Model Creator:

Technical Documentation

Written By: John Knox

Date: December 19, 2019

Version: 1.0

Outline:

Abstract:

This is documentation for the model creator API. The goal of this project is to allow users to create an agent-based model directly from the Indra website. All the back end (python) code for this project sits in two different scripts: api_endpoints.py, and model_creator_api.py. This document includes the steps to configure the API, a description for the functionality of the model creator scripts, and the future plans for this agent-based model contained within the indras project.

Getting Started:

Note 1: Links to various resources are directly embedded within this document.

Note 2: This document describes the operation of the agent-based model running a version of Ubuntu Desktop on a Windows PC.

Cloning the indras_net repository from GitHub is the first thing that should be done since this directory will be needed for running the agent-based model. There is a setup process that must be completed on your Linux operating system (such as Ubuntu). If you don't have Ubuntu, it can be downloaded here. Starting up Ubuntu should bring you to a page that looks like **Figure 1** below:

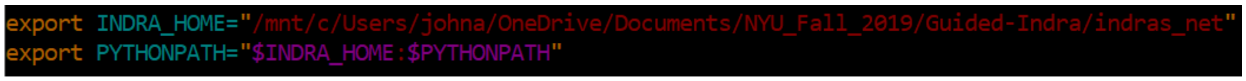
```
root@DESKTOP-67TDOSB: ~
root@DESKTOP-67TDOSB:~#
```

Figure 1: Ubuntu start up screen (root directory)

Updating .bashrc:

Perform the following steps:

1. After opening Ubuntu, type command: *vi .bashrc*
   a. If this doesn't work, make sure the current directory is correct
      i. type command: *cd ~*
      ii. Then type command: *ls -A*
      iii. If .bashrc shows up in the list of files that appear, the directory is correct
2. When prompted, press Shift + E to edit the document
3. To insert text into the document, press "i" before typing
4. Insert the following lines
   a. export INDRA_HOME="the directory to the cloned indra repository"
   b. export PYTHONPATH="$INDRA_HOME:$PYTHONPATH"
   c. The text should appear as this:



```
export INDRA_HOME="/mnt/c/Users/johna/OneDrive/Documents/NYU_Fall_2019/Guided-Indra/indras_net"
export PYTHONPATH="$INDRA_HOME:$PYTHONPATH"
```

5. Press *Ctrl+Alt+i* to stop inserting text into the document
6. Type *:wq* then *Enter* to save and exit the document

Downloading Required Libraries:

Perform the following steps:

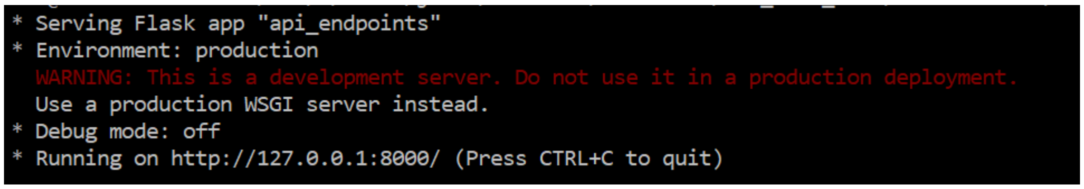
1. In Ubuntu, type command: *cd $INDRA_HOME*

    a. After having updated .bashrc, *cd $INDRA_HOME* will automatically take you to the indra repository
2. Type command: *make create_dev_env*
3. Wait for the dependencies to be done downloading
    a. Note: You may be prompted to add additional libraries such as Python3. If this happens, download any additional dependencies and run *make create_dev_env* again

Creating a Model from the API:

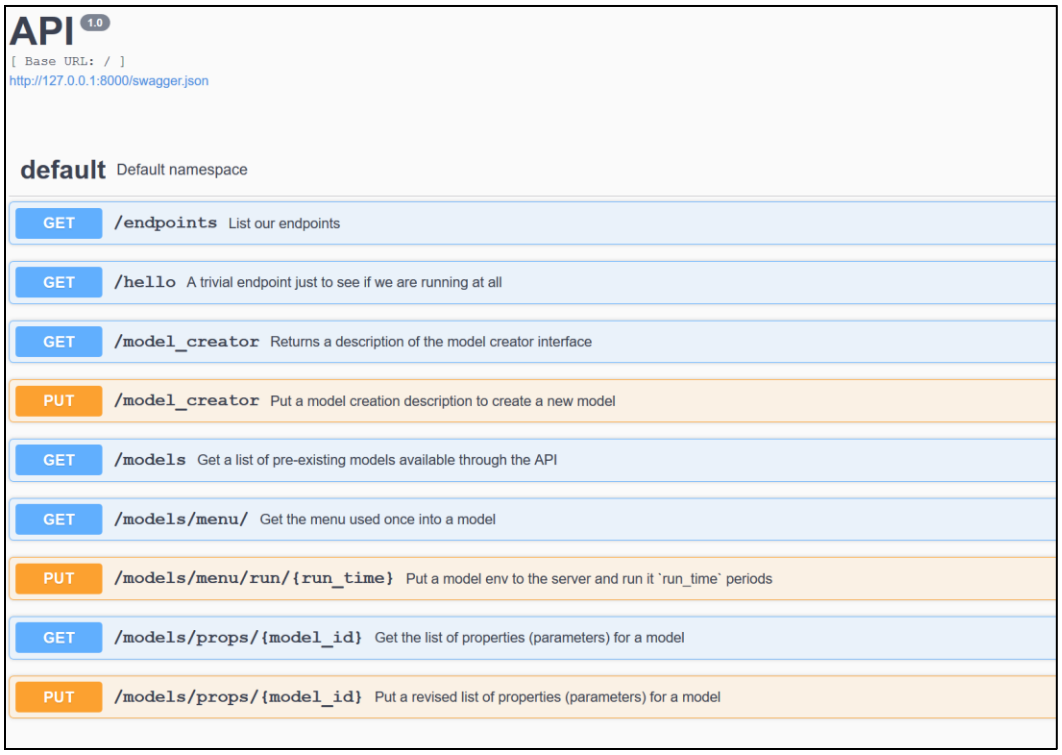
Perform the following steps:

