Synse Plugin SDK Documentation

Release 0.4.0

Vapor IO

Oct 30, 2018

Contents

1 User Guide **3**
- 1.1 Architecture . . . 3
- 1.2 Basic Usage . . . 5
- 1.3 Advanced Usage . . . 9
- 1.4 Plugin Configuration . . . 13
- 1.5 Tutorial . . . 21

2 Community Guide **35**
- 2.1 Community Plugins . . . 35
- 2.2 License . . . 36
- 2.3 Contributing . . . 36
- 2.4 Release Process . . . 36

3 Development **39**
- 3.1 Developer Setup . . . 39
- 3.2 Testing . . . 40

The Synse Plugin SDK is the official SDK used to write plugins for Synse Server in order to provide support for additional devices. Synse Server provides an HTTP API for monitoring and controlling physical and virtual devices, but it is the backing plugins that provide the support (telemetry and control) for all of the devices that Synse Server exposes.

The SDK handles most of the common functionality needed for plugins, such as configuration parsing, background read/write, transaction generation and tracking, meta-info caching, and more. This means the plugin author should only need to worry about the plugin-specific device support.

CHAPTER 1

User Guide

The official guide for using the Synse Plugin SDK. This section goes over some of the SDK basics and provides a verbose tutorial on how to build a simple plugin. With this information, along with the GoDoc and example plugins, you should be able to make the most out of the Plugin SDK.

1.1 Architecture

This page describes the SDK architecture at a high level and provides a summary of its different components and inner workings.

1.1.1 Overview

The SDK was built to make it easier to develop new plugins. It abstracts away a lot of the internal state handling and the communication layer from the plugin author, so all you have to focus on is is implementing plugin logic – not Synse integration.

At a high level, there are two levels of communication in the SDK. Communication with Synse Server, and communication with the devices it manages.

Plugin Interaction with Synse Server

When an HTTP API request comes in to Synse Server, e.g. a *read* request, that request will have some routing information associated with it (`<rack>/<board>/<device>`). This routing information is used by Synse Server to lookup the device and figure out which plugin owns it.

Once Synse Server knows where the request is going, it sends over all relevant info to the plugin via the Synse gRPC API. The capabilities of this API are summarized below in the *gRPC API* section. The plugin receives the gRPC request and processes it appropriately, returning the corresponding response back to Synse Server.

A plugin can be configured to use either TCP or Unix socket for the gRPC transport protocol.

Plugin Interaction with Devices

When a plugin is run, it will start its "data manager". The data manager will execute reads and writes for devices continuously (on a configurable interval). The read and write behavior is defined by the plugin itself, for each device. The diagram above shows the data flow for reads and writes, starting with an incoming gRPC request from Synse Server.

Reads are executed in a goroutine and the reading values are stored in a local read state cache. When a gRPC read request comes in, it gets the reading out of the cache. This means that plugin readings are not always current (e.g. if the read interval is 60s, then a reading in the cache can be 60s old at most), but with the appropriate read interval, this should be fine. It also means that device reads can happen asynchronously from API reads.

The same holds true for writes. When a gRPC write request comes in, that write transaction is put on the write queue, and at some configurable interval, the plugin will execute those writes.

Other incoming gRPC requests, like *transaction* or *device info*, are not handled by the data manager, since they deal with static information. The handling for these other requests are all built in to the SDK.

1.1.2 gRPC API

The Synse gRPC API lets plugins communicate with Synse Server, and vice versa. Below is a summary of the API methods

Test Checks that the plugin is reachable.

Version Gets the version information of the plugin.

Health Gets the health status of the plugin. A plugin's health status is determined by optional health checks.

Metainfo Get the metadata associated with the plugin. This includes things like the plugin name, maintainer, and a brief description of the plugin.

Capabilities Get the collection of plugin capabilities. This enumerates the different device kinds that a plugin supports, and the reading output types supported by each of those device kinds.

Devices Get the information for all devices registered with the plugin.

Read Read data from a specified device.

Write Write data to a specified device.

Transaction Check the status of a write transaction.

1.1.3 The Data Manager

The Data Manager is a core component of a plugin. While the user should never have to directly interact with the Data Manager, it is still good to know about.

The data manager is in charge of the read goroutine, the write goroutine, and the data that gets passed to and from them. It holds the "read cache" and the "write queue" and manages locking around data access, when necessary.

The data manager supports two run modes:

serial In serial mode, all readings happen serially, all writing happens serially, and the read loop and write loop do not run at the same time.

parallel In parallel mode, readings happen in parallel, writing happens in parallel, and the read loop and write loop can run at the same time.

Reading and writing happens in separate loops, and more specifically, in separate goroutines altogether. This is done to allow different intervals around reading and writing (e.g. you may want your plugin to update quickly – write every 1s, but you may not need to update readings as quickly – read every 30s).

1.1.4 Devices

Within the SDK, a Device represents the physical or virtual thing that the plugin is interfacing with.

The Device model holds the metadata, config information, and a reference to its DeviceHandler, which defines how it will be read from/written to.

1.1.5 Readings

A Reading describes a single data point read from a device. It consists of the reading type, the reading value, and the time at which the reading was taken.

When generating new readings within a Device's read handler, the timestamp should follow the RFC3339Nano format, which is the standard time format for plugins and Synse Server. Built-in helpers, such as `NewReading` or `Output.MakeReading`, will provide a properly formatted timestamp.

1.2 Basic Usage

This page describes some of basic features of the SDK and provides an example of a simple plugin. See the *Advanced Usage* page for an overview of some of the more advanced features of the plugin SDK.

1.2.1 Creating a Plugin

Creating a new plugin is as simple as:

```
import (
    "log"
    "github.com/vapor-ware/synse-sdk/sdk"
)

func main() {
    plugin := sdk.NewPlugin()

    if err := plugin.Run(); err != nil {
        log.Fatal(err)
    }
}
```

This creates a new Plugin instance, but doesn't do much more than that. It is always advised to use `sdk.NewPlugin` to create your plugin instance. The plugin should always be run via `plugin.Run()`.

1.2.2 Setting Plugin Metadata

At a minimum, a plugin requires a name. Ideally, a plugin should include more than a name. The current set of plugin metadata includes:

- name

- maintainer
- description
- vcs link
- tag

The Plugin tag is automatically generated from the `name` and `maintainer` info, following the template `{maintainer}/{name}`, where both the maintainer and name fields are lower-cased, dashes (-) are converted to underscored (_), and spaces converted to dashes (-).

