

content_authoring_backend.py

Lines 1-3 of the *content_authoring_backend.py* are the packages used in code. The *flask* package is what sets up server for POST, GET, and PUT requests to and from the client web application. From *flask* we import *Flask*, and *request*.

```
1. from flask import Flask, request
2. from regraph import NXGraph, Rule
3. from flask_cors import CORS
4.
5. app = Flask(__name__) #initializing Flask app
6. CORS(app) #adding flask app to CORS to allow communication
```

On line 5 of the code, we initialize the application. On line 2, we import *regraph*. This is the package which provides us with functions and tools to manipulate and modify the graph.

On line 3, we import a very important package called *flask_cors* from which we import CORS. Adding the app to CORS as shown on line 6, will allow traffic to and from your client application. If you remove it, you will get error code 500.

```
graph = NXGraph() #initializing the graph
```

On line 8, we initialize the *graph*, which is a *NetworkX* type of graph. To read more about NetworkX, follow this link: <https://networkx.org/>

When you have everything set up and you have installed all the packages, run the Flask app. Your server will be hosted on localhost or your local address.

```
$ python content_authoring_backend.py
* Serving Flask app "content_authoring_backend" (lazy loading)
* 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://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Every function call you make to this Flask app will return a graph in *D3* JSON format which looks something like the following:

```
{links: [], nodes: []}
```

The above is a JSON object, within which you have an array of links, which are the arrows or edges from one node to another, and the nodes will contain the *Dialogue Objects*.

Each *Dialogue Object* for our use case has the following information below. With exception of the *id*, *position*, *data*: {'label': ""}, and *type*, a developer is free to add as many key-value pairs where values are of any data type such as arrays or objects with more key-value pairs.

```
let init_node = {
  id: "000",
  DialogText: "Hello, nice to meet you?",
  alternates: ['Hi there', "Hello, my name is..."],
  data: {label: "000 Hello, nice to meet you?"},
  NextDialogID: [""],
  position: {x: 200, y: -100},
  type: 'input',
  section: "Greetings"
}
```

Table 1 The attributes, their data type, and a description of their use. The ones in bold are required and should not be removed.

Attribute	Data Type	Description
id	string	This is the unique identifier for each node and dialogue object that will be used to edit the node or even delete it.
DialogueText	string	This is the dialogue in text or the interview question.
Alternates	Array of strings	These are the alternate versions of the dialogue text that the script author can specify if they would like to add variability to dialogues in case a user asks the system to repeat the question/dialogue.
data: {label: ''}	Object	This is a ReactFlow requirement to display the label of the node. In the AuthoringTool interface, each node has the dialogue ID and the dialogue displayed. A developer is free to change this to whatever they prefer.
NextDialogueID	Array of strings	These are the IDs of the successors of a node; the nodes towards the end of the
position	Object {x: int, y:int}	The position of the node in the form of (x,y) coordinate
type	String	This is the type of node as described by ReactFlow. There are 3 types of nodes defined in ReactFlow. The 'input' node has no point for an incoming edge, only outgoing. This is usually the start node in the graph. The 'output' node has an incoming point, but no outgoing point. The 'default' node type has both incoming and outgoing points to allow for incoming and outgoing edges. Please read more here: https://reactflow.dev/docs/examples/overview/
Section	String	This is the variable that a developer can use to categories or group questions. For example, an interview can have several segments such as Greetings, Technical questions, Personal and Behavioral questions, questions about degree and certifications, etc.

Below are details of what each function does, what the input is and what is the output. Along with the explanation, a visual of each function is provided.

INITIALIZATION

The following on Line 9 initializes an empty graph which is in a global variable. This graph is initially empty if you chose the “CREATE NEW SCRIPT” option. This makes a HostGraph which is currently empty represented here as `<empty_graph>`.


