Wire-Cell Toolkit Configuration

Brett Viren

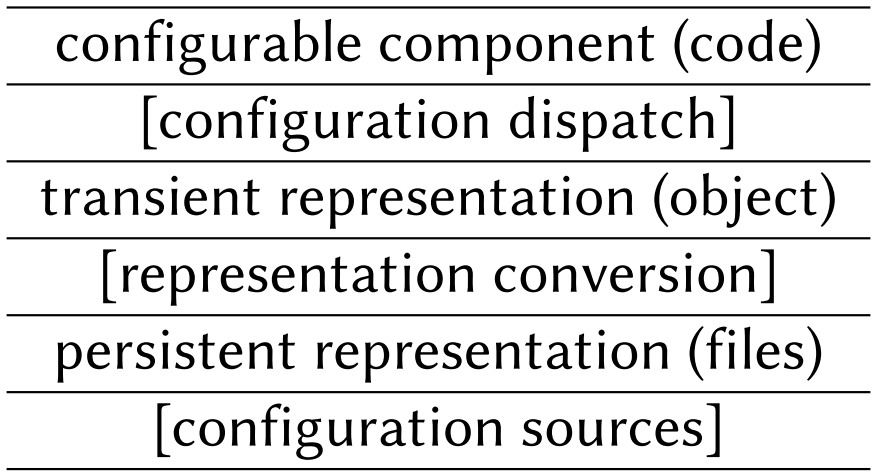
October 14, 2021

Outline

We will cover these aspects of Wire-Cell Toolkit (WCT) **configuration**:

- Layers
  - code
  - object
  - files
- Types of files
- Jsonnet primer
- Current configuration structure
- Developer tips
- Future plans

Layered Configuration Model



code C++ code expecting config objects of a certain form.

objects a "dynamic" data object representing the config info.

files source config info convertable to config objects.

C++ code layer of WCT config

in top-down order

WCT Main Configuration Sequence

`WireCell::Main` class expects an **array of objects** like:

```
[    // This is in Jsonnet language
    {type:"A", name:"a", data: {...}},
    {type:"B", name:"b", data: {...}},
    // ...
    {type:"Z", name:"z", data: {...}},
]
```

- The `Main` class is the entry point to WCT for `wire-cell` CLI and the *art* tool.
  - (though other applications are free to enter at other WCT layers)
- Main array must be ordered by "usage" dependency.
  - WCT provides a function to assure this.
- Each config object is associated with an instance of `IConfigurable`
  - C++ instance located using the (**type**, **name**) pair.
  - The **data** must follow a schema expected by the given **type**.

Brief detour: flavor of a configuration object

```
{
  type: "Drifter",  // Match C++'s WIRECELL_FACTORY()
  name: "drifter",  // name is optional, default is ""
  data: {           // some are optional if set in C++
      DL:           7.2 * wc.cm2/wc.s,
      DT:          12.0 * wc.cm2/wc.s,
      lifetime:       8 * wc.ms,
      drift_speed: 1.6 * wc.mm/wc.us,
      fluctuate:  true,
  }
}
```

This is some basic Jsonnet.

- Do not worry if it does not make sense yet.
- We will come to the syntax and how to know what to provide.

Continuing with C++ layer....

`WireCell::Persist` namespace

```
#include "WireCellUtil/Persist.h"
```

Low-level functions for:

- Locating configuration files.
  - Uses WCT's only env var: `WIRECELL_PATH`
- Loading file or string to product an object.
  - and vice-versa.
- Provides a C++ interface to C or GO Jsonnet libraries.
  - May pass Jsonnet "top level arguments" and "external variables".
- Component developers usually need not encounter this level.
  - A new type of "data" file will need a new component that does.

WCT IConfigurable

```
class MyFilter : public virtual IConfigurable {
  public:
    // IConfigurable
    virtual void configure(const WireCell::Configuration& cfg);
    virtual WireCell::Configuration default_configuration() const;
  private:
    int m_a{42};                    // a "sane" default
    float m_b{0};                   // always initialize
}
```

WCT will:

1. Call `default_configuration()` to get an initial object, treating it as opaque other than it is a JSON "object" (not scalar nor array).
2. Merge in to the default the contents of `data:{...}`.
3. Provide merged result back via `configure()`

Implementing IConfigurable

Best practice (now) is to "filter" class data members through configuration.

```
WireCell::Configuration MyFilter::default_configuration() {
    WireCell::Configuration cfg;
    cfg["a"] = m_a;
    cfg["b"] = m_b;
    return cfg;
}
```

```
void configure(const WireCell::Configuration& cfg) {
    m_a = get<int>(cfg, "a", m_a);
    m_b = get<float>(cfg, "b", m_b);
}
```

- Can do error checking and throw exceptions here, early before any processing occurs.
- Currently, we must read this kind of code to discover what to provide in `data:{...}`.
- Future plans will make all this boilerplate go away and give us configuration documentation!

`WireCell::Configuration` object

`WireCell::Configuration` namespace

```
#include "WireCellUtil/Configuration.h"
```

Mainly provides a thin wrapper over:

```
typedef Json::Value Configuration;
```

All `IConfigurable`'s use this object.