1. In Ubuntu, navigate into the indras_net directory by typing command: *cd $INDRA_HOME*
    a. After having updated .bashrc, typing command: *cd $INDRA_HOME* will automatically take you to the indra repository
2. Once in the repository, type command: *cd /APIServer*
3. Type command: *./api.sh*
    a. After typing the command, Ubuntu should look like this:



```
* Serving Flask app "api_endpoints"
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:8000/ (Press CTRL+C to quit)
```

4. Copy and paste the displayed IP address into your web browser
    a. The webpage should look like **Figure 2**:


API 1.0

[Base URL: /]

http://127.0.0.1:8000/swagger.json

default Default namespace

GET /endpoints List our endpoints

GET /hello A trivial endpoint just to see if we are running at all

GET /model_creator Returns a description of the model creator interface

PUT /model_creator Put a model creation description to create a new model

GET /models Get a list of pre-existing models available through the API

GET /models/menu/ Get the menu used once into a model

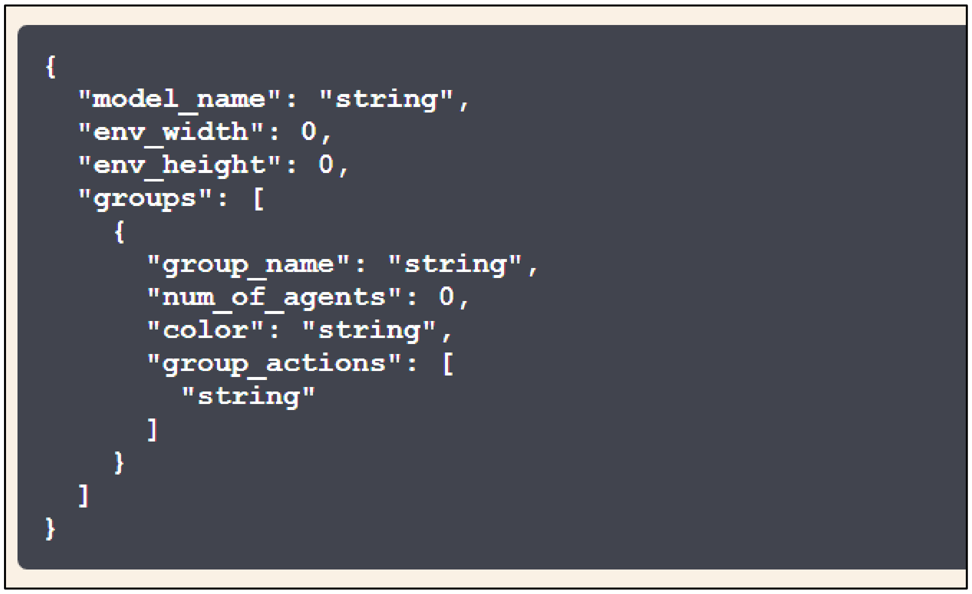
PUT /models/menu/run/{run_time} Put a model env to the server and run it `run_time` periods

GET /models/props/{model_id} Get the list of properties (parameters) for a model

PUT /models/props/{model_id} Put a revised list of properties (parameters) for a model

