# HCL Documentation

***Release guide***

**HashiCorp**

**Aug 28, 2018**

# Contents:

HCL is a toolkit for creating structured configuration languages that are both human- and machine-friendly, for use with command-line tools, servers, etc.

HCL has both a native syntax, intended to be pleasant to read and write for humans, and a JSON-based variant that is easier for machines to generate and parse. The native syntax is inspired by libucl, nginx configuration, and others.

It includes an expression syntax that allows basic inline computation and, with support from the calling application, use of variables and functions for more dynamic configuration languages.

HCL provides a set of constructs that can be used by a calling application to construct a configuration language. The application defines which argument names and nested block types are expected, and HCL parses the configuration file, verifies that it conforms to the expected structure, and returns high-level objects that the application can use for further processing.

At present, HCL is primarily intended for use in applications written in Go, via its library API.

# CHAPTER 1

# Introduction to HCL

HCL-based configuration is built from two main constructs: arguments and blocks. The following is an example of a configuration language for a hypothetical application:

```
io_mode = "async"

service "http" "web_proxy" {
  listen_addr = "127.0.0.1:8080"

  process "main" {
    command = ["/usr/local/bin/awesome-app", "server"]
  }

  process "mgmt" {
    command = ["/usr/local/bin/awesome-app", "mgmt"]
  }
}
```

In the above example, `io_mode` is a top-level argument, while `service` introduces a block. Within the body of a block, further arguments and nested blocks are allowed. A block type may also expect a number of *labels*, which are the quoted names following the `service` keyword in the above example.

The specific keywords `io_mode`, `service`, `process`, etc here are application-defined. HCL provides the general block structure syntax, and can validate and decode configuration based on the application's provided schema.

HCL is a structured configuration language rather than a data structure serialization language. This means that unlike languages such as JSON, YAML, or TOML, HCL is always decoded using an application-defined schema.

However, HCL does have a JSON-based alternative syntax, which allows the same structure above to be generated using a standard JSON serializer when users wish to generate configuration programmatically rather than hand-write it:

```
{
  "io_mode": "async",
  "service": {
    "http": {
```

(continues on next page)

(continued from previous page)

```
      "web_proxy": {
        "listen_addr": "127.0.0.1:8080",
        "process": {
          "main": {
            "command": ["/usr/local/bin/awesome-app", "server"]
          },
          "mgmt": {
            "command": ["/usr/local/bin/awesome-app", "mgmt"]
          },
        }
      }
    }
  }
}
```

The calling application can choose which syntaxes to support. JSON syntax may not be important or desirable for certain applications, but it is available for applications that need it. The schema provided by the calling application allows JSON input to be properly decoded even though JSON syntax is ambiguous in various ways, such as whether a JSON object is representing a nested block or an object expression.

The collection of arguments and blocks at a particular nesting level is called a *body*. A file always has a root body containing the top-level elements, and each block also has its own body representing the elements within it.

The term "attribute" can also be used to refer to what we've called an "argument" so far. The term "attribute" is also used for the fields of an object value in argument expressions, and so "argument" is used to refer specifically to the type of attribute that appears directly within a body.

The above examples show the general "texture" of HCL-based configuration. The full details of the syntax are covered in the language specifications.

---

**Todo:** Once the language specification documents have settled into a final location, link them from above.

---

## 1.1 Argument Expressions

The value of an argument can be a literal value shown above, or it may be an expression to allow arithmetic, deriving one value from another, etc.

```
listen_addr = env.LISTEN_ADDR
```

Built-in arithmetic and comparison operators are automatically available in all HCL-based configuration languages. A calling application may optionally provide variables that users can reference, like `env` in the above example, and custom functions to transform values in application-specific ways.

Full details of the expression syntax are in the HCL native syntax specification. Since JSON does not have an expression syntax, JSON-based configuration files use the native syntax expression language embedded inside JSON strings.

---

**Todo:** Once the language specification documents have settled into a final location, link to the native syntax specification from above.

---

CHAPTER 2

# Using HCL in a Go application

HCL is itself written in Go and currently it is primarily intended for use as a library within other Go programs.

