Trabajo Fin de Máster
Máster en Ingeniería de Telecomunicación

Herramienta para despliegue y gestión de plataformas en la nube

Autor: Alberto Rodríguez de la Cruz
Tutor: Juan Manuel Vozmediano Torres

Dpto. Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Máster
Máster en Ingeniería de Telecomunicación

Herramienta para despliegue y gestión de plataformas en la nube

Autor:
Alberto Rodríguez de la Cruz

Tutor:
Juan Manuel Vozmediano Torres
Profesor titular

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Trabajo Fin de Máster: Herramienta para despliegue y gestión de plataformas en la nube

Autor: Alberto Rodríguez de la Cruz

Tutor: Juan Manuel Vozmediano Torres

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

Este trabajo representa el fin de mis estudios en Ingeniería de Telecomunicación en la Universidad de Sevilla, el cuál representa muchísimo esfuerzo, dedicación y apoyo de mis seres queridos para lograrlos. Por ello quisiera hacer explícito aquí mi agradecimiento:

A mi hermano, que casi consigue terminar su trabajo de fin de máster antes que yo y me ha obligado a ponerme las pilas. Siempre está ahí en los momentos más difíciles.

A mis padres, por su apoyo incondicional y su insistencia para cerrar esta etapa.

A mi madrina, que siempre me recuerda lo mucho que valgo y lo mucho que me quiere.

A Jorge y Minerva, por ser referencias para mí desde siempre.

A "los Piltrafillas", que siempre están ahí cuando los necesitas.

A mis amigos de "William Prime", que me hacen evadirme en la música y me sirven de inspiración constantemente.

A mis amigos y compañeros de la ESI, por los momentos y experiencias compartidas en la escuela y fuera de ella.

Para teminar quiero agradecerle a mi tutor su guía, sus sabios consejos y su paciencia, al igual que al resto de profesores que me han ayudado a convertirme en Ingeniero.

Alberto Rodríguez de la Cruz

Sevilla, 2020

Resumen

El presente documento tiene como objeto la creación de un marco de trabajo y una herramienta que, basándose en *Terraform* y en el paradigma de la Infraestructura como Código, permita resolver alguno de los problemas de la gestión y operación de entornos en la nube. Entre ellos, se encuentra la gestión de múltiples de ellos con distintas características que comparten un mismo código, la operación compartida entre varios administradores ó el mantenimiento del estado de cada despliegue bajo control de versiones.

Para lograrlo, se propone una metodología de uso y una organización del código de *Terraform* que describe la infraestructura a desplegar. Basándose en ellas, se desarrolla la herramienta *Sonatina*, implementando una línea de comandos que permite gestionar múltiples despliegues con capacidad de personalización mediante variables ó complementos, y manteniendo el estado de cada uno bajo el sistema de control de versiones GIT.Todo ello en forma de software con licencia *Apache License 2.0*, para permitir a las compañías su uso comercial de forma libre.

El proceso de desarrollo de la herramienta se divide en dos fases. La primera es la de diseño, en la que se identifican los requisitos del software y los paquetes que lo componen, siendo cada uno responsable de aportar una serie de funcionalidades. En la segunda, se realiza la implementación de estos componentes, basándose en una serie de librerías de código abierto muy utilizadas en la industria, como *cobra*, *afero* u *go-git*.

Finalmente se ejemplifica el uso de *Sonatina* con el caso de uso de una empresa de *hosting*, en el que se gestionan los despliegues de desarrollo y producción de servidores *WordPress* dedicados por cliente, que comparten base de datos, todo ello sobre un clúster de *Kubernetes*. Con ello se demuestran las capacidades y posibilidades de la herramienta, que puede ser aplicada también a casos más complejos.

Abstract

The objective of this document is the creation of a framework and a tool that, based on *Terraform* and the Infrastructure as Code paradigm, will be able to solve some of the cloud environment management and operation problems. Some examples are the management of multiple of them defined by the same code, shared operation between several administrators, or maintain the deployment state under version control.

To achieve these goals, it's proposed a methodology and a structure for the *Terraform code* that describes the infrastructure to be deployed. Base on them, a tool called *Sonatina* is developed, which implements a command-line interface that allows the management of multiple deployments, with the possibility of customizing them using variables and plugins and maintaining its states under version control. Everything is based on the *Apache License 2.0*, to allow its use by the companies with freedom.

The tool development process is divided into a first design phase, where software requirements are identified. The packages that compose the tools are also identified, being each of them responsible of implement some features. In a second phase, the component implementation is realized, based on a set of libraries very used in the industry, like *cobra*, *afero* or *go-git*.

Finally, the case use of a hosting company is presented as an example. It consists of the management of a set of *WordPress* servers dedicated per client that uses a common database, all on a *Kubernetes* cluster. This shows the capabilities and possibilities of the tool, which can be also applied to more complex cases.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

1.1 Motivación

Con la llegada de los servicios en la nube, la forma de desplegar, gestionar y operar infraestructuras y plataformas ha cambiado, y sigue evolucionando actualmente. Las compañías buscan cambiar, introducir novedades y mejorar los servicios de forma rápida, sin interrupciones y con garantías de que los cambios funcionan correctamente y no provocan errores. Es un problema complejo, que se va haciendo más difícil a medida que el servicio crece y avanza en su ciclo de vida.

En este contexto, surge la filosofía *DevOps* [1], que busca romper con el ya clásico ciclo de desarrollo y puesta en producción del software en el que los desarrolladores únicamente se centran en la funcionalidad y los operadores en su puesta en producción y mantenimiento. Tradicionalmente se recurría a esta forma de desplegar el software porque los procesos eran muy diferentes, y la operación requería de intervención física al tener que tratar con servidores y redes propias de cada organización. La llegada de la nube ha cambiado este paradigma, permitiendo que la operación ahora pueda realizarse de forma similar al desarrollo del software, gracias a las interfaces telemáticas que se ofrecen para administrar infraestructuras.

Como consecuencia de esto, cada vez más se busca que los desarrolladores participen en todo el proceso de puesta en producción del software, abarcando también la fase de despliegue y operaciones y propiciado la aparición numerosas herramientas que acercan el proceso de operación al de desarrollo: las herramientas de Infraestructura como Código (IaC).

Si bien este tipo de herramientas permite gestionar muy bien despliegues concretos de infraestructuras y servicios, las empresas además suelen necesitar desarrollar utilidades de gestión que les permitan replicar esos entornos y adaptarlos a las necesidades de los distintos escenarios y casos de uso. Como ejemplo, el autor de este trabajo ha participado en el desarrollo de utilidades de este tipo para el caso de una plataforma de Big Data y Análisis de datos, que necesitaba ser desplegada en entornos de clientes, ajustando sus recursos, funcionalidades y configuración para cada escenario.

La motivación de este trabajo, basándose en la experiencia del autor, es la de evitar a las empresas la necesidad de desarrollar utilidades específicas para poder gestionar sus infraestructuras en los múltiples escenarios que tratan. Para ello se pretende desarrollar un marco de trabajo y una herramienta que permita abordar este tipo de necesidades de una forma genérica y reutilizable por cualquier usuario y compañía. Además, se presenta en forma de software de código abierto, para facilitar su adopción y nutrirse de la experiencia de otras empresas que necesidades similares que puedan enriquecer la herramienta y hacerla más útil y potente para todos sus usuarios.

1.2 Objetivos

El primer objetivo principal de este trabajo es definir una metodología para utilizar una de las principales herramientas de infraestructura como código en la actualidad: *Terraform*. Con ella se busca:

- Gestionar múltiples despliegues basados en una misma definición de la infraestructura, permitiendo adaptarlos con distintas configuraciones e incluso, con adaptaciones para distintos proveedores de servicios.
- Mantener un control de versiones basado en GIT del estado y configuración de la infraestructura, y no sólo de su código.
- Permitir el desarrollo de complementos que puedan añadir o modificar funcionalidades a una infraestructura, sin modificar el código de la infraestructura base.
- Ofrecer la capacidad de definir una serie de sabores o tamaños para la infraestructura, de forma que se puedan realizar despliegues similares con distintos niveles de recursos asignados.
- Facilitar la gestión colaborativa de las infraestructuras por parte de distintos administradores.
- Organizar el código de una forma concreta, ayudando así a los desarrolladores a trabajar siguiendo una serie de convenciones y permitiendo el desarrollo de herramientas que implementen los flujos de la metodología.

El segundo objetivo principal es la implementación de una herramienta como prueba de concepto que facilite a los desarrolladores la adopción de la metodología automatizando sus flujos, ofreciendo una interfaz en línea de comandos que implemente todas las operaciones de gestión necesarias.

Además, tiene un objetivo académico, ya que se se ponen en práctica conocimientos sobre gestión de infraestructuras y servicios en la nube, uso de herramientas como GIT, o desarrollo utilizando el lenguaje de programación *Golang*.

1.3 Plan de trabajo

El proceso de elaboración del trabajo se divide en varias fases, que se describen a continuación:

1. **Planteamiento de objetivos**: se proponen y valoran una serie de objetivos para el proyecto, siendo validados por el tutor.
2. **Investigación y documentación**: se exploran las alternativas existentes y se recopila la documentación necesaria para comenzar a abordar el trabajo.
3. **Diseño del marco de trabajo y de la herramienta**: basado en la experiencia del autor y la documentación recopilada, se aborda el diseño de estos elementos que serán el resultado del trabajo.
4. **Desarrollo de la herramienta**: se aborda la implementación de la utilidad, siguiendo el diseño y cumpliendo los requisitos identificados.
5. **Elaboración de la memoria**: una vez elaborada la parte técnica del trabajo, se procede a la documentación de todo el proceso en esta memoria.
6. **Revisión global**: se pulen los detalles del documento en un proceso de revisión con el tutor.

1.4 Entorno y medios de desarrollo

El grueso del desarrollo del trabajo se divide en la implementación de la herramienta *Sonatina*, que es un proyecto software en *Golang*, y la elaboración de este documento. Para abordarlo, se utilizan una serie de equipos, aplicaciones y servicios, los cuáles se detallan en los siguientes apartados.

1.4.1 Medios materiales

Para elaborar tanto el software como la memoria de este trabajo, se ha utilizado un ordenador con un sistema operativo basado en UNIX (Mac OS X concretamente). Si bien la herramienta resultante está diseñada para gestionar entornos de nube, para su desarrollo no ha sido necesario disponer de este tipo de entornos al poder utilizar recursos locales para la realización de las pruebas, reduciendo así los costes de implementación.

También se ha requerido de una conexión a Internet para la búsqueda de documentación y la obtención del software de desarrollo necesario.

1.4.2 Software

Para el desarrollo de la herramienta, se han utilizado las siguientes aplicaciones:

- Editor de código Visual Studio Code [2], con complementos para el desarrollo de software en *Golang* [3] y *Terraform* [4].
- *GIT* [5] para el mantenimiento bajo control de versiones del código desarrollado.
- *Docker* [6] y *Kubernetes* [7], para la realización de pruebas locales.
- La consola de línea de comandos *iTerm* [8] junto con la shell *Fish* [9], para la ejecución de las pruebas de la herramienta desarrollada.
- El navegador *Google Chrome*, para la búsqueda de documentación.
- La herramienta ofimática *Microsoft Word*, para la elaboración de este documento.

1.4.3 Servicios

Para la publicación y mantenimiento del software desarrollado se han empleado los siguientes servicios con capas gratuitas para proyectos de código abierto:

- *GitHub* [10]: repositorio público de software basado en GIT. Se ha utilizado para publicar la herramienta y organizar algunas de sus tareas de desarrollo.
- *GitHub Actions* [11]: servicio de integración continua utilizado para lanzar un conjunto de pruebas automáticas cada vez que se subían cambios al repositorio en *GitHub*.
- *CodeCov* [12]: servicio de análisis de cobertura de pruebas, con generación automática de informes asociados a cada conjunto de cambios en el respositorio.
- *Go Report Card* [13]: servicio de análisis estático de código, para detectar posibles problemas sin necesidad de ejecutarlo.
- *Go Doc* [14]: servicio de generación automática de documentación basada en comentarios en el código *Golang*.

2 ANTECEDENTES

Este trabajo pertenece al ámbito de la Infraestructura como Código (IaC), que trata de abordar el problema del despliegue, gestión y operación de infraestructuras en la nube siguiendo una metodología y un ciclo de vida similar al del software.

Para lograrlo, estas herramientas se basan en la automatización programática de las tareas de gestión, de forma que se hace una equivalencia entre el estado de la infraestructura y el código que la gestiona. Conforme evoluciona el código, también evoluciona la infraestructura que se despliega con él.

En este capítulo se ofrece una panorámica y una clasificación de las herramientas existentes en este ámbito, junto con otras utilidades que añaden funcionalidades a las primeras, como será el caso de la que se desarrolla en este trabajo. Ésta se basa en la herramienta de IaC llamada *Terraform*, cuyas características y funcionalidades principales son explicadas al final de este capítulo con el objetivo de facilitar la comprensión de los conceptos expuestos en el resto del trabajo.

2.1 Criterios de clasificación de herramientas de IaC

Se puede realizar una categorización en base a las siguientes características:

- **Aplicabilidad**
- **Lenguaje**
- **Descripción**
- **Recurso principal**
- **Arquitectura**

2.1.1 Aplicabilidad

Las herramientas pueden tener una aplicabilidad genérica o específica en función de si son válidas para cualquier tipo de infraestructura o proveedor de servicio, o por el contrario únicamente soportan algunos en concreto.

2.1.2 Lenguaje

Para describir la infraestructura o las operaciones a realizar, se pueden utilizar distintos lenguajes de programación. Normalmente son de alto nivel ó scripting, como *Python* [15] o *Ruby* [16]. También se suelen utilizar formatos de representación de datos como *JSON* [17] o *YAML* [18].

2.1.3 Descripción

En función de cómo se realice la descripción de la infraestructura a desplegar, se pueden clasificar las herramientas en **declarativas** o **imperativas**.

La diferencia clave entre ambos tipos de herramientas es que en las declarativas se indica **qué** se quiere desplegar, mientras que en las imperativas se indica **cómo** hacerlo.

La principal ventaja de las primeras es la facilidad para conocer el estado deseado de la infraestructura, puesto que está siendo descrita en el código directamente. Además, facilita el mantenimiento de un registro del estado, de forma que se pueda comparar lo que está descrito con lo que hay desplegado en un momento concreto.

Ésto no es posible con un lenguaje imperativo, debido a que obliga a especificar qué operaciones realizar en cada caso. Normalmente, lo deseable en este tipo herramientas es conseguir que las operaciones sean idempotentes, ya que en todo momento hay que especificar cuáles son las acciones a realizar, como en un script.

2.1.4 Recurso principal

En las herramientas que se expondrán posteriormente, se gestionan tres tipos de recursos:

- **Infraestructura**: recursos de proveedores de servicios de nube.
- **Configuración**: de servicios, normalmente desplegados en servidores.
- **Contenedores**: paquetes de software y configuración que se despliegan en servidores.

Normalmente cada utilidad se especializa en uno de estos tipos de recursos, aunque suelen presentar funcionalidades relacionadas con los tres.

2.1.5 Arquitectura

Las herramientas más orientadas a la configuración suelen utilizar un servidor que gestiona configuraciones y controles de acceso de forma centralizada. Es habitual que los sistemas gestionados tengan un agente asociado que se comunica con el servidor y realiza las operaciones necesarias sobre el recurso. Siguen un paradigma Cliente/Servidor (C/S).

Las que siguen un paradigma de sólo cliente (C), ejecutan sus operaciones directamente en el equipo del administrador, por lo que no suelen disponer de control de acceso propio ni un repositorio central de configuraciones o descripciones de infraestructura. A cambio, no tienen la necesidad de tener un servidor en ejecución para funcionar.

2.2 Clasificación e introducción a las principales herramientas de IaC

En la tabla 1 se clasifican en función de las características anteriores las principales herramientas de IaC de la actualidad.

Tabla 1 Clasificación de herramientas de IaC

Herramienta	Lenguaje	Aplicabilidad	Descripción	Recurso principal	Arquitectura
Chef	Ruby	Genérica	Imperativa	Configuración	C/S
Ansible	YAML	Genérica	Imperativa	Configuración	C
AWS CloudFormation	JSON/YAML	Sólo AWS	Declarativa	Infraestructura	C/S
OpenStack Heat	JSON	Sólo OpenStack	Declarativa	Infraestructura	C/S
Terraform	HCL	Genérica	Declarativa	Infraestructura	C
Pulumi	Varios	Genérica	Imperativa	Infraestructura	C
Kubernetes	JSON/YAML	Genérica	Declarativa	Contenedores	C/S
Docker Swarm	JSON/YAML	Genérica	Declarativa	Contenedores	C/S

Se realiza a continuación una breve descripción de cada una de las herramientas:

- ***Chef*** [19]: herramienta basada en el lenguaje de programación *Ruby* [16], que es un lenguaje procedimental muy utilizado en scripts. Su principal objetivo es la configuración como código, ya que está pensado para gestionar configuraciones de forma centralizada en su servidor, y aplicarlas en los distintos elementos de la infraestructura con agentes
- ***Puppet*** [20]: herramienta similar a *Chef*, que también está basada en *Ruby.*
- ***Ansible*** [21]: herramienta sin agente, que permite el despliegue y configuración de recursos desde el propio equipo del administrador. Se basa en ficheros YAML para especificar los pasos de despliegue. Al igual que *Chef* y *Puppet*, su principal objetivo es la configuración, aunque también permite desplegar y gestionar infraestructuras a más bajo nivel.
- ***AWS CloudFormation*** [22]: servicio ofrecido por *AWS* para el despliegue de los recursos de su nube. Al contrario que las herramientas anteriores, esta se basa en una descripción declarativa, en la que, en lugar de indicar los pasos a ejecutar, se especifica únicamente el estado deseado de la infraestructura, dejando que el servicio se encargue de realizar las operaciones necesarias para alcanzarlo.
- ***OpenStack Heat*** [23]: Equivalente a *CloudFormation* en *OpenStack*.
- ***Terraform*** [24]: herramienta sin agente y declarativa, que utiliza el lenguaje HCL [25] para describir el estado deseado de la infraestructura. No está ligada a un proveedor de servicios como sí lo están las demás herramientas con descripción declarativa ya expuestas, permitiendo añadir complementos para soportar diferentes proveedores de servicios.
- ***Pulumi*** [26]: librería que facilita la creación de aplicaciones de gestión de infraestructuras y configuraciones utilizando lenguajes de programación genéricos como *Python*, *Ruby*, *Golang*, etc.
- ***Kubernetes*** [7]: orquestador de contenedores que dispone de funcionalidades relacionadas con IaC al permitir describir la infraestructura de contenedores mediante ficheros JSON o YAML. También ofrece muchas más capacidades de gestión tanto a nivel de red como de organización de recursos que están fuera de los objetivos generales de las herramientas de IaC.
- ***Docker Swarm*** [6]: alternativa similar a *Kubernetes*, pero centrada únicamente en la tecnología de contenedores *Docker*.

Con el auge de los contenedores, liderado por *Docker*, la industria busca la inmutabilidad[1] en los servicios que despliega, con el objetivo de evitar que la infraestructura se comporte de forma distinta a la desarrollada originalmente debido a la acumulación de cambios durante su vida en producción. Esto ha hecho que comiencen a perder popularidad las herramientas más centradas en la configuración como código, transfiriéndosela a orquestadores de contenedores como *Docker Swarm* [6] ó, sobre todo, *Kubernetes* [7], ya mencionados anteriormente.

2.3 Utilidades construídas sobre las herramientas de IaC

Sobre el ecosistema de utilidades de IaC, se han construido utilidades que se basan en las herramientas anteriores, y que añaden funcionalidades adicionales a nivel de productificación e integración con procesos y aplicaciones empresariales. Los principales exponentes de estas herramientas son las siguientes:

- ***Terragrunt*** [27]: se basa en *Terraform* y le aporta nuevas funcionalidades. Entre ellas se encuentra la posibilidad de unificar el código de varios entornos en un único repositorio, la gestión automatizada de su estado, o la posibilidad de realizar ejecuciones paralelas de *Terraform* para varios entornos.
- ***Troposphere*** [28]: es una librería que permite generar código de *AWS CloudFormation* u *OpenStack Heat* a partir de un programa en *Python*. Esto permite desarrollar aplicaciones en este lenguaje que genere y aplique descripciones de infraestructura dinámicamente para estos dos proveedores.

[1] La inmutabilidad consiste en utilizar paquetes de software y configuración que no puedan cambiar, siendo necesario redesplegar completamente el software para aplicar cambios. En el artículo [56] se explican las ventajas de la inmutabilidad.

- ***Ansible-runner-service*** [29]: es un proyecto que genera una API REST para el uso de *Ansible* desde aplicaciones que no estén desarrolladas en *Python*, y para la centralización de la ejecución de su código en un servidor controlado por la organización.

2.4 Terraform

Este trabajo, al igual que *Terragrunt*, se basa en *Terraform* para desarrollar un marco de trabajo y una utilidad que aporta nuevas funcionalidades, por lo que a continuación se pretende aclarar algunos conceptos sobre esta herramienta.

Terraform permite utilizar un lenguaje declarativo en el que describir el estado deseado de la infraestructura. Las infraestructuras están compuestas de **recursos**, cada uno de los cuales describe un elemento con unas características concretas. A continuación, se muestra un ejemplo de declaración de una instancia de AWS:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

Código 1 Ejemplo de recurso de Terraform

Se puede observar en el ejemplo que no se indica en ningún momento qué operación realizar para gestionar el recurso, sólo se describe el estado deseado. *Terraform* permite hacer con esta descripción dos operaciones básicas: **aplicarla**, o **destruirla**. En el primer caso, se ejecutarían las acciones necesarias para alcanzar el estado descrito, ya sea creando el recurso o actualizándolo si ya existía. En el segundo, se eliminaría el recurso.

Para decidir qué operación realizar sobre cada recurso, *Terraform* mantiene un fichero con un registro del **estado** actual de la infraestructura. Esto le permite analizar las diferencias entre el estado actual y el deseado para determinar qué acciones son necesarias. Si, por ejemplo, el estado registrado coincide con el descrita en el código, no se realizará ninguna intervención al haberse alcancado ya el estado deseado. Por defecto se almacena en un fichero local, pero también soporta otros sistemas compartidos[2], como bases de datos SQL o sistemas de almacenamiento de objetos como *Amazon S3* [30].

Se soportan todo tipo de recursos, cuya gestión es implementada mediante complementos llamados **proveedores**. Cada uno de ellos implementa las operaciones necesarias para gestionar un conjunto de recursos, pudiéndose usar tantos como sean necesarios para describir la infraestructura, otorgándole una gran versatilidad a *Terraform*.

Otra de sus características principales es la posibilidad de definir **variables** que permitan la parametrización de características de los recursos. Para ello hace uso del lenguaje HCL[3], que ofrece además otras funcionalidades sintácticas como bucles o expresiones condicionales. Se pueden especificar en la ejecución de *Terraform* tanto directamente en el comando como referenciando uno o múltiples ficheros. En este último caso, se realizaría la combinación sobreescribiendo aquellas que hubieran por algún método anterior.

Finalmente, *Terraform* permite agrupar recursos en **módulos**, que pueden ser reutilizados y que también aceptan variables tanto de entrada como de salida. A su vez, estos módulos pueden llamar a otros, de forma que el directorio donde se ejecuta *Terraform* se convierte en el módulo principal (como la función *main* en un lenguaje de programación) que se encarga de ir llamando a los demás.

[2] En *Terraform* los sistemas de almacenamiento y compartición de estados se identifican con el término *backend* [34].
[3] HCL es el lenguaje de configuración de Hashicorp, del inglés *Hashicorp Configuration Language* [25].

3 ESPECIFICACIÓN DEL MARCO DE TRABAJO

En el capítulo anterior se presentó una panorámica de las principales herramientas de Infraestructura como Código (IaC) que se utilizan actualmente en la industria. También se introdujeron algunas utilidades que, basándose en las primeras, añadían funcionalidades o cambiaban su forma de uso.

En éste, se propone una metodología bien definida para trabajar con *Terraform*, sobre la cuál construir un marco de trabajo y una herramienta que la facilite.

La elección de *Terraform* se debe a que es la única que cumple con las siguientes características:

- Utiliza **descripciones declarativas**.
- Es de **aplicabilidad genérica**, pudiendo gestionar infraestructuras de múltiples proveedores.
- Es de **código abierto**.

Terraform es una herramienta flexible que permite al desarrollador tomar la decisión de cómo estructurar e implementar su código, sin dar una guía concreta de cómo realizar el desarrollo. Si bien esta característica lo hace una herramienta muy versátil, también hace más difícil empezar a trabajar con ella y organizar sus recursos de una forma funcional, dificultando su mantenimiento.

En el marco de trabajo propuesto se fuerza al desarrollador a seguir unas pautas más estrictas, que lo guían en la forma de organizar e implementar el código. Esto lo hace más sencillo de seguir y desarrollar, a costa de una menor flexibilidad. También facilita la creación de otras herramientas que asumen el seguimiento de unas reglas concretas.

3.1 Organización del código de *Terraform* (HCL)

Este marco de trabajo define de organizar el código que describe la infraestructura, con los siguientes objetivos:

- Maximizar la **reutilización** de código.
- Soportar **complementos** que personalicen un despliegue concreto.
- Soportar **versiones** de la infraestructura con asignaciones de recursos distintas, llamados **sabores**.
- **Separar** el código base del producto a desplegar de la configuración específica de cada despliegue.
- Soporte para infraestructuras **multi-inquilino**.

