¡Bienvenido!

El presente libro es un intento por describir sistemáticamente las mejores prácticas para el uso de Terraform y proveer de recomendaciones a los problemas más comunes que experimentan sus usuarios.

Terraform es un proyecto relativamente nuevo (como la mayoría de las herramientas DevOps, de hecho) que comenzó en el año 2014.

Terraform es bastante potente (si no es que la herramienta más potente disponible en el mercado actualmente) y una de las herramientas que permiten la administración de infraestructura como código más usadas. Terraform permite a los desarrolladores realizar una gran variedad de cosas pero no los restringe de hacerlas de maneras en las que en un futuro les serán difíciles de integrar o de dar soporte.

Algo de la información descrita tal vez no parezca como parte de las mejores prácticas, lo sé, y para ayudar a los lectores a separar cuáles son las mejores prácticas establecidas y cuáles son sólo otras formas opinadas de hacer las cosas, hago uso de sugerencias para proporcionar algo del contexto e iconos para especificar el nivel de madurez de cada subsección relacionada con las mejores prácticas.

Este libro tuvo su comienzo un día soleado de Madrid del 2018, y se encuentra disponible de manera gratuita en https://www.terraform-best-practices.com/

Años después, ha sido actualizado incluyendo a más de las mejores prácticas disponibles para Terraform 1.0. Eventualmente, este libro debería contener la mayoría de las mejores prácticas y recomendaciones indiscutibles para los usuarios de Terraform.

Patrocinadores

Please contact me if you want to become a sponsor.

Traducciones

Conceptos clave

La documentación oficial de Terraform describe todos los aspectos de la configuración a detalle, y se recomienda leerla con atención para comprender el resto de esta sección.

Recurso

Un recurso *-resource-* es, por ejemplo: `aws_vpc`, `aws_db_instance`, entre otros. Los recursos pertenecen a los proveedores, aceptan argumentos, muestran datos de salida de los atributos, tienen ciclos de vida. Estos pueden ser creados, recuperados, actualizados y eliminados.

Módulo de recurso

Un módulo de recurso *-terraform module-* es una colección de recursos conectados, los cuales en conjunto realizan una acción común (por ejemplo, el módulo de VPC de AWS de Terraform crea una VPC, sus subredes, la NAT Gateway, etc.). Este depende de la configuración del proveedor, el cual puede definirlo en sí mismo o a un nivel superior de estructura (como en un módulo de infraestructura).

Módulo de infraestructura

Un módulo de infraestructura *-infrastructure module-* es una colección de módulos de recurso, los cuales pueden no estar conectados lógicamente pero que en el proyecto, configuración o situación en la que se encuentran, sirven a un mismo propósito.

Por ejemplo, el módulo terraform-aws-atlantis utiliza módulos de recurso,tales como terraform-aws-vpc y terraform-aws-security-group, para administrar la infraestructura requerida en la ejecución de Atlantis sobre AWS Fargate.

Otro ejemplo es el módulo terraform-aws-cloudquery, en el cual múltiples módulos de terraform-aws-modules son utilizados de manera conjunta para administrar la infraestructura, así como el uso de recursos de Docker para el compilado, push y despliegue de imágenes de Docker. Todo en uno.

Composición

Una composición *-composition-* es una colección de módulos de infraestructura, la cual puede abarcar a lo largo de diferentes áreas separadas lógicamente (como Regiones de AWS, varias Cuentas de AWS). La composición es utilizada para describir la totalidad de la infraestructura requerida para todo el proyecto u organización.

La composición consiste de módulos de infraestructura compuestos de módulos de recurso, los cuales implementan recursos individuales.

Composición de infraestructura simple.

Fuente de datos

La fuente de datos -*data source*- realiza operaciones de sólo lectura y depende de la configuración del proveedor. Esta es usada en un módulo de recurso y en un módulo de infraestructura.

La fuente de datos `terraform_remote_state` actúa como un pegamento para módulos y recursos de más alto nivel.

La fuente de datos externa permite a un programa externo actuar como fuente de datos, exponiendo arbitrariamente información en cualquier lugar de la configuración de Terraform. Aquí tenemos un ejemplo del módulo terraform-aws-lambda, en el cual el nombre del archivo es procesado llamando un script externo de Python.