The plugin metadata should be set via the `SetPluginMeta` function, e.g.

```
const (
    pluginName       = "example"
    pluginMaintainer = "vaporio"
    pluginDesc       = "example plugin description"
    pluginVcs        = "github.com/foo/bar"
)

func main() {
    sdk.SetPluginMeta(
        pluginName,
        pluginMaintainer,
        pluginDesc,
        pluginVcs,
    )
}
```

1.2.3 Registering Output Types

All plugins will need to define *output types*. An output type is a definition of a device reading output, providing metadata and requirements around the reading. For example, a plugin might have a temperature sensor. The plugin will implement the logic for how to read from a temperature sensor, but that will ultimately resolve to some value. To make sense of that value, we want to associate to an output type to give it context.

```
var Temperature = sdk.OutputType{
    Name: "temperature",
    Precision: 3,
    Unit: sdk.Unit{
        Name: "celsius",
        Symbol: "C",
    },
}
```

With this context, we know that the reading value corresponds to a temperature reading with unit "celsius", and it will be rounded to a precision of 3 decimal places. The name of the output type identifies the type, so it should be unique. It is convention to namespace output types. This allows for multiple similar types to be specified, e.g.

```
var Temperature1 = sdk.OutputType{
    Name: "modelX.temperature",
    Precision: 3,
    Unit: sdk.Unit{
        Name: "celsius",
        Symbol: "C",
    },
```

(continues on next page)

(continued from previous page)

```
}

var Temperature2 = sdk.OutputType{
    Name: "modelY.temperature",
    Precision: 2,
    Unit: sdk.Unit{
        Name: "Kelvin",
        Symbol: "K",
    },
}
```

The namespacing is arbitrary, so it is up to the plugin author to decide what makes the most sense. With OutputTypes defined, they can be registered with the plugin simply:

```
func main() {
    plugin := sdk.NewPlugin()

    err := plugin.RegisterOutputTypes(
        &Temperature1,
        &Temperature2,
    )
}
```

OutputTypes can also be defined via config file, in which case, they will not need to be explicitly registered with the plugin, as seen in the example above. They will be registered when the configs are read in, during the pre-run setup.

1.2.4 Registering Device Handlers

All plugins need to define *device handlers*. A device handler defines how a particular device will be read from/written to, and if it is even capable of reads/writes. There are currently three types of functionality that a device handler can define:

- Read
- Write
- Bulk Read

Read defines how an individual device should be read. *Write* defines how an individual device should bw written to. *Bulk Read* defines read functionality for all devices that use that handler. That is to say, while a *read* happens one at a time, a *bulk read* will read all devices at once. While bulk reads have a more limited use case, they can simplify some device readings, for example, if a give device/protocol requires all registers to be read through to get a single reading (as can be the case for I2C), it can be easier to just do that bulk read once instead of re-doing it for every device on that bus.

Note: If both a "read" function and "bulk read" function are specified for a single device handler, the bulk read will be ignored and the SDK will only use the read function. If bulk read function is desired, make sure that no individual read function is specified.

If no function is specified for any of these, the SDK takes that to mean that the handler does not support that functionality. That is to say, a device handler with only a read function defined implies that those devices cannot be written to.

Defining a handler is as simple as giving it a name, and the appropriate functions:

```
var TemperatureHandler = sdk.DeviceHandler{
    Name: "temperature",
    Read: func(device *sdk.Device) ([]*sdk.Reading, error) {
        ...
    },
}
```

See the GoDoc for more details on how handlers should be defined.

Like DeviceOutputs, a DeviceHandler name identifies that handler, so it should be unique. If necessary, handler names should be namespaced, but the namespacing is arbitrary and left to the plugin to decide. With DeviceHandlers defined, they can be registered with the plugin simply:

```
func main() {
    plugin := sdk.NewPlugin()

    plugin.RegisterDeviceHandlers(
        &TemperatureHandler,
    )
}
```

1.2.5 Creating New Readings

When creating a new reading in a handler's read function, it is highly recommended to use the built-in constructors, as they automatically fill in some fields. In particular, it is important to note that the Synse platform has standardized on RFC3339 timestamp formatting, which the built-in constructors do for you.

One of the easiest ways to create a new reading is with the following pattern. Below, we have some `value`, which is whatever reading we got. The input to `GetOutput` is the name of the output type. If the output type does not exist for the device, this will cause the plugin to panic (in this particular pattern), which is typically desirable, since it is indicative of a mis-configuration in the device configs.

```
var someHandler = sdk.DeviceHandler{
    Name: "example.reader",
    Read: func(device *sdk.Device) ([]*sdk.Reading, error) {
        // plugin-specific read logic
        ...

        return []*sdk.Reading{
            device.GetOutput("example.temperature").MakeReading(value),
        }, nil
    },
}
```

1.2.6 A Complete Example

A complete example of a simple plugin that exercises all of these pieces can be found in the SDK repo's examples/simple_plugin directory.

For a slightly more complex example, see the Emulator Plugin.

1.3 Advanced Usage

This page describes some of the more advanced features of the SDK for plugin development.

1.3.1 Command Line Arguments

The SDK has some built-in command line arguments for plugins. These can be seen by running the plugin with the `--help` flag.

```
$ ./plugin --help
Usage of ./plugin:
  -debug
        run the plugin with debug logging
  -dry-run
        perform a dry run to verify the plugin is functional
  -version
        print plugin version information
```

A plugin can add its own command line args if it needs to as well. This can be done simply by defining the flags that the plugin needs, e.g.

```
import (
    "flag"
)

var customFlag bool

func init() {
    flag.BoolVar(&customFlag, "custom", false, "some custom functionality")
}
```

This flag will be parsed on plugin `Run()`, so it can only be used after the plugin has been run.

1.3.2 Pre Run Actions

Pre Run Actions are actions that the plugin will perform before it starts to run the gRPC server and start the data manager's read/write goroutines. These actions can be used for plugin-wide setup, should a plugin require it. For example, this could be used to perform some kind of authentication, verifying that some backend exists and is reachable, or to do additional config validation, etc.

Pre Run Actions should fulfil the `pluginAction` type and should be registered with the plugin before it is run. An (abridged) example:

```
// preRunAction defines a function that will run before the
// plugin starts its main run logic.
func preRunAction(p *sdk.Plugin) error {
    return backend.VerifyRunning()  // do some action
}

func main() {
    plugin  := sdk.NewPlugin()

    plugin.RegisterPreRunActions(
        preRunAction,
```

(continues on next page)

(continued from previous page)

```
    )
}
```

For more, see the Device Actions Example Plugin.

1.3.3 Post Run Actions

Post Run Actions are actions that the plugin will perform after it is shut down gracefully. A graceful shutdown of a plugin is done by passing the SIGTERM or SIGINT signal to the plugin. These actions can be used for plugin-wide shutdown/cleanup, such as cleaning up state, terminating connections, etc.

Post Run Actions should fulfil the `pluginAction` type and should be registered with the plugin before it is run. An (abridged) example:

```
// postRunAction defines a function that will run after the plugin
// has gracefully terminated.
func postRunAction(p *sdk.Plugin) error {
    return db.closeConnection() // do some action
}

func main() {
    plugin := sdk.NewPlugin()

    plugin.RegisterPostRunActions(
        postRunAction,
    )
}
```

For more, see the Device Actions Example Plugin.

1.3.4 Device Setup Actions

Some devices might need a setup action performed before the plugin starts to read or write to them. As an example, this could be performing some type of authentication, or setting some bit in a register. The action itself is plugin (and protocol) specific and does not matter to the SDK.

Device Setup Actions should fulfil the `deviceAction` type and should be registered with the plugin before it is run.

