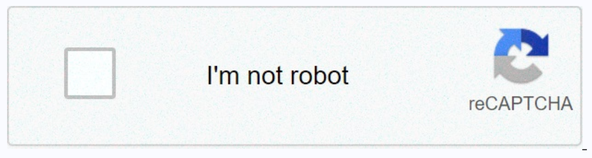

Continue

Terraform hcl format

format produces a string by formatting a number of other values according to a specification string. It is similar to the printf function in C, and other similar functions in other programming languages. » Examples > format(Hello, %s!, Ander) Hello, Ander! > format(There are lights %d, 4) There are 4 lights Simple format verbs like %s and %d behave similarly to template interpolation syntax, which is often more readable: > format(Hello, %s!, var.name) Hello, Valentina! > Hello, $'var.name'! Hello, Valentina! Therefore, the formatting function is most useful when using more complex formatting specifications, as described in the next section. » Specification syntax The specification is a string that includes format verbs that are entered with the % character. The function call must then have an additional argument for each sequence of verbs in the specification. Verbs are paired with consecutive arguments and are formatted as follows, as long as each given argument is convertible to the type required by the format verb. The specification can contain the following verbs: Verb Result %% Literal percentage sign, without consuming any values. %v Default format based on value type, as described below. %#v JSON serialization of the value, as with jsonencode. %t Convert to Boolean and produce true or false. %b Convert to integer and produce binary representation. %d Convert to integer and produce decimal representation. %o Convert to integer and produce octal representation. %x Convert to integer and produce hexadecimal representation with lowercase letters. %X As %x, but use uppercase letters. %e Convert to number and produce scientific notation, such as -1.234456e+78. %E As %e, but use an uppercase E to enter the exponent. %f Convert to number and produce decimal fraction notation without exponent, such as 123,456. %g As %e for large exponents or as %f otherwise. %G As %E for large exponents or as %f otherwise. %s Convert to String and insert the characters in the string. %q Convert to String and produce a JSON quoted string representation. When using %v, one of the following format verbs is chosen based on the value type: Type Verb String %s number %g bool %t any other %#v Null Values produces the null string if %v or %#v is formatted and cause an error for other verbs. A width modifier can be included with an optional decimal number immediately before the letter of the verb, to specify how many characters will be used to represent the value. Precision can be specified after width (optional) with a period (.) followed by a decimal number. If width or precision is omitted, the default values are selected based on the specified value. By Default sequence result %f Width and accuracy. %9f Width 9, default accuracy. %.2f Default width, precision 2. %9.2f Width 9, accuracy 2. The following additional symbols can be used immediately after the % symbol to set additional indicators: Leave Leave Symbol Result Space space where the sign would be if a number is positive. + Show the sign of a number even if it is positive. - Pad the width with spaces on the left instead of the right. 0 Pad the width with leading zeros instead of spaces. By default, % sequences consume successive arguments from the first. Entering a sequence [n] immediately before the letter of the verb, where n is a decimal integer, explicitly chooses a value argument determined by its index based on one. Subsequent calls without an explicit index will continue with n+1, n+2, and so on. The function fails if the format string requests an impossible conversion or has access to more arguments than are provided. An error also occurs for an unsupported format verb. formatdate is a specialized formatting feature for human-readable timestamps. formatlist uses the same specification syntax to generate a list of strings. Terraform configuration syntax is called HashiCorp Configuration Language (HCL). It is intended to find a balance between readable and editable for humans, as well as to be machine friendly. For ease of use of the machine, Terraform can also read JSON configurations. For general Terraform configurations, however, we recommend that you use HCL Terraform syntax. » Terraform Syntax Here is an example of Terraform HCL syntax: . */ resource aws_instance web á ami á $-var.ami count - 2 source_dest_check - fake connection - user - root - Basic bullet reference: Single line comments begin with multiline comments are adjusted with /* and */ Values are assigned with the syntax key - value (white space does not matter). The value can be any primitive (string, number, Boolean), list, or map. Strings are enclosed in double quotation marks. Strings can interpolate other values using the syntax wrapped in $, such as $. The full syntax for interpolation is documented here. Multiline strings can use shell-style syntax here doc, with the string starting with a bookmark such as <<EOF, and then the string ending with EOF on its own line. Lines in the string and end marker should not be indented. The numbers are supposed to be the base 10. If you prefix a number with 0x, it is treated as a hexadecimal number. Boolean values: true, false. Lists of primitive types can be done with square brackets ([]). Example: [foo, bar, baz]. Maps can be made with keys and two points (:): foo: bar, bar: baz. Quotes can be omitted in keys, unless the key starts with a number, in which case quotation marks are required. Commas are required between key/value pairs for single-line maps. A new line between key/value pairs is sufficient on maps several lines. In addition to the basics, the syntax supports section hierarchies, such as the resource and variable in the previous example. These sections are similar to maps, but visually they look better. For example, these are almost equivalent: ami variable - description - - AMI to use - equals: variable - [a ami: description: the AMI to use, ? Repeating multiple sections of variables creates the list of variables. When possible, use sections as they are visually clearer and more readable. » Json Syntax Terraform also supports reading JSON-formatted configuration files. The previous example converted to JSON: variable: ami: description: the AMI to use , resource: