IST 659: Server inventory and network management system

Final Project Report

Instructor: Prof. Chad Harper

Team:
Sharat Sripada

TABLE OF CONTENTS

Introduction

Server inventory and network management at scale has been a pain point for our Engineering Team. With hundreds of servers, switches and switch ports to manage across several production environments it sets a nearly perfect use-case to brace an RDBMS at the backend to store large records of structured data while presenting a simplistic frontend.

Through the knowledge acquired in IST-659 and guidance from Prof. Chad Harper the intent is to design a solution that will help greatly ease the day in a life of a developer.

Problem Statement

A production (aka prod.) environment typically comprises several servers and upstream switches (CLOS fabric comprising first-hop switches termed LEAF and a SPINE aggregate). Developers are building these prod. environments typically doing the following:

- Add servers to prod. to increase capacity

  (OR)

  Remove servers from prod. when servicing hardware faults or deprecating servers (due to end of life/expiry of service contracts)
- Configuring upstream switches to facilitate workloads or virtual machines (VM) on prod. servers to communicate with each other across the datacenter
- Setting up NFS mounts etc.
- Shuffling servers between prod. environments

Clearly, this is a cumbersome process proving detrimental to Engineering productivity. The endeavor is to automate the above steps end-to-end offering simplistic workflows via a Graphical User interface (GUI) or Application Programmable Interface (API) and utilizing an RDBMS at the backend.

Business rules and Entity Relationship Diagram (ERD)

Abstract

Consider the following narrative explaining the Business rules and scope:
Engineers run their code or features at scale building up prod. environments which would comprise servers and associative switches to facilitate communication.

Each prod. environment will have a dedicated VMWare vCenter managing workloads (Outer: to manage physical servers and Inner: to manage nested hypervisors), Runner virtual-machine (to stage, compile and execute code), an NFS mount-point which includes an ip-address and path, more than ONE dedicated virtual-LAN/VLAN IDs isolating Management, Overlay/data-path and NFS traffic.

Prod. environments are backed by ONE or more servers running VMWare ESX hypervisor and would comprise data related to server-manufacturer, a single ip-address assigned to the server to access it over the Management network, vmkernel ip-address to access NFS, memory and CPU capacities of the server and a parameter indicating whether the server was in-use. If a server were not to be in use it could either be deprecated or temporarily out-of-service servicing a hardware fault/upgrade.

Finally, all servers are managed by TWO upstream Layer-2 switches. The switch would comprise data regarding the manufacturer/vendor (this may also be needed to adjust the switch CLI syntax to automate networking actions), port, port-type, switch ip-address, switch type (Leaf/Spine) and finally more than one virtual-LAN/VLAN IDs isolating Management, Overlay/data-path and NFS traffic mapped to each port.

NOTE
Deprecated servers shall be cleaned up from the prod. environment correspondingly also cleaning up any network related references to switch or switch port.

Modeling Entities and Attributes

Based on the abstract following are the entities, attributes & datatypes:

Entities	Attributes	Attribute property	Datatype
Production	Prod. Name	Primary Key	varchar
	NFSIPAddress	Required	varchar
	NFSShare (path)	Required	varchar
	OuterVCIPAddress	Required, Unique	varchar
	InnerVCIPAddress	Required, Unique	varchar
	{VLANs}	Required, Multiple	Int
Server	Server Name	Primary Key	varchar
	Manufacturer	Required	varchar
	IPAddress	Required, Unique	varchar
	VMKernelIPAddress	Required, Unique	varchar
	MemoryInGB	Required	float
	CPUInCores	Required	int
	InUse	Required	varchar
	{VLANs}	Required, Multiple	int
Switch	Switch Name	Primary Key	varchar
	Manufacturer	Required	varchar
	Port	Required	varchar
	PortType	Required	varchar
	IPAddress	Required, Unique	varchar
	SwitchType	Required	varchar
FaultTicket	FaultID	Primary Key	varchar
	Description	Optional	varchar
	Severity	Optional	varchar

Entity Relationship Diagram (ERD)

The entities, attributes and relationships can be represented using an ERD. The relationships between entities is detailed as:

- **Prod. and Server entities**
  A prod. environment will comprise 1 or many servers
  (AND)
  A server can belong to ONE and ONLY ONE prod. environment, hence a ONE-MANY relationship
- **Server and Switch entities**
  A server would have its physical ports on Network Interface Card (NIC) connected to TWO (or multiple) switches.
  (AND)
  Likewise, a switch can comprise several servers connected to it, hence a MANY-MANY relationship.