When a device setup action is registered, it should be registered with a filter. This filter is used to identify which devices the action should apply to. An (abridged) example:

```
// deviceSetupAction defines a function we will use as a
// device setup action.
func deviceSetupAction(p *sdk.Plugin, d *sdk.Device) error {
    return utils.Validate(d) // do some action
}

func main() {
    // Create a new Plugin
    plugin := sdk.NewPlugin()

    // Register the action with all devices that have
    // the type "airflow".
    plugin.RegisterDeviceSetupActions(
```

(continues on next page)

(continued from previous page)

```
        "type=airflow",
        deviceSetupAction,
    )
}
```

For more, see the Device Actions Example Plugin.

1.3.5 Plugin Options

As other sections here describe in more detail, there may be cases where a plugin would want to override some default plugin functionality. As an example, the SDK provides a default device identifier function. What this function does is take the config for a particular device and creates a hash out of that config info in order to create a deterministic ID for the device.

The premise of the ID determinism is that a device config will generally define how to address that device (e.g. for a serial device, it could be the serial bus, channel, etc). If the config changes, we are talking to something different, so we assume that a change in config equates to a change in device identity.

Obviously, this is not always the case, which is where having a custom identifier function becomes useful. If we wanted to only take a subset of the device config, we could define a simple device identifier override function, but in order to register it with the plugin, we'd need to use a Plugin Option.

Plugin Options are passed to the plugin when it is initialized via `sdk.NewPlugin`.

```
// ProtocolIdentifier gets the unique identifiers out of the plugin-specific
// configuration to be used in UID generation.
func ProtocolIdentifier(data map[string]interface{}) string {
    return fmt.Sprint(data["id"])
}

func main() {
    plugin := sdk.NewPlugin(
        sdk.CustomDeviceIdentifier(ProtocolIdentifier),
    )
}
```

An example of this can be found in the Device Actions Example Plugin.

1.3.6 Dynamic Registration

Dynamic Registration is when devices are configured not from config YAML files, but dynamically at runtime. There are two kinds of dynamic registration functions:

- one that creates DeviceConfig(s) (e.g. it creates the configuration for a device)
- one that creates Device(s) (e.g. it creates the device directly)

By default, a plugin will not do any dynamic device registration. In enable dynamic registration for a plugin, the dynamic registration function will have to be defined, and then it will have to be passed to the plugin constructor via a PluginOption.

Dynamic registration can be useful when you do not know what devices may exist at any given time. A good example of this is IPMI. While you should know the BMC IP address, you may not know all the devices on all your BMCs. Even if you do, it would be cumbersome to have to manually enumerate these in a config file.

With device enumeration, you can just create a function that will query the BMC for its devices and then use that response to generate the devices (or the device configs) at runtime.

An extremely simple example of this can be found in the Dynamic Registration Example Plugin.

1.3.7 Configuration Policies

The SDK exposes different configuration policies that a plugin can set to modify its behavior. By default, the policies dictate that

- plugin config is optional (e.g. a plugin can use defaults)
- device config(s) are required (e.g. YAML files must be specified for device configs)
- dynamic device config(s) are optional
- output type config file(s) are optional

For many plugins, the default policies will be good enough. Some plugins may require some explicit configuration, so to enforce it, they can set the appropriate policy. As an example, there could be a hypothetical plugin that will only allow the pre-defined output types, will not allow device configs from file, requires devices to be registered from dynamic registration. The config policies allow that behavior to be enforced, and cause the plugin to terminate if any of the policies are violated.

Below is a table that lists all of the current config policies. There can only be one (or none) policy chosen from each column below at any given time, e.g. you cannot have `PluginConfigFileOptional` and `PluginConfigFileRequired` specified at the same time.

Plugin (File)	Device Config (File)	Device Config (Dynamic)	Output Type Config (File)
PluginConfigFileOptional	DeviceConfigFileOptional	DeviceConfigDynamicOptional	TypeConfigFileOptional
PluginConfigFileRequired	DeviceConfigFileRequired	DeviceConfigDynamicRequired	TypeConfigFileRequired
PluginConfigFileProhibited	DeviceConfigFileProhibited	DeviceConfigDynamicProhibited	TypeConfigFileProhibited

Setting config policies for the plugin is simple:

```
import (
    "github.com/vapor-ware/synse-sdk/sdk"
    "github.com/vapor-ware/synse-sdk/sdk/policies"
)

func main() {
    plugin := sdk.NewPlugin()

    policies.Add(policies.DeviceConfigFileOptional)
    policies.Add(policies.TypeConfigFileOptional)
}
```

An example of this can be found in the Dynamic Registration Example Plugin.

1.3.8 Health Checks

The SDK supports plugin health checks. The health of the plugin derived from these checks is surfaced via the Synse gRPC API, and can be seen via the Synse Server HTTP API.

A health check is just a function that returns an error. When run, if the function returns `nil`, the check passed. If an error is returned, the check has failed. Health checks can be registered and run in different ways, but the SDK only natively supports *periodic* checks currently.

Writing and registering a health check is simple. As an example, we could define a health check that will periodically hit a URL to see if it is reachable:

```
import (
    "github.com/vapor-ware/synse-sdk/sdk"
    "github.com/vapor-ware/synse-sdk/sdk/health"
)

func checkURL() error {
    resp, err := http.Get(someURL)
    if err != nil {
        return err
    }
    if !resp.Ok {
        return fmt.Errorf("Got non-200 response from URL")
    }
    return nil
}

func main() {
    plugin := sdk.NewPlugin()

    health.RegisterPeriodicCheck("example health check", 30*time.Second, checkURL)
}
```

1.3.9 C Backend

Plugins can be written with C backends. In general, this means that the read/write handlers or some related logic is written in C. This feature is not specific to the SDK, but is a feature of Go itself.

For more information on this, see the CGo Documentation and the C Plugin example.

1.4 Plugin Configuration

This page describes the different kinds of configuration a plugin has, and gives examples for each. There are three basic kinds of configuration:

- *Plugin Configuration*: Configuration for how the plugin should behave.
- *Device Configuration*: Configuration for the device instances that the plugin should interface with and manage.
- *Output Type Configuration*: Configuration for the supported reading outputs for the supported devices.

1.4.1 Plugin Configuration

Plugins are configured from a YAML file that defines how the plugin should operate. Most plugin configurations have sane default values, so it may not even be necessary to specify your own plugin configuration.

The plugin config file must be named `config.{yml|yaml}`.

Config Policies

The following config policies relate to plugin configuration.

- PluginConfigFileOptional *(default)*
- PluginConfigFileRequired
- PluginConfigFileProhibited

Config Locations

The default locations for the plugin configuration (in order of evaluation) are:

```
$PWD
$HOME/.synse/plugin
/etc/synse/plugin
```

Where `$PWD` (or `.`) is the directory in which the plugin binary is being run from.