```
graph = NXGraph()
```

In D3, the JSON object of the HostGraph is `{links: [], nodes: []}`

1. `init()`

- This function adds a node to the empty HostGraph initialized above.
- Input: *Dialogue Object* via PUT request
- Output: HostGraph in D3 format

Example graph visual,

`<empty_graph> :=` 

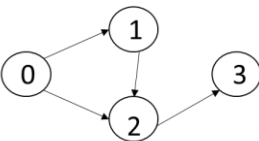
Updated *HostGraph* =

```
{
  links: [],
  nodes: [{id: '0'}]
}
```

2. `init_graph()`

- This function initializes the empty HostGraph with a previously created graph or template read from a JSON file or to allow the user to not start from scratch or to edit a previously made script. This function is called when the user chooses “CREATE FROM TEMPLATE”.
- Input: JSON Array of *Dialogue Objects* where the source node is connected to the target node
- Output: *HostGraph* in D3 format

Example graph visual,

`<empty_graph> :=` 

Updated *HostGraph* =

```
{
  links: [(source: '0', target: '1'),
          (source: '1', target: '2'),
          (source: '0', target: '2'),
  ],
  nodes: [{id: '0'}, {id: '1'}, {id: '2'}, {id: '3'}]
}
```

```

    ('source': '2', 'target': '3']],
    nodes: [{ 'id': '0'},
              { 'id': '1'},
              { 'id': '2'},
              { 'id': '3'}]
  }

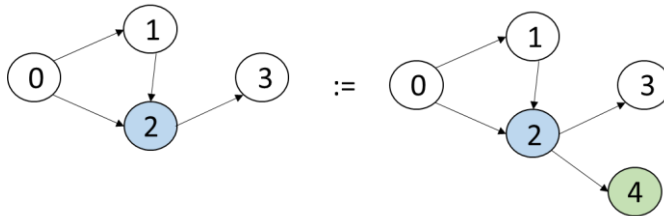
```

INSERT

1. insert_node()

- This function inserts a node from a current node and connects the current node to the added new node. Each node in the UI has 3 options—to edit, add node after, or to delete the node. When the ADD option is chosen, then this function is called.
- Input: Current node ID, new *Dialogue Object*
- Output: *HostGraph* in D3 format

Example in graph visual, where the blue node (node 2) is the current node after which we want to add a new node (node 4 in green). The user can then draw an edge from any node to node 4 using the create edge function below.



Updated *HostGraph* =

```

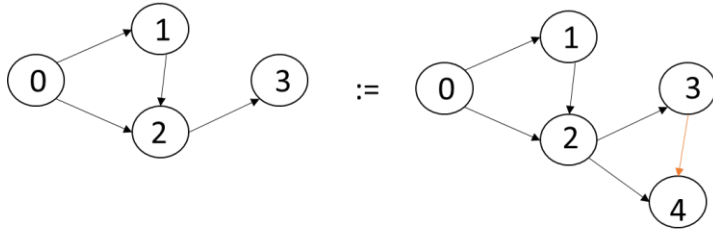
{
  links: [('source': '0', 'target': '1'),
          ('source': '1', 'target': '2'),
          ('source': '0', 'target': '2'),
          ('source': '2', 'target': '3'),
          ('source': '2', 'target': '4')],
  nodes: [{ 'id': '0'},
            { 'id': '1'},
            { 'id': '2'},
            { 'id': '3'},
            { 'id': '4'}]
}

```

2. create_edge()

- This function creates an edge between two nodes, the source node and the target node
- Input: source node ID and target node ID
- Output: *HostGraph* in D3 format

Example in graph visual, where an edge is created between the source node (node 3) and the target node (node 4)



Updated *HostGraph* =

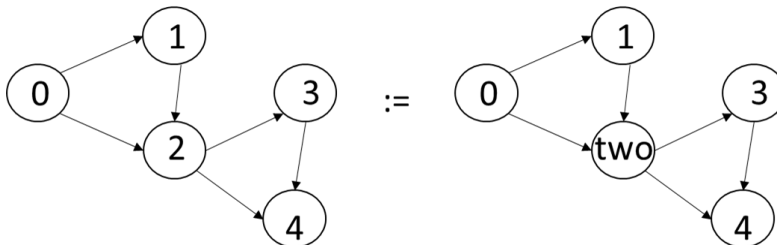
```
{
  links: [(‘source’: ‘0’, ‘target’: ‘1’),
    (‘source’: ‘1’, ‘target’: ‘2’),
    (‘source’: ‘0’, ‘target’: ‘2’),
    (‘source’: ‘2’, ‘target’: ‘3’),
    (‘source’: ‘2’, ‘target’: ‘4’),
    (‘source’: ‘3’, ‘target’: ‘4’)],
  nodes: [{‘id’: ‘0’},
    {‘id’: ‘1’},
    {‘id’: ‘2’},
    {‘id’: ‘3’},
    {‘id’: ‘4’}]
}
```

UPDATE

1. relabel_node

- This function re-label’s or changes the ID of the chosen node
- Input: node id to re-label, and the new ID to which this node will be relabeled
- Output: HostGraph in D3 Format.

Example in graph visual, where a node (node 2) is relabeled (to node ‘two’). The backend automatically changes all places in the graph referring to the node 2, to ‘two’



Updated *HostGraph* =