La fuente de datos http realiza una petición HTTP GET al URL dado y exporta la información acerca de la respuesta la cual es útil para obtener información de *endpoints* en los que no existe un proveedor nativo de Terraform.

Estado remoto

-*Remote state*- El estado de las composiciones y módulos de infraestructura debe de persistir en una locación remota la cual pueda ser accedida por otros de forma controlada (ACL, versionamiento, logging).

Proveedor, aprovisionador, etc.

Los proveedores -*providers*-, aprovisionadores -*provisioners*-, y algunos otros términos son descritos a detalle en la documentación oficial, por lo que no hay necesidad de ser repetidos en la presente. En mi opinión, tienen poco que ver con escribir buenos módulos de Terraform.

¿Por qué es tan *difícil*?

Mientras que los recursos individuales funcionan como átomos en la infraestructura, los módulos de recursos son moléculas. Un módulo es la unidad versionada más pequeña y compartible. Esta contiene la lista exacta de argumentos e implementa la lógica básica para que dicha unidad realice la función requerida. Por ejemplo, terraform-aws-security-group crea los recursos `aws_security_group` y `aws_security_group_rule` con base a las entradas -*inputs*-. Este módulo de recurso puede ser utilizado en conjunto con otros módulos para crear un módulo de infraestructura.

El acceso a la información entre moléculas (módulos de recurso e infraestructura) es realizado mediante el uso de salidas -*outputs*- y fuentes de datos -*data sources*-.

El acceso entre composiciones es a menudo realizado utilizando fuentes de datos de estados remotos. Hay múltiples formas de compartir información entre configuraciones.

Cuando disponemos los conceptos descritos anteriormente como pseudorelaciones, pueden lucir a como a continuación se muestran:

```
composition-1 {
  infrastructure-module-1 {
    data-source-1 => d

    resource-module-1 {
      data-source-2 => d2
      resource-1 (d1, d2)
      resource-2 (d2)
    }

    resource-module-2 {
      data-source-3 => d3
      resource-3 (d1, d3)
      resource-4 (d3)
    }
  }
}
```

Estructura del código

Las preguntas relacionas a la estructura del código de Terraform son por mucho las más frecuentes dentro de la comunidad. Probablemente todos en algún punto han pensado acerca de la mejor estructura para el código.

¿Cómo debería de estructurar mis configuraciones de Terraform?

Esta es una de las preguntas en las que existen muchas soluciones y es muy difícil dar un consejo general, así que comencemos por comprender con qué estamos tratando.

- ¿Cuál es la complejidad del proyecto?
  - Número de recursos relacionados.
  - Número de proveedores de Terraform.
- ¿Qué tan seguido cambia la infraestructura?
  - **De** una vez al mes, semana, día.
  - **A** continuamente (cada que hay un nuevo *commit*).
- ¿Cuáles son los iniciadores de cambio del código? *¿Se permite que un servidor CI -continuous integration- actualice el repositorio cuando un nuevo artefacto es compilado*?
  - Sólo los desarrolladores pueden *pushear* al repositorio de infraestructura.
  - Todos pueden proponer cambios mediante una PR *-pull request-* (incluyendo tareas automatizadas corriendo en un servidor CI)
- ¿Qué plataforma o servicio de despliegue se utiliza?
  - AWS CodeDeploy, Kubernetes y OpenShift requieren planteamientos diferentes.
- ¿Cómo son agrupados los entornos?
  - Por entorno, región, proyecto.

> ⓘ Los *proveedores lógicos* trabajan por completo dentro de la propia lógica de Terraform y no interactúan con ningún otro servicio, así que podemos pensar en su complejidad como O(1). Los proveedores lógicos más comunes incluyen random, local, terraform, null, time.

Comenzando con la estructuración de la configuración de Terraform

Poner todo el código en `main.tf` es una buena idea de cuando estás comenzando o escribiendo un código ejemplo. Para cualquier otro caso será mejor tener varios archivos en una separación lógica como a continuación se presenta:

- main.tf - llama a los módulos, locals y data-sources para crear todos los recursos.
- variables.tf - contiene declaraciones de variables utilizadas en el main.tf.
- outputs.tf - contiene outputs de recursos creados en main.tf.
- versions.tf - contiene requerimientos de versión para Terraform y proveedores.

terraform.tfvars no debe ser utilizado en ningún otro lugar más que en la composición.

¿Cómo pensar acerca de la estructura de configuración de Terraform?

