Contributing to the User Journey Tool

Summary	In this codelab, you'll gain a high-level understanding of the UJT implementation by contributing a new feature.
URL	ujt-contribution
Category	Web
Environment	linux
Status	Draft
Feedback Link	chuanchen@google.com
Author	Chuan Chen
Author LDAP	chuanchen

Introduction

What is the UJT?

The User Journey Tool is an open-source web app used to view the dependency topology of a system from a user journey-based perspective. It displays a graph representing the dependencies of a system.

In this codelab, we'll gain a basic understanding of the implementation of the User Journey Tool, and walk through adding a simple feature to modify the topology graph rendered by the UJT.

What you'll learn

- The basics of the Dash framework
- How to set up and use the UJT
- The high level implementation of the UJT, including specific conventions and practices

What you'll need

- A linux machine to run command line scripts
- A web browser to access the application
- Basic knowledge of Python

Getting Set Up

Installing the UJT

Let's install the UJT.

1. Clone the UJT.

```
git clone git@github.com:googleinterns/userjourneytool.git
```

2. Checkout commit <change this later>.

```
git checkout <change this later>
```

3. Create and activate a virtual environment with venv. A virtual environment ensures we have consistent Python and dependency versions.

```
python3 -m venv /path/to/venv
source /path/to/venv/bin/activate
```

4. Navigate to the local git repo and install the required dependencies.

```
cd userjourneytool
pip install -r requirements.txt
```

5. The UJT uses protocol buffers to store its internal data structures. We need to compile the message specifications (in the protos directory) into Python generated code.

```
./generate_protos
```

6. Now that we've generated the Python code, we need to install the generated code as a package within our environment. This enables each module to import the protobufs.

```
pip install --editable generated
```

Running the UJT

Now that we have the tool installed, let's try to run it and take a look at how it works.
For the purposes of this codelab, our data source will be an example server implementation which serves mock data and is provided in the git repo.

1. Generate the mock data.

```
python3 -m ujt.server.generate_data
```

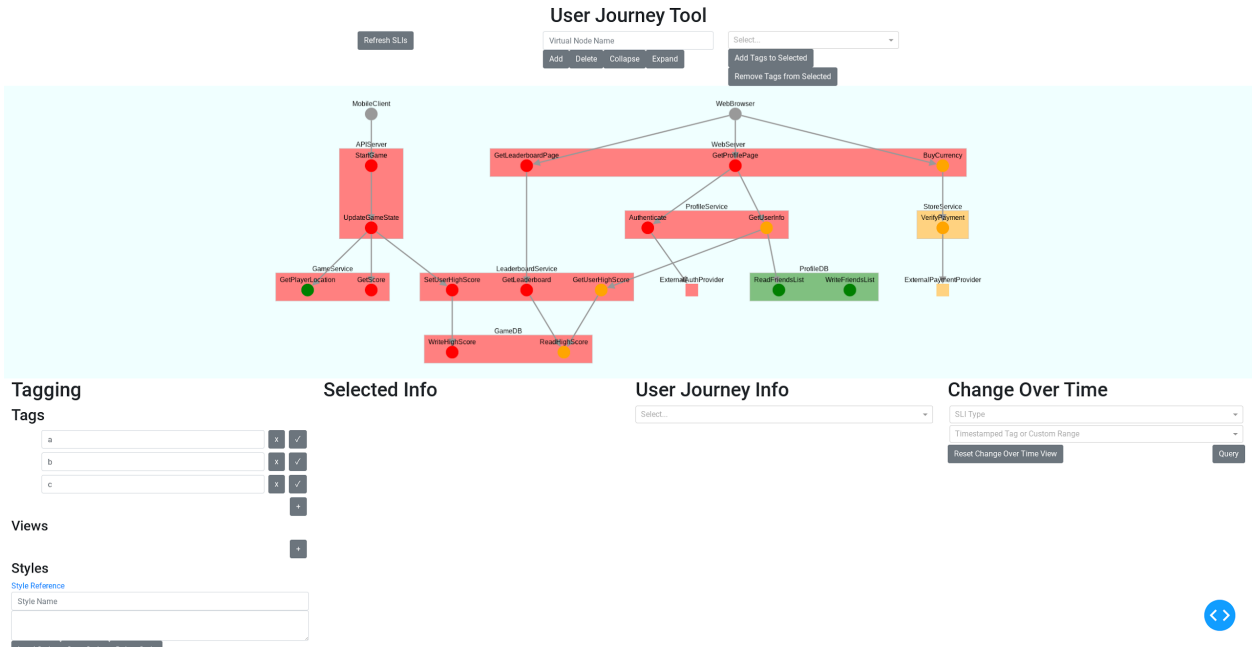
2. Start the example server.

```
python3 -m ujt.server.server
```

3. Start the UJT Dash server.

```
python3 -m ujt.ujt
```

4. Navigate to http://127.0.0.1:8050/. It should look like this, but with different colors:



Using the UJT

Data Structures

The UJT provides an overview of system topology and status from a user journey based granularity. Let's introduce the major data structures that compose each visual element in the graph.

Clients represent sources of user interactions.

- Clients each contain a set of *User Journeys*.
- Clients are visualized as a graph node.

User Journeys represent a set of interactions made by the user of the system.

- User Journeys are composed of a set of toplevel *Dependencies*.
- User Journeys are visualized as a set of paths, originating from a Client, through the graph

Dependencies represent connections between parts of a system.

- This could be a RPC call, reading/writing from a queue, etc.
- Dependencies are visualized as graph edges between *Nodes* or Clients.

Nodes could represent an endpoint, a service, an entire system, or anything that we want the UJT to monitor.

- Nodes can contain children (other Nodes), and *SLIs*.
- Nodes are visualized as a node in the graph.

SLIs (service level indicators) represent a timestamped measurement of a metric, such as throughput or latency.

- SLIs contain SLO (service level objective) bounds.