Example usage of `Json::Value`

JsonCPP is a "venerable" JSON library and provides Json::Value.

```
Configuration cfg;     // aka Json::Value
cfg["key"] = "value";
cfg["n"] = 42;
cfg["arr"] = {1,2,3};
int n = cfg["n"].asInt();
int oops = cfg["key"].asFloat();
```

Prefer helper functions over direct access

```
// Convert to C++ type
T convert<T>(const Configuration& cfg, const T& def=T());

// Get value via its attribute.dot.path into object
T get<T>(const Configuration& cfg, const std::string& dotpath,
         const T& def=T());
```

They add to `Json::Value` methods by:

- Encapsulating some error checking.
- Adding more flexible indexing.
- Adding a return-default-on-error idiom.
- Providing a more "modern" C++ typed interface.
- Somewhat shielding WCT component code from exact choice of `Configuration` type implementation.

Configuration files

Two “same but different” types of config files

cfg goes through `IConfigurable` and typically in hand-written `.jsonnet` files with official versions in `wire-cell-toolkit/cfg` repo.

data read more directly by “service” type components and typically are generated `.json.bz2` file produced by programs in `wire-cell-python` and with official copies in `wire-cell-data` repo.

These distinctions are not mandatory.

- We can pre-compile `.jsonnet` and compress to provide `.json.bz2` **cfg** type.
- Could, in principle, generate wire geometry **data** file as a Jsonnet program.

For the rest of the slides, we focus on **cfg** type written in Jsonnet.

Jsonnet Primer

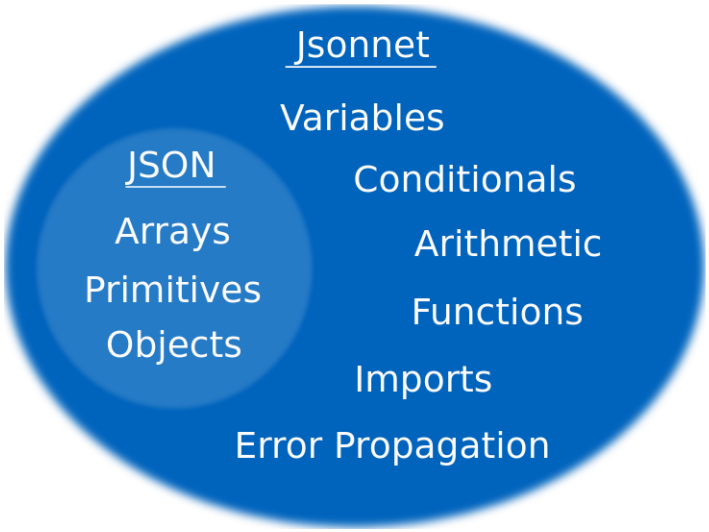
Jsonnet *data templating* language.

Pure functional programming language

- No side effects, lazy evaluation
- Full featured standard library
- "Lisp/Scheme with curly braces"

Compiles to JSON text by default

- or YAML, INI, Python (`dict`), XML(svg), TOML
- Novel formats can implemented in Jsonnet code.



Plus friendlier syntax: comments, optional trailing commas, optional quote-free object keys, single and double quotes.
Diagram from `https://jsonnet.org/`
See web site for excellent documentation.

Hello Jsonnet command line

Compile expressions as strings:

```
jsonnet -e '{hello:"world",} // quick CLI test'
```

```
{
   "hello": "world"
}
```

Or, provide Jsonnet in source files:

```
echo '2 + 2' > sum.jsonnet
jsonnet sum.jsonnet
```

```
4
```

WCT also provides `wcsonnet` which is `jsonnet` + WCT Jsonnet handling layers.

Jsonnet Types

Null only one value: `null`, equal only to itself.

Boolean two values: `true` and `false`.

Number 64 bit IEEE754 FP representation.

- exact integers $\in [-2^{53}, 2^{53}]$

String sequence of Unicode codepoints.

- may use `'single quotes'`, or `"double quotes"`.
- may index like `s[2]` but strings are not arrays.

Array Finite sequence, possibly heterogeneous in element types.

- `[1,2]`, `[true, "you", {can:, "nest"}]`

Object Collection of pairs: key of type string, value of any type.

- `{a:1, b:2, c:{greet: "hi!"}}`
- ordered by key `{b:2, a:1}` → `{"a":1, "b":2}`

Function A code context as a callable object.

- `{a::1, f::function(x=0) x+self.a, b:self.f()}` → `{"b":1}`

Some string operations

```
["%0.5f"%(1/19), "cat" < "dog", "cat"=="CAT", "catdog"[:3],
 std.asciiUpper("meow")]
```

```
[
   "0.05263",
   true,
   false,
   "cat",
   "MEOW"
]
```

- Case-sensitive values, Unicode support.
- Usual set of string functions in `std.` name space
  - `split() startsWith() substr() strReplace() format(), ...`
- Strings and arrays are distinct types.
  - A string is **not** an array of characters. There is no "char" type in Jsonnet.

Some array operations

Length, indexing, slicing, appending, concatenating.