3.1.1 Multi-inquilino

En general, el concepto "Multi-inquilino" se utiliza en el mundo del software para indicar que una misma aplicación es capaz de dar servicio a varios clientes de forma independiente. En este marco de trabajo se utiliza para reflejar que en un mismo despliegue se pueden administrar partes de la infraestructura dedicadas a un único cliente, mientras que otras partes se comparten entre todos ellos.

De aquí surge el concepto de **Componente**, que básicamente es un conjunto de elementos de la infraestructura. Puede ser de dos tipos:

- **Global**: es la parte de la infraestructura común a todos los clientes. Por ejemplo, un servidor de base de datos compartido.
- **De Usuario**: es una parte de la infraestructura que se despliega de forma exclusiva para cada cliente. Por ejemplo, un servidor web dedicado. Puede utilizar elementos de la infraestructura global, como la base de datos compartida mencionada en el ejemplo anterior.

Cada componente estará vinculado a una ejecución de *Terraform*, que a su vez tendrá asociado su propio estado. En los siguientes apartados se hará referencia a cómo se relaciona este concepto con otros explicados más adelante.

3.1.2 CTD: Definición del árbol de código

El código HCL se organiza en árboles de directorios llamados CTD (*Code Tree Definition*), y cuya estructura se representa en la Figura 1.

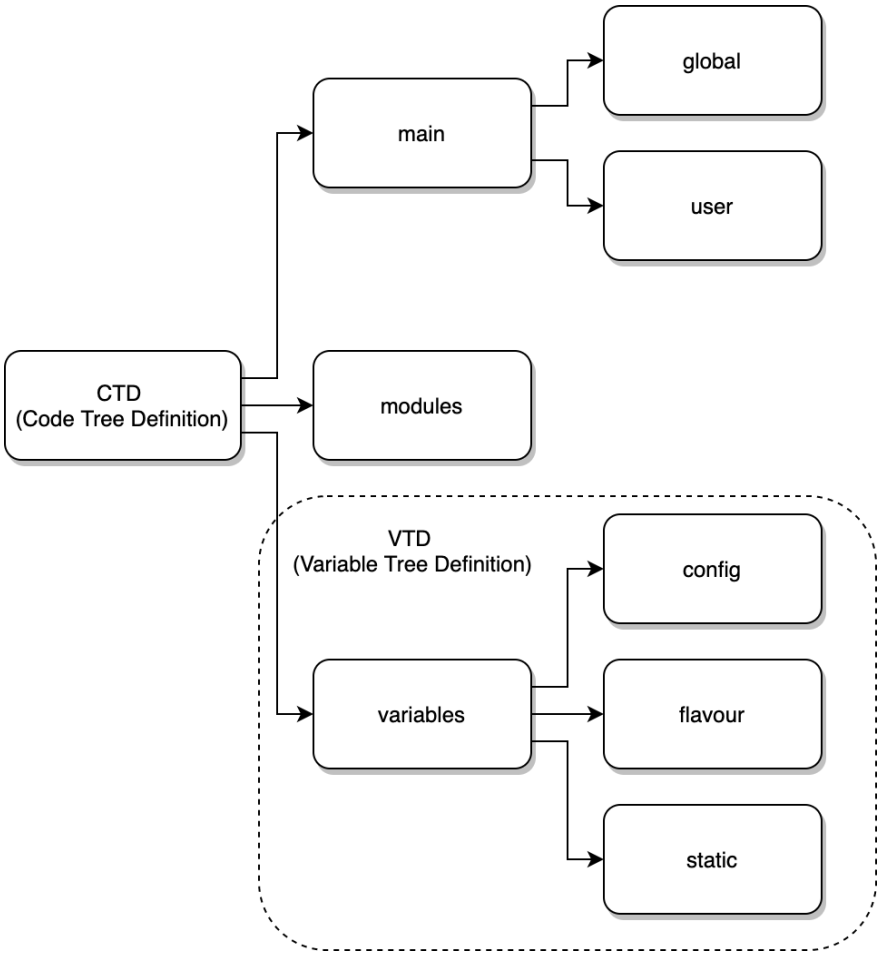

Figura 1 CTD: Code Tree Definition

El CTD está compuesto de los siguientes directorios:

- **Principal** (*Main*)
- **Módulos** (*Modules*)
- **Variables** (*VTD*)

3.1.2.1 Principal

El directorio principal está compuesto a su vez por dos directorios, uno para el componente global y otro para el de usuario. En el primero se sitúan los ficheros de *Terraform* que describen la infraestructura común a todos los clientes, mientras que en el segundo se sitúan los ficheros que describen la parte de la infraestructura que es

única para cada usuario. Gracias a que *Terraform* aplica la configuración de todos los ficheros del directorio, se puede organizar el código en múltiples ficheros, mejorando su modularidad. Además, esta característica será explotada posteriormente con los complementos.

Se debe evitar el uso de recursos definidos directamente en el módulo principal, dejándolo exclusivamente para llamar a módulos del directorio de módulos. Con esto se consigue aislar el desarrollo de los módulos, que se pueden organizar de forma arbitraria, del esquema estructural que impone el marco de trabajo en el módulo principal.

3.1.2.2 Módulos

Este directorio contiene todos los módulos de *Terraform* que definen los recursos de la infraestructura. En este caso no hay distinción por componentes globales o de usuario, ya que los módulos pueden ser referenciados desde el directorio principal de cada uno de los componentes, independientemente de su tipo.

La separación entre directorio principal y el resto módulos permite maximizar la compatibilidad con módulos ya existentes de *Terraform*, ya que todo código que se debe adaptar al marco de trabajo se encuentra en el primero.

3.1.2.3 VTD: Definición del árbol de variables

El directorio de variables se estructura para permitir la clasificación de las variables en tres tipos:

- **De configuración**: variables que puede configurar directamente el usuario que despliega la infraestructura (Por ejemplo: el nombre del servidor web a desplegar).
- **De sabor**: variables que configuran el tamaño y los recursos asignados a la infraestructura. Típicamente, se configuran aquí el tamaño y número de máquinas, la CPU, RAM y disco asignado a cada servicio, etc. Se permite definir varios sabores para poder desplegar una misma infraestructura con diferentes tamaños.
- **Estáticos**: valores asociados a una versión concreta de la infraestructura, como por ejemplo la versión del software de un servidor.

En cada uno de los subdirectorios del árbol debe haber un fichero/directorio con las variables para el despliege global, y otro para las del despliegue por cliente, tal y como se observa en la Figura 2.

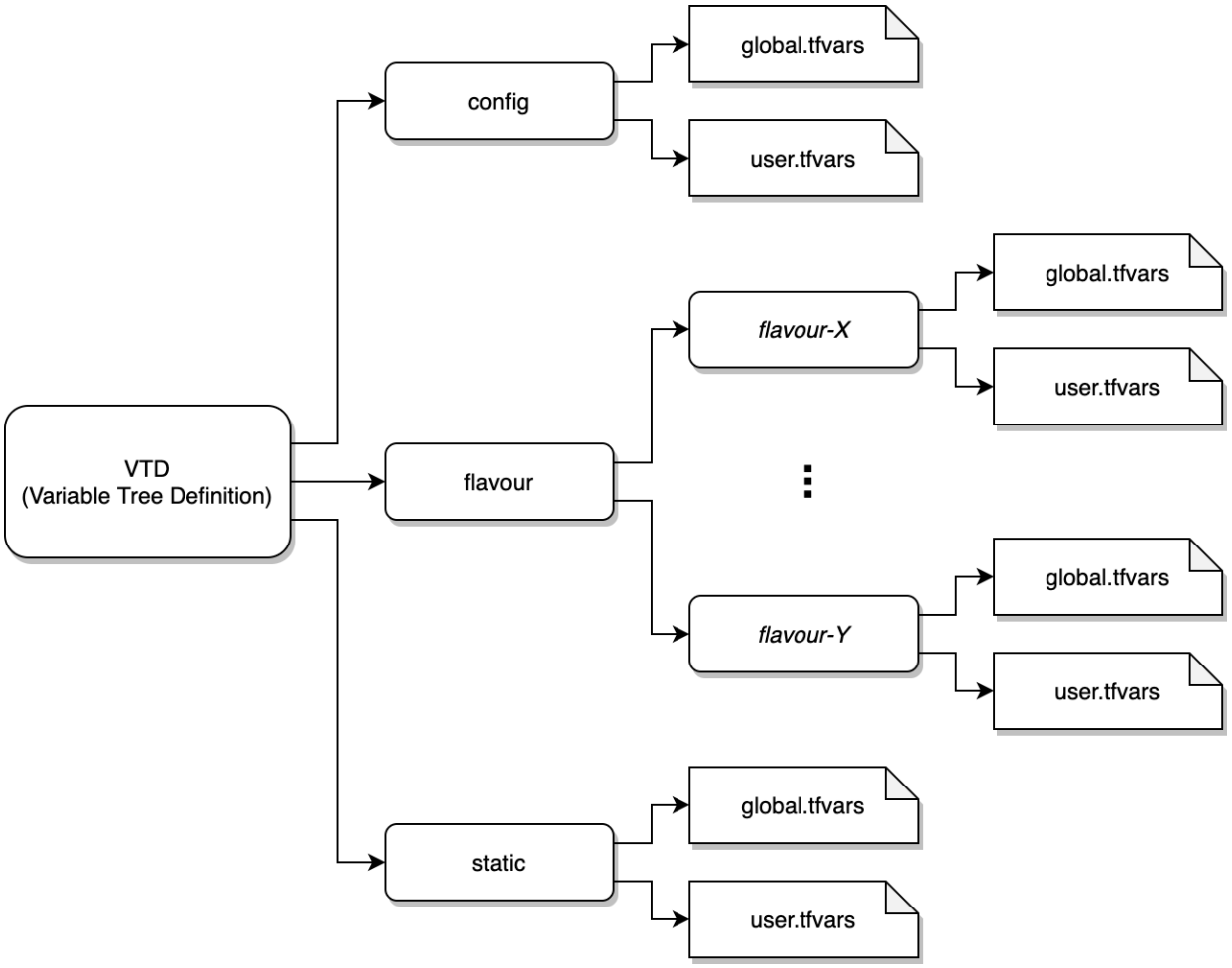

Figura 2 VTD: Variable Tree Definition

Es importante aclarar que el desarrollador del código es quien decide qué variables son de cada tipo. Normalmente, hay diferentes conjuntos de variables que encajan bien con los tipos aquí definidos, pero es responsabilidad del programador decidir cómo organizarlas en ellos.

3.1.3 Complementos

Uno de los objetivos propuestos es el soporte para realizar adaptaciones en un despliegue concreto, con el fin de adaptarla a una determinado proveedor o plataforma, o simplemente añadir nuevas funcionalidades. Para ello se introduce el concepto de complemento, que consiste en un CTD que se combina con el CTD base de la infraestructura, añadiendo o modificando elementos de la misma.

Para realizar la combinación de los CTDs se deben seguir los siguientes pasos:

1. Copiar el CTD base a un directorio de trabajo temporal.
2. Copiar los módulos del complemento al directorio de trabajo. Si se definen módulos con el mismo nombre, se sobreescribirán, teniendo prioridad los del complemento añadido más recientemente.
3. Copiar los ficheros del directorio principal del complemento al directorio de trabajo, juntos con los del base. Si hay ficheros con el mismo nombre, se sobreescriben (el de base tiene menos prioridad). Esto permite que el complemento modifique recursos definidos en el código base.

Los ficheros de variables tienen un tratamiento especial, que se explicará en el apartado 3.2. Los pasos 2 y 3 se realizan para todos los complementos que se quieran añadir. El orden importa, ya que si hay ficheros con el mismo nombre tendrán prioridad los del complemento añadido más recientemente.

Ejemplo: se tiene un CTD base con los ficheros representados en la Figura 3 y un CTD de un complemento

que se muestra en la Figura 4.

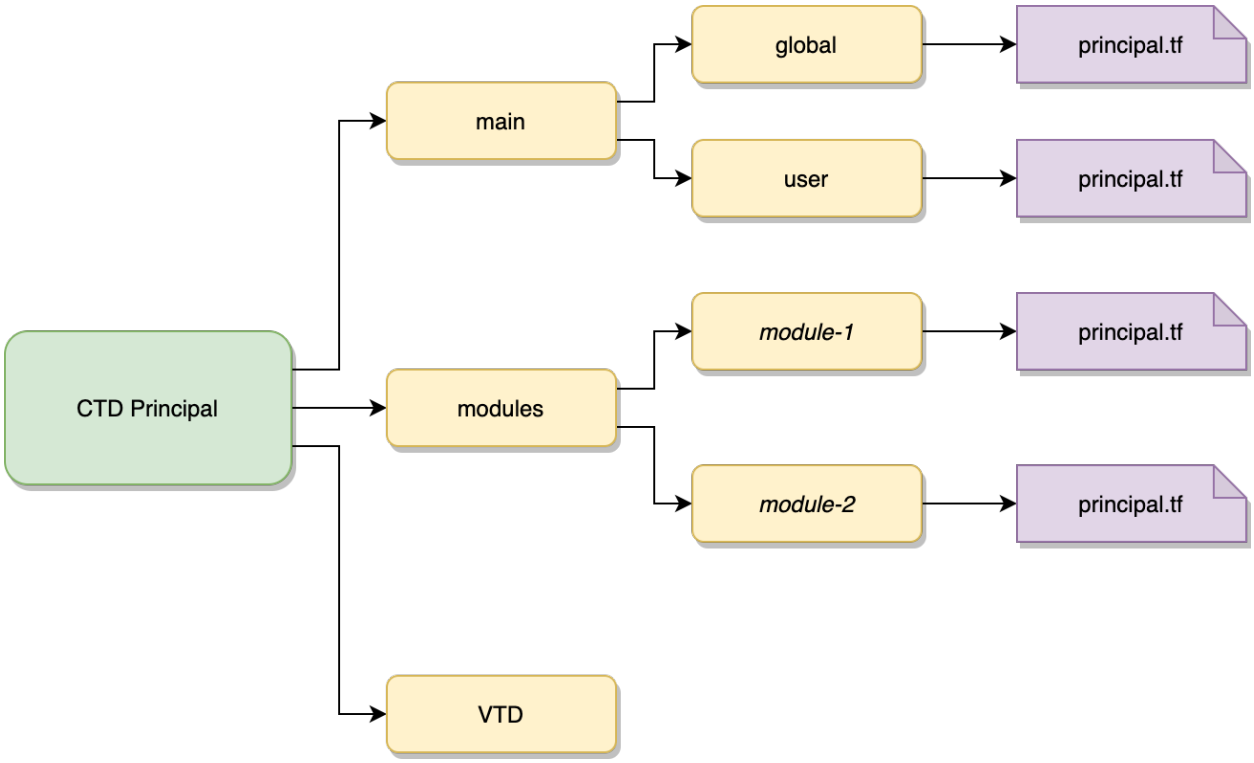

Figura 3 CTD principal de ejemplo

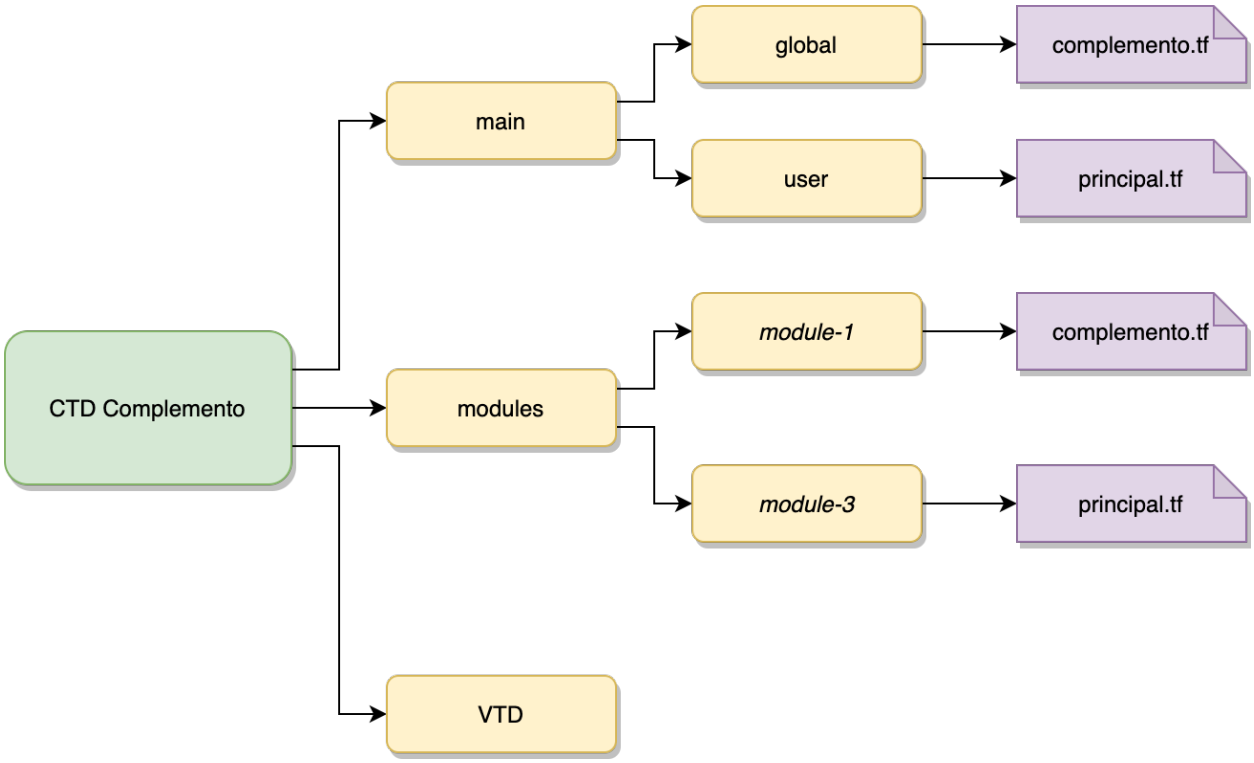

Figura 4 CTD del complemento de ejemplo

En primer lugar, se quiere desplegar la infraestructura con el complemento para el componente global. En este caso, siguiendo los pasos explicados anteriormente, se obtendría en el directorio de trabajo la estructura de la Figura 5.

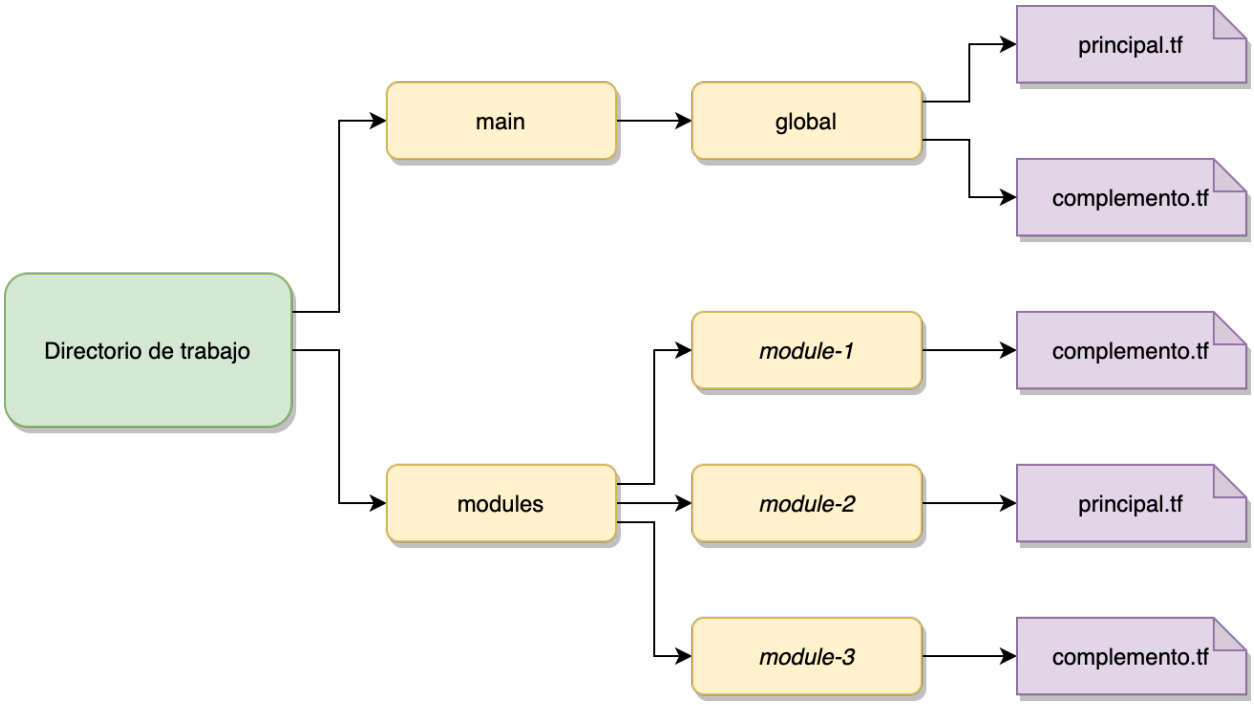

Figura 5 Directorio de trabajo de ejemplo para el componente global

Nótese que en el directorio global ha aparecido tanto el fichero del CTD base como el del complemento. En el caso de los módulos, también han aparecido todos pero, dado que el módulo *module-1* estaba repetido, ha tenido prioridad el del complemento.

Posteriormente, se quiere desplegar la infraestructura con el complemento para un componente de usuario. En este otro caso, se obtendría la estructura de la Figura 6.

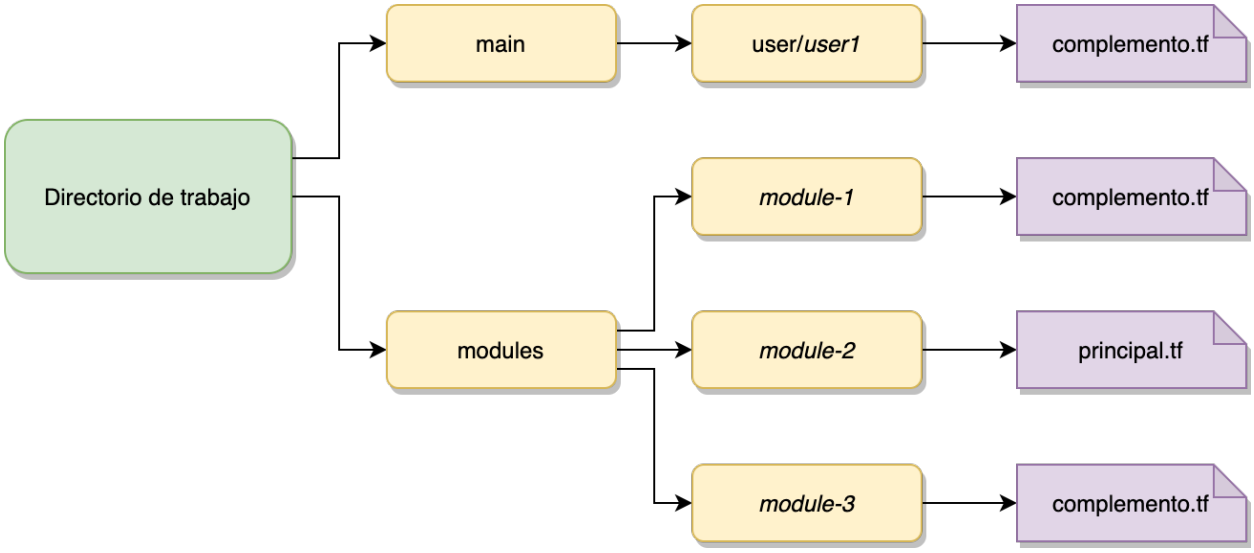

Figura 6 Directorio de trabajo de ejemplo para el componente de usuario

Nótese que el fichero del directorio principal para el usuario tenía el mismo nombre en el CTD base y en el complemento, por lo que finalmente queda únicamente el del complemento. El caso de los módulos es el mismo que para el componente global.

3.2 Gestión de variables

En el el apartado 3.1.2.3 se definía una forma concreta de organizar las variables, clasificándolas en tres tipos: configurables, de sabor y estáticas.

Aprovechando que *Terraform* es capaz de admitir varios ficheros de variables en su entrada y sobreescribir los valores repetidos, en el marco de trabajo se implementa la clasificación de variables al definirlas en ficheros diferentes. En función del orden con el que se pasan estos ficheros a Terraform, sus variables tendrán mayor o menor prioridad. El orden sería el siguiente (de mayor a menor prioridad):

1. Configurables.
2. De sabor.
3. Estáticas

El valor de una variable de un tipo de menor prioridad sería sustituido por el definido en otro de mayor prioridad en caso de coincidencia.

Cabe destacar que, al igual que con los CTDs, las variables de los complementos son más prioritarias que las del código base. Entre complementos, la prioridad la marca el orden en el que se añaden, teniendo más prioridad el complemento que se haya añadido más tarde. De esta manera, las variables estáticas de un complemento son más prioritarias que las configurables del código base.

Las variables de sabor son las únicas cuyos valores están definidos en varios ficheros distintos dentro del VTD, y que se identifican con un nombre de sabor. Al elegir un sabor, se elige un conjunto de variables a utilizar. En la ejecución de *Terraform*, sólo se le pasa por argumentos un fichero de cada tipo, lo que obliga a elegir un sabor concreto.