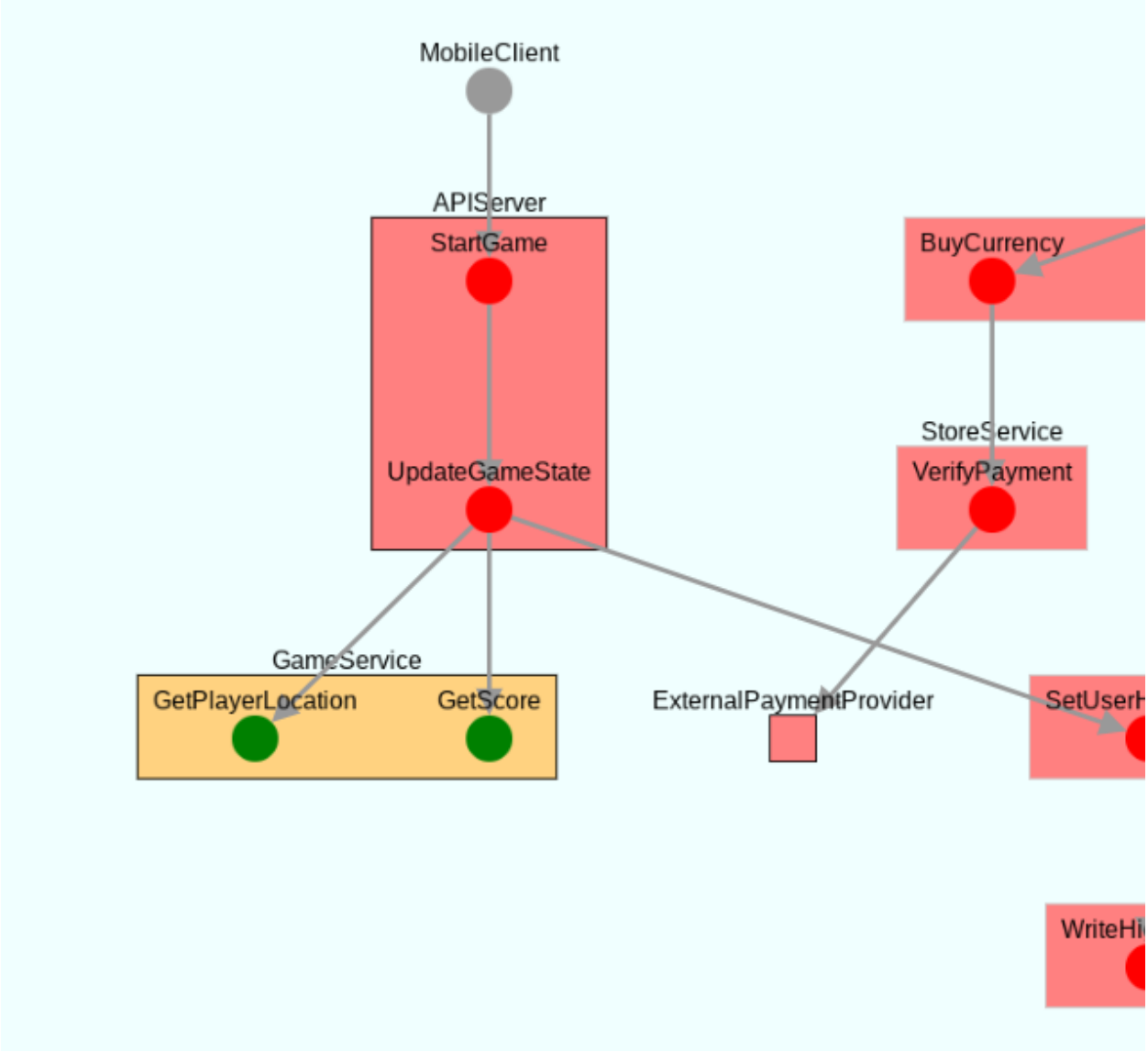
Nodes and SLIs have *Statuses*.

- SLI status is computed based on the value of the SLI relative to its SLO bounds.
- Node status is computed based on the statuses of its children and the node's own SLI statuses.
- Statuses are generally visualized through color coding in the graph

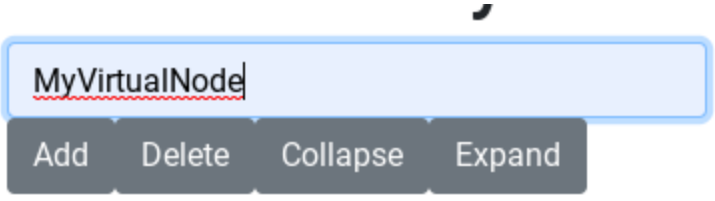
Virtual Nodes

Virtual nodes are groupings of nodes. They can help simplify the topology and identify areas of interest.

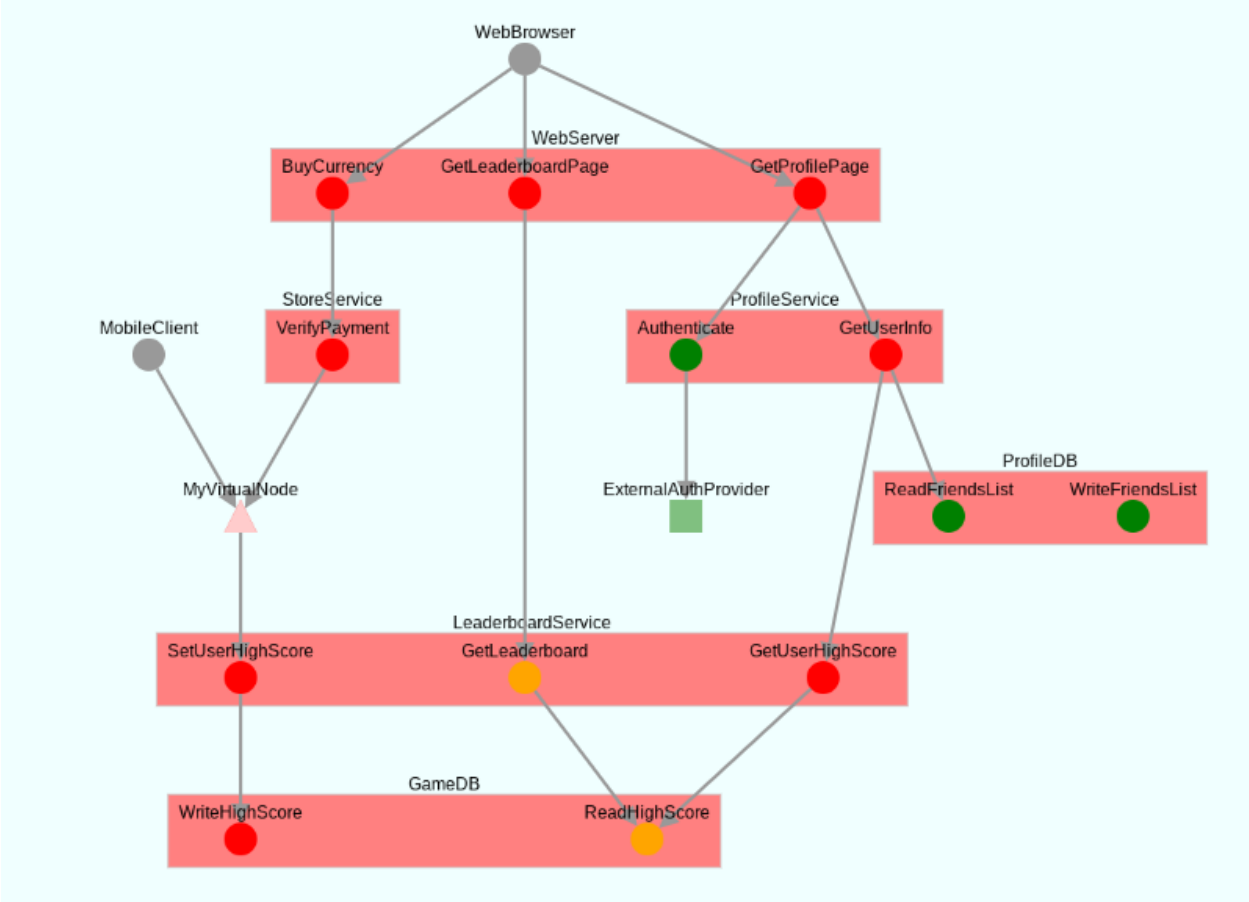
Let's try collapsing a few nodes together into a virtual node. First, hold shift and click multiple services.