- All similar to what one expects from Python.

Many more array functions under `std`.

- `join() find() map() filter() foldl() range() reverse() sort() uniq()`

```
local a = [1,2,3]; local b = [40,50,60];
{
    len: std.length(a),
    index: b[0],
    append: a + [4],
    concat: a + b,
    slice: self.concat[3:5],
}
```

```
{
   "append": [
      1,
      2,
      3,
      4
   ],
   "concat": [
      1,
      2,
      3,
      40,
      50,
      60
   ],
   "index": 40,
   "len": 3,
   "slice": [
      40,
      50
   ]
}
```

Objects - a set of key/value attributes

```
{ "b b":2, c_c:1, a:0 }
```

```
{
    "a": 0,
    "b b": 2,
    "c_c": 1
}
```

- Compiles to a JSON object, ordered by key.
- Attribute keys must be of type string.
  - Need quotes only if key does not look like a variable name.
- Attribute values can be of any Jsonnet type.
- Also, object defines a scope for *referencing* (more on this coming).

Object - : vs : :

```
{
    see_me : "yes",
    not_me :: "no",
}
```

```
{
   "see_me": "yes"
}
```

Jsonnet will use but not emit any attribute defined with `::`

- Mostly useful to assure we do not try to render a `function()` object to JSON.
- Can also be useful for "temporary" values that need to be shared outside of file scope but should not compile to final JSON.
  - Inside file scope, use of a `local` is cleaner.

Aside to understand *referencing* of values.

`local` values

Defines *file-local* value which may then be referenced directly by name.

```
local x = 42;                 // may now refer to 42 as x
local y = x + 1;              // use semicolon at top level
{                             // inside object scope
    local z = y - x,          // end with comma
    result: z                 // define an attribute
}
```

```
{
   "result": 1
}
```

Use = to define `local` values and `:` (or `::`) for object attributes.

Relative and self referencing

```
{   a: $.sub.c,    // reference from file-level top
    b: self.sub.c, // reference from current context
    sub: {
        c: self.d,
        d: "all the same",
        e: $.sub.d, f: $.a, } }
```

```
{
   "a": "all the same",
   "b": "all the same",
   "sub": {
      "c": "all the same",
      "d": "all the same",
      "e": "all the same",
      "f": "all the same"
   }
}
```

Lazy evaluation means reference can precede definition.

- $ represents the outer-most **object** (if top is another type, can't use $)
- `self` represents current lexical scoped object.

Infinite reference loops cause compile-time error (stack smash).

File referencing

```
{ // a.jsonnet (tutorial prefers a.libjsonnet)
    x = 1
}
```

```
// b.jsonnet
local myvar = import "a.jsonnet";
{
    y = myvar.x
}
```

- The `b.jsonnet` file compiles to `{"y":1}`
- The file name to `import` must be a literal string, not a computed value.
  - In part, this assures results are stable.
- Files for `import` are located via a search path given to the application.
  - `jsonnet -J /path/to/my/libjsonnet ...`
  - `wire-cell` and `wcsonnet` use `~-P` and honors `WIRECELL_PATH`

Lazy evaluation and no side effects

Lazy evaluation:

- Jsonnet will not evaluate any code that is not used for the final result.
- In previous example, the `a.jsonnet` may be a gigantic file but `b.jsonnet` used only `myvar.x` and so only `a.x` is evaluated.

No side effects:

- There is absolutely no way to modify any "variable"
  - (perhaps better to call Jsonnet "variables" as "named values")
- But we can make new values based on existing ones.

Back to objects

Object inheritance

The + operator on Object types is inheritance, or really **shallow merge**.

```
{a:1, b:2, c:{x:"hi", y:"there"}} + {b:20, c:30, d:40}
```

```
{
   "a": 1,
   "b": 20,
   "c": 30,
   "d": 40
}
```

- When attributes collide, those of the latter object "win".
- There is also std.mergePatch() which operates deeply.

(Of course, + is also concatenate for string and array and usual meaning for numbers.)

For merge, the + is optional and usually omitted

Look for two or more objects bumping against each other.

```
local one = {a:1, b:2, c:{x:"hi", y:"there"}};
// three object merge/inheritance
local three = one {b:20, c:30, d:40} {d:400};
three
```

```
{
   "a": 1,
   "b": 20,
   "c": 30,
   "d": 400
}
```

Array and Object Comprehensions

Similar to Python `list` and `dict` comprehensions:

```
local squares = [x*x for x in std.range(1,3)];
local sqrts = {["%d"%xx]:std.sqrt(xx) for xx in squares};
sqrts
```

```
{
   "1": 1,
   "4": 2,
   "9": 3
}
```

- Need to put the computed object attribute key inside `[]`'s.
- Keys need to be strings so we use string interpolation to build from number.
- **Notice** `std.range(Nmin, Nmax)` returns **inclusive range**‼
  - `std.range(1,3)` → `[1,2,3]`
  - This is **unlike** Python `range()` or C++ `std::iota()`.