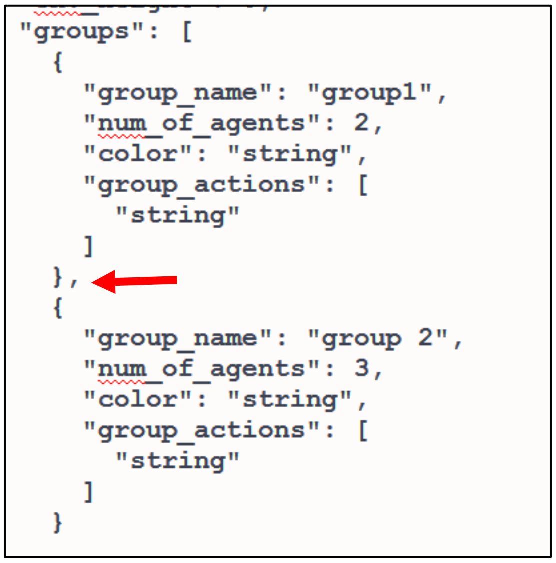
Figure 2: Indra API Homepage

5. Click on **PUT** /model_creator and press the "Try it out" button
6. Update the JSON variables listed on the Default Model Parameter List shown in **Figure 3. This list shows only one group delimited by the square bracket, [].**
    a. To add additional groups to the model, copy and paste the group parameters and insert a comma between the sets, as shown in **Figure 4.**
    b. Important Notes:
        i. the *env_width* and *env_height* need to be <u>greater than 0</u> if the *num_of_agents* in a group is <u>greater than 0.</u> Otherwise the program will crash.
        ii. *model_name* and *group_name*(s) must have unique strings. Leaving the default name may cause the program to crash.



```
{
  "model_name": "string",
  "env_width": 0,
  "env_height": 0,
  "groups": [
    {
      "group_name": "string",
      "num_of_agents": 0,
      "color": "string",
      "group_actions": [
        "string"
      ]
    }
  ]
}
```

Figure 3:Default Model Parameter List



```
"groups": [
  {
    "group_name": "group1",
    "num_of_agents": 2,
    "color": "string",
    "group_actions": [
      "string"
    ]
  },
  {
    "group_name": "group 2",
    "num_of_agents": 3,
    "color": "string",
    "group_actions": [
      "string"
    ]
  }
```

Figure 4: Format for Multiple Groups

7. After setting the model parameters, press the blue execute button **(Figure 5)**

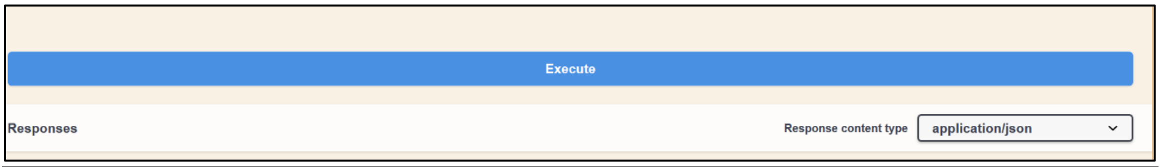
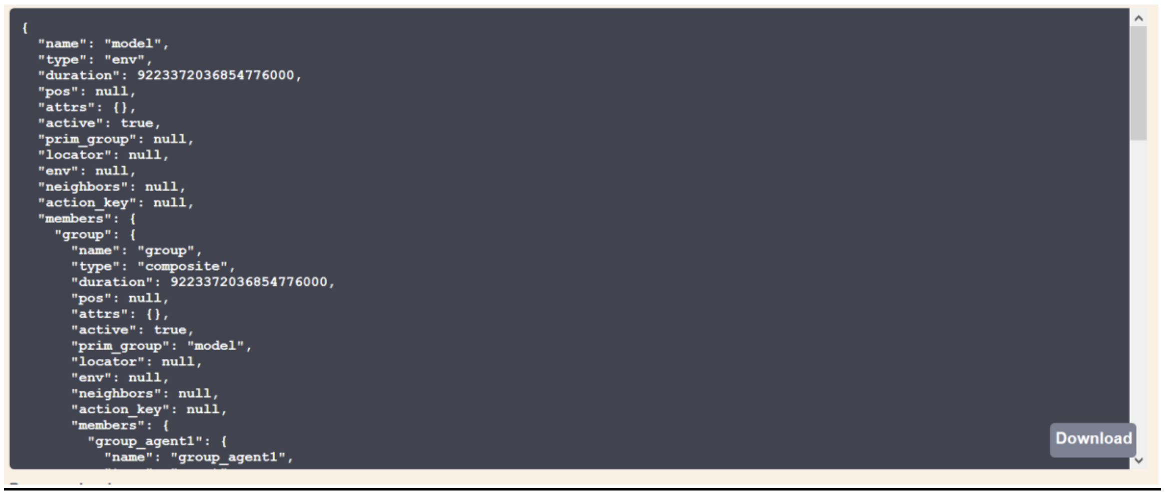


Figure 5:Execute Button

8. An example of a typical output is shown on **Figure 6**:



```
{
  "name": "model",
  "type": "env",
  "duration": 9223372036854776000,
  "pos": null,
  "attrs": {},
  "active": true,
  "prim_group": null,
  "locator": null,
  "env": null,
  "neighbors": null,
  "action_key": null,
  "members": {
    "group": {
      "name": "group",
      "type": "composite",
      "duration": 9223372036854776000,
      "pos": null,
      "attrs": {},
      "active": true,
      "prim_group": "model",
      "locator": null,
      "env": null,
      "neighbors": null,
      "action_key": null,
      "members": {
        "group_agent1": {
          "name": "group_agent1",
```

