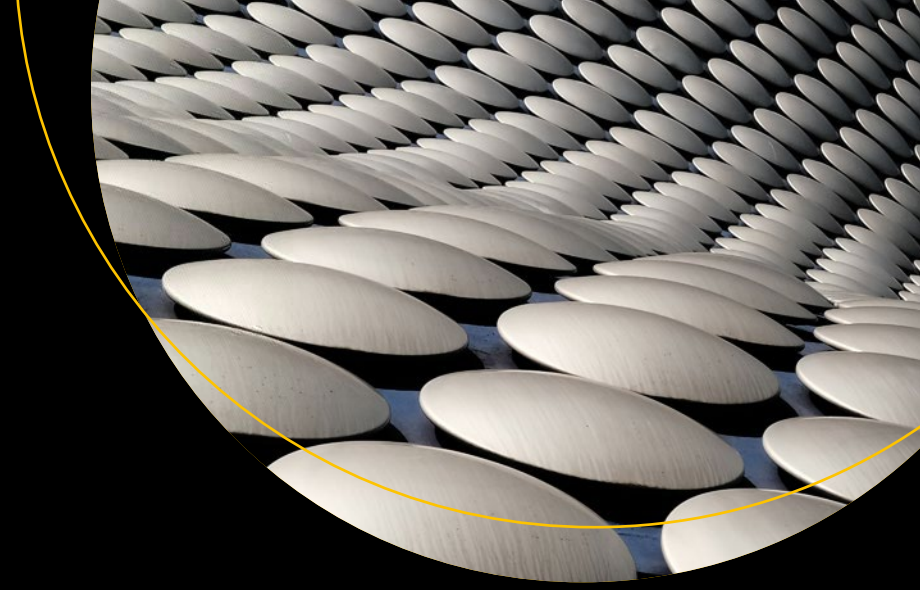

Beginning HCL Programming

Using Hashicorp Language for Automation and Configuration

Pierluigi Riti
David Flynn

Apress®

Beginning HCL Programming

Using Hashicorp Language for Automation and Configuration

Pierluigi Riti
David Flynn

Apress®

Beginning HCL Programming: Using Hashicorp Language for Automation and Configuration

Pierluigi Riti
Mullingar, Ireland

David Flynn
New Orleans, LA, USA

ISBN-13 (pbk): 978-1-4842-6633-5
ISBN-13 (electronic): 978-1-4842-6634-2
https://doi.org/10.1007/978-1-4842-6634-2

Copyright © 2021 by Pierluigi Riti and David Flynn

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image by Bilge Tekin on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484266335. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

To my family

—Pierluigi Riti

I would obviously like to dedicate this book to Pier.
I would also like to mention all the pioneers around the world who have, are, and will hopefully continue to develop, research, and explore these new technologies in order to bring exciting new avenues for technological advancement.
A further mention to all those at Mastercard and previous companies who have developed my knowledge to allow such endeavors as this.

—David Flynn

Table of Contents

About the Authors

Pierluigi Riti is a Lead Security Information Engineer and DevOps fanatic he actually work in MasterCard. He has worked for company like, Coupa Software, and Synchronoss Technologies. Prior to that, he was a senior software engineer at Ericsson and Tata. His experience includes implementing DevOps in the cloud using Google Cloud Platform, AWS, and Azure. Also, he has over 20 years of extensive experience in more general design and development of different scale applications particularly in the telco and financial industries. He has quality development skills using the latest technologies including Java, J2EE, C#, F#, .NET, Spring .NET, EF, WPF, WF, WinForm, WebAPI, MVC, Nunit, Scala, Spring, JSP, EJB, Struts, Struts2, SOAP, REST, C, C++, Hibernate, NHibernate, WebLogic, XML, XSLT, Unix script, Ruby, and Python.