Ejemplo: se tienen los siguientes ficheros en un VTD:

```
# static.tfvars
name = "default"
version = "v1"
```

```
# flavour/small.tfvars
size = "small"
```

```
# flavour/big.tfvars
size = "big"
```

```
# config.tfvars
name = "MyConfigurableName"
```

Suponiendo que se elige el sabor pequeño, la ejecución de *Terraform* se realizará con los siguientes parámetros:

```
terraform apply -var-file=static.tfvars -var-file=flavour/small.tfvars -var-file=config.tfvars
```

Las variables que realmente se aplicarían serían las siguientes:

```
name = "MyConfigurableName"
version = "v1"
size = "small"
```

Nótese que, al tener más prioridad las variables configurables, el valor de la variable *name* es el definido en el fichero *config.tfvars*.

A continuación, se añade al ejemplo el concepto de complemento (*plugin* en inglés). En este caso, se tendría definido un VTD adicional, el del complemento. Para el ejemplo, únicamente va a tener definido el siguiente fichero:

```
# plugin/static.tfvars
name = "default"
version = "v2"
plugin_color = "red"
```

Este plugin modifica el valor por defecto de la variable *version* y añade una variable nueva *plugin_color*. La

ejecución de *Terraform* se realizaría de la siguiente forma:

```
terraform apply -var-file=static.tfvars -var-file=flavour/small.tfvars -var-
file=config.tfvars -var-file=plugin/static.tfvars
```

Teniendo como resultado final de las variables:

```
  name = "MyConfigurableName"
  version = "v2"
  size = "small"
  plugin_color = "red"
```

Nótese que, aunque las variables que define el complemento son estáticas, son más prioritarias que cualquiera de las variables definidas en el VTD base.

3.3 Control de versiones

Uno de los objetivos que se planteaban en la introducción de este proyecto era el de gestionar la infraestructura mediante un sistema de control de versiones, habilitando así la posibilidad de utilizar el flujo de trabajo *GitOps*[4].

Para ello se propone el uso de dos tipos de repositorio:

- **De código**.
- **De estado**.

3.3.1 Repositorios de Código

Contienen la descripción de la infraestructura a gestionar en código HCL, siguiendo la estructura del CTD (ver apartado 3.1.2). La metodología de trabajo con este repositorio se delega en los desarrolladores como en cualquier otro proyecto de software, decidiendo ellos cómo utilizar los *commits*, ramas, etc.

Este código puede ser empleado en múltiples despliegues, por lo que no debe contener información relativa a ninguno en particular, y utilizando variables para todos aquellos parámetros que deben ser personalizados en cada entorno individual.

Por otro lado, un mismo despliegue puede basarse en varios repositorios de códigos, ya que pueden contener CTDs de complementos (ver apartado 3.1.3). Esta característica aumenta aún más las posibilidades de personalización de entornos concretos, sin necesidad de mantener código específico para cada uno de ellos.

3.3.2 Repositorio de estado

Este repositorio tiene una relación uno a uno con cada despliegue concreto, y habilita el control de versiones de su estado y su configuración. Toda la información necesaria para desplegar y gestionar un entorno específico debe almacenarse ahí, de forma que un nuevo administrador pueda continuar esa labor únicamente basándose en el código que define la infraestructura y la información almacenada en el repositorio de estado.

Aquí sí se especifica una serie de reglas para almacenar la información del despliegue. Se utilizan dos ramas: una rama de estado y otra de variables. En los siguientes subapartados se explica su uso en mayor profunidad.

3.3.2.1 Rama de estado

Se utiliza para almacenar el fichero de estado que genera *Terraform* con una instantánea de la situación de la infraestructura en un momento concreto. Este sistema de almacenamiento no está nativamente soportado por *Terraform*, por lo que se debe almacenar el fichero localmente y gestionar su seguimiento por GIT externamente.

La motivación detrás de mantener el estado bajo control de versiones es la de tener un histórico con los

[4] GITOPS es un término acuñado por la empresa *WeaveWorks*. Más información en las referencias [51] y [44].

cambios de estado que sufre la infraestructura gestionada durante su ciclo de vida, en el que también aparezca reflejado su autor. Esto facilita labores de auditoría, de regreso a estados anteriores ó de recuperación ante desastres. Adicionalmente, permite que el estado sea compartirlo entre administradores que tengan acceso al repositorio.

Cada vez que se produzca un cambio en la infraestructura se deberá crear un *commit* nuevo con el nuevo fichero de estado. El flujo se describe en la Figura 7

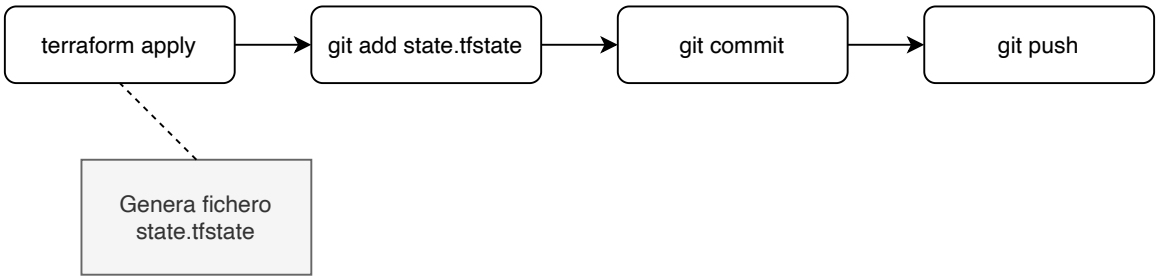

Figura 7 Diagrama de flujo con los comandos para el almacenado en git del estado

Si un colaborador quisiera modificar la infraestructura, previamente tendría que obtener el estado almacenado en el repositorio. De esta forma, *Terraform* sólo aplicaría los cambios necesarios, en lugar de desplegarlo todo otra vez. El flujo se representa en la Figura 8.

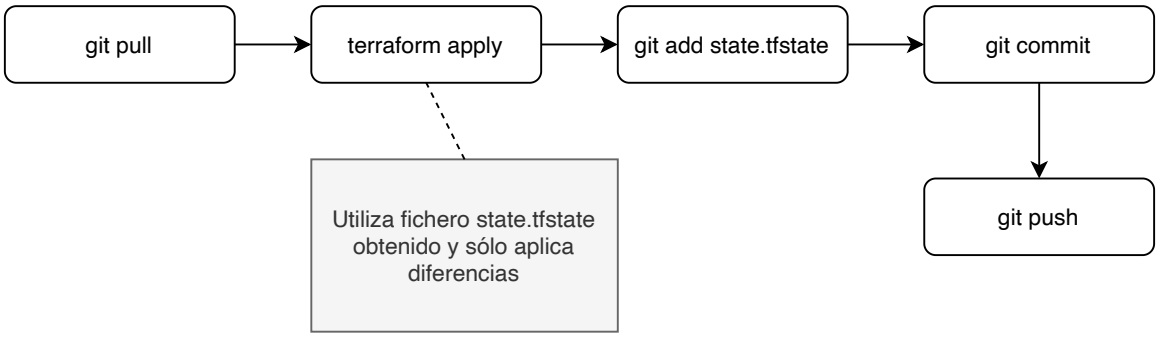

Figura 8 Diagrama de flujo con los comandos para trabajar colaborativamente con el estado

Por último, cabe destacar que cada componente tendrá un estado independiente. Tanto el componente global como cada componente de usuario guarda su propio fichero de estado, en la estructura presentada en la Figura 9. En el caso de los componentes de usuario se añade un nivel más de jerarquía en el árbol de directorios, debido a su multiplicidad.

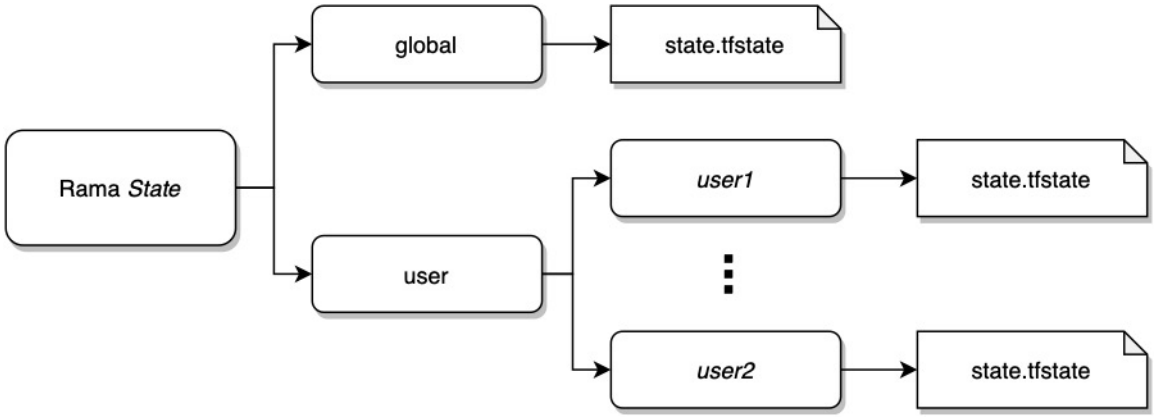

Figura 9 Estructura de directorios y ficheros de la rama de estado

3.3.2.2 Rama de variables

Se utiliza para guardar los ficheros de variables utilizados en un despliegue concreto. Al igual que con el estado, mantener las variables bajo control de versiones tener un histórico de las configuraciones aplicadas en cada despliegue, incluyendo su autor, habilitando además la posibilidad de revertir la infraestructura

gestionada a una configuración anterior.

Los flujos de ejecución son idénticos a los de la rama de estado. De hecho, cada vez que se realiza un cambio, se debe iniciar el mismo flujo en ambas ramas.

Los ficheros de variables deben organizarse siguiendo una estructura concreta, que se muestra en la

Figura 10.

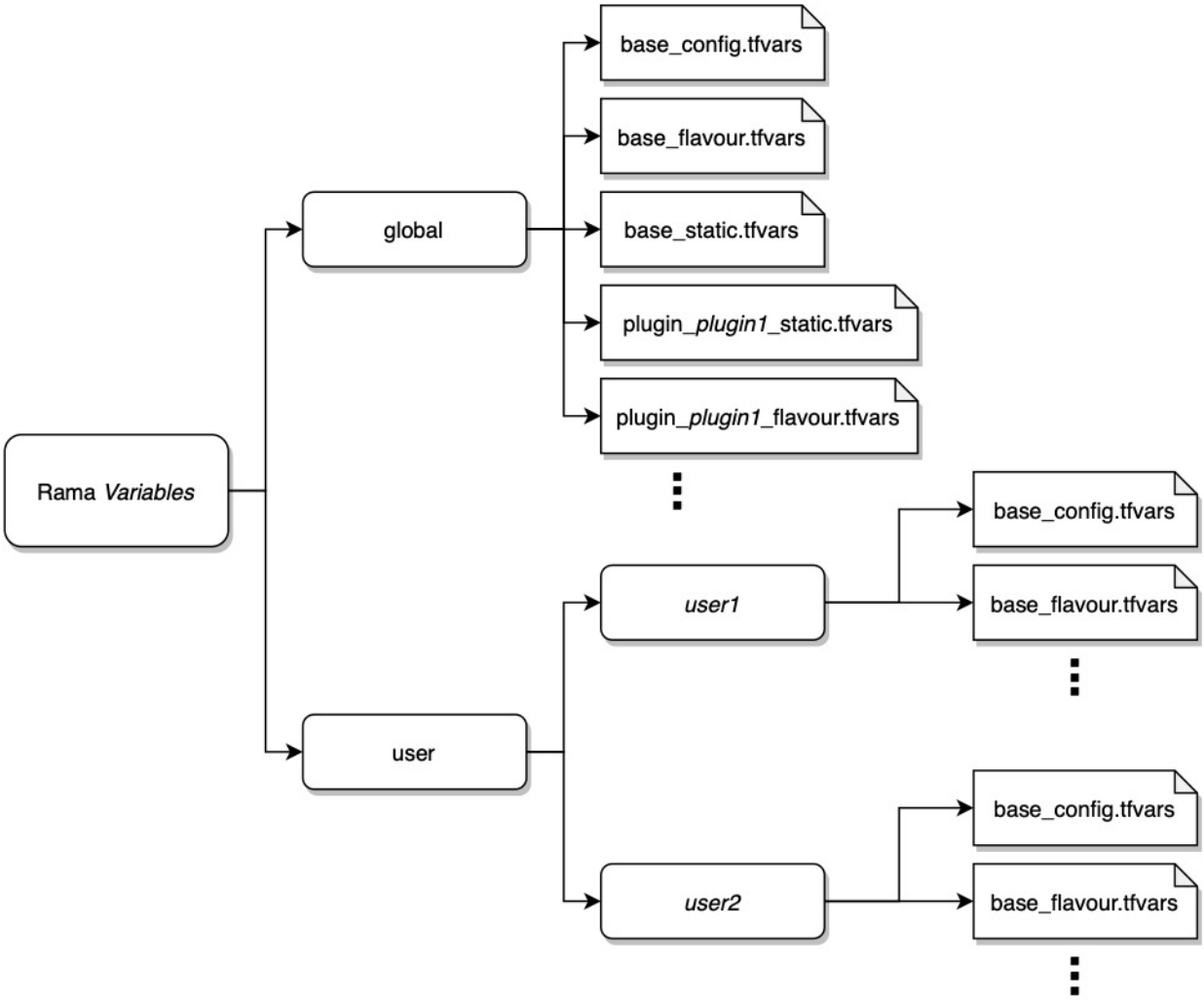

Figura 10 Estructura de directorios y ficheros de la rama de variables

Los ficheros de variables correspondientes al componente global se guardan en su correspondiente directorio *global*, mientras que los de cada usuario se almacenan en un subdirectorio con el nombre de dicho usuario dentro del directorio *user*.

Los ficheros además se renombran con respecto como se definían en el VTD para facilitar su identificación. Los correspondientes al VTD *base* se prefijan con *base_*, mientras que en el caso de los complementos se prefijan con *plugin_<nombre_complemento>*.

4 DISEÑO DE LA HERRAMIENTA

En este capítulo se aborda el diseño y la estructuración de la herramienta que implementa el marco de trabajo del proyecto, cuyo nombre es *Sonatina*. Se basa para ello en especificar una serie de requisitos que debe cumplir la herramienta, y que luego deberán ser implementados por distintos componentes del software.

Sonatina se propone como un conjunto de módulos de *Golang* que implementan una serie de librerías para seguir los flujos de trabajo y la metodología especificada en el capítulo anterior. Además, también proporciona una interfaz CLI que utiliza estas librerías para ejecutar los flujos de una forma sencilla.

4.1 Requisitos de *Sonatina*

4.1.1 Requisitos funcionales

Tabla 2 Requisitos funcionales de *Sonatina*

Identificador	Título	Descripción
RF1	Multidespliegue	Se debe poder gestionar el ciclo de vida de diferentes infraestructuras, pudiendo elegir en cada momento sobre qué infraestructura aplicar cambios y con qué código y configuración.
RF2	Estado de GIT	Se debe poder guardar en un repositorio GIT el estado de *Terraform* de la infraestructura, de forma que cada cambio en esta coincida con un commit que refleje cuándo se realizó el cambio, por quién y cuál fue el estado final de la intervención.
RF3	Variables en GIT	Se debe poder guardar en un repositorio GIT las variables de *Terraform* utilizadas para el despliegue, con el fin de tener un histórico de cambios de estas variables y de poder reproducir el despliegue desde otros equipos.
RF4	Código de *Terraform* en GIT	El código que se utiliza para describir la infraestructura debe ser obtenido de un repositorio GIT. Esto incluye tanto el código base como el de los distintos plugins que se añadan. Las referencias a estos repositorios también se deben almacenar en GIT para poder ser compartidos entre los distintos administradores de la infraestructura.
RF5	Gestión de sabores	Soporte para elegir el sabor con el que se realiza un despliegue, y guardarse como Metadatos en GIT.
RF6	Gestión de variables	Soporte para la gestión de los distintos tipos de variables definidos en el marco de trabajo (apartado 3.2), gestionando su prioridad y orden de aplicación.

RF7	Gestión de versiones	Se debe poder elegir la versión del código de *Terraform* a utilizar en cada despliegue, guardándolo como metadatos para poder ser compartido mediante GIT. Esta versión será en realidad una referencia a un *commit*, ya sea mediante un hash, una rama o una etiqueta asociada.
RF8	Interfaz CLI	Todas las operaciones que soporta *Sonatina* deben estar disponibles mediante una interfaz de línea de comandos (CLI).
RF9	Múltiples administradores	Los despliegues gestionados por *Sonatina* deben poder ser gestionados por múltiples administradores desde distintos equipos, aprovechando las capacidades de los respositorios GIT.

4.1.2 Requisitos no funcionales

Tabla 3 Requisitos no funcionales de *Sonatina*

Identificador	Título	Descripción
RNF1	Soporte para OS X, Linux y Windows	El código debe ser compatible para todos los sistemas operativos que soporta *Terraform*.
RNF2	Minimizar herramientas externas	Se debe evitar el uso de llamadas al sistema o comandos externos al código *Golang* de *Sonatina*.
RNF3	Separar librería de interfaces	Las librerías con la lógica de negocio de Sonatina deben estar separadas de las diferentes interfaces de usuario (por ejemplo, una CLI o una API REST), con el fin de facilitar la implementación de múltiples interfaces.

4.2 Elección de *Golang*

Para esta herramienta se ha decidido utilizar *Golang* por las siguientes características:

- **Lenguaje compilado**: Con *Golang* se pueden generar binarios compilados para cada uno de los sistemas en los que se quiera utilizar la herramienta, teniendo además todas sus dependencias autocontenidas. Esto facilita su distribución, al evitar la necesidad de instalar dependencias externas.
- **Mismo lenguaje que *Terraform***: se comparte el mismo lenguaje que la herramienta base, lo que facilita el uso de librerías relacionadas.
- **Buenas librerías para el caso de uso**: tiene librerías muy convenientes para *Sonatina*, como la librería *go-git* [31] para gestionar el control de versiones, o la librería *cobra* [32] para la creación de la interfaz por línea de comandos.

4.3 Componentes

Sonatina se organiza en paquetes de *Golang*, abordando cada uno de ellos una serie de funcionalidades. En este apartado se explican sus estructuras, diseños e interacciones entre sí.

4.3.1 Despliegue

En el marco de trabajo, un despliegue hace referencia a los recursos de una infraestructura concreta gestionada por *Sonatina*. El resultado final de un despliegue está determinado por los siguientes elementos:

- Código base de *Terraform* + código de complementos.
- Conjunto de variables y metadatos.
- Estado.

En *Sonatina*, los despliegues estarán representados por una interfaz llamada *Deployment* que define una serie de métodos para su gestión, y que estará implementada por objetos[5] *DeploymentImpl*. A su vez, está compuesto por otros objetos más específicos que implementan métodos para la gestión de los elementos mencionados en la lista anterior.

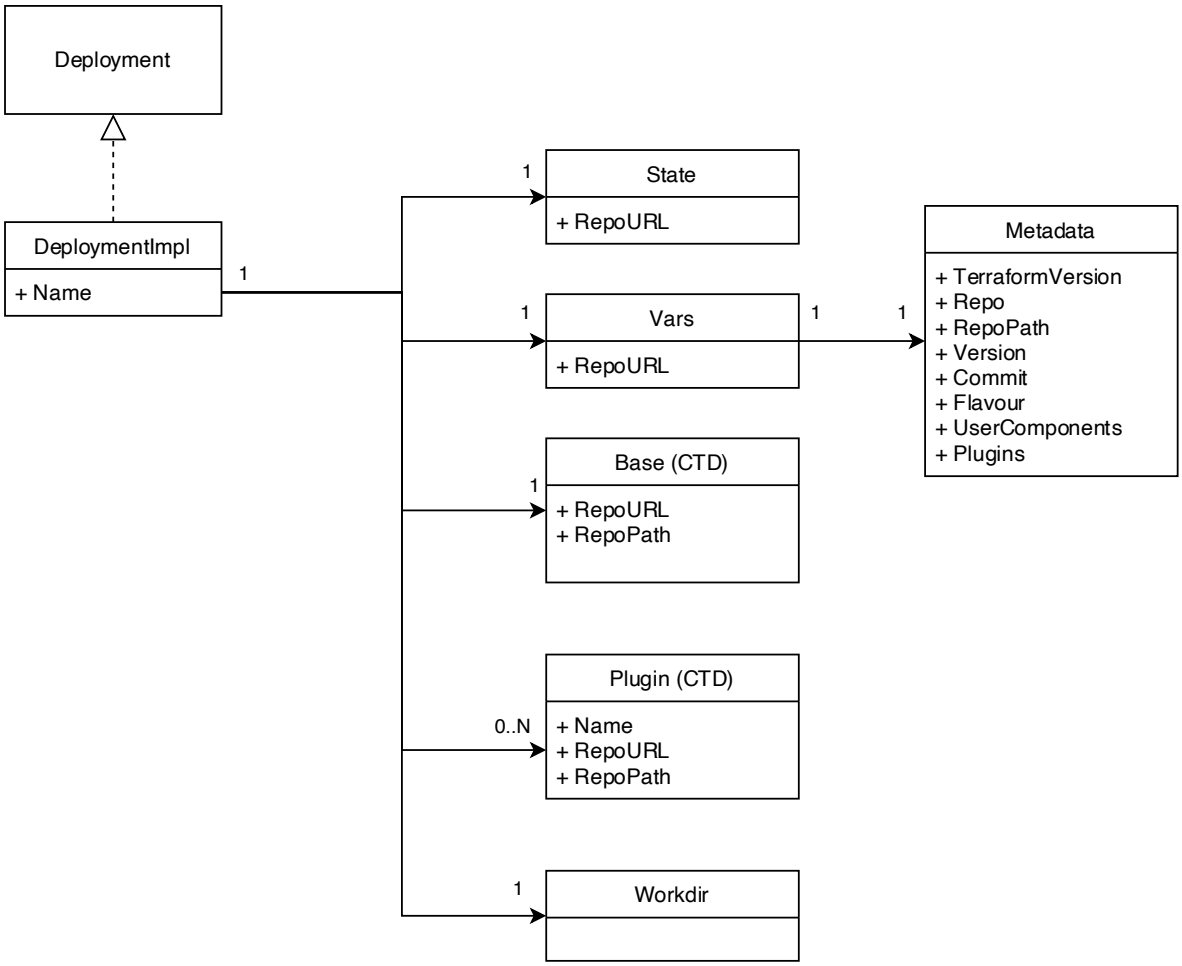

Figura 11 Diagrama de clases del paquete *Deployment*

En la Figura 11 se muestran las clases que componen a un objeto *Deployment*. Cada una de ellas gestiona un conjunto de recursos y define una serie de operaciones sobre esos recursos. La interfaz *Deployment* es la que finalmente expone los métodos que se utilizan en otras partes de la herramienta, basándose en estas clases. En los siguientes subapartados se explica de forma resumida qué funciones realizan.

4.3.1.1 Estado

La clase *State* se encarga de gestionar el estado de *Terraform*. Su principal cometido es sincronizar con el

[5] *Golang* realmente no tiene el concepto de objeto, únicamente permite definir estructuras a las que se les puede asociar métodos. Sin embargo, para facilitar la comprensión, se hará referencia a estas estructuras como objetos de aquí en adelante, dando a entender que las estructuras en este contexto siempre implementan un conjunto de métodos.

repositorio remoto el fichero de estado haciendo *commits* cada vez que se realiza una operación que implica un cambio en el mismo. También se encarga su sincronización con el repositorio, permitiendo la gestión colaborativa del despliegue.

4.3.1.2 Variables

La clase *Vars* gestiona los ficheros de variables de *Terraform* para un despliegue concreto. Al igual que con la clase *State*, los ficheros de variables se actualizan en el repositorio haciendo *commits* cada vez que se actualiza la infraestructura.

También es responsable de realizar el tratamiento de los distintos tipos de variables, tal y como se explicaba en el apartado 3.2.

Los metadatos, aunque no son variables estrictamente, se almacenan en la misma rama del repositorio GIT, por lo que también se gestionan desde esta clase, o más concretamente, desde la clase *Metadata* que forma parte de la clase *Vars*.

Los metadatos deben contener la siguiente información:

- **Versión de *Terraform* utilizada**: para garantizar el mismo comportamiento independientemente del administrador que realice las operaciones.
- **Repositorio del código base de la infraestructura**: para identificar el repositorio que contiene el código que define la infraestructura base.
- **Versión del código base**: para identificar dentro del repositorio de código qué versión concreta aplicar.
- **Sabor del componente global**: para conocer qué variables de sabor se debe utilizar.
- **Complementos del componente global**: para identificar los repositorios que contienen el código de los mismos y poder aplicar sus modificaciones al código base.
- **Componentes de usuario**: para conocer qué componentes de usuario han sido creados. También debe contener un registro del sabor y de lista de complementos añadidos a cada uno de ellos.

Gracias a estos metadatos un colaborador podría administrar el despliegue desde otro equipo, posibilitando el requisito funcional RF9.

4.3.1.3 Base y complementos

Los objetos *Base* y *Plugins* son ambos de tipo CTD (ver apartado 3.1.2). Un despliegue siempre tiene al menos un CTD base, mientras que puede tener cero o más CTDs de complementos. Estos objetos contienen información sobre dónde encontrar los ficheros del código HCL que describen la infraestructura.

