Synse Plugin SDK Documentation

Release 0.4.0

Vapor IO

Jun 12, 2018

Contents

1 User Guide **3**
- 1.1 The Basics . . . 3
- 1.2 Advanced Usage . . . 5
- 1.3 Plugin Configuration . . . 7
- 1.4 Tutorial . . . 13

2 Community Guide **27**
- 2.1 Community Plugins . . . 27
- 2.2 License . . . 27
- 2.3 Contributing . . . 28
- 2.4 Release Process . . . 28

3 Development **29**
- 3.1 Developer Setup . . . 29
- 3.2 Testing . . . 30

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, you should be able to make the most out of the Plugin SDK.

1.1 The Basics

This page describes some of the basic concepts and features of the SDK for plugin development that most, if not all plugins, will use. See the *Advanced Usage* page for an overview of some of the more advanced features of the plugin SDK.

1.1.1 Architecture Overview

Before describing the various pieces of the SDK, it makes sense to talk about the architecture of a plugin at a higher level. This will help to provide some greater context around some of the components described here and in other parts of the documentation.

For this overview, there are two levels of architecture we will look at:

Plugin Integration with Synse Server

The diagram above provides a high level depiction of how plugins are used by Synse Server. In short, when an HTTP API request comes into Synse Server, e.g. a read request, that request will come with routing information for the device it wants to read from, `<rack id>/<board id>/<device id>`.

This routing information is used by Synse Server to lookup the device and figure out which plugin owns that device and is responsible for its reads/writes. The lookup is done using a meta-info cache that is built by Synse Server on startup when it registers configured plugins and requests meta-info for the devices they manage.

Once Synse Server knows where the request is going, it sends over all relevent information to that plugin via the internal gRPC API. Then, it is up to the plugin to fulfil the request using whatever protocol it implements and return the appropriate response back to Synse Server.

Plugin Data Flow

The diagram above provides a high level depiction of some of the data flows in a plugin, namely for `reads` and `writes`. `metainfo` and `transaction` requests are less interesting to the reader since they are handled by the SDK.

When a gRPC *read* request comes in, the plugin's gRPC read handler will look up that device in a readings cache. If the device does not exist, no reading is returned. If it does exist, that reading is returned.

The reading cache itself is updated on an interval (as set via the plugin configuration) by a "reader" goroutine. This goroutine iterates through all of the configured Devices and, if they are readable, executes the read handler for that device. The reading cache is updated with the reading returned from that handler.

When a gRPC *write* request comes in, the process flow is a little different. The gRPC server's write handler will first create a transaction for the write and then put the write transaction onto a write queue. The transaction is returned to Synse Server – writes are asynchronous and the status of the write should be checked via the returned transaction.

Once a write transaction is on the queue, it will wait for the write goroutine to take it off the queue and fulfil the transaction's write request. This is also done on an interval (set via the plugin configuration).

1.1.2 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.

1.1.3 The Read and Write Loops

Reading and writing happens in separate loops. More specifically, reading and writing happens in different 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).

A plugin can run in one of two modes: `serial` or `parallel`. These mode settings directly effect the behavior of the read/write goroutines. In serial mode, device access will be locked – reading and writing cannot happen simultaneously. In parallel mode, there is no locking – reads and writes can happen simultaneously.

1.1.4 Devices

Within the SDK, a Device is really just a model that holds the configuration information joined from a prototype config and a device instance config.

The Devices hold all of the meta-information for a device as well as references to their read and write handlers and identifier handler.

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, it is important to not initialize a `Read` struct directly, but instead to use the SDK's `NewReading` function. This function will auto-populate the timestamp field with a timestamp in the RFC3339Nano format, which is the standard time format for plugins and Synse Server.

