MachineApps Template

Contents

[Overview]
[Getting Started]
[Getting started from the MachineMotion controller]
[Development environment recommendation]
[Server]
[State machine]
[State machine implementation (MachineAppEngine)]
[Runtime configuration]
[Reacting to Inputs]
[Streaming data to the web client (Notifier)]
[Client]
[Configuration editor]
[Updating the UI from streamed data]

Overview

This document details the functionality of the MachineApp template which enables anyone with a basic knowledge in Python to quickly deploy applications using a MachineMotion controller. The MachineApp template comes pre-loaded onto the MachineMotion controller and can be accessed through Cloud9. It is also possible to access the template through the following public Github repo. This template provides a framework for you to build a complex application upon, by formalizing it as a state machine and provides an example code, executable out-of-the-box.The application backend is built in Python and uses our Python API, the front-end is built in javascript. The MachineApp template will help set up the following capabilities for the MachineMotion controller:

- Multi-controller support
- Machine control (start, pause, resume, stop, software-stop activation)
- Information log template
- E-stop recovery
- Communication between UI and backend
- Offers the compability to build a custom UI and integrate it seamlessly in the Control Center
- Changing logic of input devices through different machine states

The MachineApp template is primarily split into two sections:

1. server/machine_app.py: Python server running the business logic of your program
2. client/ui.js: Web-client that updates in response to the Python server.

Getting Started

1. Clone the repository to a new directory on your machine (https://github.com/VentionCo/mm-machineapp-template)
2. Download python 3.5 (https://www.python.org/downloads/)
3. Run python --version or python3 --version in the command line to check that you have installed properly
4. Install server dependencies by running cd server && pip install -r requirements.txt (See requirements.txt to view the external libraries that the server relies on)
5. Run the server using cd server && python app.py (You may need to use python3 or event python35 instead of python, depending on how your paths were set up)
6. Begin any customizations for your project.
7. upload.py: Uploads your local MachineApp to your controller.
8. restart_server.py: Restarts your MachineApp with the latest code.

Getting started from the MachineMotion controller

The MachineMotion controller comes preloaded with the MachineApp template, however, the default setting is set to be disabled in case you would prefer

running a personal program or to use MachineLogic. To enable the template, follow the steps below:

1. Go to 192.168.7.2 on Google Chrome
2. Access Cloud9
3. On the left-hand side, access vention-control›util›mm-config.json and change custom_machine_app from false to true. Customize the template for your application under mm-applications›app_template›server›machine_app.py.
4. Customize the front-end of your application under mm-applications›app_template›client›ui.js
5. Reboot the controller.
6. Click on "Manual Control" and then click "MachineApp" at the top right-hand corner of the Control Center to access the custom application that you have developed.

Control Center to access MachineApp Template

Development environment recommendation

We recommend building your program in Visual Studio Code with the following extensions:

- Python by Microsoft - Provides debugging support and a seamless way to manage multiple versions of Python
- Python for VSCode - Provides autocompletion recommendations for python

With these extensions installed, you will be able to run the server in `Debug` mode by clicking the debug button in Visual Studio's side bar, selecting `Application` from the dropdown, and clicking the play button. Running in debug mode will allow you to set breakpoints and inspect the state of your application at run time.

After developing locally, to move the application from your development environment of choice to the MachineMotion controller, follow the steps below:

1. Connect your laptop to the MachineMotion controller through the 192.168.7.2 port. Go onto your IP address and enter 192.168.7.2.
2. Run upload.py script in the project's root file. Go to Cloud9.
3. Upload your application files from the prompt.
4. Run the restart_server.py script
5. After a few moments, enter 192.168.7.2:3011 in the address bar to see the application run. Please note that a hard reset may be required (Ctrl + Shift + R) in case the browser caches the previous version of the application.

Server

The Python server is broken down into three parts:

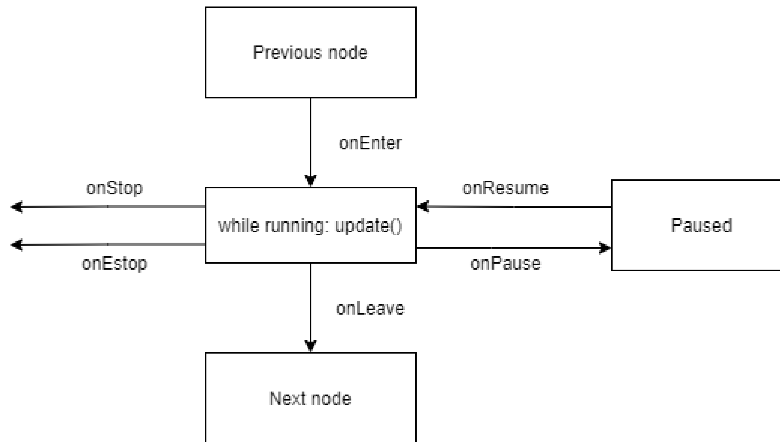
- **State machine** that runs the business logic of the MachineApp. This section of the template should be the only section that requires modification for

your application. The other two parts are meant for your knowledge. If you ever need to modify the other two parts, please contact support@vention.io for additional feature requests or support.