Este código no es aplicado directamente por *Terraform*, primero se procesan y copian los ficheros a un directorio de trabajo de la forma explicada en el apartado 3.1.3.

4.3.1.4 Directorio de trabajo

Se implementa en la clase *Workdir* y define los métodos que obtienen los ficheros HCL referenciados en cada uno de los CTDs de un despliegue (tanto base como complementos), procesándolos y copiándolos en el directorio de trabajo para que *Terraform* pueda aplicar la configuración de la infraestructura, tal y como se especifica en el apartado 3.1.3.

4.3.2 Gestor de despliegues

El módulo *Manager* es el encargado de administrar la multitud de despliegues que la herramienta puede

gestionar. Sigue la filosofía del patrón *Singleton*[6], en el sentido de que en Sonatina sólo se creará un único objeto de este tipo.

El módulo está compuesto por una interfaz con el mismo nombre. Esta interfaz está implementada inicialmente por el objecto *ManagerJSON* que gestiona la lista de despliegues gestionados por *Sonatina* guardando la información en un fichero JSON. Sin embargo, se podrían utilizar otras formas de guardar esta lista (por ejemplo, en una base de datos o un fichero YAML) implementando la interfaz.

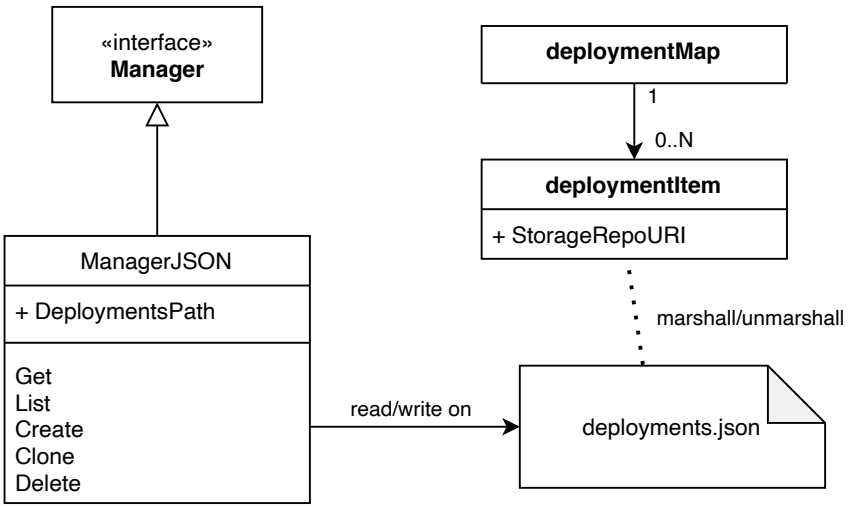

Figura 12 Diagrama de clases del componente Manager

Se utilizan los objetos *deploymentMap* y *deploymentItem* para serializar/deserializar el contenido del JSON en los atributos de dichos objetos.

Gracias a que el módulo *Manager* actúa como un *singleton*, se puede construir un objeto *Deployment* en cualquier parte del código a partir de este módulo. Funciona como una factoría de objetos de despliegue inicializados con la información almacenada en el fichero *deployments.json*.

4.3.3 Ejecución de *Terraform*

El paquete *terraformcli* implementa una serie de clases y métodos para la ejecución de comandos de *Terraform* desde *Sonatina*. Los tres comandos soportados son:

- `terraform init`
- `terraform apply`
- `terraform destroy`

Además, debe soportar el descargado automático del binario de *Terraform* para la versión configurada en el despliegue, de forma que independientemente del equipo en el que se gestione, siempre utilice la misma versión de *Terraform*.

4.3.4 Ejecución de GIT

El paquete *gitw* funciona de forma análoga al paquete *Terraformcli* en el sentido de que encapsula toda la lógica relacionada con la ejecución de comandos GIT, al igual que se hacía con los comandos de *Terraform*. La diferencia en este caso es que, en lugar de utilizar una instalación de GIT externa, se utiliza una librería que implementa sus funcionalidades nativamente en *Golang*, cuyo nombre es *go-git* [31].

El objetivo de encapsular las operaciones GIT es el de exponer una interfaz estable de uso de repositorios al resto de *Sonatina*. Encapsular la lógica de gestión de GIT permite cambiar su implementación sin que afecte a otros componentes de la herramienta.

[6] Patrón típico de la programación orientada a objetos que garantiza que una clase sólo puede ser instanciada una vez. Si se trata de instanciar una segunda vez, se devolverá la primera instancia creada. Se explica con mayor profundidad en la referencia [41]

4.3.5 Flujos de trabajo

La clase *workflow* implementa flujos de trabajo de *Sonatina* que requieren del uso conjunto de varios componentes. Esto permite implementar los flujos a nivel de librería para que diferentes interfaces de usuario puedan reutilizarlos, como por ejemplo una CLI o una API REST. Sin embargo, su uso no es obligatorio, ya que no aportan ninguna funcionalidad extra.

Los flujos que debe implementar *Sonatina* coinciden con los comandos de *Terraform* soportados (*init*, *apply* y *destroy*), ya que necesita combinar funcionalidades del paquete *Deployment* con el del paquete *Terraformcli*.

4.3.6 Interfaz de usuario: CLI

El paquete *Cmd* implementa la interfaz CLI para utilizar *Sonatina*. Está basada en la librería *Cobra* [32], que provee de una interfaz simple y muy potente para su creación.

Lo más importante a destacar a nivel de diseño es que este paquete únicamente debe contener la lógica de gestión de la CLI, estando toda la lógica de negocio encapsulada en las librerías de *Sonatina*, tal y como especifica el requisito RNF1. De esta manera, se facilita el desarrollo de otras interfaces (como por ejemplo una API REST) sin tener que duplicar código.

Los comandos estarán compuestos por hasta dos argumentos, de los cuales el primero es un verbo que indica qué operación realizar, y el segundo es un predicado que indica sobre qué tipo de recurso realizar la operación.

Se pueden agrupar los comandos en función del tipo de recurso que gestionan. En los siguientes subapartados se especifican los comandos que *Sonatina* debe implementar.

4.3.6.1 Gestión básica de despliegues

Operaciones básicas con despliegues, que en su mayoría implementadas en la librería por el paquete *manager*.

Tabla 4 Comandos para la gestión básica de despliegues

Operación	Comando
Creación de un despliegue	`sonatina create deployment <despliegue>`
Obtención de un despliegue a partir del repositorio de estado	`sonatina clone deployment <despliegue>`
Eliminación de un despliegue	`sonatina delete deployment <despliegue>`
Listado de despliegues	`sonatina list deployments`
Configuración del despliegue en curso	`sonatina use deployment <despliegue>`

El último comando es específico para el uso de la CLI, ya que permite configurar un despliegue en uso para aplicar al resto de comandos sin tener que especificarlo siempre como parámetro.

4.3.6.2 Ejecución de flujos de trabajo

Los flujos de trabajo definidos en el paquete *workflow* se asignarán directamente a la CLI sin indicar predicado, y realizando las operaciones del despliegue en curso.

Tabla 5 Comandos para la gestión básica de despliegues

Operación	Comando
Ejecución del flujo de inicialización del despliegue	`sonatina init`
Ejecución del flujo de aplicación del despliegue	`sonatina apply`
Ejecución del flujo de destrucción del despliegue	`sonatina destroy`

4.3.6.3 Gestión de componentes de usuario

Estos comandos deben permitir realizar la gestión básica de los componentes de usuario.

Tabla 6 Comandos para la gestión de componentes de usuario

Operación	Comando
Creación de un nuevo componente de usuario	`sonatina create usercomponent <componente>`
Eliminación de un componente de usuario	`sonatina delete usercomponent <componente>`
Listado de los componentes de usuario	`sonatina list usercomponents`

4.3.6.4 Gestión de sabores

Estos comandos deben permitir configurar el sabor a utilizar para el despliegue. Además, deben disponer un parámetro para elegir el componente de usuario a configurar, eligiéndose el componente global por defecto.

Tabla 7 Comandos para la gestión de sabores

Operación	Comando
Obtención del sabor configurado	`sonatina get flavour [-c <componente>]`
Configuración del sabor	`sonatina set flavour [-c <componente>]`

4.3.6.5 Gestión de complementos

Estos commandos deben permitir realizar la gestión de los complementos que se añaden al despliegue, ya sea al componente global o a los componentes de usuario.

Tabla 8 Comandos para la gestión de complementos

Operación	Comando
Creación de un complemento	`sonatina create plugin <complemento> [-c <componente>]`
Eliminación de un complemento	`sonatina delete plugin <complemento> [-c <componente>]`
Listado de los componentes de usuario	`sonatina list plugins [-c <componente>]`

4.3.6.6 Gestión de variables

Estos comandos deben permitir ver los distintos ficheros de variables en función del componente, del complemento y del tipo de variable especificado. Además, en el caso de las variables configurables, deben soportar la apertura de un editor de texto para configurarlas.

Tabla 9 Comandos para la gestión de variables

Operación	Comando
Mostrado de variables	`sonatina show <tipo> [-c <componente> -p <complemento>]`
Edición de variables configurables	`sonatina edit [-c <componente> -p <complemento>]`
Actualización de variables con el repositorio remoto	`sonatina refresh`

5 IMPLEMENTACIÓN DE LA HERRAMIENTA

En el capítulo anterior se describió el diseño general de *Sonatina*, que tiene una estructura basada en paquetes *Golang* encargados de implementar los distintos requisitos. Este capítulo trata de entrar más en el detalle concreto de la implementación, explicando los flujos, la integración de distintos elementos y explicando el por qué de las decisiones técnicas tomadas. También se pretende dar cuenta de los principales retos a resolver en el desarrollo y cómo se han llevado a cabo.

Es importante aclarar que esta descripción está muy ligada a la versión de *Sonatina* desarrollada en el momento de escribir esta memoria, y que puede haber cambiado como consecuencia de la evolución y mantenimiento del software.

5.1 Estructura básica del proyecto

El proyecto sigue una de las estructuras típicas del código fuente escrito en *Golang*. Consiste en un paquete principal donde se sitúan las funciones de inicialización del programa, como la función *main*, y una serie de paquetes que encapsulan diferentes partes del software, y que se organizan en directorios separados.

En concreto, los dos únicos ficheros con código *Golang* en el directorio raíz son *main.go* y *configuration.go*. Ambos sólo implementan la fontanería de arranque del programa y carga de configuración, dejando la lógica de negocio en otros paquetes, de los cuáles ya se ha hecho una introducción en el capítulo anterior.

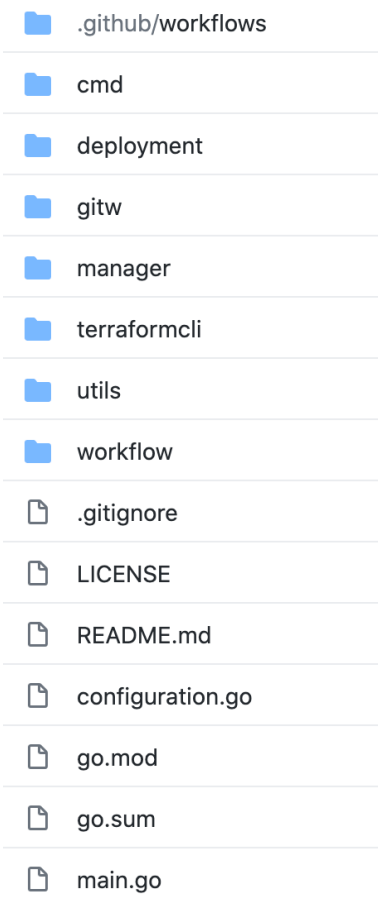

Figura 13 Directorio raíz del repositorio de Sonatina

En el directorio raíz, además del código *Golang* y los paquetes, se pueden encontrar otros ficheros auxiliares del proyecto:

- *LICENSE*: fichero que suele estar presente en la raíz de todos los proyectos software, especialmente si son de código abierto. Especifica la licencia bajo la que se publica el proyecto. En el caso de *Sonatina*, se publica bajo la licencia Apache-2.0 [33], que se caracteriza por permitir la libre distribución del software y su uso privado y comercial. Se ha elegido esta licencia para facilitar el uso de la herramienta a empresas privadas que quieran integrarla en sus flujos de desarrollo.
- *README.md*: contiene una descripción y una guía inicial del proyecto, de cara a dar una introducción a los nuevos desarrolladores que quieran comenzar a utilizar la herramienta.
- *go.mod y go.sum*: estos ficheros contienen las referencias a los paquetes y librerías externas utilizadas por *Sonatina*. Concretamente, en *go.mod* se especifican las versiones de las librerías, mientras que en *go.sum* se especifican unas firmas calculadas a partir del código de esas librerías, para garantizar que no se ha modificado.
- *.gitignore*: este fichero permite indicar a GIT qué ficheros no deben ser gestionados por el sistema de control de versiones. Por ejemplo, los artefactos de compilación no deben ser gestionador por GIT, y se añaden a este fichero para tal fin.
- *.github*: este directorio contiene los ficheros necesarios para configurar el sistema de integración continua *Github Actions* [11]. Este servicio se encarga de compilar y ejecutar pruebas de integración continua cada vez que se añade o modifica código en el repositorio, ayudando así a mantener el software siempre funcional y reduciendo el riesgo de fallos.

5.2 Arranque de la herramienta

Todo lenguaje de programación dispone de una función donde comienza el flujo de la ejecución. En el caso de *Golang* se trata de la función *main* (al igual que en muchos otros lenguajes), que no retorna ningún valor.

Sin embargo, cabe destacar que, en el caso de *Golang*, es posible ejecutar unas funciones de inicialización para los distintos paquetes, que se ejecutan **antes** que la función *main*: son las funciones nombradas con la palabra reservada *init*. Cada paquete puede definir las suyas, y en el caso del paquete raíz de *Sonatina*, se utilizan para inicializar la configuración de la herramienta, en el fichero *configuration.go.* El tratamiento de la configuración en *Sonatina* será objeto del siguiente apartado.

La función *main* únicamente realiza dos operaciones:

1. Inicializar la herramienta externa *ssh-agent* en sistemas UNIX. Esta herramienta permite utilizar el sistema de autenticación del sistema operativo para establecer conexiones SSH. Será necesaria para poder conectar con repositorios remotos GIT.
2. Arrancar la librería de gestión de línea de comandos, cuyo nombre es *Cobra*. Se explicará con más detalle en el apartado 5.9.

5.3 Configuración de la herramienta

Para configurar *Sonatina* se hace uso de la librería *viper* [34], que ofrece, entre otras, las siguientes funcionalidades:

- Configuración de valores por defecto.
- Obtención de configuración de ficheros de múltiples formatos, como JSON, YAML, TOML, etc.
- Obtención de configuración desde variables de entorno.
- Acceso a la configuración como un *Singleton*, es decir, acceso a la instancia del objeto con los valores de configuración desde cualquier parte del programa.

Realmente en *Sonatina* sólo se leen parámetros de configuración en dos paquetes: *cmd* y *manager*. En el caso de *cmd*, que es el paquete que implementa la interfaz CLI, es utilizada por otra librería, llamada *cobra*, la cuál se abordará con mayor profundidad en el apartado 5.9. En el caso del paquete *manager*, se emplea para conocer dónde guardar los ficheros relacionados con los despliegues dentro del sistema operativo. El resto de paquetes que forman el conjunto de librerías de *Sonatina* reciben la configuración por parámetros a partir de los valores configurados en los dos anteriores.

Sonatina ofrece una serie de parámetros configurables, aunque todos tienen valores por defecto razonables que permiten el uso de la herramienta sin necesidad de tener que configurar nada. Son los siguientes:

- *LogLevel*: permite configurar el nivel de detalle de las trazas generadas por la herramienta.
- *LogFile*: permite especificar en qué fichero guardar las trazas generadas por la herramienta.
- *EnableStacktrace*: permite especificar si, en caso de error, se debe mostrar por pantalla la pila de errores.
- *DeploymentsPath*: permite especificar el directorio donde guardar los ficheros necesarios para trabajar con los despliegues.
- *DeploymentFilename*: permite indicar el nombre del fichero donde se guarda el registro de despliegues gestionados por *Sonatina*.
- *ManagerConnector*: permite indicar qué tipo de gestor de despligues usar. Actualmente sólo se soporta el tipo *json*.
- *TestFilesystem*: permite indicar si se quiere utilizar un sistema de ficheros en memoria para realizar pruebas.

- *TerraformPath*: permite indicar el directorio en el que guardar los binarios de *Terraform* utilizados por la herramienta.
- *DefaultTerraformVersion*: permite indicar la versión de *Terraform* a usar por defecto para nuevos despliegues.
- *DefaultFlavour*: permite indicar el sabor a utilizar por defecto, en caso de no especificar uno concreto para un despliegue.
- *Editor*: permite indicar el editor de texto a utilizar cuando se editan las variables de configuración de un despliegue.

Los valores por defecto se configuran en el fichero *configuration.go*, en una de las funciones de inicialización. También se configura un prefijo ("*SONATINA_*") para las variables de entorno, de forma que se eviten conflictos con otras aplicaciones si se quiere especificar algún valor de la configuración de *Sonatina* utilizando este mecanismo.

5.4 Gestor de despliegues

Como se explicaba en el apartado 4.3.2, el paquete *manager* se encarga de administrar qué despliegues está gestionando la herramienta. Para ello, se define una interfaz con los siguientes métodos:

- *List*: lista los despliegues que está gestionando la herramienta.
- *Get*: dado un nombre de despliegue, devuelve el objeto *Deployment* asociado.
- *Create*: permite crear un nuevo despliegue. Para ello se necesitan los siguientes parámetros:
  - *Name*: nombre del despliegue
  - *storageRepoURI*: enlace al repositorio donde se guarda el estado, las variables y los metadatos.
  - *codeRepoURI*: enlace al repositorio donde se encuentra el código base que describe la infraestructura a desplegar.
  - *codeRepoPath*: directorio dentro del repositorio anterior donde se encuentra el CTD.
- *Clone*: permite añadir un despliegue que ya ha sido previamente creado y tiene ya configurado su repositorio de estado. Sólo le hace falta los dos primeros parámetros del punto anterior, ya que los otros dos los puede obtener de los metadatos almacenados en el repositorio de estado.
- *Delete*: elimina un despliegue para que deje de gestionarlo la herramienta. No elimina el repositorio remoto, sólo la información almacenada localmente.

Esta interfaz podría ser implementada por diferentes clases que utilizaran distintos métodos para almacenar la lista de despliegues a gestionar y sus atributos. En esta primera versión de *Sonatina* únicamente se ha implementado la clase *ManagerJSON*, que utiliza un fichero local en formato JSON para guardar esta información. A continuación, se muestra un ejemplo en el que hay dos despliegues con sus respectivas referencias a su repositorio de estado.

Código 2 Ejemplo de fichero *deployment.json*

```
{
  "example-local-docker": {
    "storage_repo_uri": "git@github.com:arodriguezdlc/sonatina-example-local-docker-
state.git"
  },
  "example-kubernetes": {
    "storage_repo_uri": "git@github.com:arodriguezdlc/sonatina-example-kubernetes-
state.git"
  }
}
```

Para gestionar el fichero JSON, se utiliza la librería nativa de *Golang* que permite gestionar un objeto JSON mediante una estructura con anotaciones. Aprovechando eso, se definen dos funciones, una función *load* que lee el fichero JSON y asigna sus valores a las variables de la estructura y una función *save*, que genera el fichero JSON a partir de los valores de la estructura.

Leyendo y escribiendo en el fichero JSON, y jugando con los valores de la estructura se implementan todos los métodos de la interfaz *Manager*.

Por último, hay otros dos métodos definidos a nivel de paquete, y que se exponen a continuación:

- *InitializeManager*: inicializa al principio de la ejecución del programa el gestor una única vez. A partir de aquí deberá obtenerse la instancia utilizando el método del siguiente punto. Además, permite elegir el tipo de gestor a utilizar, si bien actualmente sólo se soporta el basado en ficheros JSON.
- *GetManager*: devuelve la instancia del gestor inicializada al principio del programa.

5.5 Despliegue

Un despliegue en concreto se gestiona en *Sonatina* a través de una interfaz *Deployment*. Casi todas las operaciones soportadas por la herramienta se realizan a través de esta interfaz, aunque su implementación también se basa en otros paquetes.

En el apartado 4.3.1 se explicaba la distribución y diseño de los diferentes componentes que formaban un despliegue. Aquí se procede a profundizar en cómo se ha implementado.

La interfaz *Deployment* está definida de la siguiente forma:

Código 3 Interfaz *Deployment*

```
type Deployment interface {
  CreateUsercomponent(user string) error
  DeleteUsercomponent(user string) error
  ListUsercomponents() ([]string, error)

  CreatePluginGlobal(name string, repo string, repoPath string) error
  DeletePluginGlobal(name string) error
  ListPluginsGlobal() ([]string, error)

  CreatePluginUser(name string, user string) error
  DeletePluginUser(name string, user string) error
  ListPluginsUser(user string) ([]string, error)

  GetFlavourGlobal() (string, error)
  SetFlavourGlobal(flavour string) error

  GetFlavourUser(user string) (string, error)
  SetFlavourUser(flavour string, user string) error

  GenerateWorkdirGlobal() (string, error)
  GenerateWorkdirUser(user string) (string, error)

  GenerateVariablesGlobal() ([]string, error)
  GenerateVariablesUser(user string) ([]string, error)

  GetVariableFilepath(kind string, plugin string, user string) (string, error)
  ReadVariableFile(kind string, plugin string, user string) (string, error)

  Push(message string) error
  Pull() error

  StateFilePathGlobal() string
  StateFilePathUser(user string) string

  TerraformVersion() string
  CodeRepoURL() string
  CodeRepoPath() string
```

```
  Purge() error
}
```

Todos estos métodos están implementados en una clase *DeploymentImpl*, por lo que implementa la interfaz *Deployment*. La razón de esta división es facilitar las pruebas entre las distintas clases de un despliegue, de forma que se puedan utilizar versiones simplificadas de la clase que implementa la interfaz.

En muchos de estos métodos la lógica está implementada en clases específicas dentro del objeto *Deployment*, encargándose este último de realizar verificaciones globales y de ofrecer una interfaz que orqueste el uso de estas clases. Se describen a continuación estos métodos en grupos:

- *CreateUsercomponent, DeleteUsercomponent y ListUsercomponent*: estos métodos permiten gestionar operaciones de creación, listado y borrado de componentes de usuario. Estas operaciones implican modificaciones en el estado, en las variables y en los metadatos, por lo que los objetos que gestionan estos elementos implementan de forma separada estas operaciones. En el objeto *Deployment*, se llaman a todas ellas orquestando así la gestión completa del componente de usuario.
- *CreatePluginGlobal*, *DeletePluginGlobal*, *ListPluginGlobal*, *CreatePluginUser*, *DeletePluginUser*, *ListPluginUser*: este conjunto de métodos permite gestionar los complementos de un despliegue. Al igual que con los métodos de gestión de componentes de usuario, las operaciones se realizan en las subclases correspondientes, realizándose aquí únicamente la orquestación. Además, en el caso de la creación del complemento global, se realiza el clonado del repositorio del complemento para tenerlo disponible localmente. No es necesario realizar esta operación en el caso de los complementos de usuario, porque es obligatorio que haya sido añadido al componente global previamente (y por lo tanto ya se habrá clonado).
- *GetFlavourGlobal*, *GetFlavourUser*, *SetFlavourGlobal*, *SetFlavourUser*, *TerraformVersion*, *CodeRepoURL*, *CodeRepoPath*: todos estos métodos atacan directamente al correspondiente método de la subclase *Metadata*, actuando aquí simplemente como interfaz al exterior del paquete.
- *GenerateWorkdirGlobal, GenerateWorkdirUser, GenerateVarsGlobal, GenerateVarsUser*: estos métodos preparan los ficheros en las ubicaciones necesarias para poder ejecutar posteriormente *Terraform*. Atacan directamente a los métodos con el mismo nombre definidos en las subclases *Vars* y *Workdir*.
- *Push y Pull*: permiten sincronizar los repositorios locales y remotos de almacenamiento. Esta interfaz llama a los correspondientes métodos en las clases *State* y *Vars*, para que cada una gestione su rama del repositorio.
- *GetVariableFilepath y ReadVariableFil*e: permiten acceder al fichero de variables de configuración, en función de si es del código base o de un complemento, y de si es del componente global o de un componente de usuario. De esta manera se da soporte a la edición de estas variables desde las interfaces de usuario.
- *Purge*: permite eliminar todo el contenido local de un despliegue.

5.5.1 Estado

La clase de *State* tiene como objetivo gestionar la sincronización de los ficheros de estado que genera *Terraform*, almacenándolos en la rama de estado del repositorio de almacenamiento en forma de *commits* cada vez que se modifica. Para ello implementa dos métodos:

- *Pull*: se encarga de descargar los estados del repositorio remoto para sincronizarlos con el local.
- *Push*: se encarga de crear un nuevo *commit* con los últimos cambios locales de los ficheros de estado, y de subirlo al repositorio remoto.