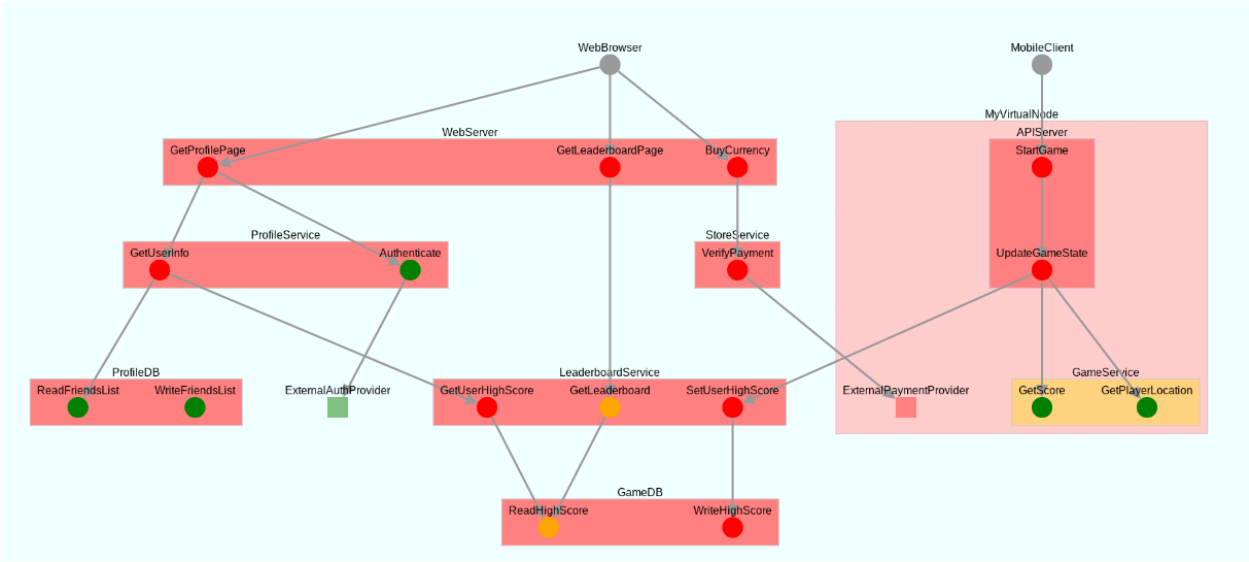
Next, type your desired virtual node name into the input box towards the top of the application, and click “Add”.



Now, notice that the selected nodes are combined into a single virtual node. Virtual nodes have a lighter background color, compared to real nodes. Collapsed virtual nodes are displayed as octagons.



Now, ensure the input box has the same name as the created virtual node, and expand the virtual node by clicking the “Expand” button.



Notice that sometimes the relative location of the elements may change after virtual nodes are added or removed.

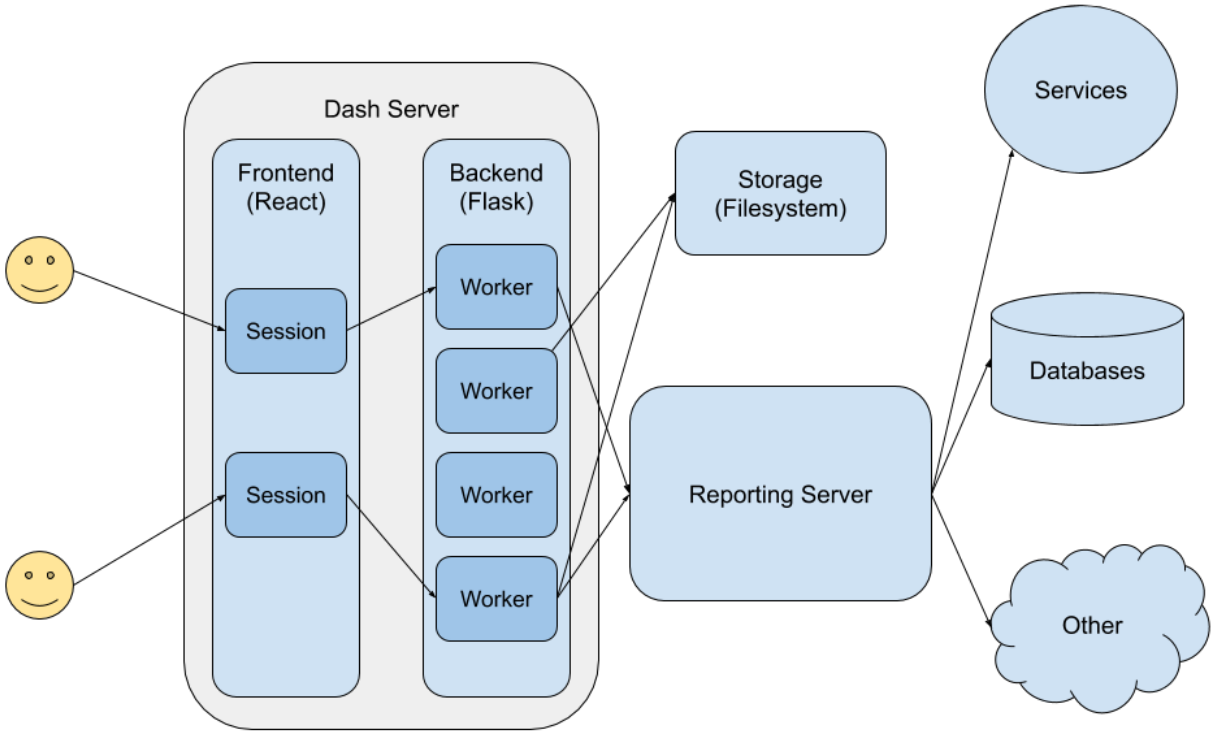
In this codelab, we'll create a feature to collapse or expand all virtual nodes with a single button.

Under the Hood

Overview

The UJT is built on Dash, a Python framework for building reactive data visualization web apps. Dash wraps Flask server processes that serve React components in the frontend. We use Dash Cytoscape, a wrapper for the widely used Cytoscape.js library, for graph visualization.

Memory isn't shared between backend workers, so we use a storage system to communicate between workers. The UJT also communicates with a reporting server, which serves as a datasource and provides both the network topology and aggregated SLI metrics from sources that we want to monitor.



We'll explore the implementation and integration of these individual components in the following sections.

Frontend

Declaring Components

Each element we see on the UJT is a Dash *component*. Examples of components include the graph, buttons, or input boxes. We use a few component libraries to provide us with the building blocks for the application, namely:

- Dash HTML Components: A 1:1 mapping of components and HTML tags
- Dash Cytoscape: Library providing graph component
- Dash Core Components: Useful interactive components including dropdown menus and datatables
- Dash Bootstrap Components: Wrapper for bootstrap classes, mostly used for layout purposes

Let's take a look at how components are declared.

components.py

```
def get_layout():
    """Generate the top-level layout for the Dash app.

    Returns:
        a Dash HTML Div component containing the top-level layout for
the app.
    """

    return html.Div(
        children=[
            html.H1(
                children="User Journey Tool",
                style={
                    "textAlign": "center",
                },
            ),
            get_top_row_components(),
            get_cytoscape_graph(),
            get_bottom_panel_components(),
            dbc.Modal(
                children=[
                    dbc.ModalHeader("Error"),
```

```
                dbc.ModalBody(id=id_constants.COLLAPSE_ERROR_MODAL_BODY),
                    dbc.ModalFooter(
                        dbc.Button(
                            "Close",
                            id=id_constants.COLLAPSE_ERROR_MODAL_CLOSE,
                            className="ml-auto",
                        ),
                    ),
                ],
                id=id_constants.COLLAPSE_ERROR_MODAL,
            ),
            get_signals(),
            get_stores(),
        ],
    )
```

The get_layout returns the top level layout of the application. In the snippet above, we see a few components, such as a Div, a H1, a Modal. We also see a few calls to functions that return components as well.