1.2 Advanced Usage

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

1.2.1 Device Enumeration Handler

The Device Enumeration Handler,

```
type DeviceEnumerator func(map[string]interface{}) ([]*config.DeviceConfig, error)
```

is a handler that allows a plugin to register device instances programmatically, not through pre-defined YAML. A good use case for this is IPMI, where the plugin will know which BMCs to reach out to, but not which devices are on the BMCs. Instead of manually going through and constructing the configuration for each server, this can be done through a device enumeration handler that connects with the BMC, get all devices it has, and then initialize plugin Device instances for those found devices.

The `map[string]interface{}` that is the input parameter to the `DeviceEnumerator` function type is the map defined in the plugin configuration under the `auto_enumerate` key. Any values can be specified there, under any nesting, but it is up to the plugin writer to parse them correctly.

For more, see the Auto Enumerate Example Plugin.

1.2.2 Pre Run Actions

Pre Run Actions are actions that the plugin should perform before it starts to run the gRPC server and start the read/write goroutines. These are actions that should be used for plugin-wide setup actions, should a plugin require it. This could be performing some kind of authentication, verifying that some backend exists and is reachable, etc.

Pre Run Actions should be defined as part of plugin initialization and should be registered with the plugin before it is run.

A pre run action should fulfil the `pluginAction` type

```
type pluginAction func(p *Plugin) error
```

The `pluginAction` should then be registered with the plugin via the plugin.RegisterPreRunActions function.

An (abridged) example:

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

(continues on next page)

(continued from previous page)

```
func main() {
    // Create a new Plugin
    plugin, err := sdk.NewPlugin(handlers, nil)
    if err != nil {
        log.Fatal(err)
    }

    // Register the action with the plugin.
    plugin.RegisterPreRunActions(
        preRunAction,
    )
}
```

For more, see the Pre Run Actions Example Plugin.

1.2.3 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 be defined as part of plugin initialization and should be registered with the plugin before it is run.

A device setup action should fulfil the `deviceAction` type

```
type deviceAction func(p *Plugin, d *Device) error
```

The `deviceAction` should then be registered with the plugin via the plugin.RegisterDeviceSetupActions function.

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, err := sdk.NewPlugin(handlers, nil)
    if err != nil {
        log.Fatal(err)
    }

    // Register the action with all devices that have
    // the type "airflow".
    plugin.RegisterDeviceSetupActions(
        "type=airflow",
        deviceSetupAction,
    )
}
```

For more, see the Pre Run Actions Example Plugin.

1.2.4 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.

1.3 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 Prototype*: Meta information for a supported device type.
- *Device Instance*: Instance information for a supported device type.

Device prototype information is relatively static and should not change much. It is considered safe to package it with the plugin, e.g. in a Docker image. The plugin configuration and device instance configuration, however, should be defined on a per-instance basis.

1.3.1 Plugin Configuration

The plugin configuration is a YAML file the defines some plugin metainfo and describes how the plugin should operate.

Default Location

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

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

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

Configuration Options

version The version of the configuration scheme.

```
version: 1.0
```

name The name of the plugin.

```
name: example
```

debug Enables debug logging.

```
debug: true
```

network Network settings for the gRPC server.

> **type** The type of networking the gRPC server should use. This should be one of "tcp" and "unix".

```
type: tcp
```

address The network address. For unix socket-based networking, this should be the name of the socket. This is typically `<plugin-name>.sock`, e.g. `example.sock`. For tcp, this can be 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" and "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.

```
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
```

auto_enumerate The auto-enumeration context for a plugin. This is dependent on the plugin and the device enumeration handler, but in general it can be anything. For more, see *Device Enumeration Handler*.

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.

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
```

Example

Below is a complete, if contrived, example of a plugin configuration.

```
version: 1.0
name: example
debug: true
network:
  type: unix
  address: example.sock
settings:
  mode: parallel
  read:
    interval: 1s
  write:
    interval: 2s
```

1.3.2 Device Prototype Configuration

Prototype configurations define the static meta-info for a given device type. Additionally, they define the expected output scheme for those devices.