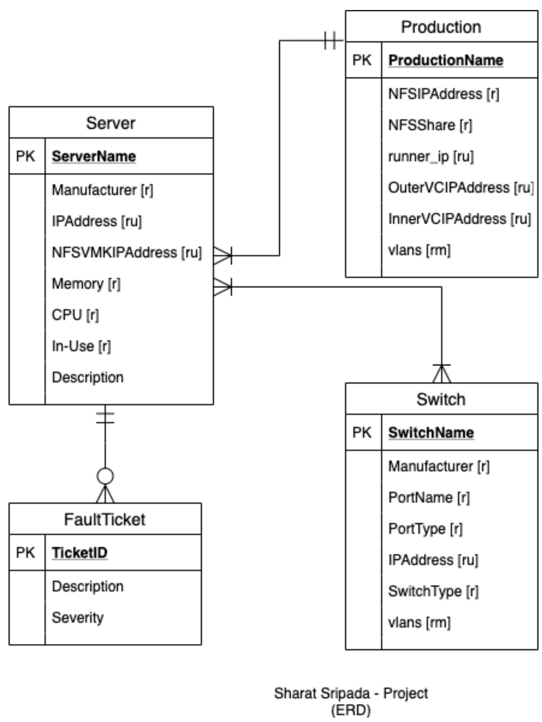


Figure-1: Entity Relationship Diagram

Logical Modeling and Enhanced Entity Relationship Diagram (E-ERD)

Translating the ERD in Figure-1 to a logical diagram, we do the following:

- Map entities to tables
- In some cases, prod. name or switch name is too long and sometimes cannot be guaranteed uniqueness so, use surrogate keys instead (int identity datatype)
- All attributes are correspondingly mapped to column names following the best-practices for naming conventions (all lower case & separate with underscore for spaces)
- Map relationships between entities as relationships between tables

- Map the MANY-MANY relationship between Server and Switch entities to a MANY-ONE/ONE-MANY relationship using a bridge or associative table called server_switch_info

- Vlans are multi-valued and comprise other non-key attributes like Vlan type indicating Management, Overlay or NFS networks. This is correspondingly mapped to the tables belongs and vlan_table.

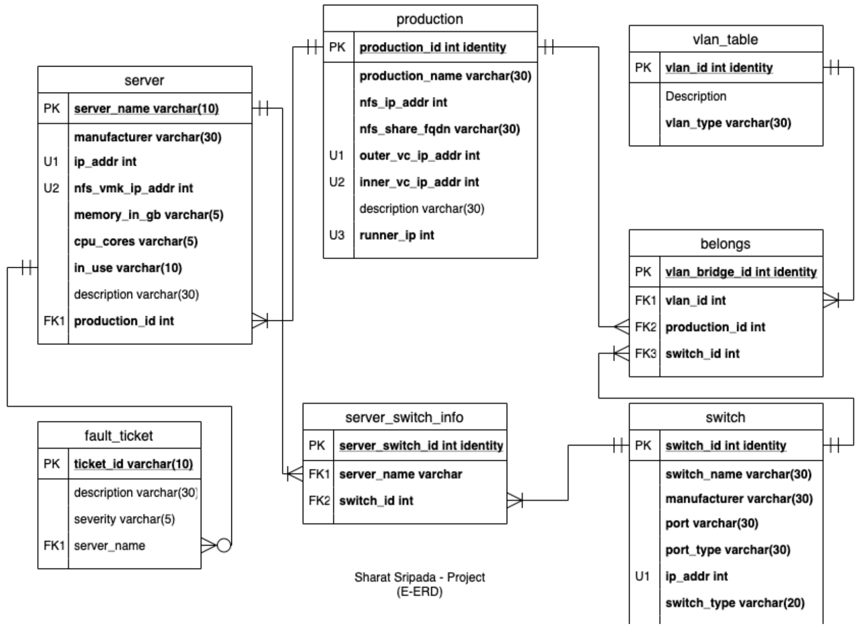


Figure-2: Enhanced Entity Relationship Diagram

Normalization

After placing real data from THREE Prod. environments and running the tables through the normalization steps (1NF -> 2NF -> 3NF process), saw the need for a few changes.

