/10$ of a second equals 2 seconds.

Note: The payload in the example above, {0014}, is a hex string. The length of the payload must be two times the length of the data type. For example, if the datatype is 16-bit, then the payload should be 4 hex digits.

Component	Description
st wattr	SmartThings Write Attribute
0x\${device.deviceNetworkId}	Device Network ID
1	Endpoint Id
8	Cluster
0x10	Attribute Set
0x21	Data Type
{0014}	Payload

7.12.3 Command

Command invokes a command on a ZigBee device and is formatted like this:

```
def on() {  
    "st cmd 0x${device.deviceNetworkId} 1 6 1 {}"  
}
```

In this example (from the “ZigBee Dimmer” device type) we are sending a ZigBee Command to turn the device on. We use the On/Off Cluster (6) and send the command to turn on (1). This commands has no payload, so there is nothing within the payload brackets. Even though there is no payload, the empty brackets are still required.

Component	Description
st cmd	SmartThings Command
0x\${device.deviceNetworkId}	Device Network ID
1	Endpoint Id
6	Cluster
1	Command
{}	Payload

7.12.4 Zdo Bind

Bind instructs a device to notify us when an attribute changes and is formatted like this:

```
def configure() {
    "zdo bind 0x${device.deviceNetworkId} 1 1 6 ${device.zigbeeId} {}"
```

In this example (using the “Centralite Switch” device type), the bind command is sent to the device using its Network ID which can be determined using 0x\${device.deviceNetworkId}. Then using source and destination endpoints for the device and hub (1 1), we bind to the On/Off Clusters (6) to get events from the device. The last part of the message contains the hub’s ZigBee id which is set as the location for the device to send callback messages to. Note that not all devices support binding for events.

Component	Description
zdo bind	SmartThings Command
0x\${device.deviceNetworkId}	Device Network ID
1	Source Endpoint
1	Destination Endpoint
0x0006	Cluster
\${device.zigbeeId} {}	ZigBee ID (“IEEE Id”)

7.12.5 ZigBee Utilities

In order to work with ZigBee you will need to use the ZigBee Cluster Library extensively to look up the proper values to send back and forth to your device. You can download this document [here](#).

7.13 ZigBee Example

An example of a ZigBee device-type is the SmartPower Outlet. You can find the code in the IDE in the “Browse Device Types” menu.

You can also find the source code in GitHub [here](#), and below.

```
metadata {
    // Automatically generated. Make future change here.
    definition (name: "SmartPower Outlet", namespace: "smarththings", author: "SmartThings") {
        capability "Actuator"
        capability "Switch"
        capability "Power Meter"
        capability "Configuration"
```

```

        capability "Refresh"
        capability "Sensor"

        fingerprint profileId: "0104", inClusters: "0000,0003,0004,0005,0006,0B04,0B05", outClusters: "0000,0003,0004,0005,0006,0B04,0B05"
    }

    // simulator metadata
    simulator {
        // status messages
        status "on": "on/off: 1"
        status "off": "on/off: 0"

        // reply messages
        reply "zcl on-off on": "on/off: 1"
        reply "zcl on-off off": "on/off: 0"
    }

    // UI tile definitions
    tiles {
        standardTile("switch", "device.switch", width: 2, height: 2, canChangeIcon: true) {
            state "off", label: '${name}', action: "switch.on", icon: "st.switches.switch_off"
            state "on", label: '${name}', action: "switch.off", icon: "st.switches.switch_on"
        }
        valueTile("power", "device.power", decoration: "flat") {
            state "power", label: '${currentValue} W'
        }
        standardTile("refresh", "device.power", inactiveLabel: false, decoration: "flat") {
            state "default", label: '', action: "refresh.refresh", icon: "st.secondary.refresh"
        }

        main "switch"
        details(["switch", "power", "refresh"])
    }
}

// Parse incoming device messages to generate events
def parse(String description) {
    log.debug "Parse description $description"
    def name = null
    def value = null
    if (description?.startsWith("read attr -")) {
        def descMap = parseDescriptionAsMap(description)
        log.debug "Read attr: $description"
        if (descMap.cluster == "0006" && descMap.attrId == "0000") {
            name = "switch"
            value = descMap.value.endsWith("01") ? "on" : "off"
        } else {
            def reportValue = description.split(",").find {it.split(":")[0].trim() == "value"}
            name = "power"
            // assume 16 bit signed for encoding and power divisor is 10
            value = Integer.parseInt(reportValue, 16) / 10
        }
    } else if (description?.startsWith("on/off:")) {
        log.debug "Switch command"
        name = "switch"
        value = description?.endsWith(" 1") ? "on" : "off"
    }
}

```

```

        def result = createEvent(name: name, value: value)
        log.debug "Parse returned ${result?.descriptionText}"
        return result
    }

    def parseDescriptionAsMap(description) {
        (description - "read attr - ").split(",").inject([:]) { map, param ->
            def nameAndValue = param.split(":")
            map += [(nameAndValue[0].trim()):nameAndValue[1].trim()]
        }
    }

    // Commands to device
    def on() {
        'zcl on-off on'
    }

    def off() {
        'zcl on-off off'
    }

    def meter() {
        "st rattr 0x${device.deviceNetworkId} 1 0xB04 0x50B"
    }

    def refresh() {
        "st rattr 0x${device.deviceNetworkId} 1 0xB04 0x50B"
    }

    def configure() {
        [
            "zdo bind 0x${device.deviceNetworkId} 1 1 6 ${device.zigbeeId} {}", "delay 200",
            "zdo bind 0x${device.deviceNetworkId} 1 1 0xB04 ${device.zigbeeId} {}"
        ]
    }

```

7.14 Submitting Device Types for Publication

To submit your Device Types for consideration for publication to the SmartThings Platform, you can create a Publication Request by clicking on the [My Publication Requests](#) tab in the [SmartThings IDE](#), then clicking on the *New Request* button in the upper-right-hand corner of the screen.

7.14.1 Review Guidelines

Once submitted, your Device Type will undergo a review and approval process. For the greatest likelihood of success, follow these guidelines:

General

- Make sure your Device Type compiles and runs, which means it works in the IDE and mobile devices.
- Do not use offensive, profane, or libelous language.
- No advertising or sponsorships.

- Every class and nontrivial public method you write should contain a comment with at least one sentence describing what it does. This sentence should start with a 3rd person descriptive verb.
- Use the *password* input type whenever you are asking a user for a password.
- Split Device Type and SmartApp functionality into two different submissions.
- Do not aggressively loop or schedule.

Web Services

- If your Device Type sends any data from the SmartThings Platform to an external service, include in the description exactly what data is sent to the remote service, how that data will be used, and include a link to the privacy policy of the remote service
- If your Device Type exposes any Web Service APIs, describe what the APIs will be used for, what data may be accessed by those APIs, and where possible, include a link to the privacy policies of any remote services that may access those APIs.

Style

- Use meaningful variable and method names.
- Maintain consistent formatting and indentation. We can't review code that we can't easily read.

Reasons for Rejection

- The device type adds minor addition or change that may be changed with a core product or UX change in a future update.
- SmartThings is already developing a first-party integration and will not accept a device type for this device.
- The device type should actually be a SmartApp instead, because it's actuating or changing a device.
- Suggested change does not fit our philosophy.
- No discovery mechanism is provided. For LAN-Connected devices, a [Service Manager SmartApp](#) should serve to discover and create the device.
- Multiple community submissions exist and we're rolling several improvements together, so this specific one is being rejected.

Cloud and LAN-Connected Devices

Cloud and LAN connected devices are devices that use either a 3rd party service, like the Ecobee thermostat, or communicate over the LAN (local area network) like the Sonos system. These devices require a unique implementation of their device handlers. Cloud and LAN connected devices use a service manager SmartApp along with a device handler for authentication, maintaining connections, and device communications. This guide will walk you through service manager and device handler creation for both of these scenarios.

Table of Contents:

8.1 Service Manager Design Pattern

8.1.1 Basic Overview

Devices that connect through the internet as a whole (cloud) or LAN devices (on your local network) require a defined Service-Manager SmartApp, in addition to the usually expected Device Handler. The service manager makes the connection with the device, handling the input and output interactions, and the device handler parses messages, as usual.

8.1.2 Cloud-Connected Devices

When using a cloud-connected device, the service manager is used to discover and initiate a connection between the device and your hub, using OAuth connections to external third parties. Then the device-handler uses this connection to communicate between the hub and device.

8.1.3 LAN-Connected Devices

When using a LAN connected device, the service manager is used to discover and initiate a connection between the device and your hub, using the protocols SSDP or mDNS/DNS-SD. Then the device-handler uses UPnP/SOAP Calls or REST Calls to communicate outgoing messages between the hub and device.

8.2 Building Cloud-Connected Device Types

Cloud connected devices use a 3rd party service to accomplish device communication. An example of such a device is the Ecobee thermostat.

When developing a device handler for a cloud connected device, you must create a service manager SmartApp that will handle authenticating with the 3rd party service, communicating with the device, and reacting to any device changes that occur.

This guide overviews the concept of the service manager/device handler architecture and also gives an example of both the service manager and device handler creation.

Table of Contents:

8.2.1 Division of Labor

The cloud connected device paradigm consists of a Service Manager and Device Handlers. The purpose of this guide is to introduce you to the core concepts of cloud connected device development, and provide some examples to help you get started.

Service-Manager Responsibilities

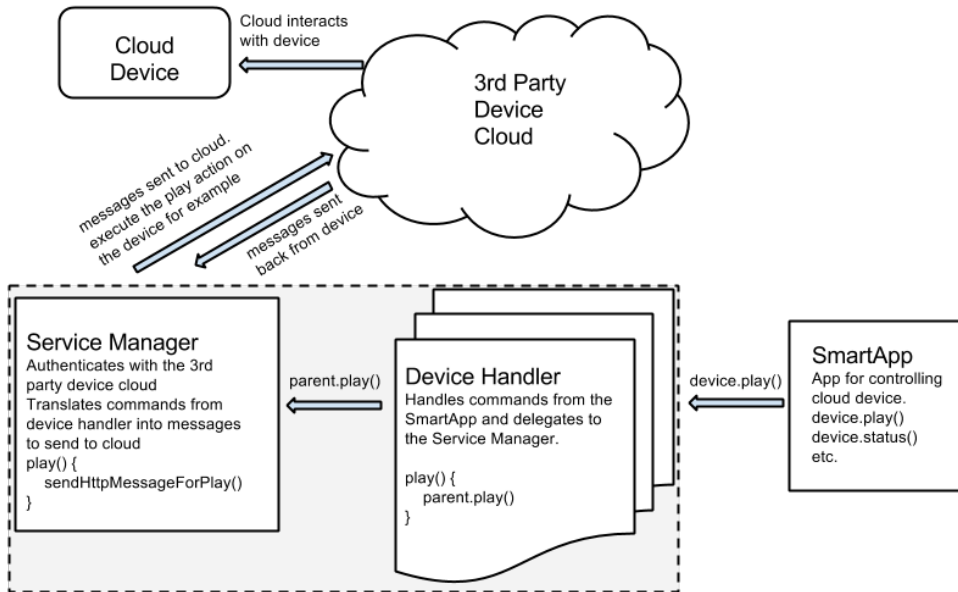
The service manager is responsible for the discovery of the devices. It sends out a request to a third party cloud and parses through the response, finding just the devices you are looking for. Upon discovery, it allows you to add device(s) that it has found. From there, it saves your connection to be able to make future interactions with the device.

Device Handler Responsibilities

The device handler is responsible for creating and receiving device specific messages, and allowing them to work within the SmartThings infrastructure. It takes in a SmartApp specific command and outputs device specific commands to be passed to the cloud. It also allows you to subscribe to responses from the device and trigger other commands as needed.

How It All Works

The following depiction gives a general overview of how a cloud connected device works. Take note of the Service Manager and Device Handler. We will dive into how to build these next.



8.2.2 Building the Service Manager

The Service Manager's responsibilities are to:

- Authenticate with the 3rd party cloud service
- Device discovery
- Add/Change/Delete device actions
- Handle sending any messages that require the authentication obtained.

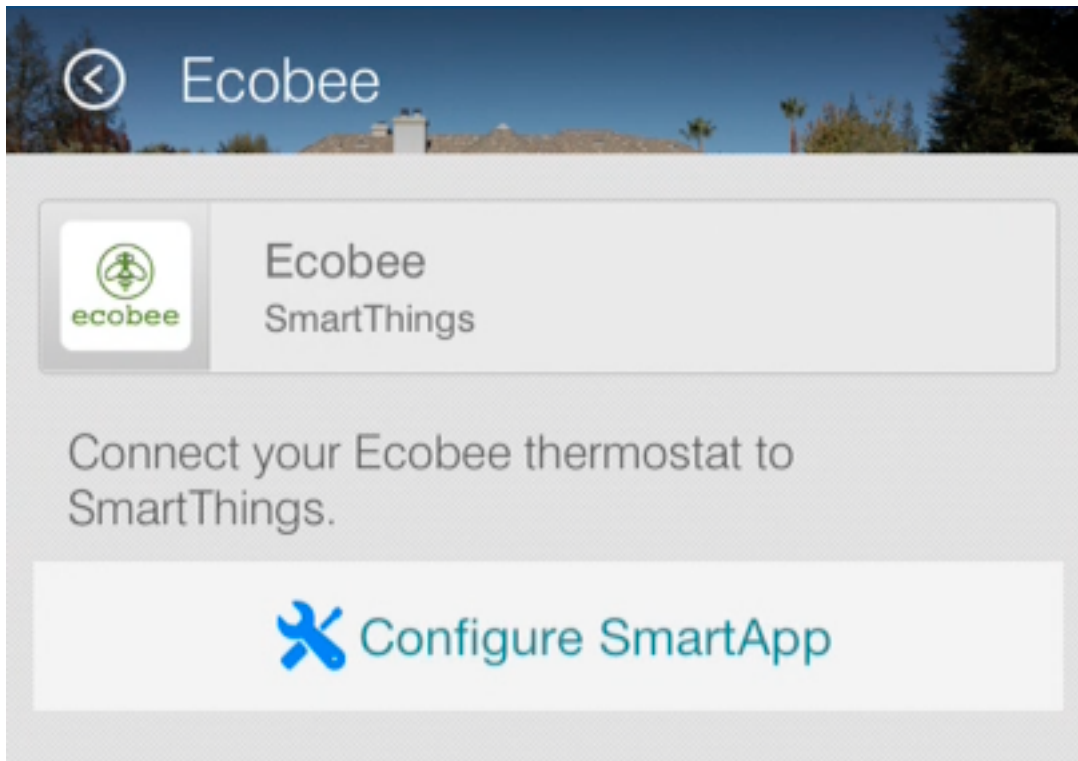
We will look at a detailed example of what is outlined above. But first, let's see an example of how what we are trying to accomplish would look like in the SmartThings application.

Authentication using OAuth

End User Experience

The experience for the end user will be fairly seamless. They will go through the following steps (illustrated using the Ecobee Thermostat)

The user selects the Service Manager application from the SmartApps within the SmartThings app. Upon selection, they are prompted with an initial landing page, describing what the application does and a link to configure.



Authorization with the third party is the first part of the configuration process. The user is driven to a page which tells them about the authorization process and how it will work. They can then click a link to move forward.

The user will be driven to a third party site, embedded within the SmartThings application chrome. They will be required to put in their username and password for the third party service.

The third party server will show what SmartThings will have access to and give the user the opportunity to accept or decline.

Upon acceptance, the user will be redirected to another page within the third party service. This page includes language about the end user clicking done on the top right of the SmartThings chrome.

After done is clicked, the user will go back to the initial configuration screen, seeing that their device is now connected. They can then click next to continue, and any other configuration can be done.

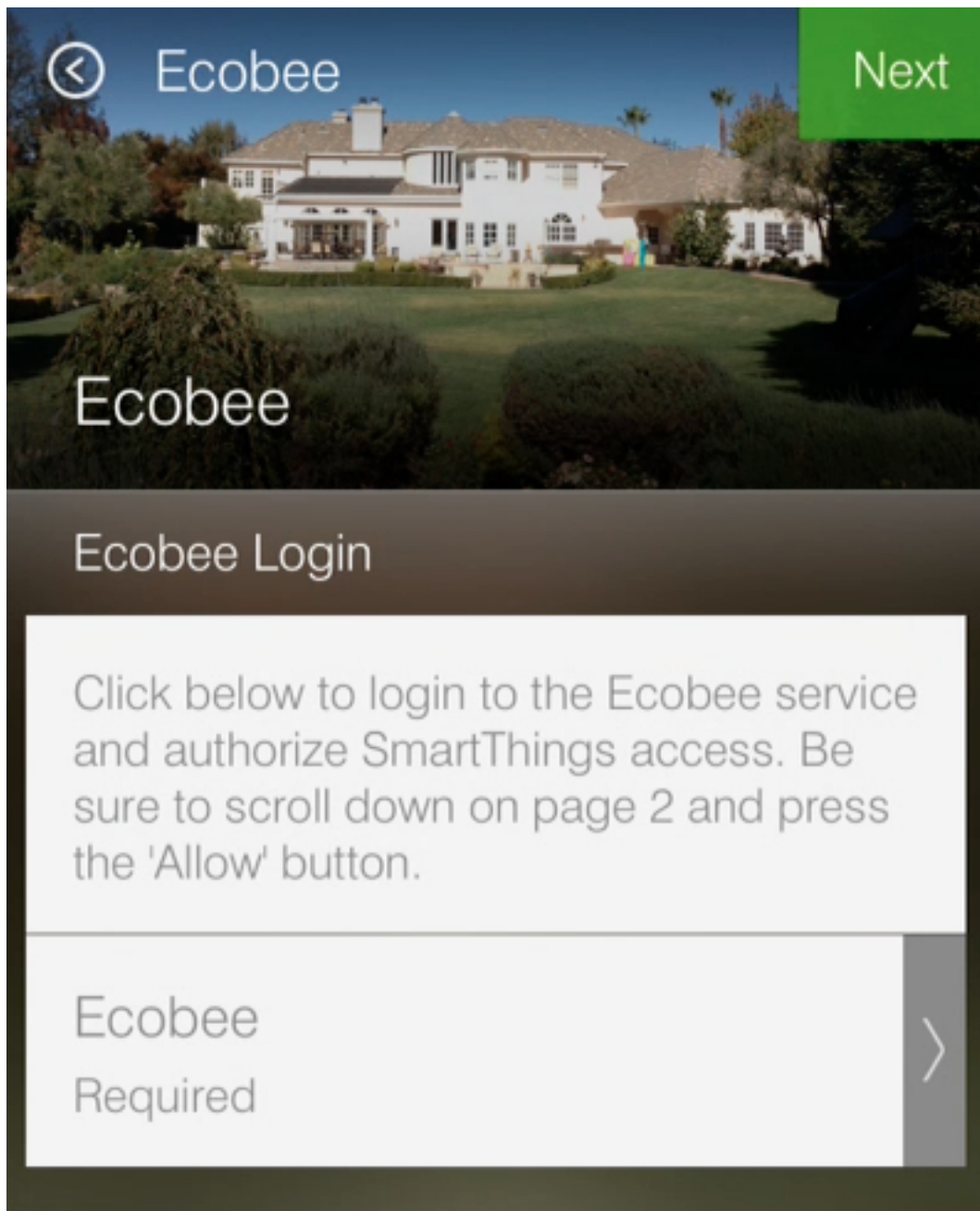
Implementation

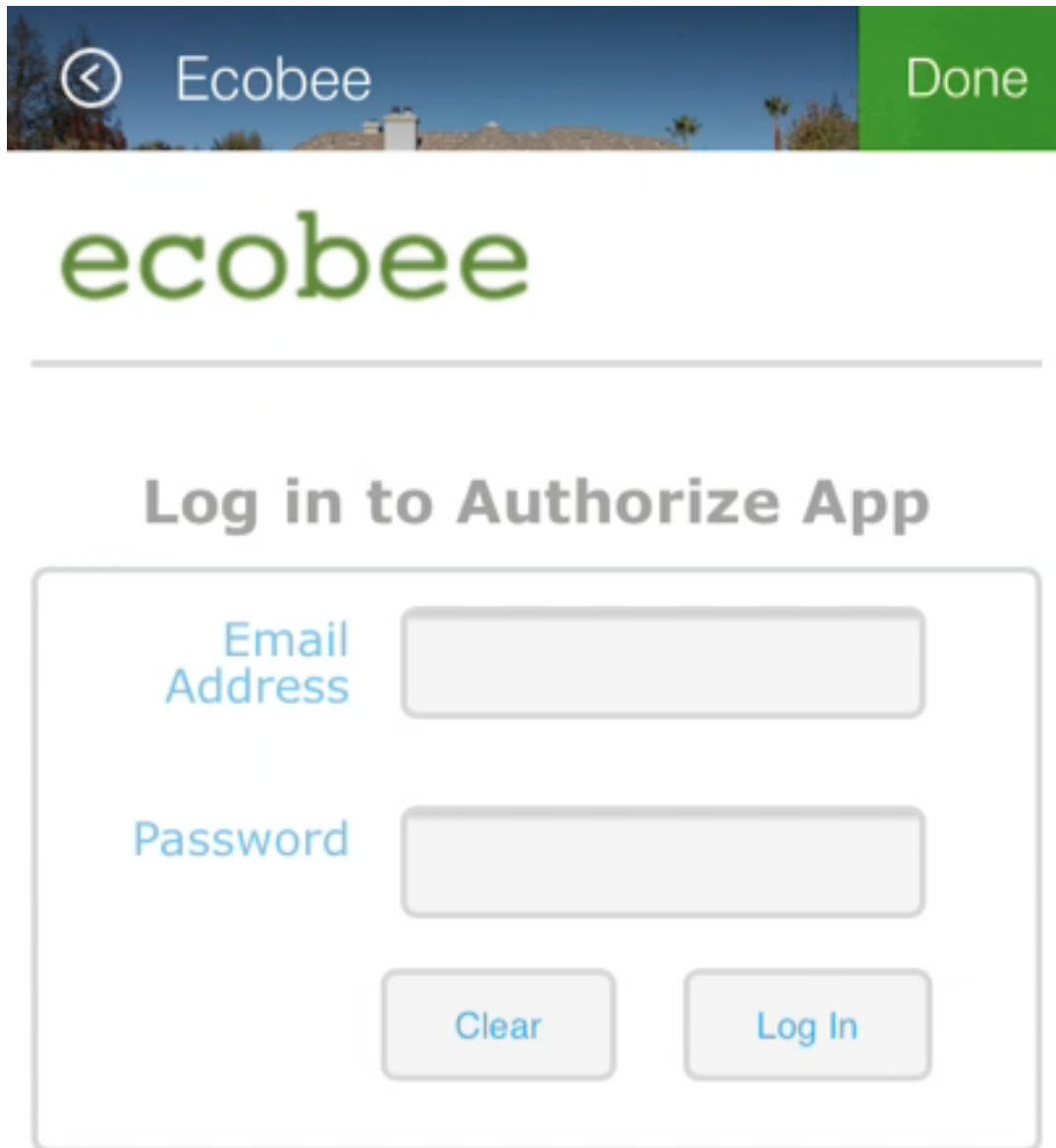
OAuth is the typical industry standard for authentication. The 3rd party service may use something other than OAuth. In that case, it is up to you to consult their documentation and implement it. The basic concepts will be the same as it is with OAuth. The following example will walk through what is necessary for OAuth authentication.

The overall idea is that you will create a page that will call out to the third party API and then map a URL to a `handlerLoads` method to be able to handle a response back with an access token.

Within your service manager preferences, you create a page for authorization.

```
preferences {  
    page(name: "Credentials", title: "Sample Authentication", content: "authPage", nextPage: "samplePage")  
    ...  
}
```





The image shows a mobile application interface for Ecobee. At the top is a header bar with a back arrow, the text 'Ecobee', and a green 'Done' button. Below the header is the 'ecobee' logo in green. A horizontal line separates the logo from the main content area. The main content area has the title 'Log in to Authorize App' in bold. Below the title is a rounded rectangle containing the login form. The form has two input fields: 'Email Address' and 'Password'. Below these fields are two buttons: 'Clear' and 'Log In'.

