State Calculation

Background

Components and dependencies

In this assessment you will code a small and simplified piece of StackState's logic pertaining to the calculation of states. StackState visualizes a dependency graph of interconnected components. Each component can be connected to multiple other components. Each relation represents a dependency. The target of each relation is a component that the other component depends upon. A dependency can be uni-directional or bi-directional (meaning both components depend on each other).

States

A component has three types of states (properties of the component). These all can be (ordered from low to high):

- `no_data` - no state is known
- `clear` - the component is running fine
- `warning` - something is wrong
- `alert` - something went wrong

The three types of state are:

1. The *check* states. Each components has zero, one or multiple of these. (In StackState they represent states that are calculated by running checks on monitoring data, which is outside the scope of this assessment so you can just change them yourself.) These check states can be expected to change all the time. The check states influence the own state of the component.

2. The *own* state, singular, is the highest state of all check states. It is not affected by anything other then the check states. If the component has no check states the own state is `no_data`. (In StackState the own state represents the state that the component itself reports.) The own state of a component influences the derived state of the component

3. The *derived* state, singular, is either A) the *own state from warning and up* or if higher B) the *highest of the derived states of the components it depends on*. *Clear* state does not propagate, so if the own state of a component is *clear* or *no_data* and its dependent

derived states are *clear* or *no_data* then the derived state is set to *no_data*.

Events

It is important for our users to have traceability of events through the system. For example when Splunk reports a higher than normal error rate, StackState might generate an alert check state, which becomes the new own state, which triggers a whole host of derived states to change.

Simple Example

Take a dependency graph with two nodes. A component named "db" and a component named "app". The app is dependent on the db. The db has two check states. One of the check states named "CPU load" of the db goes to the warning state. The own state of the db becomes warning. The derived state of the db and app become warning.

JSON

The dependency graph and events can be expressed in JSON. Here is the complete scenario above, starting from an initial state, set of events and a final state.

Initial state

```
{
  "graph": {
    "components": [
      {
        "id": "app",
        "own_state": "no_data",
        "derived_state": "no_data",
        "check_states": {
          "CPU load": "no_data",
          "RAM usage": "no_data"
        },
        "depends_on": [
          "db"
        ]
      },
      {
        "id": "db",
        "own_state": "no_data",
        "derived_state": "no_data",
        "check_states": {
          "CPU load": "no_data",
```

```
            "RAM usage": "no_data"
          },
          "dependency_of": [
            "app"
          ]
        }
      ]
    }
}
```

Events

Here are two events that change initial state above:

```
{
  "events": [
    {
      "timestamp": "1",
      "component": "db",
      "check_state": "CPU load",
      "state": "warning"
    },
    {
      "timestamp": "2",
      "component": "app",
      "check_state": "CPU load",
      "state": "clear"
    }
  ]
}
```

Final state

The resulting state after the events have been processed can be expressed in the first format.

```
{
  "graph": {
    "components": [
      {
        "id": "app",
        "own_state": "clear",
        "derived_state": "warning",
        "check_states": {
          "CPU load": "clear",
          "RAM usage": "no_data"
        },
        "depends_on": [
```

```
            "db"
          ]
        },
        {
          "id": "db",
          "own_state": "warning",
          "derived_state": "warning",
          "check_states": {
            "CPU load": "warning",
            "RAM usage": "no_data"
          },
          "dependency_of": [
            "app"
          ]
        }
      ]
    }
  }
```

Notes

Keep the following in mind:

- Events are processed in the order of their timestamp.

Assignment

Implement the above stated domain and its logic so that a particular dependency graph and associated events result in the right dependency graph state.

Delivery

Your solution should be a command-line application that takes as input two parameters, the name of the initial state JSON file and the name of the events JSON file, and writes to standard output the final state of the dependency graph after processing all events.

Your solution must be started using a shell script called `run.sh` that accepts these two parameters. After the calculation completes, your program should write it's output in the format shown below.

Your solution will be run in an environment containing only the runtime of the target platform. Do not rely on the presence of shell scripts, build tools or other external dependencies.

Example:

```
> ./run.sh sample-initial.json sample-events.json
{
  "graph": {
    "components": [
      {
        "id": "app",
        "own_state": "clear",
        "derived_state": "warning",
        "check_states": {
          "CPU load": "clear",
          "RAM usage": "no_data"
        },
        "depends_on": [
          "db"
        ]
      },
      {
        "id": "db",
        "own_state": "warning",
        "derived_state": "warning",
        "check_states": {
          "CPU load": "warning",
          "RAM usage": "no_data"
        },
        "dependency_of": [
          "app"
        ]
      }
    ]
  }
}
```