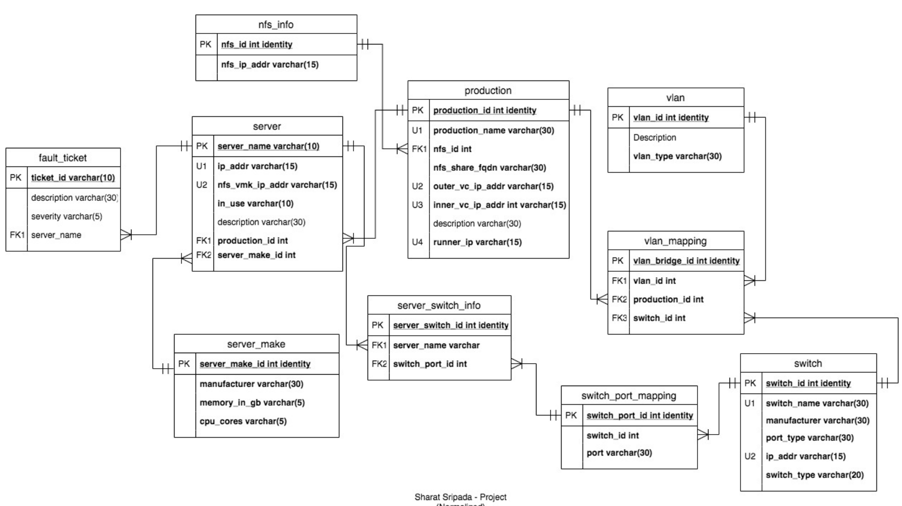


Figure-3: 3NF tables after normalization

Summary of changes:
- Total number of tables increased from SEVEN to TEN. Newly added tables are nfs_info, switch_port_mapping and server_make
- Changed type for ip-address column-names from int to varchar(15)
- Change name of table 'belongs' to 'vlan_mapping'

Implementation – Dive into my first Flask application

For the implementation, Flask - a web application framework in Python was chosen. The framework along with certain libraries offers a rich platform to develop an end-to-end solution comprising:
- Graphical User Interface (GUI) – Using the *@app.route()* decorator build HTML with embedded JavaScript/AJAX for dynamic front-end objects/charts
- API framework – Using the *@app.route()* decorator to build POST, GET, PATCH, DELETE API methods. This would then correspondingly translate to CREATE, SELECT, UPDATE and DELETE SQL operations
- RDBMS/SQL database - Sqlite3 libraries for create and administer databases

The code is organized and maintained in the following directory structure:

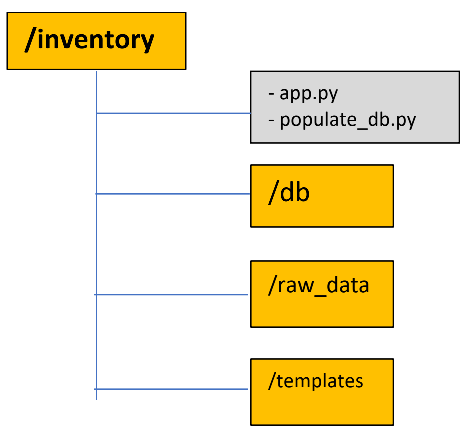


Figure-4: Organizing files **(Legend: Orange fills indicate directories)**

The application itself is developed within *app.py* under the parent directory /inventory which when running would listen on port *:5000 per this in the code:

```
/inventory[687](my_flask_app)# less app.py.
.
558 app.run(host='0.0.0.0')
```

Other directories and files:
- /db comprise the sqlite database files (.db files)
- /raw_data comprise data in csv format translated from excel files (representing database tables in 3NF).
    - *populate_db.py* will read data and populate corresponding tables
- /templates will comprise all essential front-end related html/java-script files

Here's a detailed dump of directories and files under /inventory:

```
# ls *
app.py  populate_db.py

db:
test.db  testinfo.db  testinfra.db

raw_data:
fault_ticket  nfs_info  prod  server  server_make  server_switch_info  switch
switch_port_mapping  vlan  vlan_mapping

static:

templates:
chart.html  index.html
```

NOTE:
The files under raw_data correspond 1:1 to tables in the database.