Figure 6: Model Output Example

Some Helpful Tips:

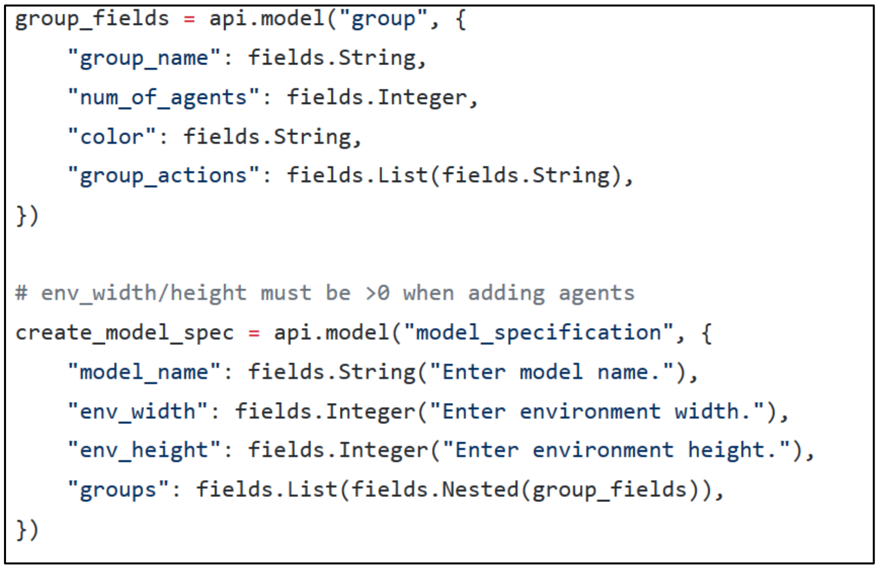
1. When adding new parameters for further model customization, the indra project uses certain "fields", or variables, from the flask-REST Plus library. The documentation for the fields can be found here.
2. For a better understanding of a basic model, review this complete example from the Indra repository, basic.py. Other, more advanced, agent-based models can be viewed in the models folder in the repository.
3. For a better understanding of the functionality behind the classes that make up a model, review the scripts located in the indra folder of the indras_net repository.

Current Functionality:

api_endpoints.py:

Link to Script

This script holds all the endpoints used to run agent-based models on the web. Endpoints tell the API where to access certain resources for the software interacting with it. The code for model creation within this script is made up of two JSON lists and a class. One JSON list defines the model variable specification (create_model_spec), and the other JSON defines the group variable specification (group_fields). The *ModelCreator* class is made up of a *put* and *get* function. Inside the model specification, the model name, environment width and height, and the groups contained in the environment are established. The working group parameters are made up of the name, and number. The color and group actions are a part of the JSON list but are currently not functional (December 2019). **Figure 7** shows the current model and group specification types:



```
group_fields = api.model("group", {
    "group_name": fields.String,
    "num_of_agents": fields.Integer,
    "color": fields.String,
    "group_actions": fields.List(fields.String),
})

# env_width/height must be >0 when adding agents
create_model_spec = api.model("model_specification", {
    "model_name": fields.String("Enter model name."),
    "env_width": fields.Integer("Enter environment width."),
    "env_height": fields.Integer("Enter environment height."),
    "groups": fields.List(fields.Nested(group_fields)),
})
```

Figure 7: Model and Group Specification Types (December 2019)

The two JSON lists are used in the *put* function, which returns the *put_model_creator* class which in turn creates the agent-based model. The *ModelCreator* class also has a *get* function which returns a description of the model creator interface. The rest of the code in this script consists of the endpoints for the model creator, and other parts of the Indra project so it can run in the web.

model_creator_api.py:

Link to Script

This script handles the creation of user-made models in the indras project. The function *put_model_creator* takes in the model features from the *api_endpoints* and returns the JSON conversion of the model. Before returning the JSON conversion of the model, the groups contained within its environment are created through the *put* method contained within the *CreateGroups* class. The *put* function creates agents and adds them to a Composite (group) class and returns the group list. Agents are created using the *addAgents* method where they are given names and actions to perform when the model is running. The actions for each agent are generated by the *generate_func* function. This function allows each agent to check how many agents from its group are within a certain range.