This section describes a number of different ways HCL can be used to define and process a configuration language within a Go program. For simple situations, HCL can decode directly into Go `struct` values in a similar way as encoding packages such as `encoding/json` and `encoding/xml`.

The HCL Go API also offers some alternative approaches however, for processing languages that may be more complex or that include portions whose expected structure cannot be determined until runtime.

The following sections give an overview of different ways HCL can be used in a Go program.

## 2.1 Parsing HCL Input

The first step in processing HCL input provided by a user is to parse it. Parsing turns the raw bytes from an input file into a higher-level representation of the arguments and blocks, ready to be *decoded* into an application-specific form.

The main entry point into HCL parsing is *`hclparse`*, which provides *`hclparse.Parser`*:

```
parser := hclparse.NewParser()
f, diags := parser.ParseHCLFile("server.conf")
```

Variable `f` is then a pointer to an `hcl.File`, which is an opaque abstract representation of the file, ready to be decoded.

Variable `diags` describes any errors or warnings that were encountered during processing; HCL conventionally uses this in place of the usual `error` return value in Go, to allow returning a mixture of multiple errors and warnings together with enough information to present good error messages to the user. We'll cover this in more detail in the next section, *Diagnostic Messages*.

### 2.1.1 Package `hclparse`

**`Parser`**

**func**
: Constructs a new parser object. Each parser contains a cache of files that have already been read, so repeated calls to load the same file will return the same object.

**func**
: Parse the given source code as HCL native syntax, saving the result into the parser's file cache under the given filename.

**func**
: Parse the contents of the given file as HCL native syntax. This is a convenience wrapper around ParseHCL that first reads the file into memory.

**func**
: Parse the given source code as JSON syntax, saving the result into the parser's file cache under the given filename.

**func**
: Parse the contents of the given file as JSON syntax. This is a convenience wrapper around ParseJSON that first reads the file into memory.

The above list just highlights the main functions in this package. For full documentation, see the hclparse godoc.

## 2.2 Diagnostic Messages

An important concern for any machine language intended for human authoring is to produce good error messages when the input is somehow invalid, or has other problems.

HCL uses *diagnostics* to describe problems in an end-user-oriented manner, such that the calling application can render helpful error or warning messages. The word "diagnostic" is a general term that covers both errors and warnings, where errors are problems that prevent complete processing while warnings are possible concerns that do not block processing.

HCL deviates from usual Go API practice by returning its own `hcl.Diagnostics` type, instead of Go's own `error` type. This allows functions to return warnings without accompanying errors while not violating the usual expectation that the absense of errors is indicated by a nil `error`.

In order to easily accumulate and return multiple diagnostics at once, the usual pattern for functions returning diagnostics is to gather them in a local variable and then return it at the end of the function, or possibly earlier if the function cannot continue due to the problems.

```
func returningDiagnosticsExample() hcl.Diagnostics {
    var diags hcl.Diagnostics

    // ...

    // Call a function that may itself produce diagnostics.
    f, moreDiags := parser.LoadHCLFile("example.conf")
    // always append, in case warnings are present
    diags = append(diags, moreDiags...)
    if diags.HasErrors() {
      // If we can't safely continue in the presence of errors here, we
      // can optionally return early.
      return diags
    }

    // ...
```

(continues on next page)

(continued from previous page)

```
    return diags
}
```

A common variant of the above pattern is calling another diagnostics-generating function in a loop, using `continue` to begin the next iteration when errors are detected, but still completing all iterations and returning the union of all of the problems encountered along the way.

In *Parsing HCL Input*, we saw that the parser can generate diagnostics which are related to syntax problems within the loaded file. Further steps to decode content from the loaded file can also generate diagnostics related to *semantic* problems within the file, such as invalid expressions or type mismatches, and so a program using HCL will generally need to accumulate diagnostics across these various steps and then render them in the application UI somehow.

### 2.2.1 Rendering Diagnostics in the UI

The best way to render diagnostics to an end-user will depend a lot on the type of application: they might be printed into a terminal, written into a log for later review, or even shown in a GUI.