Exploring database **C**reate, **R**ead**, U**pdate, **D**elete operations

With all necessary constructs in place this section now encompasses details of everything that went into building the database, corresponding tables and performing the Create, Read, Update, Delete operations, commonly hence referred as CRUD ops.

NOTE: At the time of putting this report in place, only Create and Read were implemented. See section Future Work for further details regarding pending tasks.

Create RDBMS database and tables

Using the Python sqlite3 library, capturing an excerpt of how the tables were created using SQL in *db/testinfra.db* database. Since table creation is within the __*init*__ constructor, the corresponding CREATE SQL commands kick in on *class SqlLibTestInfra()* instantiation. Checks are in place to walk all tables not comprising 'sqlite' (see LINEs 19-20) prior, to prevent redundant work.

Table creation is in a specific sequence adhering to CONSTRAINTs or references to FOREIGN_KEY attributes.

Below is the corresponding code:

```python
class SqlLibTestInfra(object):
    def __init__(self):
        self.conn = sqlite3.connect('db/testinfra.db', check_same_thread=False)
        _user_tables = ('nfs_info', 'production', 'server', 'server_make',
                'fault_ticket', 'switch_port_mapping', 'switch',
                'server_switch_info', 'vlan_mapping', 'vlan')
        _res = self.conn.execute('select name from sqlite_master where \
                type=\'table\' and name NOT LIKE \'sqlite_%\';')
        db_tables = _res.fetchall()
        for user_table in _user_tables:
            table_present = False
            for db_table in db_tables:
                if user_table == db_table[0]:
                    table_present = True
                    break
            if not table_present:
                if user_table == 'nfs_info':
                    self.conn.execute('create table nfs_info (\
                        nfs_id int PRIMARY KEY, \
                        nfs_ip_addr varchar(15) not null);')
                if user_table == 'production':
                    self.conn.execute('create table production (\
                        production_id int PRIMARY KEY, \
                        production_name varchar(2) not null, \
                        nfs_share_fqdn varchar(30) not null, \
                        outer_vc_ip varchar(15) not null, \
                        inner_vc_ip varchar(15) not null, \
                        description varchar(15) not null, \
                        runner_ip varchar(15) not null, \
                        nfs_id int not null, \
                        FOREIGN KEY (nfs_id) \
                          REFERENCES nfs_info(nfs_id));')
                if user_table == 'server':
                    self.conn.execute('create table server (\
                        server_name varchar(30) PRIMARY KEY,\
                        server_make_id int, \
                        ip_addr varchar(15), \
                        nfs_vmk_ip_addr varchar(15), \
                        in_use varchar(10), \
                        description varchar(20), \
                        production_id int not null, \
                        FOREIGN KEY (production_id) \
                          REFERENCES production(production_id));')
```

Figure-5: Table creation code using SQLite libraries/commands

API gateway – @app.route decorators

Flask has *@app.route* decorators to build very comprehensive API gateways. Below is a proponent of its usage to cater to common POST and GET methods – *common_db_crud_ops* (more of this in the next section) then forks the requests to SQL INSERT or SELECT statements respectively.

APIs would eventually be extended to REST methods PATCH and DELETE that correspond to SQL UPDATE and DELETE statements respectively.

```python
# Server/switch inventory - apis
@app.route('/api/v1/production/', methods=['POST', 'GET'])
def production():
    return common_db_crud_ops('production', request)

@app.route('/api/v1/nfs_info/', methods=['POST', 'GET'])
def nfs_info():
    return common_db_crud_ops('nfs_info', request)

@app.route('/api/v1/server/', methods=['POST', 'GET'])
def server():
    return common_db_crud_ops('server', request)

@app.route('/api/v1/server_make/', methods=['POST', 'GET'])
def server_make():
    return common_db_crud_ops('server_make', request)

@app.route('/api/v1/fault_ticket/', methods=['POST', 'GET'])
def fault_ticket():
    return common_db_crud_ops('fault_ticket', request)
```

Figure-6: APIs implemented to insert, get data

Populating data – A fully functioning database

With API gateway and database in place, we start to populate the tables with real data.