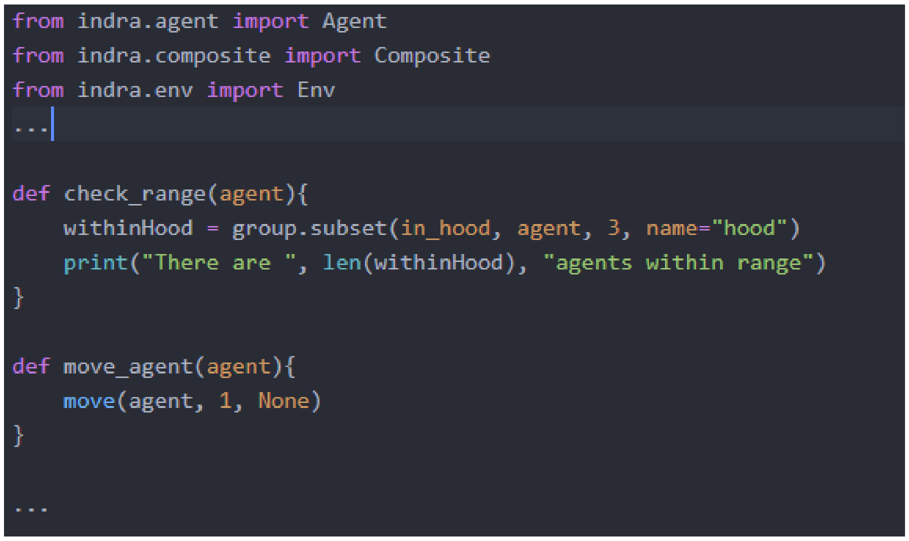
<u>Future Work:</u>

<u>This section describes three possible areas for improving the model creation functionality. These improvements include Agent Action Functionality, Automated Testing and Front-End Interface.</u>

Improvements to the Agent Actions Function

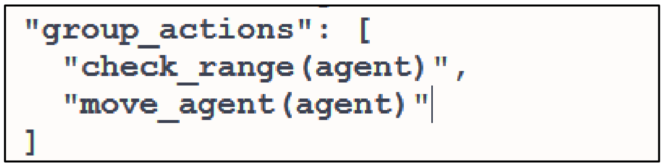
In the *model_creator_api.py* script, the *generate_func* function allows agents to count how many agents from its group are within a specified range. The goal for this function however is to run more actions based on the *group_actions* list in *api_enpoints.py*. One technique for containing all the actions that an agent can perform is by creating a new script, such as *model_creator_actions.py*, having pseudocode shown in **Figure 8.** This new script will contain functions that represent the actions an agent can perform in a model. The *group_actions* list will contain these function names as part of that script as shown in **Figure 9**. The *group_actions* list will be passed into the agents "*attrs*" parameter as shown in **Figure 10**. It's important to note that "agent" is the only parameter that can be passed into *generate_func*. *The generate_func* function will then call the *"attrs"* using the agent parameter and loop through the *group_actions* list. **Figure 11** shows the pseudocode for the *exec()* function that executes each action specified. This function turns functions in string form to be executed. Filling *group_actions* with strings instead of the actual function will make it easier to edit by the front-end team. If normal functions are found to be easier, go ahead and use them.

Note: The hardcoded values in *group.subset* and *move* are the pixel range and pixel steps respectively. These should be parameters.



```
from indra.agent import Agent
from indra.composite import Composite
from indra.env import Env
...

def check_range(agent){
    withinHood = group.subset(in_hood, agent, 3, name="hood")
    print("There are ", len(withinHood), "agents within range")
}

def move_agent(agent){
    move(agent, 1, None)
}

...
```

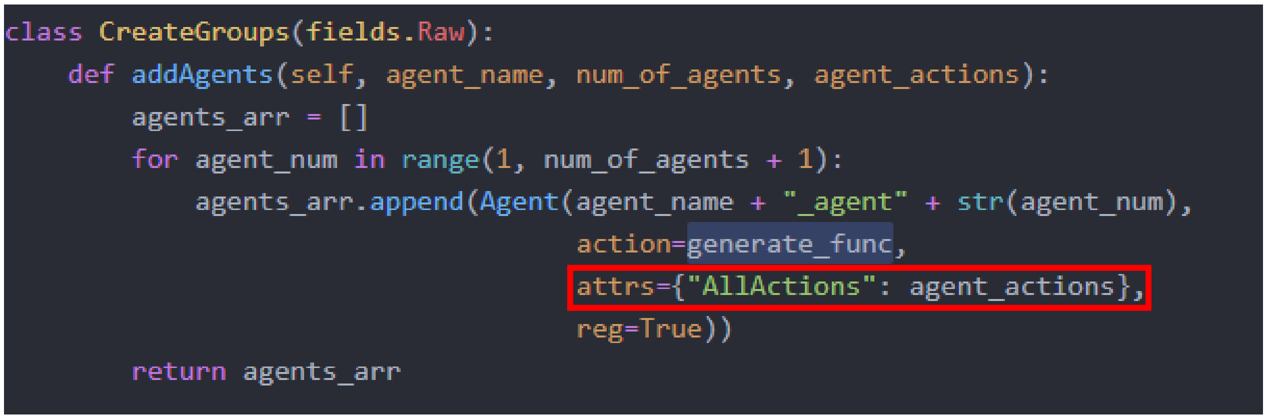
Figure 8: Pseudocode for model_creator_actions.py



```
"group_actions": [
  "check_range(agent)",
  "move_agent(agent)"
]
```