`if/then/else` switch

```
local x = true;
local have = if x
             then "Yes"
             else "No";
have
```

```
"Yes"
```

- White space / indentation does not matter.
- The `else` is optional, `if/then/else if/...` chains allowed.
- Use sparingly as it tends to obscure structure/value of the results.
  - Prefer object "inheritance" to override a pre-built object.

Functions - file scope definitions

```
// Explicit creation of function type
local bye = function(name) "bye " + name;

// simpler, implicit syntax
local hi(name) = "hi " + name;

[hi("me"), bye("you")]
```

```
[
   "hi me",
   "bye you"
]
```

Functions - object scope definitions

```
{   // object-local scope
    local greeting(g) = g,

    // attr func calling local func
    greet:: function(who)
        greeting("bye") + " " + who,

    me: self.greet("me"),
    you: self.greet("you"),
}
```

```
{
   "me": "bye me",
   "you": "bye you"
}
```

- `.greet()` may be called on this object (eg, object used elsewhere).
  - The `::` keeps `greet()` from every rendering to JSON if object is in final result.
- `greeting()` can not be seen outside the object.

Function returning internal values

```
local f(x) = {
    local a = x*x,
    res: a + 2,
}.res;
f(42)
```

```
1766
```

- When you need a “working space” object to hold intermediate values and wish to return some portion.
- Simply dereference one object attribute immediately.

Function arguments

```
local f(x, y=2, z=0) = {
    local a = x*x,
    res: a + y + z,
}.res;
[f(42), f(42, 12), f(42, z=1, y=-42*42)]
```

```
[
   1766,
   1776,
   1
]
```

Required positional args and optional keyword args with defaults (like Python)

- Keyword arguments given in any order, but must follow positional.
- No Pythonic “exploding dict”, can not call like: `f(**d)`.

Function environment

```
local a=1;
{
    f:: function(x) a + self.b + x,
    b:: " + ",
    zz: self.f("2 = 3")
}.zz
```

```
"1 + 2 = 3"
```

- The function `f(x)` "captures" (in C++ terms) the `local` variables and the object context in which it was defined.

Functions returning functions

```
local a=1;
local f(x) = function(b) a+b+x;
local five = f(5);
five(2)
```

```
8
```

- Functions are *first class* instances.
- One can form *function closures* over values.

Top-level functions

A file can evaluate to a function which we call “top-level function”.

```
// f.jsonnet
function(x, y=1) { a:x+y }
```

```
// g.jsonnet
local f = import "f.jsonnet";
f(2) // -> { "a":3 }
```

As such, this is just a consequence of Jsonnet language.

TLAs: Top-level (function) Arguments

Application can inject external values through TLAs

```
// tla.jsonnet
function(x, y=2) {x:x, y:y}
```

```
jsonnet -A x=1 -A y="foo" tla.jsonnet
jsonnet --tla-code 'x=22' tla.jsonnet
```

```
{
   "x": "1",
   "y": "foo"
}
{
   "x": 22,
   "y": 2
}
```

- CLI processing makes values given by `-A`/`--tla-str` into strings.
- TLAs with default values need not be specified.
- The `--tla-code` can set a TLA to arbitrary Jsonnet code, (take care with shell quoting).
- There is also `--tla-code-file` to put this code in a secondary file.

The power of TLAs

Break large structures into many files.

- Compose larger structure through `import` and function calls.
- Test all structure scales via command line and TLAs.
- End-user can give highest-level settings on CLI with sane defaults.
- Bake user's favorite settings in yet higher-level Jsonnet.

TLAs in today's applications

WCT's CLI args are similar to what `jsonnet` takes:

```
$ wire-cell -A x=1, --tla-code y='{a:2}' [...] -c foo.jsonnet
$ wcsonnet  -A x=1, --tla-code y='{a:2}' [...] foo.jsonnet
```

- No `-c` with `wcsonnet` and not `-tla-code-file` for either.
- But, some **big caveats**:
  - Most of WCT config was structured before TLAs. FIXME!
  - `larwirecell` lacks TLA support. FIXME!

The old bad way to inject: `std.extVar()`

```
// extvar.jsonnet
{ x: std.extVar("x") }
```

```
jsonnet -V x=2 extvar.jsonnet
jsonnet --ext-code 'x=2' extvar.jsonnet
```

```
{
   "x": "2"
}
{
   "x": 2
}
```

- Same CLI args for `wire-cell` and this **is** supported by `larwirecell`
- But, there are major problems with `std.extVar()`
  - No way to provide a default value, the caller **must** provide **all** `extVar`'s
  - `std.extVar()` calls tend to be sprinkled everywhere, hard to spot in code.
  - Can not **set** them from Jsonnet, so full composition is not possible.

stdlib - full feature but "small" library of functions

`https://jsonnet.org/ref/stdlib.html`

- enumerate or test for object keys/values
- map, filter, reduce, sort, unique, set operations on arrays
- expected math, string and array functions
- type reflection: `std.type()` and `std.parseXXX()`, `std.manifestXXX()`

There is enough support that it was fairly easily to develop elaborate functions for, eg, vector arithmetic, graph construction, rewriting and topological sort!