Explored mockaroo.com but data corresponding to some tables or columns was extremely tailored to our use-case. So, all data is real - gathered from a few prod. environments (this is a tedious/manual process and is still ongoing). It is then organized in fully compliant 3NF equivalent excel sheets before converting and exporting them in .csv format.

Python utility

Finally, a python utility *db_populate.py* that would:

- scrape .csv files in directory /raw_data
- do POST API method calls using *Requests* libary to INSERT data into corresponding tables
- (AND) do GET API method calls to verify object counts

The result of the GET API method is a JSON body comprising all entries in the table and a *result_count* indicating number of table entries.

Here's an excerpt of the python utility:

```python

with open('raw_data/%s' %FILE) as f:
        lines = [line.rstrip() for line in f]

path = {'prod': '/api/v1/production/',
        'nfs_info': '/api/v1/nfs_info/',
        'server': '/api/v1/server/',
        'server_make': '/api/v1/server_make/',
        'fault_ticket': '/api/v1/fault_ticket/',
        'switch': '/api/v1/switch/',
        'vlan_mapping': '/api/v1/vlan_mapping/',
        'vlan': '/api/v1/vlan/',
        'switch_port_mapping': '/api/v1/switch_port_mapping/',
        'server_switch_info': '/api/v1/server_switch_info/'
      }

header = True
url = endpoint + path[FILE]
for line in lines:
    if header:
        headers = line.split()
        header = False
        continue
    data = {}
    index = 0
    entry = line.split()
    while index < len(entry):
        data[headers[index]] = entry[index]
        index += 1
    r = requests.post(url = url, json = data)
    print(r)
g = requests.get(url = url)
print(g.text)
```

Figure-7: *db_populate.py* utility to populate tables

Forking function – Bridge between API and database

Function *common_db_crud_ops* forks incoming API requests to corresponding database operations (to be extended to PATCH and DELETE)

```
def common_db_crud_ops(table, request):
    if request.method == 'POST':
        if request.is_json:
            status = db_testinfra.insert(table, request.json)
            return status
    if request.method == 'GET':
        data = db_testinfra.get(table)
        return jsonify(data)
```

Figure-8: Function forking api calls to corresponding DB ops

SQL – Under the hood data persistence

And finally, it is down to the select few functions to insert and fetch data to/from the database and thus persist it. Below are the *insert* and *get* methods implemented within the SqlLibTestInfra class:

```
def insert(self, table, data_in_json):
    if table == 'nfs_info':
        _nfs_id = data_in_json['nfs_id']
        _nfs_ip_addr = data_in_json['nfs_ip_addr']
        try:
            self.conn.execute('insert into nfs_info (nfs_id, nfs_ip_addr) \
                    VALUES("%d", "%s");' %(_nfs_id, _nfs_ip_addr))
            self.conn.commit()
            return (return_codes['insert_success'][1],
                    return_codes['insert_success'][0])
        except Error as e:
            return (str(e) + '\n', 500)

    if table == 'production':
        _prod_id = int(data_in_json['production_id'])
        _prod_name = data_in_json['production_name']
        _nfs_share_fqdn = data_in_json['nfs_share_fqdn']
        _outer_vc_ip = data_in_json['outer_vc_ip']
        _inner_vc_ip = data_in_json['inner_vc_ip']
        try:
            _desc = data_in_json['description']
        except KeyError:
            _desc = ''
        _runner_ip = data_in_json['runner_ip']
        _nfs_id = int(data_in_json['nfs_id'])
        try:
            self.conn.execute('insert into production (production_id, \
                    production_name, nfs_share_fqdn, outer_vc_ip, \
                    inner_vc_ip, description, runner_ip, nfs_id)  \
                    VALUES("%s", "%s", "%s", "%s", "%s", "%s", "%s", "%s");' \
                    %(_prod_id, _prod_name, _nfs_share_fqdn, _outer_vc_ip,
                        _inner_vc_ip, _desc, _runner_ip, _nfs_id))
            self.conn.commit()
            return (return_codes['insert_success'][1],
                    return_codes['insert_success'][0])
        except Error as e:
            return (str(e) + '\n', 500)
```

```
def get(self, table):
    self.conn.row_factory = sqlite3.Row
    if table == 'nfs_info':
        rows = self.conn.execute('select * from nfs_info;')
    elif table == 'production':
        rows = self.conn.execute('select * from production;')
    elif table == 'server':
        rows = self.conn.execute('select * from server;')
    elif table == 'server_make':
        rows = self.conn.execute('select * from server_make;')
    elif table == 'fault_ticket':
        rows = self.conn.execute('select * from fault_ticket;')
    elif table == 'switch_port_mapping':
        rows = self.conn.execute('select * from switch_port_mapping;')
    elif table == 'switch':
        rows = self.conn.execute('select * from switch;')
    elif table == 'server_switch_info':
        rows = self.conn.execute('select * from server_switch_info;')
    elif table == 'vlan':
        rows = self.conn.execute('select * from vlan;')
    elif table == 'vlan_mapping':
        rows = self.conn.execute('select * from vlan_mapping;')
    _res = OrderedDict()
    _res_list = []
    for row in rows.fetchall():
        _res_list.append(dict(row))
    _res['result_count'] = len(_res_list)
    _res['results'] = _res_list
    return _res
```