```
{
  links: [(‘source’: ‘0’, ‘target’: ‘1’),
    (‘source’: ‘1’, ‘target’: ‘two’),
    (‘source’: ‘0’, ‘target’: ‘two’),
    (‘source’: ‘two’, ‘target’: ‘3’),
    (‘source’: ‘two’, ‘target’: ‘4’),
    (‘source’: ‘3’, ‘target’: ‘4’)],
  nodes: [{‘id’: ‘0’},
    {‘id’: ‘1’},
    {‘id’: ‘two’},
    {‘id’: ‘3’},
    {‘id’: ‘4’}]
}
```

```

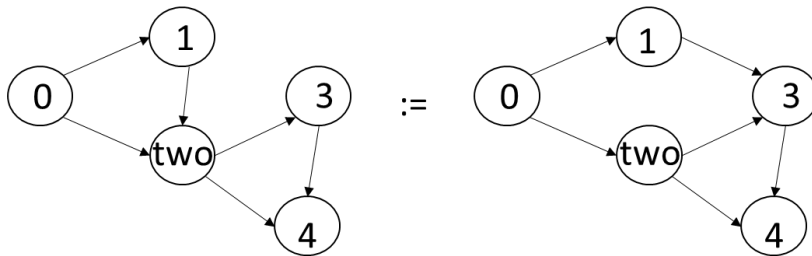
nodes: [{ 'id': '0' },
        { 'id': '1' },
        { 'id': 'two' },
        { 'id': '3' },
        { 'id': '4' } ]
}

```

2. update_edge

- This function can be called to either change the target of an edge (moving it and attaching the end to another node) when a user drags and drops the end onto another node or updating the text on the edge for the edge condition.
- Input: (source: string, target: string, label: string)
- Output: *HostGraph* in D3 format

Example in graph visual, showing two cases. One where only the edge target has been changed.



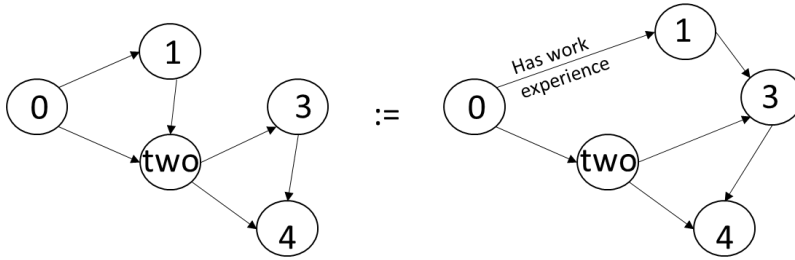
Updated *HostGraph* =

```

{
  links: [( 'source': '0', 'target': '1' ),
          ( 'source': '0', 'target': 'two' ),
          ( 'source': '1', 'target': '3' ),
          ( 'source': 'two', 'target': '3' ),
          ( 'source': 'two', 'target': '4' ),
          ( 'source': '3', 'target': '4' ) ],
  nodes: [{ 'id': '0' },
          { 'id': '1' },
          { 'id': 'two' },
          { 'id': '3' },
          { 'id': '4' } ]
}

```

Another case where a label representing a condition for transition has been added. For example, if an interviewee is asked if they have past work experience, and they say no, then the condition no will lead to another direction. Here, the example says that if the interviewee does not have past work experience, then the system will take the path from 0, 1, 3, and 4.



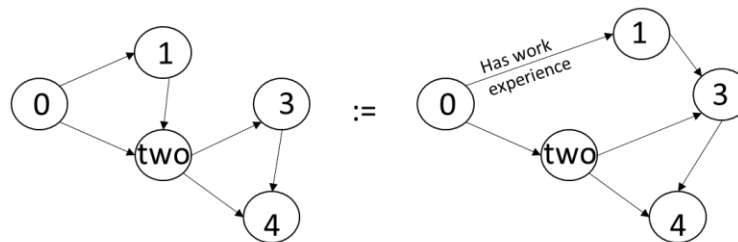
Updated *HostGraph* =