Ecobee Done

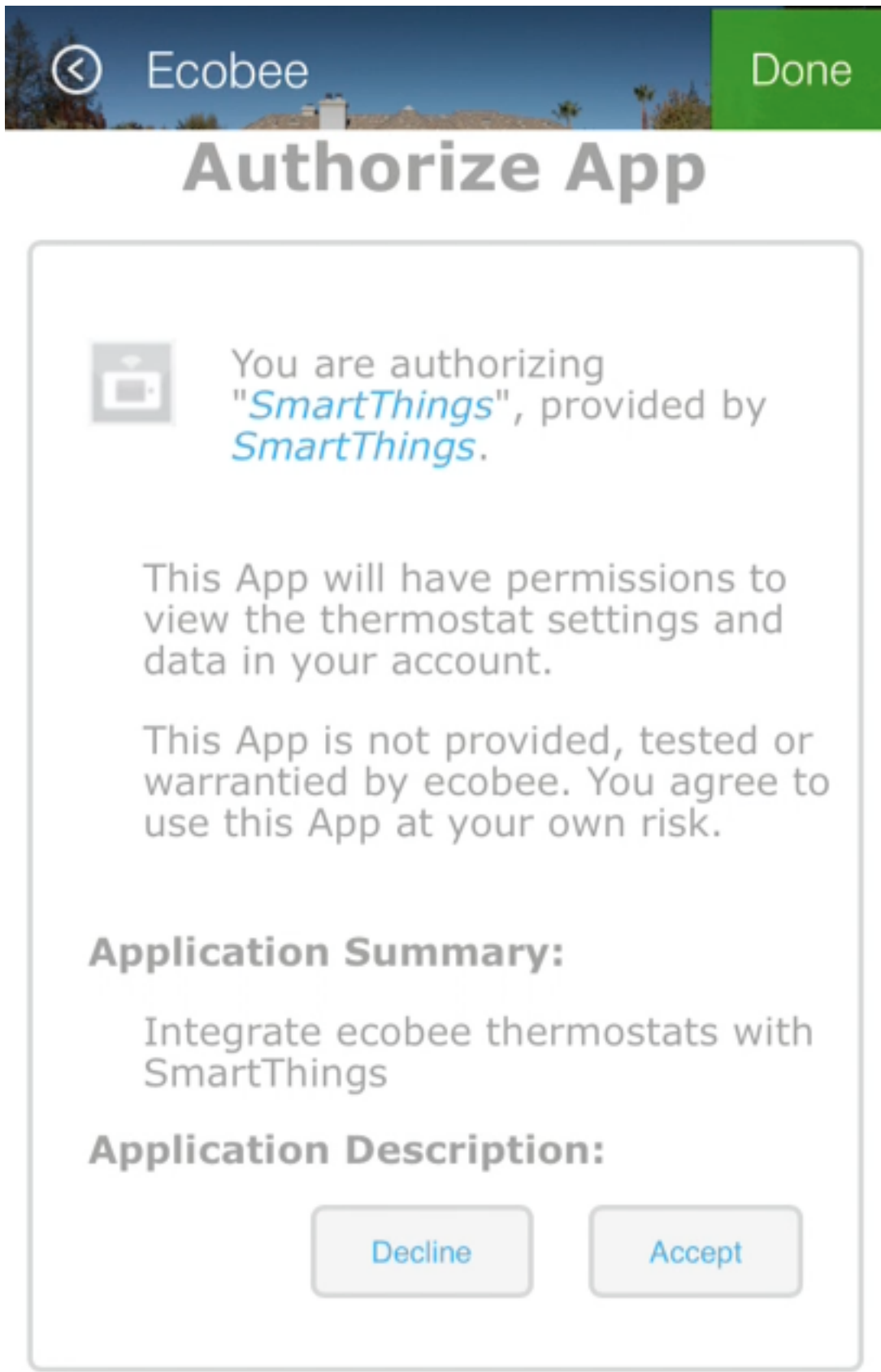
ecobee

Log in to Authorize App

Email Address

Password

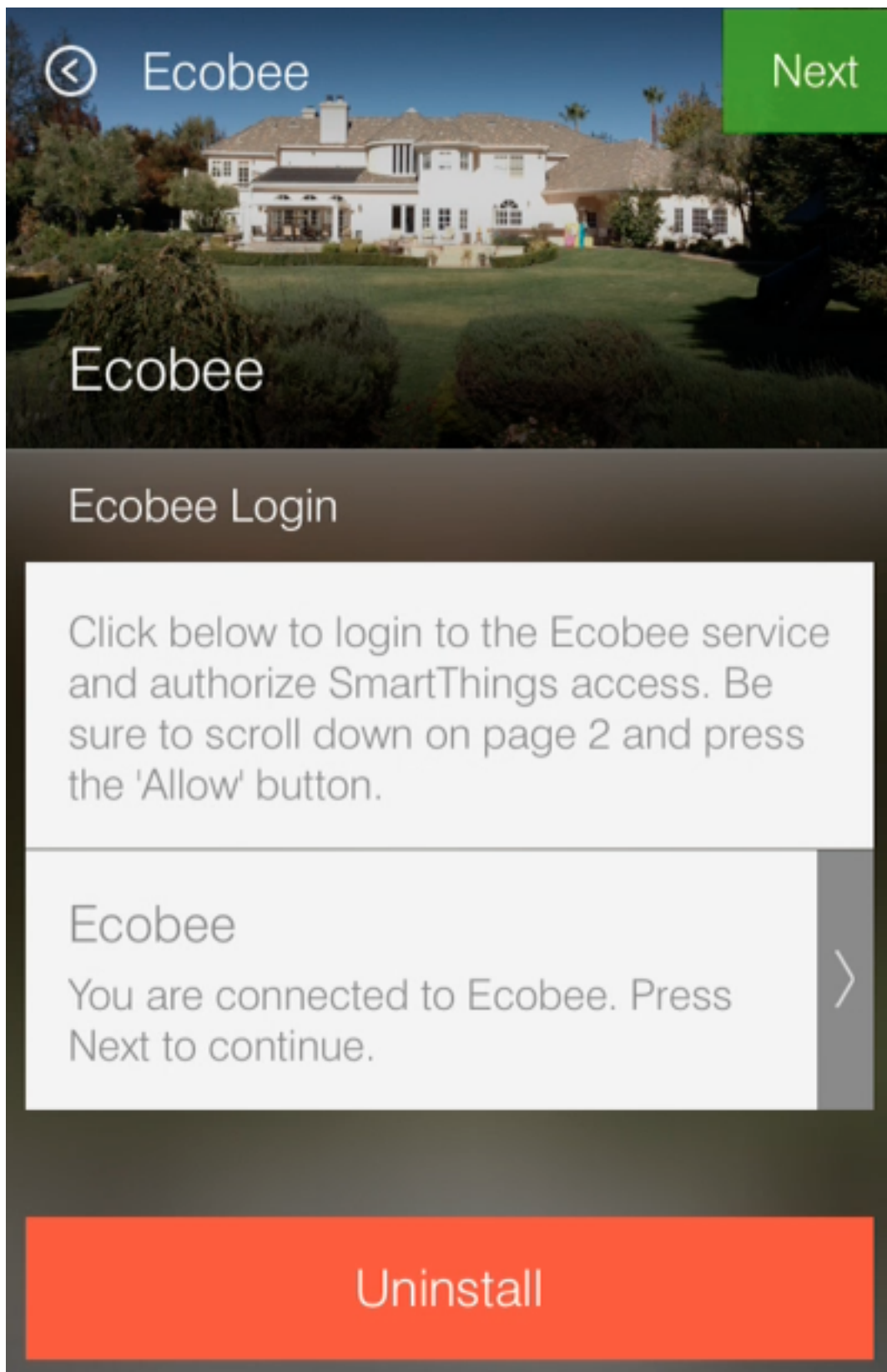
Clear Log In





Your Ecobee Account is now
connected to SmartThings!

Click 'Done' to finish setup.



and define it within a method below.

```
def authPage() {  
    if(!state.sampleAccessToken)  
        createAccessToken()  
}
```

The `authPage` method simply checks to see if there already is an access token. If not, we call a method to retrieve one. Lets take a look at the `createAccessToken` method next.

```
def createAccessToken() {  
  
    state.oauthInitState = UUID.randomUUID().toString()  
    def oauthParams = [  
        response_type: "token",  
        client_id: "XXXXXXX",  
        redirect_uri: "https://graph.api.smartthings.com/api/token/${state.accessToken}/smartapps/in  
    ]  
    def redirectUrl = "https://api.thirdpartysite.com/v1/oauth2/authorize?" + toQueryString(oauthParam  
  
    return dynamicPage(name: "Credentials", title: "Sample", nextPage: "sampleLoggedInPage", uninstal  
        section {  
            href url: redirectUrl, style: "embedded", required: false, title: "Sample", description: "Cli  
        }  
    }  
}
```

First, setup the params for your OAuth request. Then return a new page, created by the redirect URL. Finally, load up the OAuth initialization URL embedded within the app.

Once the user has authenticated through the third-party, they will be sent back to your SmartApp, and their callback needs to be handled properly.

To handle the callback, you can map a URL within your service manager. As specified, the callback will go to the following URL.

```
mappings {  
    path("/receiveToken") {  
        action: [  
            POST: "receiveToken",  
            GET: "receiveToken"  
        ]  
    }  
}
```

You also need to setup a relevant handler method that will take the `access_token` passed and save it in the state (which will persist over time). This handler should also indicate to the end user that they need to click the done button to exit the external third party flow and go back to your SmartApp.

```
def receiveToken() {  
    state.sampleAccessToken = params.access_token  
    render contentType: 'text/html', data: "<html><body>Saved. Now click 'Done' to finish setup.</body></html>"  
}
```

Refreshing the OAuth Token

OAuth tokens are available for a finite amount of time, so you will often need to account for this, and if needed, refresh your `access_token`. To do this, you need to store the `refresh_token` in your state, like so:


```
def receiveToken() {
    state.sampleAccessToken = params.access_token
    state.sampleRefreshToken = params.refresh_token
    render contentType: 'text/html', data: "<html><body>Saved. Now click 'Done' to finish setup.</body></html>"
}
```

If you run an API request and your access_token is determined invalid, for example:

```
if (resp.status == 401 && resp.data.status.code == 14) {
    log.debug "Storing the failed action to try later"
    atomicState.action = "actionCurrentlyExecuting"
    log.debug "Refreshing your auth_token!"
    refreshAuthToken()
}
```

you can use your refresh_token to get a new access_token. To do this, you just need to post to a specified endpoint and handle the response properly.

```
private refreshAuthToken() {
    def refreshParams = [
        method: 'POST',
        uri: "https://api.thirdpartysite.com",
        path: "/token",
        query: [grant_type: 'refresh_token', code: "${state.sampleRefreshToken}", client_id: XXXXXXXX],
    ]
    try {
        def jsonMap
        httpPost(refreshParams) { resp ->
            if (resp.status == 200) {
                jsonMap = resp.data
                if (resp.data) {
                    state.sampleRefreshToken = resp?.data?.refresh_token
                    state.sampleAccessToken = resp?.data?.access_token
                }
            }
        }
    }
}
```

There are some outbound connections in which we are using OAuth to connect to a third party device cloud (Ecobee, Quirky, Jawbone, etc). In these cases it is the third party device cloud that issues an OAuth token to us so that we can call their APIs.

However these same third party device clouds also support webhooks and subscriptions that allow us to receive notifications when something changes in their cloud.

In this case and ONLY in this case the SmartApp (service manager) issues its OWN OAuth token and embeds it in the callback URL as a way to authenticate the post backs from the external cloud.

Discovery

Identifying Devices in the Third-Party Device Cloud

The techniques you will use to identify devices in the third party cloud will vary, because you are interacting with unique third party APIs which all have unique parameters. Typically you will authenticate with the third party API using OAuth. Then call an API specific method. For example, it could be as simple as this:

```
def deviceListParams = [
    uri: "https://api.thirdpartysite.com",
    path: "/get-devices",
    requestContentType: "application/json",
    query: [token:"XXXX",type:"json" ]

httpGet(deviceListParams) { resp ->
    //Handle the response here
}
```

Creating Child-Devices

Within a service manager SmartApp, you create child devices for all your respective cloud devices.

```
settings.devices.each {deviceId->
    def device = state.devices.find{it.id==deviceId}
    if (device) {
        def childDevice = addChildDevice("smarththings", "Device Name", deviceId, null, name: "Device")
    }
}
```

Getting Initial Device State

Upon initial discovery of a device, you need to get the state of your device from the third party API. This would be the current status of various attributes of your device. You need to have a method defined in your Service Manager that is responsible for connecting to the API and checking for updates. You set this method to be called from a poll method in your device type, and in this case, it is called immediately on initialization. Here is a very simple example, which doesn't take into account error checking for the http request.

```
def pollParams = [
    uri: "https://api.thirdpartysite.com",
    path: "/device",
    requestContentType: "application/json",
    query: [format:"json",body: jsonRequestBody]

httpGet(pollParams) { resp ->
    state.devices = resp.data.devices { collector, stat ->
        def dni = [ app.id, stat.identifier ].join('.')
        def data = [
            attribute1: stat.attributeValue,
            attribute2: stat.attribute2Value
        ]
        collector[dni] = [data:data]
        return collector
    }
}
```

Handling Adds, Changes, Deletes

Implicit Creation of New Child Devices

When you update your settings in a Service Manager to add additional devices, the Service Manager needs to respond by adding a new device in SmartThings.

```

updated() {
    initialize()
}

initialize() {
    settings.devices.each {deviceId ->
        try {
            def existingDevice = getChildDevice(deviceId)
            if(!existingDevice) {
                def childDevice = addChildDevice("smarthings", "Device Name", deviceId, null, [name
            }
        } catch (e) {
            log.error "Error creating device: ${e}"
        }
    }
}

```

Implicit Removal of Child Devices

Similarly when you remove devices within your Service Manager, they need to be removed from SmartThings.

```

def delete = getChildDevices().findAll { !settings.devices.contains(it.deviceNetworkId) }

delete.each {
    deleteChildDevice(it.deviceNetworkId)
}

```

Also, when a Service Manager SmartApp is uninstalled, you need to remove its child devices.

```

def uninstalled() {
    removeChildDevices(getChildDevices())
}

private removeChildDevices(delete) {
    delete.each {
        deleteChildDevice(it.deviceNetworkId)
    }
}

```

Note: The `addChildDevice`, `getChildDevices`, and `deleteChildDevice` methods are a part of the *SmartApp* (page 248) API

Changes in Device Name

The device name is stored within the device and you need to monitor if it changes in the third party cloud.

Explicit Delete Actions

When a user manually deletes a device within the Things screen on the client device, you need to delete the child devices from within the Service Manager.

8.2.3 Building the Device Handler

The device handler for a cloud connected device is generally the same as any other device handler. The means in which it handles sending and receiving messages from its device is a little bit different. Let's walk through a cloud connected device handler example.

The Parse Method

The parse method for cloud connected devices will always be empty. In a cloud connected device, event data is passed down from the service manager, not from the device itself, so the parsing is handled in a separate method. The device type handler doesn't interface directly with a hardware device, which is what parse is used for.

Sending Commands to the Third-Party Cloud

Usually the actual implementation of device methods are delegated to its service manager. This is because the service manager is the entity that has the authentication information. To invoke a method on the parent service manager, you simply need to call it in the following format:

```
parent.methodName()
```

As with any other device-type, you need to define methods for all of the possible commands for the capabilities you'd like to support. Then when a user calls this method, it will pass information up to the parent service manager, who will make the direct connection to the third party cloud. You might for example want to turn a switch on, so you would call the following.

```
def on() {  
    parent.on(this)  
}
```

Receiving Events from the Third-Party Cloud

The device type handler continuously polls the third-party cloud through the service manager to check on the status of devices. When an event is fired, they can then be passed to the child device handler. Note that poll runs every 10 minutes for Service Manager SmartApps.

In the device-type handler:

```
def poll() {  
    results = parent.pollChildren()  
    parseEventData(results)  
}  
  
def parseEventData(Map results) {  
    results.each { name, value ->  
        //Parse events and optionally create SmartThings events  
    }  
}
```

In the service manager:

```
def pollChildren() {  
    def pollParams = [  
        uri: "https://api.thirdpartysite.com",  
        path: "/device",  
        requestContentType: "application/json",  
    ]  
}
```

```

    query: [format:"json",body: jsonRequestBody]
  ]

  httpGet(pollParams) { resp ->
    state.devices = resp.data.devices { collector, stat ->
      def dni = [ app.id, stat.identifier ].join('.')
      def data = [
        attribute1: stat.attributeValue,
        attribute2: stat.attribute2Value
      ]
      collector[dni] = [data:data]
      return collector
    }
  }
}

```

Generating Events at the Request of the Service Manager

You won't generate events directly within the Service Manager, but rather request that they are generated within the Device-type handler. For example:

In the service manager:

```
childName.generateEvent(data)
```

In the device handler:

```

def generateEvent(Map results) {
  results.each { name, value ->
    sendEvent(name: name, value: value)
  }
  return null
}

```

8.3 Building LAN-Connected Device Types

LAN connected devices communicate with the SmartThings hub over the LAN. An example of such a device is the Sonos system.

When developing a device handler for a LAN device, you must create a service manager SmartApp that will handle discovery of devices on the LAN, in some cases communicate with the device, and react to any device changes that occur via events.

This guide overviews the concept of the service manager/device handler architecture and also gives an example of both the service manager and device handler creation.

Table of Contents:

8.3.1 Division of Labor

The LAN connected device paradigm consists of a Service Manager and Device Handlers. The purpose of this guide is to introduce you to the core concepts of LAN connected device development, and provide some examples to help you get started.

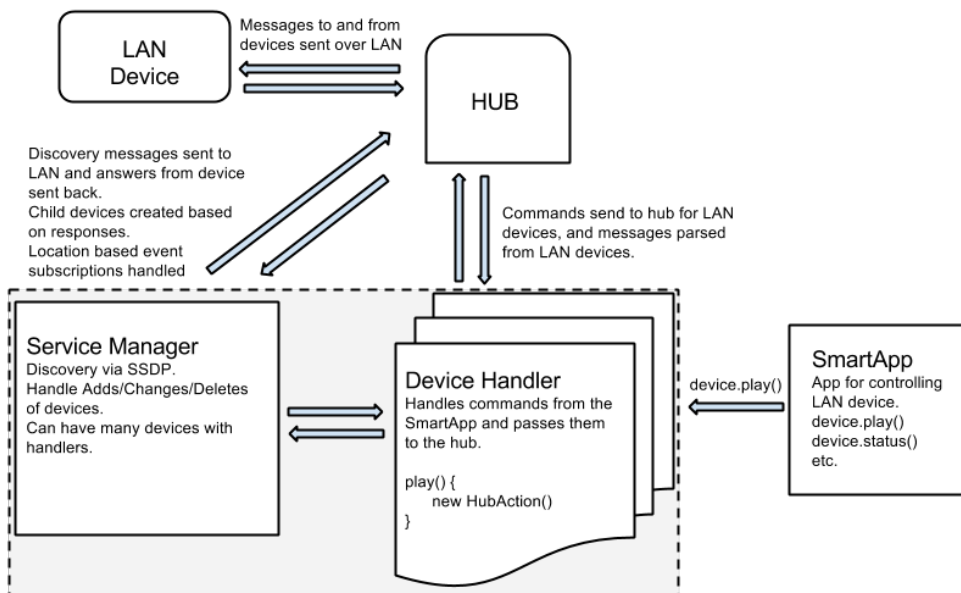
Service-Manager Responsibilities

The service manager is responsible for the discovery of the devices. It sends out a request and parses through the response, finding just the devices you are looking for. Upon discovery, it allows you to add device(s) that it has found. From there, it saves your connection to be able to make future interactions with the device.

Device Handler Responsibilities

The device type is responsible for creating and receiving device specific messages, and allowing them to work within the SmartThings infrastructure. It takes in a SmartApp specific command and outputs device specific commands. It also allows you to subscribe to responses from the device and trigger other commands as needed.

How It All Works



8.3.2 Building the Service Manager

The Service Manager's responsibilities are to:

- Discover devices
- Handles device Add/Change/Delete actions
- Maintains the connection

Let's take a look at an example of what is outlined above.

Discovery

SSDP

SSDP is the main protocol used to find devices on your network. It serves as the backbone of [Universal Plug and Play](#), which allows you easily connect new network devices to a system. To discovery new devices, you'd use something like this:

```
sendHubCommand(new physicalgraph.device.HubAction("lan discovery urn:schemas-upnp-org:device:ZonePlayer:1"))
```

Note: `sendHubCommand` and `HubAction` are supplied by the SmartThings framework

The class `physicalgraph.device.HubAction` encapsulates request information for communicating with the device.

When you create an instance of a `HubAction`, you provide details about the request, such as the request method, headers, and path. By itself, `HubAction` is little more than a wrapper for these request details.

This is an example of discovering devices using the LAN protocol. The main message to be sent through the hub is

```
lan discovery urn:schemas-upnp-org:device:ZonePlayer:1
```

The protocol is **lan**. The unique resource name (URN) is built with the Namespace Identifier (NID) **schemas-upnp-org** and Namespace Specific String (NSS) **device:ZonePlayer:1** according to the [URN Syntax Guide](#).

For Sonos, the device specific search term is **ZonePlayer:1**, but that will change per device. The search term for a particular device using UPnP should be published on documentation for the device, but you may also have to contact the manufacturer directly. The above command is typically called in a timing loop.

mDNS/DNS-SD

mDNS/DNS-SD is another popular protocol used to find devices on a network. It's made up of Multicast DNS and DNS-based service discovery. Known as Bonjour in the Apple ecosystem, Apple relies on mDNS/DNS-SD for services such as iChat or AppleTV.

How it Works

1. The device generates a unique IP address and calls out to the network to confirm the IP is not in use.
2. If the IP is not in use, it then passes a unique name to the network to confirm the name is not in use.
3. If the name not in use, the device starts a service.
4. The name is published as a record on the network, so others can find it.
5. The client searches for a particular device by name, and finds the device.

As a device type developer, you are responsible only for step 5 in the process.

More information on Bonjour can be found in [Apple's Developer documentation](#).

Your discovery request would look like this:

```
sendHubCommand(new physicalgraph.device.HubAction("lan discover mdns/dns-sd ._smarththings._tcp._site"))
```

The main message to be sent through the hub is

```
lan discover mdns/dns-sd ._smarththings._tcp._site
```

The protocol is **lan**, it's sent through **mdns/dns-sd** and the service's record name is ****._smarththings.tcp._site****.

Handling Updates (Adds/Changes/Deletes)

When there are changes within the scope of your devices, the service manager should handle those updates.

Adding Devices

A subscription is created to listen for a location event. The way the system is currently setup, we can't listen specifically for discovery events, but rather we listen for any location events, indicating a device has been added. Upon the event firing, a handler is called, in this case **locationHandler**.

```
if(!state.subscribe) {
  log.debug "subscribe to location"
  subscribe(location, null, locationHandler, [filterEvents:false])
  state.subscribe = true
}
```

And then later the locationHandler method is defined which adds the device to a collection of devices. Note that because we aren't just listening for discovery events, we have to parse the response to properly determine if a discovery has been made.

Within the LocationHandler, you need to see if the device is currently part of the devices collection in your state. You can check this via any unique identifier of your device. If it's not already registered in your state, go ahead and add it.

```
def locationHandler(evt) {
  def description = evt.description
  def hub = evt?.hubId

  def parsedEvent = parseEventMessage(description)
  parsedEvent << ["hub":hub]

  if (parsedEvent?.ssdpTerm?.contains("urn:schemas-upnp-org:device:DeviceIdentifier:1"))
  {
    def devices = getDevices()

    if (!(devices."${parsedEvent.ssdpUSN.toString()}"))
    {
      devices << ["${parsedEvent.ssdpUSN.toString()}":parsedEvent]
    }
  }

  def getDevices()
  {
    if (!state.devices) { state.devices = [:] }
    state.devices
  }
}
```

The example above uses SSDP, you could also use mDNS/DNS-SD. You just need to change what attributes are being used. For example, you could replace this:

```
if (parsedEvent?.ssdpTerm?.contains("urn:schemas-upnp-org:device:DeviceIdentifier:1"))
```

with this:

```
if(parsedEvent?.mdnsPath)
```

and this:


```
if (!(devices."${parsedEvent.ssdPUSN.toString()}"))
```

with this:

```
if (!(devices."${parsedEvent?.mac?.toString()}"))
```

Changing Devices

You need to monitor your devices networking information for changes. By using a unique identifier within your device, you can check that IP and port information hasn't changed.

Using SSDP:

```
if ((devices."${parsedEvent.ssdPUSN.toString()}")) {
    def d = devices."${parsedEvent.ssdPUSN.toString()}"
    boolean deviceChangedValues = false

    if(d.ip != parsedEvent.ip || d.port != parsedEvent.port) {
        d.ip = parsedEvent.ip
        d.port = parsedEvent.port
        deviceChangedValues = true
    }
}
```

Using mDNS/DNS-SD:

```
if ((devices."${parsedEvent?.mac?.toString()}")) {
    def d = device."${parsedEvent.mac.toString()}"
    boolean deviceChangedValues = false

    if(d.ip != parsedEvent.ip || d.port != parsedEvent.port) {
        d.ip = parsedEvent.ip
        d.port = parsedEvent.port
        deviceChangedValues = true
    }
}
```

If values did change, then you need to manually update your devices within the SmartApp.

```
if (deviceChangedValues) {
    def children = getChildDevices()
    children.each {
        if (it.getDeviceDataByName("mac") == parsedEvent.mac) {
            it.setDeviceNetworkId((parsedEvent.ip + ":" + parsedEvent.port)) //could error i
        }
    }
}
```

Deleting Devices

You don't need to handle deleting devices within the Service Manager. Devices, by nature, can become connected or disconnected at various times, and we still want them to persist. An example of this would be a laptop - if you were to take it with you somewhere, you'd still want it to pair properly later.

The enduser will need to manually delete their device within the SmartThings application.

Creating Child Devices

After you have discovered all your devices and the app has been installed, you need to add the device(s) the user has selected as a child device. You will iterate through a collection created from the user's input, and find just the devices they picked and add them.

```
selectedDevices.each { dni ->
    def d = getChildDevice(dni)
    if(!d) {
        def newDevice = devices.find { (it.value.ip + ":" + it.value.port) == dni }
        d = addChildDevice("smarththings", "Device Name", dni, newDevice?.value.hub, ["label":newDevice?.value.label])
        subscribeAll() //helper method to update devices
    }
}
```

Note: The `addChildDevice`, `getChildDevices`, and `deleteChildDevice` methods are a part of the [SmartApp](#) (page 248) API

8.3.3 Building the Device Type

The device handler for a LAN connected device is generally the same as any other device handler. The means in which it handles sending and receiving messages from its device is a little bit different. Let's walk through a LAN connected device handler example.

Making Outbound HTTP Calls with HubAction

Depending on the type of device you are using, you will send requests to your devices through the hub via REST or UPnP. You can do this using the SmartThings provided `HubAction` class.

Overview

The class `physicalgraph.device.HubAction` encapsulates request information for communicating with the device.

When you create an instance of a `HubAction`, you provide details about the request, such as the request method, headers, and path. By itself, `HubAction` is little more than a wrapper for these request details.

It is when an instance of a `HubAction` is returned from a command method that it becomes useful.

When a command method of your device handler returns an instance of a `HubAction`, the SmartThings platform will use the request information within it to actually perform the request. It will then call the device-handler's `parse` method with any response data.

Herein lies an important point - *if your `HubAction` instance is not returned from your command method, no request will be made.* It will just be an object allocating system memory. Not very useful.

So remember - the `HubAction` instance should be returned from your command method so that the platform can make the request!

Creating a HubAction Object

To create a HubAction object, you can pass in a map of parameters to the constructor that defines the request information:

```
def result = new physicalgraph.device.HubAction(
    method: "GET",
    path: "/somepath",
    headers: [
        HOST: "device IP address"
    ],
    query: [param1: "value1", param2: "value2"]
)
```

A brief discussion of the options that can be provided follows:

method The HTTP method to use for the request.

path The path to send the request to. You can add URL parameters to the request directly, or use the `query` option.

headers A map of HTTP headers and their values for this request. This is where you will provide the IP address of the device as the `HOST`.

query A map of query parameters to use in this request. You can use URL parameters directly on the path if you wish, instead of using this option.

Parsing the Response

When you make a request to your device using HubAction, any response will be passed to your device-handler's `parse` method, just like other device messages.

You can use the `parseLanMessage` method to parse the incoming message.

`parseLanMessage` example:

```
def parse(description) {
    ...
    def msg = parseLanMessage(description)

    def headersAsString = msg.header // => headers as a string
    def headerMap = msg.headers      // => headers as a Map
    def body = msg.body              // => request body as a string
    def status = msg.status          // => http status code of the response
    def json = msg.json              // => any JSON included in response body, as a data structure object
    def xml = msg.xml                // => any XML included in response body, as a document tree structure
    def data = msg.data              // => either JSON or XML in response body (whichever is specified)

    ...
}
```

For more information about the JSON or XML response formats, see the Groovy [JsonSlurper](#) and [XmlSlurper](#) documentation.

Getting the Addresses

To use `HubAction`, you will need the IP address of the device, and sometimes the hub.

How the device IP and port are stored may vary depending on the device type. There's currently not a public API to get this information easily, so until there is, you will need to handle this in your device-type handler. Consider using helper methods like these to get this information:

```
// gets the address of the hub
private getCallbackAddress() {
    return device.hub.getDataValue("localIP") + ":" + device.hub.getDataValue("localSrvPortTCP")
}

// gets the address of the device
private getHostAddress() {
    def ip = getDataValue("ip")
    def port = getDataValue("port")

    if (!ip || !port) {
        def parts = device.deviceNetworkId.split(":")
        if (parts.length == 2) {
            ip = parts[0]
            port = parts[1]
        } else {
            log.warn "Can't figure out ip and port for device: ${device.id}"
        }
    }

    log.debug "Using IP: $ip and port: $port for device: ${device.id}"
    return convertHexToIP(ip) + ":" + convertHexToInt(port)
}

private Integer convertHexToInt(hex) {
    return Integer.parseInt(hex, 16)
}

private String convertHexToIP(hex) {
    return [convertHexToInt(hex[0..1]), convertHexToInt(hex[2..3]), convertHexToInt(hex[4..5]), convertHexToInt(hex[6..7])]
}
```

You'll see the rest of the examples in this document use these helper methods.

REST Requests

`HubAction` can be used to make [REST](#) calls to communicate with the device.

