# User Journey Implementation Details

This document details some of the interesting/non-obvious aspects of the implementation of the UJT. Additionally, it describes some of the difficulties encountered during the development process.

## Data Polling and Storage

Dash heavily discourages the use of modifying in-memory state on the server side (the documentation explicitly mentions global variables, but the rationale applies to any server side state changes[1]). This is because:

a) User sessions are supposed to be isolated. Modifying server state could result in one user seeing incorrect results from another user.
b) Dash could be run on multiple processes. Thus, server state changes made to a single process will not be propagated across different processes.

For example, if a Dash app is supposed to display a bar graph displaying a statistic by year, the entire dataset would be stored statically on the server side. The client would modify and store filtering parameters (e.g. a year range).

For our application, network topology data (Services, Endpoints, Clients, their Dependencies) can be loaded and saved by the Dash server at start-time as these data remain static.

However, due to the restrictions described above (mainly (b)), we cannot simply have a Dash callback poll the reporting server and store SLI/status data in the Dash server. This will cause the SLI data to be updated on a single instance of the server, and not all users will see correct results.

To poll SLI data, I see three implementations:

1) Clients (browsers) independently ask the Dash server to poll SLI data from the reporting server at a specified interval. The SLI data may have to be saved in the browser (example 1), as an intermediate step. This data isn't persisted on the Dash server. The Dash server combines the SLI data with the static graph data, and returns a Cytoscape graph to the frontend.
    a) Drawback: The data isn't cached by the server across users, so load on reporting server scales (linearly?) with the amount of users
        i) Polling interval is 5s. Client 1 joins at t = 0, while Client 2 joins at t = 3. Then, requests are made to the server at t = 0, 3, 5, 8, 10, 13, ...
2) Clients independently ask the Dash server to poll SLI data. Dash server processes use Flask-Cache (with redis or filesystem cache) to save SLI data and the latest fetch timestamp on the server side (example 3). This makes the SLI data and fetch timestamp

[1] For example, we couldn't wrap the dash app in a class and modify fields to store state.

accessible to all processes. For all following requests by clients to *any* Dash server process, the Dash server will either serve the cached SLI data, or fetch new SLI data if the cached fetch timestamp is too old.
    a) Drawback: Adds new dependency. Refresh time will vary per request and per client, leading to inconsistent UX.
3) Use a separate process to fetch SLI data and compute statuses on the Dash server side. This separate process writes to a shared location (e.g. local filesystem, a database), and each Dash server process reads from said location. (see here)
    a) Drawback: may require hacks to force the refreshed data to appear on client side

Approach 2 is implemented in this PR.

We may have to restructure our proto definitions and our mental model accordingly. Instead of Services and Endpoints *containing* SLIs, we should think of SLIs as separate objects that have to be combined with the topology to display the whole system status to the user.

## Data Distribution across Reporting Server vs Dash Server

Flask-caching's filesystem cache allows the Dash server to persist information across different server processes and over server restarts.

The reporting server communicates with the UJT and relays information regarding the dependency topology and SLIs. This includes Nodes, Dependencies, and Clients.

UJT specific data that's unrelated to the underlying topology is stored only on the Dash server. These include virtual nodes, tags, and views.

How should we handle comments and status override? These are semi-related to the actual data structures but are only used by Dash server.

# Data Structure Representation

## Services, Endpoints, Systems vs. Nodes

Services were previously allowed to contain Endpoints by maintaining a list of references to its child Endpoints. Endpoints were forced to be contained within a parent Service, and held a reference to its parent. Aside from these differences, both Endpoints and Services held identical properties (SLIs, Dependencies, Statuses, etc) and supported an identical set of operations. This indicated that we could consider these two to be special cases of the same type. As a result, we consolidated these Services and Endpoints into Nodes, with a NodeType field indicating the specific type of the Node. This change enables us to more directly support higher-level abstractions of services into systems (and can be further extended to arbitrary levels of grouping.)

NodeType is not computed based on the number of children or parents of the node, as we want to support cases such as a black-box service with no endpoints, or an ungrouped service that's not contained within a service.

## Reference vs Composition

In many protos in our application, we have the choice to hold references to related protos, or embed the related protos into the message itself. As a guiding principle, we choose to use references to implement relationships between Nodes (e.g. a parent/child relationship or dependency). This is because we often need to reference arbitrary Nodes when doing status computation.

For other relationships such as Nodes and their SLIs, or Clients and their UserJournerys, we choose to use composition. It's unlikely that we would need to access arbitrary SLIs or UserJournerys outside of the context of their containing Nodes.

## Derived Fields

Often, some proto fields can be derived from other fields within the same message. The choice to store the derived fields or compute them dynamically results in a tradeoff between compute and storage. Neither option is clearly superior for the scale of this project, especially without knowledge of the usage patterns of the derived fields a priori.

This choice arises from the need to store Node names, and the names of their children and parent. For simplicity and consistency, we choose to use a fully-qualified, period-delimited name to identify Nodes. That is, names should be in the format "System1.Service2.Endpoint3". We store the full name for the node itself, as well as its parent and children for convenient indexing into dictionaries mapping the Node name to the Node message. In this manner, we can avoid both graph traversals to access arbitrary Nodes, as well as the need for names to be globally unique.