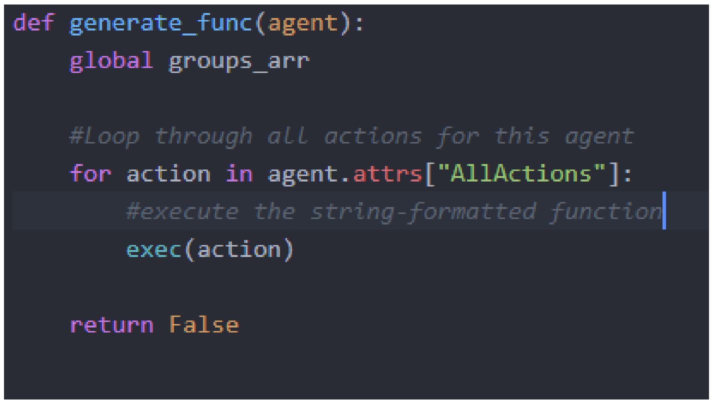
Figure 9:Sample group_actions list from the Indra API

Note: “agent_actions” represents the group_actions list



```
class CreateGroups(fields.Raw):
    def addAgents(self, agent_name, num_of_agents, agent_actions):
        agents_arr = []
        for agent_num in range(1, num_of_agents + 1):
            agents_arr.append(Agent(agent_name + "_agent" + str(agent_num),
                                    action=generate_func,
                                    attrs={"AllActions": agent_actions},
                                    reg=True))
        return agents_arr
```

Figure 10: The "attrs" parameter containing the group_actions list

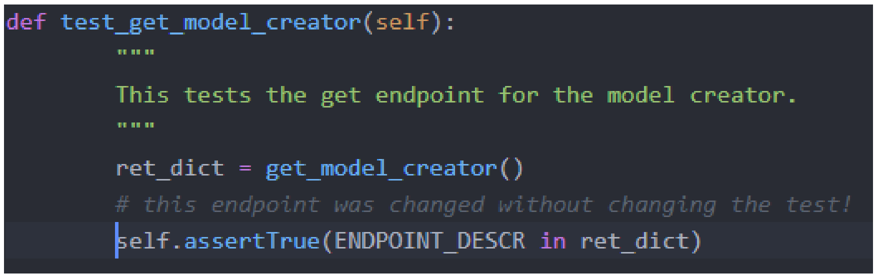


```
def generate_func(agent):
    global groups_arr

    #Loop through all actions for this agent
    for action in agent.attrs["AllActions"]:
        #execute the string-formatted function
        exec(action)

    return False
```

Figure 11: Pseudocode for generate_func function

Adding Automated Tests to Model Creator

Within the APIServer folder in the repository, there is a folder called *test.* The *test* folder contains a script called *test_model_creator_api.py*. This script, and other test scripts, will hold at least one test for each function used in *model_creator_api.py*, as well as two additional functions for resetting test results. In the current version, this script contains three functions, namely, *setUp()*, *tearDown()*, and *test_get_model_creator()*. The first two functions, *setUp()* and *tearDown()*, will clear any past test results, so the output is reliable. All functions after these two are associated with testing a function in *model_creator_api.py*. Every test function should include a description at the top followed by the test being run. An example is shown in **Figure 12**. This test function will return a value of **true** if the variable *ENDPOINT_DESCR* is within the function (which it does if you check *model_creator_api.py*).



```
def test_get_model_creator(self):
        """
        This tests the get endpoint for the model creator.
        """
        ret_dict = get_model_creator()
        # this endpoint was changed without changing the test!
        self.assertTrue(ENDPOINT_DESCR in ret_dict)
```

Figure 12: Test Function in test_model_creator_api.py

It is expected that some test cases will be more complex. It is important to mention that any new script added to the project, should have a test script associated with it. To achieve a better understanding of how these test function should be constructed, review the following YouTube link which discusses test-driven development.

The Front-End Interface

Right now, the back-end (Python) code needs user inputs in order to work. Another goal of this project is to allow the user to create a model through a front-end user interface. Instead of manually typing the JSON values into the API, the values will be filled in through the interactions with a graphical user interface (UI). **Figure 13** shows a concept of a simple UI for adding groups and associated actions. Numeric values for the environment's pixel width (W) and height (H) would be entered in the top left corner of the screen. A user would add a new group to the model by pressing the plus sign at the bottom of the left column. As groups are added, an icon representing the new group would be added across the bottom of the panel. For the example shown in **Figure 13**, there is only one group represented by the blue circle. The characteristics for each group can be edited by clicking on their respective icon across the bottom. For example, **Figure 14** shows the UI after the blue circle icon was selected for editing. In this mode, the user would drag actions onto the large icon. The selection for different group actions are available in the edit mode. For the example shown in **Figure 14**, the selectable actions appear as icons in the rightmost column. **Figure 15** shows the selection of the "check range" action that is dragged in the blue circle group. Additional actions include, but will not be limited to, move agent, reproduce, and remove agent to name a few. Based on the actions and parameters, such as "Range" in **Figure 15**, the JavaScript used for the UI will generate the string representation of the action function and add it to the *group_actions* list in *api_endpoints.py*. As of December 2019, implementation of the front-end began and is still in its early stages. Link to the code can be found here.