Here's a quick example:

```
def myCommand() {
    def result = new physicalgraph.device.HubAction(
        method: "GET",
        path: "/yourpath?param1=value1&param2=value2",
        headers: [
            HOST: getHostAddress()
        ]
    )
}
```

```

    return result
}

```

UPnP/SOAP Requests

Alternatively, after making the initial connection you can use UPnP. UPnP uses **SOAP** (Simple Object Access Protocol) messages to communicate with the device.

SmartThings provides the `HubSoapAction` class for this purpose. It is similar to the `HubAction` class (it actually extends the `HubAction` class), but it will handle creating the soap envelope for you.

Here's an example of using `HubSoapAction`:

```

def someCommandMethod() {
    return doAction("SetVolume", "RenderingControl", "/MediaRenderer/RenderingControl/Control", [Inst
}

def doAction(action, service, path, Map body = [InstanceID:0, Speed:1]) {
    def result = new physicalgraph.device.HubSoapAction(
        path:      path,
        urn:        "urn:schemas-upnp-org:service:$service:1",
        action:     action,
        body:       body,
        headers:    [Host:getHostAddress(), CONNECTION: "close"]
    )
    return result
}

```

Subscribing to Device Events

If you'd like to hear back from a LAN connected device upon a particular event, you can subscribe using a `HubAction`. The `parse` method will be called when this event is fired on the device.

Here's an example using UPnP:

```

def someCommand() {
    subscribeAction("/path/of/event")
}

private subscribeAction(path, callbackPath="") {
    log.trace "subscribe($path, $callbackPath)"
    def address = getCallBackAddress()
    def ip = getHostAddress()

    def result = new physicalgraph.device.HubAction(
        method: "SUBSCRIBE",
        path: path,
        headers: [
            HOST: ip,
            CALLBACK: "<http://$address/notify$callbackPath>",
            NT: "upnp:event",
            TIMEOUT: "Second-28800"
        ]
    )
}

```

```
)  
  
    log.trace "SUBSCRIBE $path"  
  
    return result  
}
```

References and Resources

- [UPnP](#)
- [SOAP](#)
- [REST](#)

Arduino ThingShield

Using the SmartThings Arduino Shield (ThingShield), you can add SmartThings capability to any Arduino compatible board with the R3 pinout, including the Uno, Mega, Duemilanove, and Leonardo.

Specs:

- Works with: Uno, Mega, Duemilanove, Leonardo
- Dimensions: 2.5 x 1.9 x 0.3"
- Weight: 8 ounces

The ThingShield is available for purchase on our [online shop](#).

9.1 Installing the Library

To install, copy the entire SmartThings directory into the 'libraries' directory in your sketchbook. Your sketchbook location is set in the Arduino IDE preferences, by default, the location will be:

Windows: 'My Documents\Arduinolibraries\SmartThings'

OSX: '~/Documents/Arduino/libraries/SmartThings'

You can download the [SmartThings Arduino Library](#) here.

9.2 Pairing the Shield

To join the shield to your SmartThings hub, go to "Add SmartThings" mode in the SmartThings app by hitting the "+" icon in the desired location, and then press the Switch button on the shield. You should see the shield appear in the app.

To unpair the shield, press and hold the Switch button for 6 seconds and release. The shield will now be unpaired from your SmartThings Hub. Make sure to delete from your account if you plan to re-pair it!

9.3 Changing the Device Type

By changing the device type in the SmartThings cloud you can change how to interact with your Arduino + ThingShield. When a shield first pairs, it has no functionality and only serves to help identify the device in the mobile app. We have some pre-built device types that you can use for most functionality. One pre-built Arduino device handler is the "On/Off Shield (example)"

To change your device type, log into <http://graph.api.smarthings.com/> and click on “Devices” Navigate to and click on the Arduino ThingShield then click on “Edit” on the bottom left of the page.

Select the “Type” drop down menu.

Choose “On/Off Shield (example)”

Hit the “Update” button

Your Arduino will now be able to accept the commands “on” “off”, and “hello”

[Here is what the Arduino sketch looks like](#) and [here is the device handler](#).

[Here is a different device type that can read a string sent from an Arduino and display it in a tile.](#)

9.4 Arduino Examples

We have created some example Arduino Sketches (code) to use as a reference for building your own devices. The following is meant to go with the “On/Off Shield (example)” device type.

[Download all of our examples here.](#)

Capabilities Reference

Capabilities are core to the SmartThings architecture. They allow us to abstract specific devices into their underlying capabilities.

An application interacts with devices based on their capabilities, so once we understand the capabilities that are needed by a SmartApp, and the capabilities that are provided by a device, we can understand which devices (based on the Device's declared capabilities) are eligible for use within a specific SmartApp.

Capabilities themselves are decomposed into both Commands and Attributes. Commands represent ways in which you can control or actuate the device, whereas Attributes represent state information or properties of the device.

Capabilities are created and maintained by the SmartThings internal development team.

This page serves as a reference for the supported capabilities.

Note: This document is a work in progress. Many capabilities are not yet fully documented. We are continually working to document all the capabilities.

10.1 At a Glance

The Capabilities reference table below lists all capabilities. The various columns are:

Name: The name of the capability that is used by a Device Handler.

Preferences Reference: The string you would use in a SmartApp to allow a user to select from devices supporting this capability.

Attributes: The attributes that the capability defines.

Commands: The commands (and their signatures) that the capability defines.

Name	Preferences Reference	Attributes	Commands
<i>Acceleration</i> <i>Sensor</i> (page 221)	capability.accelerationSensor	<ul style="list-style-type: none"> acceleration 	
<i>Actuator</i> (page 222)	capability.actuator		
Continued on next page			

Table 10.1 – continued from previous page

Name	Preferences Reference	Attributes	Commands
<i>Alarm</i> (page 222)	capability.alarm	<ul style="list-style-type: none"> alarm 	<ul style="list-style-type: none"> off() strobe() siren() both()
<i>Battery</i> (page 223)	capability.battery	<ul style="list-style-type: none"> battery 	
<i>Beacon</i> (page 224)	capability.beacon	<ul style="list-style-type: none"> presence 	
<i>Button</i> (page 224)	capability.button	<ul style="list-style-type: none"> button 	
<i>Carbon Monoxide Detector</i> (page 225)	capability.carbonMonoxideDetector	<ul style="list-style-type: none"> carbonMonoxide 	
<i>Color Control</i> (page 226)	capability.colorControl	<ul style="list-style-type: none"> hue saturation color 	<ul style="list-style-type: none"> setHue(number) setSaturation(number) setColor(color_map)
<i>Configuration</i> (page 227)	capability.configuration		<ul style="list-style-type: none"> configure()
<i>Contact Sensor</i> (page 227)	capability.contactSensor	<ul style="list-style-type: none"> contact 	
<i>Door Control</i> (page 228)	capability.doorControl	<ul style="list-style-type: none"> door 	<ul style="list-style-type: none"> open() close()
<i>Energy Meter</i> (page 228)	capability.energyMeter	<ul style="list-style-type: none"> energy 	
<i>Illuminance Measurement</i> (page 229)	capability.illuminanceMeasurement	<ul style="list-style-type: none"> illuminance 	
<i>Image Capture</i> (page 230)	capability.imageCapture	<ul style="list-style-type: none"> image 	<ul style="list-style-type: none"> take()
<i>Lock</i> (page 230)	capability.lock	<ul style="list-style-type: none"> lock 	<ul style="list-style-type: none"> lock() unlock()
Continued on next page			

Table 10.1 – continued from previous page

Name	Preferences Reference	Attributes	Commands
<i>Media Controller</i> (page 231)	capability.mediaController	<ul style="list-style-type: none"> activities currentActivity 	<ul style="list-style-type: none"> startActivity(string) getAllActivities() getCurrentActivity()
<i>Momentary</i> (page 231)	capability.momentary		<ul style="list-style-type: none"> push()
<i>Motion Sensor</i> (page 232)	capability.motionSensor	<ul style="list-style-type: none"> motion 	
<i>Music Player</i> (page 233)	capability.musicPlayer	<ul style="list-style-type: none"> status level trackDescription trackData mute 	<ul style="list-style-type: none"> play() pause() stop() nextTrack() playTrack(string) setLevel(number) playText(string) mute() previousTrack() unmute() setTrack(string) resumeTrack(string) restoreTrack(string)
<i>Notification</i> (page 234)	capability.notification		<ul style="list-style-type: none"> deviceNotification(string)
<i>Polling</i> (page 234)	capability.polling		<ul style="list-style-type: none"> poll()
<i>Power Meter</i> (page 234)	capability.powerMeter	<ul style="list-style-type: none"> power 	
<i>Presence Sensor</i> (page 235)	capability.presenceSensor	<ul style="list-style-type: none"> presence 	
<i>Refresh</i> (page 236)	capability.refresh		<ul style="list-style-type: none"> refresh()
<i>Relative Humidity Measurement</i> (page 236)	capability.relativeHumidityMeasurement	<ul style="list-style-type: none"> humidity 	
<i>Relay Switch</i> (page 237)	capability.relaySwitch	<ul style="list-style-type: none"> switch 	<ul style="list-style-type: none"> on() off()
<i>Sensor</i> (page 237)	capability.sensor		

Continued on next page

Table 10.1 – continued from previous page

Name	Preferences Reference	Attributes	Commands
<i>Signal Strength</i> (page 237)	capability.signalStrength	<ul style="list-style-type: none"> lqi rsi 	
<i>Sleep Sensor</i> (page 238)	capability.sleepSensor	<ul style="list-style-type: none"> sleeping 	
<i>Smoke Detector</i> (page 238)	capability.smokeDetector	<ul style="list-style-type: none"> smoke 	
<i>Speech Synthesis</i> (page 238)	capability.speechSynthesis		<ul style="list-style-type: none"> speak(string)
<i>Step Sensor</i> (page 239)	capability.stepSensor	<ul style="list-style-type: none"> steps goal 	
<i>Switch</i> (page 239)	capability.switch	<ul style="list-style-type: none"> switch 	<ul style="list-style-type: none"> on() off()
<i>Switch Level</i> (page 240)	capability.switchLevel	<ul style="list-style-type: none"> level 	<ul style="list-style-type: none"> setLevel(number, number)
<i>Temperature Measurement</i> (page 240)	capability.temperatureMeasurement	<ul style="list-style-type: none"> temperature 	
<i>Thermostat</i> (page 241)	capability.thermostat	<ul style="list-style-type: none"> temperature heatingSetpoint coolingSetpoint thermostatSetpoint thermostatMode thermostatFanMode thermostatOperatingState 	<ul style="list-style-type: none"> setHeatingSetpoint(number) setCoolingSetpoint(number) off() heat() emergencyHeat() cool() setThermostatMode(enum) fanOn() fanAuto() fanCirculate() setThermostatFanMode(enum) auto()
<i>Thermostat Cooling Setpoint</i> (page 242)	capability.thermostatCoolingSetpoint	<ul style="list-style-type: none"> coolingSetpoint 	<ul style="list-style-type: none"> setCoolingSetpoint(number)
Continued on next page			

Table 10.1 – continued from previous page

Name	Preferences Reference	Attributes	Commands
<i>Thermostat Fan Mode</i> (page 242)	capability.thermostatFanMode	<ul style="list-style-type: none"> thermostatFanMode 	<ul style="list-style-type: none"> fanOn() fanAuto() fanCirculate() setThermostatFanMode(enum)
<i>Thermostat Heating Setpoint</i> (page 242)	capability.thermostatHeatingSetpoint	<ul style="list-style-type: none"> heatingSetpoint 	<ul style="list-style-type: none"> setHeatingSetpoint(number)
<i>Thermostat Mode</i> (page 243)	capability.thermostatMode	<ul style="list-style-type: none"> thermostatMode 	<ul style="list-style-type: none"> off() heat() emergencyHeat() cool() auto() setThermostatMode(enum)
<i>Thermostat Operating State</i> (page 243)	capability.thermostatOperatingState	<ul style="list-style-type: none"> thermostatOperatingState 	
<i>Thermostat Setpoint</i> (page 243)	capability.thermostatSetpoint	<ul style="list-style-type: none"> thermostatSetpoint 	
<i>Three Axis</i> (page 244)	capability.threeAxis	<ul style="list-style-type: none"> threeAxis 	
<i>Tone</i> (page 244)	capability.tone		<ul style="list-style-type: none"> beep()
<i>Touch Sensor</i> (page 244)	capability.touchSensor	<ul style="list-style-type: none"> touch 	
<i>Valve</i> (page 245)	capability.valve	<ul style="list-style-type: none"> contact 	<ul style="list-style-type: none"> open() close()
<i>Water Sensor</i> (page 245)	capability.waterSensor	<ul style="list-style-type: none"> water 	

10.2 Acceleration Sensor

The Acceleration Sensor capability allows for acceleration detection.

Some use cases for SmartApps using this capability would be detecting if a washing machine is vibrating, or if a case has moved (particularly useful for knowing if a weapon case has been moved).

Capability Name	Preferences Reference
Acceleration Sensor	capability.accelerationSensor

Attributes:

Attribute	Type	Possible Values
acceleration	String	"active" if acceleration is detected. "inactive" if no acceleration is detected.

Commands:

None.

SmartApp Example

```
// preferences reference
preferences {
    input "accelerationSensor", "capability.accelerationSensor"
}

def installed() {
    // subscribe to active acceleration
    subscribe(accelerationSensor, "acceleration.active", accelerationActiveHandler)

    // subscribe to inactive acceleration
    subscribe(accelerationSensor, "acceleration.inactive", accelerationInactiveHandler)

    // subscribe to all acceleration events
    subscribe(accelerationSensor, "acceleration", accelerationBothHandler)
}
```

10.3 Actuator

The Actuator capability is a “tagging” capability. It defines no attributes or commands.

In SmartThings terms, it represents that a Device has commands.

10.4 Alarm

The Alarm capability allows for interacting with devices that serve as alarms.

Note: Z-Wave sometimes uses the term “Alarm” to refer to an important notification. The *Alarm* Capability is used in SmartThings to define a device that acts as an Alarm in the traditional sense (e.g., has a siren and such).

Capability Name	SmartApp Preferences Reference
Alarm	capability.alarm

Attributes:

Attribute	Type	Possible Values
alarm	String	"strobe" if the alarm is strobing. "siren" if the alarm is sounding the siren. "off" if the alarm is turned off. "both" if the alarm is strobing and sounding the alarm.

Commands:*strobe()* Strobe the alarm*siren()* Sound the siren on the alarm*both()* Strobe and sound the alarm*off()* Turn the alarm (siren and strobe) off**SmartApp Example:**

```
// preferences reference
preferences {
    input "alarm", "capability.alarm"
}

def installed() {
    // subscribe to alarm strobe
    subscribe(alarm, "alarm.strobe", strobeHandler)
    // subscribe to all alarm events
    subscribe(alarm, "alarm", allAlarmHandler)
}

def strobeHandler(evt) {
    log.debug "${evt.value}" // => "strobe"
}

def allAlarmHandler(evt) {
    if (evt.value == "strobe") {
        log.debug "alarm strobe"
    } else if (evt.value == "siren") {
        log.debug "alarm siren"
    } else if (evt.value == "both") {
        log.debug "alarm siren and alarm"
    } else if (evt.value == "off") {
        log.debug "alarm turned off"
    } else {
        log.debug "unexpected event: ${evt.value}"
    }
}
```

10.5 Battery

Defines that the device has a battery.

Capability Name	SmartApp Preferences Reference
Battery	capability.battery

Attributes:

Attribute	Type	Possible Values
battery		

Commands:

None

SmartApp Example:

```
preferences {
    section() {
        input "thebattery", "capability.battery"
    }
}

def installed() {
    def batteryValue = thebattery.latestValue("battery")
    log.debug "latest battery value: $batteryValue"
    subscribe(thebattery, "battery", batteryHandler)
}

def batteryHandler(evt) {
    log.debug "battery attribute changed to ${evt.value}"
}
```

10.6 Beacon

Capability Name	SmartApp Preferences Reference
Beacon	capability.beacon

Attributes:

Attribute	Type	Possible Values
presence	String	"present" "not present"

Commands:

None.

SmartApp Example:

```
preferences {
    section() {
        input "thebeacon", "capability.beacon"
    }
}

def installed() {
    def currBeacon = thebeacon.currentValue("presence")
    log.debug "beacon is currently: $currBeacon"
    subscribe(thebeacon, "presence", beaconHandler)
}

def beaconHandler(evt) {
    log.debug "beacon presence is: ${evt.value}"
}
```

10.7 Button

Capability Name	SmartApp Preferences Reference
Button	capability.button

Attributes:

Attribute	Type	Possible Values
button	String	"held" if the button is held (longer than a push) "pushed" if the button is pushed

Commands:

None.

SmartApp Code Example:

```

preferences {
    section() {
        input "thebutton", "capability.button"
    }
}

def installed() {
    // subscribe to any change to the "button" attribute
    // if we wanted to only subscribe to the button be held, we would use
    // subscribe(thebutton, "button.held", buttonHeldHandler), for example.
    subscribe(thebutton, "button", buttonHandler)
}

def buttonHandler(evt) {
    if (evt.value == "held") {
        log.debug "button was held"
    } else if (evt.value == "pushed") {
        log.debug "button was pushed"
    }

    // Some button devices may have more than one button. While the
    // specific implementation varies for different devices, there may be
    // button number information in the jsonData of the event:
    try {
        def data = evt.jsonData
        def buttonNumber = data.buttonNumber as Integer
        log.debug "evt.jsonData: $data"
        log.debug "button number: $buttonNumber"
    } catch (e) {
        log.warn "caught exception getting event data as json: $e"
    }
}

```

10.8 Carbon Monoxide Detector

Capability Name	SmartApp Preferences Reference
Carbon Monoxide Detector	capability.carbonMonoxideDetector

Attributes:

Attribute	Type	Possible Values
carbonMonoxide	String	"tested" "clear" "detected"

Commands:

None.

SmartApp Example:

```

preferences {
    section() {
        input "smoke", "capability.smokeDetector", title: "Smoke Detected", required: false, multiple: true
    }
}

def installed() {
    subscribe(smoke, "carbonMonoxide.detected", smokeHandler)
}

def smokeHandler(evt) {
    log.debug "carbon alert: ${evt.value}"
}

```

10.9 Color Control

Capability Name	SmartApp Preferences Reference
Color Control	capability.colorControl

Attributes:

Attribute	Type	Possible Values
hue	Number	0-100 (percent)
saturation	Number	0-100 (percent)
color	Map	See the table below for the color options

Color Options:

key	value
hue	0-100 (percent)
saturation	0-100 (percent)
hex	"#000000" - "#FFFFFF" (Hex)
level	0-100 (percent)
switch	"on" or "off"

Commands:

setHue(number) Sets the colors hue value

setSaturation(number) Sets the colors saturation value

setColor(color_map) Sets the color to the passed in maps values

SmartApp Example:

```

preferences {
    section("Title") {
        input "contact", "capability.contactSensor", title: "contact sensor", required: true, multiple: true
        input "bulb", "capability.colorControl", title: "pick a bulb", required: true, multiple: false
    }
}

def installed() {
    subscribe(contact, "contact", contactHandler)
}

```

```
def contactHandler(evt) {
    if("open" == "$evt.value") {
        bulb.on() // Turn the bulb on when open (this method does not come directly from the colorContr
        bulb.setHue(80)
        bulb.setSaturation(100) // Set the color to something fancy
        bulb.setLevel(100) // Make sure the light brightness is 100%
    } else {
        bulb.off() // Turn the bulb off when closed (this method does not come directly from the colorC
    }
}
```

10.10 Configuration

Note: This capability is meant to be used only in device handlers. The implementation of the `configure()` method will be very specific to the physical device. The commands that populate the `configure()` method will most likely be found in the device manufacturer's documentation. During the device installation lifecycle, the `configure()` method is called after the device has been assigned a Device Handler.

Capability Name	SmartApp Preferences Reference
Configuration	capability.configuration

Attributes:

None.

Commands:

`configure()` This is where the device specific configuration commands can be implemented.

Device Handler Example:

```
def configure() {
    // update reporting frequency
    def cmd = delayBetween([
        zwave.configurationV1.configurationSet(parameterNumber: 101, size: 4, scaledConfigurationValue: 4
        zwave.configurationV1.configurationSet(parameterNumber: 111, size: 4, scaledConfigurationValue: 3
        zwave.configurationV1.configurationSet(parameterNumber: 102, size: 4, scaledConfigurationValue: 8
        zwave.configurationV1.configurationSet(parameterNumber: 112, size: 4, scaledConfigurationValue: 3
        zwave.configurationV1.configurationSet(parameterNumber: 103, size: 4, scaledConfigurationValue: 3
        zwave.configurationV1.configurationSet(parameterNumber: 113, size: 4, scaledConfigurationValue: 3
    ])
    log.debug cmd
    cmd
}
```

10.11 Contact Sensor

Capability Name	SmartApp Preferences Reference
Contact Sensor	capability.contactSensor

Attributes:

Attribute	Type	Possible Values
contact	String	"open" "closed"

Commands:

None.

SmartApp Example:

```
preferences {
    section("Contact Example") {
        input "contact", "capability.contactSensor", title: "pick a contact sensor", required: true, multiple: true
    }
}

def installed() {
    subscribe(contact, "contact", contactHandler)
}

def contactHandler(evt) {
    if("open" == evt.value)
        // contact was opened, turn on a light maybe?
        log.debug "Contact is in ${evt.value} state"
    if("closed" == evt.value)
        // contact was closed, turn off the light?
        log.debug "Contact is in ${evt.value} state"
}
```

10.12 Door Control

Capability Name	SmartApp Preferences Reference
Door Control	capability.doorControl

Attributes:

Attribute	Type	Possible Values
door	String	"unknown" "closed" "open" "closing" "opening"

Commands:

open() Opens the door

close() Closes the door

10.13 Energy Meter

Capability Name	SmartApp Preferences Reference
Energy Meter	capability.energyMeter

Attributes:

Attribute	Type	Possible Values
energy	Number	numeric value representing energy consumption

Commands:

None.

```
preferences {
    section("Title") {
        input "outlet", "capability.switch", title: "outlet", required: true, multiple: false
    }
}

def installed() {
    subscribe(outlet, "energy", myHandler)
    subscribe(outlet, "switch", myHandler)
}

def myHandler(evt) {
    log.debug "$outlet.currentEnergy"
}
```

10.14 Illuminance Measurement

Capability Name	SmartApp Preferences Reference
Illuminance Measurement	capability.illuminanceMeasurement

Attributes:

Attribute	Type	Possible Values
illuminance	Number	numeric value representing illuminance

Commands:

None.

SmartApp Example:

```
preferences {
    section("Title") {
        input "lightSensor", "capability.illuminanceMeasurement"
        input "light", "capability.switch"
    }
}

def installed() {
    subscribe(lightSensor, "illuminance", myHandler)
}

def myHandler(evt) {
    def lastStatus = state.lastStatus
    if (lastStatus != "on" && evt.integerValue < 30) {
        light.on()
        state.lastStatus = "on"
    }
    else if (lastStatus != "off" && evt.integerValue > 50) {
        light.off()
        state.lastStatus = "off"
    }
}
```

10.15 Image Capture

Capability Name	SmartApp Preferences Reference
Image Capture	capability.imageCapture

Attributes:

Attribute	Type	Possible Values
image	String	string value representing the image captured

Commands:

take() Capture an image

SmartApp Example:

```
preferences {
  section("Choose one or more, when..."){
    input "motion", "capability.motionSensor", title: "Motion Here", required: false, multiple: true
  }
  section("Take a burst of pictures") {
    input "camera", "capability.imageCapture"
  }
}

def installed() {
  subscribe(motion, "motion.active", takePhotos)
}

def takePhotos(evt) {
  camera.take()
  (1..4).each {
    camera.take(delay: (1000 * it))
  }
  log.debug "$camera.currentImage"
}
```

10.16 Lock

Capability Name	SmartApp Preferences Reference
Lock	capability.lock

Attributes:

Attribute	Type	Possible Values
lock	String	"locked" "unlocked"

Commands:

lock() Lock the device

unlock() Unlock the device

SmartApp Example:

```

preferences {
    section("Title") {
        input "lock", "capability.lock", title:"door lock", required: true, multiple: false
        input "motion", "capability.motionSensor", title:"motion", required: true, multiple: false
    }
}

def installed() {
    subscribe(motion, "motion", myHandler)
}

def myHandler(evt) {
    if(!("locked" == lock.currentLock) && "active" == evt.value) {
        lock.lock()
    }
}

```

10.17 Media Controller

Capability Name	SmartApp Preferences Reference
Media Controller	capability.mediaController

Attributes:

Attribute	Type	Possible Values
activities		
currentActivity		

Commands:

startActivity(string) Start the activity with the given name

getAllActivities() Get a list of all the activities

getCurrentActivity() Get the current activity

10.18 Momentary

Capability Name	SmartApp Preferences Reference
Momentary	capability.momentary

Attributes:

None.

Commands:

push() Press the momentary switch

SmartApp Example:

```
preferences {
  section("Title") {
    input "doorOpener", "capability.momentary", title: "Door Opener", required: true, multiple: false
    input "presence", "capability.presenceSensor", title: "presence", required: true, multiple: false
  }
}

def installed() {
  subscribe(presence, "presence", myHandler)
}

def myHandler(evt) {
  if("present" == evt.value) {
    doorOpener.push()
  }
}
```

10.19 Motion Sensor

Capability Name	SmartApp Preferences Reference
Motion Sensor	capability.motionSensor

Attributes:

Attribute	Type	Possible Values
motion	String	"active" "inactive"

Commands:

None.

SmartApp Example:

```
preferences {
  section("Choose one or more, when..."){
    input "motion", "capability.motionSensor", title: "Motion Here", required: true, multiple: true
    input "myswitch", "capability.switch", title: "switch", required: true, multiple: false
  }
}

def installed() {
  subscribe(motion, "motion", myHandler)
}

def myHandler(evt) {
  if("active" == evt.value) {
    myswitch.on()
  } else if("inactive" == evt.value) {
    myswitch.off()
  }
}
```

10.20 Music Player

Note: The music player capability is still under development. It currently supports the Sonos system and as such is implemented in a way that is tailored to Sonos.

Capability Name	SmartApp Preferences Reference
Music Player	capability.musicPlayer

Attributes:

Attribute	Type	Possible Values
status	String	state of the music player as a string
level	Number	0-100 (percent)
trackDescription	String	description of the current playing track
trackData	JSON	a JSON data structure that represents current track data
mute	String	"muted" "unmuted"

Commands:

play() Start music playback

pause() Pause music playback

stop() Stop music playback

nextTrack() Advance to next track

playTrack(string) Play the track matching the given string (the string is a URI for the track to be played)

setLevel(number) Set the volume to the specified level (the number represents a percent)

playText(string) play the given string as text to speech

mute() Mute playback

previousTrack() Go back to the previous track

unmute() Unmute playback

setTrack(string) Set the track to be played (does not play the track)

resumeTrack(map) Set and play the given track and maintain queue position

restoreTrack(map) Restore the track with the given data

SmartApp Example:

```

preferences {
    section("Title") {
        input "player", "capability.musicPlayer", title: "music player", required: true, multiple: false
        input "frontDoor", "capability.contactSensor", title: "front door", required: true, multiple: false
    }
}

def installed() {
    subscribe(frontDoor, "contact", myHandler)
}

def myHandler(evt) {
    if("open" == evt.value) {

```

```
player.playText("The front door is open")
}
}
```

10.21 Notification

Capability Name	SmartApp Preferences Reference
Notification	capability.notification

Attributes:

None.

Commands:

deviceNotification(string) Send the device the specified notification.