INTERMISSION

WCT configuration structure

File layout

Official configuration file set is under wire-cell-toolkit/cfg/.

`wirecell.jsonnet` WCT system of units and basic helper functions

`vector.jsonnet` simple vector arithmetic

`pgraph.jsonnet` support for describing WCT data flow graphs

`pgrapher/` directory structure holding experiment configs

- `common/` holds generic "base" data structures
- `experiment/<name>/` holds per-experiment derived structures
- despite the name, valid for both `Pgrapher` and `TbbFlow`

Current experiments:

- `dune-vd`, `dune10kt-1x2x6`, `icarus`, `iceberg`, `pdps`, `sbnd`, `uboone`.

Caveat: the current `cfg/` area really needs a big refactor and cleanup!

Main steps to defining a WCT config:

1. Define individual component configuration objects "`cfg`"
   - *ie* `{type:"...", name:"...", data:{...}}`
2. If component is also a DFP node, embed it `cfg` in a `pnode` object.
3. Form the graph from `pnodes` culminating in a single aggregate `pnode`.
4. Define `Pgrapher` or `TbbFlow` component holding graph `edges`.
5. Optionally, define special `wire-cell` component to avoid having to specify a bunch of CLI args.
6. Emit the WCT "main sequence" array of `cfg` objects.

We now take each in trun.

1. Component `cfg` object - high level parameters

We may write each `cfg` object *by hand* like we saw with the `Drifter` example but that will be error prone as we must share the same values across many. Eg, the parameter...

```
drift_speed: 1.6 * wc.mm/wc.us,
```

...is needed in a few places. So, we set it once in a shared `params` data structure...

```
local wc = import "wirecell.jsonnet";
local params = import "pgrapher/common/params.jsonnet";
params.lar.drift_speed/(wc.mm/wc.us)
```

```
1.6000000000000001
```

...and we override that value per each experiment:

```
local wc = import "wirecell.jsonnet";
local params = import "pgrapher/experiment/uboone/params.jsonnet";
params.lar.drift_speed/(wc.mm/wc.us)
```

```
1.0980000000000001
```

1. Component `cfg` object - construction tools

To apply `params` itself consistently we make set of helper `tools` and from them make a set of *makers*:

```
local wc = import "wirecell.jsonnet";
local tool_maker = import "pgrapher/common/tools.jsonnet";
local params = import "pgrapher/experiment/uboone/params.
    jsonnet";
local tools = tool_maker(params); // uses TLAs!
local sim_maker = import "pgrapher/common/sim/nodes.jsonnet";
local sim = sim_maker(params, tools);
sim.drifter.uses[1].data.drift_speed/(wc.mm/wc.us)
```

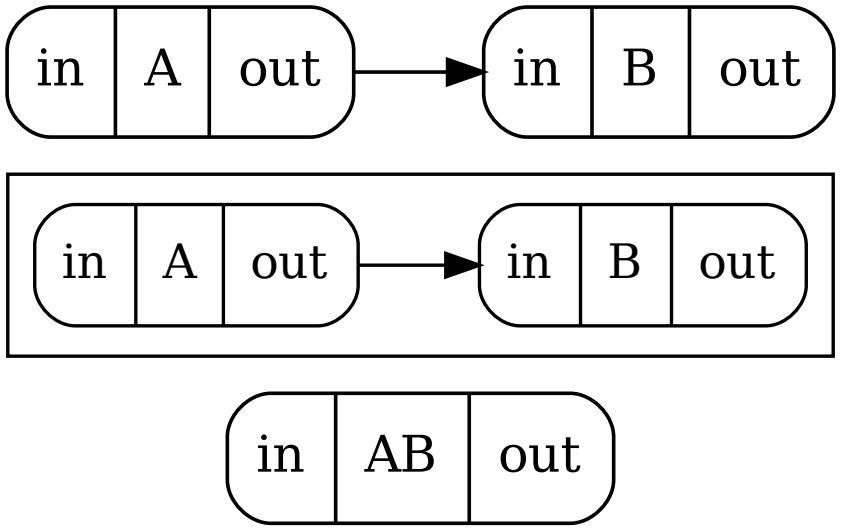
```
1.0980000000000001
```

- Do not worry about the form of this last line, we normally need not dig into what `sim.drifter` holds as it is already a `pnode`.
- Construction is nicely automated but using it requires some more complicated boilerplate than it really should.
- When diving into the guts of the "tools" and "makers" can be hard to understand what and how to change something.

Aside: ported graph model.

A WCT data flow graph is made of *nodes* with *ports*.

- A node has `nin` *input* and `nout` *output* ports, each zero or more.
- An *edge* is from an output port to an input port.
- A graph is *complete* when every port has exactly one edge.
- A *subgraph* of nodes may form an *aggregate* node.
- The aggregate node exposes any as-yet unconnected ports.