- **RESTful http server** that fields requests from the web client and propagates to the MachineApp.
- **Web-socket server** that streams real time data of the MachineApp template to the web client. The web client connects directly to this socket.

State machine

The server is a state machine made up of state nodes (i.e. MachineAppStates). Each node defines some behavior for each type of state transition. The following image demonstrates the lifecycle of a single node in our state machine:



Single node lifecycle

To implement a node, we inherit the MachineAppState class, and define the onEnter method. For example, a simple node that moves us from the current state called "Waiting" to a new state called "DoingSomething" after three seconds might look like this:

Example

```
class WaitingState(MachineAppState):
 def onEnter(self):
  self.startTimeSeconds = time.time()
  self.logger.info('Entered waiting state')

 def update(self):
  if time.time() - self.startTimeSeconds > 3.0:
   self.gotoState('DoingSomething')

 def onLeave(self):
  self.logger.info('Left waiting state')

class DoingSomethingState(MachineAppState):
 def onEnter(self):
  self.logger.info('Entered DoingSomething state')
 ...etc
```

Going between states is as easy invoking the self.gotoState method with the name of the state that you'd like to transition to. Any other business logic simply gets implemented by overriding the defined methods.

State machine implementation (MachineAppEngine)

After understanding how to build Vention's separate state nodes, this section covers how to combine the nodes together in our state machine. The state machine is also known as the MachineAppEngine in server/machine_app.py. This class is the core of your MachineApp. It fields requests from the REST server and manages the state transitions of your application. All of the interactions with the REST server are abstracted by its superclass called BaseMachineAppEngine in server/internal/base_machine_app.py (note: there is no need to modify any files in the internal folder. If you are missing any functionality, please reach out to support@vention.io).

Taking our example from before, a MachineAppEngine that handles those two states might look something like:

Example

```
class MachineAppEngine(BaseMachineAppEngine):
 def initialize(self):
  self.machineMotion = MachineMotion('127.0.0.1')
  self.machineMotion.configAxis(1, 8, 250)
  self.machineMotion.configAxis(2, 8, 250)
  self.machineMotion.configAxis(3, 8, 250)
  self.machineMotion.configAxisDirection(1, 'positive')
  self.machineMotion.configAxisDirection(2, 'positive')
  self.machineMotion.configAxisDirection(3, 'positive')

 def onStop(self):
  self.machineMotion.emitStop()

 def onPause(self):
  self.machineMotion.emitStop()

 def beforeRun(self):
  pass

 def afterRun(self):
  pass

 def getMasterMachineMotion(self):
  return self.machineMotion

 def getDefaultState(self):
  return 'Waiting'

 def buildStateDictionary(self):
  stateDictionary = {
   'Waiting': WaitingState(self),
   'DoingSomething': DoingSomethingState(self)
  }

  return stateDictionary
```

A minimal example defines the states that are used in buildStateDictionary, returns the default state in getDefaultState, and defines a single instance of MachineMotion in initialize as the primary controller being communicated to. All other methods are optional helpers. You can get more information about MachineAppEngine in server/machine_app.py.

Runtime configuration

In addition to implementing logic via a state machine, you may want to specify some configurable data to your MachineApp at runtime. This is a very common facet of any MachineApp. For example, you may want to send things like how many times a loop should run, or how long we should wait in our WaitingState, etc.

To do this, you have access to a MachineAppState.configuration and MachineAppEngine.configuration while your state machine is active. This configuration is a python dictionary that is sent by the frontend when you click the "Play" button. We will explain how this data is defined in the Client section later on.

Reacting to Inputs

This section will detail how to enable a logic sequence to trigger from an input parameter. For example, a state change will only be desired upon a button being pushed by an operator. The MachineApp template fulfills this requirement by providing you with the MachineAppState.registerCallback. This function takes as its parameters (1) the machine motion whose topics you want to subscribe to, (2) the topic that you want to subscribe to, and (3) a callback to be invoked when we receive data on that topic.

