# cogs

## Configuration Object Generation System

Brett Viren

June 30, 2020

# Some requirements for a configuration system

Configuration information:

- must be **valid**
  - well defined structure
  - constraints on values
  - valid-by-construction patterns
  - centralized validation methods
- is needed in **multiple contexts**
  - contract between producers and consumers
  - authoring, displaying, storing, serializing, native code types
- must support **varied and changing forms**
  - many types of applications and services
  - some common "base" for implementing "roles"
  - application- and instance-specific variety
  - evolution of structure and values over time

# Caveat ✿cogs✿-itate

- The initial development of ✿cogs✿ focuses on harder core problems and is not yet intended for "user/developers".
- Much still remains to be designed, implemented, integrated, etc.

# The ✿cogs✿ approach

### schema

Define formal **schema** to describe structure and constraints.

### codegen

Generate code to validate, produce, transport and consume configuration.

### correctness

Enable the pattern "single source of truth" (SSOT).

### automate

Minimize human effort and the chaos it brings.

# cogs definition of schema

a **schema** is a **data structure**
which may be **interpreted**
as **describing** the **structure of data**
( *including that of schema!* )

# Categories of schema interpretation

- `translate(schema)` $\rightarrow$ `schema`
- `codegen(schema, template)` $\rightarrow$ `code`
- `validate(schema, data)` $\rightarrow$ *true* | *false*

These functions are largely provided to ✿cogs✿ by the moo tool.

# Defining schema

✿cogs✿ supports authoring schema with functions of an **abstract base schema** in the Jsonnet data templating language[1].

```
function(schema) {
   types: [schema.string(pattern="^[a-zA-Z][a-zA-Z0-9_]*$")],
}
```

- When called, function defines an *application-level* schema consisting of a single **string type** taking a **valid value** that must match the given pattern.
- The `schema` object holds functions that return schema from a particular **schema domain** (eg, Avro, JSON Schema)
- App-level schema defined abstractly in terms these function calls.

[1] ✿cogs✿ (via moo) also supports defining schema in other languages (JSON, YAML, INI, XML or languages that generate these) but these lack support for the abstract base schema.

# Larger Schema Example

Describe the configuration for a “node” with “ports” and “components” from the ✿cogs✿ demo.

```
function(schema) {
  // ... other locals ...

  local node = schema.record("Node", fields=[
      schema.field("ident", ident,
              doc="Identify the node instance"),
      schema.field("portdefs", schema.sequence("Port"),
              doc="Define ports used by components"),
      schema.field("compdefs", schema.sequence("Comp"),
              doc="Describe components needing ports"),
  ], doc="A node configures ports and components"),

  types: [ ltype, link, port, comp, node ],
}
```

# Abstract base schema

```
function(schema) {
   types: [schema.string(pattern="^[a-zA-Z][a-zA-Z0-9_]*$")],
}
```

`schema` object is like OO "abstract base class" instance.

✿`cogs`✿ demo includes these concrete **domain schema**:

- `avro-schema.jsonnet` for codegen with Avro CPP and `moo` using serialization provided by `nlohmann::json`.
- `json-schema.jsonnet` for object validation via JSON Schema and `moo`.

Expected future work:

- New domains: Protobuf / Cap'N Proto, depending on RPC choices.
- Jsonnet functions for valid-by-construction configuration authoring.
- A totally different, simpler abstraction pattern (see backup slides).

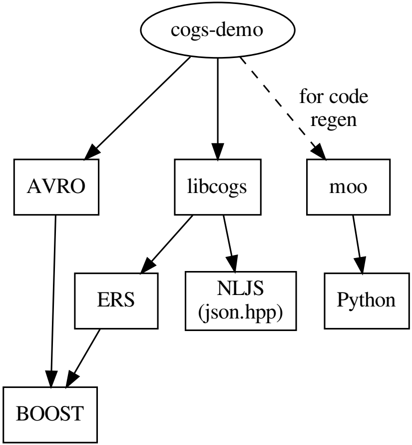
# moo

…provides a Python3 CLI and module for processing of schema defined in Jsonnet, JSON, XML, YAML, INI, etc, validation of objects in the same languages and template-based file generation using Jinja.

```
$ moo --help
Usage: moo [OPTIONS] COMMAND [ARGS]...

  moo command line interface

Options:
  --help  Show this message and exit.

Commands:
  compile      Compile a model to JSON
  imports      Emit a list of imports required by the model
  many         Render many files
  render       Render a template against a model.
  render-many  Render many files for a project.
  validate     Validate a model against a schema
```

moo essentially replaces a large set of other tools (`jsonnet`, `jq`, `j2`, `grep`, `awk`, etc) and the shell glue to connect them.

# ✿cogs✿ package dependency graph



# ✿cogs✿ package features

**configuration stream** methods for **deserialization** of configuration objects from multiple sources and formats.

**configurable base** an abstract base mixin class for user code to receive **dynamically or statically typed** configuration objects.

**tech opinions** ERS for exceptions, `nlohmann::json` for dynamic typed intermediate data representation.

**non-trival demo** Avro for C++ config `struct` types, component-based mocked framework and main application (link to doc).