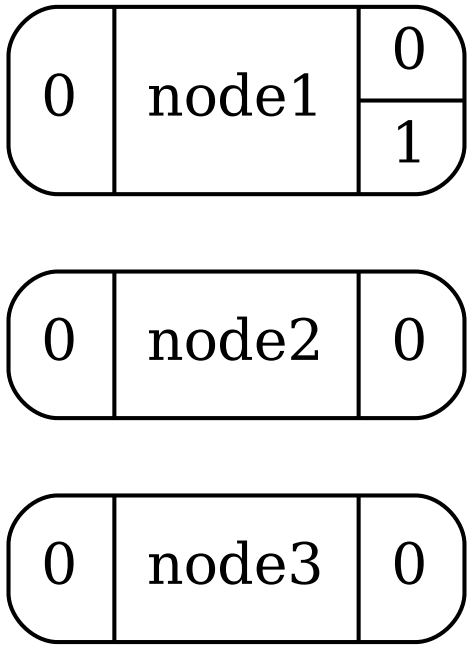
2. Form graph nodes - primitive nodes

```
local pg = import "pgraph.jsonnet";
local drifter = pg.pnode({type:"Drifter", name:"drifter"},
                         nin=1, nout=1, uses=[tools.random]);
```

- First arg to `pnode()` is a component `cfg` object.
- Must give `nin` and `nout` which is the input/output node *port cardinality*.
  - Used to catch graph construction errors.
- The `uses` argument names a dependency list to assure these components are represented in the final configuration sequence.

3. Form aggregates

Assume these nodes:



3. Form aggregate - `intern()`

```
local e1 = pg.edge(node1, node2);
local e2 = pg.edge(node2, node3);
local pl1 = pg.intern(
    innodes      = [node1],
    centernodes = [node2],
    outnodes     = [node3],
    edges        = [e1, e2]);
```

- `edge(n1, n2, p1=0, p2=0)` selects ports from nodes to connect.
- `intern()` is a heavy lifter: can connect up arbitray subgraph.
- Result aggregate is pipeline: `node1 -> node2 -> node3`
- Exposing remaining ports:

input	`node1` input port 0
output	`node1` output port 1 and `node3` output port 0

3. Form aggregate - `pipeline()`

Same result but written more simply:

```
local pl2 = pg.pipeline([node1, node2, node3]);
```

However, `pipeline()` will only hook up port 0's.

3. Form aggregate - etc

The `pgraph.jsonnet` also provides:

- "fans" 1-N `fanin()` and N-1 `fanout()` pattern
- "taps" a 1-2 fanout with port 1 "capped" with a sink
- "insert" may use `insert_one()/insert_node()` to "break" an edge to insert a new node

4. Define main “app”

```
  local graph = make_graph(); // top aggregate node
  local app = {
  type: 'Pgrapher',  // or "TbbFlow" for multithread
  data: {
    edges: g.edges(graph),
  },
};
```

- The edges are simply lists of pairs of node “type:name” and port numbers, all calculated in Jsonnet from the top aggregate node here called `graph`.
- The WCT `Pgrapher` or `TbbFlow` “app” will use `type:name` to lookup the C++ `INode` components and perform edge connections given their port numbers.

5. Special `wire-cell` component

```
local cmdline = {
    type: "wire-cell",
    data: {
        plugins: ["WireCellGen", "WireCellPgraph",
                  "WireCellSio", "WireCellSigProc",
                  "WireCellRoot"],
        apps: ["Pgrapher"]      // or "TbbFlow"
    }
};
```

- This special "component" configuration is purely to avoid giving CLI options.
  - It is used by `WireCell::Main` so useful for both `wire-cell` and `larwirecell`.
- Here or on CLI, one must give all plugins providing any referenced component "type".

6. The last line: configuration sequence

Finally, the actual Jsonnet "program" is almost trivial!

```
[cmdline] + g.uses(graph) + [app]
```

The `uses(graph)` does heavy lifting

- all `cfg` objects either held directly via `pnode` or via `pnode` attribute `uses`
- sorted by order of the "uses" dependency graph
- prepend `cmdline` and append `app` by hand.

Development tips

Editors

Emacs Use this `jsonnet-mode`
https://github.com/mgyucht/jsonnet-mode

VI Run `:JSONNETFMT` which I guess runs the `jsonnetfmt` program (sorry, I'm not a VI'er)

Others Likely provide some contributed support.

Generally, formatting support in an editor is not required to develop Jsonnet but it does help to provide hints that some syntax is wrong.

Workflow guidelines

- Start with small files.
- Test compile as you go.
- Keep files small, factor into more files, use `import` liberally.
- Use top level functions, provide TLAs with sane defaults.
- Avoid `if/then/else`.
- Avoid copy-paste. Refactor and then `import`.
- Avoid passing megaliths to functions in favor of just what is needed.

Current WCT config does not always provide the best role model!