Notice that components generally accept a few main arguments:

- children: Accepts a string, a number, a single component, or a list of components. This argument allows Dash to nest components. For example, refer to the outermost Div component declared on line 32.
- id: Accepts a string. This argument is used to uniquely identify components for later use in callbacks, which power the interactivity in the application. For convenience, we declare all id strings in the id_constants.py module.

Other arguments can be specific to the individual component, or the component library itself. For instance, all Dash Bootstrap components, such as the dbc.Button, support the className argument, which allows us to provide css classes (usually from Bootstrap) to style the component.

Serving Components

Now that we understand the high-level structure of components, let's investigate how they're served to the browser.

dash_app.py

```
cyto.load_extra_layouts()
app = dash.Dash(__name__, external_stylesheets=[dbc.themes.BOOTSTRAP])
```

```
cache = Cache()
cache.init_app(
    app.server,
    config={
        "CACHE_TYPE": "filesystem",
        "CACHE_DIR": "cache_dir",
        "CACHE_DEFAULT_TIMEOUT": 0,
        "CACHE_THRESHOLD": 0,
    },
)
```

In this snippet, we create a Dash object and assign it to the app variable. Now, other modules can access the Dash application through this variable by importing this module.

In the subsequent lines, we create a filesystem Cache for the worker, which serves as a means to communicate between worker processes. We'll discuss the cache and communication methods in a later section.

Now, open up `ujt.py`.

ujt.py

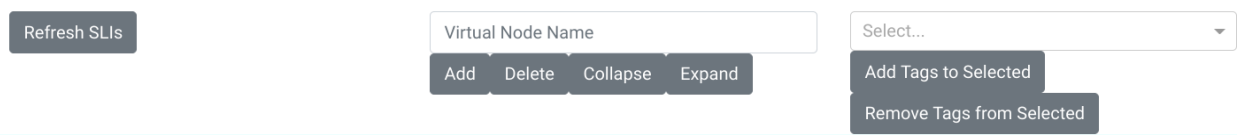
```
if __name__ == "__main__":
    initialize_ujt()
    app.layout = components.get_layout()
    app.run_server(debug=True)
```

When we run the ujt, the components we declared earlier are loaded into the layout property of the Dash app. Calling app.run_server starts the Dash application and serves the frontend to the browser.

The full source of `ujt.py` contains some additional code run on startup. We'll provide more detail on those later.

Your turn

Since we want to create a button that toggles the collapsed state of all virtual nodes, let's return to `components.py`, and take a look at `get_top_row_components()`. As the name implies, this function returns the components in the top row.



components.py

```
def get_top_row_components():
    return html.Div(
        children=[
            dbc.Container(
                dbc.Row(
                    children=[
                        dbc.Col(
                            dbc.Button(
                                id=id_constants.REFRESH_SLI_BUTTON,
                                children="Refresh SLIs",
                            ),
                        ),

dbc.Col(children=get_virtual_node_control_components()),

dbc.Col(children=get_batch_apply_tag_components()),
                    ],
                ),
            ),
        ],
    )
```

This function returns a Div, which contains three columns. We want to add a button related to virtual nodes, so let's investigate `get_virtual_node_control_components()`.

components.py

```
def get_virtual_node_control_components():
    components = [
        dbc.Input(
            id=id_constants.VIRTUAL_NODE_INPUT,
            type="text",
            placeholder="Virtual Node Name",
        ),
        dbc.Button(
            id=id_constants.ADD_VIRTUAL_NODE_BUTTON,
            children="Add",
```

```
        ),
        dbc.Button(
            id=id_constants.DELETE_VIRTUAL_NODE_BUTTON,
            children="Delete",
        ),
        dbc.Button(
            id=id_constants.COLLAPSE_VIRTUAL_NODE_BUTTON,
            children="Collapse",
        ),
        dbc.Button(
            id=id_constants.EXPAND_VIRTUAL_NODE_BUTTON,
            children="Expand",
        ),
    ]
    return components
```

It looks like we found where to place our new button. Let's declare a new button and append it to the end of the components list.

```
dbc.Button(
    id=id_constants.TOGGLE_ALL_VIRTUAL_NODE_BUTTON,
    children="Toggle All Collapsed State"
),
```

Let's declare a new id for the button in `id_constants.py`. Place the new variable under the other virtual node related buttons id variables.

id_constants.py

```
TOGGLE_ALL_VIRTUAL_NODE_BUTTON = "toggle-all-virtual-node-button"
```

Save your changes. If there aren't any syntax errors, the Dash server will hot-reload your code and refresh the browser page automatically. You should see something like this in the top bar:

Backend

Callbacks

Now, we'll explore how to add interactivity features to the declared components. Dash uses a callback-based model to update properties of output components. Dash callbacks declare Output, Input, and State component/property pairs. Callbacks are fired whenever any of the registered Input component properties are modified, and the return value of callbacks are placed into the registered properties of the Output components. State component/property pairs don't trigger the callback, but are still provided as inputs to the callback function itself.

Let's take a look at a simple callback function.

callbacks/virtual_node_callbacks.py

```
@app.callback(
    Output(id_constants.SIGNAL_VIRTUAL_NODE_DELETE, "children"),
    Input(id_constants.DELETE_VIRTUAL_NODE_BUTTON,
"n_clicks_timestamp"),
    State(id_constants.VIRTUAL_NODE_INPUT, "value"),
    prevent_initial_call=True,
)
def delete_virtual_node(
    delete_n_clicks_timestamp,
    virtual_node_name,
):
    """Handles deleting virtual nodes.

    This function is called:
        when the delete button is clicked

    Args:
        delete_n_clicks_timestamp: Timestamp of when the delete button
was clicked. Value unused, input only provided to register callback.
        virtual_node_name: The name of the virtual node to delete.

    Returns:
```

```
        A string to be placed in the children property of the
SIGNAL_VIRTUAL_NODE_DELETE hidden div.
        This hidden div is used to ensure the callbacks to update the
error modal visibility and cytoscape graph
        are called in the correct order.
    """

    virtual_node_map = state.get_virtual_node_map()
    if virtual_node_name not in virtual_node_map:
        return "Error: The entered name doesn't match any existing
virtual nodes."

    state.delete_virtual_node(virtual_node_name)

    return constants.OK_SIGNAL
```

The callback is registered at *import time* through the `app.callback` decorator.
In this example, the callback returns the `children` property of the `SIGNAL_VIRTUAL_NODE_DELETE` component. The callback is fired when the `n_clicks_timestamp` of property of the `DELETE_VIRTUAL_NODE_BUTTON` is changed, but *not* when the value of the `VIRTUAL_NODE_INPUT` is changed.

Signals

Recall that the output of the delete_virtual_node function was a *signal* component. Signals are invisible components that allow us to modularize callbacks, and ensure a consistent callback order.

With Dash, only a *single* callback can update a given output property of a specific component. For instance, if we have two buttons that modify the text of a div, we can't have a separate callback for each button, and have both callbacks output the desired text.