Figure-9: *insert* function to INSERT data into database AND
get function to SELECT data and return in key-value/JSON format

Analytics

The idea with this framework was to make it efficient for Developers to consume resources within their prod. environments by inheriting APIs outlined in this document (including but not limited to, yet).

Subsequently, it is also critical to capture information and monitor resources given the scale spans several servers and switches. After all, 'No resource pool is infinite' AND '..everybody wants resources!'. While the dashboard will significantly evolve in due course of time this section attempts to capture in gist, the naïve first few steps.

SQL – Beyond native CRUD

Putting together some explorations with SQL statements (comprising JOINs, GROUP BY etc.) that helped gain insights to set the stage for rich analytics.

Here are a few initial metrics to visualize:

- Server usage – grouped by each prod./Dev environment
- Switch usage – group by each prod./Dev environment
- Server usage – grouped by make, memory & cpu specs.

Below are corresponding SQL queries:

```
    def custom_queries(self, query_type):
        if query_type == 'GET_SERVERS_PER_PROD':
            _res = self.conn.execute('select production.production_name, ' \
                                     'count(server.server_name) from server ' \
                                     'JOIN production ON ' \
                                     'server.production_id = production.production_id ' \
                                     'GROUP BY production.production_name')
        if query_type == 'GET_SWITCHES_PER_PROD':
            _res = self.conn.execute('select production.production_name, ' \
                                     'count(switch_port_mapping.switch_port_id) ' \
                                     'from server '\
                                     'JOIN production ON '\
                                     'server.production_id = production.production_id '\
                                     'JOIN server_switch_info ON '\
                                     'server.server_name = server_switch_info.server_name ' \
                                     'JOIN switch_port_mapping ON '\
                                     'server_switch_info.switch_port_id = switch_port_mapping.switch_port_id ' \
                                     'GROUP BY production.production_name')
        if query_type == 'GET_SERVER_TYPES':
            _res = self.conn.execute('select server_make.manufacturer || server_make.memory_in_gb ||  server_make.cpu_cores AS server_type, '\
                                     'count(server_make.manufacturer) from server '\
                                     'JOIN server_make on server.server_make_id = server_make.server_make_id '\
                                     'GROUP BY (server_type)')
        return _res.fetchall()
```

Figure-10: Queries to generate data insights

Dashboard

Data from *custom_queries* is packaged as labels and values using the python zip function before using Flask's *render_template* utility:

```
    return render_template('chart.html',
                           title_1='SERVERS - Grouped by Production',
                           max_1=int(1.1 * max(values_1)),
                           labels_1=labels_1,
                           values_1=values_1,
                           title_2='SWITCHES - Grouped by Production',
                           max_2=int(1.1 * max(values_2)),
                           labels_2=labels_2,
                           values_2=values_2,
                           title_3='SERVERS - Grouped by H/W Specs',
                           max_3=int(1.1 * max(values_3)),
                           set=zip(values_3, labels_3, colors)
                           )
```

Figure-11: Queries to generate data insights

The html code in *chart.html* translates the data into bar and pie charts.