A non-default location can be used by setting the `PLUGIN_CONFIG` environment variable to either the directory containing the config file, or to the config file itself.

```
PLUGIN_CONFIG=/tmp/plugin/config.yml
```

Configuration Options

version The version of the configuration scheme.

```
version: 1.0
```

debug Enables debug logging.

```
debug: true
```

network Network settings for the gRPC server. If this is not specified, it will default to a *type* of tcp with an *address* of localhost:5001.

type The type of networking protocol the gRPC server should use. This should be one of "tcp" or "unix".

```
type: tcp
```

address The network address. For unix socket-based networking, this should be the path to the socket. For tcp, this can be ip/host[:port].

```
address: ":5001"
```

settings Settings for how the plugin should run, particularly the read/write behavior.

mode The run mode. This can be one of "serial" or "parallel". In serial mode, locking is done to ensure reads and writes are not done simultaneously. In parallel mode, no locking is done so reads and writes can occur simultaneously. *(default: serial)*

```
mode: serial
```

read Settings for device reads.

enabled Blanket enable/disable of reading for the plugin. *(default: true)*

```
enabled: false
```

interval Perform device reads every *interval*. That is to say for an interval of `1s`, the plugin would read from all devices every second. *(default: 1s)*

```
interval: 750ms
```

buffer The size of the read buffer. This is the size of the channel that passes readings from the read goroutine to the readings cache. *(default: 100)*.

```
buffer: 150
```

write Settings for device writes.

enabled Blanket enable/disable of writing for the plugin. *(default: true)*

```
enabled: false
```

interval Perform device writes every *interval*. That is to say for an interval of `1s`, the plugin would write *max* writes from the write queue every second. *(default: 1s)*

```
interval: 750ms
```

buffer The size of the write buffer. This is the size of the channel that passes writings from the gRPC write handler to the write goroutine. *(default: 100)*.

```
buffer: 150
```

max The max number of write transactions to process in a single pass of the write loop. This generally only matters when in *serial* mode. *(default: 100)*

```
max: 150
```

transaction Settings for write transactions.

ttl The time to live for a transaction in the transaction cache, after which it will be removed. *(default: 5m)*

```
ttl: 10m
```

dynamicRegistration Settings and configurations for the dynamic registration of devices by a plugin.

config The configurations to use for dynamic registration. This should be a list of maps, where the key is a string, and the value can be anything. The data in each map will be passed to the plugin's configured dynamic registration handler function(s).

limiter Configurations for a rate limiter against reads and writes. Some backends may limit interactions, e.g. some HTTP APIs. This configuration allows a limiter to be set up to ensure that a limit is not exceeded.

rate The limit, or maximum frequency of events.

A rate of `0` signifies an unlimited rate.

```
rate: 500
```

burst The bucket size for the limiter, or maximum number of events that can be fulfilled at once.

If this is `0`, it will be the same number as the *rate*.

```
burst: 30
```

health Configuration for plugin health checks.

useDefaults A flag that determines whether the plugin should use the built-in default health checks or not. *(default: true)*

context Configurable context for the plugin. This is generally not used, but is made available as a general map in order to pass values in/around the plugin if needed.

Example

Below is an example of a plugin configuration.

```
version: 1.0
debug: true
network:
  type: tcp
  address: ":5001"
settings:
  mode: parallel
  read:
    interval: 1s
  write:
    interval: 2s
```

1.4.2 Device Configuration

Device configurations define the devices that a plugin will interface with and expose to Synse Server.

All device configs are unified into a single config when the plugin reads them in and validates them. Device configurations can be specified in a single file, or across multiple files. The file name does not matter, but it must have a .yml or .yaml extension.

Config Policies

The following config policies relate to device configuration.

For file configuration:

- DeviceConfigFileOptional

- DeviceConfigFileRequired *(default)*
- DeviceConfigFileProhibited

For dynamic configuration:

- DeviceConfigDynamicOptional *(default)*
- DeviceConfigDynamicRequired
- DeviceConfigDynamicProhibited

Config Locations

The default locations for the device configuration(s) (in order of evaluation) are:

```
./config/device
/etc/synse/plugin/config/device
```

A non-default location can be used by setting the `PLUGIN_DEVICE_CONFIG` environment variable to either the directory containing the config file, or to the config file itself.

```
PLUGIN_DEVICE_CONFIG=/tmp/device/config.yml
```

Configuration Options

version The version of the configuration scheme.

```
version: 1.0
```

locations A list of location definitions. Device instances specify their location by referencing the locations defined here.

```
locations:
  - name: r1b1
    rack:
        fromEnv: RACK
    board:
        name: board1
```

<location>.name The name given to the location. This is how the location is identified and referenced. There cannot be different locations with the same name.

<location>.rack The specification for the rack location. This is a map that contains one of the following:

name The name of the rack.

fromEnv The name of the environment variable holding the name of the rack.

<location>.board The specification for the board location. This is a map that contains one of the following:

name The name of the board.

fromEnv The name of the environment variable holding the name of the board.

devices A list of device kinds, where each item in the list is referenced as `kind`, below.

```
devices:
  - name: temperature
    metadata:
        model: example-temp
    instances:
      - channel: "0014"
        location: r1b1
        info: Temperature Device 1
```

<item>.name The name of the device kind. This name should be unique to the device kind for the plugin. This can be arbitrarily namespaced, but the last element of the namespace should be the type of device, e.g. "temperature".

```
name: foo.bar.temperature
```

<item>.metadata Metadata associated with the device kind. This is a mapping of string to string. There is no limit to the amount of metadata stored here. This metadata should be for the device kind level, so it could contain information like a product ID, model number, manufacturer, etc. This is optional and just used to help identify the devices.

```
metadata:
    model: example-temp
    manufacturer: vaporio
```

<item>.handlerName Specifies the name of the DeviceHandler to match to this device kind. By default, a device kind will match to a handler using its *Name* field. If this field is set, it will override that behavior and match to a handler with the name specified here. This field is optional.

```
handlerName: foo.example
```

<item>.outputs A list of the reading output types provided by device instances for this device kind. A device instance can specify its own outputs, but if all instances for a kind will support the same outputs, it is cumbersome to re-specify them for every device, so they can be specified here and will be inherited by the device instances. See the output type config options, below.

<item>.instances A list of device instances configured for this device kind. The instance configurations define the devices that the plugin will ultimately read from and write to. See the device instance config options, below.

Output Type Config Options

type The name of the output type that describes the output format for a device reading output.

```
type: foo.temperature
```

info Any info that can be used to provide a short human-understandable label, description, or summary of the reading output. This is optional.

```
info: On-board temperature reading value
```

data A map where the key is a string and the value is anything. This data contains any protocol/output specific configuration associated with the device output. Most device outputs will not need their own configuration data specified here, in which case this can be left empty. It is the responsibility of the plugin to handle these values correctly.

```
data:
    channel: 3
    port: /dev/ttyUSB0
```

Device Instance Config Options

info A short human-understandable label, description, or summary of the device instance. While this is not required, it is recommended to used, as it makes identifying devices much easier.

```
info: top right temperature sensor
```

location The location of the device. This should be a string that references the `name` of a location that was specified in the `locations` block of the config. This field is required.

```
location: r1b1
```

data Any protocol/device specific configuration for this device instance. This will often be data used to communicate with the device. It is the responsibility of the plugin to handle these values correctly.

```
data:
    channel: 5
    port: /dev/ttyUSB0
    id: 14
```

outputs A list of the output types for the readings that this device supports. A device instance will need to have at least one output type, but can have more. It can inherit output types from its device kind. For more, see the section on device outputs, above.

```
outputs:
    - type: foo.temperature
    - type: foo.humidity
```

disableOutputInheritance A flag that, when set, will prevent this instance from inheriting output types from its parent device kind. This is false by default (so it will inherit by default).

```
disableOutputInheritance: true
```

handlerName The name of a device handler to match to this device instance. By default, a device instance will match with a device handler using the Name field of its device kind. This field can be set to override that behavior and match to a handler with the name specified here. This field is optional.

```
handlerName: foo.bar.something
```