10.22 Polling

Capability Name	SmartApp Preferences Reference
Polling	capability.polling

Attributes:

None.

Commands:

poll() Poll devices

10.23 Power Meter

Capability Name	SmartApp Preferences Reference
Power Meter	capability.powerMeter

Attributes:

Attribute	Type	Possible Values
power		

Commands:

None.

SmartApp Example:

```

preferences {
    section {
        input(name: "meter", type: "capability.powerMeter", title: "When This Power Meter...", required: true, description: "Reports Above...", required: true, description: "Reports Above...")
        input(name: "threshold", type: "number", title: "Reports Above...", required: true, description: "Reports Above...")
    }
    section {
        input(name: "switches", type: "capability.switch", title: "Turn Off These Switches", required: true, description: "Turn Off These Switches")
    }
}

def installed() {
    subscribe(meter, "power", meterHandler)
}

def meterHandler(evt) {
    def meterValue = evt.value as double
    def thresholdValue = threshold as int
    if (meterValue > thresholdValue) {
        log.debug "${meter} reported energy consumption above ${threshold}. Turning off switches."
        switches.off()
    }
}

```

10.24 Presence Sensor

Capability Name	SmartApp Preferences Reference
Presence Sensor	capability.presenceSensor

Attributes:

Attribute	Type	Possible Values
Presence	String	"present" "not present"

Commands:

None.

SmartApp Example:

```

preferences {
    section("Title") {
        input "presence", "capability.presenceSensor", title: "presence", required: true, multiple: false
        input "myswitch", "capability.switch", title: "switch", required: true, multiple: true
    }
}

def installed() {
    subscribe(presence, "presence", myHandler)
}

def myHandler(evt) {
    if("present" == evt.value) {
        myswitch.on()
    } else {
        myswitch.off()
    }
}

```

```
}  
}
```

10.25 Refresh

Capability Name	SmartApp Preferences Reference
Refresh	capability.refresh

Attributes:

None.

Commands:

refresh() Refresh

10.26 Relative Humidity Measurement

Capability Name	SmartApp Preferences Reference
Relative Humidity Measurement	capability.relativeHumidityMeasurement

Attributes:

Attribute	Type	Possible Values
humidity		

Commands:

None.

SmartApp Example:

```
preferences {  
    section("Bathroom humidity sensor") {  
        input "bathroom", "capability.relativeHumidityMeasurement", title: "Which humidity sensor?"  
    }  
    section("Coffee maker to turn on") {  
        input "coffee", "capability.switch", title: "Which switch?"  
    }  
}  
  
def installed() {  
    subscribe(bathroom, "humidity", coffeeMaker)  
}  
  
def coffeeMaker(shower) {  
    if (shower.value.toInteger() > 50) {  
        coffee.on()  
    }  
}  
}
```

10.27 Relay Switch

Capability Name	SmartApp Preferences Reference
Relay Switch	capability.relaySwitch

Attributes:

Attribute	Type	Possible Values
switch	String	"off" "on"

Commands:

off() Turn the switch off

on() Turn the switch on

10.28 Sensor

Capability Name	SmartApp Preferences Reference
Sensor	capability.sensor

Attributes:

None.

Commands:

None.

10.29 Signal Strength

Capability Name	SmartApp Preferences Reference
Signal Strength	capability.signalStrength

Attributes:

Attribute	Type	Possible Values
lqi	Number	A number representing the Link Quality Indication
rssi	Number	A number representing the Received Signal Strength Indication

Commands:

None.

10.30 Sleep Sensor

Capability Name	SmartApp Preferences Reference
Sleep Sensor	capability.sleepSensor

Attributes:

Attribute	Type	Possible Values
sleeping	String	"not sleeping" "sleeping"

Commands:

None.

10.31 Smoke Detector

Capability Name	SmartApp Preferences Reference
Smoke Detector	capability.smokeDetector

Attributes:

Attribute	Type	Possible Values
smoke	String	"detected" "clear" "tested"

Commands:

None.

SmartApp Example:

```
preferences {
    section("Title") {
        input "smoke", "capability.smokeDetector", title: "smoke", required: true, multiple: false
    }
}

def installed() {
    subscribe(smoke, "smoke", myHandler)
}

def myHandler(evt) {
    if("detected" == evt.value) {
        // Sound an alarm! Send a SMS! or Change a HUE bulb color
    }
}
```

10.32 Speech Synthesis

Capability Name	SmartApp Preferences Reference
Speech Synthesis	capability.speechSynthesis

Attributes:

None.

Commands:*speak(string)* It can talk!

10.33 Step Sensor

Capability Name	SmartApp Preferences Reference
Step Sensor	capability.stepSensor

Attributes:

Attribute	Type	Possible Values
steps		
goal		

Commands:

None.

10.34 Switch

Capability Name	SmartApp Preferences Reference
Switch	capability.switch

Attributes:

Attribute	Type	Possible Values
switch	String	"off" "on"

Commands:*on()* Turn the switch on*off()* Turn the switch off**SmartApp Example:**

```

preferences {
    section("Title") {
        input "myswitch", "capability.switch", title: "switch", required: true, multiple: false
        input "motion", "capability.motionSensor", title: "motion", required: true, multiple: false
    }
}

def installed() {
    subscribe(motion, "motion", myHandler)
}

def myHandler(evt) {
    if("active" == evt.value && "on" != myswitch.currentSwitch) {
        myswitch.on()
    } else if ("inactive" == evt.value && "off" != myswitch.currentSwitch) {
        myswitch.off()
    }
}

```

10.35 Switch Level

Capability Name	SmartApp Preferences Reference
Switch Level	capability.switchLevel

Attributes:

Attribute	Type	Possible Values
level	Number	A number that represents the current light level, usually 0 – 100 in percent

Commands:

setLevel(number, number) Set the level to the given numbers

SmartApp Example:

```
preferences {
    section("Title") {
        input "myswitch", "capability.switchLevel", title: "switch", required: true, multiple: false
        input "motion", "capability.motionSensor", title: "motion", required: true, multiple: false
    }
}

def installed() {
    subscribe(motion, "motion", myHandler)
}

def myHandler(evt) {
    if("active" == evt.value && "on" != myswitch.currentSwitch) {
        myswitch.setLevel(90) // also turns on the switch
    } else if ("inactive" == evt.value && "off" != myswitch.currentSwitch) {
        myswitch.setLevel(10)
    }
}
```

10.36 Temperature Measurement

Capability Name	SmartApp Preferences Reference
Temperature Measurement	capability.temperatureMeasurement

Attributes:

Attribute	Type	Possible Values
temperature	Number	A number that usually represents the current temperature

Commands:

None.

SmartApp Example:


```

preferences {
    section("Cooling based on the following devices") {
        input "sensor", "capability.temperatureMeasurement", title: "Temp Sensor", required: true, multiple: false
        input "outlet", "capability.switch", title: "outlet", required: true, multiple: false
    }
    section("Set the desired temperature to cool to..."){
        input "setpoint", "decimal", title: "Set Temp"
    }
}

def installed() {
    subscribe(sensor, "temperature", myHandler)
}

def myHandler(evt) {
    if(evt.doubleValue > setpoint && "off" == outlet.currentSwitch) {
        outlet.on()
    } else if(evt.doubleValue < setpoint && "on" == outlet.currentSwitch) {
        outlet.off()
    }
}

```

10.37 Thermostat

Capability Name	SmartApp Preferences Reference
Thermostat	capability.thermostat

Attributes:

Attribute	Type	Possible Values
temperature		
heatingSetpoint		
coolingSetpoint		
thermostatSetpoint		
thermostatMode	String	"auto" "emergency heat" "heat" "off" "cool"
thermostatFan-Mode	String	"auto" "on" "circulate"
thermostatOperatingState	String	"heating" "idle" "pending cool" "vent economizer" "cooling" "pending heat" "fan only"

Commands:

setHeatingSetpoint(number)

setCoolingSetpoint(number)

off()

heat()

emergencyHeat()

cool()

setThermostatMode(enum)

fanOn()

fanAuto()

fanCirculate()

setThermostatFanMode(enum)

auto()

10.38 Thermostat Cooling Setpoint

Capability Name	SmartApp Preferences Reference
Thermostat Cooling Setpoint	capability.thermostatCoolingSetpoint

Attributes:

Attribute	Type	Possible Values
coolingSetpoint		

Commands:

setCoolingSetpoint(number)

10.39 Thermostat Fan Mode

Capability Name	SmartApp Preferences Reference
Thermostat Fan Mode	capability.thermostatFanMode

Attributes:

Attribute	Type	Possible Values
thermostatFanMode	String	"on" "auto" "circulate"

Commands:

fanOn()

fanAuto()

fanCirculate()

setThermostatFanMode(enum)

10.40 Thermostat Heating Setpoint

Capability Name	SmartApp Preferences Reference
Thermostat Heating Setpoint	capability.thermostatHeatingSetpoint

Attributes:

Attribute	Type	Possible Values
heatingSetpoint		

Commands:*setHeatingSetpoint(number)*

10.41 Thermostat Mode

Capability Name	SmartApp Preferences Reference
Thermostat Mode	capability.thermostatMode

Attributes:

Attribute	Type	Possible Values
thermostatMode	String	"emergency heat" "heat" "cool" "off" "auto"

Commands:*off()**heat()**emergencyHeat()**cool()**auto()**setThermostatMode(enum)*

10.42 Thermostat Operating State

Capability Name	SmartApp Preferences Reference
Thermostat Operating State	capability.thermostatOperatingState

Attributes:

Attribute	Type	Possible Values
thermostatOperatingState	String	"idle" "fan only" "vent economizer" "cooling" "pending heat" "heating" "pending cool"

Commands:

None.

10.43 Thermostat Setpoint

Capability Name	SmartApp Preferences Reference
Thermostat Setpoint	capability.thermostatSetpoint

Attributes:

Attribute	Type	Possible Values
thermostatSetpoint		

Commands:

None.

10.44 Three Axis

Capability Name	SmartApp Preferences Reference
Three Axis	capability.threeAxis

Attributes:

Attribute	Type	Possible Values
threeAxis		

Commands:

None.

10.45 Tone

Capability Name	SmartApp Preferences Reference
Tone	capability.tone

Attributes:

None.

Commands:

beep() Beep the device.

10.46 Touch Sensor

Capability Name	SmartApp Preferences Reference
Touch Sensor	capability.touchSensor

Attributes:

Attribute	Type	Possible Values
touch	String	"touched"

Commands:

None.

10.47 Valve

Capability Name	SmartApp Preferences Reference
Valve	capability.valve

Attributes:

Attribute	Type	Possible Values
contact	String	"closed" "open"

Commands:

Command	Parameters	Description
open()		
close()		

10.48 Water Sensor

Capability Name	SmartApp Preferences Reference
Water Sensor	capability.waterSensor

Attributes:

Attribute	Type	Possible Values
water	String	"dry" "wet"

Commands:

None.

API Documentation

This is where you can find API-level documentation for the various objects available in your SmartApps and Device Handlers.

How to Read The Docs

Objects

SmartThings objects are rarely created directly by SmartApp or Device Handler developers. Instead, various objects are already created and available in your applications.

You will rarely see constructor documentation for this reason. Each object will contain a summary at the top of the document that discusses some of the common ways to get a reference to the object.

Also worth noting is that some of the “objects” documented are not really objects at all. A SmartApp is not an object, in the strict sense of the word, for example (either is a Device Handler). But each running execution of a SmartApp or Device Handler has available to it many methods and properties. For convenience, we have organized the methods available to SmartApps and Device Handlers into the SmartApp or Device Handler API documentation.

Object Wrappers

You may notice in various log messages or error messages that the objects are actually wrapper objects. For example, `Event` is actually an instance of `EventWrapper`. We have wrapped many of our core objects with wrapper objects to protect access to the underlying system object.

This should be transparent to developers, since these objects cannot be instantiated directly. The underlying wrapper class may even change at some point (the supported APIs should not, without notice).

But, should you be confused about messages in the log, this is why.

Dynamic Methods

The Groovy programming language offers a powerful feature called [Metaprogramming](#) that (among other things) allows for Groovy programs to be written in a way that *methods can be created dynamically at run time*.

SmartThings makes use of this powerful feature in a few ways. For example, you can get a reference to a Device configured in a SmartApp preference by simply referencing the name of the device configured in the preference. Another example is getting various Attribute values for a Device by invoking a method in the form `<someDevice>.current<AttributeName>`.

This powerful feature can make documenting all available methods difficult, since methods may not exist until runtime. For any dynamic methods, the method or property will be enclosed in `<>`, and a description and example will be given.

Conventions

All methods and properties are listed in alphabetical order, with the exception of SmartApp and Device Handler methods that are expected to be defined by SmartApps and Device Handlers - those will be listed first.

Methods are listed with a `()` after the name. Properties do not have a `()`. For example: `subscribe()` is a method, `floatValue` is a property.

Note: Groovy follows the JavaBean convention, and adds some syntactic sugar on top. Any zero-arg getter can be retrieved via property access directly. For example, `isPhysical()` *could* be invoked as `physical`. Because this relies upon details of the backing internal object, we recommend that you use the documented version.

Some methods may have many signatures. For example, the `schedule` method available to SmartApps can be called with a variety of arguments. We have documented all forms in one location (`schedule()`). All supported signatures will be listed, as well as all parameters for the various signatures.

Optional parameters will be listed inside brackets `[]` in the method signature.

Code examples may not be executable as-is. Since SmartApps and Device Handlers execute in response to various schedules or events, and rely upon having other metadata defined, the examples have been written with brevity in mind. The code samples may need to be defined inside an event handler or otherwise executable code block to fully function.

When appropriate, we have included various tips or warnings. In cases where an API is not adequately documented currently, we have called attention to that. We plan to add the supporting documentation soon!

API Contents

11.1 SmartApp

A SmartApp is a Groovy-based program that allows developers to create automations for users to tap into the capabilities of their devices.

They are created through the “New SmartApp” action in the IDE. There is no “class” for a SmartApp per se, but there are various methods and properties available to SmartApps that are documented below.

When a SmartApp executes, it executes in the context of a certain installation instance. That is, a user installs a SmartApp on their mobile application, and configures it with devices or rules unique to them. A SmartApp is not continuously running; it is executed in response to various schedules or subscribed-to events.

The following methods should be defined by all SmartApps. They are called by the SmartThings platform at various points in the SmartApp lifecycle.

11.1.1 installed()

Note: This method is expected to be defined by SmartApps.

Called when an instance of the app is installed. Typically subscribes to events from the configured devices and creates any scheduled jobs.

Signature: `void installed()`

Returns: `void`

Example:

```
def installed() {
    log.debug "installed with settings: $settings"

    // subscribe to events, create scheduled jobs.
}
```


11.1.2 updated()

Note: This method is expected to be defined by SmartApps.

Called when the preferences of an installed app are updated. Typically unsubscribes and re-subscribes to events from the configured devices and unschedules/reschedules jobs.

Signature: void uninstalled()

Returns: void

Example:

```
def updated() {
    unsubscribe()
    // resubscribe to device events, create scheduled jobs
}
```

11.1.3 uninstalled()

Note: This method may be defined by SmartApps.

Called, if declared, when an app is uninstalled. Does not need to be declared unless you have some external cleanup to do. subscriptions and scheduled jobs are automatically removed when an app is uninstalled, so you don't need to do that here.

Signature: void uninstalled()

Returns: void

Example:

```
def uninstalled() {
    // external cleanup. No need to unsubscribe or remove scheduled jobs
}
```

The following methods and attributes are available to call in a SmartApp:

11.1.4 <device or capability preference name>

A reference to the device or devices selected during app installation or update.

Returns: *Device* (page 304) or a list of Devices - the Device with the given preference name, or a list of Devices if `multiple:true` is specified in the preferences.

Example:

```
preferences {
    ...
    input "theswitch", "capability.switch"
    input "theswitches", "capability.switch", multiple:true
    ...
}

...
// the name of the preference becomes the reference for the Device object
theswitch.on()
theswitch.off()

// multiple:true means we get a list of devices
theswitches.each {log.debug "Current switch value: ${it.currentSwitch}"

// we can still call methods directly on the list; it will apply the method to each device:
theswitches.on() // turn all switches on
```

11.1.5 <number or decimal preference name>

A reference to the value entered for a number or decimal input preference.

Returns: `BigDecimal` - the value entered for a number or decimal input preference.

Example:

```
preferences {
    ...
    input "num1", "number"
    input "dec1", "decimal"
    ...
}

...
// preference name is a reference to a BigDecimal that is the value the user entered.
log.debug "num1: $num1" //=> value user entered for num1 preference
log.debug "dec1: $dec1" //=> value user entered for dec1 preference
...
```

11.1.6 <text, mode, or time preference name>

A reference to the value entered for a text, mode, or time input type.

The following table explains the value and format returned for the various input types:

Input Type	Return Value
text	<code>String</code> - the value entered as text
mode	<code>String</code> - the name of the mode selected
time	<code>String</code> - the full date string in the format of “yyyy-MM-dd’T’HH:mm:ss.SSSZ”

Example:

```

preferences {
    ...
    input "mytext", "text"
    input "mymode", "mode"
    input "mytime", "time"
    ...
}

log.debug "mytext: $mytext"
log.debug "mymode: $mymode"
log.debug "mytime: $mytime"

// time is in format compatible with most scheduling APIs.
// we can pass the value directly to the APIs that accept a date string:
runOnce(mytime, someHandlerMethod)
schedule(myTime, someHandlerMethod)

```

11.1.7 addChildDevice()

Adds a child device to a SmartApp. An example use is in service manager SmartApps.

Signature: DeviceWrapper addChildDevice(String namespace, String typeName, String deviceNetworkId, hubId, Map properties)

Throws: UnknownDeviceTypeException

Parameters: String namespace - the namespace for the device. Defaults to installedSmartApp.smartAppVersionDTO.smartAppDTO.namespace

String typeName - the device type name

String deviceNetworkId - the device network id of the device

hubId - (optional) The hub id. Defaults to null

Map properties (optional) - A map with device properties.

Returns: DeviceWrapper - The device that was created.

11.1.8 apiServerUrl()

Returns the URL of the server where this SmartApp can be reached for API calls, along with the specified path appended to it. Use this instead of hard-coding a URL to ensure that the correct server URL for this installed instance is returned.

Signature: String apiServerUrl(String path)

Parameters: String path - the path to append to the URL

Returns: The URL of the server for this installed instance of the SmartApp.

Example:

```
// logs <server url>/my/path
log.debug "apiServerUrl: ${apiServerUrl("/my/path")}"

// The leading "/" will be added if you don't specify it
// logs <server url>/my/path
log.debug "apiServerUrl: ${apiServerUrl("my/path")}"
```

11.1.9 atomicState

A map of name/value pairs that SmartApp can use to save and retrieve data across SmartApp executions. This is similar to *state* (page 271), but will immediately write and read from the backing data store. Prefer using *state* over *atomicState* when possible.

Signature: Map atomicState

Returns: Map - a map of name/value pairs.

```
atomicState.count = 0
atomicState.count = atomicState.count + 1

log.debug "atomicState.count: ${atomicState.count}"

// use array notation if you wish
log.debug "atomicState['count']: ${atomicState['count']}"

// you can store lists and maps to make more interesting structures
atomicState.listOfMaps = [[key1: "val1", bool1: true],
                          [otherKey: ["string1", "string2"]]]
```

11.1.10 canSchedule()

Returns true if the SmartApp is able to schedule jobs. Currently SmartApps are limited to 4 scheduled jobs. That limit includes operations such as *runIn* and *runOnce*.

Signature: Boolean canSchedule()

Returns: Boolean - true if additional jobs can be scheduled, false otherwise.

Example:

```
log.debug "Can schedule? ${canSchedule()}"
```

11.1.11 deleteChildDevice()

Deletes the child device with the specified device network id.

Signature: void deleteChildDevice(String deviceNetworkId)

Throws: NotFoundException

Parameters: String deviceNetworkId - the device network id of the device

Returns: void

11.1.12 getAllChildDevices()

Returns a list of all child devices, including virtual devices. This is a wrapper for `getChildDevices(true)`.

Signature: `List getChildDevices()`

Returns: `List` - a list of all child devices.

11.1.13 getApiServerUrl()

Returns the URL of the server where this SmartApp can be reached for API calls. Use this instead of hard-coding a URL to ensure that the correct server URL for this installed instance is returned.

Signature: `String getApiServerUrl()`

Returns: `String` - the URL of the server where this SmartApp can be reached.

11.1.14 getChildDevice()

Returns a device based upon the specified device network id. This is mostly used in service manager SmartApps.

Signature: `DeviceWrapper getChildDevice(String deviceNetworkId)`

Parameters: `String deviceNetworkId` - the device network id of the device

Returns: `DeviceWrapper` - The device found with the given device network ID.

11.1.15 getChildDevices()

Returns a list of all child devices. An example use would be in service manager SmartApps.

Signature: `List getChildDevices(Boolean includeVirtualDevices)`

Parameters: `Boolean true` if the returned list should contain virtual devices. Defaults to *false*. (*optional*)

Returns: `List` - A list of all devices found.

11.1.16 getSunriseAndSunset()

Gets a map containing the local sunrise and sunset times.

Signature: Map getSunriseAndSunset([Map options])

Parameters:

Map options (*optional*)

The supported options are:

Option	Description
zipCode	<p>String - the zip code to use for determining the times.</p> <p>If not specified then the coordinates of the hub location are used.</p>
locationString	<p>String - any location string supported by the Weather Underground APIs.</p> <p>If not specified then the coordinates of the hub location are used</p>
sunriseOffset	<p>String - adjust the sunrise time by this amount.</p> <p>See <i>timeOffset()</i> (page 274) for supported formats</p>
sunsetOffset	<p>String - adjust the sunset time by this amount.</p> <p>See <i>timeOffset()</i> (page 274) for supported formats</p>

Returns: Map - A Map containing the local sunrise and sunset times as **Date** objects: [sunrise: Date, sunset: Date]

Example:

```
def noParams = getSunriseAndSunset()
def beverlyHills = getSunriseAndSunset(zipCode: "90210")
def thirtyMinsBeforeSunset = getSunriseAndSunset(sunsetOffset: "-00:30")

log.debug "sunrise with no parameters: ${noParams.sunrise}"
log.debug "sunset with no parameters: ${noParams.sunset}"
log.debug "sunrise and sunset in 90210: $beverlyHills"
log.debug "thirty minutes before sunset at current location: ${thirtyMinsBeforeSunset.sunset}"
```

11.1.17 getWeatherFeature()

Calls the Weather Underground API to to return weather forecasts and related data.

Signature: Map getWeatherFeature(String featureName [, String location])

Note: `getWeatherFeature` simply delegates to the Weather Underground API, using the specified `featureName` and `location` (if specified). For full descriptions on the available features and return information, please consult the [Weather Underground API docs](#).

Parameters: `String featureName` The weather feature to get. This corresponds to the available “Data Features” in the Weather Underground API.

`String location` (*optional*) The location to get the weather information for (ZIP code). If not specified, the location of the user’s hub will be used.

Returns: `Map` - a Map containing the weather information requested. The data returned will vary depending on the feature requested. See the Weather Underground API documentation for more information.

11.1.18 `httpDelete()`

Executes an HTTP DELETE request and passes control to the specified closure. The closure is passed one `HttpResponseDecorator` argument from which the response content and header information can be extracted.

Signature: `void httpDelete(String uri, Closure closure)`

`void httpDelete(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP DELETE call to.

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
<code>uri</code>	Either a URI or URL of of the endpoint to make a request from.
<code>path</code>	Request path that is merged with the URI.
<code>query</code>	Map of URL query parameters.
<code>headers</code>	Map of HTTP headers.
<code>contentType</code>	Forced response content type and request Accept header.
<code>requestContent-Type</code>	Content type for the request, if it is different from the expected response content-type.
<code>body</code>	Request body that will be encoded based on the given <code>contentType</code> .

`Closure closure` - The closure that will be called with the response of the request.

Returns: `void`

11.1.19 `httpGet()`

Executes an HTTP DELETE request and passes control to the specified closure. The closure is passed one `HttpResponseDecorator` argument from which the response content and header information can be extracted.

If the response content type is JSON, the response data will automatically be parsed into a data structure.

Signature: `void httpGet(String uri, Closure closure)`

`void httpGet(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP GET call to

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
<code>uri</code>	Either a URI or URL of of the endpoint to make a request from.
<code>path</code>	Request path that is merged with the URI.
<code>query</code>	Map of URL query parameters.
<code>headers</code>	Map of HTTP headers.
<code>contentType</code>	Forced response content type and request Accept header.
<code>requestContent-Type</code>	Content type for the request, if it is different from the expected response content-type.
<code>body</code>	Request body that will be encoded based on the given <code>contentType</code> .

`Closure closure` - The closure that will be called with the response of the request.

Example:

```
def params = [
    uri: "http://httpbin.org",
    path: "/get"
]

try {
    httpGet(params) { resp ->
        resp.headers.each {
            log.debug "${it.name} : ${it.value}"
        }
        log.debug "response contentType: ${resp.contentType}"
        log.debug "response data: ${resp.data}"
    }
} catch (e) {
    log.error "something went wrong: $e"
}
```

11.1.20 httpHead()

Executes an HTTP HEAD request and passes control to the specified closure. The closure is passed one `HttpResponseDecorator` argument from which the response content and header information can be extracted.

Signature: `void httpHead(String uri, Closure closure)`

`void httpHead(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP HEAD call to

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
<code>uri</code>	Either a URI or URL of of the endpoint to make a request from.
<code>path</code>	Request path that is merged with the URI.
<code>query</code>	Map of URL query parameters.
<code>headers</code>	Map of HTTP headers.
<code>contentType</code>	Forced response content type and request Accept header.
<code>requestContent-Type</code>	Content type for the request, if it is different from the expected response content-type.
<code>body</code>	Request body that will be encoded based on the given <code>contentType</code> .

`Closure closure` - The closure that will be called with the response of the request.

11.1.21 httpPost()

Executes an HTTP POST request and passes control to the specified closure. The closure is passed one [HttpResponseDecorator](#) argument from which the response content and header information can be extracted.

If the response content type is JSON, the response data will automatically be parsed into a data structure.

Signature: `void httpPost(String uri, String body, Closure closure)`

`void httpPost(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP GET call to

`String body` - The body of the request

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
<code>uri</code>	Either a URI or URL of the endpoint to make a request from.
<code>path</code>	Request path that is merged with the URI.
<code>query</code>	Map of URL query parameters.
<code>headers</code>	Map of HTTP headers.
<code>contentType</code>	Forced response content type and request Accept header.
<code>requestContent-Type</code>	Content type for the request, if it is different from the expected response content-type.
<code>body</code>	Request body that will be encoded based on the given <code>contentType</code> .

`Closure closure` - The closure that will be called with the response of the request.

Example:

```
try {
    httpPost("http://mysite.com/api/call", "id=XXX&value=YYY") { resp ->
        log.debug "response data: ${resp.data}"
        log.debug "response contentType: ${resp.contentType}"
    }
} catch (e) {
    log.debug "something went wrong: $e"
}
```

11.1.22 httpPostJson()

Executes an HTTP POST request with a JSON-encoded body and content type, and passes control to the specified closure. The closure is passed one [HttpResponseDecorator](#) argument from which the response content and header information can be extracted.

If the response content type is JSON, the response data will automatically be parsed into a data structure.