Not allowed

```
@app.callback(
    Output("div", "children"),
    Input("button1", "n_clicks_timestamp"),
)
def button1_callback(n_clicks_timestamp):
    return "updated by button1"
```

```
@app.callback(
    Output("div", "children"),
    Input("button2", "n_clicks_timestamp"),
)
def button2_callback(n_clicks_timestamp):
    return "updated by button2"
```

There are two solutions to this issue.

1. Register both buttons as inputs to the same callback. Dash provides tools to determine the context under which the callback was fired (e.g. which component/property triggered the callback). This approach is simple and has lower overhead, but the complexity of the callback function quickly grows as the number of inputs grows. It also doesn't scale well when unrelated inputs need to update the same output. Moreover, if each Input requires reading its own State, the number of arguments to the function becomes large as well.

```
@app.callback(
    Output("div", "children"),
    [
        Input("button1", "n_clicks_timestamp"),
        Input("button2", "n_clicks_timestamp"),
    ]
)
def div_callback(button1_n_clicks_timestamp,
button2_n_clicks_timestamp):
    if dash.callback_context.triggered[0]["prop_id"] ==
"button1.n_clicks_timestamp":
        return "updated by button1"
    else:
        return "updated by button2"
```

2. We'd like to avoid having to place everything in a single callback for each output. However, to factor out the functionality of each input component, we need some way to communicate between the callbacks. To address this issue, we use signals and a centralized state store.

```
@app.callback(
    Output("signal1", "children"),
    Input("button1", "n_clicks_timestamp"),
)
```

```
def button1_callback(n_clicks_timestamp):
    state.set_message("updated by button1")
    return "OK"

@app.callback(
    Output("signal2", "children"),
    Input("button2", "n_clicks_timestamp"),
)
def button2_callback(n_clicks_timestamp):
    state.set_message("updated by button2")
    return "OK"

@app.callback(
    Output("composite_signal", "children"),
    [
        Input("signal1", "children"),
        Input("signal2", "children"),
    ],
)
def generate_composite_signal(signal1, signal2):
    return "OK"

@app.callback(
    Output("div", "children"),
    Input("composite_signal", "children"),
)
def div_callback(composite_signal):
    return state.get_message()
```

In this approach, each button generates a signal. Each individual signal triggers the `generate_composite_signal` callback, which generates a composite signal. Then, the div updating callback can subscribe to the single composite signal, and avoids having to implement the functionality associated with each individual input. In this example, we could have `div_callback` subscribe to both individual signals as well.

Despite requiring more code for this small example, the signal method scales much better to a large number of various inputs by enabling us to modularize the code. However, it comes at a slight performance cost, as Dash needs to transmit the value of the signals over the network in

order to place them into the DOM. It makes sense to use this approach when we want to group semantically related inputs together (e.g. the virtual node related buttons).

We often use a combination of these approaches in the UJT, depending on the situation.

Updating the Graph

Now, let's investigate the central callback of the UJT, `update_graph_elements`. This callback handles the majority of visual changes in the cytoscape graph.

graph_callbacks.py

```
@app.callback(
    Output(id_constants.CYTOSCAPE_GRAPH, "elements"),
    [
        Input(id_constants.REFRESH_SLI_BUTTON, "n_clicks_timestamp"),
        Input({id_constants.USER_JOURNEY_DATATABLE: ALL},
"selected_row_ids"),
        Input(id_constants.SIGNAL_VIRTUAL_NODE_UPDATE, "children"),
        Input({id_constants.OVERRIDE_DROPDOWN: ALL}, "value"),
        Input(id_constants.SIGNAL_COMPOSITE_TAGGING_UPDATE, "children"),
        Input(id_constants.CHANGE_OVER_TIME_SLI_STORE, "data"),
    ],
    [
        State(id_constants.CYTOSCAPE_GRAPH, "elements"),
        State(id_constants.CYTOSCAPE_GRAPH, "selectedNodeData"),
        State(id_constants.CYTOSCAPE_GRAPH, "tapNode"),
        State(id_constants.VIEW_STORE, "data"),
    ],
)
def update_graph_elements(...):
    ...
    elements = state.get_cytoscape_elements()
    ...
    elements = transformers.apply_virtual_nodes_to_elements(elements)
    ...
```

As shown in the snippet above, this callback registers some components (approach 1, e.g. `REFRESH_SLI_BUTTON`, `OVERRIDE_DROPDOWN`), and some signals (approach 2, e.g. `SIGNAL_VIRTUAL_NODE_UPDATE`, `SIGNAL_COMPOSITE_TAGGING_UPDATE`).

Generally, we register the components directly if the functionality associated with that component doesn't need to read any associated state or do any preprocessing. For instance, the `REFRESH_SLI_BUTTON` doesn't need to refer to any other components or change any internal state. In contrast, the virtual node buttons that produce the `SIGNAL_VIRTUAL_NODE_UPDATE` each modify internal state and read from the virtual node name text box.

This callback combines the underlying data structures describing the system status and topology, with the user interactions from the frontend, and produces `elements`, a list of dictionaries, in a format recognizable by the Cytoscape library for rendering.

Notice that this callback is called every time the SIGNAL_VIRTUAL_NODE_UPDATE is modified. `elements` is originally constructed by converting the underlying topology of Nodes, Clients, and Dependencies into a list of dictionaries, with each dictionary describing properties about the graph element, such as it's label, id, source, or target.

Next, based on the current state of the UJT, the callback function applies various transformations to the list of elements. In this line shown above, the callback applies a transformation to add or remove the relevant elements to produce the virtual nodes.

Your Turn

Based on what we just learned about callbacks and signals, let's write the function header for our toggle collapse all callback.

callbacks/virtual_node_callbacks.py

```
@app.callback(
    Output(id_constants.SIGNAL_VIRTUAL_NODE_TOGGLE_ALL, "children"),
    Input(id_constants.TOGGLE_ALL_VIRTUAL_NODE_BUTTON, "n_clicks"),
    prevent_initial_call=True,
)
def toggle_all_collapse_virtual_node(toggle_all_n_clicks_timestamp: int)
-> str:
    """Collapses and expands all virtual nodes.

    This function is called:
        when the toggle all virtual node button is clicked

    Args:
        toggle_all_n_clicks_timestamp: Number of times the toggle all
virtual node button was clicked.
```

```
    Returns:
        OK_SIGNAL.
    """
    pass
```

Notice that we included the line `prevent_initial_call=True`. Dash will automatically fire the callback at load time, if we don't supply an initial value for the property when we declare the output component. We can avoid this behavior by providing this argument.