Implementa también dos constructores. Se utilizará uno u otro en función de si el despliegue es completamente nuevo o ha sido ya previamente inicializado por alguna ejecución de *Sonatina*, ya sea en el mismo equipo o en cualquier otro:

- *createState*: genera un nuevo repositorio local apuntando a la rama de estado, crea la estructura de directorios necesaria y genera un primer *commit* de inicialización.
- *cloneState*: descarga la rama de estado del repositorio remoto, que previamente había sido inicializada.

En ambos casos únicamente se requiere conocer la URL del repositorio de almacenamiento.

5.5.2 Variables

La clase *Vars* tiene dos objetivos principales:

- Gestionar la sincronización de los ficheros de variables de un despliegue concreto con su repositorio de estado, por lo que realiza operaciones con GIT sobre la rama de variables del repositorio.
- Calcular y copiar los ficheros de variables adecuados desde los CTD y VTD a la rama de variables del repositorio de estado, es decir, implementa el procedimiento descrito en el apartado 3.1.2.3.

Además, hay un tercer objetivo, la gestión de metadatos, que está delegada a la clase *Metadata*, pero que está contenida en el interior de la clase *Variables* porque el fichero que almacena los metadatos también se sincroniza en la rama de variables, como si fuera otro fichero de variables. Se profundizará sobre la clase *Metadata* en el apartado 5.5.3.

Para lograr el primer objetivo, se implementan dos métodos:

- *Pull*: se encarga de obtener los últimos cambios almacenados en la rama de variables del repositorio de estado.
- *Push*: se encarga de crear un nuevo *commit* con los últimos cambios locales en la rama de variables y subirlas al repositorio de estado.

Estos dos métodos realizan operaciones de GIT, para lo cuál utiliza el paquete *gitw*, que centraliza las acciones de control de versiones realizadas por *Sonatina*.

Para el segundo objetivo, se implementan dos métodos que realizan las mismas operaciones, pero ligeramente adaptadas en función de si se está trabajando con el componente global o con un componente de usuario específico.

- *GenerateGlobal*: se encarga de generar, a partir de los VTD del código base y los complementos, los ficheros de variables resultantes en el repositorio de estado. En primer lugar, se copian los ficheros del código base, prefijándolos con "*base_*". Posteriormente se copian los ficheros de los complementos añadidos (se consultan de los metadatos), prefijándolos con "*plugin_<nombre_complemento>_*". En el caso del complemento global, la copia se realiza al directorio *global* dentro de la rama de variables.
- *GenerateUser*: implementa las mismas operaciones que *GenerateGlobal*, pero para un componente de usuario concreto. La copia se realiza al directorio "*user/<nombre_de_usuario>*", también dentro de la rama de variables.

Además, se definen métodos específicos para tratar la copia de cada tipo de ficheros de variables, pero que se utilizan de forma privada (no se exponen en el paquete *Golang*).

Por último, en este paquete se definen dos constructores de la clase Vars, en función de si el despliegue es completamente nuevo o ya ha sido creado en alguna otra ejecución de *Sonatina*, ya sea en el mismo equipo o en otro. Son los siguientes:

- *createVars*: genera un nuevo repositorio local para las variables, creando además la rama correspondiente a las variables. Requiere de los parámetros básicos para inicializar los metadatos (*terraformVersion*, *codeRepoURL*, *codeRepoPath*, *flavour*), además de la referencia al respositorio de almacenamiento (*storageRepoURL*) que utilizará para guardar el estado y las variables. Una vez creada la estructura de directorios y el fichero inicial de metadatos, realiza un primer *commit* en el repositorio.
- *cloneVars*: en lugar de generar un repositorio nuevo, clona uno ya existente a partir de la referencia al repositorio de almacenamiento (*storageRepoURL*). Asume que el repositorio ya ha sido inicializado,

por lo que no necesita los parámetros que necesitaba la función *createVars*, con realizar el clonado es suficiente.

5.5.3 Metadatos

Los metadatos se gestionan mediante la clase *Metadata*, que actúa como un modelo con atributos que se persisten en un fichero JSON, utilizando la librería nativa de *Golang*. Esto permite serializar o deserializar la clase Metadata, haciéndola persistente entre ejecuciones de la herramienta. El modelo tiene los atributos mostrados en la figura

Código 4 Estructura *Metadata*

```
type Metadata struct {
  fs       afero.Fs
  filePath string

  TerraformVersion string                    `json:"terraform_version"`
  Repo             string                    `json:"repo"`
  RepoPath         string                    `json:"repo_path"`
  Version          string                    `json:"version"`
  Commit           string                    `json:"commit"`
  Flavour          string                    `json:"flavour"`
  UserComponents   map[string]userComponent `json:"user_components"`
  Plugins          []globalPlugin            `json:"plugins"`
}

type userComponent struct {
  Plugins []userPlugin `json:"plugins"`
  Flavour string       `json:"flavour"`
}

type globalPlugin struct {
  Name     string `json:"name"`
  Repo     string `json:"repo"`
  RepoPath string `json:"repo_path"`
  Version  string `json:"version"`
  Commit   string `json:"commit"`
}

type userPlugin struct {
  Name string `json:"name"`
}
```

Los atributos que se serializan en JSON son los que empiezan en mayúscula, los otros dos (*fs* y *filePath*) son variables auxiliares utilizadas en la lectura y escritura del fichero. A continuación, se profundiza en cada uno de los atributos:

- *TerraformVersion*: versión de *Terraform* utilizada en este despliegue concreto.
- *Repo*: URL del repositorio con el código base utilizado para gestionar la infraestructura.
- *RepoPath*: ruta en la que se encuentra el código dentro del repositorio anterior.
- *Version* y *Commit*: versión específica del código del repositorio que se está utilizando en el despliegue.
- *UserComponents*: lista de componentes de usuario añadidos al despliegue. Cada uno de ellos puede tener asociado un sabor y un conjunto de complementos.
- *Plugins*: son los complementos añadidos al despliegue. Cada a su vez debe tener las referencias al código del complemento: *Repo*, *RepoPath*, *Version* y *Commit*.

Tal y como se muestra en la figura, cada uno de estos parámetros tiene una etiqueta que indica en qué clave del fichero JSON se almacena cada atributo.

La clase *Metadata* tiene una serie de métodos asociados para trabajar con sus atributos, y que realizan verificaciones para garantizar que la información siempre es consistente. Por ejemplo, si una operación implica

la edición de un componente de usuario o de un complemento, se comprueba que existen, devolviendo error si no es así.

Finalmente, para serializar y deserializar la clase se implementan dos métodos, *load* y *save*. El primero lee el fichero JSON y carga sus atributos en la clase, mientras que el segundo escribe en formato JSON los valores actuales de los atributos de la clase.

Cabe destacar que la interfaz *Deployment* recubre en muchos casos las llamadas a los métodos de la clase *Metadata* para exponerlos a otras partes de la herramienta, pero en ningún caso se utiliza directamente la clase desde fuera.

5.5.4 Definición del árbol de código

La clase *CTD* provee de información relacionada con un árbol de definición de código concreto. Esta información básicamente consiste en las referencias al repositorio donde está el código y la ruta donde se encuentra su copia local, para facilitar luego su procesamiento en el directorio de trabajo. Las funciones que implementa son las siguientes:

- *ListMainGlobalFiles*: lista los ficheros .tf del directorio principal para el caso del componente global.
- *ListMainUserFiles*: lista los ficheros .tf del directorio principal para el caso de un componente de usuario concreto.
- *ListModules*: lista lo módulos definidos en el CTD.
- *Pull*: actualiza la copia del código del CTD local con los últimos cambios del repositorio remoto.
- *Checkout*: cambia a un *commit*, rama o versión específica del código.

5.5.5 Directorio de trabajo

Terraform, en sus commandos de aplicar y destruir la infraestructura, recibe como parámetro el directorio donde se encuentran los ficheros de código HCL. En *Sonatina*, estos ficheros inicialmente no se encuentran todos juntos en un mismo directorio, si no que se distribuyen en los distintos CTD que configuran la descripción de la infraestructura. Por ello, es necesario generar un directorio de trabajo donde ubicar la combinación resultante de todos los CTDs, de forma que *Terraform* pueda aplicar la configuración de un único directorio.

La clase *Workdir* se encarga de realizar esta tarea, procesando todos los CTDs necesarios y generando el directorio de trabajo. Para ello expone dos métodos que realizan las mismas operaciones, dependiendo su uso de si se está trabajando con el componente global o con un componente de usuario: *GenerateGlobal* y *GenerateUser*. El flujo de ejecución se muestra en la Figura 14.

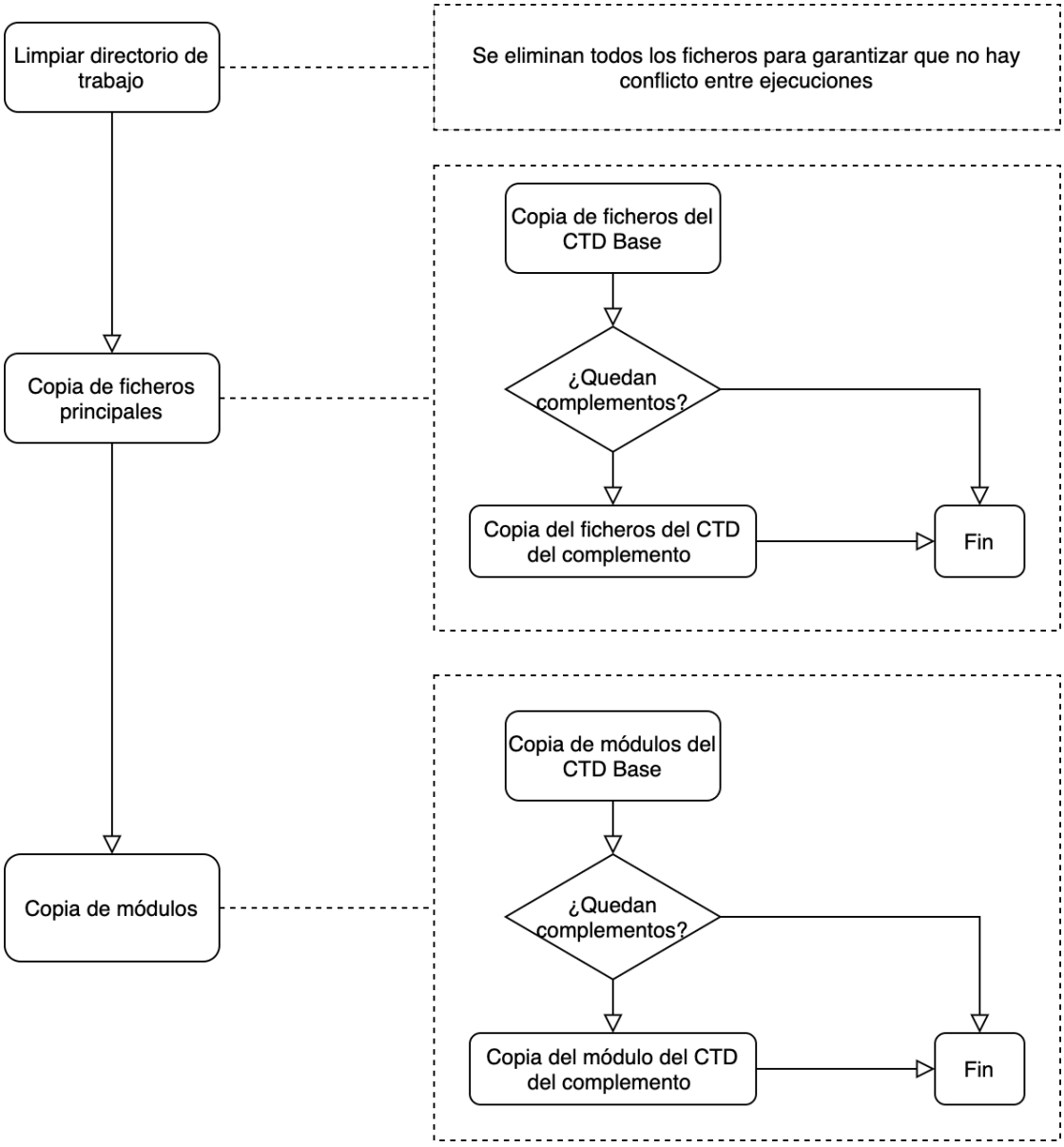

Figura 14 Diagrama de flujo de la generación del directorio de trabajo

Internamente, cada copia ya sea de ficheros del directorio principal o módulos, se realiza en dos fases:

1. Cálculo de ficheros a copiar
2. Copia de ficheros (sobreescribiendo, si es necesario).

Para estas operaciones se utilizan funciones definidas en el paquete de utilidades *utils*, que contiene funciones para listado de ficheros con una extensión determinada o copia recursiva de ficheros.

5.6 Ejecución de *Terraform*

Para la gestión de la ejecución de *Terraform* se ha dedicado en *Sonatina* un paquete específico: *terraformcli*. Recubre los distintos comandos soportados y garantiza que se utiliza la versión correcta de *Terraform* en cada despliegue, llegando incluso a descargar el binario si fuera necesario.

La ejecución del binario se realiza gracias a la librería *os/exec* de *Golang*, que ofrece la capacidad de ejecutar comandos externos gestionando tanto salidas como códigos de ejecución. Cada uno de los comandos soportados por *Sonatina* está implementado por un método de la clase *Terraform*, que a su vez está compuesto por la clase *Binary*, que es la que ejecuta el comando.

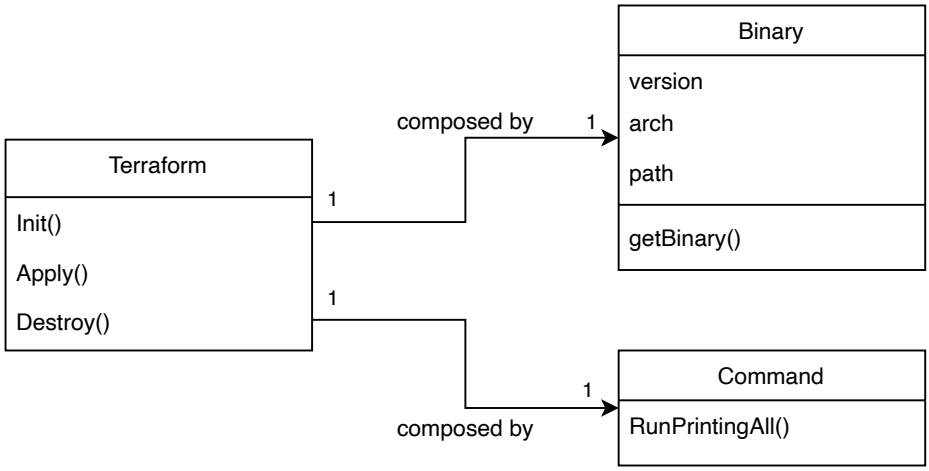

Figura 15 Diagrama de clases del paquete *terraformcli*

El módulo también implementa la función *New*, que construye el objeto *Terraform* y el objeto *Binary*, descargándose si es necesario el binario de *Terraform* de la web de Hashicorp para una versión y arquitectura concreta. De esta manera, *Sonatina* puede gestionar despliegues con diferentes versiones de *Terraform*, ya que se encargará automáticamente de obtener el binario correcto.

5.7 Ejecución de GIT

En el caso de GIT, no se utiliza un comando externo, sino una librería escrita puramente en *Golang*: *go-git*. De esta manera se reducen las dependencias externas y el acoplamiento con el equipo anfitrión.

Este paquete está compuesto únicamente por la clase *Command*, cuyo objetivo es replicar el comportamiento de los comandos GIT necesarios para la herramienta con una interfaz similar a la CLI de GIT. Cabe destacar que cada objeto de tipo *Command* está asociado a una ruta en el equipo local, que es donde se realizarán las operaciones. Los métodos que ofrece son los siguientes:

- *Clone*: clona un repositorio dada su URL.
- *CloneBranch*: clona un repositorio dada su URL, pero descargando únicamente la rama especificada. Se utiliza por las clases *Vars* y *State*, ya que trabajan siempre con una rama concreta.
- *Init*: Inicializa un nuevo repositorio. Se utiliza por las clases *Vars* y *State* durante el proceso de creación de un nuevo despliegue.
- *CheckoutNewBranch*: crea una nueva rama en el repositorio y la fija como rama actual. Es el equivalente al comando *git checkout -b <rama>*.
- *AddGlob*: Añade todos los ficheros del repositorio al espacio de trabajo para añadirlos al siguiente *commit*.
- *Commit*: genera una nueva confirmación o *commit* con los cambios añadidos al espacio de trabajo.
- *RemoteAdd*: añade una nueva referencia remota al repositorio.
- *Pull*: sincroniza el repositorio local con los últimos cambios del repositorio remoto.
- *Push*: sube los últimos cambios del repositorio local al repositorio remoto.

GIT soporta muchos más comandos y funcionalidades, pero aquí únicamente se han implementado los necesarios para el funcionamiento de *Sonatina*.

5.8 Flujos de trabajo

Como ya se introdujo en el apartado 4.3.5, la clase *Workflow* implementa una serie de flujos de ejecución que

utilizan los paquetes *Deployment* y *Terraformcli*, con el fin de unificar la implementación para distintas interfaces de la librería. Los flujos definidos se explican con mayor profundidad en los siguientes subapartados.

5.8.1 Init

Este flujo realiza todo el preprocesado de los ficheros .TF explicado en el apartado 3.1.2, para posteriormente ejecutar el comando *Terraform Init*, que se encarga de inicializar *Terraform*, descargando los módulos y *providers* necesarios. El flujo es el siguiente:

1. **Generación del directorio de trabajo (*workdir*)**: Esta generación está implementada en el paquete *Deployment*, y es el que se encarga de mezclar los distintos CTDs (base, platform y plugins), preparándolo para que *Terraform* pueda aplicar la configuración resultante tras la combinación.
2. **Ejecución del comando *Terraform Init***: Una vez preparado el directorio de trabajo, se ejecuta el comando de *Terraform* que, basándose en el contenido de los ficheros .TF, inicializará los módulos y *providers* necesarios para la posterior ejecución del *apply* o *destroy*.

Es importante destacar que, como este flujo no modifica la infraestructura desplegada (o a desplegar), no se realiza ningún *commit* ni ninguna modificación del repositorio de estado.

5.8.2 Apply

Este flujo se encarga de aplicar los cambios necesarios para que la infraestructura alcance el estado descrito por el código de *Terraform*. Este flujo también incorpora los mismos pasos que el flujo de *Init*, por lo que realmente podría ejecutarse sin necesidad de lanzar el flujo anterior. Se compone de los siguientes pasos:

1. **Generación del directorio de trabajo (*workdir*)**: es el mismo paso que el explicado en el flujo *Init*.
2. **Procesamiento de ficheros de variables**: En este paso, que está implementado en el paquete *Deployment*, se copian los ficheros de variables de los distintos VTD, teniendo en cuenta metadatos como el sabor, a la rama *vars* del repositorio de estado local (procedimiento descrito en el apartado 3.2, y cuya implementación se explica en el apartado 5.5.2). En el último paso se hará *commit* de estos ficheros y se subirán al repositorio remoto. Este paso devuelve la lista de ficheros de variables que deberán utilizarse en el *Apply*.
3. **Ejecución del comando *Terraform Init***: es el mismo paso que en el flujo *Init*.
4. **Ejecución del comando *Terraform Apply***: en este paso se ejecuta el comando de *Terraform* basándose tanto en los ficheros del *workdir*, como en los ficheros de variables que se procesaron en el paso 2. Esto aplica los cambios necesarios en la infraestructura y genera un fichero de estado de *Terraform*.
5. **Actualización del repositorio de estado**: una vez aplicados los cambios, se hace *commit* de los ficheros de variables y de estado en sus respectivas ramas y se suben los cambios al repositorio de estado.

5.8.3 Destroy

Este flujo es muy similar al *apply,* con la única diferencia de que, en lugar de ejecutarse el comando *Terraform apply* en el paso 4, se ejecuta el comando *Terraform destroy*. También se hace *commit* y se suben los cambios al repositorio de estado, al ser un cambio aplicado a la infraestructura.

5.9 Interfaz de usuario: línea de comandos

La CLI está implementada en el paquete *cmd* basándose en la librería *Cobra*, ya introducida en el apartado 4.3.6, donde también se especificaban los comandos que debía tener la interfaz.

Cada comando definido con Cobra es una instancia de la estructura *Command*. En esta estructura, se puede

configurar varios aspectos del comando, como las palabras que lo invocan, descripciones y ayudas, o la gestión de argumentos. Finalmente, también contiene una referencia a la función a ejecutar cuando se invoca el comando.

Algunos comandos también tienen referencia a subcomandos. En el caso de *Sonatina*, se aprovecha esta característica para definir un comando con verbo que indica la operación, y un subcomando con el predicado que indica sobre qué elemento se realiza la operación. Los distintos subcomandos están clasificados en subpaquetes:

- ***Operations***: contiene la definición de todos los verbos.
- ***Deploymentcmd, Usercomponent, Plugin y Flavour***: contienen los predicados que hacen referencia a los despliegues, componentes de usuario, complementos y sabores respectivamente.

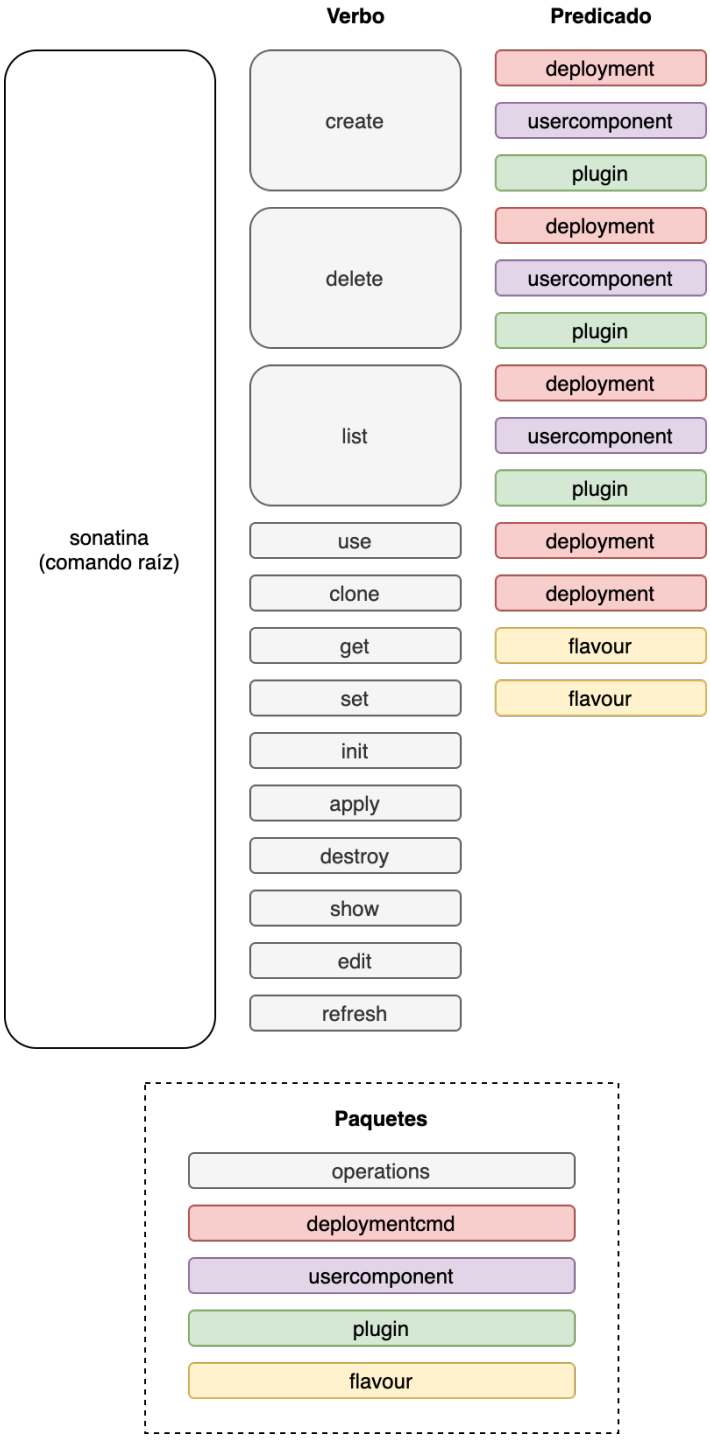

Figura 16 Subcomandos de Sonatina clasificados por paquetes

Además, se define un subpaquete adicional *common*, que contiene funciones utilizadas en muchos de los comandos, sirviendo de librería compartida. Concretamente, implementa tres funciones:

- *InitializeTerraform*: inicializa un objeto de tipo *terraformcli*, para las operaciones de flujo de trabajo.
- *GetCurrentDeployment*: obtiene el despliegue en curso sobre el que realizar las operaciones, en caso de que no se especifiquen en un argumento del comando.
- *SetCurrentDeployment*: modifica el despliegue en curso.