Signature: `void httpPostJson(String uri, String body, Closure closure)`

`void httpPostJson(String uri, Map body, Closure closure)`

`void httpPostJson(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP POST call to

`String body` - The body of the request

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
uri	Either a URI or URL of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
contentType	Forced response content type and request Accept header.
requestContent-Type	Content type for the request, if it is different from the expected response content-type.
body	Request body that will be encoded based on the given contentType.

`Closure closure` - The closure that will be called with the response of the request.

Example:

```
def params = [
    uri: "http://postcatcher.in/catchers/<yourUniquePath>",
    body: [
        param1: [subparam1: "subparam 1 value",
                  subparam2: "subparam2 value"],
        param2: "param2 value"
    ]
]

try {
    httpPostJson(params) { resp ->
        resp.headers.each {
            log.debug "${it.name} : ${it.value}"
        }
        log.debug "response contentType: ${resp.contentType}"
    }
} catch (e) {
    log.debug "something went wrong: $e"
}
```

11.1.23 httpPut()

Executes an HTTP PUT request and passes control to the specified closure. The closure is passed one `HttpResponseDecorator` argument from which the response content and header information can be extracted.

If the response content type is JSON, the response data will automatically be parsed into a data structure.

Signature: `void httpPut(String uri, String body, Closure closure)`

`void httpPut(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP GET call to

`String body` - The body of the request

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
uri	Either a URI or URL of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
contentType	Forced response content type and request Accept header.
requestContent-Type	Content type for the request, if it is different from the expected response content-type.
body	Request body that will be encoded based on the given contentType.

Closure *closure* - The closure that will be called with the response of the request.

Example:

```
try {
    httpPut("http://mysite.com/api/call", "id=XXX&value=YYY") { resp ->
        log.debug "response data: ${resp.data}"
        log.debug "response contentType: ${resp.contentType}"
    }
} catch (e) {
    log.error "something went wrong: $e"
}
```

11.1.24 httpPutJson()

Executes an HTTP PUT request with a JSON-encoded body and content type, and passes control to the specified closure. The closure is passed one [HttpResponseDecorator](#) argument from which the response content and header information can be extracted.

If the response content type is JSON, the response data will automatically be parsed into a data structure.

Signature: `void httpPutJson(String uri, String body, Closure closure)`

`void httpPutJson(String uri, Map body, Closure closure)`

`void httpPutJson(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP PUT call to

`String body` - The body of the request

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
uri	Either a URI or URL of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
contentType	Forced response content type and request Accept header.
requestContent-Type	Content type for the request, if it is different from the expected response content-type.
body	Request body that will be encoded based on the given contentType.

Closure *closure* - The closure that will be called with the response of the request.

11.1.25 location

The *Location* (page 325) into which this SmartApp has been installed.

Signature: `Location location`

Returns: *Location* (page 325) - The Location into which this SmartApp has been installed.

11.1.26 now()

Gets the current Unix time in milliseconds.

Signature: `Long now()`

Returns: *Long* - the current Unix time.

11.1.27 parseJson()

Parses the specified string into a JSON data structure.

Signature: `Map parseJson(stringToParse)`

Parameters: *String* `stringToParse` - The string to parse into JSON

Returns: *Map* - a map that represents the passed-in string in JSON format.

11.1.28 parseXml()

Parses the specified string into an XML data structure.

Signature: `GPathResult parseXml(stringToParse)`

Parameters: *String* `stringToParse` - The string to parse into XML

Returns: *GPathResult* - A *GPathResult* instance that represents the passed-in string in XML format.

11.1.29 parseLanMessage()

Parses a Base64-encoded LAN message received from the hub into a map with header and body elements, as well as parsing the body into an XML document.

Signature: `Map parseLanMessage(stringToParse)`

Parameters: *String* `stringToParse` - The string to parse

Returns: *Map* - a map with the following structure:

key	type	description
header	<i>String</i>	the headers of the request as a single string
headers	<i>Map</i>	a Map of string/name value pairs for each header
body	<i>String</i>	the request body as a string

11.1.30 parseSoapMessage()

Parses a Base64-encoded LAN message received from the hub into a map with header and body elements, as well as parsing the body into an XML document. This method is commonly used to parse [UPNP SOAP](#) messages.

Signature: `Map parseLanMessage(stringToParse)`

Parameters: `String stringToParse` - The string to parse

Returns: `Map` - A map with the following structure:

key	type	description
header	<code>String</code>	the headers of the request as a single string
headers	<code>Map</code>	a Map of string/name value pairs for each header
body	<code>String</code>	the request body as a string
xml	<code>GPathResult</code>	the request body as a <code>GPathResult</code> object
xmlError	<code>String</code>	error message from parsing the body, if any

11.1.31 runIn()

Executes a specified `handlerMethod` after `delaySeconds` have elapsed.

Signature: `void runIn(delayInSeconds, handlerMethod [, options])`

Tip: It's important to note that we will attempt to run this method at this time, but cannot guarantee exact precision. We typically expect per-minute level granularity, so if using with values less than sixty seconds, your mileage will vary.

Parameters: `delayInSeconds` - The number of seconds to execute the `handlerMethod` after.

`handlerMethod` - The method to call after `delayInSeconds` has passed. Can be a string or a reference to the method.

`options` (*optional*) - A map of parameters. Currently only the value `[overwrite: true/false]` is supported. Normally, if within the time window between calling `runIn()` and the `handlerMethod` being called, if you call `runIn(300, 'handlerMethod')` method again we will stop the original schedule and just use the new one. In this case there is at most one schedule for the *handlerMethod*. However, if you were to call `runIn(300, 'handlerMethod', [overwrite: false])`, then we let the original schedule continue and also add a new one for another 5 minutes out. This could lead to many different schedules. If you are going to use this, be sure to handle multiple calls to the 'handlerMethod' method.

Returns: `void`

Example:

```
runIn(300, myHandlerMethod)
runIn(400, "myOtherHandlerMethod")

def myHandlerMethod() {
    log.debug "handler method called"
}

def myOtherHandlerMethod() {
    log.debug "other handler method called"
}
```

11.1.32 runEvery5Minutes()

Creates a recurring schedule that executes the specified `handlerMethod` every five minutes. Using this method will pick a random start time in the next five minutes, and run every five minutes after that.

Signature: `void runEvery5Minutes(handlerMethod)`

Tip: This is preferred over using `schedule(cronExpression, handlerMethod)` for a regular schedule like this because with a cron expression all installations of a SmartApp will execute at the same time. With this method, the executions will be spread out over the 5 minute period.

Parameters: `handlerMethod` - The method to call every five minutes. Can be the name of the method as a string, or a reference to the method.

Returns: `void`

Example:

```
runEvery5Minutes(handlerMethod1)
runEvery5Minutes(handlerMethod2)

def handlerMethod1() {
    log.debug "handlerMethod1"
}

def handlerMethod2() {
    log.debug "handlerMethod2"
}
```

11.1.33 runEvery10Minutes()

Creates a recurring schedule that executes the specified `handlerMethod` every ten minutes. Using this method will pick a random start time in the next ten minutes, and run every ten minutes after that.

Signature: `void runEvery10Minutes(handlerMethod)`

Tip: This is preferred over using `schedule(cronExpression, handlerMethod)` for a regular schedule like this because with a cron expression all installations of a SmartApp will execute at the same time. With this method, the executions will be spread out over the ten minute period.

Parameters: `handlerMethod` - The method to call every ten minutes. Can be the name of the method as a string, or a reference to the method.

Returns: `void`

Example:

```
runEvery10Minutes(handlerMethod1)
runEvery10Minutes(handlerMethod2)

def handlerMethod1() {
    log.debug "handlerMethod1"
}

def handlerMethod2() {
```

```
log.debug "handlerMethod2"  
}
```

11.1.34 runEvery15Minutes()

Creates a recurring schedule that executes the specified `handlerMethod` every fifteen minutes. Using this method will pick a random start time in the next five minutes, and run every five minutes after that.

Signature: `void runEvery15Minutes(handlerMethod)`

Tip: This is preferred over using `schedule(cronExpression, handlerMethod)` for a regular schedule like this because with a cron expression all installations of a SmartApp will execute at the same time. With this method, the executions will be spread out over the fifteen minute period.

Parameters: `handlerMethod` - The method to call every fifteen minutes. Can be the name of the method as a string, or a reference to the method.

Returns: `void`

Example:

```
runEvery15Minutes(handlerMethod1)  
runEvery15Minutes(handlerMethod2)  
  
def handlerMethod1() {  
    log.debug "handlerMethod1"  
}  
  
def handlerMethod2() {  
    log.debug "handlerMethod2"  
}
```

11.1.35 runEvery30Minutes()

Creates a recurring schedule that executes the specified `handlerMethod` every thirty minutes. Using this method will pick a random start time in the next thirty minutes, and run every thirty minutes after that.

Signature: `void runEvery30Minutes(handlerMethod)`

Tip: This is preferred over using `schedule(cronExpression, handlerMethod)` for a regular schedule like this because with a cron expression all installations of a SmartApp will execute at the same time. With this method, the executions will be spread out over the thirty minute period.

Parameters: `handlerMethod` - The method to call every thirty minutes. Can be the name of the method as a string, or a reference to the method.

Returns: `void`

Example:

```
runEvery30Minutes(handlerMethod1)
runEvery30Minutes(handlerMethod2)

def handlerMethod1() {
    log.debug "handlerMethod1"
}

def handlerMethod2() {
    log.debug "handlerMethod2"
}
```

11.1.36 runEvery1Hour()

Creates a recurring schedule that executes the specified `handlerMethod` every hour. Using this method will pick a random start time in the next hour, and run every hour after that.

Signature: `void runEvery1Hour(handlerMethod)`

Tip: This is preferred over using `schedule(cronExpression, handlerMethod)` for a regular schedule like this because with a cron expression all installations of a SmartApp will execute at the same time. With this method, the executions will be spread out over the one hour period.

Parameters: `handlerMethod`- The method to call every hour. Can be the name of the method as a string, or a reference to the method.

Returns: `void`

Example:

```
runEvery1Hour(handlerMethod1)
runEvery1Hour(handlerMethod2)

def handlerMethod1() {
    log.debug "handlerMethod1"
}

def handlerMethod2() {
    log.debug "handlerMethod2"
}
```

11.1.37 runEvery3Hours()

Creates a recurring schedule that executes the specified `handlerMethod` every three hours. Using this method will pick a random start time in the next hour, and run every three hours after that.

Signature: `void runEvery3Hours(handlerMethod)`

Tip: This is preferred over using `schedule(cronExpression, handlerMethod)` for a regular schedule like this because with a cron expression all installations of a SmartApp will execute at the same time. With this method, the executions will be spread out over the three hour period.

Parameters: `handlerMethod` - The method to call every three hours. Can be the name of the method as a string, or a reference to the method.

Returns: `void`

Example:

```
runEvery3Hours(handlerMethod1)
runEvery3Hours(handlerMethod2)

def handlerMethod1() {
    log.debug "handlerMethod1"
}

def handlerMethod2() {
    log.debug "handlerMethod2"
}
```

11.1.38 runOnce()

Executes the `handlerMethod` once at the specified date and time.

Signature: `void runOnce(dateTime, handlerMethod)`

Parameters: `dateTime` - When to execute the `handlerMethod`. Can be either a [Date](#) object or an ISO-8601 date string. For example, `new Date() + 1` would run at the current time tomorrow, and `"2017-07-04T12:00:00.000Z"` would run at noon GMT on July 4th, 2017.

`handlerMethod` - The method to execute at the specified `dateTime`. This can be a reference to the method, or the method name as a string.

Returns: `void`

Example:

```
// execute handler at 4 PM CST on October 21, 2015 (e.g., Back to the Future 2 Day!)
runOnce("2015-10-21T16:00:00.000-0600", handler)

def handler() {
    ...
}
```

11.1.39 schedule()

Creates a scheduled job that calls the `handlerMethod` once per day at the time specified, or according to a cron schedule.

Signature: `void schedule(dateTime, handlerMethod)`

`void schedule(cronExpression, handlerMethod)`

Parameters:

`dateTime` - A [Date](#) object, an ISO-8601 formatted date time string.

[String](#) `cronExpression` - A cron expression that specifies the schedule to execute on.

`handlerMethod` - The method to call. This can be a reference to the method itself, or the method name as a string.

Returns: void

Tip: Since calling `schedule()` with a `dateTime` argument creates a recurring scheduled job to execute *every day* at the specified time, the *date information is ignored. Only the time portion of the argument is used.*

Tip: Full documentation for the cron expression format can be found in the [Quartz Cron Trigger Tutorial](#)

Example:

```
preferences {
    section() {
        input "timeToRun", "time"
    }
}

...
// call handlerMethod1 at time specified by user input
schedule(timeToRun, handlerMethod1)

// call handlerMethod2 every day at 3:36 PM CST
schedule("2015-01-09T15:36:00.000-0600", handlerMethod2)

// execute handlerMethod3 every hour on the half hour
schedule("0 30 & & & ?", handlerMethod3)
...

def handlerMethod1() {...}
def handlerMethod2() {...}
def handlerMethod3() {...}
```

11.1.40 `sendEvent()`

Creates and sends an event constructed from the specified properties. If a device is specified, then a DEVICE event will be created, otherwise an APP event will be created.

Note: SmartApps typically *respond to events*, not create them. In more rare cases, certain SmartApps or Service Manager SmartApps may have reason to send events themselves. `sendEvent` can be used for those cases.

Signature: void `sendEvent`(Map properties)

void `sendEvent`(Device device, Map properties)

Parameters: [Map](#) properties - The properties of the event to create and send.

Here are the available properties:

Property	Description
name (required)	String - The name of the event. Typically corresponds to an attribute name of a capability.
value (required)	The value of the event. The value is stored as a string, but you can pass numbers or other objects.
descriptionText	String - The description of this event. This appears in the mobile application activity for the device. If not specified, this will be created using the event name and value.
displayed	Pass <code>true</code> to display this event in the mobile application activity feed, <code>false</code> to not display. Defaults to <code>true</code> .
linkText	String - Name of the event to show in the mobile application activity feed.
isStateChange	<code>true</code> if this event caused a device attribute to change state. Typically not used, since it will be set automatically.
unit	String - a unit string, if desired. This will be used to create the <code>descriptionText</code> if it (the <code>descriptionText</code> option) is not specified.

Device (page 304) device - The device for which this event is created for.

Tip: Not all event properties need to be specified. ID properties like `deviceId` and `locationId` are automatically set, as are properties like `isStateChange`, `displayed`, and `linkText`.

Returns: void

Example:

```
sendEvent(name: "temperature", value: 72, unit: "F")
```

11.1.41 sendLocationEvent()

Sends a LOCATION event constructed from the specified properties. See the [Event](#) (page 315) reference for a list of available properties. Other SmartApps can receive location events by subscribing to the location. Examples of existing location events include sunrise and sunset.

Signature: void sendLocationEvent(Map properties)

Parameters: [Map](#) properties - The properties from which to create and send the event.

Here are the available properties:

Property	Description
name (required)	String - The name of the event. Typically corresponds to an attribute name of a capability.
value (required)	The value of the event. The value is stored as a string, but you can pass numbers or other objects.
descriptionText	String - The description of this event. This appears in the mobile application activity for the device. If not specified, this will be created using the event name and value.
displayed	Pass <code>true</code> to display this event in the mobile application activity feed, <code>false</code> to not display. Defaults to <code>true</code> .
linkText	String - Name of the event to show in the mobile application activity feed.
isStateChange	<code>true</code> if this event caused a device attribute to change state. Typically not used, since it will be set automatically.
unit	String - a unit string, if desired. This will be used to create the <code>descriptionText</code> if it (the <code>descriptionText</code> option) is not specified.

Returns: void

11.1.42 sendNotification()

Sends the specified message and displays it in the *Hello, Home* portion of the mobile application.

Signature: void sendNotification(String message [, Map options])

Parameters: [String](#) message - The message to send to *Hello, Home*

[Map](#) options (optional) - Options for the message. The following options are available:

option	description
method	String - One of "phone", "push", or "both". Defaults to "both".
event	<code>false</code> to suppress displaying in <i>Hello, Home</i> . Defaults to <code>true</code> .
phone	String - The phone number to send the SMS message to. Required when the method is "phone". If not specified and method is "both", then no SMS message will be sent.

Returns: void

Example:

```
sendNotification("test notification - no params")
sendNotification("test notification - push", [method: "push"])
sendNotification("test notification - sms", [method: "phone", phone: "1234567890"])
sendNotification("test notification - both", [method: "both", phone: "1234567890"])
sendNotification("test notification - no event", [event: false])
```

11.1.43 sendNotificationEvent()

Displays a message in *Hello, Home*, but does not send a push notification or SMS message.

Signature: void sendNotificationEvent(String message)

Parameters: [String](#) message - The message to send to *Hello, Home*

Returns: void

Example:

```
sendNotificationEvent("some message")
```

11.1.44 sendNotificationToContacts()

Sends the specified message to the specified contacts.

Signature: void sendNotificationToContacts(String message, String contact, Map options=[:])
 void sendNotificationToContacts(String message, Collection contacts, Map options=[:])

Parameters: String message - the message to send

String contact - the contact to send the notification to. Typically set through the contacts input type.

‘Collection’ contacts - the collection of contacts to send the notification to. Typically set through the contacts input type.

Map options (*optional*) - a map of additional parameters. The valid parameter is [event: boolean] to specify if the message should be displayed in the Notifications feed. Defaults to true (message will be displayed in the Notifications feed).

Returns: void

Example:

```
preferences {
    section("Send Notifications?") {
        input("recipients", "contact", title: "Send notifications to") {
            input "phone", "phone", title: "Warn with text message (optional)",
                description: "Phone Number", required: false
        }
    }
}

...
if (location.contactBookEnabled) {
    sendNotificationToContacts("Your house talks!", recipients)
}
...
```

Tip: It's a good idea to assume that a user *may not* have any contacts configured. That's why you see the nested "phone" input in the preferences (user will only see that if they don't have contacts), and why we check location.contactBookEnabled.

11.1.45 sendPush()

Sends the specified message as a push notification to users mobile devices and displays it in *Hello, Home*.

Signature: void sendPush(String message)

Parameters: String message - The message to send

Returns: void

Example:

```
sendPush("some message")
```

11.1.46 sendPushMessage()

Sends the specified message as a push notification to users mobile devices but does not display it in *Hello, Home*.

Signature: void sendPushMessage(String message)

Parameters: String message - The message to send

Returns: void

Example:

```
sendPushMessage("some message")
```

11.1.47 sendSms()

Sends the message as an SMS message to the specified phone number and displays it in Hello, Home. The message can be no longer than 140 characters.

Signature: void sendSms(String phoneNumber, String message)

Parameters: String phoneNumber - the phone number to send the SMS message to.

String message - the message to send. Can be no longer than 140 characters.

Returns: void

Example:

```
sendSms("somePhoneNumber", "some message")
```

11.1.48 sendSmsMessage()

Sends the message as an SMS message to the specified phone number but does not display it in Hello, Home. The message can be no longer than 140 characters.

Signature: void sendSmsMessage(String phoneNumber, String message)

Parameters: String phoneNumber - the phone number to send the SMS message to.

String message - the message to send. Can be no longer than 140 characters.

Returns: void

Example:

```
sendSms("somePhoneNumber", "some message")
```

11.1.49 settings

A map of name/value pairs containing all of the installed SmartApp's preferences.

Signature: Map settings

Returns: Map - a map containing all of the installed SmartApp's preferences.

Example:

```
preferences {
    section() {
        input "myswitch", "capability.switch"
        input "mytext", "text"
        input "mytime", "time"
    }
}

...

log.debug "settings.mytext: ${settings.mytext}"
log.debug "settings.mytime: ${settings.mytime}"

// if the input is a device/capability, you can get the device object
// through the settings:
log.debug "settings.myswitch.currentSwitch: ${settings.myswitch.currentSwitch}"
...
```

11.1.50 state

A map of name/value pairs that SmartApps can use to save and retrieve data across SmartApp executions.

Signature: Map state

Returns: Map - a map of name/value pairs.

```
state.count = 0
state.count = state.count + 1

log.debug "state.count: ${state.count}"

// use array notation if you wish
log.debug "state['count']: ${state['count']}"

// you can store lists and maps to make more interesting structures
state.listOfMaps = [[key1: "val1", bool1: true],
                    [otherKey: ["string1", "string2"]]]
```

Warning: Though state can be treated as a map in most regards, certain convenience operations that you may be accustomed to in maps will not work with state. For example, `state.count++` will not increment the count - use the longer form of `state.count = state.count + 1`.

11.1.51 stringToMap()

Parses a comma-delimited string into a map.

Signature: Map stringToMap(String string)

Parameters: String string - A comma-delimited string to parse into a map.

Returns: Map - a map created from the comma-delimited string.

Example:

```
def testStr = "key1: value1, key2: value2"
def testMap = stringToMap(testStr)

log.debug "stringToMap: ${testMap}"
log.debug "stringToMap.key1: ${testMap.key1}" // => value1
log.debug "stringToMap.key2: ${testMap.key2}" // => value2
```

11.1.52 subscribe()

Subscribes to the various events for a device or location. The specified handlerMethod will be called when the event is fired.

All event handler methods will be passed an *Event* (page 315) that represents the event causing the handler method to be called.

Signature: void subscribe(deviceOrDevices, String attributeName, handlerMethod)

```
void subscribe(deviceOrDevices, String attributeNameAndValue,
handlerMethod)
```

```
void subscribe(Location location, handlerMethod)
```

```
void subscribe(app, handlerMethod)
```

Parameters: deviceOrDevices - The *Device* (page 304) or list of devices to subscribe to.

String attributeName - The attribute to subscribe to.

String attributeNameAndValue - The specific attribute value to subscribe to, in the format "<attributeName>.<attributeValue>"

handlerMethod - The method to call when the event is fired. Can be a String of the method name or the method reference itself.

Location (page 325) location - The location to subscribe to

app - Pass in the available app property in the SmartApp to subscribe to touch events in the app.

Returns: void

Example:

```
preferences {
    section() {
        input "mycontact", "capability.contactSensor"
        input "myswitches", "capability.switch", multiple: true
    }
}
// subscribe to all state change events for ``contact`` attribute of a contact sensor
```



```

subscribe(mycontact, "contact", handlerMethod)

// subscribe to all state changes for all switch devices configured
subscribe(myswitches, "switch", handlerMethod)

// subscribe to the "open" event for the contact sensor - only when the state changes to "open" will
subscribe(mycontact, "contact.open", handlerMethod)

// subscribe to all state change events for the installed SmartApp's location
subscribe(location, handlerMethod)

// subscribe to touch events for this app - handlerMethod called when app is touched
subscribe(app, appTouchMethod)

// all event handler methods must accept an event parameter
def handlerMethod(evt) {
    log.debug "event name: ${evt.name}"
    log.debug "event value: ${evt.value}"
}

```

11.1.53 subscribeToCommand()

Subscribes to device commands that are sent to a device or devices. The specified `handlerMethod` will be called whenever the specified `command` is sent.

Signature: `void subscribeToCommand(deviceOrDevices, commandName, handlerMethod)`

Parameters:

`deviceOrDevices` - The *Device* (page 304) or list of devices to subscribe to.

`String commandName` - The command to subscribe to to

`handlerMethod` - the method to call when the command is called.

Returns: `void`

Example:

```

preferences {
    section() {
        input "switch1", "capability.switch"
    }
}
...
subscribeToCommand(switch1, "on", onCommand)
...
// called when the on() command is called on switch1
def onCommand(evt) {...}

```

11.1.54 timeOfDayIsBetween()

Find if a given date is between a lower and upper bound.

Signature: Boolean timeOfDayIsBetween(Date start, Date stop, Date value, TimeZone timeZone)

Parameters: Date start - The start date to compare against.

Date stop - The end date to compare against.

Date value - The date to compare to start and stop.

TimeZone timeZone - The time zone for this comparison.

Returns: Boolean - true if the specified date is between the start and stop dates, false otherwise.

Example:

```
def between = timeOfDayIsBetween(new Date() - 1, new Date() + 1,
                                new Date(), location.timeZone)
log.debug "between: $between" => true
```

11.1.55 timeOffset()

Gets a time offset in milliseconds for the specified input.

Signature: Long timeOffset(Number minutes)

Long timeOffset(String hoursAndMinutesString)

Parameters: Number minutes - The number of minutes to get the offset in milliseconds for.

String hoursAndMinutesString - A string in the format of "hh:mm" to get the offset in milliseconds for. Negative offsets are specified by prefixing the string with a minus sign ("-02:30").

Returns: Long - the time offset in milliseconds for the specified input.

Example:

```
def off1 = timeOffset(24)           // => 1440000
def off2 = timeOffset("2:30")       // => 9000000
def off2again = timeOffset(150)     // => 9000000
def off3 = timeOffset("-02:30")     // => -9000000
```

11.1.56 timeToday()

Gets a Date object for today's date, for the specified time in the date-time parameter.

Signature: Date timeToday(String timeString [, TimeZone timeZone])

Parameters: String timeString - Either an ISO-8601 date string as returned from time input preferences, or a simple time string in "hh:mm" format ("21:34").

TimeZone timeZone (optional) - The time zone to use for determining the current day.

Warning: Although the timeZone argument is optional, it is *strongly encouraged* that you use it. Not specifying the timeZone results in the SmartThings platform trying to calculate the time zone based on the date and time zone offsets in the input string.

To avoid time zone errors, you should specify the timeZone argument (you can get the time zone from the location object: location.timeZone)

Future releases may remove the option to call timeToday without a time zone.

Returns: `Date` - the Date that represents today's date for the specified time.

Example:

```
preferences {
    section() {
        input "startTime", "time"
        input "endTime", "time"
    }
}
...
def start = timeToday(startTime, location.timeZone)
def end = timeToday(endTime, location.timeZone)
```

11.1.57 timeTodayAfter()

Gets a `Date` object for the specified input that is guaranteed to be after the specified starting date.

Signature: `Date timeTodayAfter(String startTimeString, String timeString [, TimeZone timeZone])`

Parameters: `String startTimeString` - The time for which the returned date must be after. Can be an ISO-8601 date string as returned from time input preferences, or a simple time string in "hh:mm" format ("21:34").

`String timeString` - The time string to get the date object for. Can be an ISO-8601 date string as returned from time input preferences, or a simple time string in "hh:mm" format ("21:34").

`TimeZone timeZone` (*optional*) - The time zone used for determining the current date and time.

Warning: Although the `timeZone` argument is optional, it is *strongly encouraged* that you use it. Not specifying the `timeZone` results in the SmartThings platform trying to calculate the time zone based on the date and time zone offsets in the input string.

To avoid time zone errors, you should specify the `timeZone` argument (you can get the time zone from the `location` object: `location.timeZone`)

Future releases may remove the option to call `timeToday` without a time zone.

Returns: `Date` - the Date for the specified `timeString` that is guaranteed to be after the `startTimeString`.

Example:

```
preferences {
    section() {
        input "time1", "time"
        input "time2", "time"
    }
}
...
// assume time1 entered as 20:20
// assume time2 entered as 14:05
// nextTime would be tomorrow's date, 14:05 time.
def nextTime = timeTodayAfter(time1, time2, location.timeZone)
...
```

11.1.58 timeZone()

Get a *TimeZone* object for the specified time value entered as a SmartApp preference. This will get the current time zone of the mobile app (not the hub location).

Signature: `TimeZone timeZone(String timePreferenceString)`

Parameters: `String timeZoneString` - The time zone string in ISO-8061 format as used by SmartApp time preferences.

Returns: `TimeZone` - the `TimeZone` for the time zone as specified by the `timeZoneString`.

