Introduction to the HCL library

Or the HashiCorp configuration language (language)

Adrien Delorme
January 2022

Some disclaimers

- HCL is currently at version 2 and stable.
- I did not create HCL and I only slightly contribute to HCL.
- I made Packer transition to HCL.

I learned HCL along the way.

By looking at other Projects like Terraform or Nomad.

I don't recommend doing that.

And I really hope this presentation can help you get started.

HCL is not a data structure serialization language

Like JSON or YAML

HCL allows to define configuration languages for tools

So it is better to say :

- The Terraform Configuration language
- The Nomad Configuration language
- The Packer Configuration language
- etc.

Applications intending to use HCL should think of it as a language design task in its own.

I'm going to show how to do this.

First some basics

Some vocabulary

HCL files are made of :

- bodies :

```
// this is an empty body
```

- blocks :

```
my "block" {
    // this is the body of a block
}
```

- arguments :

```
my_string = "value"
my_number = 42
my_array  = ["foo", "bar", "baz"]
```

Example file

```
my_tool_version = "1.2.3"

block {
    input = "value"

    nested_block "name" {
        other_input = "value"
    }

    nested_block "second" {
    }
}

block {
    // ...
}
```

One way to parse an HCL file

```
file, diags := hclsyntax.ParseConfig(bytes, filename, hcl.Pos{Byte: 0, Line: 1, Column: 1})
```

- diagnostics can contain contextual warning or errors

HCL's super powers :

- Partial reads (get block a)
- Variables
- Functions

Defining the Cooking configuration Language

It displays recipes nicely.

I want to be able describe how to:

- `slice` ingredients
- `boil` ingredients.
- stash ingredients together.

And then maybe print the recipe.

Slicing:

```
slice "cheese" {
} // It's totally the reblochon kind 🧀
```

This is going to create a sliced_cheese variable.

Slicing:

```
slice "cheese" {
} // It's totally the reblochon kind 🧀
```

```
var preparationActionsSchema = &hcl.BodySchema{
    Blocks: []hcl.BlockHeaderSchema{
        {Type: "slice", LabelNames: []string{"ingredient"}},
```

BodySchema tells what blocks can be present in a body

Slicing:

```
slice "cheese" {
} // It's totally the reblochon kind 🧀
```

```
var preparationActionsSchema = &hcl.BodySchema{
    Blocks: []hcl.BlockHeaderSchema{
        {Type: "slice", LabelNames: []string{"ingredient"}},
```

```
    preparationContent, prepRest, diags := file.Body.PartialContent(preparationActionsSchema)
```

- PartialContent return us 'just what we want' from our schema.
- The rest of the AST is in 'prepRest'.

Slicing:

```
slice "cheese" {
} // It's totally the reblochon kind 🧀
```

```
var preparationActionsSchema = &hcl.BodySchema{
    Blocks: []hcl.BlockHeaderSchema{
        {Type: "slice", LabelNames: []string{"ingredient"}},
```

```
    preparationContent, prepRest, diags := file.Body.PartialContent(preparationActionsSchema)
```

```
    for _, block := range preparationContent.Blocks {
        action := Action{
            Verb: block.Type,
            What: block.Labels[0],
        }
        switch block.Type {
        case "slice":
            // nothing to do here since this is empty
```

Nice, we can parse "slice". Now to boil.

Boiling:

```
boil "potatoes" { // 🥔 🥔 🥔
  duration = minutes(30)
} // no need to peel
```

This is going to create a boiled_potatoes variable.