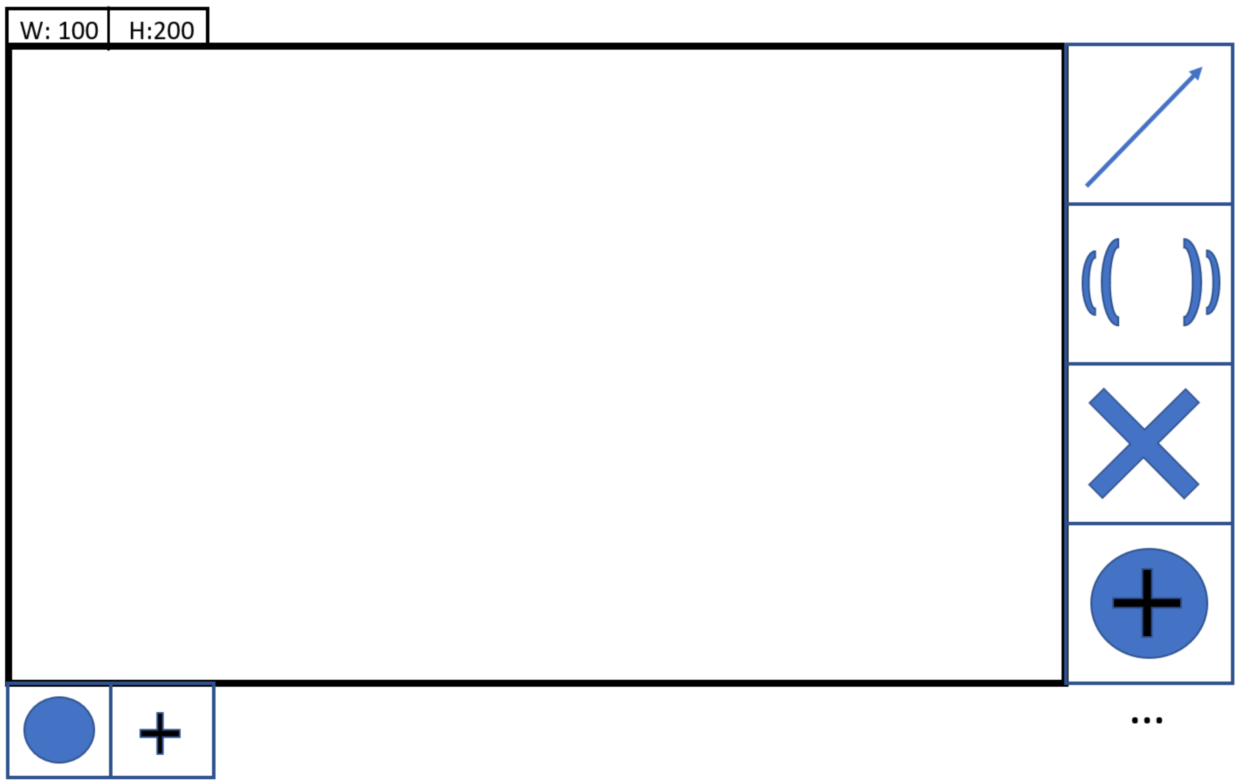


Figure 13 Basic graphical user interface

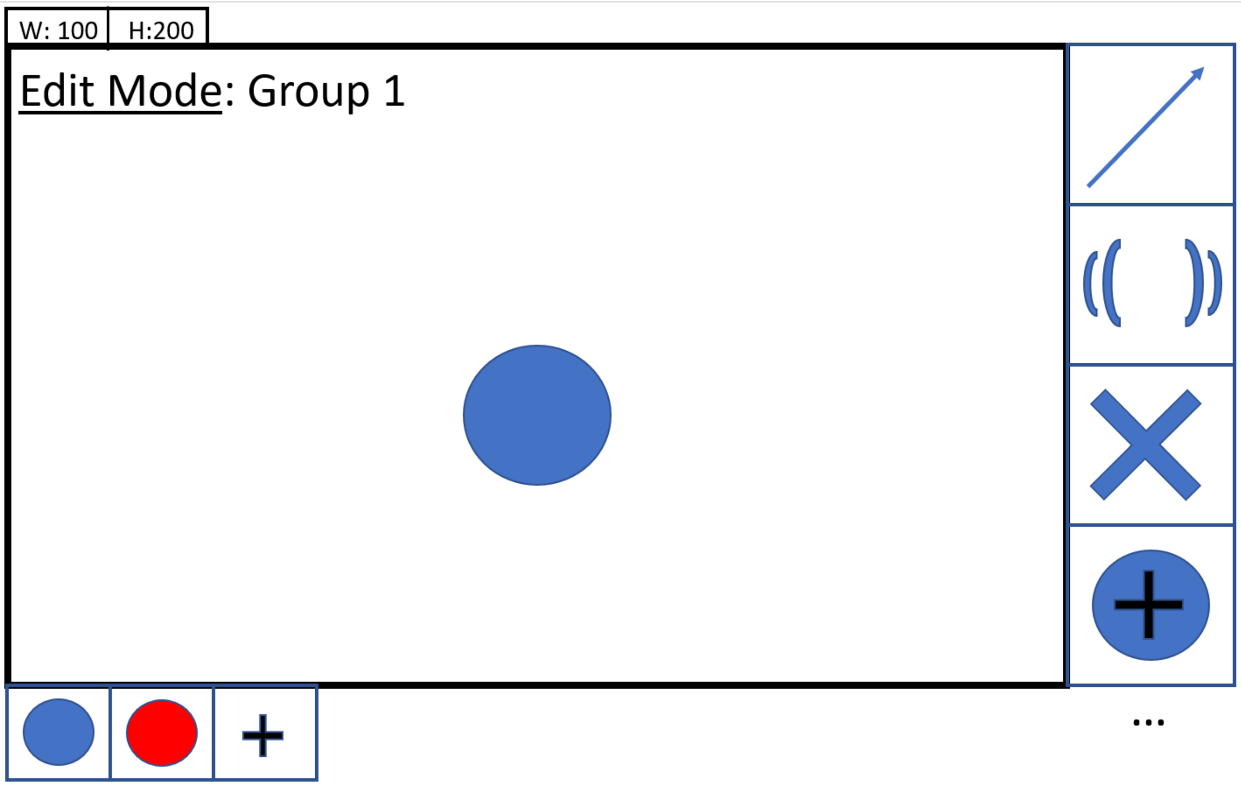