Example

Below is an example of a device configuration.

```
version: 1.0
locations:
  - name: r1vec
    rack:
      name: rack-1
    board:
      name: vec
```

(continues on next page)

(continued from previous page)

```
devices:
  - name: temperature
    metadata:
      model: example-temp
      manufacturer: vaporio
    outputs:
      - type: temperature
    instances:
      - info: Example Temperature Sensor 1
        location: r1vec
        data:
          id: 1
      - info: Example Temperature Sensor 2
        location: r1vec
        data:
          id: 2
      - info: Example Temperature Sensor 3
        location: r1vec
        data:
          id: 3
```

1.4.3 Output Type Configuration

Output type configurations define output types which describe how a device reading should be formatted and adds context info around the reading output. Output type configurations can be specified directly in the code, so they do not need to be set via config file. Since these should not change frequently, it is recommended to define them in-code, but that may not work well for all plugins, so the option to define them via config exists.

Config Policies

The following config policies relate to output type configuration.

- TypeConfigFileOptional *(default)*
- TypeConfigFileRequired
- TypeConfigFileProhibited

Config Locations

The default locations for the output type configuration(s) (in order of evaluation) are:

```
./config/type
/etc/synse/plugin/config/type
```

A non-default location can be used by setting the `PLUGIN_TYPE_CONFIG` environment variable to either the directory containing the config file, or to the config file itself.

```
PLUGIN_DEVICE_CONFIG=/tmp/type/config.yml
```

Configuration Options

version The version of the configuration scheme.

```
version: 1.0
```

name The name of the output type. Output type names should be unique for a plugin. The name can be arbitrarily namespaced.

```
name: foo.temperature
```

precision The decimal precision that the reading should be rounded to. This is only applied to readings that provide float values. This specifies the number of decimal places to round to.

```
precision: 3
```

unit The unit of reading.

```
unit:
  name: millimeters per second
  symbol: mm/s
```

name The full name of the unit.

symbol The symbolic representation of the unit.

scalingFactor A factor that the reading value can be multiplied by to get the final output value. This is optional and will be 1 if not specified (e.g. the reading value will not change). This value should resolve to a numeric. Negatives and fractional values are supported. This can be the value itself, e.g. "0.01", or a mathematical representation of the value, e.g. "1e-2".

```
scalingFactor: -.4E10
```

1.5 Tutorial

This page will go through a step by step tutorial on how to create a plugin. The plugin we will create in this tutorial will be simple and provide readings for a single device. For examples of more complex plugins, see the `examples` directory in the source repo, or see the Emulator Plugin.

The plugin we will build here will provide a single "memory" device which will give readings for total memory, free memory, and the used percent. To get this memory info we will use https://github.com/shirou/gopsutil.

1.5.1 0. Figure out desired behavior

Before we dive in and start writing code, its always a good idea to lay out what we want that code to do. In this case, we'll outline what we want the plugin to do, and what data we want it to provide. Since this is a simple, somewhat contrived plugin, these are all pretty basic.

Goals

- Provide readings for memory usage
- Do not support writing (doesn't make sense in this case)
- Have the readings be updated every 5 seconds

Devices

- **One kind of device will be supported: a "memory usage" device. It will provide readings for:**
  - Total Memory (in bytes)
  - Free Memory (in bytes)
  - Used Percent (percent)

With this outline of what we want in mind, we can start framing up the plugin.

1.5.2 1. Create the plugin skeleton

If you have read the documentation on plugin configuration, you will know that there are three types of configurations that a plugin uses: plugin config, device config, and output type config. What each does is explained in the configuration documentation.

We will not need to define the output type config, since we will have our output types built directly into the plugin. That means we only need to specify the device config and the plugin config.

We will include those with our plugin, as well as a file to define the plugin.

```
tutorial-plugin
    config
        device
           mem.yml
   config.yml
   plugin.go
```

Note: There are different ways a plugin can be structured. This example does not aim to define the "correct" way. Since it is a simple plugin, it just has a simple structure.

First, we will focus on writing the configuration for the plugin and the supported devices. Note that the plugin configuration does not need to be written first. For this tutorial we are writing if first, though, to help build an understanding of how devices are defined and how the plugin will ultimately use them.

1.5.3 2. Write the configurations

First we'll start with the plugin configuration, then we will look at the device configuration.

Plugin Configuration

The plugin configuration defines how the plugin itself will operate. Since this is a simple, somewhat contrived plugin with only a single readable device, the configuration will not be too complicated. See the plugin configuration documentation for more info on how to configure plugins.

First, we will want to decide what protocol we want the plugin to use. In this case, we will use unix socket, but it should be trivial to use TCP instead, should you decide to.

As per the Goals we laid out in section 0, we want the readings to be updated every 5 seconds. That means we will need to set the read interval to `5s`. All together, this would look like:

Listing 1: config.yml

```
version: 1.0
debug: false
network:
  type: unix
  address: memory.sock
settings:
  read:
    interval: 5s
```

In the above, `version` refers to the version of the configuration file scheme, not the version of the plugin itself. We've also set `debug: false` to disable debug logging. If you wish to see debug logs, just set this to `true`.

Device Configuration

Next, we will define the device configuration for our memory device.

In this simple case, we can say that our device is a "memory" type device. Although optional, we will also specify some metadata with it, namely a model (that we will make up for the sake of the tutorial). The name of the device kind needs to be unique, but since this is the only device we have here, we don't need to worry about it.

Another component to the instance configurations is defining the device location. If you are familiar with Synse Server, you will know that we currently reference devices via a rack/board/device hierarchy, e.g. `read/rack-1/board-1/device-1`. These are effectively just labels to namespace devices, so they can be whatever you want them to be. For this tutorial, we'll say that the rack is `local` and the board is `host`. This should result in the Synse Server URI `read/local/host/<device-id>`.

Note: Synse Server 2.0 uses the `<rack>/<board>/<device>` notation for identifying all devices. This notation is largely historical from the initial design of Synse Server, which did not aim to be as generalized as it is now. In future versions (e.g. 3.0), early planning and discussion has the strict rack-board-device requirements phased out in favor of more generalized labeling. This should not be any concern now, but something to look for in the future.

Additionally, we will need to specify the output types of the device readings. We have not defined those in code yet, but we know from section 0 that we want a single device that outputs:

- Total Memory (in bytes)
- Free Memory (in bytes)
- Used Percent (percent)

So we can call those outputs `memory.total`, `memory.free`, and `percent_used`, respectively. Later, we will define the output types corresponding to those names.

The final piece to our configuration is specifying the config for the memory device instance. Here we will only want one device instance (we're only getting memory from one place, so we only need a single device to do it). As we will see in the next section, we will need a way to reliably identify this device. For protocols like HTTP, RS-485, and others, we can do this by using the addressing configuration as part of the ID composite (if device X can only be reached via unique address A, then address A can help to identify device X). Since we do not need any protocol-specific configurations for our memory device, we will just add in an `id` field that will provide a reliable unique identifier for that device (since we only have one device, it may seem weird, but if we were to have two memory devices, we'd need a way to differentiate).