5.10 Utilidades transversales

El paquete *utils* contiene una serie de funciones comunes que son utilizadas en varios paquetes de la herramienta. Se dividen en tres ficheros en función del tipo de utilidad:

- *array.go*: contiene funciones para gestionar listas y mapas.
    - FindString: permite, dada una lista de cadenas de caracteres, encontrar la posición de una en concreto.
    - RemoveDuplicatedStrings: elimina elementos duplicados en una lista de cadenas de caracteres.
- *filesystem.go*: contiene funciones relacionadas con la gestión de ficheros.
    - *NewFileIfNotExist*: crea un nuevo fichero vacío en la ruta indicada sólo si no existe previamente.
    - *NewFileWithContentIfNotExist*: crea un nuevo fichero con el contenido especificado en la ruta indicada sólo si no existe previamente.
    - *NewDirectoryWithKeep*: crea un directorio y le añade un fichero *.keep* vacío. Se utiliza para mantener el directorio registrado en GIT aunque no contenga ficheros.
    - *FileCopy*: copia ficheros desde una ruta de origen a una ruta de destino.
    - *FileCopyRecursively*: copia los ficheros de un directorio recursivamente desde uan ruta de origen a una ruta de destino.
    - *FileListRecursively*: lista los ficheros y directorios dentro un directorio recursivamente.
    - *FileListRecursivelyWithoutDir*: lista los ficheros dentro de un directorio recursivamente, sin los propios directorios.
- *http.go*: contiene funciones relacionadas con peticiones con el protocolo HTTP.
    - *HTTPDownloadFile:* descarga un fichero de una URL.

5.11 Sistema de ficheros

Para la gestión del sistema de ficheros, *Sonatina* utiliza la librería *Afero* [35], que define una interfaz abstracta común para muchos tipos de sistemas de ficheros. Esto permite utilizar, por ejemplo, el sistema de ficheros del sistema operativo ó un sistema de ficheros en memoria para pruebas, entre otras muchas opciones.

Muchas de las clases definidas en *Sonatina* tienen un atributo con un objeto de tipo *Afero.fs*, que les permite usar un sistema de ficheros dado. El objetivo de esta implementación es poder elegir para cada clase el sistema de ficheros a usar, lo cuál es muy útil para los tests unitarios.

5.12 Gestión de errores

En *Golang* la gestión de errores se realiza de manera explícita en lugar de mediante excepciones[7]. Se utiliza la convención de que las funciones que pueden producir errores deben devolver como último valor retornado un objeto que implemente la interfaz de errores.

Si bien *Golang* nativamente ya ofrece el tipo *error* para la gestión de errores, en *Sonatina* se utiliza la librería *github.com/pkg/errors* que añade algunas funcionalidades, como generación de pilas de errores para facilitar la depuración.

Como no todos los errores son generados directamente por *Sonatina*, para poder aprovechar las funcionalidades de la librería, se utiliza la función *Wrap* cada vez que se gestiona un error no generado por *Sonatina.* Esta función guarda el error original dentro de un error gestionado por la librería, permitiendo así aprovechar todas sus funcionalidades.

[7] Ver artículo en el blog de *Golang* sobre la gestión de errores [40]

6 CASO DE USO: SERVIDORES WORDPRESS CON BASE DE DATOS COMPARTIDA

6.1 Planteamiento del problema

Se plantea un caso en el que una empresa quiere ofrecer un servicio de hosting con servidores *WordPress* en el que cada cliente dispone de una instancia dedicada. Estos servidores necesitan para su funcionamiento de una base de datos *MySQL*, que se quiere compartir entre clientes para reducir costes. Para garantizar la separación de los datos de cada cliente, dentro del servidor de base de datos, se tendrían bases de datos lógicas independientes, cada una con sus respectivas credenciales de acceso.

La empresa dispondría de un clúster de *Kubernetes* sobre el que desplegar contenedores con los servidores. Se asume que el clúster está ya desplegado y que se dispone de acceso con permisos de administración al mismo.

Se necesita un mecanismo de gestión de la infraestructura que permita desplegar de forma automatizada el servicio de hosting. Un operador sería el encargado de dar de alta o de baja nuevos clientes, utilizando este mecanismo.

Adicionalmente, se pide una serie de requisitos en la gestión de la infraestructura:

- **Múltiples despliegues**: se quiere poder gestionar varias réplicas de la infraestructura, de forma que se pueda tener un entorno de producción y otro de desarrollo sobre el que realizar pruebas. El entorno de desarrollo debería utilizar menos recursos que el de producción para ahorrar costes.
- **Tamaño de servidor diferente por cliente**: se quiere poder ofrecer servidores con distintas capacidades en función del cliente.
- **Acceso HTTPS para determinados clientes**: se quiere tener la posibilidad de añadir encriptación a la comunicación con el WordPress de determinados clientes, dejándola deshabilitada para los demás.
- **Múltiples administradores**: varios operadores deben poder gestionar la infraestructura, cada uno desde su equipo.

6.2 Evaluación de alternativas

Dado que uno de los requisitos es poder desplegar de forma repetible la infraestructura del servicio, se hace necesario el uso de herramientas de IaC que sean capaces de realizar esta operación de forma automatizada.

La gran cantidad de herramientas de IaC existentes posibilitan una gran varidad de soluciones para este problema. A continuación, se exponen algunos motivos por los que descartar varias de estas herramientas, si bien no significa que no se pudiera llegar a una solución al problema con ellas.

- *Contenedores*: el requisito de utilizar contenedores hace menos conveniente el uso de herramientas más centradas en la configuración como código, como *Chef*, *Puppet* o *Ansible*. Los contenedores se

plantean como un conjunto de software y configuración inmutables, lo que va en contra de gestionar la configuración con herramientas externas.

- *Kubernetes:* El uso de este orquestador no está ligado a ningún proveedor de servicios de nube, por lo que se deben descartar herramientas como *AWS CloudFormation* u *OpenStack Heat*.

Las alternativas restantes son utilizar *Terraform,* ó directamente manifiestos de *Kubernetes* para realizar el despliegue, siendo ambas soluciones válidas para este caso de uso. Sin embargo, los requisitos de gestión adicionales obligan al desarrollo de alguna herramienta que orqueste el uso de *Terraform* o *Kubernetes*, ya que ninguna de los dos ofrece esas funcionalidades.

Es aquí donde entra en juego *Sonatina*, aportándole a *Terraform* la capa de gestión que posibilita el cumplimiento de los requisitos:

- ***WordPress* por cliente**: gracias a los componentes de usuario, se puede desplegar un conjunto de recursos por cada cliente, siendo en este caso el servidor *WordPress* y la base de datos lógica.
- **Base de datos compartida**: utilizando el componente global, se puede compartir la base de datos entre los distintos componentes de usuario.
- **Múltiples despliegues**: *Sonatina* es capaz de gestionar múltiples despliegues basados en un mismo código HCL, aportando además la posibilidad de configurar sus variables de forma independiente. De esta manera se podría tener un despliegue de desarrollo configurado con un conjunto de variables, y otro despliegue de producción, con otras variables distintas.
- **Tamaño de servidor diferente por cliente**: la gestión de sabores que aporta *Sonatina* permite asignar valores diferentes a las variables que definen el tamaño de la infraestructura de cada uno de ellos.
- **Acceso HTTPS para determinados clientes**: *Sonatina* ofrece la posibilidad de añadir complementos al código de la infraestructura base, tanto para el componente global como para los componentes de usuario. Esto permite que algunos clientes puedan tener elementos de la infraestructura adicionales, como un proxy que aporte cifrado SSL en el acceso a *WordPress*.
- **Múltiples administradores**: gracias a la integración con GIT, se puede administrar la infraestructura desde varios puestos de operación que tengan acceso al repositorio de estado, sirviéndo este como punto de sincronización.

Sonatina sería por tanto capaz de resolver las necesidades de la empresa sin necesidad de abordar el desarrollo de herramientas adicionales.

6.3 Organización del código

Si bien *Sonatina* ofrece una interfaz con la que realizar la gestión de la infraestructura conforme a los requisitos solicitados, sigue siendo necesario desarrollar el código de *Terraform* que describe los diferentes elementos de la infraestructura.

Para ello se seguirá la estructura propuesta por el marco de trabajo, que consiste en el desarrollo de módulos de *Terraform* que son organizados a través del módulo principal de *Sonatina*.

6.3.1 Módulos de Terraform

En primer lugar, se plantea qué módulos de *Terraform* desarrollar para realizar el despliegue. *Sonatina* no impone ninguna restricción en el desarrollo de módulos de Terraform, por lo que el desarrollador podría elegir libremente cómo realizar su implementación.

Atendiendo a las especificaciones del problema, se distinguen tres elementos diferenciados a gestionar, pudiéndose describir en módulos independientes:

1. El servidor de base de datos *MySQL* compartida.
2. Los servidores *WordPress* de cada cliente.

3. La base de datos lógica de cada cliente, con sus respectivas credenciales.

El desarrollo interno de cada uno de los módulos no forma parte de este trabajo, ya que son conceptos y conocimientos relacionados únicamente con *Terraform* y *Kubernetes*. Por ese motivo, en los siguientes subapartados únicamente se describen los elementos por los que se compone el módulo, sin exponer su proceso de desarrollo.

6.3.1.1 Módulo de servidor de base de datos

Este es el único módulo por el que está compuesto el componente global, y contiene los recursos de *Kubernetes* necesarios para desplegar y configurar la base de datos *MySQL*. Está compuesto por los siguientes recursos:

- **Conjunto con estado (*Stateful set*)**: agrupa una serie de contenedores asociados a un disco, de forma que pueda mantener un estado. En este caso, este conjunto estará compuesto por un solo contenedor con el servidor *MySQL*, que guardará su información en un volumen. En el caso de que se reinicie o se elimine el contenedor, *Kubernetes* se encargará de crear uno nuevo que utilice el mismo volumen como almacén de datos.
- **Servicio**: se encarga de permitir el descubrimiento a través de DNS del servicio ofrecido por *MySQL*, y de enrutar el tráfico al contenedor correspondiente.
- **Mapa de configuración**: contiene un documento clave/valor con elementos de configuración. En este caso se utiliza para guardar el nombre DNS con el que los servidores *WordPress* de los componentes de usuario se deberán conectar a la base de datos *MySQL*.
- **Secreto**: almacena información confidencial. En este caso se utiliza para guardar la contraseña del usuario raíz de la base de datos.

En la Figura 17 se muestran las relaciones entre los elementos. La base de datos obtiene del secreto mediante variables de entorno la contraseña del usuario raíz. Por otro lado, el servicio expone un nombre DNS dentro de *Kubernetes* con el que hacerla accesible en el clúster.

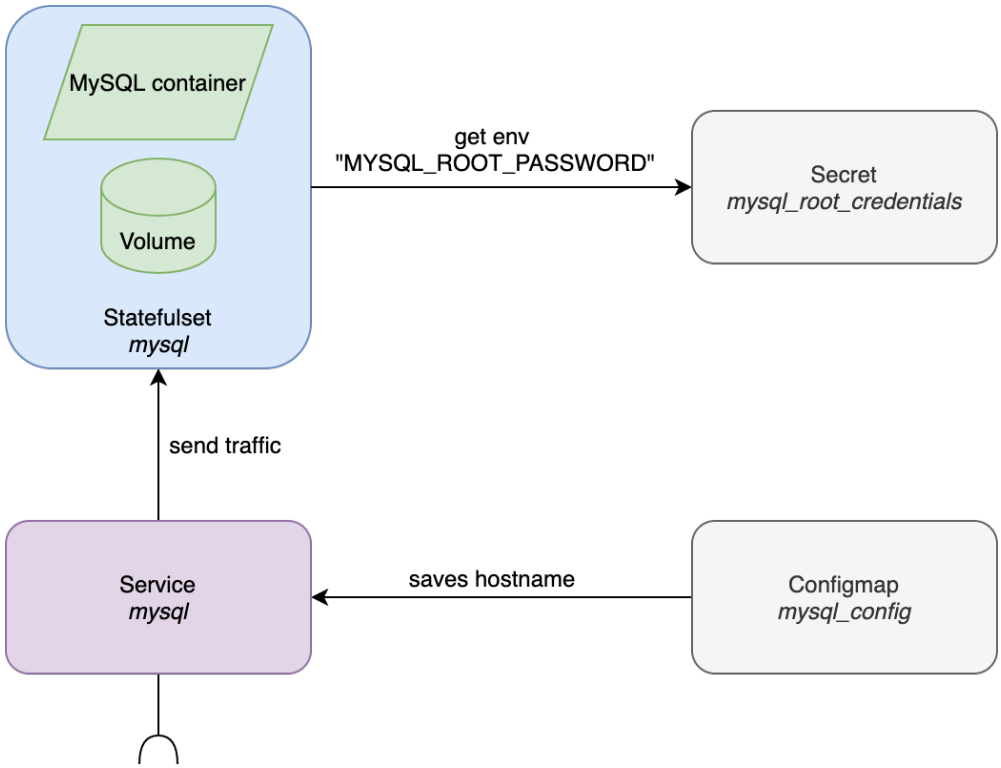

Figura 17 Recursos del módulo de servidor de base de datos

Las variables que acepta este módulo son las siguientes:

- Contraseña del usuario raíz.
- Imagen de *Docker* del servidor *MySQL*.
- Requisitos y límites de CPU y RAM para el contenedor.
- Tamaño del volumen de almacenamiento.

Más adelante, se explicará cómo se mapean estas variables a las de *Sonatina*.

6.3.1.2 Módulo de servidor WordPress

Este módulo contiene los recursos relacionados con el servidor de *WordPress*. Está compuesto por los siguientes recursos:

- **Conjunto sin estado (Deployment)**: este recurso gestiona los contenedores con el servicio de *Wordpress*. Puede estar compuesto por uno o más contendores, permitiendo así escalar horizontalmente el servicio.
- **Servicio**: este elemento es el encargado de permitir el descubrimiento a través de DNS del servicio ofrecido por los servidores *WordPress*, y de enrutar el tráfico a los contenedores correspondientes.
- **Mapa de configuración**: almacena la configuración que se aplicará al contenedor de *WordPress* a través de variables de entorno.
- **Secreto**: almacena información confidencial, en este caso la contraseña del usuario de la base de datos, que se pasará al contenedor de *WordPress* a través de una variable de entorno.

En la Figura 18, se muestra las relaciones entre los elementos. El servidor de *WordPress* obtiene su configuración a través del mapa de configuración y del servicio, cuyos valores son asignados a variables de entorno. El servicio expone un punto de entrada al servicio de *WordPress* asignándole un nombre DNS.

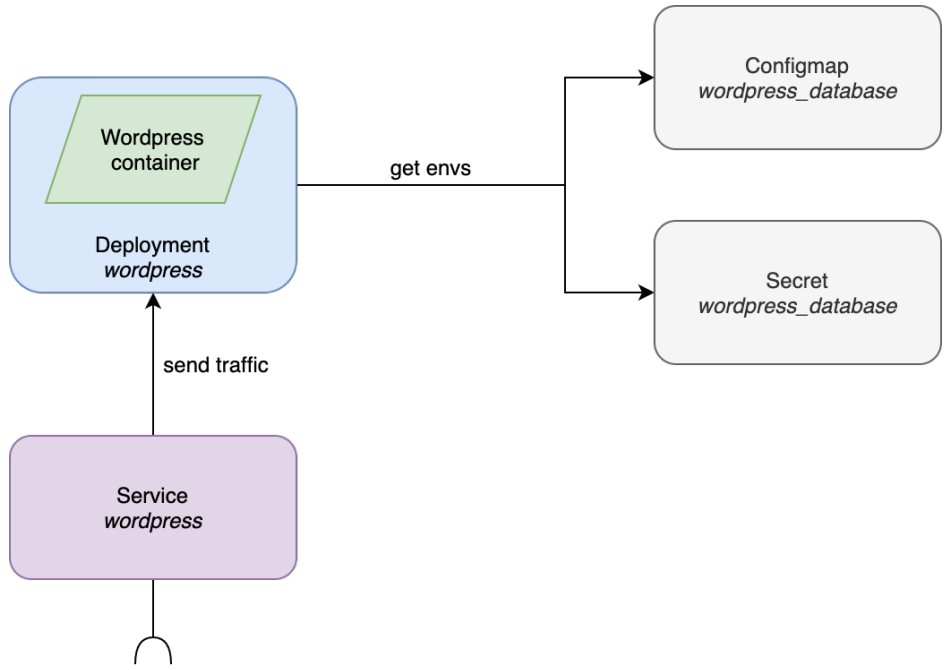

Figura 18 Recursos del módulo de Wordpress

Las variables de entrada que acepta son las siguientes:

- Componente de usuario (indica qué componente de usuario se está desplegando. Se utiliza para diferenciar los recursos que se crean por cada componente de usuario).
- Credenciales para el acceso a la base de datos.
- Imagen de *Docker* del servidor *WordPress*.

- Requisitos y límites de CPU y RAM para el contenedor de *WordPress*.
- Réplicas del servidor *WordPress*.

6.3.1.3 Módulo de inicialización de la base de datos

Si bien hay un único servidor de base de datos compartido para todos los clientes, cada uno de ellos debe tener su propia base de datos lógica con sus respectivas credenciales de acceso. Por ello este módulo ejecuta un trabajo de inicialización, en el que un contenedor se conecta a la base de datos, crea el usuario, contraseña y base de datos lógica, y finalmente termina. Este módulo está compuesto por los siguientes recursos:

- **Trabajo**: lanza un contenedor que realiza una ejecución temporal. Al contrario que un servicio que debe permanecer en ejecución, este contenedor finaliza. Se utiliza para ejecutar el script de creación de usuario en *MySQL*.
- **Mapa de configuración**: en este caso, se utiliza este elemento para guardar el script que ejecutará el trabajo.
- **Secreto**: contiene las credenciales del usuario raíz utilizados para la creación de los usuarios en la base de datos.

En la Figura 19, se muestran estos elementos. En este caso el trabajo también utiliza variables de entorno obtenidas de un mapa de configuración que es creado por el componente global, indicado en la figura con bordes punteados.

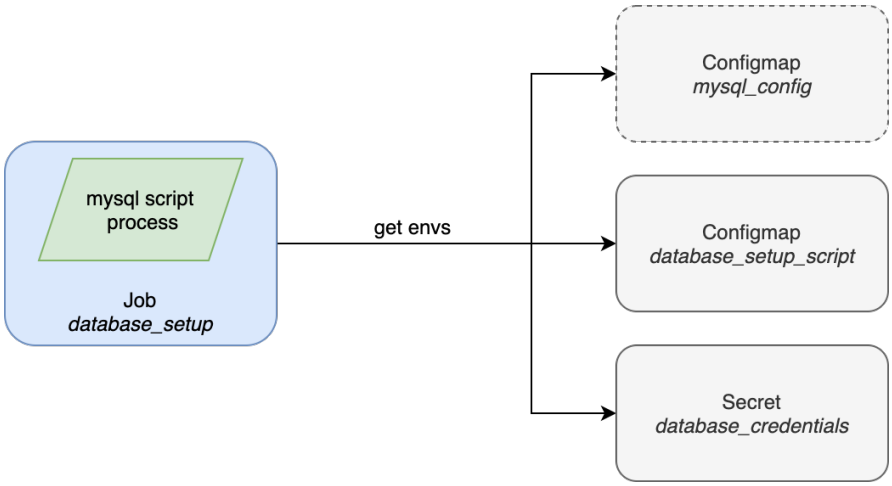

Figura 19 Recursos del módulo de inicialización de base de datos

Las variables de entrada para este módulo son las siguientes:

- Credenciales del usuario a crear en la base de datos.
- Imagen *Docker* de *MySQL* con la que ejecutar el script de creación de usuario.

6.3.2 Módulo principal y variables

Una vez desarrollados los módulos de *Terraform* que describen cada uno de los elementos de la infraestructura, llega el momento de seguir la organización del código propuesta en el marco de trabajo.

El módulo principal debe contener las referencias a estos módulos, que se encontrarán en ficheros del componente global o de usuario en función de si deben compartirse o cada usuario debe tener su propia copia de los recursos. Dado que el servidor de base de datos debe ser compartido por los distintos clientes[8], se asociará al componente global, mientras que los servidores WordPress y las bases de datos lógicas se asociarán

[8] En este caso de uso se realiza una equivalencia entre cliente y usuario, de forma que cada cliente de la empresa tiene asignado un componente de usuario.

a los componentes de usuario, tal y como se muestra en la Figura 20.

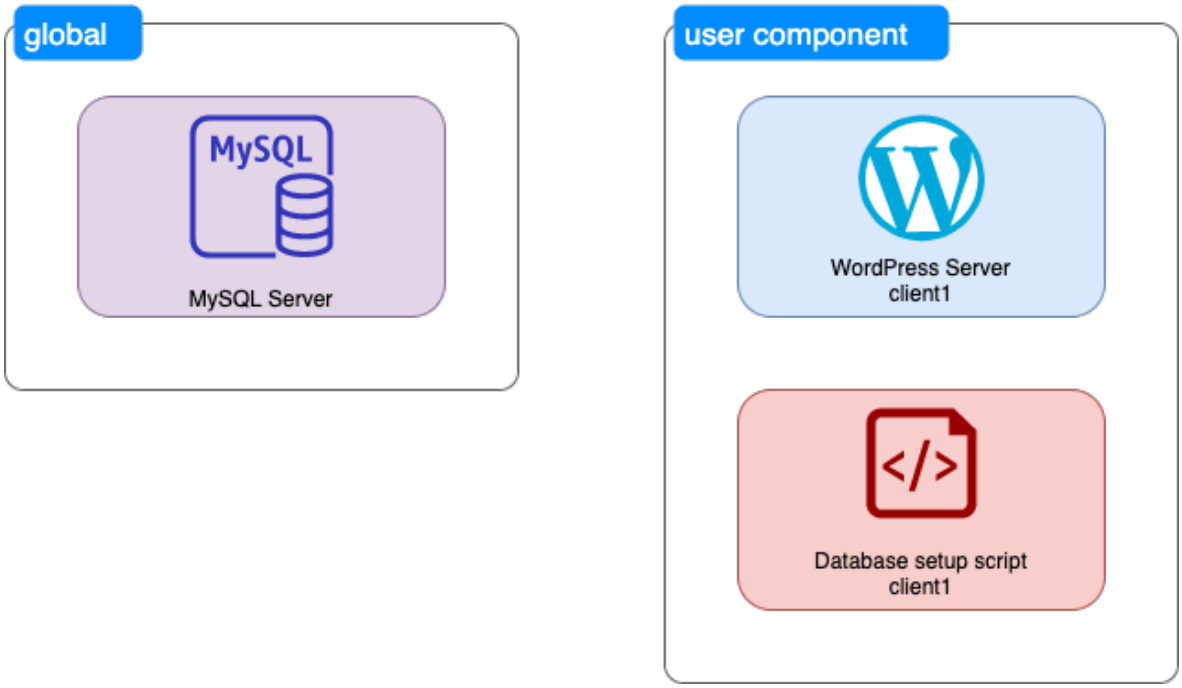

Figura 20 Módulos y componentes del caso de uso

Para cumplir el requisito de tener tamaños de servidores diferentes en función del cliente o del entorno, se utilizan las variables de tipo "sabor" que propone *Sonatina*. Esto permite definir varios tamaños tanto para el componente global como los componentes de usuario. En las siguientes tablas se clasifican las variables del caso de uso en función de su tipo y se justifica el por qué se declaran en ese tipo concreto.

Tabla 10 Variables para el componente global del caso de uso

Tipo	Variables	Justificación
Estáticas	Imagen del servidor *MySQL*	La imagen va ligada a la versión del código de la infraestuctura, por lo que obtiene un valor estático en cada versión.
Sabor	Requisitos y limites de CPU y RAM. Tamaño del volumen de almacenamiento.	En función del sabor elegido, se asignarán más o menos recursos a la base de datos.
Configurables	Contraseña del usuario raíz	El usuario debe configurar obligatoriamente la contraseña en cada despliegue.

En el caso de los componentes de usuario:

Tabla 11 Variables del componente de usuario del caso de uso

Tipo	Variables	Justificación
Estáticas	Imagen del servidor *Wordpress* y del cliente *MySQL*.	Las imágenes van ligadas a la versión del código de la infraestuctura, por lo que obtiene unos valores estáticos en cada versión.
Sabor	Requisitos y limites de CPU y RAM para los contenedores de	En función del sabor elegido, se asignarán más o menos recursos, y más o menos réplicas a los

	Wordpress. Número de replicas del servidor de *Worpress*.	servidores de *Wordpress*.
Configurables	Nombre del componente de usuario. Contraseña del usuario de la base de datos.	Se debe configurar el nombre del componente de usuario para distinguir los recursos de cada uno. La contraseña también debe ser obligatoriamente configurada en cada despliegue.

El mapeo entre las variables definidas y organizadas a través de *Sonatina* y las variables que aceptan los módulos de *Terraform* se realiza en las referencias a estos últimos desde el módulo principal. A continuación, se muestra como ejemplo la referencia al módulo del servidor de la base de datos.

Código 5 Referencia al módulo *database* del caso de uso

```
module "database" {
  source = "../../modules/database-server"

  root_password = var.root_password
  mysql_image   = var.mysql_image

  cpu_request = var.database_cpu_request
  cpu_limit   = var.database_cpu_limit
  ram_request = var.database_ram_request
  ram_limit   = var.database_ram_limit
  storage     = var.database_storage
}
```

6.3.3 Complemento para soportar HTTPS

En el apartado 3.1.3 se mostraba cómo el marco de trabajo plantea el soporte para añadir o modificar funcionalidades a un despliegue base. En este caso se va a utilizar para añadir a algunos componentes de usuario el soporte para conexiones cifradas con HTTPS.

Como se muestra en la figura, el certificado será creado en el componente global, y almacenado en un secreto. Cada componente de usuario desplegará un proxy inverso con *Nginx*, tomando como certificado el almacenado en el secreto del componente global.