Example:

```
preferences {
    section() {
        input "mytime", "time"
    }
}

...
def enteredTimeZone = timeZone(mytime)
...
```

11.1.59 toDateTime()

Get a *Date* object for the specified string.

Signature: `Date toDateTime(dateTimeString)`

Parameters: `String dateTimeString` - the date-time string for which to get a `Date` object, in ISO-8061 format as used by time preferences

Returns: `Date` - the `Date` for the specified `dateTimeString`.

Example:

```
preferences {
    section() {
        input "mytime", "time"
    }
}

...
Date myTimeAsDate = toDateTime(mytime)
...
```

11.1.60 unschedule()

Deletes all scheduled jobs for the installed SmartApp.

Signature: `void unschedule()`

Returns: `void`

Note: This can be an expensive operation; make sure you need to do this before calling. Typically called in the `updated()` (page 249) method if the SmartApp has set up recurring schedules.

11.1.61 unsubscribe()

Deletes all subscriptions for the installed SmartApp, or for a specific device or devices if specified.

Typically should be called in the `updated()` (page 249) method, since device preferences may have changed.

Signature: `unsubscribe([deviceOrDevices])`

Parameters: `deviceOrDevices` (*optional*) - The device or devices for which to unsubscribe from. If not specified, all subscriptions for this installed SmartApp will be deleted.

Returns: `void`

Example:

```
def updated() {  
    unsubscribe()  
}
```

11.2 Device Handler

Device Handlers, or Device Types, are the virtual representation of a physical device. They are created by creating a new *SmartDevice* in the IDE.

A Device Handler defines a `metadata()` (page 292) method that defines the device's definition, UX information, as well as how it should behave in the IDE simulator.

A Device Handler typically also defines a `parse()` (page 278) method that is responsible for transforming raw messages from the device into events for the SmartThings platform.

Device Handlers must also define methods for any supported commands, either through its supported capabilities, or device-specific commands.

For more information about the structure of Device Handlers, refer to the Device Handler's Guide.

Tip: Writing a Device Handler is considered a somewhat advanced topic. Understanding of how a Device Handler is organized and operates is assumed in this reference documentation. You should be familiar with the contents of the Device Handler's Guide to get the most out of this documentation.

Methods expected to be defined by Device Handlers:

11.2.1 <command name>()

Note: This method is expected to be defined by Device Handlers.

The definition for a Command supported by this Device Handler. Every Command that a Device Handler supports, either through its capabilities or custom commands, must have a corresponding command method defined.

Commands are the things that a device can do. For example, the “Switch” capability defines the commands “on” and “off”. Every Device that supports the “Switch” capability must define an implementation of these commands. This is done by defining methods with the name of the command. For example, `def on() {}` and `def off()`.

The exact implementation of a command method will vary greatly depending upon the device. The command method is responsible for sending protocol and device-specific commands to the physical device.

Signature: `Object <command name>([arguments])>`

Returns: `Object` - Commands may return any object, but typically do not return anything since they perform some type of action.

Example:

```
metadata {
    // Automatically generated. Make future change here.
    definition (name: "Centralite Switch", namespace: "smarthings", author: "SmartThings") {
        ...
        capability "Switch"
        ...
    }
    ...
    // capability "Switch" declared, so all supported commands
    // of "Switch" must be implemented:
    def on() {
        // device-specific commands to turn the switch on
    }

    def off() {
        // device-specific commands to turn the switch off
    }
    ...
}
```

11.2.2 parse()

Note: This method is expected to be defined by Device Handlers.

Called when messages from a device are received from the hub. The parse method is responsible for interpreting those messages and returning [Event](#) (page 315) definitions. Event definitions are maps that contain, at a minimum, name and value entries. They may also contain unit, displayText, displayed, isStateChange, and linkText entries if the default, automatically generated values of these event properties are to be overridden. See the [createEvent\(\)](#) (page 283) documentation for a description of these properties.

Because the `parse()` method is responsible for handling raw device messages, their implementations vary greatly across different device types.

The `parse()` method may return a map defining the [Event](#) (page 315) to create and propagate through the SmartThings platform, or a list of events if multiple events should be created. It may also return a `HubAction` or list of `HubAction` objects in the case of LAN-connected devices.

Signature: Map parse(String description)

List<Map> parse(String description)

HubAction parse(String description)

List<HubAction> parse(String description)

Example:

```
def parse(String description) {
    log.debug "Parse description $description"
    def name = null
    def value = null
    if (description?.startsWith("read attr -")) {
        def descMap = parseDescriptionAsMap(description)
        log.debug "Read attr: $description"
        if (descMap.cluster == "0006" && descMap.attrId == "0000") {
            name = "switch"
            value = descMap.value.endsWith("01") ? "on" : "off"
        } else {
            def reportValue = description.split(",").find {it.split(":")[0].trim() == "value"}?.split(":").last()
            name = "power"
            // assume 16 bit signed for encoding and power divisor is 10
            value = Integer.parseInt(reportValue, 16) / 10
        }
    } else if (description?.startsWith("on/off:")) {
        log.debug "Switch command"
        name = "switch"
        value = description.endsWith(" 1") ? "on" : "off"
    }

    // createEvent returns a Map that defines an Event
    def result = createEvent(name: name, value: value)
    log.debug "Parse returned ${result?.descriptionText}"

    // returning the Event definition map creates an Event
    // in the SmartThings platform, and propagates it to
    // SmartApps subscribed to the device events.
    return result
}
```

11.2.3 apiServerUrl()

Returns the URL of the server where this Device Handler can be reached for API calls, along with the specified path appended to it. Use this instead of hard-coding a URL to ensure that the correct server URL for this installed instance is returned.

Signature: String apiServerUrl(String path)

Parameters: String path - the path to append to the URL

Returns: The URL of the server for this installed instance of the Device Handler.

Example:

```
// logs <server url>/my/path
log.debug "apiServerUrl: ${apiServerUrl("/my/path")}"
```

```
// The leading "/" will be added if you don't specify it
// logs <server url>/my/path
log.debug "apiServerUrl: ${apiServerUrl("my/path")}"
```

11.2.4 attribute()

Called within the *definition()* (page 284) method to declare that this Device Handler supports an attribute not defined by any of its declared capabilities.

For any supported attribute, it is expected that the Device Handler creates and sends events with the name of the attribute in the *parse()* (page 278) method.

Signature: `void attribute(String attributeName, String attributeType [, List possibleValues])`

Parameter: `String attributeName` - the name of the attribute

`String attributeType` - the type of the attribute. Available types are “string”, “number”, and “enum”

`List possibleValues` (*optional*) - the possible values for this attribute. Only valid with the “enum” attributeType.

Returns: void

Example:

```
metadata {
    definition (name: "Some Device Name", namespace: "somenamespace",
               author: "Some Author") {
        capability "Switch"
        capability "Polling"
        capability "Refresh"

        // also support the attribute "myCustomAttribute" - not defined by supported capabilities.
        attribute "myCustomAttribute", "number"

        // enum attribute with possible values "light" and "dark"
        attribute "someOtherName", "enum", ["light", "dark"]
    }
    ...
}
```

11.2.5 capability()

Called in the *definition()* (page 284) method to define that this device supports the specified capability.

Important: Whatever commands and attributes defined by that capability should be implemented by the Device Handler. For example, the “Switch” capability specifies support for the “switch” attribute and the “on” and “off” commands - any Device Handler supporting the “Switch” capability must define methods for the commands, and support the “switch” attribute by creating the appropriate events (with the name of the attribute, e.g., “switch”)

Signature: `void capability(String capabilityName)`

Parameters: `String` `capabilityName` - the name of the capability. This is the long-form name of the Capability name, not the “preferences reference”.

Returns: `void`

Example:

```
metadata {
    definition (name: "Cerbco Light Switch", namespace: "lennyv62",
               author: "Len Veil") {
        capability "Switch"
        ...
    }
    ...
}

def parse(description) {
    // handle device messages, determine what value of the event is
    return createEvent(name: "switch", value: someValue)
}

// need to define the on and off commands, since those
// are supported by "Switch" capability
def on() {
    ...
}

def off() {
}
```

11.2.6 carouselTile()

Called within the `tiles()` (page 297) method to define a tile often used in conjunction with the Image Capture capability, to allow users to scroll through recent pictures.

Signature: `void carouselTile(String tileName, String attributeName [,Map options, Closure closure])`

Parameters: `String` `tileName` - the name of the tile. This is used to identify the tile when specifying the tile layout.

`String` `attributeName` - the attribute this tile is associated with. Each tile is associated with an attribute of the device. The typical pattern is to prefix the attribute name with "device." - e.g., "device.water".

`Map` `options` (*optional*) - Various options for this tile. Valid options are found in the table below:

option	type	description
width	Integer	controls how wide the tile is. Default is 1.
height	Integer	controls how tall this tile is. Default is 1.
canChangeIcon	Boolean	true to allow the user to pick their own icon. Defaults to false.
canChangeBackground	Boolean	true to allow a user to choose their own background image for the tile. Defaults to false.
decoration	String	specify "flat" for the tile to render without a ring.
range	String	used to specify a custom range. In the form of "<lower bound>..<upper bound>"

Closure `closure` (*optional*) - a closure that defines any states for the tile.

Returns: void

Example:

```
tiles {
    carouselTile("cameraDetails", "device.image", width: 3, height: 2) { }
}
```

11.2.7 command()

Called within the *definition()* (page 284) method to declare that this Device Handler supports a command not defined by any of its declared capabilities.

For any supported command, it is expected that the Device Handler define a *<command name>()* (page 278) method with a corresponding name.

Signature: void `command(String commandName [, List parameterTypes])`

Parameter: **String** `commandName` - the name of the command.

List `parameterTypes` (*optional*) - a list of strings that defines the types of the parameters the command requires (in order), if any. Typical values are “string”, “number”, and “enum”.

Returns: void

Example:

```
metadata {
    definition (name: "Some Device Name", namespace: "somenamespace",
               author: "Some Author") {
        capability "Switch"
        capability "Polling"
        capability "Refresh"

        // also support the attribute "myCustomCommand" - not defined by supported capabilities.
        command "myCustomCommand"

        // commands can take parameters
        command "myCustomCommandWithParams", ["string", "number"]

    }
    ...
}

def myCustomCommand() {
    ...
}

def myCustomCommandWithParams(def stringArg, def numArg) {
    ...
}
```

11.2.8 controlTile()

Called within the `tiles()` (page 297) method to define a tile that allows the user to input a value within a range. A common use case for a control tile is a light dimmer.

Signature: `void controlTile(String tileName, String attributeName, String controlType [, Map options, Closure closure])`

Returns: `void`

Parameters: `String tileName` - the name of the tile. This is used to identify the tile when specifying the tile layout.

`String attributeName` - the attribute this tile is associated with. Each tile is associated with an attribute of the device. The typical pattern is to prefix the attribute name with "device." - e.g., "device.water".

`String controlType` - the type of control. Either "slider" or "control".

`Map options` (*optional*) - Various options for this tile. Valid options are found in the table below:

option	type	description
width	Integer	controls how wide the tile is. Default is 1.
height	Integer	controls how tall this tile is. Default is 1.
canChangeIcon	Boolean	true to allow the user to pick their own icon. Defaults to false.
canChangeBackground	Boolean	true to allow a user to choose their own background image for the tile. Defaults to false.
decoration	String	specify "flat" for the tile to render without a ring.
range	String	used to specify a custom range. In the form of "<lower bound>..<upper bound>"

`Closure closure` (*optional*) - A closure that calls any `state()` (page 295) methods to define how the tile should appear for various attribute values.

Example:

```
tiles {
    controlTile("levelSliderControl", "device.level", "slider", height: 1,
        width: 2, inactiveLabel: false, range: "(0..100)") {
        state "level", action: "switch level.setLevel"
    }
}
```

11.2.9 createEvent()

Creates a Map that represents an *Event* (page 315) object. Typically used in the `parse()` (page 278) method to define Events for particular attributes. The resulting map is then returned from the `parse()` method. The SmartThings platform will then create an Event object and propagate it through the system.

Signature: `Map createEvent(Map options)`

Parameters: `Map options` - The various properties that define this Event. The available options are listed below. It is not necessary, or typical, to define all the available options. Typically only the `name` and `value` options are required.

Property	Type	Description
name (required)	String	the name of the event. Typically corresponds to an attribute name of a capability.
value (required)	Object	the value of the event. The value is stored as a string, but you can pass numbers or other objects.
descriptionText	String	the description of this event. This appears in the mobile application activity for the device. If not specified, this will be created using the event name and value.
displayed	Boolean	specify <code>true</code> to display this event in the mobile application activity feed, <code>false</code> to not display. Defaults to <code>true</code> .
linkText	String	name of the event to show in the mobile application activity feed.
isState-Change	Boolean	specify <code>true</code> if this event caused a device attribute to change state. Typically not used, since it will be set automatically.
unit	String	a unit string, if desired. This will be used to create the <code>descriptionText</code> if it (the <code>descriptionText</code> option) is not specified.

Example:

```
def parse(String description) {
    ...

    def evt1 = createEvent(name: "someName", value: "someValue")
    def evt2 = createEvent(name: "someOtherName", value: "someOtherValue")

    return [evt1, evt2]
}
```

11.2.10 definition()

Called within the *metadata()* (page 292) method, and defines some basic information about the device, as well as the supported capabilities, commands, and attributes.

Signature: `void definition(Map definitionData, Closure closure)`

Parameters: `Map definitionData` - defines various metadata about this Device Handler. Valid options are:

option	type	description
name	String	the name of this Device Handler
namespace	String	the namespace for this Device Handler. Typically the same as the author's github user name.
author	String	the name of the author.

`Closure closure` - A closure with method calls to *capability()* (page 280) , *command()* (page 282) , or *attribute()* (page 280) .

Returns: `void`

Example:

```
metadata {
    definition (name: "My Device Name", namespace: "mynamespace",
               author: "My Name") {
        capability "Switch"
        capability "Polling"
    }
}
```

```

        capability "Refresh"

        command "someCustomCommand"

        attribute "someCustomAttribute", "number"
    }
    ...
}

```

11.2.11 details()

Used within the *tiles()* (page 297) method to define the order that the tiles should appear in.

Signature: void details(List<String> tileDefinitions)

Parameters: List<String> tileDefinitions - A list of tile names that defines the order of the tiles (left-to-right, top-to-bottom)

Returns: void

Example:

```

tiles {
    standardTile("switchTile", "device.switch", width: 2, height: 2,
                canChangeIcon: true) {
        state "off", label: '${name}', action: "switch.on",
            icon: "st.switches.switch.off", backgroundColor: "#ffffff"
        state "on", label: '${name}', action: "switch.off",
            icon: "st.switches.switch.on", backgroundColor: "#E60000"
    }
    valueTile("powerTile", "device.power", decoration: "flat") {
        state "power", label: '${currentValue} W'
    }
    standardTile("refreshTile", "device.power", decoration: "ring") {
        state "default", label: '', action: "refresh.refresh",
            icon: "st.secondary.refresh",
    }

    main "switchTile"

    // defines what order the tiles are defined in
    details(["switchTile", "powerTile", "refreshTile"])
}

```

11.2.12 device

the Device object, from which its current properties and history can be accessed. As of now this object is of a different class than the Device object available in SmartApps. As some point these will be merged, but for now the properties and methods of the device object available to the device type handler are discussed in the example below:

```

...
// Gets the most recent State for the given attribute
def state1 = device.currentState("someAttribute")
def state2 = device.latestState("someOtherAttribute")

```

```
// Gets the current value for the given attribute
// Return type will vary depending on the device
def curVal1 = device.currentValue("someAttribute")
def curVal2 = device.latestValue("someOtherAttribute")

// gets the display name of the device
def displayName = device.displayName

// gets the internal unique system identifier for this device
def thisId = device.id

// gets the internal name for this device
def thisName = device.name

// gets the user-defined label for this device
def thisLabel = device.label
```

11.2.13 fingerprint()

Called within the *definition()* (page 284) method to define the information necessary to pair this device type to the hub. See the Fingerprinting Section of the Device Handler guide for more information.

11.2.14 getApiServerUrl()

Returns the URL of the server where this Device Handler can be reached for API calls. Use this instead of hard-coding a URL to ensure that the correct server URL for this installed instance is returned.

Signature: `String getApiServerUrl()`

Returns: `String` - the URL of the server where this Device Handler can be reached.

11.2.15 httpDelete()

Executes an HTTP DELETE request and passes control to the specified closure. The closure is passed one `HttpResponseDecorator` argument from which the response content and header information can be extracted.

Signature: `void httpDelete(String uri, Closure closure)`

`void httpDelete(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP DELETE call to.

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
uri	Either a URI or URL of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
contentType	Request content type and Accept header.
requestContent-Type	Content type for the request, if it is different from the expected response content-type.
body	Request body that will be encoded based on the given contentType.

Closure closure - The closure that will be called with the response of the request.

Returns: void

11.2.16 httpGet()

Executes an HTTP GET request and passes control to the specified closure. The closure is passed one [HttpResponseDecorator](#) argument from which the response content and header information can be extracted.

If the response content type is JSON, the response data will automatically be parsed into a data structure.

Signature: void httpGet(String uri, Closure closure)

void httpGet(Map params, Closure closure)

Parameters: String uri - The URI to make the HTTP GET call to

Map params - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
uri	Either a URI or URL of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
contentType	Request content type and Accept header.
requestContent-Type	Content type for the request, if it is different from the expected response content-type.
body	Request body that will be encoded based on the given contentType.

Closure - closure - The closure that will be called with the response of the request.

Example:

```
def params = [
    uri: "http://httpbin.org",
    path: "/get"
]

try {
    httpGet(params) { resp ->
        resp.headers.each {
            log.debug "${it.name} : ${it.value}"
        }
        log.debug "response contentType: ${resp.contentType}"
        log.debug "response data: ${resp.data}"
    }
} catch (e) {
    log.error "something went wrong: $e"
}
```

11.2.17 httpHead()

Executes an HTTP HEAD request and passes control to the specified closure. The closure is passed one [HttpResponseDecorator](#) argument from which the response content and header information can be extracted.

Signature: `void httpHead(String uri, Closure closure)`
`void httpHead(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP HEAD call to

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
<code>uri</code>	Either a URI or URL of the endpoint to make a request from.
<code>path</code>	Request path that is merged with the URI.
<code>query</code>	Map of URL query parameters.
<code>headers</code>	Map of HTTP headers.
<code>contentType</code>	Request content type and Accept header.
<code>requestContent-Type</code>	Content type for the request, if it is different from the expected response content-type.
<code>body</code>	Request body that will be encoded based on the given <code>contentType</code> .

`Closure closure` - The closure that will be called with the response of the request.

11.2.18 httpPost()

Executes an HTTP POST request and passes control to the specified closure. The closure is passed one [HttpResponseDecorator](#) argument from which the response content and header information can be extracted.

If the response content type is JSON, the response data will automatically be parsed into a data structure.

Signature: `void httpPost(String uri, String body, Closure closure)`
`void httpPost(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP GET call to

`String body` - The body of the request

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
<code>uri</code>	Either a URI or URL of the endpoint to make a request from.
<code>path</code>	Request path that is merged with the URI.
<code>query</code>	Map of URL query parameters.
<code>headers</code>	Map of HTTP headers.
<code>contentType</code>	Request content type and Accept header.
<code>requestContent-Type</code>	Content type for the request, if it is different from the expected response content-type.
<code>body</code>	Request body that will be encoded based on the given <code>contentType</code> .

`Closure closure` - The closure that will be called with the response of the request.

Example:


```
try {
    httpPost("http://mysite.com/api/call", "id=XXX&value=YYY") { resp ->
        log.debug "response data: ${resp.data}"
        log.debug "response contentType: ${resp.contentType}"
    }
} catch (e) {
    log.debug "something went wrong: $e"
}
```

11.2.19 httpPostJson()

Executes an HTTP POST request with a JSON-encoded body and content type, and passes control to the specified closure. The closure is passed one [HttpResponseDecorator](#) argument from which the response content and header information can be extracted.

If the response content type is JSON, the response data will automatically be parsed into a data structure.

Signature: void httpPostJson(String uri, String body, Closure closure)

void httpPostJson(String uri, Map body, Closure closure)

void httpPostJson(Map params, Closure closure)

Parameters: String uri - The URI to make the HTTP POST call to

String body - The body of the request

Map params - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
uri	Either a URI or URL of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
contentType	Request content type and Accept header.
requestContent-Type	Content type for the request, if it is different from the expected response content-type.
body	Request body that will be encoded based on the given contentType.

Closure closure - The closure that will be called with the response of the request.

Example:

```
def params = [
    uri: "http://postcatcher.in/catchers/<yourUniquePath>",
    body: [
        param1: [subparam1: "subparam 1 value",
                 subparam2: "subparam2 value"],
        param2: "param2 value"
    ]
]

try {
    httpPostJson(params) { resp ->
        resp.headers.each {
            log.debug "${it.name} : ${it.value}"
        }
        log.debug "response contentType: ${resp.contentType}"
    }
}
```

```
}  
} catch (e) {  
    log.debug "something went wrong: $e"  
}
```

11.2.20 httpPut()

Executes an HTTP PUT request and passes control to the specified closure. The closure is passed one [HttpResponseDecorator](#) argument from which the response content and header information can be extracted.

If the response content type is JSON, the response data will automatically be parsed into a data structure.

Signature: void httpPut(String uri, String body, Closure closure)

void httpPut(Map params, Closure closure)

Parameters: String uri - The URI to make the HTTP GET call to

String body - The body of the request

Map params - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
uri	Either a URI or URL of of the endpoint to make a request from.
path	Request path that is merged with the URI.
query	Map of URL query parameters.
headers	Map of HTTP headers.
contentType	Request content type and Accept header.
requestContent-Type	Content type for the request, if it is different from the expected response content-type.
body	Request body that will be encoded based on the given contentType.

Closure closure - The closure that will be called with the response of the request.

Example:

```
try {  
    httpPut("http://mysite.com/api/call", "id=XXX&value=YYY") { resp ->  
        log.debug "response data: ${resp.data}"  
        log.debug "response contentType: ${resp.contentType}"  
    }  
} catch (e) {  
    log.error "something went wrong: $e"  
}
```

11.2.21 httpPutJson()

Executes an HTTP PUT request with a JSON-encoded body and content type, and passes control to the specified closure. The closure is passed one [HttpResponseDecorator](#) argument from which the response content and header information can be extracted.

If the response content type is JSON, the response data will automatically be parsed into a data structure.

Signature: `void httpPutJson(String uri, String body, Closure closure)`
`void httpPutJson(String uri, Map body, Closure closure)`
`void httpPutJson(Map params, Closure closure)`

Parameters: `String uri` - The URI to make the HTTP PUT call to

`String body` - The body of the request

`Map params` - A map of parameters for configuring the request. The valid parameters are:

Parameter	Description
<code>uri</code>	Either a URI or URL of the endpoint to make a request from.
<code>path</code>	Request path that is merged with the URI.
<code>query</code>	Map of URL query parameters.
<code>headers</code>	Map of HTTP headers.
<code>contentType</code>	Request content type and Accept header.
<code>requestContent-Type</code>	Content type for the request, if it is different from the expected response content-type.
<code>body</code>	Request body that will be encoded based on the given <code>contentType</code> .

`Closure closure` - The closure that will be called with the response of the request.

11.2.22 main()

Used to define what tile appears on the main “Things” view in the mobile application. Can be called within the `tiles()` (page 297) method.

Signature: `void main(String tileName)`

Parameters: `String tileName` - the name of the tile to display as the main tile.

Returns: `void`

Example:

```
tiles {
    standardTile("switchTile", "device.switch", width: 2, height: 2,
        canChangeIcon: true) {
        state "off", label: '${name}', action: "switch.on",
            icon: "st.switches.switch.off", backgroundColor: "#ffffff"
        state "on", label: '${name}', action: "switch.off",
            icon: "st.switches.switch.on", backgroundColor: "#E60000"
    }
    valueTile("powerTile", "device.power", decoration: "flat") {
        state "power", label: '${currentValue} W'
    }
    standardTile("refreshTile", "device.power", decoration: "ring") {
        state "default", label: '', action: "refresh.refresh",
            icon: "st.secondary.refresh",
    }

    // The "switchTile" will be main tile, displayed in the "Things" view
    main "switchTile"
    details(["switchTile", "powerTile", "refreshTile"])
}
```

11.2.23 metadata()

Used to define metadata such as this Device Handler's supported capabilities, attributes, commands, and UX information.

Signature: void metadata(Closure closure)

Parameters: Closure closure - a closure that defines the metadata. The closure is expected to have the following methods called in it: *definition()* (page 284) , *simulator()* (page 293) , and *tiles()* (page 297) .

Returns: void

Example:

```
metadata {
    definition(name: "device name", namespace: "yournamespace", author: "your name") {
        // supported capabilities, commands, attributes,
    }
    simulator {
        // simulator metadata
    }
    tiles {
        // tiles metadata
    }
}
```

11.2.24 reply()

Called in the *simulator()* (page 293) method to model the behavior of a physical device when a virtual instance of the device type is run in the IDE.

The simulator matches command strings generated by the device to those specified in the `commandString` argument of a reply method and, if a match is found, calls the device handler's `parse` method with the corresponding `messageDescription`.

For example, the reply method `reply "2001FF,2502": "command: 2503, payload: FF"` models the behavior of a physical Z-Wave switch in responding to an Basic Set command followed by a Switch Binary Get command. The result will be a call to the `parse` method with a Switch Binary Report command with a value of 255, i.e., the turning on of the switch. Modeling turn off would be done with the reply method `reply "200100,2502": "command: 2503, payload: 00"`.

Signature: void reply(String commandString, String messageDescription)

Parameters: String commandString - a String that represents the command.

String messageDescription - a String that represents the message description.

Returns: void

Example:

```
metadata {
    ...

    // simulator metadata
    simulator {
        // 'on' and 'off' will appear in the messages dropdown, and send
        // "on/off: 1 to the parse method"
```

```

status "on": "on/off: 1"
status "off": "on/off: 0"

// simulate reply messages from the device
reply "zcl on-off on": "on/off: 1"
reply "zcl on-off off": "on/off: 0"
}
...
}

```

11.2.25 sendEvent()

Create and fire an [Event](#) (page 315) . Typically a Device Handler will return the map returned from [createEvent\(\)](#) (page 283) , which will allow the platform to create and fire the event. In cases where you need to fire the event (outside of the [parse\(\)](#) (page 278) method), `sendEvent()` is used.

Signature: `void sendEvent(Map properties)`

Parameters: `Map properties` - The properties of the event to create and send.

Here are the available properties:

Property	Description
name (required)	String - The name of the event. Typically corresponds to an attribute name of a capability.
value (required)	The value of the event. The value is stored as a string, but you can pass numbers or other objects.
descriptionText	String - The description of this event. This appears in the mobile application activity for the device. If not specified, this will be created using the event name and value.
displayed	Pass <code>true</code> to display this event in the mobile application activity feed, <code>false</code> to not display. Defaults to <code>true</code> .
linkText	String - Name of the event to show in the mobile application activity feed.
isStateChange	<code>true</code> if this event caused a device attribute to change state. Typically not used, since it will be set automatically.
unit	String - a unit string, if desired. This will be used to create the <code>descriptionText</code> if it (the <code>descriptionText</code> option) is not specified.

Tip: Not all event properties need to be specified. ID properties like `deviceId` and `locationId` are automatically set, as are properties like `isStateChange`, `displayed`, and `linkText`.