> Por favor, asegúrate de comprender los conceptos clave -módulo de recurso, módulo de infraestructura y composición, a como son utilizados en los siguientes ejemplos.

Recomendaciones generales para estructurar el código

- Es más fácil y rápido trabajar con un menor número de recursos.
- `terraform plan` y `terraform apply` realizan llamadas API a la nube para verificar el estatus de los recursos.
- Si tienes toda tu infraestructura en una sola composición, puede tomar varios minutos.
- El radio de afectación *-blast radius-* es más pequeño con menos recursos.
- Aislar recursos no relacionados unos de otros al ponerlos en composiciones separadas reduce el riesgo de que algo salga mal.
- Comienza tu proyecto utilizando estados remotos.
- Tu laptop no es lugar como fuente de la verdad de tu infraestructura.
- Administrar un archivo `tfstate` en git es una pesadilla.
- Luego, cuando las capas de la infraestructura comiencen a crecer en todas direcciones (número de dependencias o recursos) será más fácil mantener las cosas bajo control.
- Practica una estructura y convención de nombrado consistente.
- Como en el código procedural, el código de Terraform debe ser escrito para que, en primera, sea comprensible para las personas. La consistencia ayudará cuando se realicen cambios del día de hoy a dentro de seis meses.
- Es posible mover recursos en el archivo del estado de Terraform, pero podría ser más difícil si realizas un estructurado y nombrado inconsistente.
- Mantén los módulos de recursos tan planos como sea posible.
- No *hardcodees* valores que pueden ser pasados como variables o descubiertos utilizando fuentes de datos *-data sources-*.
- Usa las fuentes de datos y `terraform_remote_state`, específicamente, como pegamento entre los módulos de infraestructura dentro de la composición.

En el presente libro, los ejemplos de proyectos están agrupados por _complejidad -_de pequeñas a muy grandes infraestructuras. Está separación no es restrictiva, así que revisa también por otras infraestructuras.

Orquestación de módulos de infraestructura y composiciones

Tener una infraestructura pequeña significa que también se cuenta con un pequeño número de dependencias así como de recursos. Conforme el proyecto crece, la necesidad de encadenar la ejecución de configuraciones de Terraform, conectando diferentes módulos de infraestructura y pasando valores dentro de una composición, se vuelve visible.

Existen al menos 5 diferentes grupos de soluciones de orquestación que los desarrolladores utilizan:

1. Sólo Terraform - Muy directo, los desarrolladores sólo tienen que saber Terraform para tener el trabajo hecho.
2. Terragrunt - Herramienta de orquestación pura que se puede utilizar para orquestar toda la infraestructura y gestionar las dependencias. Terragrunt opera con módulos y composiciones de infraestructura de manera nativa, lo que reduce la duplicación de código.
3. Scripts propios - A menudo, esto sucede como un punto de partida hacia la orquestación y antes de descubrir Terragrunt.
4. Ansible otra herramienta de automatización de propósito general similar - Usualmente utilizada cuando Terraform es adoptado después de Ansible, o cuando Ansible UI es utilizada activamente.
5. Crossplane y otras soluciones inspiradas en Kubernetes - En algunas ocasiones hace sentido el utilizar el ecosistema de Kubernetes y emplear alguna funcionalidad de reconciliación de bucle para alcanzar el estado deseado de la configuración de Terraform. Ve el vídeo Crossplane vs Terraform para mayor información.

Con esto en mente, estaremos revisando las primeras dos estructuras de proyectos, sólo Terraform y Terragrunt.

Ejemplos de estructura del código

Estructuras de código de Terraform

> Estos ejemplos son presentados con AWS como proveedor pero la mayoría de los principios mostrados en los ejemplos pueden ser aplicados a otros proveedores de nube pública así como a otro tipo de proveedores (DNS, DB, Monitoring, etc).