Let's also declare a new id for the signal we want to produce, and add it to the `SIGNALS` list.

id_constants.py

```
SIGNAL_VIRTUAL_NODE_TOGGLE_ALL = "toggle-all-virtual-node-signal"
...
SIGNALS = [
    SIGNAL_VIRTUAL_NODE_ADD,
    SIGNAL_VIRTUAL_NODE_DELETE,
    SIGNAL_VIRTUAL_NODE_COLLAPSE,
    SIGNAL_VIRTUAL_NODE_EXPAND,
    SIGNAL_VIRTUAL_NODE_TOGGLE_ALL,
    SIGNAL_VIRTUAL_NODE_UPDATE,
    ...
]
```

Now, components.py will automatically create an invisible div component for our new signal.

components.py

```
def get_signals():
    signals = [
        html.Div(
            id=signal_id,
            style={"display": "none"},
        )
        for signal_id in id_constants.SIGNALS
    ]

    return html.Div(id=id_constants.SIGNAL_WRAPPER_DIV,
children=signals)
```

Now, navigate to callbacks/signal_callbacks.py. In order to produce a general signal composed from the signals of all virtual node related operations, we need to add our new signal to the `COMPOSITE_SIGNAL_MAP`.

callbacks/signal_callbacks.py

```
COMPOSITE_SIGNAL_MAP = {
    id_constants.SIGNAL_VIRTUAL_NODE_UPDATE: (
        id_constants.SIGNAL_VIRTUAL_NODE_ADD,
        id_constants.SIGNAL_VIRTUAL_NODE_DELETE,
        id_constants.SIGNAL_VIRTUAL_NODE_EXPAND,
        id_constants.SIGNAL_VIRTUAL_NODE_COLLAPSE,
        id_constants.SIGNAL_VIRTUAL_NODE_TOGGLE_ALL,
    ),
    ...
}

...

def generate_composite_signals():
    for output_signal_id, input_signal_ids in
COMPOSITE_SIGNAL_MAP.items():
        dash_output = Output(output_signal_id, "children")
        dash_inputs = [
            Input(input_signal_id, "children") for input_signal_id in
input_signal_ids
        ]
        app.callback(dash_output, dash_inputs,
prevent_initial_call=True)(
            generate_generic_update_signal
        )
```

Now, when `generate_composite_signals()` is called at startup, it will dynamically register a callback that takes the individual virtual node signals as input, and produces `SIGNAL_VIRTUAL_NODE_UPDATE` as output.

Notice that because `SIGNAL_VIRTUAL_NODE_UPDATE` was already registered as an input to `update_graph_elements`, no changes are needed to that callback.

State Handling

There are three places where data is stored in the UJT.

Components

- Some data is stored in component properties.
- This could be the value of an input box or dropdown menu.
- This data persists per-session, meaning that each window with the Dash app open has its own component state, isolated from the others windows.
- This data is cleared when a session is ended.
- Callbacks read from this state to produce outputs.
- Signals are implemented with component data.

Dash Memory

- Some data is stored in the memory of Dash worker processes.
- This memory isn't shared between workers.
- In general, we can only rely on the data to persist within a single callback, not between callbacks.

Flask-Caching

- The UJT uses the filesystem cache as the Flask-Caching backend.
- This cache is used to persist state on the Dash server side, and stores topology data.
- Each worker can read and write from the cache.
- However, **the current design of the UJT does not support multiple processes**, since reads and writes are not synchronized among callbacks in different processes.

The `state.py` module provides a centralized interface for making cached state changes in the UJT.

Your Turn

`state.py` contains a few functions of interest for our feature.

state.py

```
def get_virtual_node_map() -> Dict[str, VirtualNode]:
    """Gets a dictionary mapping virtual node names to virtual node
messages.

    Returns:
        A dictionary mapping virtual node names to virtual node objects.
    """
    return cache.get(id_constants.VIRTUAL_NODE_MAP)
```

```
...

def set_virtual_node_collapsed_state(virtual_node_name: str, collapsed:
bool):
    """Sets the collapsed state of a virtual node.

    Updates the corresponding virtual node object within the virtual
node map.

    Args:
        virutal_node_name: The name of the virtual node to update.
        collapsed: The new collapsed state
    """
    virtual_node_map = get_virtual_node_map()
    virtual_node = virtual_node_map[virtual_node_name]
    virtual_node.collapsed = collapsed
    set_virtual_node_map(virtual_node_map)
```

Conveniently, we already have an API for setting the collapsed state of a given virtual node. We also have a map, which keeps the names of virtual nodes as keys. With this, we have everything we need to implement our callback now.

callbacks/virtual_node_callbacks.py

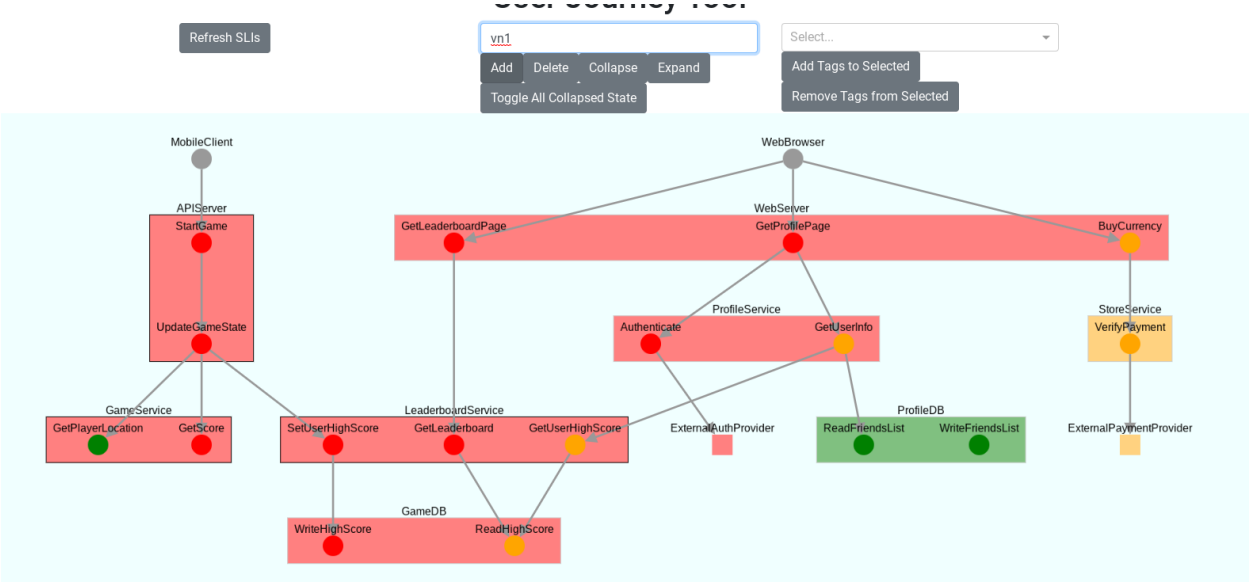
```
@app.callback(
    Output(id_constants.SIGNAL_VIRTUAL_NODE_TOGGLE_ALL, "children"),
    Input(id_constants.TOGGLE_ALL_VIRTUAL_NODE_BUTTON, "n_clicks"),
)
def toggle_all_collapse_virtual_node(toggle_all_n_clicks: int) -> str:
    """Collapses and expands all virtual nodes.

    This function is called:
        when the toggle all virtual node button is clicked

    Args:
        toggle_all_n_clicks: Number of times the toggle all virtual node
button was clicked.

    Returns:
```