HCL leaves the responsibility for rendering diagnostics to the calling application, but since rendering to a terminal is a common case for command-line tools, the *hcl* package contains a default implementation of this in the form of a "diagnostic text writer":

```
wr := hcl.NewDiagnosticTextWriter(
    os.Stdout,      // writer to send messages to
    parser.Files(), // the parser's file cache, for source snippets
    78,             // wrapping width
    true,           // generate colored/highlighted output
)
wr.WriteDiagnostics(diags)
```

This default implementation of diagnostic rendering includes relevant lines of source code for context, like this:

```
Error: Unsupported block type

  on example.tf line 4, in resource "aws_instance" "example":
   2: provisionr "local-exec" {

Blocks of type "provisionr" are not expected here. Did you mean "provisioner"?
```

If the "color" flag is enabled, the severity will be additionally indicated by a text color and the relevant portion of the source code snippet will be underlined to draw further attention.

## 2.3 Decoding Into Native Go Values

The most straightforward way to access the content of an HCL file is to decode into native Go values using `reflect`, similar to the technique used by packages like `encoding/json` and `encoding/xml`.

Package `gohcl` provides functions for this sort of decoding. Function `DecodeBody` attempts to extract values from an HCL *body* and write them into a Go value given as a pointer:

```
type ServiceConfig struct {
  Type       string `hcl:"type,label"`
  Name       string `hcl:"name,label"`
  ListenAddr string `hcl:"listen_addr"`
```

(continues on next page)

(continued from previous page)

```
}
type Config struct {
  IOMode   string          `hcl:"io_mode"`
  Services []ServiceConfig `hcl:"service,block"`
}

var c Config
moreDiags := gohcl.DecodeBody(f.Body, nil, &c)
diags = append(diags, moreDiags...)
```

The above example decodes the *root body* of a file `f`, presumably loaded previously using a parser, into the variable `c`. The field labels within the struct types imply the schema of the expected language, which is a cut-down version of the hypothetical language we showed in *Introduction to HCL*.

The struct field labels consist of two comma-separated values. The first is the name of the corresponding argument or block type as it will appear in the input file, and the second is the type of element being named. If the second value is omitted, it defaults to `attr`, requesting an attribute.

Nested blocks are represented by a struct or a slice of that struct, and the special element type `label` within that struct declares that each instance of that block type must be followed by one or more block labels. In the above example, the `service` block type is defined to require two labels, named `type` and `name`. For label fields in particular, the given name is used only to refer to the particular label in error messages when the wrong number of labels is used.

By default, all declared attributes and blocks are considered to be required. An optional value is indicated by making its field have a pointer type, in which case `nil` is written to indicate the absense of the argument.

The sections below discuss some additional decoding use-cases. For full details on the *gohcl* package, see the godoc reference.

### 2.3.1 Variables and Functions

By default, arguments given in the configuration may use only literal values and the built in expression language operators, such as arithmetic.

The second argument to `gohcl.DecodeBody`, shown as `nil` in the previous example, allows the calling application to additionally offer variables and functions for use in expressions. Its value is a pointer to an `hcl.EvalContext`, which will be covered in more detail in the later section *Expression Evaluation*. For now, a simple example of making the id of the current process available as a single variable called `pid`:

```
type Context struct {
    Pid string
}
ctx := gohcl.EvalContext(&Context{
    Pid: os.Getpid()
})
var c Config
moreDiags := gohcl.DecodeBody(f.Body, ctx, &c)
diags = append(diags, moreDiags...)
```

`gohcl.EvalContext` constructs an expression evaluation context from a Go struct value, making the fields available as variables and the methods available as functions, after transforming the field and method names such that each word (starting with an uppercase letter) is all lowercase and separated by underscores.

```
name = "example-program (${pid})"
```

### 2.3.2 Partial Decoding

In the examples so far, we've extracted the content from the entire input file in a single call to `DecodeBody`. This is sufficient for many simple situations, but sometimes different parts of the file must be evaluated separately. For example:

- If different parts of the file must be evaluated with different variables or functions available.
- If the result of evaluating one part of the file is used to set variables or functions in another part of the file.

There are several ways to perform partial decoding with `gohcl`, all of which involve decoding into HCL's own types, such as `hcl.Body`.