A topic could be passed directly or alternatively, get the topic of a particular input by its registered name. To register an input for a particualr machine motion, you can do the following in server/machine_app.py:

Register an input

```
class MachineAppEngine(BaseMachineAppEngine):
 def initialize(self):
  self.machineMotion = MachineMotion('127.0.0.1')
  # ... Configure your axes and whatnot ...
  self.machineMotion.registerInput('push_button_1', 1, 1)  # Registering IO module 1 and pin 1 to the name 'push_button_1'

 ... etc
```

From the MachineAppState, the commmand can wait on the push button shown below:

Waiting on input state

```
class WaitingOnInputState(MachineAppState):
 def onEnter(self):
  self.registerCallback(self.engine.machineMotion, self.engine.machineMotion.getInputTopic('push_button_1'), self__onMqttMessageRecei
 def __onMqttMessageReceived(self, topic, msg):
  if msg == 'true':
   self.gotoState('ButtonClickedState')
```

This state machine node waits for a message containing “true” to be published to the fictitious push_button_1 input. Alternatively, an MQTT topic could be passed directly to the MachineAppState.registerCallback function.

Streaming data to the web client (Notifier)

The last part of the server that will be interacted with is the Notifier, located in server/internal/notifier.py. The Notifier provides a mechanism that allows data streaming directly to the web client over a websocket. The streamed data can be obtained in the “Information Console” panel on the frontend.
Each MachineAppState that you initialize has a reference to the global notifier by default, so you should never have to worry about constructing one yourself.

For example, if WaitingState as mentioned earlier is used, information could be sent to the client when the 3 second timeout is complete. An implementation example is shown below:

Example

```
class WaitingState(MachineAppState):
 def onEnter(self):
  self.startTimeSeconds = time.time()
  self.logger.info('Entered waiting state')

 def update(self):
  if time.time() - self.startTimeSeconds > 3.0:
   self.notifier.sendMessage(NotificationLevel.INFO, '3 seconds are up!', { 'waitedFor': 3 })
   self.gotoState('DoingSomething')

 def onLeave(self):
  self.logger.info('Left waiting state')
```

Client

The client is a simple web page that relies on JQuery. It is served up as three separate JavaScript files and two separate CSS files by the Python http server. The files that you should concern yourself with mostly are:

- client/ui.js - Contains all custom frontend logic
- client/widgets.js - Contains widgets that are helpful for building forms
- client/styles/ui.css - Contains all custom frontend styles

Configuration editor

As mentioned in the server's configuration section, a runtime configuration can be published to the MachineApp engine when the "play" button is clicked. This configuration is defined entirely on the frontend in client/ui.js.

An example would be sending a "wait time in seconds" to be sent to the server, "WaitingState". In client/ui.js, it could be implemented like so:

Example

```
function getDefaultConfiguration() {
 return {
  waitTimeSeconds: 3.0
 }
}

function buildEditor(pConfiguration) {
 const lEditorWrapper = $('<div>').addClass('configuration-editor'),
  lFullSpeedEitor = numericInput('Wait Time (seconds)',  pConfiguration.waitTimeSeconds, function(pValue) {
    pConfiguration.waitTimeSeconds = pValue;
   }).appendTo(lEditorWrapper);

 return lEditorWrapper;
}
```

getDefaultConfiguration defines the data that will be sent to the backend regardless of whether or not the user edits any of it in the editor. buildEditor constructs a user interface for our data using the widgets from client/widgets.js. The widgets in client/widgets.js are available to help build a custom UI interface.

In the backend, under "WaitingState", it's possible to access the waitTimeSeconds variable like so:

Backend of Waiting State

```
class WaitingState(MachineAppState):
 def onEnter(self):
  self.waitTimeSeconds = self.configuration["waitTimeSeconds"]
 ...
```

Updating the UI from streamed data

As explained in the server's notifier section, the server can stream data to the client while it is running via a WebSocket. The client establishes this connection in client/index.js when the page is loaded. When a message is received from this connection, add it to the "Information Console" with an icon describing what type of message it is (this happens in client/index.js). The message is then passed to the onNotificationReceived callback in client/ui.js. This section will enable custom UI creation for various messages.

For an example, if WaitingState from the previous section looked like this:

Example

```
class WaitingState(MachineAppState):
 def onEnter(self):
  self.waitTimeSeconds = self.configuration["waitTimeSeconds"]
  self.notifier.sendMessage(NotificationLevel.INFO, 'Received wait time', { waitTimeSeconds: self.waitTimeSeconds })
 ...
```

The onNotificationReceived could then be implemented in client/ui.js like the following example to append the “waitTimeSeconds” variable to the custom container:

Append variable to custom container

```
function onNotificationReceived(pLevel, pMessageStr, pMessagePayload) {
 const lCustomContainer = $('#custom-container');
 if (pMessagePayload.waitTimeSeconds) {
  lCustomContainer.append($('<div>').text(pMessagePayload.waitTimeSeconds));
 }
}
```