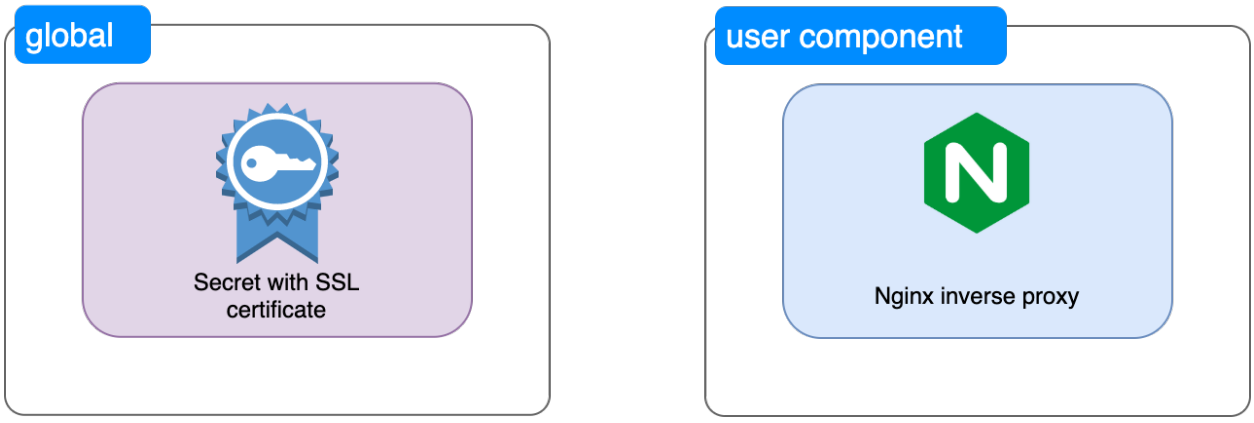

Figura 21 Módulos del complemento del caso de uso

Los complementos añaden al directorio de trabajo de *Terraform* nuevos ficheros .TF, que pueden llamar a nuevos módulos o sustituir módulos existentes. Al aplicarse en conjunto, el código del complemento puede utilizar variables y referencias a recursos como si estuviera definido en el despliegue base. Esta característica se utiliza en el complemento para obtener tanto la variable que indica qué componente de usuario se está desplegando, como para obtener el nombre del servicio al que redirigir las peticiones.

Código 6 Referencia al módulo *proxy* del caso de uso

```
module "proxy" {
  user_component = var.user_component
  backend_service = module.wordpress.service_name

  ...
}
```

Ninguna de las dos variables está definida en el complemento, pero son accesibles al estar definidas en el código base.

El código del complemento puede estar en un repositorio diferente al del código base de la infraestructura. El de este caso de uso se puede encontrar en el repositorio https://github.com/arodriguezdlc/sonatina-example-wordpress-kubernetes-ssl-plugin.git.

6.4 Despliegue del entorno de desarrollo

Una vez tenemos diseñada la solución e implementado el código de los distintos módulos de *Terraform*, se puede utilizar *Sonatina* para realizar el despliegue del entorno de desarrollo. Como ya se mencionó anteriormente, el código se encuentra en el repositorio https://github.com/arodriguezdlc/sonatina-example-wordpress-kubernetes.

En primer lugar, se necesita crear un repositorio GIT vacío para almacenar el estado y las variables de *Sonatina*. En este caso se creará un repositorio en GitHub, aunque podría utilizarse cualquier repositorio GIT. Las instrucciones para su creación se pueden encontrar en la referencia [10]. En este caso el repositorio creado tiene la referencia de GIT:

```
git@github.com:arodriguezdlc/wordpress-devel.git
```

Una vez se tiene el repositorio de estado, se procederá a la creación de un nuevo despliegue de *Sonatina*. Para ello hay que indicar la URL del repositorio con el código que describe la infraestructura (*code-repo-uri)* y del recién creado repositorio de estado[9] (*storage-repo-uri*). Además, como argumento se debe indicar un nombre. En este caso se ha elegido "devel", al tratarse del entorno de desarrollo. El comando para realizar esta creación del despliegue sería el siguiente:

```
$ sonatina create deployment devel \
  --storage-repo-uri git@github.com:arodriguezdlc/wordpress-devel.git \
  --code-repo-uri https://github.com/arodriguezdlc/sonatina-example-local-docker.git \
  --terraform-version 0.13.4
```

Internamente clonará el repositorio con el código, e inicializará el repositorio de estado. A continuación se listarán los despliegues de *Sonatina* para comprobar que se ha creado correctamente:

```
$ sonatina list deployments
DEPLOYMENTS:
 * devel
```

Para continuar trabajando con el despliegue se debe ejecutar el flujo de trabajo de inicialización. Esto se encargará de procesar los CTD base y de complementos que hayan sido añadidos (por ahora ninguno), y generará el directorio de trabajo desde el que ejecutar *Terraform*. También se realizará el procesado de variables para poder editarlas. El comando que lanza este flujo es el siguiente:

```
$ sonatina init
```

Si la infraestructura tiene variables que hayan sido definidas como configurables, es el momento de editarlas para asignarles el valor deseado. Para ello, *Sonatina* ofrece el comando `sonatina edit`, que abre un editor de texto para configurar estas variables. En este caso, únicamente hay que configurar el valor de la contraseña para el usuario administrador de la base de datos para el componente global. El fichero quedaría así:

[9] En el caso del repositorio de estado, sólo se soportan URL de tipo SSH, debido a la necesidad de autenticación para escribir en él.

```
$ sonatina edit
# Config global vars (This is a template file)

root_password = 1234
```

También se puede elegir el sabor para el despliegue del componente, aunque por defecto *Sonatina* utilizará el sabor "default". Con el siguiente comando se pueden listar los sabores definidos para la infraestructura actual, quedando marcado con asterisco el sabor seleccionado actualmente:

```
$ sonatina list flavours
FLAVOURS:
 * default
 - small
 - big
```

Dado que se está desplegando un entorno de desarrollo, se podría cambiar al sabor pequeño. De esa manera, se utilizarán las variables definidas en el fichero de ese sabor, que contiene valores reducidos para los tamaños de los recursos.

```
$ sonatina set flavour small
Configured
```

Una vez que configuradas las variables, se procederá a la aplicación de la descripción de la infraestructura. Para ello se utiliza el flujo de aplicación, implementado con el comando `sonatina apply`. Este comando acepta un mensaje descriptivo para añadirlo al commit en el repositorio de estado. La ejecución se muestra a continuación:

```
$ sonatina apply "Despliegue inicial del componente global"
Initializing modules...

Initializing provider plugins...
- Using previously-installed hashicorp/kubernetes v1.13.3

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, we recommend adding version constraints in a required_providers block
in your configuration, with the constraint strings suggested below.

* hashicorp/kubernetes: version = "~> 1.13.3"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
module.database.kubernetes_secret.mysql_root_credentials: Creating...
module.database.kubernetes_secret.mysql_root_credentials: Creation complete after 0s
[id=default/mysql-root-credentials]
module.database.kubernetes_stateful_set.mysql: Creating...
module.database.kubernetes_stateful_set.mysql: Still creating... [10s elapsed]
module.database.kubernetes_stateful_set.mysql: Still creating... [20s elapsed]
module.database.kubernetes_stateful_set.mysql: Still creating... [30s elapsed]
module.database.kubernetes_stateful_set.mysql: Still creating... [40s elapsed]
module.database.kubernetes_stateful_set.mysql: Creation complete after 46s
[id=default/mysql]
module.database.kubernetes_service.mysql: Creating...
module.database.kubernetes_service.mysql: Creation complete after 0s
[id=default/mysql]
module.database.kubernetes_config_map.mysql_config: Creating...
module.database.kubernetes_config_map.mysql_config: Creation complete after 0s
[id=default/mysql-config]
```

```
Apply complete! Resources: 4 added, 0 changed, 0 destroyed.
```

Se puede observar en la salida del comando el progreso de creación de los recursos descritos. Para verificar que se ha realizado la aplicación correctamente, se listan los recursos creados en *Kubernetes*:

```
$ kubectl get all
NAME          READY   STATUS    RESTARTS   AGE
pod/mysql-0   1/1     Running   0          2m33s

NAME                 TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
service/kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP    4d8h
service/mysql        ClusterIP   10.109.103.7   <none>        3306/TCP   107s

NAME                     READY   AGE
statefulset.apps/mysql   1/1     2m33s
```

Una vez desplegado el componente global, se pueden desplegar los componentes de usuario. Para ello primero hay que crearlos en *Sonatina*. Se crearán los componentes de usuario "client1" y "client2":

```
$ sonatina create usercomponent client1
Created
$ sonatina create usercomponent client2
Created
```

Listándolos se puede verificar que se han creado correctamente:

```
$ sonatina list usercomponents
USER COMPONENTS:
client1
client2
```

Tras crear los componentes de usuario, se debe realizar el mismo procedimiento que con el componente global, pero especificando el componente de usuario concreto con la opción -c. Primero hay que inicializar el despliegue con el comando `sonatina init`, Después se editan las variables de configuración y finalmente se aplican los cambios a la infraestructura. En este caso, es necesario indicar el en la variable `user_component` el componente de usuario que se está desplegando. El fichero de variables quedaría de la siguiente forma:

```
$ sonatina init -c client1
$ sonatina edit -c client1
# Config user vars (This is a template file)

user_component = "client1"
database_password = "client1-password"
```

La ejecución del flujo de aplicación se muestra a continuación:

```
$ sonatina apply -c client1 "Despliegue inicial del componente de usuario client1"
Initializing modules...

Initializing provider plugins...
- Using previously-installed hashicorp/kubernetes v1.13.3

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, we recommend adding version constraints in a required_providers block
in your configuration, with the constraint strings suggested below.

* hashicorp/kubernetes: version = "~> 1.13.3"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
```

```
commands will detect it and remind you to do so if necessary.
module.database_user.kubernetes_config_map.database_setup_script: Creating...
module.database_user.kubernetes_secret.database_credentials: Creating...
module.database_user.kubernetes_secret.database_credentials: Creation complete after
0s [id=default/database-credentials-client1]
module.database_user.kubernetes_config_map.database_setup_script: Creation complete
after 0s [id=default/database-setup-script-client1]
module.database_user.kubernetes_job.database_setup: Creating...
module.database_user.kubernetes_job.database_setup: Creation complete after 2s
[id=default/database-setup-client1]
module.wordpress.kubernetes_config_map.wordpress_database: Creating...
module.wordpress.kubernetes_secret.wordpress_database: Creating...
module.wordpress.kubernetes_secret.wordpress_database: Creation complete after 0s
[id=default/wordpress-database-client1]
module.wordpress.kubernetes_config_map.wordpress_database: Creation complete after 0s
[id=default/wordpress-database-client1]
module.wordpress.kubernetes_deployment.wordpress: Creating...
module.wordpress.kubernetes_deployment.wordpress: Creation complete after 1s
[id=default/wordpress-client1]
module.wordpress.kubernetes_service.wordpress: Creating...
module.wordpress.kubernetes_service.wordpress: Creation complete after 0s
[id=default/wordpress]

Apply complete! Resources: 7 added, 0 changed, 0 destroyed.
```

Se podrían desplegar tantos componentes de usuario como fueran necesarios, teniendo cada uno su propio servidor *WordPress* y conectando con la base de datos *MySQL* común.

Para comprobar que *WordPress* funciona correctamente, se puede mapear el puerto del servicio a un puerto local con el comando:

```
$ kubectl port-forward svc/wordpress-client2 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
Handling connection for 8080
Handling connection for 8080
```

Este comando crea un túnel entre el puerto 8080 de la máquina local y el puerto 80 del servicio de *Kubernetes* para WordPress. En la Figura 22 se muestra el resultado del acceso a través de la URL http://localhost:8080.

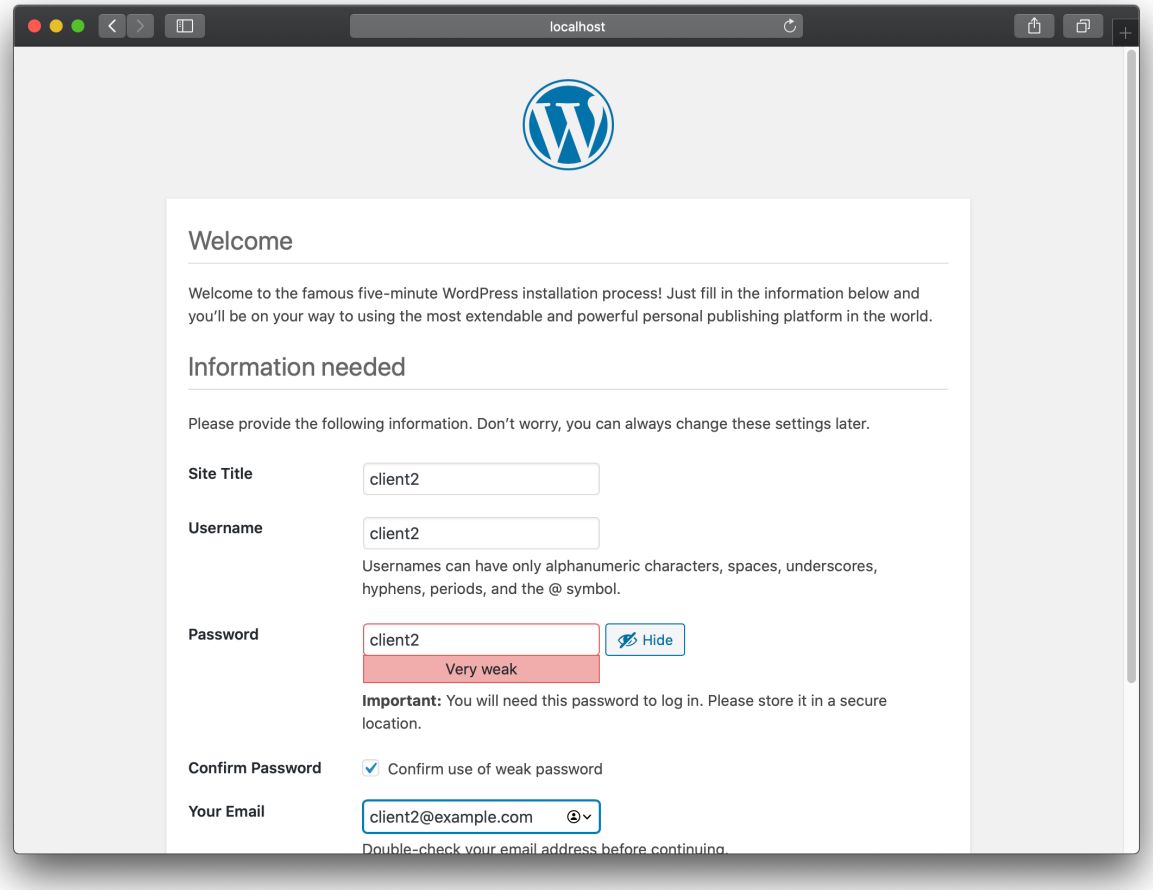

Figura 22 Acceso al WordPress desde el navegador

6.5 Uso sabores por cliente

En el apartado anterior ya se configuró para el componente global un sabor pequeño, al tratarse de un entorno de desarrollo. Sin embargo, los sabores también se pueden configurar individualmente por componente de usuario, de forma que un cliente podría tener contratado un sabor grande, y otro uno pequeño.

El procedimiento para aplicar el cambio de sabor a un componente de usuario es el mismo que el ya explicado, sólo que especificando con la opción -c el componente de usuario a configurar. Posteriormente, habría además que aplicar los cambios con el comando `sonatina apply`.

A modo de ejemplo, se va a asignar al componente de usuario "client1" el sabor grande. Los comandos se muestran a continuación:

```
$ sonatina set flavour big -c client1
Configured

$ sonatina apply -c client1 "Sabor grande para client2"
Initializing modules...

Initializing provider plugins...
- Using previously-installed hashicorp/kubernetes v1.13.3

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, we recommend adding version constraints in a required_providers block
```

```
in your configuration, with the constraint strings suggested below.

* hashicorp/kubernetes: version = "~> 1.13.3"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
module.database_user.kubernetes_config_map.database_setup_script: Refreshing state...
[id=default/database-setup-script-client1]
module.database_user.kubernetes_secret.database_credentials: Refreshing state...
[id=default/database-credentials-client1]
module.database_user.kubernetes_job.database_setup: Refreshing state...
[id=default/database-setup-client1]
module.wordpress.kubernetes_secret.wordpress_database: Refreshing state...
[id=default/wordpress-database-client1]
module.wordpress.kubernetes_config_map.wordpress_database: Refreshing state...
[id=default/wordpress-database-client1]
module.wordpress.kubernetes_deployment.wordpress: Refreshing state...
[id=default/wordpress-client1]
module.wordpress.kubernetes_service.wordpress: Refreshing state...
[id=default/wordpress]
module.wordpress.kubernetes_deployment.wordpress: Modifying... [id=default/wordpress-
client1]
module.wordpress.kubernetes_deployment.wordpress: Modifications complete after 0s
[id=default/wordpress-client1]

Apply complete! Resources: 0 added, 1 changed, 0 destroyed.
```

Es interesante observar que, al ya existir el componente de usuario, *Terraform* sólo ha aplicado los cambios necesarios para llevar la infraestructura al estado deseado. Es decir, sólo ha cambiado los contenedores de *WordPress* por otros con más recursos, pero no las configuraciones ni los servicios.

6.6 Adición del complemento para SSL

Con el complemento desarrollado para esta infrastructura que se describía en el apartado 6.3.3, se puede añadir a componentes de usuario específicos acceso mediante HTTPS a su servidor de WordPress. Los complementos deben ser añadidos primero al componente global y luego al de usuario, ya que en general, los componentes de usuario podrían depender de algún recurso del componente global.

El código de los complementos se puede definir en repositorios diferente al del código base, por lo que al añadirlo hay que especificar la URL del repositorio. En este caso, el código del complemento se encuentra en el repositorio https://github.com/arodriguezdlc/sonatina-example-wordpress-kubernetes-ssl-plugin.git

Para añadir el complemento al componente global, se utiliza el siguiente comando:

```
$ sonatina create plugin ssl -r https://github.com/arodriguezdlc/sonatina-example-
wordpress-kubernetes-ssl-plugin.git
Created
```

Si el plugin requiriese la configuración de variables, se podría utilizar también el comando `sonatina edit` con la opción -p y el nombre del plugin:

```
$ sonatina edit -p ssl
```

En este caso, no es necesario porque el complemento no añade ninguna variable configurable. Tras esto, se aplicarían los cambios:

```
$ sonatina apply "Complemento SSL añadido al componente global"
Initializing modules...
- certificate in ../../modules/certificate

Initializing provider plugins...
- Using previously-installed hashicorp/kubernetes v1.13.3
- Finding latest version of hashicorp/tls...
- Installing hashicorp/tls v3.0.0...
- Installed hashicorp/tls v3.0.0 (signed by HashiCorp)

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, we recommend adding version constraints in a required_providers block
in your configuration, with the constraint strings suggested below.

* hashicorp/kubernetes: version = "~> 1.13.3"
* hashicorp/tls: version = "~> 3.0.0"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
module.database.kubernetes_secret.mysql_root_credentials: Refreshing state...
[id=default/mysql-root-credentials]
module.database.kubernetes_stateful_set.mysql: Refreshing state... [id=default/mysql]
module.database.kubernetes_service.mysql: Refreshing state... [id=default/mysql]
module.database.kubernetes_config_map.mysql_config: Refreshing state...
[id=default/mysql-config]
module.certificate.tls_private_key.key: Creating...
module.certificate.tls_private_key.key: Creation complete after 0s
[id=b5f767cce0a0de7ff3e87accf233b8e691818184]
module.certificate.tls_self_signed_cert.cert: Creating...
module.certificate.tls_self_signed_cert.cert: Creation complete after 0s
[id=163560560616772396455577285392050702l1]
module.certificate.kubernetes_secret.certificate: Creating...
module.certificate.kubernetes_secret.certificate: Creation complete after 0s
[id=default/ssl-certificate]
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
```

De nuevo se puede observar que *Terraform* únicamente aplica los cambios necesarios, teniendo encuenta la infraestructura ya desplegada.

Ya solo faltaría añadir el complemento al componente de usuario correspondiente. Suponiendo que se quiere dar soporte para HTTPS al componente de usuario “client1”, los comandos para realizarlo serían los siguientes:

```
$ sonatina create plugin ssl -c client1
Created
$ sonatina apply "Complemento SSL añadido al componente de usuario client1"
```

Listando los recursos desplegados en *Kubernetes*, se puede observar la existencia del nuevo despliegue de *NGINX* que ofrece la conectividad HTTPS:

```
$ kubectl get deployment
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
nginx-client1       1/1     1            1           101m
wordpress-client1   1/1     1            1           7h58m
wordpress-client2   1/1     1            1           7h47m
```

6.7 Despliegue del entorno de producción

Para desplegar el entorno de producción con *Sonatina*, basta con crear un nuevo despliegue, que tendrá que utilizar un nuevo repositorio de estado. En este caso se creará en github el repositorio `git@github.com:arodriguezdlc/wordpress-prod.git`, y con el siguiente comando, el despliegue en *Sonatina*:

```
sonatina create deployment prod \
  --storage-repo-uri git@github.com:arodriguezdlc/wordpress-prod.git \
  --code-repo-uri https://github.com/arodriguezdlc/sonatina-example-local-docker.git \
  --terraform-version 0.13.4

Created
```

A partir de aquí, se podrían seguir los mismos pasos para realizar el despliegue en desarrollo. Se podrían añadir complementos, elegir sabores, y crear componentes de usuario exactamente de la misma manera, basándose en el mismo código y en la misma herramienta de gestión.

6.8 Gestión compartida del despliegue

Gracias a la integración con GIT, el despliegue puede ser gestionado por múltiples administradores que tengan acceso al repositorio de estado de *Sonatina*. Para ello basta con clonar el despliegue con el siguiente comando:

```
$ sonatina clone deployment wordpress-devel \
  --storage-repo-uri git@github.com:arodriguezdlc/wordpress.git
Cloned
```

Esto clonará el repositorio de estado y a partir de la información almacenada en los metadatos descargará automáticamente todos los repositorios de código necesarios, incluyendo los complementos. Una vez clonado, también se puede volver a sincronizar con el comando

```
$ sonatina refresh
```

6.9 Conclusiones

Gracias a la organización del código definida en el marco de trabajo y a *Sonatina*, se ha logrado disponer de una herramienta que cumple con los requisitos de la empresa.

El desarrollo del código que describe la infraestructura debe ser abordado igualmente, como con cualquier otra herramienta de IaC, pero *Sonatina* aporta una capa de gestión que evita la necesidad de mantener código específico para cada despliegue, y facilita el desarrollo y la administración cooperativa de los entornos.

Si bien este caso de uso no es muy complejo, sí que hace explícitas las necesidades de muchas empresas de mantener diferentes entornos con configuraciones diferentes, tratando de reducir al máximo el esfuerzo en gestión. Estas necesidades son las que *Sonatina* aborda tal y como se ha mostrado en este capítulo.

7 CONCLUSIONES Y LÍNEAS DE CONTINUACIÓN

Este proyecto propone una forma concreta de utilizar *Terraform*, que ayuda a la organización del código de la infraestructura y al desarrollo de herramientas que extienden su funcionalidad, como es el caso de *Sonatina*. Esta utilidad logra añadir un conjunto de funcionalidades muy útiles para las compañías que necesitan gestionar múltiples despliegues de sus infraestructuras, con características variadas.

El despliegue de entornos similares pero personalizados, la posibilidad de ajustar sus tamaños para optimizar los costes, el soporte para compartir partes de la infraestructura entre clientes o el control de versiones para la gestión colaborativa son características de gestión muy valoradas por las empresas, que logra implementar *Sonatina*. Si bien es habitual que cada una desarrolle sus propias herramientas específicas para resolver estos problemas, *Sonatina* los aborda de forma genérica y libre, permitiendo mediante su uso que las compañías puedan reducir su esfuerzo en gestión de infraestructuras y se centren en el desarrollo de sus productos, que son los que les aportan valor.

También se ha propuesto un caso de uso que, siendo bastante más simple que los que hacen imprescindible el uso de este tipo de soluciones, ilustra muy bien su funcionamiento sin que el lector se pierda en los detalles de otras plataformas más complejas.

Finalmente el trabajo, además de aportar soluciones empresariales útiles e interesantes, también logra el objetivo académico de desarrollar un proyecto de software basado en las últimas tecnologías del mundo de las telecomunicaciones y la informática, incrementando la experiencia y los conocimientos del autor.

7.1 Líneas de continuación

Además de lograr los objetivos propuestos, el trabajo sienta las bases para implementar nuevas funcionalidades y mejoras a las ya propuestas, que aumenten las posibilidades y la flexibilidad en la gestión de los despliegues. *Sonatina* es un software en evolución, aunque en esta memoria se haya descrito su estado en un momento concreto.