The most general approach is to declare an additional struct field of type `hcl.Body`, with the special field tag type `remain`:

```
type ServiceConfig struct {
  Type       string   `hcl:"type,label"`
  Name       string   `hcl:"name,label"`
  ListenAddr string   `hcl:"listen_addr"`
  Remain     hcl.Body `hcl:",remain"`
}
```

When a `remain` field is present, any element of the input body that is not matched is retained in a body saved into that field, which can then be decoded in a later call, potentially with a different evaluation context.

Another option is to decode an attribute into a value of type *hcl.Expression*, which can then be evaluated separately as described in expression-eval.

## 2.4 Decoding With Dynamic Schema

In section *Decoding Into Native Go Values*, we saw the most straightforward way to access the content from an HCL file, decoding directly into a Go value whose type is known at application compile time.

For some applications, it is not possible to know the schema of the entire configuration when the application is built. For example, HashiCorp Terraform uses HCL as the foundation of its configuration language, but parts of the configuration are handled by plugins loaded dynamically at runtime, and so the schemas for these portions cannot be encoded directly in the Terraform source code.

HCL's `hcldec` package offers a different approach to decoding that allows schemas to be created at runtime, and the result to be decoded into dynamically-typed data structures.

The sections below are an overview of the main parts of package `hcldec`. For full details, see the package godoc.

### 2.4.1 Decoder Specification

Whereas *`gohcl`* infers the expected schema by using reflection against the given value, `hcldec` obtains schema through a decoding *specification*, which is a set of instructions for mapping HCL constructs onto a dynamic data structure.

The `hcldec` package contains a number of different specifications, each implementing `hcldec.Spec` and having a `Spec` suffix on its name. Each spec has two distinct functions:

- Adding zero or more validation constraints on the input configuration file.
- Producing a result value based on some elements from the input file.

The most common pattern is for the top-level spec to be a `hcldec.ObjectSpec` with nested specifications defining either blocks or attributes, depending on whether the configuration file will be block-structured or flat.

```
spec := hcldec.ObjectSpec{
    "io_mode": &hcldec.AttrSpec{
        Name: "io_mode",
        Type: cty.String,
    },
    "services": &hcldec.BlockMapSpec{
        TypeName:   "service",
        LabelNames: []string{"type", "name"},
        Nested:     hcldec.ObjectSpec{
            "listen_addr": &hcldec.AttrSpec{
                Name:     "listen_addr",
                Type:     cty.String,
                Required: true,
            },
            "processes": &hcldec.BlockMapSpec{
                TypeName:   "service",
                LabelNames: []string{"name"},
                Nested:     hcldec.ObjectSpec{
                    "command": &hcldec.AttrSpec{
                        Name:     "command",
                        Type:     cty.List(cty.String),
                        Required: true,
                    },
                },
            },
        },
    },
}
val, moreDiags := hcldec.Decode(f.Body, spec, nil)
diags = append(diags, moreDiags...)
```

The above specification expects a configuration shaped like our example in *Introduction to HCL*, and calls for it to be decoded into a dynamic data structure that would have the following shape if serialized to JSON:

```
{
  "io_mode": "async",
  "services": {
    "http": {
      "web_proxy": {
        "listen_addr": "127.0.0.1:8080",
        "processes": {
          "main": {
            "command": ["/usr/local/bin/awesome-app", "server"]
          },
          "mgmt": {
            "command": ["/usr/local/bin/awesome-app", "mgmt"]
          }
        }
      }
    }
  }
}
```

### 2.4.2 Types and Values With `cty`

HCL's expression interpreter is implemented in terms of another library called *cty*, which provides a type system which HCL builds on and a robust representation of dynamic values in that type system. You could think of *cty* as being a bit like Go's own `reflect`, but for the results of HCL expressions rather than Go programs.

The full details of this system can be found in its own repository, but this section will cover the most important highlights, because `hcldec` specifications include *cty* types (as seen in the above example) and its results are *cty* values.

`hcldec` works directly with *cty* — as opposed to converting values directly into Go native types — because the functionality of the *cty* packages then allows further processing of those values without any loss of fidelity or range. For example, *cty* defines a JSON encoding of its values that can be decoded losslessly as long as both sides agree on the value type that is expected, which is a useful capability in systems where some sort of RPC barrier separates the main program from its plugins.

