So you want to build a template?

An explanation and motivation for templates, from a producer's point of view

This specification describes the mindset we have for each mu.semte.ch template. It describes the goals of a template, the soft constraints, hints at method naming, and lists key features which a template should have.

# Why templates

Templates make it easier to implement new microservices. They help you get started with the technology you know and love, or with a new technology.

Templates should be predictable. The naming and structure of templates should therefore match certain standards. Templates make it easy to get started with the technology you know and love, or help you understand the structure of a service written in a language you barely know.

Templates make it easy to follow the constraints mu.semte.ch places on services. They should therefore contain a basic set of features which ensures users can easily develop applications in them.

Templates are our way to familiarize users with mu.semte.ch and to ensure users follow the best practices where possible.

# Template features

A template should make sure that the mu.semte.ch ecosystem keeps running. The construction of a template happens in a set of stages. First one should implement a basic service in the base technology, to get a feeling of which technologies are suited for the template. Next up is the construction of a PoC (Proof of Concept). Then comes the standardisation, which requires some extra features to ensure the template can be used in practice.

### Required for the PoC:

* Respond to HTTP requests
  + Parse JSON & JSONAPI payloads
  + Support implementation of GET PUT PATCH POST DELETE
  + On port 80 during production
* Send out SPARQL queries & updates over HTTP
  + Perform JSON request to SPARQL HTTP endpoint
  + Parse JSON query response
  + Support Virtuoso Open Source [1]
* Generate HTTP response
  + Build JSON & JSONAPI body
  + Set outgoing HTTP headers
* User-code
  + Load user-code from /app
  + Download & install dependencies on startup

### Requirements for full template:

* **All requirements for the PoC**
* Configuration
  + Access environment variables [2]
  + Support live-reload for development [3]
* Respond to HTTP requests
  + Parse incoming HTTP headers
  + Destructure request URL
  + Parse query parameters [4]
* Send out SPARQL queries & updates over HTTP
  + Set headers of the SPARQL HTTP request [5]
  + Provide escaping methods for SPARQL input
* User-code
  + Load user configuration code from /config
  + Use ONBUILD to automate template extension [6]
  + Support the generation of UUIDs, namely uuid-v1
* Enhancing workflow
  + Use /template-name/setup.sh for downloading all dependencies into the image[7]
  + Use /template-name/startup.sh for launching the webserver.
  + Use the ENVIRONMENT variable to enable development mode & live-reload[8]
    - Debugging console (if possible)
    - Enable debugging symbols (if possible)
    - Enable error output (if configurable)

# Levels of abstraction

Most developers like to abstract things. Make sure that nothing is repeated. Many of us like to get started quickly. These two ideas often don't go hand-in-hand. We dismiss non-generic abstractions and require consistent naming.

Getting started should be trivial and has the highest priority. It should be intuitive to start using a new template. It should be easy to read code that was written in a template, even if it isn't yours.

The following is a description of various topics on which consistency should be held. The description should be adapted to the use of your language. Ie: if you'd call sparql\_query in Ruby, sparqlQuery in Java and sparql-query in Common Lisp.

## Defining HTTP endpoints

Users will define HTTP endpoints to which their service will respond. Separate methods should be defined for separate types of call. The method should simply contain the name of HTTP verb, and the URL to which we respond. The system should allow the user to destructure the input and access the destructured contents. Calls may be defined in a block, but if that is the case, then the block should be repeatable.

Essentials:

* Use simple name of HTTP verb
* Allow straight-forward destructuring of URL
* Support finding query parameters

Example in Common Lisp:

| (defcall :get (“say” “hello” “to” name)  ;; responds to GET /say/hello/to/Aad?title=Sir  ;; name is bound to Aad  ;; (get-parameter “title”) yields “Sir”  ) |
| --- |

Example in Ruby:

| get “say/hello/to/:name” do |name|  # responds to GET /say/hello/to/Erika?title=Madam  # name is bound to erika  # params[:title] yields “Madam”  end |
| --- |

Another example in Ruby:

| define\_calls do |server|  server.get “say/hello/to/:name” do |name|  # responds to GET /say/hello/to/Erika?title=Madam  # name is bound to erika  # params[:title] yields “Madam”  end  end |
| --- |

## Executing SPARQL queries

We should mitigate SPARQL-injection whilst still ensuring the SPARQL queries are easy to recognise. Allow for injecting variables in a SPARQL-query string. Support escaping as separate methods. Query results should be parsed automatically so the user can easily process them.