Tipo	Descripción	Disponibilidad
pequeña	Pocos recursos, sin dependencias externas. Una sola cuenta de AWS. Una sola región. Un sólo entorno.	Disponible
mediana	Varias cuentas y entornos en AWS, módulos estándar de infraestructura utilizando Terraform.	Disponible
grande	Muchas cuentas de AWS, muchas regiones, necesidad urgente de reducir el copiado y pegado, módulos de infraestructura personalizados, uso intensivo de composiciones utilizando Terraform.	TEP (Trabajo en proceso)
muy grande	Varios proveedores (AWS, GCP, Azure). Despliegues multi nube utilizando Terraform.	No Disponible

Estructuras de código de Terraform

Tipo	Descripción	Disponibilidad
mediana	Varias cuentas y entornos de AWS, módulos estándar de infraestructura, patrón de composición con Terragrunt.	No Disponible
grande	Muchas cuentas de AWS, muchas regiones, urgente necesidad de reducir el copiado y pegado, módulos de infraestructura personalizados, uso intensivo de composiciones utilizando Terragrunt.	No Disponible
muy grande	Varios proveedores (AWS, GCP, Azure). Despliegues multi nube utilizando Terragrunt.	No Disponible

Infraestructura de tamaño pequeño con Terraform

Fuente: https://github.com/antonbabenko/terraform-best-practices/tree/master/examples/small-terraform

La presente integra código como ejemplo de la estructuración de la configuración para una infraestructura de tamaño pequeño, en donde no son utilizadas dependencias externas.

- Perfecta para comenzar y refactorizar a medida que avanzas.
- Perfecta para módulos de recurso pequeños.
- Bueno para módulos de infraestructura pequeños y lineares (por ejemplo, terraform-aws-atlantis).
- Bueno para un número pequeño de recursos (menos de 20 o 30).

Contar con un sólo archivo de estado para todos los recursos puede hacer lento el proceso de trabajo con Terraform si el número de recursos está creciendo (considerar el uso del argumento `-target` para limitar el número de recursos).

Infraestructura de tamaño mediano con Terraform

Fuente: https://github.com/antonbabenko/terraform-best-practices/tree/master/examples/medium-terraform

La presente integra código como ejemplo de la estructuración de la configuración para una infraestructura de tamaño mediano que utiliza:

- 2 cuentas de AWS.
- 2 entornos separados (`prod` y `stage`, los cuales no comparten nada). Cada entorno vive en una cuenta separada de AWS.
- Cada entorno utiliza diferentes versiones del módulo estándar de infraestructura (`alb`) proveniente del Registro de Terraform -*Terraform Registry*-.
- Cada entorno hace uso de la misma versión del módulo interno `modules/network` dado que es procedente de un directorio local.

- Perfecta para proyectos en donde la infraestructura está separa lógicamente (cuentas separadas de AWS).
- Buena cuando no hay necesidad de modificar recursos compartidos entre las diferentes cuentas de AWS (un entorno = una cuenta AWS = un archivo de estado).
- Buena cuando no hay necesidad de orquestación de los cambios entre los entornos.
- Buena cuando los recursos de infraestructura son diferentes por entorno intencionalmente y no pueden ser generalizados (por ejemplo, algunos recursos están ausentes en un entorno o en algunas regiones).

A medida de que el proyecto crezca, estos entornos serán más difíciles de mantener actualizados al día unos de otros. Considerar utilizar módulos de infraestructura (estándares o propios) para tareas repetibles.

Infraestructura de tamaño grande con Terraform

Fuente: https://github.com/antonbabenko/terraform-best-practices/tree/master/examples/large-terraform

La presente integra código como ejemplo de la estructuración de la configuración para una infraestructura de tamaño grande que utiliza:

- 2 cuentas de AWS.
- 2 entornos separados (`prod` y `stage` , los cuales no comparten nada). Cada entorno vive en cuentas separadas de AWS.
- Cada entorno utiliza diferentes versiones del módulo estándar de infraestructura (*alb)* proveniente de Registro de Terraform -*Terraform Registry*-.
- Cada entorno hace uso de la misma versión del módulo interno *modules/network* dado que es procedente de un directorio local.

En proyectos grandes como el aquí descrito, los beneficios de utilizar Terragrunt se hace palpable. Revisar Ejemplos de Estructuras de Código con Terragrunt**.**

- Perfecta para proyectos en donde la infraestructura está separa lógicamente (cuentas separadas de AWS).
- Buena cuando no hay necesidad de modificar recursos compartidos entre las diferentes cuentas de AWS (un entorno = una cuenta AWS = un archivo de estado).
- Buena cuando no hay necesidad de orquestación de los cambios entre los entornos.
- Buena cuando los recursos de infraestructura son diferentes por entorno intencionalmente y no pueden ser generalizados (por ejemplo, algunos recursos están ausentes en un entorno o en algunas regiones).