Returns: `void`

Example:

```
sendEvent (name: "temperature", value: 72, unit: "F")
```

11.2.26 simulator()

Defines information used to simulate device interaction in the IDE. Can be called in the [metadata\(\)](#) (page 292) method.

Signature: void simulator(Closure closure)

Parameters: Closure closure - the closure that defines the *status()* (page 296) and *reply()* (page 292) messages.

Returns: void

Example:

```
metadata {
    ...

    // simulator metadata
    simulator {
        // 'on' and 'off' will appear in the messages dropdown, and send
        // "on/off: 1 to the parse method"
        status "on": "on/off: 1"
        status "off": "on/off: 0"

        // simulate reply messages from the device
        reply "zcl on-off on": "on/off: 1"
        reply "zcl on-off off": "on/off: 0"
    }
    ...
}
```

11.2.27 standardTile()

Called within the *tiles()* (page 297) method to define a tile to display current state information. For example, to show that a switch is on or off, or that there is or is not motion.

Signature: void standardTile(String tileName, String attributeName [, Map options, Closure closure])

Returns: void

Parameters: String tileName - the name of the tile. This is used to identify the tile when specifying the tile layout.

String attributeName - the attribute this tile is associated with. Each tile is associated with an attribute of the device. The typical pattern is to prefix the attribute name with "device." - e.g., "device.water".

Map options (*optional*) - Various options for this tile. Valid options are found in the table below:

option	type	description
width	Integer	controls how wide the tile is. Default is 1.
height	Integer	controls how tall this tile is. Default is 1.
canChangeIcon	Boolean	true to allow the user to pick their own icon. Defaults to false.
canChangeBackground	Boolean	true to allow a user to choose their own background image for the tile. Defaults to false.
decoration	String	specify "flat" for the tile to render without a ring.

Closure closure (*optional*) - A closure that calls any *state()* (page 295) methods to define how the tile should appear for various attribute values.

Example:

```
tile {
    standardTile("water", "device.water", width: 2, height: 2) {
        state "dry", icon:"st.alarm.water.dry", backgroundColor:"#ffffff"
        state "wet", icon:"st.alarm.water.wet", backgroundColor:"#53a7c0"
    }
}
```

11.2.28 state

A map of name/value pairs that a Device Handler can use to save and retrieve data across executions.

Signature: Map state

Returns: Map - a map of name/value pairs.

```
state.count = 0
state.count = state.count + 1

log.debug "state.count: ${state.count}"

// use array notation if you wish
log.debug "state['count']: ${state['count']}"

// you can store lists and maps to make more interesting structures
state.listOfMaps = [[key1: "vall", bool1: true],
                    [otherKey: ["string1", "string2"]]]
```

Warning: Though state can be treated as a map in most regards, certain convenience operations that you may be accustomed to in maps will not work with state. For example, `state.count++` will not increment the count - use the longer form of `state.count = state.count + 1`.

11.2.29 state()

Called within any of the various tiles method's closure to define options to be used when the current value of the tile's attribute matches the value argument.

Signature: void state(stateName, Map options)

Parameters: String stateName - the name of the attribute value for which to display this state for.

Map options - a map that defines additional information for this state. The valid options are:

option	type	description
action	String	the action to take when this tile is pressed. The form is <capabilityReference>.<command>.
back-ground-Color	String	a hexadecimal color code to use for the background color. This has no effect if the tile has decoration: "flat".
back-ground-Colors	String	specify a list of maps of attribute values and colors. The mobile app will match and interpolate between these entries to select a color based on the value of the attribute.
default-State	Boolean	specify true if this state should be the active state displayed for this tile.
icon	String	the identifier of the icon to use for this state. You can view the icon options here.
label	String	the label for this state.

Returns: void

Example:

```
...
standardTile("water", "device.water", width: 2, height: 2) {
    // when the "water" attribute has the value "dry", show the
    // specified icon and background color
    state "dry", icon:"st.alarm.water.dry", backgroundColor:"#ffffff"

    // when the "water" attribute has the value "wet", show the
    // specified icon and background color
    state "wet", icon:"st.alarm.water.wet", backgroundColor:"#53a7c0"
}

valueTile("temperature", "device.temperature", width: 2, height: 2) {
    state("temperature", label:'${currentValue}°',
        backgroundColors:[
            [value: 31, color: "#153591"],
            [value: 44, color: "#1e9cbb"],
            [value: 59, color: "#90d2a7"],
            [value: 74, color: "#44b621"],
            [value: 84, color: "#f1d801"],
            [value: 95, color: "#d04e00"],
            [value: 96, color: "#bc2323"]
        ]
    )
}
...
```

11.2.30 status()

The status method is called in the *simulator()* (page 293) method, and populates the select box that appears under virtual devices in the IDE. Can be called in the *simulator()* (page 293) method.

Signature: void status(String name, String messageDescription)

Parameters: *String* name - any unique string and is used to refer to this status message in the select box.

String messageDescription - should be a parseable message for this device type. It's passed to the device type handler's parse method when select box entry is sent in the simulator. For example, status "on": "command: 2003, payload: FF" will send a Z-Wave Basic Report command to the device handler's parse method when the on option is selected and sent.

Returns: void

Example:

```
metadata {
    ...

    // simulator metadata
    simulator {
        // 'on' and 'off' will appear in the messages dropdown, and send
        // "on/off: 1 to the parse method"
        status "on": "on/off: 1"
        status "off": "on/off: 0"

        // simulate reply messages from the device
        reply "zcl on-off on": "on/off: 1"
        reply "zcl on-off off": "on/off: 0"
    }
    ...
}
```

11.2.31 tiles()

Defines the user interface for the device in the mobile app. It's composed of one or more [standardTile\(\)](#) (page 294) , [valueTile\(\)](#) (page 298) , [carouselTile\(\)](#) (page 281) , or [controlTile\(\)](#) (page 283) methods, as well as a [main\(\)](#) (page 291) and [details\(\)](#) (page 285) method.

Signature: void tiles(Closure closure)

Parameters: Closure closure - A closure that defines the various tiles and metadata.

Returns: void

Example:

```
tiles {
    standardTile("switchTile", "device.switch", width: 2, height: 2,
        canChangeIcon: true) {
        state "off", label: '${name}', action: "switch.on",
            icon: "st.switches.switch.off", backgroundColor: "#ffffff"
        state "on", label: '${name}', action: "switch.off",
            icon: "st.switches.switch.on", backgroundColor: "#E60000"
    }
    valueTile("powerTile", "device.power", decoration: "flat") {
        state "power", label: '${currentValue} W'
    }
    standardTile("refreshTile", "device.power", decoration: "ring") {
        state "default", label: '', action: "refresh.refresh",
            icon: "st.secondary.refresh",
    }

    main "switchTile"
    details(["switchTile", "powerTile", "refreshTile"])
}
```

11.2.32 valueTile()

Defines a tile that displays a specific value. Typical examples include temperature, humidity, or power values. Called within the `tiles()` (page 297) method.

Signature: `void valueTile(String tileName, String attributeName [, Map options, Closure closure])`

Returns: `void`

Parameters: `String tileName` - the name of the tile. This is used to identify the tile when specifying the tile layout.

`String attributeName` - the attribute this tile is associated with. Each tile is associated with an attribute of the device. The typical pattern is to prefix the attribute name with "device." - e.g., "device.power".

`Map options` (*optional*) - Various options for this tile. Valid options are found in the table below:

option	type	description
width	Integer	controls how wide the tile is. Default is 1.
height	Integer	controls how tall this tile is. Default is 1.
canChangeIcon	Boolean	true to allow the user to pick their own icon. Defaults to false.
canChangeBackground	Boolean	true to allow a user to choose their own background image for the tile. Defaults to false.
decoration	String	specify "flat" for the tile to render without a ring.

`Closure closure` (*optional*) - A closure that calls any `state()` (page 295) methods to define how the tile should appear for various attribute values.

Example:

```
tiles {
    valueTile("power", "device.power", decoration: "flat") {
        state "power", label: '${currentValue} W'
    }
}
```

11.2.33 zigbee

Warning: The documentation for this property is incomplete.

A utility class for parsing and formatting ZigBee messages.

Signature: `Zigbee zigbee`

Returns: A reference to the ZigBee utility class.

11.2.34 zwave

The utility class for parsing and formatting Z-Wave command messages.

Signature: `ZWave zwave`

Returns: A reference to the ZWave helper class. See the *Z-Wave Reference* (page 335) for more information.

Example:

```
// On command implementation for a Z-Wave switch
def on() {
    delayBetween([
        zwave.basicV1.basicSet(value: 0xFF).format(),
        zwave.switchBinaryV1.switchBinaryGet().format()
    ])
}
```

11.3 Attribute

An Attribute represents specific information about the state of a device. For example, the “Temperature Measurement” capability has an attribute named “temperature” that represents the temperature data.

The Attribute object contains metadata information about the Attribute itself - its name, data type, and possible values.

You will typically interact with Attributes values directly, for example, using the *current<Uppercase attribute name>* (page 305) method on a *Device* (page 304) instance. That will get the *value* of the Attribute, which is typically what SmartApps are most interested in.

You can get the supported Attributes of a Device through the Device’s *supportedAttributes* (page 314) method.

Warning: Referring to an Attribute directly from a Device by calling `someDevice.attributeName` will return an Attribute object with only the `name` property available. This is available for legacy purposes only, and will likely be removed at some time.

To get a reference to an Attribute object, you should use the `supportedAttributes` method on the Device object, and then find the desired Attribute in the returned List.

You can view the available attributes for all Capabilities in our *Capabilities Reference* (page 217).

11.3.1 dataType

Gets the data type of this Attribute.

Signature: `String dataType`

Returns: `String` - the data type of this Attribute. Possible types are “STRING”, “NUMBER”, “VECTOR3”, “ENUM”.

Example:

```
preferences {
    section() {
        input "thetemp", "capability.temperatureMeasurement"
    }
}
...
def attrs = thetemp.supportedAttributes
attrs.each {
    log.debug "${thetemp.displayName}, attribute ${it.name}, dataType: ${it.dataType}"
}
```

```
}  
...
```

11.3.2 name

The name of the Attribute.

Signature: `String name`

Returns: `String` - the name of this attribute

Example:

```
preferences {  
    section() {  
        input "myswitch", "capability.switch"  
    }  
}  
...  
// switch capability has an attribute named "switch"  
def switchAttr = myswitch.switch  
log.debug "switch attribute name: ${switchAttr.name}"  
...
```

11.3.3 values

The possible values for this Attribute, if the data type is “ENUM”.

Signature: `List<String> values`

Returns: `List < String >` - the possible values for this Attribute, if the data type is “ENUM”. An empty list is returned if there are no possible values or if the data type is not “ENUM”.

Example:

```
preferences {  
    section() {  
        input "thetemp", "capbility.temperatureMeasurement"  
    }  
}  
...  
def attrs = thetemp.supportedAttributes  
attrs.each {  
    log.debug "${thetemp.displayName}, attribute ${it.name}, values: ${it.values}"  
    log.debug "${thetemp.displayName}, attribute ${it.name}, dataType: ${it.dataType}"  
}  
...
```

11.4 Capability

The Capability object encapsulates information about a certain Capability.

A Capability object cannot be created. You can get the Capabilities for a given device using the capabilities method on a [Device](#) (page 304) instance:

```
def capabilities = mydevice.capabilities
```

For documentation for the available Capabilities, you can refer to the [Capabilities Reference](#) (page 217).

11.4.1 name

The name of the capability.

Signature: String name

Returns: String - the name of the capability.

Example:

```
preferences {
    section() {
        input "mySwitch", "capability.switch"
    }
}
...
def mySwitchCaps = mySwitch.capabilities

// log each capability supported by the "mySwitch" device
mySwitchCaps.each {cap ->
    log.debug "Capability name: ${cap.name}"
}
...
```

11.4.2 attributes

Signature: List<Attribute> attributes

Returns: List <[Attribute](#) (page 299)> - A list of Attributes of this capability. An empty list will be returned if this Capability has no Attributes.

Example:

```
preferences {
    section() {
        input "mySwitch", "capability.switch"
    }
}
...
def mySwitchCaps = mySwitch.capabilities

// log each capability supported by the "mySwitch" device, along
// with all its supported attributes
mySwitchCaps.each {cap ->
    log.debug "Capability name: ${cap.name}"
    cap.attributes.each {attr ->
        log.debug "-- Attribute name: ${attr.name}"
    }
}
```

```
}  
}  
...
```

11.4.3 commands

Signature: `List<Command> commands`

Returns: `List<Command>` (page 302) - A list of Commands of this capability. An empty list will be returned if this Capability has no commands.

Example:

```
preferences {  
    section() {  
        input "mySwitch", "capability.switch"  
    }  
}  
...  
def mySwitchCaps = mySwitch.capabilities  
  
// log each capability supported by the "mySwitch" device, along  
// with all its supported commands  
mySwitchCaps.each {cap ->  
    log.debug "Capability name: ${cap.name}"  
    cap.commands.each {comm ->  
        log.debug "-- Command name: ${comm.name}"  
    }  
}  
...  
}
```

11.5 Command

A Command represents an action you can perform on a Device.

An instance of a Command object encapsulates information about that Command. You cannot create a Command object; you can retrieve them from a [Capability](#) (page 300) or from a [Device](#) (page 304):

```
preferences {  
    section() {  
        input "theswitch", "capability.switch"  
    }  
}  
...  
  
// Get a list of Commands supported by theswitch:  
def switchCommands = theswitch.supportedCommands  
log.debug "switchCommands: $switchCommands"  
  
// Iterate through the supported capabilities, log all supported commands:  
// commands property available via the Capability object  
def caps = theswitch.capabilities
```

```
caps.commands.each {comm ->
    log.debug "-- Command name: ${comm.name}"
}
```

11.5.1 arguments

The list of argument types for the command.

Signature: `List<String> arguments`

Returns: `List<String>` - A list of the argument types for this command. One of “*STRING*”, “*NUMBER*”, “*VECTOR3*”, or “*ENUM*”.

Example:

```
preferences {
    section() {
        input "theSwitchLevel", "capability.switchLevel"
    }
}
...
def supportedCommands = theSwitchLevel.supportedCommands

// logs each command's arguments
supportedCommands.each {
    log.debug "arguments for swithLevel command ${it.name}: ${it.arguments}"
}
...
```

11.5.2 name

The name of the command.

Signature: `String name`

Returns: `String` - the name of this command.

Example:

```
preferences {
    section() {
        input "theswitch", "capability.switch"
    }
}
...
def supportedCommands = theswitch.supportedCommands

// logs each command name supported by theswitch
supportedCommands.each {
    log.debug "command name: ${it.name}"
}
...
```

11.6 Device

The Device object represents a physical device in a SmartApp. When a user installs a SmartApp, they typically will select the devices to be used by the SmartApp. SmartApps can then interact with these Device objects to get device information, or send commands to the Device.

Device objects cannot be instantiated, but are created by the SmartThings platform and available via the name given in the preferences definition of a SmartApp:

```
preferences {
  section() {
    // prompt user to select a device that supports the switch capability.
    // assign the chosen device to a variable named "theswitch"
    input "theswitch", "capability.switch"
  }
}
...
// access Device instance using the input name:
def deviceDisplayName = theswitch.displayName
...
```

Note: Event history is limited to the last seven days. Methods that query devices for event history will only query the last seven days. This will be called out in those methods, but is good to be generally aware of.

11.6.1 <attribute name>State

The latest [State](#) (page 329) instance for the specified Attribute.

The exact name will vary depending on the device and its available attributes.

For example, the Thermostat capability supports several attributes. To get the State for any of the attributes, simply use the attribute name to construct the call. Consider the case of the “temperature” and “heatingSetpoint” attributes:

```
somethermostat.temperatureState
somethermostat.heatingSetpointState
```

Signature: State <attribute name>State

Returns: [State](#) (page 329) - The latest State instance for the specified Attribute.

Example:

```
preferences {
  section() {
    input "thetemp", "capability.temperatureMeasurement"
  }
}
...
// The Temperature Measurement has a "temperature" attribute.
// so the form is <attribute name>State = temperatureState
def tempState = thetemp.temperatureState
...
```


11.6.2 <command name>()

Executes the specified command on the Device.

The method name will vary on the Device and Command being called.

For example, a Device that supports the Switch capability has both the `on()` and `off()` commands.

Some commands may take parameters; you will pass those parameters to the command as well.

Signature: `void <command name>()`

```
void <command name>([delay: Number])
void <command name>(arguments)
void <command name>(arguments, [delay: Number'])
```

Parameters: `arguments` - The arguments to the command, if required.

Map options - A map of options to send to the command. Only the `delay` option is currently supported:

option	type	description
delay	Number	The number of milliseconds to wait before sending the command to the device.

Returns: `void`

Example:

```
preferences {
    section() {
        input "theswitch", "capability.switch"
        input "thethermostat", "capability.thermostat"
    }
}
...
// call the "on" command on theswitch - no arguments
theswitch.on()

// call the "setHeatingSetpoint" command on thethermostat - takes an argument:
thethermostat.setHeatingSetpoint(72)

// A map specifying command options can be specified as the last parameter.
// Only supported options are "delay":
theswitch.on([delay: 30000]) // send command after 30 seconds
thethermostat.setHeatingSetpoint(72, [delay: 30000])
...
```

11.6.3 current<Uppercase attribute name>

The latest reported values for the specified attribute.

The specific signature will vary depending on the attribute name. Follow the pattern of `current` plus the attribute name, with the *first letter capitalized*.

For example, the Carbon Monoxide Detector capability has an attribute “carbonMonoxide”. To get the latest value for this attribute, you would call:

```
def currentCarbon = somedevice.currentCarbonMonoxide
```

Signature: `Object current<Uppercase attribute name>`

Returns: **Object** - the latest reported values for the specified attribute. The specific type of object returned will vary depending on the specific attribute.

Tip: The exact returned type for various attributes depends upon the underlying capability and Device Handler.

Example:

```
preferences {
    section() {
        input "theswitch", "capability.switch"
        input "thetemp", "capability.temperatureMeasurement"
    }
}
...
def switchattr = theswitch.currentSwitch
def tempattr = thetemp.currentTemperature

log.debug "current switch: $switchattr"
log.debug "current temp: $tempattr"

// switch attribute returned as a string
log.debug "switchattr instanceof String? ${switchattr instanceof String}"

// temperature attribute returned as a Number
log.debug "tempattr instanceof Number? ${tempattr instanceof Number}"
...
```

11.6.4 capabilities

The List of Capabilities provided by this Device.

Signature: `List<Capability> capabilities`

Returns: `List<Capability>` (page 300) - a List of Capabilities supported by this Device.

Example:

```
def supportedCaps = somedevice.capabilites
supportedCaps.each {cap ->
    log.debug "This device supports the ${cap.name} capability"
}
```

11.6.5 currentState()

Gets the latest *State* (page 329) for the specified attribute.

Signature: `State currentState(String attributeName)`

Parameters: `String attributeName` - The name of the attribute to get the State for.

Returns: *State* (page 329) - The latest State instance for the specified attribute.

Example:

```

preferences {
    section() {
        input "temp", "capability.temperatureMeasurement"
    }
}
...
def tempState = temp.currentState("temperature")
log.debug "state value: ${tempState.value}"
...

```

11.6.6 currentValue()

Gets the latest reported values of the specified attribute.

Signature: Object currentValue(String attributeName)

Parameters: String attributeName - The name of the attribute to get the latest values for.

Returns: Object - The latest reported values of the specified attribute. The exact return type will vary depending upon the attribute.

Warning: The exact returned type for various attributes is not adequately documented at this time. Until they are, we recommend that you save often and experiment, or even look at the specific Device Handler for the device you are working with.

Example:

```

preferences {
    section() {
        input "theswitch", "capability.switch"
        input "thetemp", "capability.temperatureMeasurement"
    }
}
...
def switchattr = theswitch.currentValue("switch")
def tempattr = thetemp.currentValue("temperature")

log.debug "current switch: $switchattr"
log.debug "current temp: $tempattr"

// switch attribute returned as a String
log.debug "switchattr instanceof String? ${switchattr instanceof String}"

// temperature attribute returned as a Number
log.debug "tempattr instanceof Number? ${tempattr instanceof Number}"
...

```

11.6.7 displayName

The label of the Device assigned by the user.

Signature: String label

Returns: `String` - the label of the Device assigned by the user, `null` if no label is set.

Example:

```
def devLabel = someDevice.displayName
if (devLabel) {
    log.debug "label set by user: $devLabel"
} else {
    log.debug "no label set by user for this device"
}
```

11.6.8 id

The unique system identifier for this Device.

Signature: `String id`

Returns: `String` - the unique system identifier for this Device.

11.6.9 events()

Get a list of Events for the Device in reverse chronological order (newest first).

Note: Only Events in the last seven days will be returned via the `events()` method.

Signature: `List<Event> events([max: N])`

Parameters: `Map` options (*optional*) - Options for the query. Supported options below:

option	Type	Description
<code>max</code>	<code>Number</code>	The maximum number of Events to return.

Returns: `List<Event>` (page 315) - A list of events in reverse chronological order (newest first).

Example:

```
def theEvents = someDevice.events()
def mostRecent20Events = someDevice.events(max: 20)
```

11.6.10 eventsBetween()

Get a list of Events between the specified start and end dates.

Note: Only Events from the *last seven days* is query-able. Using a date range that ends more than seven days ago will return zero events.

Signature: `List<Event> eventsBetween(Date startDate, Date endDate [, Map options])`

Parameters: `Date` startDate - the lower Date range for the query.

`Date` endDate - the upper Date range for the query.

`Map` options (*optional*) - Options for the query. Supported options below:

option	Type	Description
max	Number	The maximum number of Events to return.

Returns: `List` <:ref:event_ref> - a list of Events between the specified start and end dates.

Example:

```
// 3 days ago
def startDate = new Date() - 3

// today
def endDate = new Date()

def theEvents = somedevice.eventsBetween(startDate, endDate)
log.debug "there were ${theEvents.size()} events in the last three days"

// events in the last 3 days - maximum of 5 events
def limitedEvents = somedevice.eventsBetween(startDate, endDate, [max: 5])
```

11.6.11 eventsSince()

Get a list of Events since the specified date.

Note: Only Events from the *last seven days* is query-able. Using a date range that ends more than seven days ago will return zero events.

Signature: `List<Event> eventsSince(Date startDate [, Map options])`

Parameters: `Date` startDate - the date to start the query from.

`Map` options (*optional*) - options for the query. Supported options below:

option	Type	Description
max	Number	The maximum number of Events to return.

Returns: `List` <*Event* (page 315)> - a list of Events since the specified date.

Example:

```
def eventsSinceYesterday = somedevice.eventsSince(new Date() - 1)
log.debug "there have been ${eventsSinceYesterday.size()} since yesterday"
```

11.6.12 hasAttribute()

Determine if this Device has the specified attribute.

Tip: Attribute names are case-sensitive.

Signature: `Boolean hasAttribute(String attributeName)`

Parameters: `String` `attributeName` - the name of the attribute to check if the Device supports.

Returns: `Boolean` - `true` if this Device has the specified attribute. Returns a non-true value if not (may be `null`).

Example:

```
preferences {
    section() {
        input "theswitch", "capability.switch"
        input "thetemp", "capability.temperatureMeasurement"
    }
}
...
def hasTempAttr = thetemp.hasAttribute("temperature")
// true, since this device supports the 'temperature' capability
log.debug "${thetemp.displayName} has temperature attribute? $hasTempAttr"

def hasTempAttrCaseSensitive = thetemp.hasAttribute("Temperature")
if (hasTempAttrCaseSensitive) {
    log.debug "${thetemp.displayName} supports the Temperature attribute."
} else {
    // this block will execute, since attribute names are case sensitive
    log.debug "${thetemp.displayName} does NOT support the Temperature attribute."
}
...
```

11.6.13 hasCapability()

Determine if this Device supports the specified capability name.

Tip: Capability names are case-sensitive.

Signature: `Boolean hasCapability(String capabilityName)`

Parameters: `String` `capabilityName` - the name of the capability to check if the Device supports.

Returns: `Boolean` - `true` if this Device has the specified capability. Returns a non-true value if not (may be `null`).

Example:

```
preferences {
    section() {
        input "theswitch", "capability.switch"
        input "thetemp", "capability.temperatureMeasurement"
    }
}
...
def hasSwitch = theswitch.hasCapability("Switch")
def hasSwitchCaseSensitive = theswitch.hasCapability("switch")
def hasPower = theswitch.hasCapability("Power")

// true
log.debug "${theswitch.displayName} has Switch capability? $hasSwitch"

if (!hasSwitchCaseSensitive) {
    // enters this block (names are case-sensitive!)
```

```

    log.debug "${theswitch.displayName} does not have the switch capability"
}

// true
log.debug "${theswitch.displayName} also has Power capability? $multiCapabilities"

...

```

11.6.14 hasCommand()

Determine if this Device has the specified command name.

Tip: Command names are case-sensitive.

Signature: Boolean hasCommand(String commandName)

Parameters: String commandName - the name of the command to check if the Device supports.

Returns: Boolean - true if this Device has the specified command. Returns a non-true value if not (may be null).

Example:

```

preferences {
    section() {
        input "theswitch", "capability.switch"
        input "switchlevel", "capability.switchLevel"
    }
}

...

def hasOn = theswitch.hasCommand("on")
def hasOnCaseSensitive = theswitch.hasCommand("On")

// true
log.debug "${theswitch.displayName} has on command? $hasOn"

if (!hasOnCaseSensitive) {
    // enters this block - case-sensitive!
    log.debug "${theswitch.displayName} does not have On command"
}

def hasSetLevelCommand = switchlevel.hasCommand("setLevel")
// true
log.debug "${switchlevel.displayName} has command setLevel? $hasSetLevelCommand"

...

```

11.6.15 latestState()

Get the latest Device State record for the specified attribute.

Signature: State latestState(String attributeName)

Parameters: String attributeName - The name of the attribute to get the State record for.

Returns: *State* (page 329) - The latest State record for the attribute specified for this Device.

Example:

```
def latestDeviceState = somedevice.latestState("someAttribute")
log.debug "latest state value: ${latestDeviceState.value}"
```

11.6.16 latestValue()

Get the latest reported value for the specified attribute.

Signature: `Object latestValue(String attributeName)`

Parameters: *String* attributeName - the name of the attribute to get the latest value for.

Returns: *Object* - the latest reported value. The exact type returned will vary depending upon the attribute.

Warning: The exact returned type for various attributes is not adequately documented at this time. Until they are, we recommend that you save often and experiment, or even look at the specific Device Handler for the device you are working with.

Example:

```
preferences {
    section() {
        input "theswitch", "capability.switch"
        input "thetemp", "capability.temperatureMeasurement"
    }
}
...
def switchattr = theswitch.latestValue("switch")
def tempattr = thetemp.latestValue("temperature")

log.debug "current switch: $switchattr"
log.debug "current temp: $tempattr"

// switch attribute returned as a String
log.debug "switchattr instanceof String? ${switchattr instanceof String}"

// temperature attribute returned as a Number
log.debug "tempattr instanceof Number? ${tempattr instanceof Number}"

...
```

11.6.17 name

The internal name of the Device. Typically assigned by the system and editable only by a user in the IDE.

Signature: `String name`

Returns: *String* - the internal name of the Device.

11.6.18 label

The name of the Device set by the user in the mobile application or Web IDE.

Signature: `String label`

Returns: `String` - the name of the Device as configured by the user.