aws_instance: ? web: ami: $-var.ami, count: 2, source_dest_check: false, connection: . The disadvantages of JSON are less human readability and lack of comments. Otherwise, the two are fully interoperable. For a hands-on tutorial, try the hint introduction in HashiCorp Learn. Terraform uses its own configuration language, designed to allow concecisous descriptions of the infrastructure. The Terraform language is declarative, describing an intended goal rather than the steps to achieve that goal. » Resources and Modules The primary purpose of the Terraform language is to declare resources. All other language features exist only to make resource definition more flexible and convenient. A resource group can be gathered into a module, which creates a larger configuration unit. A resource describes a single infrastructure object, while a module can describe a set of objects and the relationships required between them to create a top-level system. An terraform configuration consists of a root module, where evaluation begins, along with a tree of child modules created when one module calls another. » Arguments, blocks, and expressions Terraform language syntax consists of only a few basic elements: the main aws_vpc resource - cidr_block var.base_cidr_block <BLOCK type-> á - Block Body - Argument - Los<BLOCK label-><BLOCK label-> <IDENTIFIER> <EXPRESSION>blocks are containers for other content and typically represent the configuration of some type of object , as a resource. Blocks have a block type, can have zero or more tags, and have a body that contains any number of nested arguments and blocks. Most Terraform features are controlled by top-level blocks in a configuration file. Arguments assign a value to a name. They appear inside blocks. Expressions represent a value, either literally or by referencing and combining other values. They appear as values for arguments or within other expressions. For complete details about Terraform syntax, see: Configuration Syntax Expressions » Code Organization The Terraform language uses configuration files that are named with the .tf file extension. There is also a JSON-based variant of the language that is named with the .tf.json file extension. The archives configuration should always use UTF-8 encoding, and by convention are typically maintained with Unix-style line terminations (LF) instead of</EXPRESSION> </IDENTIFIER> </BLOCK> </BLOCK> </BLOCK> </BLOCK> (CRLF), although both are accepted. A module is a collection of .tf or .tf.json files saved together in a directory. The root module is constructed from the configuration files in the current working directory when Terraform is running, and this module can reference child modules in other directories, which in turn can reference other modules, and so on. The simplest configuration of Terraform is a single root module that contains a single .tf file. A configuration can grow gradually as more resources are added, either by creating new configuration files within the root module or by organizing resource sets into child modules. » Configuration sorting Because the Terraform configuration language is declarative, the order of the blocks is generally not significant. (The order of provisioner blocks within a resource is the only major feature where the order of blocks is important.) Terraform automatically processes resources in the correct order based on the relationships defined between them in the configuration, so you can organize resources into source files in any way that makes sense for your infrastructure. » Terraform CLI vs. Providers Terraform's command-line interface (CLI) is a general engine for evaluating and applying Terraform configurations. Defines the syntax and overall structure of the Terraform language, and coordinates the sequences of changes that must be made for the remote infrastructure to match the given configuration. This general engine has no knowledge of specific types of infrastructure objects. Instead, Terraform uses plugins called providers that define and manage a set of resource types. Most providers are associated with a particular on-premises or cloud infrastructure service, allowing Terraform to manage infrastructure objects within that service. Terraform does not have a platform-independent resource type concept: resources are always linked to a provider, because similar resource characteristics can vary greatly from a provider to a provider. But the Terraform CLI shared configuration engine ensures that the same language and syntax constructs are available in all services and allows resource types from different services to be combined as needed. » Example The following simple example describes a simple network topology for Amazon Web Services, just to give an idea of the overall structure and syntax of the Terraform language. Similar configurations can be created for other virtual network services, using resource types defined by other providers, and a practical network configuration will often contain additional ones not shown here. terraform required_providers of the hashicorp/aws version of required_providers version version > 1.0.4 - variable aws_region - variable base_cidr_block - description - CIDR range definition /16, such as 10.1.0.0/16, which VPC will use default 10.1.0.0/16 - variable availability_zones - description - A list of Availability Zones in which to create subnets type - list(string) ? • var.aws_region - main aws_vpc - - The reference to the base_cidr_block variable allows you to change the network address without changing the settings. cidr_block - var.base_cidr_block - aws_subnet az - Create a subnet for each Availability Zone given var.availability_zones. use one of the specified Availability Zones. availability_zone var.availability_zones[count.index] - When referencing the aws_vpc.main object, Terraform knows that the subnet should only be created after the VPC is aws_vpc vpc_id. Here we create a /20 prefix for each subnet, using consecutive addresses for each Availability Zone, such as 10.1.16.0/20 . cidr_block : cidrsubnet(aws_vpc.main.cidr_block, 4, count.index+1) - For more information about the configuration items shown here, use site navigation to explore Terraform language documentation subsections. To get started, see Configuring Resources. Configuration.