TsLuaConfig

Oath:

Introduction

Alan M. Carroll (Committer, Oath)

Syeda “Persia” Aziz (Committer, Oath)

Configuration Support using Lua

It is current policy that all new configuration should be done using Lua as a base. This work is an attempt to provide a generalized API for using Lua configurations.

Design Goals

- Avoid need to write Lua code or use Lua C API calls.
  - No need to understand Lua internals
- Easy access to configuration data after loading.
- Provide as much diagnostic feedback as possible during configuration processing.

Background

This work was originally started to reuse the TsConfig front end with a Lua backend in order to make using Lua for configuration not require learning Lua and Lua internals.

Additional requirements from prototype use lead to the current design which takes a very different approach to access configuration data. This was driven primarily to provide additional features to make use by both developers and administrators easier.

Design and Theory

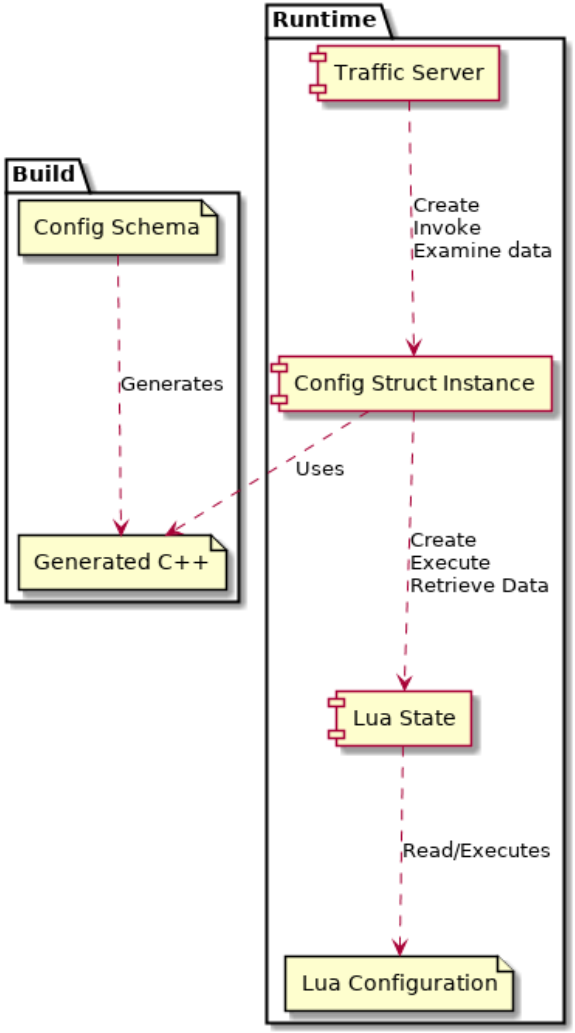
Schema Based Configuration

TsLuaConfig files are defined by a *schema*. This specifies the structure of the input data.

The schema is used to generate C++ and Lua code which implements loading and verifying the configuration file contents.

The developer writes the schema, builds the code, and at run time instantiates a class and tells it to load the file. The configuration data is placed in this class for later access.

Schema Use


Runtime
Traffic Server
Create
Invoke
Examine data
Build
Config Schema
Generates
Config Struct Instance
Uses
Generated C++
Create
Execute
Retrieve Data
Lua State
Read/Executes
Lua Configuration

Rationalization 1

The current configuration support for `records.config` can be considered to be a very primitive schema. It has types and a little support for input validation. TsLuaConfig is simply a much richer and powerful upgrade.

This should also make expanding and tweaking Lua configuration processing much easier.

Rationalization 2

A key design goal for the schema is to provide much clearer feedback for administrators. The point is to not just detect errors but provide clear *corrective* feedback.

“‘Enable Debug’ should be an integer, not a string.”

“Unrecognized tag ‘Leif’ – the valid options are ‘amc’, ‘Persia’, or ‘Phil’.”.

Doing this well is a lot of work, but can be done more easily with schema based automation.

Implementation and Usage

Schema Background

Although the schema design is Traffic Server specific, it is essentially a rip off of JSON schema design from `json-schema.org`.

The primary difference is the schema is in Lua, not JSON, because I think it is overall better to use one language (Lua) rather than adding yet another (JSON) to the mix.

JSON schema features that didn’t seem useful are not included.

Code Generation

The schema is used to generate C++ code.

- A set of *configuration classes* which load and contain the configuration data from the file. The structure follows that of the schema. The outermost configuration class is called the *main* configuration class.
- Static data classes that encode schema data that is the same every instance of the configuration class.

Dynamic vs. static

The configuration class constructors are designed to be light weight. They only

- Initialize the configuration data member to a default, if any.
- Load a pointer to the static schema data.

This is the reason the static data is split in to separate classes.

Usage

Usage is simple. The main configuration class is instantiated and the load method called on the configuration file. Valid data is loaded in to the class instance and an error report returned. Valid data is accessed directly from storage in the configuration classes.

Schema Details

Supported types are

- Integer
- Number
- String
- Enumeration
- Object
- Array

Lua Types

Strings, Integers, and Numbers are the obvious mapping. An Object is a table with explicit keys. An Array is a table with integer keys and stored as a `std::vector`.