Figure 14 UI in edit mode

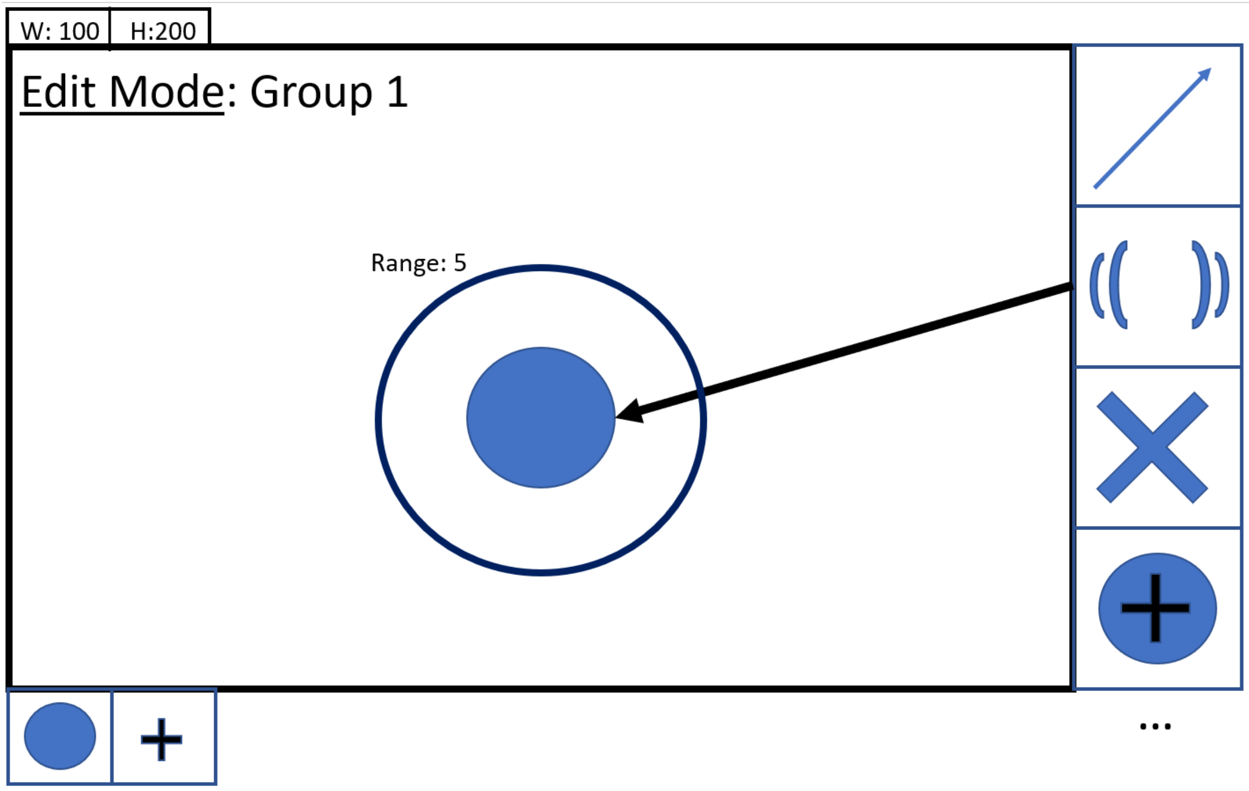


Figure 15 Selection of the "check range" action dragged onto the group