Types are instances of `cty.Type`, and are constructed from functions and variables in *cty* as shown in the above example, where the string attributes are typed as `cty.String`, which is a primitive type, and the list attribute is typed as `cty.List(cty.String)`, which constructs a new list type with string elements.

Values are instances of `cty.Value`, and can also be constructed from functions in *cty*, using the functions that include `Val` in their names or using the operation methods available on `cty.Value`.

In most cases you will eventually want to use the resulting data as native Go types, to pass it to non-*cty*-aware code. To do this, see the guides on Converting between types (staying within *cty*) and Converting to and from native Go values.

### 2.4.3 Partial Decoding

Because the `hcldec` result is always a value, the input is always entirely processed in a single call, unlike with *gohcl*.

However, both *gohcl* and *hcldec* take `hcl.Body` as the representation of input, and so it is possible and common to mix them both in the same program.

A common situation is that *gohcl* is used in the main program to decode the top level of configuration, which then allows the main program to determine which plugins need to be loaded to process the leaf portions of configuration. In this case, the portions that will be interpreted by plugins are retained as opaque `hcl.Body` until the plugins have been loaded, and then each plugin provides its `hcldec.Spec` to allow decoding the plugin-specific configuration into a `cty.Value` which be transmitted to the plugin for further processing.

In our example from *Introduction to HCL*, perhaps each of the different service types is managed by a plugin, and so the main program would decode the block headers to learn which plugins are needed, but process the block bodies dynamically:

```
type ServiceConfig struct {
  Type         string    `hcl:"type,label"`
  Name         string    `hcl:"name,label"`
  PluginConfig hcl.Body `hcl:",remain"`
}
type Config struct {
  IOMode   string          `hcl:"io_mode"`
  Services []ServiceConfig `hcl:"service,block"`
}

var c Config
moreDiags := gohcl.DecodeBody(f.Body, nil, &c)
```

(continues on next page)

(continued from previous page)

```
diags = append(diags, moreDiags...)
if moreDiags.HasErrors() {
    // (show diags in the UI)
    return
}

for _, sc := range c.Services {
    pluginName := block.Type

    // Totally-hypothetical plugin manager (not part of HCL)
    plugin, err := pluginMgr.GetPlugin(pluginName)
    if err != nil {
        diags = diags.Append(&hcl.Diagnostic{ /* ... */ })
        continue
    }
    spec := plugin.ConfigSpec() // returns hcldec.Spec

    // Decode the block body using the plugin's given specification
    configVal, moreDiags := hcldec.Decode(sc.PluginConfig, spec, nil)
    diags = append(diags, moreDiags...)
    if moreDiags.HasErrors() {
        continue
    }

    // Again, hypothetical API within your application itself, and not
    // part of HCL. Perhaps plugin system serializes configVal as JSON
    // and sends it over to the plugin.
    svc := plugin.NewService(configVal)
    serviceMgr.AddService(sc.Name, svc)
}
```

### 2.4.4 Variables and Functions

The final argument to `hcldec.Decode` is an expression evaluation context, just as with `gohcl.DecodeBlock`.

This object can be constructed using *the gohcl helper function* as before if desired, but you can also choose to work directly with *`hcl.EvalContext`* as discussed in *Expression Evaluation*:

```
 ctx := &hcl.EvalContext{
     Variables: map[string]cty.Value{
         "pid": cty.NumberIntVal(int64(os.Getpid())),
     },
 }
val, moreDiags := hcldec.Decode(f.Body, spec, ctx)
diags = append(diags, moreDiags...)
```

As you can see, this lower-level API also uses *`cty`*, so it can be particularly convenient in situations where the result of dynamically decoding one block must be available to expressions in another block.

## 2.5 Expression Evaluation

Each argument attribute in a configuration file is interpreted as an expression. In the HCL native syntax, certain basic expression functionality is always available, such as arithmetic and template strings, and the calling application can extend this by making available specific variables and/or functions via an *evaluation context*.

We saw in *Decoding Into Native Go Values* and *Decoding With Dynamic Schema* some basic examples of populating an evaluation context to make a variable available. This section will look more closely at the `hcl.EvalContext` type and how HCL expression evaluation behaves in different cases.