Below is a first peek at the dashboard accessed via a browser:
URL http://<ip-address>:5000/

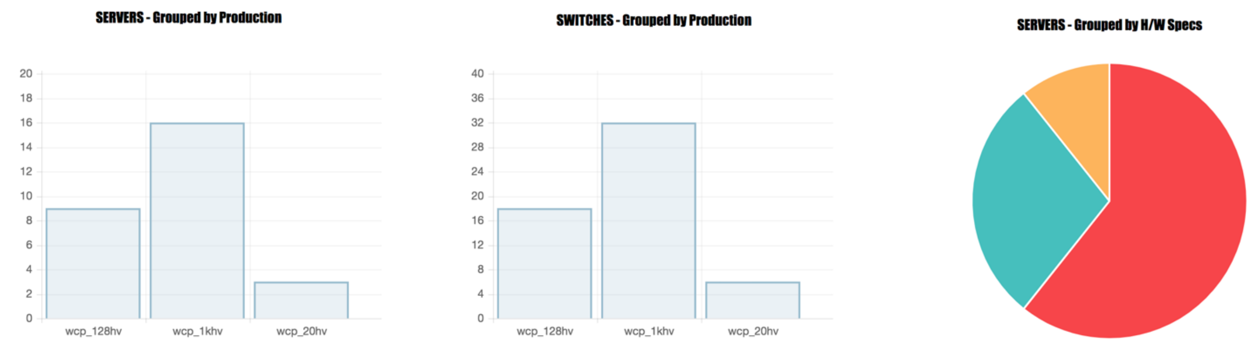


Figure-12: Infrastructure analytics dashboard

Conclusion and Future work

This project has long been on my mind and I am grateful it is finally underway (and beyond) during the course of this semester.

Achievements include:

- Conceptualization to realization using techniques imbibed via IST-659 in designing databases
    - All tables organically went from ERD through logical-modeling and normalization process before being 3NF compliant
- All tables have real data from THREE of our prod. environments which greatly establishes a workflow we have in place for consumption
- Setting in place a rich web application framework

Future work

There is significant work lined up and I have captured a few:

1. Implementation of PATCH and DELETE API methods that correspondingly translate to SQL UPDATE and DELETE operations. This is an essential part of the CRUD operations that is pending implementation.

   Since server allocations can constantly change hands across prod. environments it is imperative these methods be available before rolling the framework out to Developer community.

2. Use concepts of IST-659:
    - Use of procedures/functions to reduce error and code-redundancy
    - Efficient SQL commands