Default Location

The default location for device instance configurations is

```
/etc/synse/plugin/config/proto
```

Configuration Options

version The version of the configuration scheme.

```
version: 1.0
```

prototypes A list of prototype objects.

<proto>.type The type of the device. This should match up with the `type` of the corresponding instance configuration(s).

```
type: temperature
```

<proto>.model The model of the device. This should match up with the `model` of the corresponding instance configuration(s).

```
model: example-temp
```

<proto>.manufacturer The manufacturer of the device.

```
manufacturer: Vapor IO
```

<proto>.protocol The protocol that the device uses to communicate. This is often the same as the kind of plugin, e.g. "ipmi", "rs485".

```
protocol: i2c
```

<proto>.output See the output configuration details, below.

The output configuration is a list of reading types. This is separated from the `<proto>.output` above only to give it more room on the page.

output A list of the supported reading outputs for the device.

type The type of the reading. This will be the `type` field of an sdk.Reading.

data_type The type of the data. This is the type that the data will be cast to in Synse Server, e.g. "int", "float", "string".

unit The specification for the reading's unit.

name The name of the unit, e.g. "millimeters per second"

symbol The symbol of the unit, e.g. "mm/s"

precision *(Optional)* The decimal precision of the readings. e.g. a precision of 3 would round a reading to 3 decimal places.

range *(Optional)* The range of permissible values for the reading.

min The minimum permissible reading value.

max The maximum permissible reading value.

Example

Below is a complete, if contrived, example of a device prototype configuration.

```
version: 1.0
prototypes:
  - type: temperature
    model: example-temp
    manufacturer: Vapor IO
    protocol: example
    output:
      - type: temperature
        unit:
          name: degrees celsius
          symbol: C
        precision: 2
        range:
          min: 0
          max: 100
```

1.3.3 Device Instance Configuration

Device instance configurations define the instance-specific configurations for a device. This is often, but not exclusively, the information needed to connect to a device, e.g. an IP address or port. Because device instance configurations should be unique to an instance of a device, parts of these configurations are also used to generate the composite id hash for the device.

Default Location

The default location for device instance configurations is

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

Configuration Options

version The version of the configuration scheme.

```
version: 1.0
```

locations A mapping of location alias to location object. Device instances specify their location by referencing the location alias key.

```
locations:
  r1b1:
    rack: rack1
    board: board1
```

<location>.rack The name of the rack for the <location> object. This can be either a string, in which case it is the rack name, or it can be a mapping. The mapping only supports a single key `from_env`, where the value should be the environment variable to get the name from, e.g. `from_env: HOSTNAME`

```
rack:
  from_env: HOSTNAME
```

<location>.board The name of the board for the <location> object. This should be a string.

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

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

<item>.type The type of the device. This should match up with the type specified in the corresponding prototype config.

```
type: temperature
```

<item>.model The model of the device. This should match up with the model specified in the corresponding prototype config.

```
model: example-temp
```

<item>.instances A list of instances for the given device type/model. The items in the list are objects with no restrictions on the fields/values, except that `info` and `location` are reserved. Each item in the instances list should have a `location` specified (the value being a valid location alias, defined in the `locations` object, above). The `info` field is not required, but is used as a human readable tag for the device which is exposed in the device metainfo. All other fields are up to the plugin to define and handle and are typically configurations for connecting to or otherwise communicating with the device.

```
instances:
  - device_address: "/dev/ttyUSB3"
    base_address: 15
    slave_address: 2
    baud_rate: 19200
    parity: E
    location: r1b1
    info: Example Device 1
```

Example

Below is a complete, if contrived, example of a device instance configuration.

```
version: 1.0
locations:
  r1vec:
    rack: rack-1
    board: vec
devices:
  - type: temperature
    model: example-temp
    instances:
      - id: "1"
        location: r1vec
        info: Example Temperature Sensor 1
      - id: "2"
        location: r1vec
```

(continues on next page)

(continued from previous page)

```
    info: Example Temperature Sensor 2
  - id: "3"
    location: r1vec
    info: Example Temperature Sensor 3
```

1.3.4 Environment Overrides

It may not be convenient to place the configuration files into their default locations, e.g. when testing locally or mounting into a container. Environment overrides exist that allow you to tell the plugin where to look for its configuration.