This section does not discuss in detail the expression syntax itself. For more information on that, see the HCL Native Syntax specification.

**`EvalContext`**
: `hcl.EvalContext` is the type used to describe the variables and functions available during expression evaluation, if any. Its usage is described in the following sections.

### 2.5.1 Defining Variables

As we saw in *Decoding With Dynamic Schema*, HCL represents values using an underlying library called *`cty`*. When defining variables, their values must be given as `cty.Value` values.

A full description of the types and value constructors in *`cty`* is in the reference documentation. Variables in HCL are defined by assigning values into a map from string names to `cty.Value`:

```
ctx := &hcl.EvalContext{
    Variables: map[string]cty.Value{
        "name": cty.StringVal("Ermintrude"),
        "age":  cty.NumberIntVal(32),
    },
}
```

If this evaluation context were passed to one of the evaluation functions we saw in previous sections, the user would be able to refer to these variable names in any argument expression appearing in the evaluated portion of configuration:

```
message = "${name} is ${age} ${age == 1 ? "year" : "years"} old!"
```

If you place `cty`'s *object* values in the evaluation context, then their attributes can be referenced using the HCL attribute syntax, allowing for more complex structures:

```
ctx := &hcl.EvalContext{
    Variables: map[string]cty.Value{
        "path": cty.ObjectVal(map[string]cty.Value{
            "root":    cty.StringVal(rootDir),
            "module":  cty.StringVal(moduleDir),
            "current": cty.StringVal(currentDir),
        }),
    },
}
```

```
source_file = "${path.module}/foo.txt"
```

### 2.5.2 Defining Functions

Custom functions can be defined by you application to allow users of its language to transform data in application-specific ways. The underlying function mechanism is also provided by *`cty`*, allowing you to define the arguments a given function expects, what value type it will return for given argument types, etc. The full functions model is described in the `cty` documentation section Functions System.

There are a number of "standard library" functions available in a `stdlib` package within the *`cty`* repository, avoiding the need for each application to re-implement basic functions for string manipulation, list manipulation, etc. It also

includes function-shaped versions of several operations that are native operators in HCL, which should generally *not* be exposed as functions in HCL-based configurationf formats to avoid user confusion.

You can define functions in the `Functions` field of *`hcl.EvalContext`*:

```
ctx := &hcl.EvalContext{
    Variables: map[string]cty.Value{
        "name": cty.StringVal("Ermintrude"),
    },
    Functions: map[string]function.Function{
        "upper":  stdlib.UpperFunc,
        "lower":  stdlib.LowerFunc,
        "min":    stdlib.MinFunc,
        "max":    stdlib.MaxFunc,
        "strlen": stdlib.StrlenFunc,
        "substr": stdlib.SubstrFunc,
    },
}
```

If this evaluation context were passed to one of the evaluation functions we saw in previous sections, the user would be able to call any of these functions in any argument expression appearing in the evaluated portion of configuration:

```
message = "HELLO, ${upper(name)}!"
```

### 2.5.3 Expression Evaluation Modes

HCL uses a different expression evaluation mode depending on the evaluation context provided. In HCL native syntax, evaluation modes are used to provide more relevant error messages. In JSON syntax, which embeds the native expression syntax in strings using "template" syntax, the evaluation mode determines whether strings are evaluated as templates at all.

If the given *`hcl.EvalContext`* is `nil`, native syntax expressions will react to users attempting to refer to variables or functions by producing errors indicating that these features are not available at all, rather than by saying that the specific variable or function does not exist. JSON syntax strings will not be evaluated as templates *at all* in this mode, making them function as literal strings.

If the evaluation context is non-`nil` but either `Variables` or `Functions` within it is `nil`, native syntax will similarly produce "not supported" error messages. JSON syntax strings *will* parse templates in this case, but can also generate "not supported" messages if e.g. the user accesses a variable when the variables map is `nil`.

If neither map is nil, HCL assumes that both variables and functions are supported and will instead produce error messages stating that the specific variable or function accessed by the user is not defined.

## 2.6 Advanced Decoding With The Low-level API

## 2.7 Design Patterns for Complex Systems

CHAPTER 3

# Configuration Language Design

# Golang Package Index

# Index