Overview of the problem

Configuring large systems is a very complex problem. Often hundreds or thousands of different machines like servers, working-stations (what would be third good example here?) require constant maintenance. Updating, installing new software or reconfiguring only one machine often leads to unexpected results. Performing those actions on such extreme scale and under rigid requirements lead to non-trivial approaches – often creation of new tools which are complex programs on their own. Among them are DSLs (Domain Specific Languages) which are offering standardized techniques for operating with large scale systems. To name a few: LCFG, Puppet, SmartFrog, Ansible and L3. In this paper project L3 language is going to be described and then compared to another existing technology – Google's App Engine configuration. Finally it would be also tested if L3 can be used to improve solution from Google.

Problems of Declarative languages??

Configuration languages??

Declarative programming is one of many styles of programming. In contrast to others it describes final outcome of a program instead of steps required to achieve that, in another words: what has to be done instead of how it has to be done. Expressiveness and other desired characteristics of declarative language are achieved by reducing control over underlying implementation. Because of that declarative languages are often, domain specific. Such languages are not meant to solve every possible problem but instead they are tailored for a specific set of problems. Some examples of DLSs are: HTML (which describes structure of a websites), CSS (which describe how websites look) and also many system configuration languages. Moving away implementation details makes writing correct programs easier, a very desirable property for critical software. Unfortunately if a problem is out of the scope of the DSL’s expressiveness it might become very hard or impossible to solve. Thus DSL is a bad choice if domain of the problem is too broad or unknown. System configuration is a closed domain with well described specification hence most of solutions are DSL’s.

Google's configuration language

On April 2008 Google launched Google App Engine (GAE), cloud computing platform for developing and hosting web applications on servers owned by Google. Since then GAE attracted many customers and became one of the Google’s best products. Among features of GAE is a simplified process of deployment and managing components like databases, virtual-machines and firewalls. Currently except command-line deployer Google is not providing users any specialized tool for configuring their product. Instead a combination of technologies not developed for configuring systems is used: data-serialization format Yaml, procedural programming language Python and templating language designed for webpages and webapps Jinja2. This decision could be driven by popularity of Jinja2 and Python among developers, although this argument does not applies to Yaml, which could be substituted by JSON – well-known data format. Nevertheless current approach makes it harder for users to properly configure their applications. Undocumented behavior and many unexpected nuances are leading to code that is harder to write and maintain.

Configuration of an application in GAE starts with a single file with Yaml format. Every part of infrastructure (called resource), like virtual-machine, firewall or database requires to be described inside Yaml file. Every resource has own properties which are describing its characteristics, like name (arbitrary, chosen by developer), type (type specifies what resource it actually is) or computing power. As application grows single file might become hard to maintain, Google offers Jinja2 templates and Python-generated files as a solution. Templates are also helpful when creating several similar instances, for example different virtual-machines for different continents. Each template must be placed in its own file, and requires to be imported into main Yaml file in order to be used. Code snippets below are examples of using templates.

Yaml file:

imports:  
- path: vm-template.jinja  
  
resources:  
- name: vm-instance  
  type: vm-template.jinja

Vm-template.jinja file:

resources:  
- name: vm-instance  
  type: compute.v1.instance  
  properties:  
    disks:  
    - deviceName: boot  
      type: PERSISTENT  
      boot: true  
      autoDelete: true  
      initializeParams:  
        sourceImage: https://www.googleapis.com/compute/v1/projects/debian-cloud/global/images/debian-7-wheezy-v20150526  
    machineType: https://www.googleapis.com/compute/v1/projects/myproject/zones/us-central1-f/machineTypes/f1-micro  
    networkInterfaces:  
    - network: $(ref.a-new-network.selfLink)  
      accessConfigs:  
      - name: External NAT  
        type: ONE\_TO\_ONE\_NAT  
    zone: us-central1-f

Vm-template.jinja file uses another useful feature of Jinja2 – references.

‘$(ref.a-new-network.selfLink)’ access property ‘selfLink’ from resource ‘a-new-network’. Google’s deployer will substitute referenced value during deployment, it also resolves all imports.

It is also possible to nest templates - include templates inside another templates. Example below illustrates that concept.

Firewall-template.jinja file:

resources:  
- name: a-firewall-rule  
  type: compute.v1.firewall  
  properties:  
    network: $(ref.a-new-network.selfLink)  
    sourceRanges: ["0.0.0.0/0"]  
    allowed:  
    - IPProtocol: TCP  
      ports: ["80"]

Network-template.jinja file:

resources:  
- name: a-new-network  
  type: compute.v1.network  
  properties:  
    IPv4Range: 10.0.0.1/16

Compute-engine-template.jinja file that combines all previously created templates:

resources:  
- name: vm-instance  
  type: vm-template.jinja  
- name: network-1  
  type: network-template.jinja  
- name: firewall-1  
  type: firewall-template.jinja

Yaml file:

imports:  
- path: vm-template.jinja  
- path: network-template.jinja  
- path: firewall-template.jinja  
- path: compute-engine-template.jinja  
  
resources:  
- name: compute-engine-setup  
  type: compute-engine-template.jinja

Note that all imports have to be defined inside Yaml file, not inside Compute-engine-template.jinja file that is combining other templates.

Last feature of GAE configuration are environmental and template variables. By using them it is possible to provide values for properties from files using templates, or in case of an environmental variables to obtain values like deployment name or the project ID. Vm-template.jinja file rewritten to use template variables looks like this:

resources:  
**- name: {{ env["name"] }}**  
  type: compute.v1.instance  
  properties:  
    disks:  
    - autoDelete: true  
      type: PERSISTENT  
      boot: true  
      deviceName: boot  
      initializeParams:  
        sourceImage:  https://www.googleapis.com/compute/v1/projects/debian-cloud/global/images/debian-7-wheezy-v20150526  
    **machineType: https://www.googleapis.com/compute/v1/projects/{{ env["project"] }}/zones/{{ properties["zone"] }}/machineTypes/{{ properties["machineType"] }}**  
    networkInterfaces:  
    **- network: $(ref.{{ properties["network"] }}.selfLink)**  
      accessConfigs:  
      - name: External NAT  
        type: ONE\_TO\_ONE\_NAT  
    **zone: {{ properties["zone"] }}**

Now file that uses that template can provide values for name, zone, machineType and network properties. Relevant part of that file might look like that:

**{% set NETWORK\_NAME = "a-new-network" %}**

resources:  
- name: the-first-vm  
  type: vm-template.jinja  
  **properties:  
    machineType: f1-micro  
    zone: us-central1-f  
    network:** {{ NETWORK\_NAME }}

This code snippet also shows usage of Jinja2 variables, in this example "a-new-network" will be substituted for every ‘**{{ NETWORK\_NAME }}’.** This reduces redundancy of information and will make it easier in the future to change value of network. Full reference and official Google’s tutorials are available here <https://cloud.google.com/deployment-manager/overview> .

(WIP)

Here I will start talking about problems

Process of configuration combines two different languages of which none is dedicated to system configuration.

It is possible to import a template but not use it which later might become a source of confusion, especially when multiple developers are working on same code, thus at least warning should be shown, if not an error that stops configuration from being deployed.

L3 configuration language

(??) here I will describe features and semantics of the L3? How deeply should I go with that?

Can L3 be used with Google's?

Conclusions