Some current elements of demo are general and will be factored into ✿cogs✿ or `moo`.

Choice of Avro is for demo only, ✿cogs✿ does not require it.

# ✿cogs✿ configuration stream

A configuration is delivered as an ordered sequence (stream) of objects.

```
std::string uri = "....";
stream_p s = cogs::make_stream(uri);
cogs::object o = s->pop();
```

- The `make_stream()` factory returns steam based on parsing URI.
  - Stream will draw configuration bytes from resource at URI.
- The returned `unique_ptr<cogs::Stream>` is abstract.
- `cogs::object` is a `typedef` for `nlohmann::json` and provides a dynamic typed intermediate data representation layer.
- Exceptions defined by ERS may be thrown if stream is corrupt or an attempt is made to `pop()` past its end.

# ✿cogs✿ stream types

URIs with built-in support:

`file://config.json` a JSON array of configuration objects

`file://config.jstream` a JSON Stream of configuration objects

Potential future stream types URIs:

- Files via `https://` addressing.
- RPC server address (eg, hardwired host/port)
- ZeroMQ/ZIO port spec (eg, direct or auto-discovered address)
- Factory improvements for streams from shared lib / plugins.

# ✿cogs✿ delivery of configuration to consumer

A consumer may receive its configuration object by inheriting from a **virtual mixin** class and implementing the method:

## A **dynamic typed** interface

The user code must interpret a **dynamic object**.

```
struct ConfigurableBase {
    virtual void configure(cogs::object obj) = 0;
};
```

## A **static typed** interface

The user code receives C++ `struct`.

```
template<class CfgObj>
struct Configurable : virtual public ConfigurableBase {
    virtual void configure(CfgObj&& cfgobj) = 0;
};
```

In the ✿cogs✿ demo, the `struct` is generated from **schema** via Avro.

# ✿cogs✿ demo stream

The ✿cogs✿ demo stream assumes a pair-wise ordering:

| |
|---|
| component 1: `democfg::ConfigHeader` |
| component 1: corresponding config object |
| … |
| component N: `democfg::ConfigHeader` |
| component N: corresponding config object |

Each pair:

header identifies a component **implementation** and **instance** name

payload provides config object for the identified component

This *stream-level* contract is governed by schema in the ✿cogs✿ demo.
In general, it is up to the application to define.

# Demo stream model and schema

Building model with helper functions (not shown)

```
model: [
    head("demoSource", "mycomp_source1"),
    source(42),

    head("demoNode", "mynode_inst1"),
    node("mynode1",
         ports=[portdef("src",[
             link("bind","tcp://127.0.0.1:5678")])],
      comps=[compdef("mycomp_source1", "demoSource", ["src"])]
],
schema: [
    schema.head,
    schema.comp,
    schema.head,
    schema.node,
],
```

Details on `schema` array next.

# Demo stream schema (more)

JSON Schema requires types to be defined in a special location in the structure. The `compound()` function helps prepare that.

```
local jscm = import "json-schema.jsonnet";
local compound(types, top=null) = {
    ret : {
        definitions: {[t._name]:t for t in types}
    } + if std.type(top) == "null"
    then types[std.length(types)-1]
    else top,
}.ret;

local schema = {
    head: compound(head_schema(jscm).types),
    comp: compound(comp_schema(jscm).types),
    node: compound(node_schema(jscm).types),
};
```

**tl;dr:** understand *stream-level* schema then factor this complexity away from user view.

# Perform validation

The moo tool can dig data structure it is given a schema and model and perform validation on a single object (default) or on an array.

```
$ moo validate --sequence \
  -S schema -s demo/demo-config.jsonnet \
  -D model demo/demo-config.jsonnet
```

Currently returns `null` for success or a traceback into the model and schema data structures showing where validation failed.

# Some work still needed for DUNE FD DAQ

- Redesign the *abstract schema pattern* from using functions to using to meta-schema objects (details in backups)
- Move general parts from demo to `moo`.
- Integration with DUNE FD DAQ `appfmk` may include:
  - a "stream manager" hooking into appfmk factory
- A choice of RPC for larger CCM will influence replacement of Avro (eg with Protobuf, Cap'N Proto, etc).
- Understand if ✿cogs✿ and `moo` approach can help with connecting CCM RPC to `appfmk`.
- Understand larger configuration issues (authoring, version control, schema evolution, wholesale validation).

$\mathcal{FIN}$

# Problem with function-based schema definition

The functional programming with `schema.*` functions is rigorous and flexible but problematic:

- functional programming is not "normal" for many
- abstracting application-level patterns must carry the closure
- effectively forces use of Jsonnet to define schema
- does not exploit Jsonnet's simple and powerful inheritance idioms

# Post-demo revisions for meta-schema:

- define an *abstract schema vocabulary*
  - conceptually, a union of Avro, JSON Schema, Protobuf
- define a JSON Schema (in Jsonnet!) which will validate *user schema objects* against this abstract vocabulary
- develop *domain translation functions* which convert *user schema objects* to *domain schema objects*
  - eg: to an Avro schema object for generation and a JSON Schema object for validation

As a side-effect, user schema could be defined in JSON, XML, YAML, INI, as well as Jsonnet.