```
{
  links: [(‘source’: ‘0’, ‘target’: ‘1’, label: ‘Has work experience’),
    (‘source’: ‘1’, ‘target’: ‘3’),
    (‘source’: ‘0’, ‘target’: ‘two’),
    (‘source’: ‘two’, ‘target’: ‘3’),
    (‘source’: ‘two’, ‘target’: ‘4’),
    (‘source’: ‘3’, ‘target’: ‘4’)],
  nodes: [{‘id’: ‘0’},
    {‘id’: ‘1’},
    {‘id’: ‘two’},
    {‘id’: ‘3’},
    {‘id’: ‘4’}]
}
```

3. update_node_attrs

- The form in the UI on the front-end allows users to change the attributes (see attributes in Table 1). This function is called when any change is made to a node.
- Input: The *Dialogue Object* or updated JSON object of the node.
- Output: *HostGraph* in D3 format.

Example in graph visual shows we can change the attributes of the node. Here the alternates have been removed and the section name has been changed.



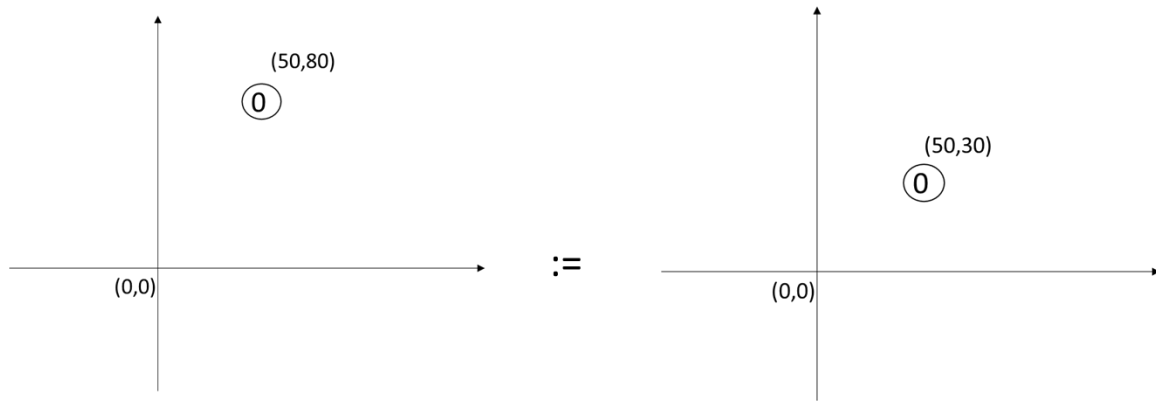
```
id: "1",
DialogText: "Please tell me more about your work experience",
alternates: ["Tell me more about your work experience",
  "Please tell me about your previous job"],
data: {label: "1 Please tell me more about your work experience"},
NextDialogID: ["1", "two"],
position: {x: 200, y: -100},
type: 'input',
section: "PostWorkExperience"
```

```
id: "1",
DialogText: "Please tell me more about your work experience",
alternates: ["",
  ""],
data: {label: "1 Please tell me more about your work experience"},
NextDialogID: ["1", "two"],
position: {x: 200, y: -100},
type: 'input',
section: "PastWorkExperience"
```

4. on_position_change()

- This function is called when a node is dragged from one position to another. The x and y coordinates are changed.
- Input: Dialogue Object with updated node position
- Output: HostGraph in D3 format.

Example visual, the node is placed on the screen, where the origin can be at the bottom left, top-left, or the center. Here we assume that the origin is in the center of the screen. On moving the node, the (x, y) coordinates are changed and updated on the backend graph.

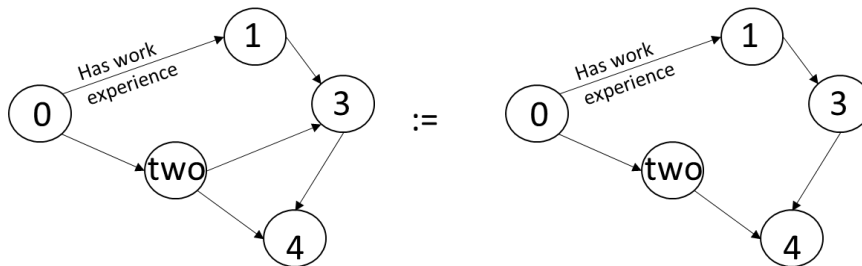


DELETE

1. remove_edge

- This function removes the specified edge where an edge is defined as (source:string, target:string)
- Input: (source, target)
- Output: HostGraph in D3 format.

Example in graph visual shows the updated graph after specified edge is removed. Here, the edge between node two (source) and node 3 (target) is removed.



Updated *HostGraph* =