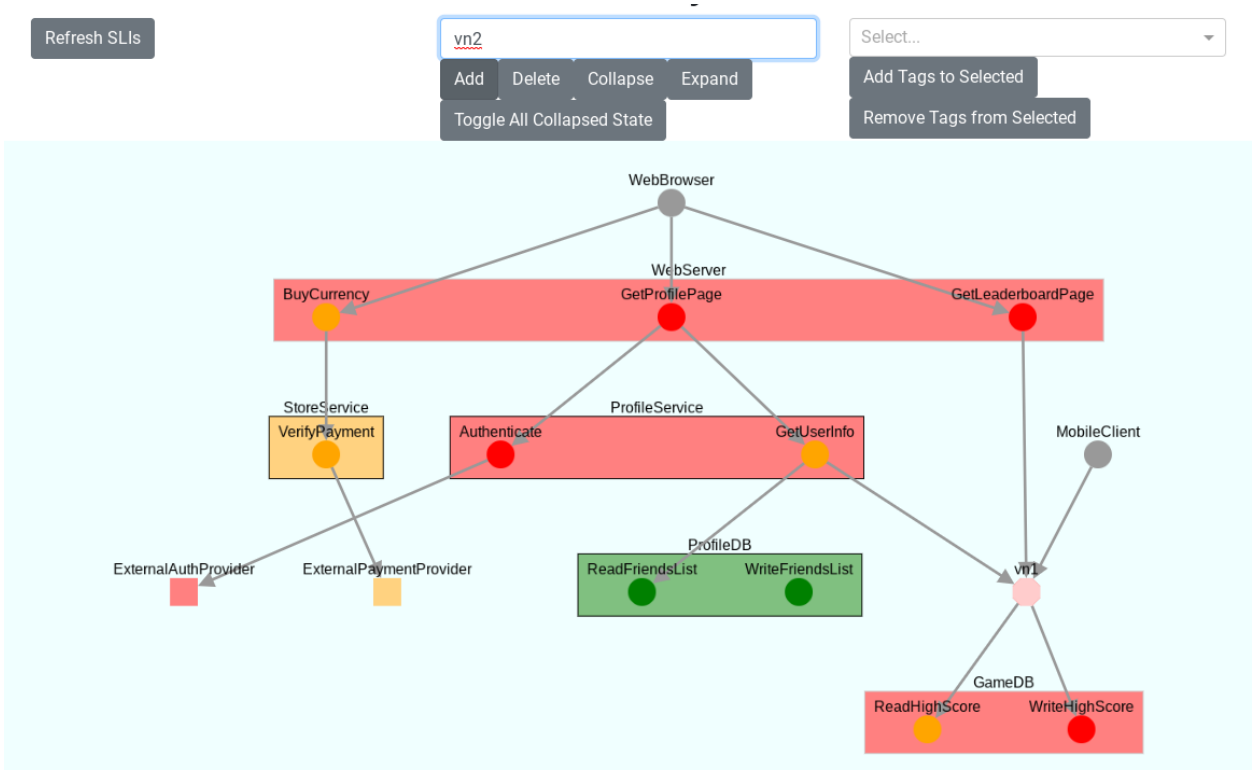
```
        OK_SIGNAL.
    """
    collapsed = toggle_all_n_clicks % 2 == 0
    virtual_node_map = state.get_virtual_node_map()
    for virtual_node_name in virtual_node_map:
        state.set_virtual_node_collapsed_state(virtual_node_name,
collapsed=collapsed)

    return constants.OK_SIGNAL
```

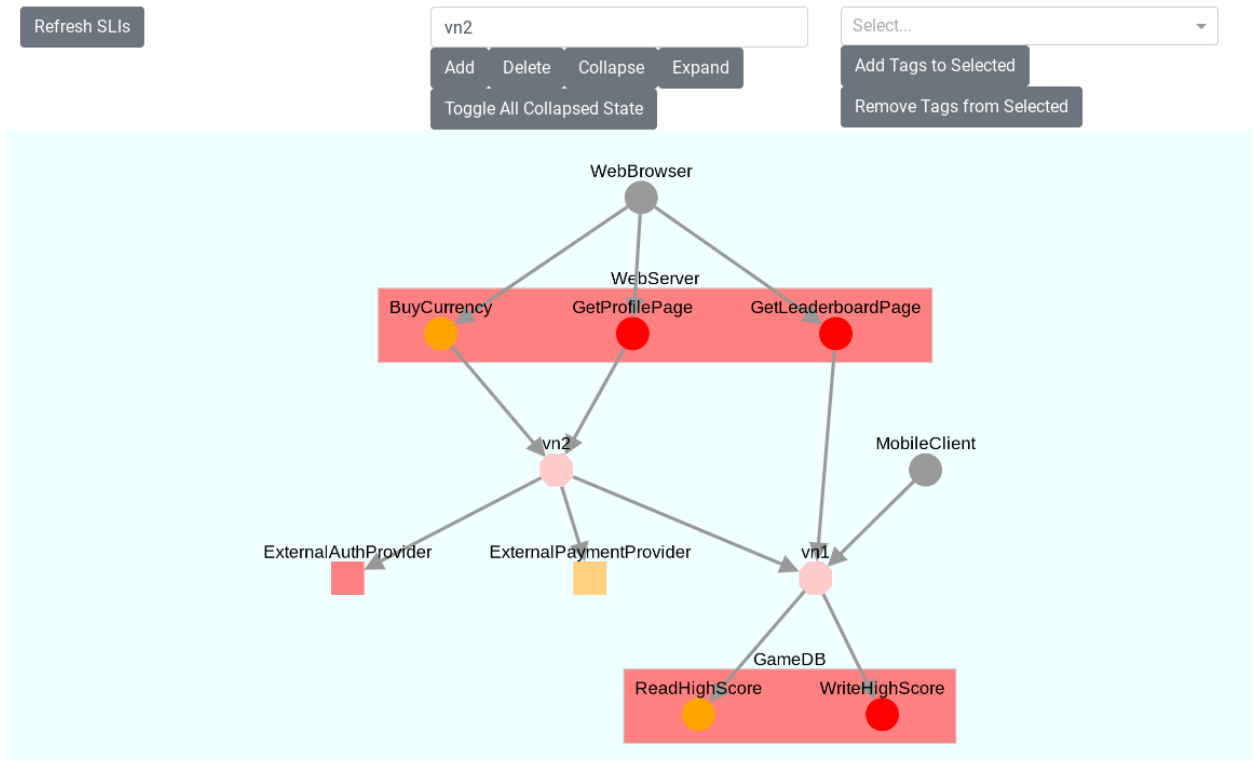
We iterate through the map of virtual nodes, and set the collapsed state of each one. The collapsed state is determined by the parity of the number of clicks to the toggle all button.

Try Our Feature

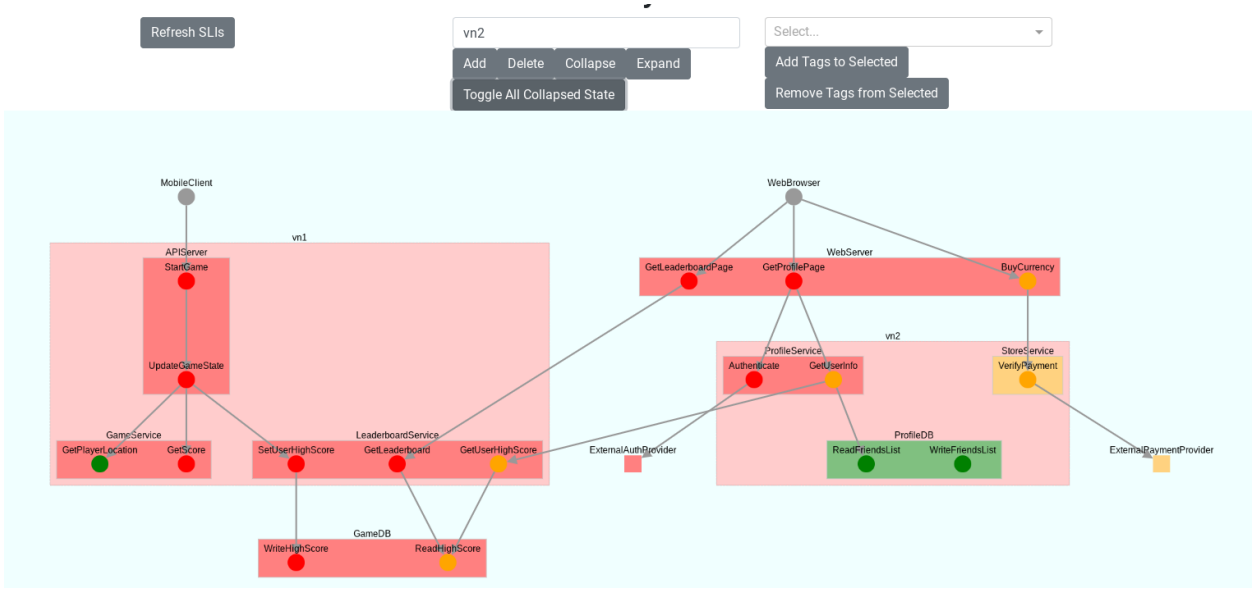
Let's create two virtual nodes to try our new feature.