- **PLUGIN_CONFIG** : Specifies the *directory* which contains the plugin configuration file, `config.yml`.
- **PLUGIN_DEVICE_CONFIG** : Specifies the *directory* which contains `proto` and `config` subdirectories that hold the configuration YAMLs for the prototype and instance configurations, respectively.
- **PLUGIN_PROTO_PATH** : Specifies the *directory* which contains the prototype configuration YAMLs.
- **PLUGIN_DEVICE_PATH** : Specifies the *directory* which contains the device instance configuration YAMLs.

1.4 Tutorial

This page will go through a step by step tutorial on how to create a simple 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 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.

1.4.1 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.

1.4.2 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 instance config, and device prototype config. What each does is explained in the configuration documentation. We will need to include those with our plugin, as well as a file to define the plugin.

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

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.

1.4.3 2. Write the plugin configurations

As mentioned in the previous section, plugins have three types of configuration: - Plugin configuration - Device instance configuration - Device prototype configuration

First, we'll start with the plugin 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 name the plugin. Since its job is to provide memory info, we'll call it `memory`.

We'll also want to decide how we want to communicate with the plugin – via TCP, or via Unix Socket. Either is fine, but for the tutorial, we'll use unix socket. Typically, when naming the unix socket for a plugin, we follow the pattern `<PLUGIN_NAME>.sock`, so here it will be `memory.sock`.

Finally, 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
name: memory
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 Prototype Configuration

Next, we want to define the prototype configuration for the memory device. The prototype configuration is basically device configuration that doesn't change between device instances. This is largely device meta-information. The example here will be simple because our plugin/device is simple. See the documentation on device prototype configuration for more detailed information.

The prototype configuration really consists of two types of information: device metainfo, and device output info. The metainfo helps to identify the device. The output info acts as a template for the readings the device provides and how those readings should be formatted.

In this simple case, we can say that our device is a "memory" type device. We need to specify the model as well as the type, since those two bits of info are used to match prototype configs to their instance configs. We will also define some device manufacturer and the device protocol, for completeness.

For this example, there isn't *really* a manufacturer (its just the amount of memory we have available), so we can feel free to put whatever we want. Similarly, there isn't a well-defined protocol that we are using to communicate with the device (e.g. HTTP, IPMI, RS-485, etc), so we can also specify whatever we find useful there.

Finally, we'll need to define the device outputs. As described in section 0, we want to be able to read the total memory, free memory, and percent used. We can call these types "total", "free", and "percent_used", respectively.

Listing 2: config/proto/mem.yml

```
version: 1.0
prototypes:
  - type: memory
    model: tutorial-mem
    manufacturer: virtual
    protocol: os
    output:
      - type: total
        data_type: int
        unit:
          name: bytes
          symbol: B
      - type: free
        data_type: int
        unit:
          name: bytes
          symbol: B
      - type: percent_used
        data_type: float
        unit:
          name: percent
          symbol: "%"
```

In the above config, the `version` is the version of the configuration scheme. Note that we also specified a unit for each reading output. The unit is not required, but since we expect to get bytes and a percentage for the readings, we can explicitly call that out here.

Device Instance Configuration

Having a prototype instance is not enough; we need an instance to fulfill that prototype. This is where the device instance configurations come in. These configs will be joined up with the existing prototype configs by matching the device type and the device model (e.g. `type: memory` and `model: tutorial-mem`).

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.