Listing 2: config/device/mem.yml

```
version: 1.0
locations:
  - name: local
    rack:
      name: local
    board:
      name: host
devices:
  - name: memory
    metadata:
      model: tutorial-mem
    outputs:
      - type: memory.total
      - type: memory.free
      - type: percent_used
    devices:
      - info: Virtual Memory Usage
        location: local
        data:
          id: 1
```

In the above config, the `version` is the version of the configuration scheme.

1.5.4 3. Define the output types

As mentioned in the previous section, we still need to define the output types that we used in the device configuration. While we could define these in their own config files, its easier to just define them right in the code.

We know that both free memory and total memory should describe the number of bytes and percent used should be a percentage. Knowing this and what we are calling these output types is all we need

```
var (
    memoryTotal = sdk.OutputType{
        Name: "memory.total",
        Unit: sdk.Unit{
            Name: "bytes",
            Symbol: "B",
        },
    }

    memoryFree = sdk.OutputType{
        Name: "memory.free",
        Unit: sdk.Unit{
            Name: "bytes",
            Symbol: "B",
        },
    }

    percentUsed = sdk.OutputType{
        Name: "percent_used",
        Unit: sdk.Unit{
            Name: "percent",
            Symbol: "%",
        },
```

(continues on next page)

(continued from previous page)

```
    }
)
```

1.5.5 4. Write handlers for the device(s)

If you've read through some of the documentation on plugin basics, you should know that in order to handle the configured devices, handlers for those devices need to be defined.

We only want our memory device to support reading, so we only need to define a read function for our device handler. To read the memory info, we will use https://github.com/shirou/gopsutil which can be gotten via

```
$ go get github.com/shirou/gopsutil/mem
```

Using that package, we will define the read functionality for the `memory` device. Note that because this tutorial is simple, we are putting everything in one file, but this is not required and is discouraged for plugins that do anything beyond serve as an example. See the SDK repo's `examples` directory or the emulator plugin for examples of how to structure plugins.

Device Handler

Next we'll define the read-write handler for our device. We won't do any writing for the device, so its more of a read handler in this case. To read the memory info, we can use https://github.com/shirou/gopsutil which can be gotten via

```
$ go get github.com/shirou/gopsutil/mem
```

We can use that package to define our read functionality for the `memory` device. Note that because this tutorial is simple, we are putting everything in one file, but this is not required and is discouraged for plugins that do anything beyond serve as an example. See the SDK repo's `examples` directory or the emulator plugin for examples of how to structure plugins.

```
var memoryHandler = sdk.DeviceHandler{
    Name: "memory",
    Read: func(device *sdk.Device) ([]*sdk.Reading, error) {
            v, err := mem.VirtualMemory()
            if err != nil {
                    return nil, err
            }
            return []*sdk.Reading{
                    device.GetOutput("memory.total").MakeReading(v.Total),
                    device.GetOutput("memory.free").MakeReading(v.Free),
                    device.GetOutput("percent_used").MakeReading(v.UsedPercent),
            }, nil
    },
}
```

Now we have our configuration defined and our handler defined. Next, we put together the plugin, configure it, and register the handlers.

1.5.6 5. Create and configure the plugin

The creation, configuration, registration, and running of a plugin can all be done within the `main()` function. In short, the things that need to happen are:

- register plugin metadata
- create the `Plugin`
- register the output types
- register all handlers
- run the plugin

If that sounds simple – that's because it should be!

All plugins have some metadata associated with them. At a minimum, all plugins require a name, but should also have a maintainer and short description and can have a VCS link as well. We will call the plugin "tutorial plugin" and will have "vaporio" be the maintainer.

```
func main() {
    // Set plugin metadata
    sdk.SetPluginMeta(
            "tutorial plugin",
            "vaporio",
            "a simple plugin that reads virtual memory - used as a tutorial",
            "",
    )

    // Create the plugin
    plugin := sdk.NewPlugin()

    // Register output types
    err := plugin.RegisterOutputTypes(
            &memoryTotal,
            &memoryFree,
            &percentUsed,
    )
    if err != nil {
            log.Fatal(err)
    }

    // Register the device handler
    plugin.RegisterDeviceHandlers(
            &memoryHandler,
    )

    // Run the plugin.
    if err := plugin.Run(); err != nil {
        log.Fatal(err)
    }
}
```

Note: There are more things that can be done during plugin setup, from registering pre-run/post-run actions, to modifying various behaviors, to adding health checks. For more on this, see the *Advanced Usage* section.

1.5.7 6. Plugin Summary

To summarize, we should now have a file structure that looks like:

```
 tutorial-plugin
     config
         device
            mem.yml
    config.yml
    plugin.go
```

With the configuration files:

Listing 3: config.yml

```
version: 1.0
debug: false
network:
  type: unix
  address: memory.sock
settings:
  read:
    interval: 5s
```

Listing 4: config/device/mem.yml

```
version: 1.0
locations:
  - name: local
    rack:
      name: local
    board:
      name: host
devices:
  - name: memory
    metadata:
      model: tutorial-mem
    outputs:
      - type: memory.total
      - type: memory.free
      - type: percent_used
    devices:
      - info: Virtual Memory Usage
        location: local
        data:
          id: 1
```

And the plugin source code:

Listing 5: plugin.go

```
package main

import (
    "log"

    "github.com/shirou/gopsutil/mem"

    "github.com/vapor-ware/synse-sdk/sdk"
)
```

(continues on next page)

(continued from previous page)

```
var (
    memoryTotal = sdk.OutputType{
            Name: "memory.total",
            Unit: sdk.Unit{
                    Name: "bytes",
                    Symbol: "B",
            },
    }

    memoryFree = sdk.OutputType{
            Name: "memory.free",
            Unit: sdk.Unit{
                    Name: "bytes",
                    Symbol: "B",
            },
    }

    percentUsed = sdk.OutputType{
            Name: "percent_used",
            Unit: sdk.Unit{
                    Name: "percent",
                    Symbol: "%",
            },
    }
)

var memoryHandler = sdk.DeviceHandler{
    Name: "memory",
    Read: func(device *sdk.Device) ([]*sdk.Reading, error) {
            v, err := mem.VirtualMemory()
            if err != nil {
                    return nil, err
            }
            return []*sdk.Reading{
                    device.GetOutput("memory.total").MakeReading(v.Total),
                    device.GetOutput("memory.free").MakeReading(v.Free),
                    device.GetOutput("percent_used").MakeReading(v.UsedPercent),
            }, nil
    },
}

func main() {
    // Set plugin metadata
    sdk.SetPluginMeta(
            "tutorial plugin",
            "vaporio",
            "a simple plugin that reads virtual memory - used as a tutorial",
            "",
    )

    // Create the plugin
    plugin := sdk.NewPlugin()

    // Register output types
    err := plugin.RegisterOutputTypes(
            &memoryTotal,
            &memoryFree,
```

(continues on next page)

(continued from previous page)

```
            &percentUsed,
    )
    if err != nil {
            log.Fatal(err)
    }

    // Register the device handler
    plugin.RegisterDeviceHandlers(
            &memoryHandler,
    )

    // Run the plugin.
    if err := plugin.Run(); err != nil {
        log.Fatal(err)
    }
}
```

1.5.8 7. Build and run the plugin

Next we will build and run the plugin locally, without Synse Server in front of it. In order to interface with the plugin, we'll use the Synse CLI.