Dump default configurations

```
$ wire-cell --help
$ wire-cell -a ConfigDumper -p WireCellApps -p WireCellImg
```

```
[
  {
    "data": {
      "spans": 1
    },
    "name": "",
    "type": "BlobClustering"
  },
...]
```

Tip: pipe through `jq -C | less -R`

- replace `WireCellImg` with any other WCT plugin to see its components

Understanding Jsonnet errors

```
RUNTIME ERROR: array bounds error: 0 not within [0, 0)
/home/bv/wrk/ls4gan/toyzero/wire-cell-toolkit/cfg/pgraph.jsonnet:19:15-30 object <b>
std.jsonnet:1338:50-54 thunk <b>
std.jsonnet:1338:42-54 function <anonymous>
std.jsonnet:1338:42-54 function <anonymous>
std.jsonnet:1342:11-23 function <anonymous>
std.jsonnet:231:30-65
std.jsonnet:231:19-66 function <anonymous>
/home/bv/wrk/ls4gan/toyzero/wire-cell-toolkit/cfg/wirecell.jsonnet:325:30-44 thunk <a>
/home/bv/wrk/ls4gan/toyzero/wire-cell-toolkit/cfg/wirecell.jsonnet:325:30-49 function <anony
/home/bv/wrk/ls4gan/toyzero/wire-cell-toolkit/cfg/wirecell.jsonnet:325:30-49 function <func>
std.jsonnet:789:24-47 thunk <running>
std.jsonnet:789:9-57 function <aux>
std.jsonnet:789:9-57 function <aux>
std.jsonnet:790:5-28 function <anonymous>
/home/bv/wrk/ls4gan/toyzero/wire-cell-toolkit/cfg/wirecell.jsonnet:326:22-55 function <anony
/home/bv/wrk/ls4gan/toyzero/wire-cell-toolkit/cfg/pgraph.jsonnet:177:21-48 function <anonymo
cfg/wcls-sim-adc.jsonnet:137:12-27 object <anonymous>
cfg/wcls-sim-adc.jsonnet:(136:9)-(138:4) object <app>
cfg/wcls-sim-adc.jsonnet:141:19-22 thunk <array_element>
During manifestation
```

- A little cryptic but one gets used to reading these stack traces.
- Error message is first line, then stack trace.
- First line of stack trace is the direct error site.
- Last line is the main-file entry point.
- A `thunk` is simply a non-function line of code.
- Numbers: `line:col1-col2`, useful to check source code.

Debugging - `std.trace(msg, res)`

`msg` string printed to `stderr` and `res` is returned

```
// extrace1.jsonnet
local x = true;
{ a: if x then std.trace('x is true', 42)
     else std.trace('x is false', -1) }
```

```
jsonnet extrace1.jsonnet 2>&1 1>/dev/null
```

```
TRACE: extrace1.jsonnet:3 x is true
```

```
// extrace2.jsonnet
local o = {a:1, b:"Bee"};
std.trace(std.toString(o), o)
```

```
jsonnet extrace2.jsonnet 2>&1 1>/dev/null
```

```
TRACE: extrace2.jsonnet:2 {"a": 1, "b": "Bee"}
```

Debugging - temporarily produce intermediate value

We want to build `three` but get an error, comment out and temporarily build intermediates:

```
local one = [1];
local two = one[1];
local three = two*2;
// three // hmm, leads to stack trace.
// two   // okay, try this.  Nope also crashes.
one      // works and, ah ha, problem obious now !
```

We see we should have used instead

```
local two = one[0];
```

Assert truth

One may assert the truth of something which “should” always be so.

```
// exassert.jsonnet
local f(x) =
    assert x >= 0 : 'x must be positive';
    std.sqrt(x);
f(-1)
```

```
jsonnet exassert.jsonnet      2>&1 || true
```

```
RUNTIME ERROR: x must be positive
exassert.jsonnet:(3:5)-(4:16) function <f>
exassert.jsonnet:5:1-6
```

There is no way to “catch” an `assert`, this is not a exception mechanism.

Raise error

```
jsonnet -e 'local a = [error "no zero element", 1, 2]; a[1]'
```

```
1
```

```
jsonnet -e 'local a = [error "no zero element", 1, 2]; a[0]'
    2>&1 || true
```

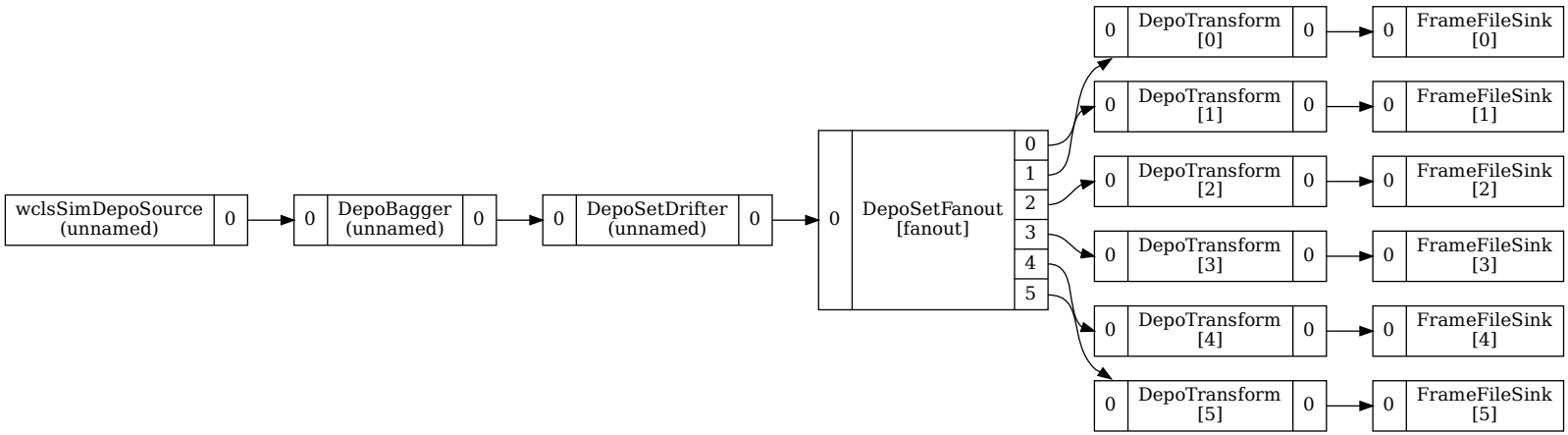
```
RUNTIME ERROR: no zero element
<cmdline>:1:12-35 thunk <array_element>
<cmdline>:1:44-48
```