A SPARQL endpoint often differs in query and update endpoints. Hence we advise to use those two names for executing the queries.

For languages which don't allow injecting variables into strings, magic is allowed. Keep it to an absolute minimum and try to mimic what is commonly found in other languages. It should be something which you can explain in a single sentence.

Common Lisp example:

| (sparql-query “SELECT ?s ?p ?o WHERE  GRAPH <<http://mu.semte.ch/application>> {  ?s a foaf:Agent;  foaf:name ~A;  ?p ?o.  }”, (sparql-escape-string “Felix”))  (sparql-update “INSERT DATA  GRAPH <<http://mu.semte.ch/application>> {  ext:Aad a foaf:Agent.  }”) |
| --- |

J

Ruby example:

| query “SELECT ?s ?p ?o WHERE  GRAPH <<http://mu.semte.ch/application>> {  ?s a foaf:Agent;  foaf:name #{username.sparql\_escape};  ?p ?o.  }” |
| --- |

# Getting started

Building a template without knowing the domain is near impossible. The following steps guide you into building a usable template.

First select the technology you would like to support. Look around for web frameworks which match the ideology of microservices. Search for the ‘Ruby Sinatra’ competitor for your language.

Your next challenge is to implement a simple microservice for this purpose. As an example, you can write a service that returns the amount of Tasks in the store, and which allows you to set a Task as done. The service itself is not of importance, you'll learn lessons about the framework when you implement the microservice.

Don't forget to run your microservice in a toy mu.semte.ch project. That way you'll know that it actually works. [Mu-project](https://github.com/mu-semtech/mu-project) offers a good starting point.

Once your service works, you can start abstracting it. Much of the code you've built would be used again when you implement another service in the same language. Go over the list of the PoC, ensure you've implemented each feature, and put this in a separate project. This project will become the template to use for the Tasks service. If all goes well, you'll end with a PoC template service, and a much shorter Tasks service.

Next up is going over the list of features needed for the full template. Go over the list of features and try to implement them in a clean fashion. More complex, and realistic microservices, like those automatically creating Tasks for all starred books you haven't read yet now become possible. This is the space where your microservice should be in.

Document the use of the microservice, and get us to publish it on our GitHub space.

# Notes

1. This constraint should be dropped in the distant future, but it is our default choice at the time. If you don't do anything special, this will be ok.
2. Docker makes it easy to set POSIX environment variables. These are the variables you'd be able to access in a shell ($PATH is a good example). These should be accessible so services can override your service.
3. Live-reload does not exist on all systems, but you can almost always fake it. Some languages need a restart (like nodejs and ruby), for some others you should connect to the service and reload the code through your editor (like Common Lisp with slime). This reloading does not need to auto-reload the dependencies, as these rarely change and thus warrant restarting the service.
4. Query parameters is the content that goes behind the question mark (?) in the URL. It constitutes of key-value pairs in which the keys needn't be unique.
5. This requirement comes forth from the work Jonathan Langens is performing on Sharing Knowledge (link should be added to basic ground work in the field).
6. The ONBUILD statement of Docker allows you to execute commands when someone extends your image. This means you can automatically copy the relevant contents of the repository in your /app and your /config. This further reduces the overhead of implementing a new template.
7. This construction makes it predictable how extensions of this template should behave. If someone builds a template on top of your template (inception-style), then it's easily understandable what needs to be called.
8. It may be that development mode can not be enabled using an environment variable as it requires too many extra dependencies. This is strongly advised, but it is an option. In that case, the development image should be maintained in the same repository as the template repository. The name of the development should be semtech/<template-name>-dev