3. Migrating data for several hundred servers and switches across prod. environments to the web framework and database.

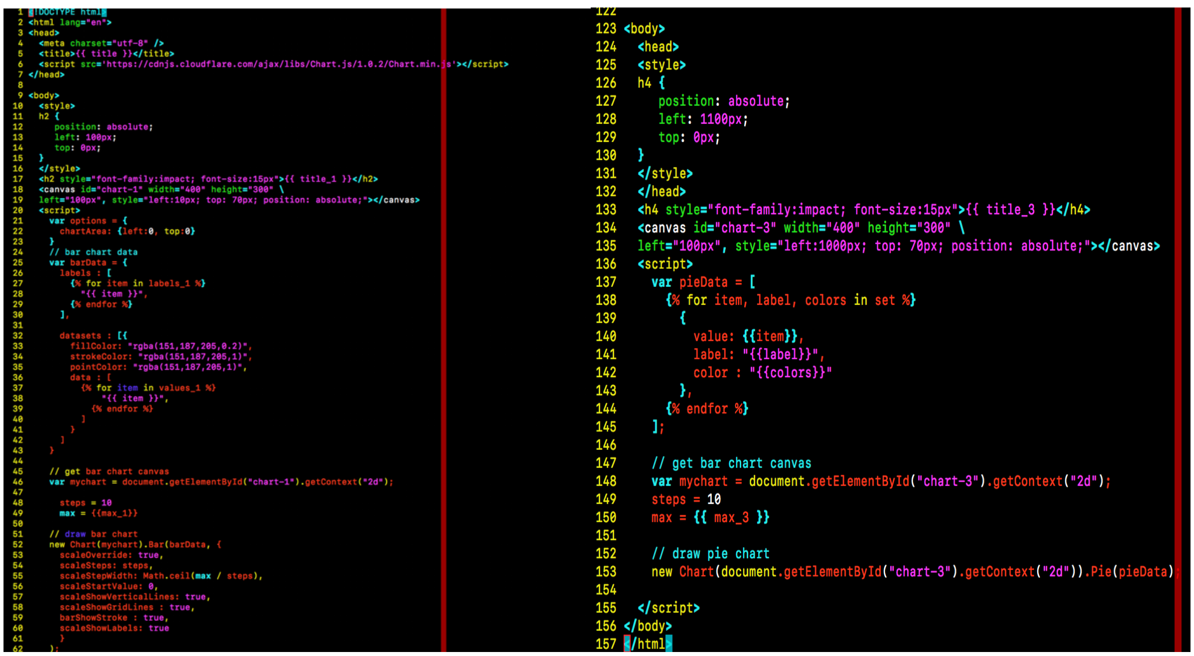
Subsequently, programming switches with network configuration using Tcl/Tk, expect or ansible seems a tangible option since all the data is co-located and available across a few tables in the database.

4. Extend analytics dashboard to comprise more data and visualizations – work with our internal stakeholders regarding critical metrics.

Appendix

Front-end code for dashboard (chart.html)

Below is the code under /templates, namely chart.html which is used by Flask to route GUI requests via the render_template utility



```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>{{ title }}</title>
  <script src='https://cdnjs.cloudflare.com/ajax/libs/Chart.js/1.0.2/Chart.min.js'></script>
</head>

<body>
  <style>
  h2 {
     position: absolute;
     left: 100px;
     top: 0px;
  }
  </style>
  <h2 style="font-family:impact; font-size:15px">{{ title_1 }}</h2>
  <canvas id="chart-1" width="400" height="300" \
  left="100px", style="left:10px; top: 70px; position: absolute;"></canvas>
  <script>
    var options = {
      chartArea: {left:0, top:0}
    }
    // bar chart data
    var barData = {
      labels : [
        {% for item in labels_1 %}
          "{{ item }}",
        {% endfor %}
      ],

      datasets : [{
        fillColor: "rgba(151,187,205,0.2)",
        strokeColor: "rgba(151,187,205,1)",
        pointColor: "rgba(151,187,205,1)",
        data : [
          {% for item in values_1 %}
              "{{ item }}",
            {% endfor %}
          ]
        }
      ]
    }

    // get bar chart canvas
    var mychart = document.getElementById("chart-1").getContext("2d");

      steps = 10
      max = {{max_1}}

    // draw bar chart
    new Chart(mychart).Bar(barData, {
      scaleOverride: true,
      scaleSteps: steps,
      scaleStepWidth: Math.ceil(max / steps),
      scaleStartValue: 0,
      scaleShowVerticalLines: true,
      scaleShowGridLines : true,
      barShowStroke : true,
      scaleShowLabels: true
      }
    );
```

```
<body>
  <head>
  <style>
  h4 {
     position: absolute;
     left: 1100px;
     top: 0px;
  }
  </style>
  </head>
  <h4 style="font-family:impact; font-size:15px">{{ title_3 }}</h4>
  <canvas id="chart-3" width="400" height="300" \
  left="100px", style="left:1000px; top: 70px; position: absolute;"></canvas>
  <script>
    var pieData = [
      {% for item, label, colors in set %}
        {
          value: {{item}},
          label: "{{label}}",
          color : "{{colors}}"
        },
      {% endfor %}
    ];

    // get bar chart canvas
    var mychart = document.getElementById("chart-3").getContext("2d");
    steps = 10
    max = {{ max_3 }}

    // draw pie chart
    new Chart(document.getElementById("chart-3").getContext("2d")).Pie(pieData)

  </script>
</body>
</html>
```

Figure-13: HTML code to create a bar-chart and pie-chart