Like C's `abort()`. Note, effect of lazy evaluation. Again, not an exception to be caught.

Visualize the configured graph

Install `wire-cell-python` and:

```
$ wirecell-pgraph dotify --no-params --jpath=-1 \
      mycfg.jsonnet mycfg.dot
$ dot -Tpdf -o mycfg.pdf mycfg.dot
```



- Finds connection errors immediately and helps understand the overall structure.
- Remove `--no-params` to see the `cfg` object attributes.

The `--jpath=-1` tells the program how to find the graph data structure in the overall `.jsonnet/.json` file.

jq for "querying" the configuration

```
$ cd wire-cell-toolkit/cfg/

$ jsonnet -J . pgrapher/experiment/pdsp/wct-sim-check.jsonnet |
    wc -l
162837

$ jsonnet -J . pgrapher/experiment/pdsp/wct-sim-check.jsonnet |
    jq '.type' | wc -l
138

$ jsonnet -J . pgrapher/experiment/pdsp/wct-sim-check.jsonnet |
    jq '.[0]'
# -> prints first cfg object

$ jsonnet -J . pgrapher/experiment/pdsp/wct-sim-check.jsonnet |
   jq '.[] | select(.type=="Pgrapher").data.edges[0]'
 # -> prints first edge object of Pgrapher cfg
```

- jq is like grep/sed/awk for JSON.
- Very powerful, has a learning curve but worth learning a few basics.

Plans for WCT configuration improvement

The main problems

- Every C++ component author repeats config related boilerplate.
- Each forced to invent novel config object setting/getting patterns.
- Generally not done safely and not with uniform exception patterns.
- An implicit object schema lurks in each component, diffused throughout the C++ source.
- Even with `ConfigDumper` can only discovery part of the schema where defaults were given.
- No comprehensive reference documentation of configuration objects.

Strategy

- Define *schema* describing expectation of type of configurable object.
- Generate C++ code from *schema* to provide
  - C++ `struct` reflecting the schema.
  - Bidirectional serialization between C++ `struct` and JSON object/file.
- Lift the `configure()` and `default_configure()` methods of all `IConfigurable` implementations into an `Aux::Configurable` base.
  - It is templated on `<struct>`, holds as `protected` data member.
  - `IConfigurable` implementation then becomes automatic.
  - Still provide a "hook" method for code wanting to run at configure time.
- C++ component then gets ready-to-use and safe access to config in explicit, static `struct` type.

Implementing schema-based configuration

For DUNE DAQ I made moo which provides all we need (and more).

- In the cfgschema branch of my fork of wire-cell-toolkit has the start.
- Work largely consists of, for each component, define a schema, rip out existing usage, replace with `struct`

Goal: keep external configuration expectations 100% unchanged.

A wrinkle: JsonCPP → `nlohmann::json`

- moo really wants to use "JSON for Modern C++" (nlohmann::json).
- WCT uses JsonCPP and its API is not friendly to moo
  - I've tried/failed to make it work already.
- JsonCPP as become not so great for other reasons
  - old fashioned C++ idioms, neglected development, hard to find docs (at times).
- Replacement would remove entire JsonCPP shared library dependency in exchange for just a single C++ header file.
- This swap will touch a lot of code. Easy, but very tedious work. Still, I want to do this.

Simplifying/refactoring `wire-cell-toolkit/cfg/`

- Leverage *schema* to generate Jsonnet code.
  - Provide `cfg` object "constructor" functions.
  - Use them instead of directly making `cfg` objects.
  - Jsonnet then forces valid schema.
  - Much of this is done already in `moo`.
- Develop "bottom-up" functions to build larger structures from these "constructor functions".
  - Eg, one general function to create "a sim pipeline"
  - We have that now but not well factored.
- Develop "top-down" functions to meet end-user patterns.
  - Eg, one general function to create "a sim job".
  - Any top-level file should be only a few unique lines long, not 100s of copy-pasted lines typical now!
- Strongly exploit composition via TLAs.
  - Need to add TLA support to `larwirecell`.
    - ★ Can have a `std.ExtVar()` to TLA layer in Jsonnet.

FIN