To avoid confusion it is not permitted to have a table with both integer (Array) and non-integer (Object) keys.

Enumerations and Objects require special handling.

Objects

Objects are modeled as configuration classes. The configuration class has a member for each member of the Lua object. The class nesting of the configuration class exactly follows the nesting of objects in the schema.

Enumerations

An Enumeration is stored internally as a table mapping strings to integers. The Lua configuration value of an Enumeration can be either a string or an integer. It is converted to an integer when loaded from Lua. Validation is done to verify the Lua value is a key or value in the enumeration table.

The enumeration table is stored in the static data of configuration class twice, once by key and once by value.

State of Work

Persia has updated her SNI based configuration project to use the equivalent of the generated C++ to validate the code works.

The schema is mostly designed and work is being done on an official schema.

Code generation work has not yet been done.

Example

Active Work

Persia’s SNI based configuration pull request

https://github.com/persiaAziz/trafficserver/pull/6

contains a working example of Lua config, written by hand instead of generated.

Example: Resolver Configuration

```
name_servers={
  round_robin={
    count=100,
    style="STRICT"
  },
  ns={"ns-1.oath.com","ns-2.oath.com"}
}
```

```
{
  ["$schema"]="http://trafficserver.apache.org/schema/dns_resolver",
  lua_global="name_servers",
  c_name="ResolverConfig",
  type="object",
  description="Nameserver resolver configuration.",
  properties={
    round_robin={
      type="object",
      properties={
        style={
          type="enum",
          kv={"STRICT":0,"TIMED":1}
        },
        time={
          type="integer",
          description="Time interval for a single nameserver before shifting to the next."
        },
        count={
          type="integer",
          description="Number of queries for a single namserver before shifting to the next."
        }
      }
    },
    ns={
      type="array",
      description="List of nameservers",
      items={
        type="string",
        description="FQDN or IP address of nameserver."
      }
    }
  }
}
```

Generated Code Outline Version

```
struct ResolverConfig : public TSConfigBase {
  struct Round_Robin_Config : public TSConfigBase {
    int style;
    int time;
    int count;
  } round_robin;
  std::vector<std::string> ns;
  ts::Errata load(ts::string_view path); /// External load method.
  ts::Errata loader(LuaState *); /// Internal load method.
};
```

Generated Code
Meta data

```
struct ResolverConfig : public TSConfigBase {
  struct Round_Robin_Config : TSConfigBase {
    Round_Robin_Config() : _meta_style(&_META_style) {}

    int style;

    static TsConfigEnumDescriptor _META_style;
    TSConfigEnum<Round_Robin_Config> _meta_style;
    /// ...
  } round_robin;
  /// ...
};
```

Use

```
ResolverConfig ns_config;
ts::Errata zret = ns_config.load(“resolver.lua”);
printf(“The round robin type is %d\n”,
        ns_config.round_robin.style);
```

Issues

Easy to describe issues

- There does not seem to be a way to get line numbers in validation messages. It is hoped the availability of static schema data will provide sufficient context to provide useful error reports.
- Enumeration support requires a non-trivial deviance from the JSON schema, as its enumeration support is just a list, not a mapping of keys ↔ values.

Enumerations and Lua Constants

For enumerations it might be preferable to have Lua constants with the values rather than literal strings. This makes error detection better as Lua does the checks instead of the generated code.

The exact naming and style of these is a bit more difficult to decide. My preference would be schema data that specifies the global name in which to put a table that maps names to integers.

Prepopulation and assignment

Direct assignment doesn’t work because the nested tables do not exist. These could be prepopulated easily based on the schema but that will prevent source tracking. This may be doable with some metatable cleverness. From the example

```
name_servers.round_robin.count=100;
name_servers.round_robin.style="STRICT";
name_servers.ns[0]="ns-1.oath.com";
name_servers.ns[1]="ns-2.oath.com";
```

Lua vs. traffic_ctl

Give a Lua based configuration, how can specific values in that configuration be updated from the command line?

Either this will need to be restricted to leaf, primitive values or Lua processing will be needed inside traffic_ctl. It seems requiring a full configuration reload will not be feasible in general.

Future Work

Documentation Generation

I would like the schema to become rich enough that reference documentation can be generated directly from it. This would put all of the boiler plate information about configuration in a single place, thereby ensuring consistency between documentation and code.

This will likely require adding description fields not in the JSON schema and not required for actual operation.

Variants

The original design (from TsConfig) was based on the direct mapping of Lua variables to C++ variants. This is not needed with the current schema as the exact type of all values is specified.

It may be desirable to allow explicitly variant types in the Schema (as is allows by the JSON schema). The utility of this is unclear however.

The only use case I have is the promotion of singletons of type T to arrays of T if the schema specifies an array.

Lua assignment support

Although this design is data driven, it would be easy to add additional Lua code to support a more function oriented style (as with the logging configuration).

Frequenly Asked Questions

What happens if an administrator alters the schema file?

Nothing. The schema files are used only during the build phase. After that all of the schema information is embedded the generated C++ code. Changes to a schema only take effect after rebuilding Traffic Server.

Isn’t this a lot more work than records.config?

Yes, but with a much bigger payoff of being able to use Lua with type safety, input validation, useful error message and hopefully generated documentation.