# Callback Organization

In Dash, each component property can only be used as an output to a single callback function. However, each callback function can have multiple explicitly defined outputs. Additionally, outputs can be registered by pattern matching.

Thus, a natural approach is to have each callback function correspond to a single output component property. This callback determines the context under which it was invoked (i.e. which input component caused the callback to fire), and calls a helper function based on each input source.

A previous approach was to associate input components with individual callback functions. In this paradigm, the callback would be responsible for all UI changes (possibly across different output properties) resulting from a specific interaction. However, due to the limitations of Dash, this was shown to be infeasible.

Moreover, official guidance on organization of callback functions does not exist. We use this recommendation to organize the callbacks and the Dash app.

## Collapsing and Expanding Virtual Nodes

We decide to only allow collapsing top-level nodes (those without parents) to avoid the edge case of collapsing a node but not its parent.
We don't modify the underlying Nodes when creating virtual nodes. Instead, we create a Virtual Node data structure that keeps track of a set of the names of Nodes contained in the virtual node (its children).

We use this data structure to modify the list of cytoscape elements. Thus, virtual nodes can be thought of as a "view" over the graph.

Collapsing virtual nodes is relatively simple. We first remove all the nodes contained within the virtual node. Then, we iterate over the edges and replace the source/target of each edge with the virtual node name if the original source/target was in the set of virtual node's children. Finally, we remove edges with the same source/target (which were originally starting and ending from nodes within the same virtual node).

Expanding nodes is more difficult. A few approaches considered include:

1. Regenerate the set of nodes and edges that were previously replaced, from the set of real Nodes.
    a. It's easy to regenerate the nodes that were replaced by iterating over the virtual node's children. Since this set was flattened, we need to refer to the underlying Node protos to build parent/child relationships again.
    b. It's difficult to regenerate the edges. We need to iterate over every Node's Dependencies, and regenerate an edge if the Dependency target was a node inside the virtual node. However, this is difficult if we want to support nested virtual nodes, since Dependency source/targets point to real proto Nodes only. It's also difficult to regenerate the outgoing edges (originating from nodes within the virtual node).
2. Keeping a list of elements that were added and removed during collapsing.
    a. This works for collapsing and expanding a single virtual node, but becomes infeasible when with more than one virtual node.
    b. Consider: An edge exists between Node A and Node B. Node A is collapsed into Virtual Node X. The removed edge is (A, B) and the added edge is (X, B). Now, Node B is collapsed into Virtual Node Y. The removed edge is (X, B) and the

        added edge is (X, Y). If we want to expand Virtual Node X, we can’t directly add edge (X, B) back into the graph.

3. Hold an expanded/collapsed flag in the VirtualNode proto. Also keep a separate map associating node (virtual or non-virtual) name with their containing virtual node name.
    a. Regenerate the elements from the underlying proto Nodes every time a change is made (e.g. highlighting a user journey, collapsing/expanding nodes). Then apply the collapse operation to the collapsed virtual nodes, and simply add a parent virtual node for expanded virtual nodes.

Approach 3 is chosen. This approach also supports nesting virtual nodes.
Virtual nodes are currently persisted on the Dash server, meaning that changes involving virtual nodes will be viewed across all clients.
However, they aren’t persisted on the reporting server.

## Element Rendering Algorithm

The UJT uses Node and Client protos to generate a set of "elements," which are Nodes and Edges in a format understandable (dictionary) to Cytoscape, the graph visualization library. This is essentially a 1:1 transformation (no traversals are done). These cytoscape elements need to change based on various user interactions (e.g. highlighting a user journey, collapsing a node), while the Node and Client protos remain relatively static.

A central question arose when designing the rendering process: is it preferable to re-generate the elements from protos every time a change is necessary to the elements of the graph?

Some possible approaches:

1. We can always use protos to generate the cytoscape elements. Then, pass the cytoscape elements through a pipeline of additional transformations (highlight user journeys, collapse nodes).
    a. The benefit of this approach is that steps in the transformation pipeline don’t need to account for subsequent steps. In general, this approach performs repeated work but allows us to start with a clean slate every time.
    b. The initial generation of the elements takes O(V+E) (where V and E are # nodes and # edges in the entire graph). We could be able to avoid this computation cost with the following approach.
2. Alternatively, we can keep track of the state of the elements, and directly perform a single requested transform on the elements without referring to the protos, based on which user interaction was triggered.
    a. In some cases, this is asymptotically faster -- highlighting a user journey path takes O(nodes in user journey + edges in entire graph), but this doesn't seem important for small graphs. However, we avoid doing repeated work for the nodes that are unmodified.

   b. Moreover, since the user interactions could be performed in any order, each type of transformation on the cytoscape elements has to account for the others (usually preserving the changes made previously).
3. We generate the cytoscape elements from protos once, and memoize the result. If a change occurs that would modify the protos, we invalidate the elements and regenerate them. Then, we apply the entire set of transformations. This approach is a hybrid of 1 and 2.

Approach 3 is chosen to balance the complexity of implementation and the computational savings.