David Flynn is an Associate Analyst in Employee Access Business Operations at MasterCard. He is an electronic engineer with experience in telecommunications, networks, software, security, and financial systems. David started out as a telecommunications engineer working on voice, data, and wireless systems for Energis and later Nortel Networks, supporting systems such as Lucent G3r, Alcatel E10, and Nortel Passport. He then did some time in transport and private security abroad before retraining in computing, cyber security, and cloud systems plus doing cyber security and telecomm research for the Civil Service. He has completed separate diplomas in computing and cloud focusing on Windows, C#, Google, AWS, and PowerShell among other technologies. David has also worked as a C# engineer. More recently, David has worked for various fintech companies including Bank of America and Merrill Lynch, focusing on technical and application support encompassing such technologies as Rsa Igl, Rsa SecurID, IBM Tam/Isam, Postgres/Oracle databases, mainframes, Tandem, CyberArk, MaxPro, and Active Directory.

Introduction

HashiCorp offers a full range of products to improve the life of every DevOps engineer. Our goal with this book is to introduce various software applications and show how to use the HCL language to configure them. This book is not meant to be an exhaustive guide to all possible scenarios, but to introduce the software options and how to use them together to create a complete Infrastructure as Code. To get the most out of this book, you should have a basic knowledge of Bash PowerShell scripting and a basic knowledge of programming in general.

CHAPTER 1

Introduction to HCL

HashiCorp is a significant player in the cloud revolution. The company produces most of the essential tools for any DevOps engineer or cloud engineer.

The HashiCorp ecosystem is quite huge; the aim of this book is to introduce the configuration language and the different HashiCorp software components.

HCL, A Brief Introduction

HCL is a configuration language designed to be both human- and machine-readable. HCL is an automation language that is used to manage, create, and release Infrastructure as Code. Based on a study conducted on the GitHub repository, HCL was the third highest in terms of language growth popularity in 2019, which indicates how important the HCL platform has become, which in turn was probably aided by HashiCorp applications like Terraform, Consul, and Vault.

HCL is designed to ease the maintenance of cloud and multi-cloud hybrid solutions. The language is structural, with the goal of being easily understood by humans and machines.

HCL is fully JSON-compatible and the language is intended to be used to build DevOps tools and servers, manage PKI passwords, and release Docker images. HCL gets its inspiration from *libucl*, the Nginx configuration, and other configuration languages.

---

libucl, the Universal Configuration Library Parser, is the main inspiration for the HCL language. As you will see, HCL uses a similar structure, and UCL is heavy inspired by Nginx configuration.

When HCL was designed, the choice was made to mix together the power of a general-purpose language like Ruby, Python, or Java with the simplicity and human readability of JSON. HashiCorp designed its own DSL language.

© Pierluigi Riti and David Flynn 2021
P. Riti and D. Flynn, *Beginning HCL Programming*, https://doi.org/10.1007/978-1-4842-6634-2_1

The major usage for HCL is with Terraform. Terraform is HashiCorp's Infrastructure as Code (IaC) or cloud infrastructure automation tool. Both HCL and Terraform enable any DevOps engineer to develop their own tools.

The term *general-purpose language* (GPL) covers the family of programming languages used to develop and design any type of application. This includes Ruby, Python, Go, and Java. On the opposite side are DSLs (domain-specific languages). This family of languages is used in a specific domain, for example, the HTML language. The big difference between a GPL and a DSL language is essentially the use. With a GPL language, we can create and solve any type of problem. A DSL language is designed to solve one specific problem; for example, HTML is used to define how a webpage must be visualized on the screen.

Syntax Overview

HCL comprises a family of DSLs. In this book, we will focus on HCL2, which emphasizes simplicity. HCL has a similar structure to JSON, which allows for a high probability of equivalence between JSON and HCL.

HCL was designed to be JSON-compatible, and every HashiCorp product has a specific call for the relevant API and/or configuration. The entire product suite encompasses this basic syntax. Similar to other languages there are some primitive data types and structures:

- String
- Boolean
- Number
- Tuple
- Object

These are the basic structures and data types that can be used to write HCL code. To create a variable, we can use this syntax: `key = value` *(the* space doesn't matter). The `key` is the name of the value, and the `value` can be any of the primitive types such as string, number, or boolean:

```
description = "This is a String"
number = 1
```

String

The string is defined using the double-quoted syntax and is based (only) on the UTF-8 character set:

```
hello = "Hello World"
```

It is not possible to create a multi-line string. To create multi-line strings, we need to use another syntax.