The final piece to our configuration is specifying the config for the memory device instance. Here we will only want one device (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 3: config/device/mem.yml

```
version: 1.0
locations:
  localhost:
    rack: local
    board: host
devices:
  - type: memory
    model: tutorial-mem
    instances:
      - id: "1"
        location: localhost
        info: Virtual Memory Usage
```

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

Now, we should have all three configurations completed and ready.

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

All that is left is to start writing the plugin itself.

1.4.4 3. 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.

There are a few kinds of handlers:

- **device handler**: the read/write handler specific to a single device
- **device identifier**: the handler that determines how to generate unique ids for *all devices managed by the plugin*
- **device enumerator**: the handler for generating Device instances programmatically, e.g. not from device instance configuration files.

For our simple plugin, we will not need a device enumerator (we've already created the configuration for the device instance anyways). All plugins require one device handler per configured device type (e.g. per prototype). Additionally, all plugins require a device identifier because without it, we would not be able to reliably create deterministic unique ids for all devices.

Device Identifier Handler

We'll start with the device identifier handler, since its the easiest. In the previous section when we defined the instance config, we made note that we need an `id` field to help uniquely identify the device. Our device identifier will simply extract that field from the config for us.

```
func GetIdentifiers(data map[string]string) string {
    return data["id"]
}
```

The `data` map coming in is a map that is populated with the instance data from the instance configuration YAML, e.g. in this case

```
id: "1"
location: localhost
info: Virtual Memory Usage
```

We get the ID out and return. This gets used in the Plugin SDK as part of a composite of locational info, prototype metainfo, and this instance info to generate the unique (and reproducible) device id hash.

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.

```
func Read(device *sdk.Device) ([]*sdk.Reading, error) {
    v, err := mem.VirtualMemory()
    if err != nil {
        return nil, err
    }

    return []*sdk.Reading{
        sdk.NewReading("total", fmt.Sprintf("%v", v.Total)),
        sdk.NewReading("free", fmt.Sprintf("%v", v.Free)),
        sdk.NewReading("percent_used", fmt.Sprintf("%v", v.UsedPercent)),
```

(continues on next page)

(continued from previous page)

```
    }, nil
}
```

And finally, we'll need to associate this read function with the device handler itself

```
var memoryHandler = sdk.DeviceHandler{
    Type: "memory",
    Model: "tutorial-mem",
    Read: Read,
    Write: nil,
}
```

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

1.4.5 4. 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:

- create the `Handlers`
- create the `Plugin`
- register all handlers
- run the plugin

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

```
func main() {

    // The device identifier and device enumerator handlers.
    handlers, err := sdk.NewHandlers(GetIdentifiers, nil)
    if err != nil {
        log.Fatal(err)
    }

    // Create the plugin and register the handlers. The second
    // parameter here is nil -- this signifies that no override
    // configuration is being used and to just get the configs
    // from file.
    plugin, err := sdk.NewPlugin(handlers, nil)
    if err != nil {
        log.Fatal(err)
    }

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

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

(continues on next page)

(continued from previous page)

```
    }
}
```

There is a lot more that can be done when setting up the plugin, such as specifying a device enumerator, specifying pre-run actions, and specifying device setup actions. Since this example plugin is simple, there is no need for that, but those capabilities are described in the advanced usage documentation.

1.4.6 5. Plugin Summary

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

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

With the configuration files:

Listing 4: config.yml

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

Listing 5: config/proto/mem.yml

```
version: 1.0
prototypes:
  - type: memory
    model: tutorial-mem
    manufacturer: virtual
    protocol: os
    output:
      - type: total
        data_type: int
        unit:
          name: bytes
          symbol: B
      - type: free
        data_type: int
        unit:
          name: bytes
          symbol: B
      - type: percent_used
        data_type: float
```

(continues on next page)

(continued from previous page)

```
        unit:
          name: percent
          symbol: "%"
```

Listing 6: config/device/mem.yml