A medida de que el proyecto crezca, estos entornos serán más difíciles de mantener actualizados al día unos de otros. Considerar utilizar módulos de infraestructura (estándares o propios) para tareas repetibles.

Convención del nombrado

Convenciones generales

> ⓘ No debería haber razón para no seguirlas.

> ⓘ Tenga en cuenta que los recursos reales de la nube a menudo tienen restricciones en los nombres permitidos. Algunos recursos, por ejemplo, no pueden contener guiones, algunos deben ser en una convención de mayúsculas y minúsculas conocido como *camel-case*. Las convenciones en este libro se refieren a los nombres propios de Terraform.

1. Utilizar `_` (guión bajo) en lugar de `-` (guión) en todo: nombres de recursos, nombres de fuentes de datos *-data sources-*, nombres de variables, salidas *-outputs-*, etc.
2. Utilizar preferentemente letras y números en letras pequeñas *-lowercased-* (incluso si UTF-8 es soportado).

Argumentos de recurso y fuente de datos

1. No repetir el tipo de recurso en el nombre del recurso (ni de manera parcial, ni completa):

✓
```
resource “aws_route_table” “public” {}
```

⚠
```
resource "aws_route_table" "public_route_table" {}
```

⚠
```
resource "aws_route_table" "public_aws_route_table" {}
```

2. El nombre del recurso debería ser `this` si no hay un nombre más descriptivo o general disponible, o si el módulo del recurso crea un recurso único de este tipo (por ejemplo, en el módulo de VPC de AWS sólo hay un recurso del tipo `aws_nat_gateway` y multiples recursos del tipo `aws_route_table`, así que `aws_nat_gateway` __ debería ser llamado `this` __ y `aws_route_table` __ debería tener nombres más descriptivos como `private`, `public`, `database`).

3. Siempre utilizar sustantivos singulares para los nombres.

4. Utilizar `-` dentro de los valores de los argumentos y en lugares en donde el valor será expuesto a un ser humano (por ejemplo, dentro del nombre del DNS o de una instancia RDS).

5. Incluir el argumento `count`/`for each` dentro del bloque de recursos como primer argumento en la parte superior y separarlo por una nueva línea después de este.

6. Incluir el argumento `tags` __ si es soportado por el recurso como el último argumento real, seguido por `depends_on` y `lifecycle`, si es necesario. Todo esto debería ser separado por una sola línea vacía.

7. Cuando se utiliza un condición en el argumento `count`/`for each`, utilizar un valor booleano si esto tiene sentido, de otra forma, utilizar `length` u otra expresión.

Ejemplos del código de `recurso`

Uso de `count` / `for each`

main.tf

```
resource "aws_route_table" "public" {
  count = 2

  vpc_id = "vpc-12345678"
  # ... remaining arguments omitted
}

resource "aws_route_table" "private" {
  for_each = toset(["one", "two"])

  vpc_id = "vpc-12345678"
  # ... remaining arguments omitted
}
```

main.tf

```
resource "aws_route_table" "public" {
  vpc_id = "vpc-12345678"
  count  = 2

  # ... remaining arguments omitted
}
```

Colocación de `tags` *-etiquetas-*

main.tf

```
resource "aws_nat_gateway" "this" {
  count = 2

  allocation_id = "..."
  subnet_id     = "..."

  tags = {
    Name = "..."
  }

  depends_on = [aws_internet_gateway.this]

  lifecycle {
    create_before_destroy = true
  }
}
```

main.tf

```
resource "aws_nat_gateway" "this" {
  count = 2

  tags = "..."

  depends_on = [aws_internet_gateway.this]

  lifecycle {
    create_before_destroy = true
  }

  allocation_id = "..."
  subnet_id     = "..."
}
```

Condicionales en `count`

outputs.tf

```
resource "aws_nat_gateway" "that" {    # Best
  count = var.create_public_subnets ? 1 : 0
}

resource "aws_nat_gateway" "this" {    # Good
  count = length(var.public_subnets) > 0 ? 1 : 0
}
```

Variables

1. No reinventar la rueda en los módulos de recursos: usar los valores de variables `name`, `description` y `default` como se define en la sección “Referencia de Argumentos” para el recurso en el que estás trabajando.