Boiling:

```
boil "potatoes" { // 🥔 🥔 🥔
  duration = minutes(30)
} // no need to peel
```

```
var preparationActionsSchema = &hcl.BodySchema{
    Blocks: []hcl.BlockHeaderSchema{
        {Type: "slice", LabelNames: []string{"ingredient"}},
        {Type: "boil", LabelNames: []string{"ingredient"}},
```

Boiling:

```
boil "potatoes" { // 🥔 🥔 🥔
  duration = minutes(30)
} // no need to peel
```

```
var preparationActionsSchema = &hcl.BodySchema{
    Blocks: []hcl.BlockHeaderSchema{
        {Type: "slice", LabelNames: []string{"ingredient"}},
        {Type: "boil", LabelNames: []string{"ingredient"}},
```

```
type Action struct {
    Duration string `hcl:"duration"`
```

Boiling:

```
    for _, block := range preparationContent.Blocks {
```

```
        switch block.Type {
        case "slice":
            // nothing to do here since this is empty
        case "boil":
            boilBody := block.Body
            diags := gohcl.DecodeBody(boilBody, durationEvalCtx, &action)
```

```
type Action struct {
    Duration string `hcl:"duration"`
```

Here:

- **`duration`** will be loaded into the duration field because of the struct tag.
- **diags** — returned by almost all HCL calls — is a slice of diagnostics with a context, they could be warnings or errors (ex: "file.go:L32 something is not right")
- **`gohcl`** deduces HCL schemas using hcl tags.

Okay, now we need a way to stack these ingredients

Stacking stuff :

```
stack "tartiflette" {
  in   = "cast iron pan"

  add {
    what     = boiled_potatoes
    quantity = "500G"
  }

  add {
    what     = sliced_cheese
    quantity = "400G" // just enough :)
  }

  // I don't have any onions ¯\_(ツ)_/¯
```

Stacking stuff :

```
stack "tartiflette" {
  in   = "cast iron pan"
```

```
    var stack struct {
        What string
        In   string   `hcl:"in,optional"`
        Rest hcl.Body `hcl:",remain"`
    }
```

```
            diags := gohcl.DecodeBody(stackBody, nil, &stack)
```

Here, we partially decode our stack block, and put everything unexpected in stack.Rest.

'add'

```
  add {
    what     = boiled_potatoes
    quantity = "500G"
  }
```

```
    nestedStackContent, moreDiags := stack.Rest.Content(stackContentSchema)
```

```
    for _, block := range nestedStackContent.Blocks {
```

'add' schema

```
        case "add":
            var stackSpec = &hcldec.ObjectSpec{
                "what":     &hcldec.AttrSpec{Name: "what", Type: cty.String, Required: true},
                "quantity": &hcldec.AttrSpec{Name: "quantity", Type: cty.String},
            }
```

```
            v, diags := hcldec.Decode(block.Body, stackSpec, nil)
```

- Schemas allow to define the layout of an object more explicitly.
- These can be sent over the network.
- All of these hcldec types define the hcldec.Spec interface

Now, we know:

- how to open an HCL file
- how to extract blocks and attributes from a body

Now let's learn how to use variables and function.

EvalContext allows to define variables and functions.

```
  duration = minutes(30)
```

```
var durationEvalCtx = &hcl.EvalContext{
    Functions: map[string]function.Function{
        "minutes": HCLMinutesFunc,
    },
    Variables: map[string]cty.Value{
        "my_var": cty.StringVal("my_val"),
    },
}
```

```
            diags := gohcl.DecodeBody(boilBody, durationEvalCtx, &action)
```

To list variable references of a block

```
add {
  what     = boiled_potatoes
  quantity = "500G"
}
```

To list variable references of a block

```
add {
  what     = boiled_potatoes
  quantity = "500G"
}
```

```
            // tell what is required by this block :
            traversals := hcldec.Variables(block.Body, stackSpec)
            for _, traversal := range traversals {
                split := traversal.SimpleSplit()
                stackAddsRequire = append(stackAddsRequire, split.RootName())
            }
```

Pro tips

- Write mock HCL early to get a feeling.
- Decouple decoding from validation from execution.
- Define wether you will need a tree early.
- gohcl from hcl/v2 seems to be the most imported

	gohcl (v1)	v2/hcldec	v2/gohcl
Boundary	✓		
Consul	✓		
Nomad	✓	✓	✓
Packer		✓	✓
Terraform		✓	✓
Vault	✓		
Waypoint			✓

* Nomad's old versions uses hcl v1

Thank you

Adrien Delorme
January 2022
adrien@hashicorp.com (mailto:adrien@hashicorp.com)

https://github.com/azr/hcl_intro (https://github.com/azr/hcl_intro)