```
version: 1.0
locations:
  localhost:
    rack: local
    board: host
devices:
  - type: memory
    model: tutorial-mem
    instances:
      - id: "1"
        location: localhost
        info: Virtual Memory Usage
```

And the plugin source code:

Listing 7: plugin.go

```
package main

import (
    "log"
    "fmt"

    "github.com/shirou/gopsutil/mem"

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

func GetIdentifiers(data map[string]string) string {
    return data["id"]
}

func Read(device *sdk.Device) ([]*sdk.Reading, error) {
    v, err := mem.VirtualMemory()
    if err != nil {
        return nil, err
    }
    return []*sdk.Reading{
        sdk.NewReading("total", fmt.Sprintf("%v", v.Total)),
        sdk.NewReading("free", fmt.Sprintf("%v", v.Free)),
        sdk.NewReading("percent_used", fmt.Sprintf("%v", v.UsedPercent)),
    }, nil
}

var memoryHandler = sdk.DeviceHandler{
    Type: "memory",
    Model: "tutorial-mem",
    Read: Read,
    Write: nil,
}
```

(continues on next page)

(continued from previous page)

```
func main() {

    // The device identifier and device enumerator handlers.
    handlers, err := sdk.NewHandlers(GetIdentifiers, nil)
    if err != nil {
        log.Fatal(err)
    }

    // Create the plugin and register the handlers. The second
    // parameter here is nil -- this signifies that no override
    // configuration is being used and to just get the configs
    // from file.
    plugin, err := sdk.NewPlugin(handlers, nil)
    if err != nil {
        log.Fatal(err)
    }

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

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

1.4.7 6. 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
```

Doing this and looking through the output logs, you'll see that no devices are registered and some errors were logged around finding device configurations. This is because the SDK looks in the default `/etc/synse/plugin` directory for configs, but our configs are local.

We can set an environment variable to tell it the correct place to look.

```
$ PLUGIN_DEVICE_CONFIG=config ./plugin
```

Now you should see a single registered `tutorial-mem` device and no errors. To interact with the plugin, we can use the CLI

Getting the plugin meta-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.

1.4.8 7. 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

Listing 8: Dockerfile

```
FROM alpine

COPY plugin plugin
COPY config/proto /etc/synse/plugin/config/proto

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"
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_MEMORY=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 9: tutorial.yml

```
version: "3"
services:
  synse-server:
    container_name: synse-server
    image: vaporio/synse-server
    ports:
      - 5000:5000
    environment:
      SYNSE_PLUGIN_TCP_MEMORY: 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/2.0/scan
{
  "racks":[
    {
      "id":"local",
      "boards":[
        {
```

(continues on next page)

(continued from previous page)

```
          "id":"host",
          "devices":[
            {
              "id":"65f660ac428556804060c13349e500de",
              "info":"Virtual Memory Usage",
              "type":"memory"
            }
          ]
        }
      ]
    }
  ]
}
```

- 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/2.0/read/local/host/65f660ac428556804060c13349e500de
{
  "type":"memory",
  "data":{
    "total":{
      "value":2096066560,
      "timestamp":"2018-04-19T16:58:53.1370289Z",
      "unit":{
        "symbol":"B",
        "name":"bytes"
      }
    },
    "free":{
      "value":91377664,
      "timestamp":"2018-04-19T16:58:53.1370605Z",
      "unit":{
        "symbol":"B",
        "name":"bytes"
      }
    },
    "percent_used":{
      "value":23.1238824782,
      "timestamp":"2018-04-19T16:58:53.137088Z",
      "unit":{
        "symbol":"%",
        "name":"percent"
      }
    }
  }
}
```

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.

.

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
```

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
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. Do not update existing dependencies.
dep-update      Ensure and prune dependencies. Update existing dependencies.
docs            Build the docs (via Slate)
examples        Build the examples
fmt             Run goimports on all go files
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.