From within the `tutorial-plugin` directory,

```
$ go build -o plugin
```

Congratulations, the plugin is now built! Now we can run it

```
$ ./plugin
```

You should see a single registered `memory` device and no errors. To interact with the plugin, we can use the CLI.

> **Warning:** The CLI may not be fully updated for SDK 1.0 yet, so not all of the CLI commands below may work. These docs will be updated once the CLI is updated.

Getting the plugin device info

```
$ synse plugin -u /tmp/synse/procs/memory.sock meta
ID                                TYPE      MODEL          PROTOCOL   RACK      BOARD
65f660ac428556804060c13349e500de  memory    tutorial-mem   os         local     host
```

Getting a reading from the device

```
$ synse plugin -u /tmp/synse/procs/memory.sock read local host␣
↪65f660ac428556804060c13349e500de
TYPE           VALUE               TIMESTAMP
total          8589934592          Thu Apr 19 11:19:36 EDT 2018
free           324714496           Thu Apr 19 11:19:36 EDT 2018
percent_used   73.24576377868652   Thu Apr 19 11:19:36 EDT 2018
```

The device doesn't support writes, so writing should fail

```
$ synse plugin -u /tmp/synse/procs/memory.sock write local host␣
↪65f660ac428556804060c13349e500de total 123
rpc error: code = Unknown desc = writing not enabled for device local-host-
↪65f660ac428556804060c13349e500de (no write handler)
```

Now, you've configured, created, and run a plugin. The only thing left to do is connect it with Synse Server and access the data it provides via Synse Server's HTTP API.

1.5.9 8. Using with Synse Server

In this section, we'll go over how to deploy a plugin with Synse Server. While there are a few ways of doing it, the recommended way is to run the plugin as a container and link it to the Synse Server container. This means the plugin will be getting memory info from the container, not the host machine, but this section just serves as an example of how to do it.

The first thing we will need to do is containerize the plugin. For this, we can write a Dockerfile. For our Dockerfile, we'll assume that the binary was built locally, but examples exist in other repos of how to use docker build stages to containerize the build process as well.

It is also important to note that all configs can be included in the Dockerfile with the plugin, but it is best practice to not do this. The prototype configs can be included, since they should not change based on the deployment, but the instance and plugin configs may change, so they should be provided at runtime.

First, we'll make sure we have our plugin build locally. We will use the alpine linux base image, so we want to build it for linux. If you are running on linux, this can be done simply with

```
$ go build -o plugin
```

If running on a non linux/amd64 architecture, e.g. Darwin, you will need to cross-compile

```
$ GOOS=linux GOARCH=amd64 go build -o plugin
```

Now, we can write our Dockerfile. While the configs can be built-in, we will not do so here, since it is good practice to provide the configs at runtime for that particular deployment.

Listing 6: Dockerfile

```
FROM alpine

COPY plugin plugin

CMD ["./plugin"]
```

We can build the image as `vaporio/tutorial-plugin`

```
$ docker build -t vaporio/tutorial-plugin .
```

Before we run the image, we'll want to update the plugin configuration that we will use. Instead of using unix sockets for networking, we'll use TCP over port 5001. Change `config.yml` to:

```
version: 1.0
name: memory
debug: false
network:
  type: tcp
  address: ":5001"
```

(continues on next page)

(continued from previous page)

```
settings:
  read:
    interval: 5s
```

Running via Docker

Now we can run the plugin, supplying the plugin and instance configurations. We will also need to specify environment variables so the plugin knows where to look for these configurations.

```
$ docker run -d \
    -p 5001:5001 \
    --name=tutorial-plugin \
    -v $PWD/config/device:/etc/synse/plugin/config/device \
    -v $PWD/config.yml:/tmp/config.yml \
    -e PLUGIN_CONFIG=/tmp \
    vaporio/tutorial-plugin
```

The plugin should now be running and waiting. You can check `docker logs tutorial-plugin` to view the logs and make sure everything is running correctly.

To connect it to Synse Server, you'll need the Synse Server image. The easiest way is to just pull it from DockerHub:

```
$ docker pull vaporio/synse-server
```

We'll also need to create a network to link them across.

```
$ docker network create synse
$ docker network connect synse tutorial-plugin
```

We'll now run Synse Server and connect it to the network. Here, we register the tutorial plugin with Synse Server by using its environment configuration.

```
$ docker run -d \
    --name=synse-server \
    --network=synse \
    -p 5000:5000 \
    -e SYNSE_PLUGIN_TCP=tutorial-plugin:5001 \
    vaporio/synse-server
```

Now, you should be ready to use Synse Server to interact with the plugin. See the *Interacting via Synse Server* section, below.

Running via Docker Compose

All of the above can be done somewhat simpler via docker compose, using a compose file

Listing 7: tutorial.yml

```
version: "3"
services:
  synse-server:
    container_name: synse-server
    image: vaporio/synse-server
    ports:
```

(continues on next page)

(continued from previous page)

```
      - 5000:5000
    environment:
      SYNSE_PLUGIN_TCP: tutorial-plugin:5001
    links:
      - tutorial-plugin

  tutorial-plugin:
    container_name: tutorial-plugin
    image: vaporio/tutorial-plugin
    ports:
      - 5001:5001
    volumes:
      - ./config/device:/etc/synse/plugin/config/device
      - ./config.yml:/tmp/config.yml
    environment:
      PLUGIN_CONFIG: /tmp
```

Then, just bring up the compose file

```
$ docker-compose -f tutorial.yml up -d
```

You should now be ready to use Synse Server to interact with the plugin. See the next section for how to do so.

Interacting via Synse Server

With Synse Server now running locally, we can interact with its HTTP API using `curl`.

- Check that the server is up and ready

```
$curl localhost:5000/synse/test
{
  "status":"ok",
  "timestamp":"2018-04-19T16:56:16.085286Z"
}
```

- Get `scan` information (e.g., see which devices are available). We should expect to see the single memory device managed by the plugin.

```
$ curl localhost:5000/synse/2.1/scan
{
  "racks":[
    {
      "id":"local",
      "boards":[
        {
          "id":"host",
          "devices":[
            {
              "id":"baeb1223219e634446c4af115be089e7",
              "info":"Virtual Memory Usage",
              "type":"memory"
            }
          ]
        }
      ]
    }
```

(continues on next page)

(continued from previous page)

```
  ]
}
```

- We can `read` from that device, and we should expect to get back the total, free, and percent_used readings from the memory device.

```
$ curl localhost:5000/synse/2.1/read/local/host/baeb1223219e634446c4af115be089e7
{
  "kind":"memory",
  "data":{
    "total":{
      "value":2096058368,
      "timestamp":"2018-06-19T13:28:31.0881264Z",
      "unit":{
        "symbol":"B",
        "name":"bytes"
      },
      "type":"total",
      "info":""
    },
    "free":{
      "value":211611648,
      "timestamp":"2018-06-19T13:28:31.0881454Z",
      "unit":{
        "symbol":"B",
        "name":"bytes"
      },
      "type":"free",
      "info":""
    },
    "percent_used":{
      "value":69.7154570841,
      "timestamp":"2018-06-19T13:28:31.0881577Z",
      "unit":{
        "symbol":"%",
        "name":"percent"
      },
      "type":"percent_used",
      "info":""
    }
  }
}
```

Now, you have successfully created, configured, and ran a Synse Plugin both on its own and as part of a deployment with Synse Server. Explore the Synse Server API to see what else you can do with it.