To create a multi-line string, HCL uses a *here document* syntax, which is a multi-line string starting with << followed by the delimiter identifier (normally a three-letter word like EOF) and succeeded by the same delimiter identifier.

A here document is a file literal or input stream used in particular in the Unix system for adding a multi-line string in a piece of code. Typically this type of syntax starts with << EOF and ends with an EOF.

To create a multi-line string in this way, we can use any text as the delimiter identifier. In this case, it is EOF:

```
<< EOF
Hello
HCL
EOF
```

Number

In HCL, all number data types have a default base of 10, which can be either of the following:

- Integer
- Float

For example,

```
first=10

second=10.56
```

The variable `first` is an integer number while the variable `second` is a float number. A number can be expressed in hexadecimal by adding the prefix `0x`, octal using the prefix number `0`, or scientific format using `1e10`. For example, we can define the number data types as follows:

```
hexadecimal=0x1E

octal=07

scientific=2e15
```

Tuple

HCL supports tuple creation using the square brace syntax, for example:

```
array_test=["first",2,"third"]
```

The value written in an array can be of different types. In the previous example, you can see two string data types and one number data type. In HCL, it is possible to create an array with any type of object:

```
test_array=[true,
    << ERRDOC
      Hello
      Array
     ERRDOC,
     "Test"]
```

Object

In HCL, objects and nested objects can be created using this syntax:

```
<type> <variable/object name> {...}:
.:
```

```
provider "aws" {
       description = "AWS server"
}
```

We can use the same structure for the object to define an *input variable*:

```
variable "provider" {
       name = "AWS"
}
```

Boolean

A Boolean variable in HCL follows the same rules of the other languages. The only value it can have is either `true` or `false`:

```
variable "active"{
  value = True
}
```

Comment

A single line of comment can be created using the # or the `//`:

```
provider "aws" {
       # This is a single line comment
       // This is another single line comment
}
```

To create a multi-line comment, the `/*....*/` format can be used:

```
provider "aws" {
       /*
       This is a multi-line comment example
       */
}
```

HIL and HCL

HCL is used for the majority of the use-case scenarios with the Terraform tool. This symbiosis has become a significant factor in the growth of the popularity of HCL.

The HCL that is used to create a template can be translated into JSON by the parser, an important step for creating a valid and usable template for HIL.

HIL, or HashiCorp Interpolating Language, is the language used for *interpolating* any string. It is primarily used in combination with HCL to use a variable defined in other parts of the configuration. HIL uses the syntax `${..}` for interpolating the variable, as shown:

```
test = "Hello ${var.world}"
```

The HIL is used to have something similar to a template. This language is mostly used in Terraform. The goal is to have a rich language definition of the infrastructure. The idea behind the creation of HIL was to extract the definition language used in Terraform and then clean it up to create a better and more powerful language.

HIL has its own syntax, which it shares with HCL, such as comments, multi-line comments, Boolean, etc. With HIL, it is possible to create a function for the call, which can be used in the interpolation syntax of the function. This is, in turn, is called with the syntax `func(arg1, arg2, ....)`. For example, we can create a function with the HIL in this way:

```
test = "${ func("Hello", ${hello.var})}"
```

HIL is utilized in more depth when we use Terraform and other software like Nomad.

How HCL Works

You just got a concise introduction to HCL and HIL. But in order to progress beyond this point, you need to create a basic template to illustrate how both components work.

HCL and HIL use the GPL language to create JSON code for the necessary configurations. JSON itself is quite capable of producing the necessary code or configurations so why are HCL/HIL needed? JSON lacks the ability in insert comments, which is essential for reviewing code or configurations, particularly for the massive infrastructure that HCL/HIL is aimed at.

HCL consists of three distinct, integrated sublanguages. All three work together to permit us to create the configuration file:

- *Structural language*
- *Expression language*
- *Template language*

The *structural language* is used to define the overall structure, the hierarchical configuration structure, and its serialization. The three main parts for an HCL snippet are defined as *bodies*, the *block*, and *attributes*.