11.6.19 statesBetween()

Get a list of Device [State](#) (page 329) objects for the specified attribute between the specified times in reverse chronological order (newest first).

Note: Only State instances from the *last seven days* is query-able. Using a date range that ends more than seven days ago will return zero State objects.

Signature: `List<State> statesBetween(String attributeName, Date startDate, Date endDate [, Map options])`

Parameters: `String` `attributeName` - The name of the attribute to get the States for.

`Date` `startDate` - The beginning date for the query.

`Date` `endDate` - The end date for the query.

`Map` `options` (*optional*) - options for the query. Supported options below:

option	Type	Description
<code>max</code>	<code>Number</code>	The maximum number of Events to return.

Returns: `List <State` (page 329)> - A list of State objects between the dates specified. A maximum of 1000 [State](#) (page 329) objects will be returned.

Example:

```
preferences {
    section() {
        input "theswitch", "capability.switch"
    }
}
...
def start = new Date() - 5
def end = new Date() - 1

def theStates = theswitch.statesBetween("switch", start, end)
log.debug "There are ${theStates.size()} between five days ago and yesterday"
...
```

11.6.20 statesSince()

Get a list of Device [State](#) (page 329) objects for the specified attribute since the date specified.

Note: Only State instances from the *last seven days* is query-able. Using a date range that ends more than seven days ago will return zero State objects.

Signature: `List<State> statesSince(String attributeName, Date startDate [, Map options])`

Parameters: `String attributeName` - The name of the attribute to get the States for.

`Date startDate` - The beginning date for the query.

`Map options` (*optional*) - options for the query. Supported options below:

option	Type	Description
max	Number	The maximum number of Events to return.

Returns: `List<State>` (page 329) - A list of State records since the specified start date. A maximum of 1000 `State` (page 329) instances will be returned.

Example:

```
preferences {
    section() {
        input "theswitch", "capability.switch"
    }
}
...
def theStates = theswitch.statesBetween("switch", new Date() -3)
log.debug "There are ${theStates.size()} State records in the last 3 days"
...
```

11.6.21 supportedAttributes

The list of `Attribute` (page 299) s for this Device.

Signature: `List<Attribute> supportedAttributes`

Returns: `List<Attribute>` (page 299) - the list of Attributes for this Device. Includes both capability attributes as well as Device-specific attributes.

Example:

```
preferences {
    section() {
        input "theswitch", "capability.switch"
    }
}
...
def theAtts = theswitch.supportedAttributes
theAtts.each {att ->
    log.debug "Supported Attribute: ${att.name}"
}
...
```

11.6.22 supportedCommands

The list of `Command` (page 302) s for this Device.

Signature: `List<Command> supportedCommands`

Returns: `List <Command>` (page 302) - the list of Commands for this Device. Includes both capability commands as well as Device-specific commands.

Example:

```
preferences {
    section() {
        input "theswitch", "capability.switch"
    }
}
...
def theCommands = theswitch.supportedCommands
theCommands.each {com ->
    log.debug "Supported Command: ${com.name}"
}
...
```

11.7 Event

Events are core to the SmartThings platform. They allow SmartApps to respond to changes in the physical environment, and build automations around them.

Event instances are not created directly by SmartApp or Device Handlers. They are created internally by the SmartThings platform, and passed to SmartApp event handlers that have subscribed to those events.

Note: In a SmartApp or Device Handler, the method `createEvent` exists to create a Map that defines properties of an Event. Only by returning the resulting map from a Device Handler's `parse` method is an actual Event instance created and propagated through the SmartThings system.

The reference documentation here lists all properties and methods available on an Event object instance.

11.7.1 date

Acquisition time of this device state record.

Signature: `Date date`

Returns: `Date` - the date and time this event record was created.

Example:

```
def eventHandler(evt) {
    log.debug "event created at: ${evt.date}"
}
```

11.7.2 id

The unique system identifier for this event.

Signature: `String id`

Returns: `String` - the unique device identifier for this event.

Example:

```
def eventHandler(evt) {  
    log.debug "event id: ${evt.id}"  
}
```

11.7.3 dateValue

The value of the event as a `Date` object, if applicable.

Signature: `Date dateValue`

Returns: `Date` - If the value of this event is date, a `Date` will be returned. `null` will be returned if the value of the event is not parseable to a `Date`.

Warning: Calling `dateValue` on an Event that does not have a value that is parseable into a `Date` object will throw an exception.
You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {  
    // get the value of this event as a Date  
    // throws an exception if the value is not convertible to a Date  
    try {  
        log.debug "The dateValue of this event is ${evt.dateValue}"  
        log.debug "evt.dateValue instanceof Date? ${evt.dateValue instanceof Date}"  
    } catch (e) {  
        log.debug "Trying to get the dateValue for ${evt.name} threw an exception: $e"  
    }  
}
```

11.7.4 description

The raw description that generated this Event.

Signature: `String description`

Returns: `String` - the raw description that generated this Event.

Example:

```
def eventHandler(evt) {  
    log.debug "event raw description: ${evt.description}"  
}
```

11.7.5 descriptionText

The description of the event that is to be displayed to the user in the mobile application.

Signature: `String descriptionText`

Returns: `String` - the description of this event to be displayed to the user in the mobile application.

Example:

```
def eventHandler(evt) {  
    log.debug "event description text: ${evt.descriptionText}"  
}
```

11.7.6 device

The *Device* (page 304) associated with this Event.

Signature: `Device device`

Returns: *Device* (page 304) - the Device associated with this Event, or null if no Device is associated with this Event.

11.7.7 displayName

Signature: `String displayName`

Returns: `String` - The user-friendly name of the source of this event. Typically the user-assigned device label.

Example:

```
def eventHandler(evt) {  
    log.debug "event display name: ${evt.displayName}"  
}
```

11.7.8 deviceId

The unique system identifier of the *Device* (page 304) associated with this Event.

Signature: `String deviceId`

Returns: `String` - the unique system identifier of the device associated with this Event, or null if there is no device associated with this Event.

Example:

```
def eventHandler(evt) {  
    log.debug "The device id for this event: ${evt.deviceId}"  
}
```

11.7.9 doubleValue

The value of this Event, if the value can be parsed to a Double.

Signature: Double doubleValue

Returns: Double - the value of this Event as a Double.

Warning: doubleValue will throw an Exception if the value of the event is not parseable to a Double. You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {  
    // get the value of this event as an Double  
    // throws an exception if the value is not convertible to a Double  
    try {  
        log.debug "The doubleValue of this event is ${evt.doubleValue}"  
        log.debug "evt.doubleValue instanceof Double? ${evt.doubleValue instanceof Double}"  
    } catch (e) {  
        log.debug "Trying to get the doubleValue for ${evt.name} threw an exception: $e"  
    }  
}
```

11.7.10 floatValue

The value of this Event as a Float, if it can be parsed into a Float.

Signature: Float floatValue

Returns: Float - the value of this Event as a Float.

Warning: floatValue will throw an Exception if the Event's value is not parseable to a Float. You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {  
    // get the value of this event as an Float  
    // throws an exception if not convertible to Float  
    try {  
        log.debug "The floatValue of this event is ${evt.floatValue}"  
        log.debug "evt.floatValue instanceof Float? ${evt.floatValue instanceof Float}"  
    } catch (e) {  
        log.debug "Trying to get the floatValue for ${evt.name} threw an exception: $e"  
    }  
}
```

11.7.11 hubId

The unique system identifier of the Hub associated with this Event.

Signature: String hubId

Returns: *String* - the unique system identifier of the Hub associated with this Event, or null if no Hub is associated with this Event.

Example:

```
def eventHandler(evt) {  
    log.debug "The hub id associated with this event: ${evt.hubId}"  
}
```

11.7.12 installedSmartAppId

The unique system identifier of the SmartApp instance associated with this Event.

Signature: *String* installedSmartApp

Returns: *String* - the unique system identifier of the SmartApp instance associated with this Event.

Example:

```
def eventHandler(evt) {  
    log.debug "The installed SmartApp id associated with this event: ${evt.installedSmartAppId}"  
}
```

11.7.13 integerValue

The value of this Event as an Integer.

Signature: *Integer* integerValue

Returns: *Integer* - the value of this Event as an Integer.

Warning: integerValue throws an Exception if the Event value cannot be parsed to an Integer. You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {  
    // get the value of this event as an Integer  
    // throws an exception if not convertible to Integer  
    try {  
        log.debug "The integerValue of this event is ${evt.integerValue}"  
        log.debug "The integerValue of this event is an Integer: ${evt.integerValue instanceof Integer}"  
    } catch (e) {  
        log.debug "Trying to get the integerValue for ${evt.name} threw an exception: $e"  
    }  
}
```

11.7.14 isDigital()

true if the Event is from the digital actuation (non-physical) of a Device, false otherwise.

Signature: Boolean physical()

Returns: Boolean - true if the Event is from the digital actuation of a Device, false otherwise.

Example:

```
def eventHandler(evt) {  
    log.debug "event from digital actuation? ${evt.isDigital()}"  
}
```

11.7.15 isoDate

Acquisition time of this Event as an ISO-8601 String.

Signature: String isoDate

Returns: String - The acquisition time of this Event as an ISO-8601 String.

Example:

```
def eventHandler(evt) {  
    log.debug "event isoDate: ${evt.isoDate}"  
}
```

11.7.16 isPhysical()

true if the Event is from the physical actuation of a Device, false otherwise.

Signature: Boolean physical()

Returns: Boolean - true if the Event is from the physical actuation of a Device, false otherwise.

Example:

```
def eventHandler(evt) {  
    log.debug "event from physical actuation? ${evt.isPhysical()}"  
}
```

11.7.17 isStateChange()

true if the Attribute value for this Event is different than the previous one.

Signature: Boolean stateChange()

Returns: Boolean - true if the Attribute value for this Event is different than the previous one.

Example:


```
def eventHandler(evt) {  
    log.debug "Is this event a state change? ${evt.isStateChange()}"  
}
```

11.7.18 jsonValue

Value of the Event as a parsed JSON data structure.

Signature: `Object jsonValue`

Returns: `Object` - The value of the Event as a JSON structure

Warning: `jsonValue` throws an Exception if the value of the Event cannot be parsed into a JSON object. You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {  
    // get the value of this event as a JSON structure  
    // throws an exception if the value is not convertible to JSON  
    try {  
        log.debug "The jsonValue of this event is ${evt.jsonValue}"  
    } catch (e) {  
        log.debug "Trying to get the jsonValue for ${evt.name} threw an exception: $e"  
    }  
}
```

11.7.19 linkText

Warning: Deprecated.
Using the `linkText` property is deprecated. Use *displayName* (page 317) instead.

The user-friendly name of the source of this event. Typically the user-assigned device label.

11.7.20 location

The Location associated with this Event.

Signature: `Location location`

Returns: *Location* (page 325) - The Location associated with this Event, or `null` if no Location is associated with this Event.

11.7.21 locationId

The unique system identifier for the *Location* (page 325) associated with this Event.

Signature: String locationId

Returns: String - the unique system identifier for the *Location* (page 325) associated with this Event.

11.7.22 longValue

The value of this Event as a Long.

Signature: Long longValue

Returns: Long - the value of this Event as a Long.

Warning: longValue throws an Exception if the value of the Event cannot be parsed to a Long. You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {  
    // get the value of this event as an Long  
    // throws an exception if not convertible to Long  
    try {  
        def evtLongValue = evt.longValue  
        log.debug "The longValue of this event is evtLongValue"  
        log.debug "evt.longValue instanceof Long? ${evtLongValue instanceof Long}"  
    } catch (e) {  
        log.debug "Trying to get the longValue for ${evt.name} threw an exception: $e"  
    }  
}
```

11.7.23 name

The name of this Event.

Signature: String name

Returns: String - the name of this event.

Example:

```
def eventHandler(evt) {  
    log.debug "the name of this event: ${evt.name}"  
}
```

11.7.24 numberValue

The value of this Event as a Number.

Signature: BigDecimal numberValue

Returns: [BigDecimal](#) - the value of this event as a BigDecimal.

Warning: numberValue throws an Exception if the value of the Event cannot be parsed to a BigDecimal. You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {
    // get the value of this event as an Number
    // throws an exception if the value is not convertible to a Number
    try {
        def evtNumberValue = evt.numberValue
        log.debug "The numberValue of this event is ${evtNumberValue}"
        log.debug "evt.numberValue instanceof BigDecimal? ${evtNumberValue instanceof BigDecimal}"
    } catch (e) {
        log.debug "Trying to get the numberValue for ${evt.name} threw an exception: $e"
    }
}
```

11.7.25 numericValue

The value of this Event as a Number.

Signature: BigDecimal numericValue

Returns: [BigDecimal](#) - the value of this event as a BigDecimal.

Warning: numericValue throws an Exception if the value of the Event cannot be parsed to a BigDecimal. You should wrap calls in a try/catch block.

Example:

```
def eventHandler(evt) {
    // get the value of this event as an Number
    // throws an exception if the value is not convertible to a Number
    try {
        def evtNumberValue = evt.numericValue
        log.debug "The numericValue of this event is ${evtNumberValue}"
        log.debug "evt.numericValue instanceof BigDecimal? ${evtNumberValue instanceof BigDecimal}"
    } catch (e) {
        log.debug "Trying to get the numericValue for ${evt.name} threw an exception: $e"
    }
}
```

11.7.26 source

The source of the Event.

Signature: `String source`

Returns: `String` - the source of the Event. The following table lists the possible sources and their meaning:

Source	Description
“APP”	Event originated by an app touch event in the mobile application.
“APP_COMMAND”	Event originated by using the mobile application (for example, using the mobile application to turn a light off)
“COMMAND”	Event originated by a SmartApp or Device Handler calling a command on a device.
“DEVICE”	Event originated by the physical actuation of a device.
“HUB”	Event originated on the hub.
“LOCATION”	Event originated by a Location state change (for example, sunrise and sunset events)
“USER”	

Example:

```
def eventHandler(evt) {  
    log.debug "The source of this event is: ${evt.source}"  
}
```

11.7.27 stringValue

The value of this Event as a String.

Signature: `String stringValue`

Returns: `String` - the value of this event as a String.

Example:

```
def eventHandler(evt) {  
    log.debug "The value of this event as a string: ${evt.stringValue}"  
}
```

11.7.28 unit

The unit of measure for this Event, if applicable.

Signature: `String unit`

Returns: `String` - the unit of measure of this Event, if applicable. `null` otherwise.

Example:

```
def eventHandler(evt) { log.debug "The unit for this event: ${evt.unit}"  
}
```

11.7.29 value

The value of this Event as a String.

Signature: String stringValue

Returns: String - the value of this event as a String.

Example:

```
def eventHandler(evt) {
    log.debug "The value of this event as a string: ${evt.value}"
}
```

11.7.30 xyzValue

Value of the event as a 3-entry Map with keys 'x', 'y', and 'z' with BigDecimal values. For example:

```
[x: 1001, y: -23, z: -1021]
```

Typically only useful for getting position data from the “Three Axis” Capability.

Signature: Map<String, BigDecimal> xyzValue

Returns: Map < String , BigDecimal > - A map representing the X, Y, and Z coordinates.

Warning: xyzValue throws an Exception if the value of the Event cannot be parsed to an X-Y-Z data structure. You should wrap calls in a try/catch block.

Example:

```
def positionChangeHandler(evt) {
    // get the value of this event as a 3 entry map with keys
    // 'x', 'y', 'z', and BigDecimal values
    // throws an exception if the value is not convertible to a Date
    try {
        log.debug "The xyzValue of this event is ${evt.xyzValue }"
        log.debug "evt.xyzValue instanceof Map? ${evt.xyzValue  instanceof Map}"
    } catch (e) {
        log.debug "Trying to get the xyzValue for ${evt.name} threw an exception: $e"
    }
}
```

11.8 Location

A Location represents a user’s geo-location, such as “Home” or “office”. Locations do not have to have a SmartThings hub, but generally do.

All SmartApps and Device Handlers are injected with a location property that is the Location into which the SmartApp is installed.

11.8.1 contactBookEnabled

true if this location has contact book enabled (has contacts), false otherwise.

Signature: Boolean contactBookEnabled

Returns: true if this location has contact book enabled (has contacts), false otherwise.

11.8.2 currentMode

The current Mode for the Location.

Signature: Mode currentMode

Returns: *Mode* (page 329) - The current mode for the Location.

Example:

```
log.debug "location.currentMode: ${location.currentMode}"
```

11.8.3 id

The unique internal system identifier for the Location.

Signature: String id

Returns: *String* - the unique internal system identifier for the Location.

Example:

```
log.debug "location.id: ${location.id}"
```

11.8.4 latitude

Geographical latitude of the location. Southern latitudes are negative. Requires that location services are enabled in the mobile app.

Signature: BigDecimal latitude

Returns: *BigDecimal* - the latitude for the Location.

Example:

```
log.debug "location.latitude: ${location.latitude}"
```

11.8.5 longitude

Geographical longitude of the location. Western longitudes are negative. Requires that location services are enabled in the mobile app.

Signature: `BigDecimal longitude`

Returns: `BigDecimal` - the longitude for the Location.

Example:

```
log.debug "location.longitude: ${location.longitude}"
```

11.8.6 mode

The current Mode name for the Location.

Signature: `String mode`

Returns: `String` - the name of the current Mode for the Location.

Example:

```
log.debug "location mode name: ${location.mode}"
```

11.8.7 modes

List of Modes for the Location.

Signature: `List<Mode> modes`

Returns: `List<Mode>` (page 329) - the List of Modes for the Location.

Example:

```
log.debug "Modes for this location: ${location.modes}"
```

11.8.8 name

The name of the Location, as assigned by the user.

Signature: `String name`

Returns: `String` - the name of the Location as assigned by the user.

Example:

```
log.debug "The name of this location is: ${location.name}"
```

11.8.9 setMode()

Set the mode for this location.

Signature: `void setMode(String mode) void setMode(Mode mode)`

Returns: `void`

Warning: `setMode()` will raise an error if the specified mode does not exist for the location. You should verify the mode exists as in the example below.

Example:

```
def modeToSetTo = "Home"
if (location.modes?.find {it.name == modeToSetTo}) {
    location.setMode("Home")
}
```

11.8.10 temperatureScale

The temperature scale (“F” for fahrenheit, “C” for celsius) for this location.

Signature: `String temperatureScale`

Returns: `String` - the temperature scale set for this location. Either “F” for fahrenheit or “C” for celsius.

Example:

```
def tempScale = location.temperatureScale
log.debug "Temperature scale for this location is $tempScale"
```

11.8.11 timeZone

The time zone for the Location. Requires that location services are enabled in the mobile application.

Signature: `TimeZone timeZone`

Returns: `TimeZone` - the time zone for the Location.

Example:

```
log.debug "The time zone for this location is: ${location.timeZone}"
```

11.8.12 zipCode

The ZIP code for the Location, if in the USA. Requires that location services be enabled in the mobile application.

Signature: `String zipCode`

Returns: `String` - the ZIP code for the Location.

Example:

```
log.debug "The zip code for this location: ${location.zipCode}"
```

11.9 Mode

Modes can be thought of as behavior filters for your home. Users want to change how things act or behave in thier home based on the mode you're in.

SmartThings developers cannot create a new Mode. The most common way to interact with a Mode instance is by using the [Location](#) (page 325) to get Mode information:

```
// Get the current Mode
def curMode = location.currentMode

// Get a list of all Modes for this location
def allModesForLocation = location.modes
```

11.9.1 id

The unique internal system identifier of the Mode.

Signature: String id

Returns: String - the unique internal system identifier for the Mode.

```
def curMode = location.currentMode
log.debug "The current mode ID is: ${curMode.id}"
```

11.9.2 name

The name of the Mode.

Signature: String name

Returns: String - the name of the Mode, usually assigned by the user.

Example:

```
def curMode = location.currentMode
log.debug "The current mode name is: ${curMode.name}"
```

11.10 State

A State object encapsulates information about a particular [Attribute](#) (page 299) at a particular moment in time.

State objects are associated with a [Device](#) (page 304) - a Device may have zero-to-many [Attribute](#) (page 299) s, and an Attribute has zero-to-many associated State records.

Refer to the Devices section of the SmartApp Guide for more information about the relationship between Devices, Attributes, and State.

A few ways to get a State object instance from a device (See the [Device](#) (page 304) API reference for detailed information):

```
preferences {
    section() {
        input "thecontact", "capability.contactSensor"
    }
}
...
// <device>.<attributeName>State
def latestState = thecontact.contactState

// <device>.currentState(<attributeName>)
def latestState2 = thecontact.currentState("contact")

// get a list of states between two dates
def recentStates = thecontact.statesBetween(new Date() - 5, new Date())
```

11.10.1 date

The date and time the State object was created.

Signature: Date date

Returns: Date - the Date this State object was created.

Example:

```
def stateDate = contactSensor?.currentState("contact").date
```

11.10.2 dateValue

The value of the underlying attribute as a Date.

Signature: Date dateValue

Returns: Date - the value if the underlying attribute as a Date. Returns null if the attribute value cannot be parsed into a Date.

11.10.3 doubleValue

The value of the underlying Attribute as a Double.

Signature: Double doubleValue

Returns: Double - the value of the underlying attribute as a Double.

Warning: doubleValue throws an Exception if the underlying attribute value cannot be parsed into a Double. You should wrap calls in a try/catch block.

Example:

```
try {
  def latestStateAsDouble = someDevice.currentState("someAttribute").doubleValue
  log.debug "latestStateAsDouble: $latestStateAsDouble"
} catch (e) {
  log.debug "caught exception trying to get double for state record"
}
```

11.10.4 floatValue

The value of the underlying Attribute as a Float.

Signature: Float floatValue

Returns: Float - the value of the underlying Attribute as a Float.

Warning: doubleValue throws an Exception if the underlying attribute value cannot be parsed into a Double. You should wrap calls in a try/catch block.

Example:

```
try {
  def latestStateAsFloat = someDevice.currentState("someAttribute").floatValue
  log.debug "latestStateAsFloat: $latestStateAsFloat"
} catch (e) {
  log.debug "caught exception trying to get floatValue for state record"
}
```

11.10.5 id

The unique system identifier for the State object.

Signature: String id

Returns: String - the unique system identifier for the State object.

Example:

```
def latestState = someDevice.currentState("someAttribute")
log.debug "latest state id: ${latestState.id}"
```

11.10.6 integerValue

The value of the underlying Attribute as an Integer.

Signature: Integer floatValue

Returns: Integer - the value of the underlying Attribute as a Integer.

Warning: `integerValue` throws an Exception if the underlying attribute value cannot be parsed into a Integer. You should wrap calls in a try/catch block.

Example:

```
try {
    def latestStateAsInt = someDevice.currentState("someAttribute").integerValue
    log.debug "latestStateAsInt: $latestStateAsInt"
} catch (e) {
    log.debug "caught exception trying to get integerValue for state record"
}
```

11.10.7 isoDate

The acquisition time of this State object as an ISO-8601 String

Signature: `String isoDate`

Returns: `String` - the time this Sate object was created as an ISO-8601 Strring

Example:

```
def latestState = someDevice.currentState("someAttribute")
log.debug "latest state isoDate: ${latestState.isoDate}"
```

11.10.8 jsonValue

Value of the underlying Attribute parsed into a JSON data structure.

Signature: `Object jsonValue`

Returns: `Object` - the value if the underlying Attribute parsed into a JSON data structure.

Warning: `jsonValue` throws an Exception of the underlying attribute value cannot be parsed into a Integer. You should wrap calls in a try/catch block.

Example:

```
try {
    def latestStateAsJSONValue = someDevice.currentState("someAttribute").jsonValue
    log.debug "latestStateAsJSONValue: $latestStateAsJSONValue"
} catch (e) {
    log.debug "caught exception trying to get jsonValue for state record"
}
```

11.10.9 longValue

The value of the underlying Attribute as a Long.

Signature: Long longValue

Returns: Long - the value if the underlying Attribute as a Long.

Warning: longValue throws an Exception of the underlying attribute value cannot be parsed into a Long. You should wrap calls in a try/catch block.

Example:

```
try {
    def latestStateAsLong = someDevice.currentState("someAttribute").longValue
    log.debug "latestStateAsLong: $latestStateAsLong"
} catch (e) {
    log.debug "caught exception trying to get longValue for state record"
}
```

11.10.10 name

The name of the underlying Attribute.

Signature: String name

Returns: String - the name of the underlying Attribute.

Example:

```
def latest = contactSensor.currentState("contact")
log.debug "name: ${latest.name}"
```

11.10.11 numberValue

The value of the underlying Attribute as a BigDecimal.

Signature: BigDecimal numberValue

Returns: BigDecimal - the value if the underlying Attribute as a BigDecimal.

Warning: numberValue throws an Exception of the underlying attribute value cannot be parsed into a BigDecimal. You should wrap calls in a try/catch block.

Example:

```
try {
    def latestStateAsNumber = someDevice.currentState("someAttribute").numberValue
    log.debug "latestStateAsNumber: $latestStateAsNumber"
} catch (e) {
    log.debug "caught exception trying to get numberValue for state record"
}
```

11.10.12 numericValue

The value of the underlying Attribute as a BigDecimal.

Signature: BigDecimal numericValue

Returns: BigDecimal - the value if the underlying Attribute as a BigDecimal.

Warning: numericValue throws an Exception if the underlying attribute value cannot be parsed into a BigDecimal.
You should wrap calls in a try/catch block.

Example:

```
try {
    def latestStateAsNumber = someDevice.currentState("someAttribute").numericValue
    log.debug "latestStateAsNumber: $latestStateAsNumber"
} catch (e) {
    log.debug "caught exception trying to get numericValue for state record"
}
```

11.10.13 stringValue

The value of the underlying Attribute as a String

Signature: String stringValue

Returns: String - the value of the underlying Attribute as a String.

Example:

```
def latest = contactSensor.currentState("contact")
log.debug "stringValue: ${latest.stringValue}"
```

11.10.14 unit

The unit of measure for the underlying Attribute.

Signature: String unit

Returns: String - the unit of measure for the underlying Attribute, if applicable, null otherwise.

Example:

```
def latest = tempSensor.currentState("temperature")
log.debug "unit: ${latest.unit}"
```

11.10.15 value

The value of the underlying Attribute as a String

Signature: String value

Returns: String - the value of the underlying Attribute as a String.

Example:

```
def latest = contactSensor.currentState("contact")
log.debug "stringValue: ${latest.value}"
```

11.10.16 xyzValue

Value of the underlying Attribute as a 3-entry Map with keys 'x', 'y', and 'z' with BigDecimal values. For example:

```
[x: 1001, y: -23, z: -1021]
```

Typically only useful for getting position data from the “Three Axis” Capability.

Signature: Map<String, BigDecimal> xyzValue

Returns: Map < String , BigDecimal > - A map representing the X, Y, and Z coordinates.

Warning: xyzValue throws an Exception if the value of the Event cannot be parsed to an X-Y-Z data structure. You should wrap calls in a try/catch block.

Example:

```
def latest = threeAxisDevice.currentState("threeAxis")

// get the value of this event as a 3 entry map with keys
// 'x', 'y', 'z', and BigDecimal values
// throws an exception if the value is not convertible to a Date
try {
    log.debug "The xyzValue of this event is ${latest.xyzValue}"
    log.debug "latest.xyzValue instanceof Map? ${latest.xyzValue instanceof Map}"
} catch (e) {
    log.debug "Trying to get the xyzValue threw an exception: $e"
}
```

11.11 Z-Wave Reference

You can find the reference documentation for the Z-Wave library [here](#) (requires login).