CHAPTER 2

Community Guide

Learn about the Synse Plugin SDK ecosystem and community. This section outlines the community guidelines, provides license info, and gives details on how to contribute to the Plugin SDK.

2.1 Community Plugins

2.1.1 Contributing

Below are some open sourced plugins developed by Vapor IO and the Synse Community. If you have developed your own Synse Plugin and would like to share it with the community, let us know by creating a new issue or opening a pull request to add it to this list.

2.1.2 Plugins

Synse Emulator Plugin (GitHub)

A plugin that provides emulated devices with no back-end dependency. This plugin can be used for development, testing, and to just get familiar with Synse and plugins.

Synse SNMP Plugin (GitHub)

A general-purpose SNMP plugin for Synse Server.

Synse Modbus-IP Plugin (GitHub)

A general-purpose Modbus-over-IP plugin for Synse Server.

Synse AMT Plugin (GitHub)

Intel AMT plugin for Synse Server.

Synse IPMI Plugin (GitHub)

A general-purpose IPMI plugin for Synse Server.

2.2 License

The Synse Plugin SDK is licensed under the GPL-3.0 license.

Briefly, that means that:

> *You may copy, distribute and modify the software as long as you track changes/dates in source files. Any modifications to or software including (via compiler) GPL-licensed code must also be made available under the GPL along with build & install instructions.*
>
> – tldrleagal

For the full license, see the LICENSE file in the source repo.

2.3 Contributing

2.3.1 Reporting an Issue

If you find a bug or experience unexpected behavior with the Synse Plugin SDK, feel free to open an issue on GitHub

2.3.2 Requesting a Feature

If there is functionality missing from the Synse Plugin SDK that you think would be nice to have, please open a feature request issue on GitHub.

2.4 Release Process

The following guidelines describe the criteria for new releases. The Synse Plugin SDK is versioned with the format `major.minor.micro`.

2.4.1 Major Version

A major release will include breaking changes. When a new major release is cut, it will be versioned as `X.0.0`. For example, if the previous release version was `1.4.2`, the next version would be `2.0.0`.

Breaking changes are changes which break backwards compatibility with previous versions. Typically, this would mean changes to the API. Major releases may also include bug fixes.

2.4.2 Minor Version

A minor release will not include breaking changes to the API, but may otherwise include additions, updates, or bug fixes. If the previous release version was `1.4.2`, the next minor release would be `1.5.0`.

Minor version releases are backwards compatible with releases of the same major version number.

2.4.3 Micro Version

A micro release will not include any breaking changes and will typically only include minor changes or bug fixes that were missed with the previous minor version release. If the previous release version was `1.4.2`, the next micro release would be `1.4.3`.

CHAPTER 3

Development

Learn about the development processes for the Synse Plugin SDK. If you want to contribute to, play around with, or fork the Plugin SDK, this section will familiarize you with the development workflow, testing practices, etc.

3.1 Developer Setup

This section goes into detail on how to get set up to develop the SDK as well as various development workflow steps that we use here at Vapor IO.

3.1.1 Getting Started

When first getting started with developing the SDK, you will first need to have Go (version 1.9+) installed. To check which version you have, e.g.,

```
$ go version
go version go1.9.1 darwin/amd64
```

Then, you will need to get the SDK source either by checking out the repo via git,

```
$ git clone https://github.com/vapor-ware/synse-sdk.git
$ cd synse-sdk
```

Or via `go get`

```
$ go get -u github.com/vapor-ware/synse-sdk/sdk
$ cd $GOPATH/src/github.com/vapor-ware/synse-sdk
```

Finally, you will need to get the dependencies. We use `dep` for dependency vendoring. A makefile target is included to both get `dep` if you don't already have it and to update the vendored packages specified in `Gopkg.lock`.

```
$ make dep
```

Now, you should be ready to start developing on the SDK.

3.1.2 Workflow

To aid in the developer workflow, Makefile targets are provided for common development tasks. To see what targets are provided, see the project `Makefile`, or run `make help` out of the project repo root.

```
$ make help
build            Build the SDK locally
check-examples   Check that the examples run without failing.
ci               Run CI checks locally (build, test, lint)
clean            Remove temporary files
cover            Run tests and open the coverage report
dep              Ensure and prune dependencies
dep-update       Ensure, update, and prune dependencies
docs             Build the docs locally
examples         Build the examples
fmt              Run goimports on all go files
github-tag       Create and push a tag with the current version
godoc            Run godoc to get a local version of docs on port 8080
help             Print usage information
lint             Lint project source files
setup            Install the build and development dependencies
test             Run all tests
version          Print the version of the SDK
```

In general when developing, tests should be run (e.g. `make test`) and the could should be formatted (`make fmt`) and linted (`make lint`). This ensures that the code works and is consistent and readable. Tests should also be added or updated as appropriate (see the *Testing* section).

3.1.3 CI

All commits and pull requests to the Synse Plugin SDK trigger a build in Circle CI. The CI configuration can be found in the repo's `.circleci/config.yml` file. In summary, a build triggered by a commit will:

- Install dependencies
- Run linting
- Check formatting
- Run tests with coverage reporting (and upload results to CodeCov)
- Build the example plugins in the `examples` directory

When a tag is pushed to the repo, CI checks that the tag version matches the SDK version specified in the repo, then generates a changelog and drafts a new release for that version.

3.2 Testing

The Synse Plugin SDK strives to follow the Golang testing best practices. Tests for each file are found in the same directory following the pattern `FILENAME_test.go`, so given a file named `plugin.go`, the test file would be `plugin_test.go`.

3.2.1 Writing Tests

There are many articles and tutorials out there on how to write unit tests for Golang. In general, this repository tries to follow them as best as possible and also tries to be consistent with how tests are written. This makes them easier to read and maintain. When writing new tests, use the existing ones as a guide.

Whenever additions or changes are made to the code base, there should be tests that cover them. Many unit tests already exists, so some changes may not require tests to be added. To help ensure that the SDK is well-tested, we upload coverage reports to CodeCov. While good code coverage does not ensure bug-free code, it can still be a useful indicator.

3.2.2 Running Tests

Tests can be run with `go test`, e.g.

```
$ go test ./sdk/...
```

For convenience, there is a make target to do this

```
$ make test
```

While the above make target will report coverage at a high level, it can be useful to see a detailed coverage report that shows which lines were hit and which were missed. For that, you can use the make target

```
make cover
```

This will run tests and collect and join coverage reports for all packages/sub-packages and output them as an HTML page.