2. El soporte para la validación en variables es bastante limitado (por ejemplo, no se puede acceder a otras variables o realizar búsquedas). Planificar en consecuencia de esto porque en muchos casos esta característica es inútil.

3. Utilizar la forma plural en un nombre de variable cuando el tipo sea `list (...)` o `map(...)`.

4. Ordenar las claves en un bloque variable como este: `description`, `type`, `default`, `validation`.

5. Incluir siempre una `description` de todas las variables, incluso si cree que es obvio (lo necesitará en el futuro).

6. Procurar usar tipos simples (`number`, `string`, `list(...)`, `map(...)`, `any`) sobre tipos específicos como `object()` a menos que necesite tener restricciones estrictas en cada clave.

7. Utilizar tipos específicos como mapa `map(map(string))` si todos los elementos del mapa tienen el mismo tipo (por ejemplo, `string`) o se pueden convertir a él (por ejemplo, el tipo de `number` se puede convertir en `string`).

8. Utilizar el tipo `any` para deshabilitar la validación de tipos a partir de una cierta profundidad o cuando deban admitirse varios tipos.

9. El valor `{}` a veces es un mapa, pero a veces un objeto. Usar `tomap (...)` para hacer un mapa porque no hay forma de hacer un objeto.

Outputs -*salidas*-

Nombrar las salidas -*outputs*- es importante para hacerlas consistentes y comprensibles fuera de su alcance (cuando el usuario está utilizando un módulo siempre debería de ser obvio qué tipo y atributo del valor es regresado).

1. El nombre de la salida debe describir la propiedad que contiene y ser menos libre de lo que normalmente desearía.

2. Una buena estructura para el nombre de la salida se parece a `{name}_{type}_{attribute}` o `{nombre}_{tipo}_{atributo}` donde:

1. `{name}` es un recurso o fuente de datos -*data sorce*- sin un prefijo de proveedor. `{name}` para `aws_subnet` es `subnet`, para `aws_vpc` es `vpc`.
2. `{type}` es un tipo de fuente de recursos.
3. `{attribute}` es un atributo devuelto por la salida o *output*.
4. Ver ejemplos.

3. Si la salida devuelve un valor con funciones de interpolación y varios recursos, `{name}` y `{type}` deben ser lo más genéricos posible (`this` como prefijo debe omitirse). Ver ejemplo.

4. Si el valor devuelto es una lista, debe tener un nombre en plural. Ver ejemplo.

5. Incluir siempre la `description` de todos las salidas, incluso si se cree que esta es obvia.

6. Evitar establecer argumentos sensibles (`sensitive`)a menos que controle completamente el uso de dicha salida en todos los lugares de todos los módulos.

7. Preferir el uso de `try()` (disponible desde Terraform 0.13) sobre `element(concat(...))` (enfoque heredado de versiones anteriores a la 0.13).

Ejemplos del código de `output` -*salida*-

Devuelve como máximo una ID del grupo de seguridad:

outputs.tf

```
output "security_group_id" {
  description = "The ID of the security group"
  value       = try(aws_security_group.this[0].id, aws_security_group.name_prefix[
}
```

Cuando tenga varios recursos del mismo tipo, `this` debe omitirse en el nombre de la salida:

outputs.tf

```
output "this_security_group_id" {
  description = "The ID of the security group"
  value       = element(concat(coalescelist(aws_security_group.this.*.id, aws_secu
}
```

Utilizar el nombre en plural si el valor devuelto es una lista

outputs.tf

```
output "rds_cluster_instance_endpoints" {
  description = "A list of all cluster instance endpoints"
  value       = aws_rds_cluster_instance.this.*.endpoint
}
```

Estilo del código

> ⓘ
> - Los ejemplos y módulos de Terraform deberían integrar documentación que describa sus funcionalidades y el cómo utilizarlas.
> - Enlaces en el sitio web del Terraform Registry son relevantes y no funcionarán, así que hacer uso de rutas absolutas en el README.md.
> - La documentación puede incluir diagramas creados con mermaid **** y planos creados con cloudfcraft.co.
> - Hacer uso de los **** hooks de pre-commit de Terraform **** para asegurarse de que el código es válido, de que está en el formato adecuado y está documentado adecuadamente antes de que sea *pusheado* a git y revisado por una persona.\

Documentación

Documentación generada automáticamente

pre-commit **** es un marco de trabajo **-***framework*- para administrar y mantener hooks de precommit multi lenguaje. Está escrito en Python y es una herramienta potente para hacer algo de manera automática en la máquina del desarrollador antes de que el código sea *commiteado* al repositorio de git. Normalmente, es utilizado para ejecutar *linters* y formatear código (ver hooks soportados).