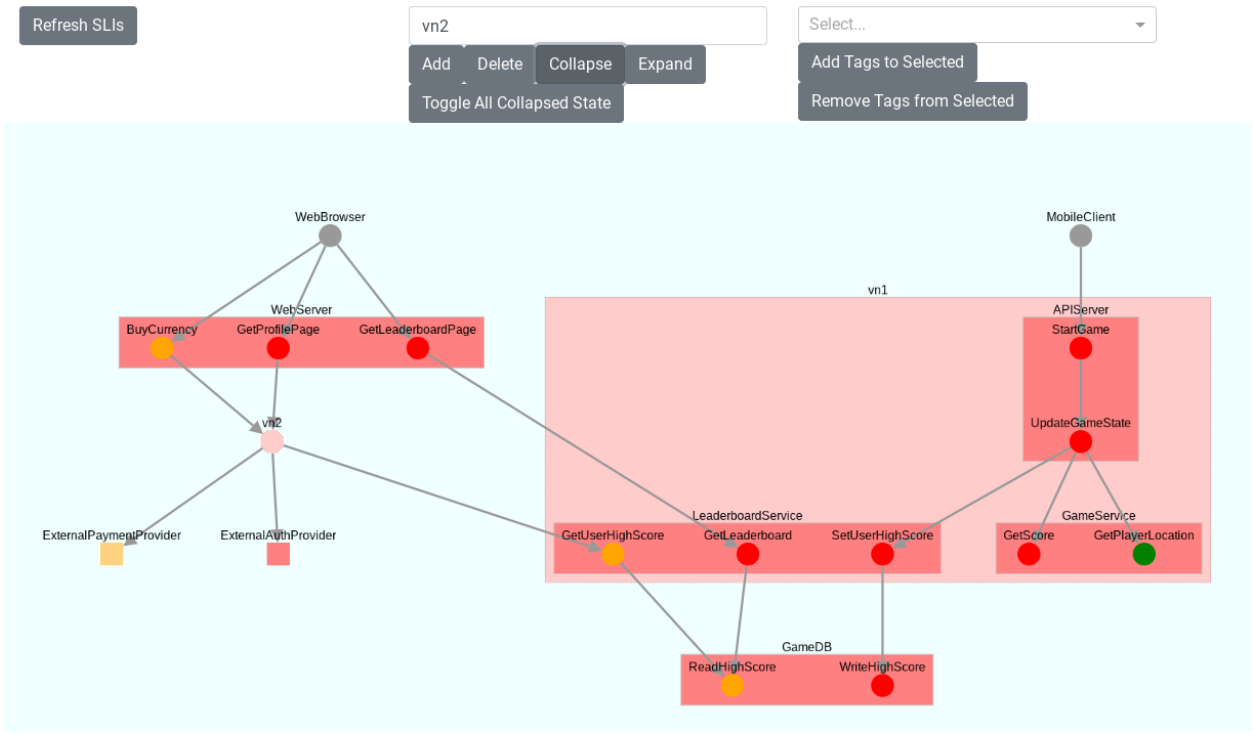




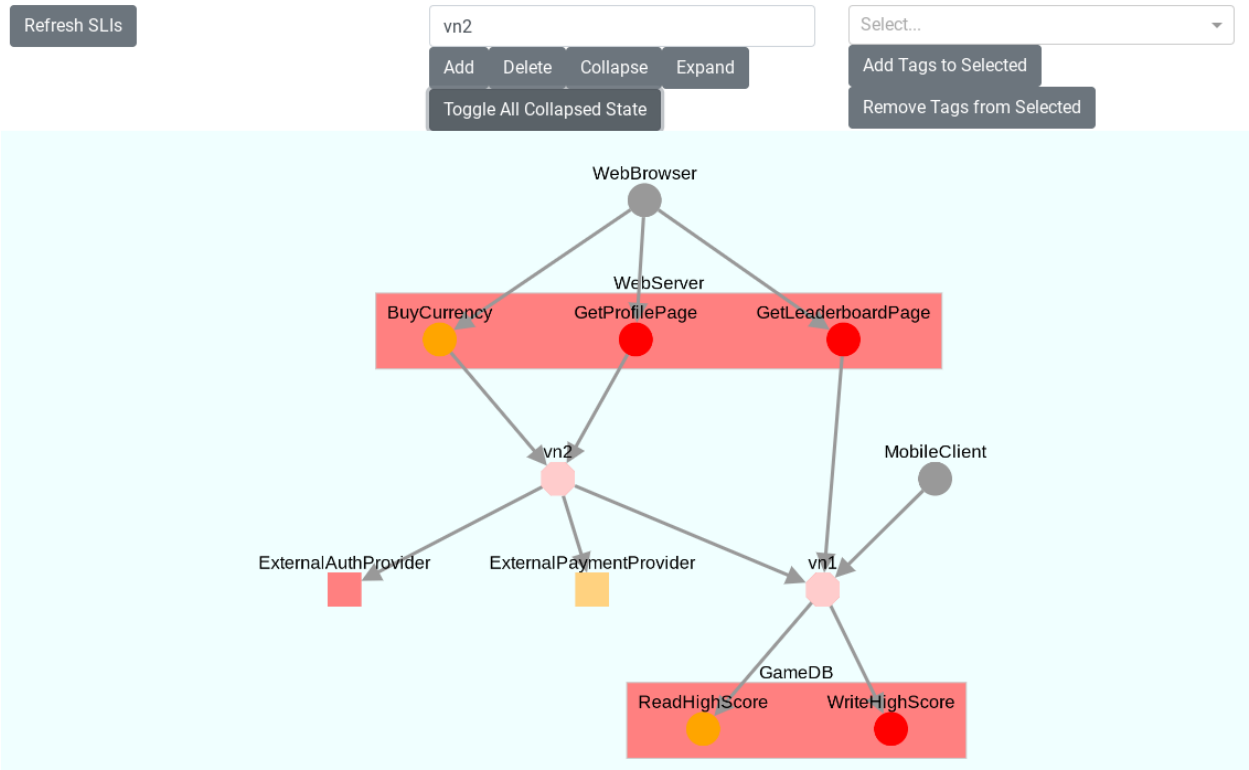
Now, click our new button.



Both nodes expanded! Let’s collapse one of the nodes individually.



Now let’s try to click the toggle all button again.



Great! It looks like our feature is working.