```
{
  links: [(‘source’: ‘0’, ‘target’: ‘1’, label: ‘Has work experience’),
    (‘source’: ‘1’, ‘target’: ‘3’),
    (‘source’: ‘0’, ‘target’: ‘two’),
    (‘source’: ‘two’, ‘target’: ‘4’),
    (‘source’: ‘3’, ‘target’: ‘4’)],
  nodes: [{‘id’: ‘0’},
```



```

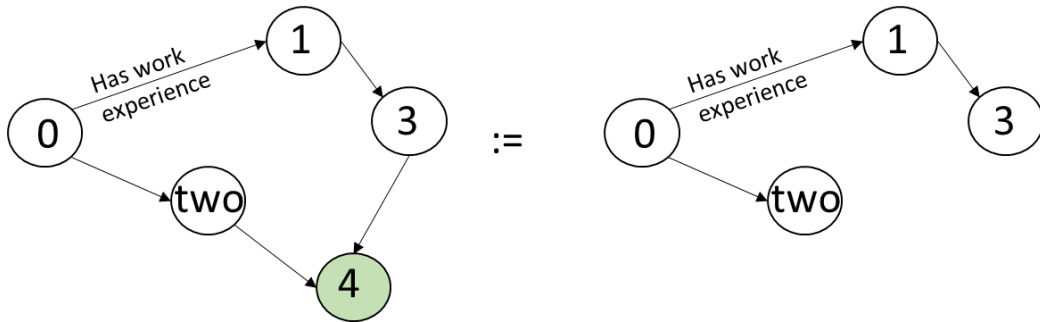
    {'id': '1'},
    {'id': 'two'},
    {'id': '3'},
    {'id': '4'}]
}

```

2. delete_node

- This function deletes the specified node from the graph. It also deletes any edges connected to and from this node. It also updates the attribute 'NextDialogueID' from its predecessor because the previous node is not connected to this deleted node anymore.
- Input: ID of node to delete.
- Output: HostGraph in D3 Format.

Example in graph visual where we want to delete node 4. The edges between node 'two' and node 4 and node 3 and node 4 are deleted. This function also updates the node attribute NextDialogueID of node 'two' and node 3 to remove node 4 from the array list.



```

id: "two",
data: {label: "two"},
NextDialogID: ["4"],
position: {x: 100, y: -50},
type: 'input'

```

```

id: "3",
data: {label: "3"},
NextDialogID: ["4"],
position: {x: 150, y: 0},
type: 'input'

```

```

id: "two",
data: {label: "two"},
NextDialogID: [],
position: {x: 100, y: -50},
type: 'input'

```

```

id: "3",
data: {label: "3"},
NextDialogID: [],
position: {x: 150, y: 0},
type: 'input'

```

Updated *HostGraph* =

```

{
  links: [(('source': '0', 'target': '1', label: 'Has work experience'),
    ('source': '1', 'target': '3'),
    ('source': '0', 'target': 'two'))],
  nodes: [{id: '0'},
    {'id': '1'},
    {'id': 'two'},
    {'id': '3'}]
}

```

}

VERIFY

1. exists_node

- This function checks whether a specified node exists in the HostGraph. This function is called when a author of the interview script tries to relabel a node. The name exists, then the system shows and error as every node needs to have a unique ID.
- Input: ID of node to check
- Output: Boolean, true or false

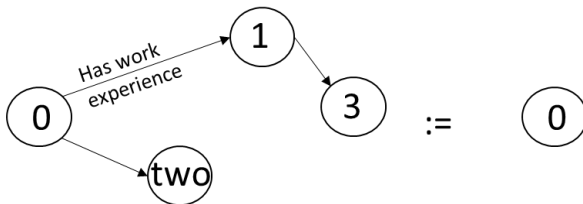
2. exists_edge

- This function checks whether an edge exists between two nodes to prevent a user from making a second edge between the same two nodes.
- Input: (source, target)
- Output: Boolean, true and false.

RESET

1. Reset

- This function empties the HostGraph so that the user can start from scratch. It calls the init() function to add a default start node as defined in the client.
- Input: None
- Output: HostGraph in D3 format



Updated *HostGraph* =

```
{
  links: [],
  nodes: [{id: '0'}]
}
```