Con las configuraciones de Terraform, el `pre-commit` puede ser utilizado para formatear y validar código, así como actualizar documentación.

Se recomienda revisar el repositorio de pre-commit-terraform para la familiarización con el mismo, y con otros repositorios existentes (por ejemplo, terraform-aws-vpc) en donde está siendo utilizado.

@porhacer: Documentar versiones de módulos, liberaciones, GH Actions.

Recursos

1. Página principal de pre-commit.

2. Colección de git hooks para Terraform a ser usados con el framework pre-commit.

3. Publicación de blog por Dean Wilson: hooks de pre-commit y terraform – una red segura para tus repositorios.

Preguntas y respuestas frecuentes

PFT (Problemas Frecuentes de Terraform)

¿Cuáles son las herramientas que debería tener en cuenta y considerar utilizar?

- **Terragrunt** - herramienta de orquestación.
- **tflint** - linter de código.
- **tfenv** - gestor de versiones.
- **Atlantis** - automaticación de Pull Request.
- **pre-commit-terraform** - Colección de hooks de git para Terraform a usar con pre-commit framework.
- **Infracost** - Cloud estima los costes de Terraform mediante un sistema de extraccion y trabaja con Terragrunt, Atlantis y pre-commit-terraform

¿Cuál es la solución al dependency hell *-infierno de las dependencias*- con los módulos?

Las versiones de los recursos y los módulos de la infraestructura deben ser especificados. Los proveedores deben ser configurados fuera de los módulos, solamente en la composición. Las versiones de los proveedores y Terraform también se pueden bloquear.

No hay una herramienta de gestión de dependencias maestra pero, hay varios tips para hacer el *dependency hell* menos problemático. Por ejemplo, **Dependabot** **** puede ser utilizado para automatizar la actualización de dependencias. Dependabot crea *pull requests* para mantener las dependencias seguras y actualizadas al día. Dependabot soporta configuraciones de Terraform.

Referencias

> Hay muchas personas que crean contenido excelente y administran proyectos de código abierto relevantes para la comunidad de Terraform, pero no puedo pensar en la mejor estructura para incluir estos enlaces aquí sin copiar listas como awesome-terraform.

https://twitter.com/antonbabenko/lists/terraform-experts - Lista de personas que trabajan activamente con Terraform y que te pueden decir mucho (si les preguntas).

https://github.com/shuaibiyy/awesome-terraform -Lista seleccionada de recursos en Terraform de HashiCorp.

http://bit.ly/terraform-youtube - Canal de YouTube "Your Weekly Dose of Terraform" de Anton Babenko. Transmisiones en vivo con reseñas, entrevistas, preguntas y respuestas, codificación en vivo y algo de *hacking* con Terraform.

https://weekly.tf/ - Boletín semanal de Terraform. Varias noticias en el mundo de Terraform (proyectos, anuncios, debates) por Anton Babenko.

Escribiendo configuraciones de Terraform

Uso de locales -*locals*- para especificar explícitamente dependencias entre los recursos**

Es una manera útil para dar una pista a Terraform de que algunos recursos deberían de ser borrados antes incluso de que no haya una dependencia directa en las configuraciones de Terraform.

https://raw.githubusercontent.com/antonbabenko/terraform-best-practices/master/snippets/locals.tf

Terraform 0.12 – Argumentos Opcionales vs Requeridos

1. El argumento requerido `index_document` __ debe ser definido si `var.website` no es un `map` vacío.

2. El argumento opcional `error_document` puede ser omitido.

> main.tf
>
> ```
> variable "website" {
>   type    = map(string)
>   default = {}
> }
>
> resource "aws_s3_bucket" "this" {
>   # omitted...
>
>   dynamic "website" {
>     for_each = length(keys(var.website)) == 0 ? [] : [var.website]
>
>     content {
>       index_document = website.value.index_document
>       error_document = lookup(website.value, "error_document", null)
>     }
>   }
> }
> ```

> terraform.tfvars
>
> ```
> website = {
>   index_document = "index.html"
> }
> ```