Se proponen las siguientes mejoras y líneas de continuación

- Creación de una **API REST** como nueva interfaz de usuario para *Sonatina* adicional al ya implementado en forma de interfaz CLI. Esta funcionalidad podría ser especialmente útil para compañías que quieran ofrecer el servicio de desplegar infraestructuras o componentes de usuario a través de una API.
- **Bloqueo de estado** basado en GIT. Como ya se exponía en el trabajo, una de las funcionalidades deseables (pero no implementadas) era el bloqueo del estado para evitar que varias ejecuciones simultáneas de *Sonatina* entraran en conflicto. Sería fundamental en caso de querer implementar la API que se proponía en el punto anterior.
- Generación de **esqueletos de código** automático: sería muy útil que *Sonatina* generase las estructuras básicas del código de forma automática, para facilitar su adopción.

- **Retroceso** a un estado anterior: una funcionalidad muy demandada por las organizaciones es el poder retroceder a un estado anterior de la infraestructura. *Sonatina* podría implementarlo gracias al histórico de estados que almacena en GIT.
- **Compartición de variables** entre el componente global y los componentes de usuarios: sería útil poder acceder a variables definidas en el componente global y que podrían ser compartidas para los componentes de usuario.
- **Templatizado de ficheros de *Terraform***: generación de ficheros de *Terraform* a partir de plantillas, permitiendo así generar valores dinámicamente como por ejemplo las rutas a los módulos, y aumentar la personalización del código.
- **Gestión de señales**: para evitar pérdida de datos en caso de recibir una señal de interrupción en el programa.

ANEXO A: MANUAL DE REFERENCIA DE SONATINA

A.1 Instalación

El código fuente de *Sonatina* se encuentra el repositorio de GitHub https://github.com/arodriguezdlc/sonatina.

Para clonarlo, se puede utilizar el siguiente comando:

```
$ git clone https://github.com/arodriguezdlc/sonatina.git
```

Sonatina está desarrollado en *Golang*, y su compilador es capaz de obtener automáticamente las dependencias externas. Para compilarlo, es suficiente con ejecutar los siguientes comandos:

```
$ cd sonatina
$ go build
```

Como resultado se generará un binario con nombre "Sonatina". Puede ser instalado en algún directorio dentro del *Path* por defecto del sistema:

```
$ mv sonatina /usr/local/bin/
```

A.2 Conceptos

- **Árbol de definición de Código (*Code Tree Definition*, CTD)**: estructura de directorios y ficheros de código HCL estandarizada por *Sonatina*. Permite combinarse con otros CTD para habilitar la implementación de complementos al código de la infraestructura base.
- **Árbol de definición de Variables (*Variables Tree Definition*, VTD):** estructura de ficheros de variables que permite a *Sonatina* identificarlas por tipos y aplicarlas con una determinada prioridad.
- **Despliegue (Deployment)**: infraestructura concreta a gestionar basada en el código definido en los CTDs.
- **Módulos**: son bloques de código de *Terraform*. Cuando se hace referencia a módulos que no son el principal, se trata de módulos estándar, sin ninguna restricción impuesta por *Sonatina* en su implementación.
- **Repositorio de Código**: respositorio de GIT donde se encuentra el código que describe la infraestructura siguiendo la estructura del CTD.
- **Repositorio de Estado**: repositorio de GIT donde se almacena el estado y las variables de un despliegue concreto.
- **Directorio de trabajo**: carpeta temporal en la que *Sonatina* mezcla los CTDs del código base y los complementos para dar como resultado la descripción final a aplicar por parte de *Terraform*.
- **Variables**: parámetros que acepta la descripción de la infraestructura, para su personalización

- **Estado (state)**: fichero que registra cuál es el estado de los recursos desplegados por *Terraform*. Habrá un fichero de estado por cada componente, ya sea global o de usuario.
- **Componentes de usuario (user components)**: conjunto de recursos que se despliegan de forma específica para un usuario. Pueden desplegarse múltiples de ellos en cada despliegue. Los recursos que no pertenecen de forma específica a un usuario son los que forman el componente global.
- **Sabor (flavour)**: identifica el uso de un conjunto de variables que forman ese sabor. Normalmente se utilizan con variables que definen el tamaño de la infraestructura, permitiendo elegir entre sabores de infraestructura con más o menos recursos.
- **Complementos (plugins)**: son CTDs que se combinan con el CTD del código base para incrementar o modificar las funcionalidades de la infraestructura.

A.3 Guía básica de uso

A.3.1 Creación de un nuevo despliegue

Para crear un nuevo despliegue se necesita un repositorio donde almacenar su estado y sus variables. Utilizando el siguiente enlace se podría crear uno en GitHub. Nótese que debe ser privado, ya que podría almacenar información sensible sobre el despliegue:

https://github.com/new

También se necesita tener el código base con la estructura del CTD. Para esta guía se utilizará un CTD de "hola mundo", publicado en el repositorio https://github.com/arodriguezdlc/sonatina-hello-world.

Para crear el despligue se utilizaría el siguiente comando:

```
$ sonatina create deployment hello-world -c https://github.com/arodriguezdlc/sonatina-hello-world.git  -s <REPOSITORIO_DE_ESTADO>
```

Nótese que la URL del repositorio de estado debe ser de tipo SSH, para disponer de acceso autenticado que permita la escritura en el mismo.

Si un colaborador quisiera gestionar ese despliegue, podría añadirlo a *Sonatina* con el siguiente comando:

```
$ sonatina clone deployment hello-world -s <REPOSITORIO_DE_ESTADO>
```

Para verificar que se ha creado correctamente, se pueden listar despliegues añadidos:

```
$ sonatina list deployments
```

A.3.2 Aplicación de cambios a la infraestructura

En primer lugar hay que inicializar el despliegue con el siguiente comando:

```
$ sonatina init
```

Esta operación crea el directorio de trabajo, realiza el procesamiento de las variables e inicializa *Terraform*. Tras esto se pueden editar las variables de configuración con el comando, aunque en el caso del ejemplo no es necesario inicializar ninguna.

```
$ sonatina edit
```

Una vez configuradas, la aplicación de los cambios se realizaría con el comando mostrado a continuación:

```
$ sonatina apply "Despliegue del hola mundo"
```

A.3.3 Creación de un componente de usuario

La creación de nuevos componentes de usuario se realiza con el siguiente comando:

```
$ sonatina create usercomponent usuario1
```

Para poder aplicarlo a la infraestructura se tiene que realizar las mismas operaciones que para el componente global, pero especificando ese componente de usuario concreto:

```
$ sonatina init -c usuario1
$ sonatina edit -c usuario1
$ sonatina apply "Despliegue del componente de usuario" -c usuario1
```

A.3.4 Creación de un complemento

Para añadir un nuevo complemento a un despliegue, debe añadirse siempre en primer lugar al componente global, especificando el repositorio en el que se encuentra su CTD. Para el ejemplo se usará también un complemento al estilo "hola mundo", publicado en el repositorio https://github.com/arodriguezdlc/sonatina-plugin-hello-world.

El comando sería el siguiente:

```
$ sonatina create plugin hello-world -r https://github.com/arodriguezdlc/sonatina-plugin-hello-world.git
```

Una vez añadido al componente global, se puede añadir a un componente de usuario concreto sin necesidad de especificar el repositorio:

```
$ sonatina create plugin hello-world -c usuario1
```

El hecho de crear en *Sonatina* el complemento no implica que se hayan aplicado los cambios a la infraestructura. Para ello, hay que volver a ejecutar el comando de aplicación de cambios:

```
$ sonatina apply "añadido complemento al componente global"
$ sonatina apply -c usuario1 "añadido complemento a usuario1"
```

A.3.5 Cambio de sabor de un despliegue

Si los CTDs de la infraestructura definen varios sabores, se puede elegir cuál utilizar en el despliegue. En el ejemplo se configurará el flavour "big" con el siguiente comando:

```
$ sonatina set flavour big
```

De nuevo para aplicar los cambios:

```
$ sonatina apply "Cambio de sabor a big"
```

A.3.6 Eliminación de los recursos

La eliminación de los recursos de la infraestructura se realiza mediante el comando contrario al de aplicación (destroy). Se recomienda siempre destruir los componentes de usuario antes del componente global, aunque no es obligatorio. Para este ejemplo serían los siguientes comandos:

```
$ sonatina destroy -c usuario1 "Eliminacion de usuario1"
$ sonatina destroy "Eliminación de la infraestructura"
```

A.3.7 Eliminación del despliegue

Para eliminar localmente el despliegue, se utiliza el siguiente comando:

```
$ sonatina delete deployment hellow-world
```

Esto no elimina el respositorio de estado ni los recursos de la infraestructura (si no se han destruido explícitamente), solo elimina los ficheros locales. Para recuperarlos se puede utilizar el comando:

```
$ sonatina clone deployment hello-world -s <REPOSITORIO_DE_ESTADO>
```

Esto recuperaría toda la información necesaria para continuar gestionando el despliegue.

A.4 Referencia de comandos

A.4.1 Creación de un despliegue

Descripción:

Crea un nuevo despliegue que nunca ha sido inicializado anteriormente.

Comando:

```
$ sonatina create deployment <nombre_despliegue> [opciones]
```

Argumentos:

- `nombre_despliegue`: identifica el despliegue a crear.

Opciones:

- `-c, --code-repo-uri`: URI del repositorio de código donde se encuentra el CTD base que define la infraestructura. Es obligatorio.
- `-p, --code`: ruta dentro del repositorio de código donde se encuentra el CTD base que define la infraestructura. Por defecto se usa la raíz del repositorio.
- `-f, --flavour`: sabor a utilizar en el componente base del despliegue. Por defecto es “default”.
- `-s, --storage-repo-uri`: URI del repositorio de estado asignado a este despliegue. Debe ser un URI de tipo SSH para permitir el acceso autenticado.
- `-t, --terraform-version`: versión de Terraform a utilizar para gestionar este despliegue.

A.4.2 Clonado de un despliegue

Descripción:

Clona un despliegue que ya tiene un repositorio de estado inicializado.

Comando:

```
$ sonatina clone deployment <nombre_despliegue> [opciones]
```

Argumentos:

- `nombre_despliegue`: identifica el despliegue a clonar.

Opciones:

- `-s, --storage-repo-uri`: URI del repositorio de estado asignado a este despliegue. Debe ser un URI de tipo SSH para permitir el acceso autenticado.

A.4.3 Eliminación de un despliegue

Descripción:

Elimina un despliegue del entorno local. No elimina la infraestructura ni la información del repositorio de estado.

Comando:

```
$ sonatina delete deployment <nombre_despliegue>
```

Argumentos:

- `nombre_despliegue`: identifica el despliegue a eliminar.

A.4.4 Listado de despliegues

Descripción:

Lista los despliegues gestionados por *Sonatina*

Comando:

```
$ sonatina list deployments <nombre_despliegue>
```

A.4.5 Uso de un despliegue

Descripción:

Configura al despliegue indicado como despliegue en uso. Los siguientes comandos que actúan sobre un despliegue utilizarán ese por defecto.

Comando:

```
$ sonatina use deployment <nombre_despliegue>
```

Argumentos:

- `nombre_despliegue`: identifica el despliegue a usar.

A.4.6 Creación de un componente de usuario

Descripción:

Crea un nuevo componente de usuario para el despliegue.

Comando:

```
$ sonatina create usercomponent <componente_usuario> [opciones]
```

Argumentos:

- `componente_usuario`: nombre del componente de usuario a crear.

Opciones:

- `-d, --deployment`: despliegue al que aplicar la modificación. Por defecto utiliza el despliegue en uso actual.

A.4.7 Eliminación de un componente de usuario

Descripción:

Elimina el componente de usuario especificado del despliegue.

Comando:

```
$ sonatina delete usercomponent <componente_usuario> [opciones]
```

Argumentos:

- `componente_usuario`: nombre del componente de usuario a eliminar.

Opciones:

- `-d, --deployment`: despliegue al que aplicar la modificación. Por defecto utiliza el despliegue en uso actual.

A.4.8 Configuración del sabor

Descripción:

Configura un sabor para un componente. Si no se especifica ninguno, se configura para el componente global.

Comando:

```
$ sonatina set flavour <sabor> [opciones]
```

Argumentos:

- `sabor`: sabor a configurar.

Opciones:

- `-c, --user-component`: componente de usuario al que configurar el sabor. Si no se especifica ninguno, se configura al componente global.
- `-d, --deployment`: despliegue al que aplicar la modificación. Por defecto utiliza el despliegue en uso actual.

A.4.9 Obtención del sabor

Descripción:

Obtiene el sabor configurado para un componente. Si no se especifica ninguno, se obtiene el del componente global.

Comando:

```
$ sonatina get flavour [opciones]
```

Opciones:

- `-c, --user-component`: componente de usuario al que configurar el sabor. Si no se especifica ninguno, se configura al componente global.
- `-d, --deployment`: despliegue del que obtener el valor. Por defecto utiliza el despliegue en uso actual.

A.4.10 Creación de un complemento

Descripción:

Añade un complemento al despliegue, ya sea del componente global o del componente de usuario.

Comando:

```
$ sonatina create plugin <complemento> [opciones]
```

Argumentos:

- `complemento`: nombre del complemento a añadir.

Opciones:

- `-r, --repo-uri`: URI del repositorio de código donde se encuentra el CTD del complemento. Sólo se tiene que especificar (de forma obligatoria) en el caso del componente global.
- `-p, --repo-path`: ruta dentro del repositorio de código donde se encuentra el CTD del complemento. Por defecto se usa la raíz del repositorio.
- `-c, --user-component`: componente de usuario al que añadir el complemento. Si no se especifica ninguno, se añade al componente global. Nótese que para añadirlo a un componente de usuario debe haber sido añadido al global previamente.
- `-d, --deployment`: despliegue al que añadir el complemento. Por defecto utiliza el despliegue en uso actual.

A.4.11 Eliminación de un complemento

Descripción:

Elimina un complemento del despliegue, ya sea del componente global o del componente de usuario.

Comando:

```
$ sonatina delete plugin <complemento> [opciones]
```

Argumentos:

- `complemento`: nombre del complemento a eliminar.

Opciones:

- `-c, --user-component`: componente de usuario del que eliminar el complemento. Si no se especifica ninguno, se añade al componente global.
- `-d, --deployment`: despliegue del que eliminar el complemento. Por defecto utiliza el despliegue en uso actual.

A.4.12 Listado de complementos

Descripción:

Lista los complementos añadidos para un componente, ya sea global o de usuario.

Comando:

```
$ sonatina list plugins [opciones]
```

Opciones:

- `-c, --user-component`: componente de usuario del que listar los complementos. Si no se especifica ninguno, se listan del componente global.
- `-d, --deployment`: despliegue del que listar los complementos. Por defecto utiliza el despliegue en uso actual.

A.4.13 Inicialización del despliegue

Descripción:

Lanza el flujo de inicialización del despliegue para un componente. Esta operación genera el directorio de trabajo, preprocesa el directorio de variables e inicializa *Terraform*.

Comando:

```
$ sonatina init [opciones]
```

Opciones:

- `-c, --user-component`: componente de usuario al que aplicar el flujo. Si no se especifica ninguno, se aplica al componente global.
- `-d, --deployment`: despliegue al que aplicar el flujo. Por defecto utiliza el despliegue en uso actual.

A.4.14 Aplicación de cambios al despliegue

Descripción:

Lanza el flujo de aplicación de cambios al despliegue para un componente. Esta operación ejecuta el comando `terraform apply` con los argumentos adecuados, actualizando posteriormente el repositorio de estado con la información relativa a la nueva situación de la infraestructura.

Comando:

```
$ sonatina apply <mensaje> [opciones]
```

Argumentos:

- `mensaje`: cadena de texto con un mensaje descriptivo de los cambios a realizar. Se asignará este mensaje al nuevo *commit* del repositorio de estado.

Opciones:

- `-c, --user-component`: componente de usuario al que aplicar el flujo. Si no se especifica ninguno, se aplica al componente global.

- `-d, --deployment`: despliegue al que aplicar el flujo. Por defecto utiliza el despliegue en uso actual.

A.4.15 Destrucción del despliegue

Descripción:

Lanza el flujo de destrucción de la infraestructura de un componente. Esta operación ejecuta el comando `terraform destroy` con los argumentos adecuados, actualizando posteriormente el repositorio de estado con la información relativa a la nueva situación de la infraestructura.

Comando:

```
$ sonatina destroy <mensaje>[opciones]
```

Argumentos:

- `mensaje`: cadena de texto con un mensaje descriptivo de los cambios a realizar. Se asignará este mensaje al nuevo *commit* del repositorio de estado.

Opciones:

- `-c, --user-component`: componente de usuario al que aplicar el flujo. Si no se especifica ninguno, se aplica al componente global.
- `-d, --deployment`: despliegue al que aplicar el flujo. Por defecto utiliza el despliegue en uso actual.

A.4.16 Mostrado de variables

Descripción:

Muestra el fichero de variables en función del su tipo, y del componente ó complemento al que pertenece.

Comando:

```
$ sonatina show <tipo> [opciones]
```

Argumentos:

- `tipo`: nombre del tipo de variable a mostrar. Los valores válidos son config, flavour y static.

Opciones:

- `-c, --user-component`: componente de usuario del que mostrar las variables. Si no se especifica ninguno, se mostrarán las del componente global.
- `-d, --deployment`: despliegue al que aplicar el flujo. Por defecto utiliza el despliegue en uso actual.
- `-p, --plugin`: complemento del que mostrar las variables. Por defecto se muestran las del código base.

A.4.17 Edición de variables

Descripción:

Abre un editor de texto para modificar el fichero de variables correspondiente a un componente y un complemento determinado.

Comando:

```
$ sonatina edit [opciones]
```

Opciones:

- `-c, --user-component`: componente de usuario del que editar las variables. Si no se especifica ninguno, se editarán las del componente global.
- `-p, --plugin`: complemento del que editar las variables. Por defecto se editan las del código base.
- `-d, --deployment`: despliegue al que aplicar el flujo. Por defecto utiliza el despliegue en uso actual.

A.4.18 Actualización con el repositorio de estado

Descripción:

Actualiza la información de estado local con la del repositorio de estado remoto.

Comando:

```
$ sonatina refresh [opciones]
```

Opciones:

- `-d, --deployment`: despliegue a actualizar. Por defecto utiliza el despliegue en uso actual.

REFERENCIAS

[1] N. B. A. Ramtin Jabbari, «What is DevOps?: A Systematic Mapping Study on Definitions and Practices,» de *The Scientific Workshop XP2016*, 2016.

[2] Microsoft, «Página oficial de Visual Studio Code,» [En línea]. Available: https://code.visualstudio.com/.

[3] Golang, «Plugin de Go en VS Code Marketplace,» [En línea]. Available: https://marketplace.visualstudio.com/items?itemName=golang.Go.

[4] Hashicorp, «Plugin Terraform en VS Code Marketplace,» [En línea]. Available: https://marketplace.visualstudio.com/items?itemName=HashiCorp.terraform.

[5] Software Freedom Conservancy, «Página oficial de GIT,» [En línea]. Available: https://git-scm.com/.

[6] Docker Inc., «Documentación de Docker Swarm,» [En línea]. Available: https://docs.docker.com/engine/swarm/.

[7] The Linux Foundation, «Qué es Kubernetes,» 2020. [En línea]. Available: https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/.

[8] G. Nachman, «Página oficial de iTerm,» [En línea]. Available: https://iterm2.com/.

[9] Fishshell, «Página oficial de Fish Shell,» [En línea]. Available: https://fishshell.com/.

[10] GitHub, «Documentación de Github, creación de un repositorio,» [En línea]. Available: https://docs.github.com/es/free-pro-team@latest/github/getting-started-with-github/create-a-repo.

[11] GitHub Inc., «GitHub Actions,» [En línea]. Available: https://github.com/features/actions.

[12] Codecov, «Página oficial de Codecov,» [En línea]. Available: https://codecov.io/.

[13] S. Smith y H. Schaaf, «Web de Go Report Card,» [En línea]. Available: https://goreportcard.com/.

[14] Golang, «Repositorio de documentación GoDoc,» [En línea]. Available: https://godoc.org/.

[15] Python Software Foundation, «Web oficial de Python,» [En línea]. Available: https://www.python.org/.

[16] Ruby, «Página oficial del lenguaje Ruby,» [En línea]. Available: https://www.ruby-lang.org/.

[17] ECMA International, «The JSON Data Interchange Syntax,» 2017.

[18] YAML, «The Official YAML Web Site,» [En línea]. Available: https://yaml.org/.

[19] Chef, «Página oficial de Chef,» [En línea]. Available: https://www.chef.io/.

[20] Puppet, «Página oficial de Puppet,» [En línea]. Available: https://puppet.com/.

[21] Red Hat, «Página oficial de Ansible,» [En línea]. Available: https://www.ansible.com/.

[22] Amazon, «Página oficial de AWS Cloudformation,» [En línea]. Available: https://aws.amazon.com/es/cloudformation/.

[23] Openstack, «Documentación oficial de Openstack Heat,» [En línea]. Available: https://docs.openstack.org/heat/latest/.

[24] Hashicorp, «Página oficial de Terraform,» [En línea]. Available: https://www.terraform.io/.

[25] S. Lobo, «What is HCL (Hashicorp Configuration Language), how does it relate to Terraform, and why is it growing in popularity?,» Packt, 18 July 2019. [En línea]. Available: https://hub.packtpub.com/what-is-hcl-hashicorp-configuration-language-how-does-it-relate-to-terraform-and-why-is-it-growing-in-popularity/.

[26] Pulumi, «Página oficial de Pulumi,» [En línea]. Available: https://www.pulumi.com/.

[27] Gruntwork Inc., «Página oficial de Terragrunt,» [En línea]. Available: https://terragrunt.gruntwork.io/.

[28] CloudTools, «Repositorio de Troposphere en GitHub,» [En línea]. Available: https://github.com/cloudtools/troposphere.

[29] Red Hat, «Repositorio de ansible-runner-service en GitHub,» [En línea]. Available: https://github.com/ansible/ansible-runner-service.

[30] Amazon, «Página oficial de Amazon S3,» [En línea]. Available: https://aws.amazon.com/es/s3/.

[31] go-git, «Repositorio de go-git,» [En línea]. Available: https://github.com/go-git/go-git.

[32] spf13, «Repositorio de la librería Cobra,» [En línea]. Available: https://github.com/spf13/cobra.

[33] Apache Software Foundation, «Apache License, Version 2.0,» 2004. [En línea]. Available: https://www.apache.org/licenses/LICENSE-2.0.

[34] spf13, «Repositorio de la librería Viper,» [En línea]. Available: https://github.com/sp13/viper.

[35] spf13, «Repositorio de la librería Afero,» [En línea]. Available: https://github.com/spf13/afero.

[36] W. i. o. software, «What is opinionated software?,» 2008. [En línea]. Available: http://lavasoftware.net/what-is-opinionated-software/.

[37] R. Nayak, «When to use which Infrastructure-as-code tool,» Medium, 2019. [En línea]. Available: https://medium.com/cloudnativeinfra/when-to-use-which-infrastructure-as-code-tool-665af289fbde.

[38] K. Morris, Infrastructure as Code, O'Reilly Media, Inc., 2016.

[39] M. G. D. A. T. F. P. Michele Guerreiro, «Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry,» [En línea]. Available: https://fpalomba.github.io/pdf/Conferencs/C42.pdf.

[40] A. Gerrand, «Error Handling and Go,» 2011. [En línea]. Available: https://blog.golang.org/error-handling-and-go.

[41] P. U. Castro, «Programación orientada a objetos: Patrón Singleton,» 2018. [En línea]. Available: https://medium.com/@pablolloacastro/programaci%C3%B3n-orientada-a-objetos-patr%C3%B3n-singleton-423e2755614b.

[42] Y. Brikman, Terraform - Up and Running, 2nd Edition, O'Reilly Media, Inc., 2019.

[43] O. Ben-Kiki, C. Evans y I. d. Net, «YAML 1.2 specification,» 01 10 2009. [En línea]. Available: https://yaml.org/spec/1.2/spec.html.

[44] Weaveworks, «What is GitOps,» 21 Agosto 2018. [En línea]. Available: https://www.weave.works/blog/what-is-gitops-really.

[45] Cloud Native Foundation, «Web oficial de Kubernetes,» [En línea]. Available: https://kubernetes.io.

[46] Hashicorp, «Web oficial de Hashicorp,» [En línea]. Available: https://www.hashicorp.com/.

[47] Golang, «Repositorio de DEP en GitHub,» [En línea]. Available: https://github.com/golang/dep.

[48] WordPress, «Página oficial de WordPress,» [En línea]. Available: https://es.wordpress.org.

[49] Docker, «Página oficial de Docker,» [En línea]. Available: https://www.docker.com.

[50] Mozilla, «Mozilla Public License Version 2.0,» 2012. [En línea]. Available: https://www.mozilla.org/en-US/MPL/2.0/.

[51] Weaveworks, «Guide To GitOps,» [En línea]. Available: https://www.weave.works/technologies/gitops/.

[52] Github, Inc., «GitHub Actions,» [En línea]. Available: https://github.com/features/actions.

[53] Hashicorp, «Documentación oficial de Terraform,» [En línea]. Available: https://www.terraform.io/docs/.

[54] Hashicorp, «Documentación de Providers de Terraform,» [En línea]. Available: https://www.terraform.io/docs/providers/.

[55] Hashicorp, «Backend documentation,» [En línea]. Available: https://www.terraform.io/docs/backends/index.html.

[56] S. Roze, «Why I think we should all use immutable Docker Images,» 2015. [En línea]. Available: https://medium.com/sroze/why-i-think-we-should-all-use-immutable-docker-images-9f4fdcb5212f.