The *expression language* is used to express the value of the attribute, which can be expressed in either of two ways: a literal or a derivation of another value.

The *template language* is used to compose the value into a string. The template language uses one of the several types of expression defined in the expression language.

When code is composed in HCL, all three sublanguages are normally used. The structural language is used at the top level to define the basis of the configuration file. The expression language and the template language can also be used in isolation or to implement a feature like a REPL, a debugger that can be integrated into more limited HCL syntaxes such as the JSON profile itself.

Syntax Components

A fundamental part of every language is the syntax. Now we'll introduce the basic grammar of HCL and the fundamental parts used to build the language. You've seen which data and type structures are allowed in the HCL language. Now we will delve deeper into syntax. The basic syntax in HCL has these basic rules:

- Every name starting with an uppercase letter is a global production. This means it is common to all syntax specified in the document used to define the program. This is similar to a global variable in other languages.
- Every name starting with a lowercase letter is a local production. This means it is valid only in the part of the document where it is defined. This is similar to a local variable in other languages.
- Double quotes (“) or single quotes (‘) are used to mark a literal character sequence. This can be a punctuation marker or a keyword.

- The default operator for combining items is the *concatenation, the operator* +.
- The symbol | is a logical OR, which means one of the two operators, left or right, must be present.
- The symbol * indicates a "zero or more" repetition of the item on the left. This means we can have a variable number of elements, with the minimum value of 0.
- The symbol ? indicates one or more repetitions of the item to the left.
- The parentheses, ( ) , are used to group items in order to apply the previous operator to them collectively.

These are the basic syntax notations used to define the grammar of the language. They are used in combination with the structure and data types for creating the configuration file(s).

When a HCL configuration file is created, a certain set of rules are used to describe the syntax and grammar involved. There are three distinct sections of the file:

- *Attributes*, where we assign a value to a specific value
- *The block*, which is used to create a child body annotated by a name and an optional label
- *The body content,* which is essentially a collection of attributes and the block

This structure defines the model of the configuration file. A *configuration file* is nothing more than a sequence of characters used to create the body. If we want to define a similar BNF grammar, we can define the basic structure for a configuration file as follows:

```
ConfigFile   = Body;
Body         = (Attribute | Block | OneLineBlock)*;
Attribute    = Identifier "=" Expression Newline;
Block        = Identifier (StringLit|Identifier)* "{" Newline Body "}"
               Newline;
OneLineBlock = Identifier (StringLit|Identifier)* "{" (Identifier "="
               Expression)? "}" Newline;
```

A BNF (Backus-Naur Form) grammar is a notation technique used for free-form grammar. With this technique, we can define a new type of grammar for our own language. This is normally used when we create a new language, like HCL. The BNF is largely used when defining the language and is very helpful when we need to understand the language itself. There is a new version of the BNF called EBFN (Extend-Backus-Naur Form). The BNF is a simple language used in particular in the academic world. There is no unique definition and it is mostly used to describe metacode to be read to a human and is normally written on one line. The EBFN lets us write a more complex model representation of the code; it is possible to define a variable and function with a more complex syntax.

Identifiers

Identifiers are used to assign a name to a block, an attribute, or a variable. An identifier is a string of characters, beginning with a letter or a certain unambiguous punctuation token, followed by any number or letter of Unicode.

The standard used to define an identifier is the Unicode standard, defined in the document UAX #31- Section 2. This document also defines the BNF grammar we can use to write our identifiers. The grammar is as follows:

```
<Identifier> := <Start> <Continue>* (<Medial> <Continue>+)*
```

To define an identifier, this notation is used:

```
Identifier = ID_Start (ID_Continue | '-')*;
```

where

- `ID_Start` consists of sequence of Unicode letters and certain unambiguous punctuation.
- `ID_Continue` defines a set of Unicode letters, combining marks and such, as defined in the Unicode standard.

In addition to the first two characters, `ID_Start` and `ID_Continue`, we use the character `'-'`; this character can also be used to define identifiers. The usage of the `'-'` character allows the identifier to have this character as part of the name.