UNIVERSIDAD
NEBRIJA

DESARROLLO DE UNA APLICACIÓN WEB Y ESTUDIO DE HERRAMIENTAS PARA EL DESPLIEGUE DE SU INFRAESTRUCTURA

UNIVERSIDAD NEBRIJA

GRADO EN INGENIERÍA INFORMÁTICA

MEMORIA PRÁCTICAS EN EMPRESA

Óscar Salvador Sotoca

Enero/2023

UNIVERSIDAD
NEBRIJA

DESARROLLO DE UNA APLICACIÓN WEB Y ESTUDIO DE HERRAMIENTAS PARA EL DESPLIEGUE DE SU INFRAESTRUCTURA

UNIVERSIDAD NEBRIJA

GRADO EN INGENIERÍA INFORMÁTICA

MEMORIA PRÁCTICAS EN EMPRESA

Óscar Salvador Sotoca

Enero/2023

Tutor académico: Carlos Castellanos Manzaneque

Índice

1. Experiencia en empresa **3**

1.1. Introduccion, mis responsabilidades 3

1.2. Onboarding 3

1.3. Organigrama 3

2. Proyecto **4**

2.1. Antecedentes 4

2.2. Estudio del problema 4

2.3. Objetivos 4

3. Aplicación **6**

3.1. Inicio de sesión 7

3.2. CRUD de contenido de usuarios 10

4. Infraestructura **11**

4.1. Grupos de contenedores 11

4.2. Cuenta de almacenamiento 11

4.3. Contexto 12

5. Terraform **13**

5.1. Funcionamiento 14

5.2. Implementación 17

6. Ansible **18**

7. Comparación **18**

8. Conclusión **18**

9. Lecciones aprendidas **18**

Índice de figuras

1. Diseño original a alto nivel (O. Salvador, 2022) 7
2. Paso de mensajes durante login (O. Salvador, 2022) 8
3. Inspección de los mensajes durante login usando Wireshark (O. Salvador, 2022) 9
4. Paso de mensajes para añadir post (O. Salvador, 2022) 10
5. Comando de arranque del servidor, `Dockerfile` (O. Salvador, 2022) 11
6. Depuración de CORS (O. Salvador, 2022) 11
7. Sintaxis de referencia, documentación de Hashicorp **hashicorp_lang1** 13
8. Extracto del grafo del frontend (O. Salvador, 2022) 15
9. Planificación de operaciones contra la plataforma (O. Salvador, 2022) 16

1 Experiencia en empresa

tbc

1.1. Introduccion, mis responsabilidades

1.2. Onboarding

1.3. Organigrama

2 Proyecto

El estado del arte para el aprovisionado de infraestructura es contratarla, de manera flexible, a un proveedor cloud. En particular, aún más recientemente se ha apostado por la descripción de la infraestructura a contratar, en lenguajes declarativos y de programación general.
Este nuevo paradigma, Infraestructura como Código, ofrece replicabilidad y reutilización, pero en particular permite afrontar el mantenimiento de esta con técnicas de desarrollo convencionales como el control de versiones.

En este proyecto exploraré dos de las soluciones de Infraestructura como Código más competentes y populares: Terraform y Ansible (en particular Ansible Playbooks). Las usaré para contratar en un proveedor cloud los recursos necesarios para un sistema full stack desarrollado específicamente para este proyecto.

Después de explicar cómo funciona cada una y ponerlas en uso, compararé sus características y contrastaré sus ventajas.

2.1. Antecedentes

Este no es un proyecto original, la aplicación que quiero implementar debería ser una maqueta representativa de los elementos y comportamientos del sistema moderno medio del mercado. El valor añadido será la explicación del de esta, y la comparación entre herramientas comerciales.

2.2. Estudio del problema

He descompuesto el problema en tres fases de trabajo:

1. Programación de componentes lógicos (front-end, middleware) en local, usando Docker para prototipar e iterar rápido
2. Creación manual de los recursos y migración a la nube
3. Descripción de la infraestructura como código con ambas herramientas y comparación de ellas

2.3. Objetivos

1. Implementar un sistema full-stack representativo de la arquitectura que se podría encontrar en una aplicación comercial
    a) Servidor de estáticos, con el Front-end
    b) Servidor de contenido, con imágenes que se usen en el Front-end

   c) Middleware de una sola capa, API con GraphQL
      - Base de datos para tokens de sesión
   d) Back-end: Persistencia usando una base de datos MongoDB

2. Desarrollar el código necesario para contratar la infraestructura necesaria en Terraform y Ansible, y compararlos

3 Aplicación

Propongo una aplicación web para subir fotos con comentarios como caso de uso. Basada en los contenidos de la asignatura de *Programación de interfaces web*. Parte de las bases de las prácticas cuatro (full stack, CRUD en REST) y cinco (solo front-end, GraphQL); y un ejercicio de clase (full stack, GraphQL con tokens de verificación), todas en React. Pero compone un esfuerzo propio, estando formada por piezas nuevas, investigadas para este proyecto, y un desarrollo propio desde el principio,

La página permite ver *posts*, compuestos por: el nombre del usuario que lo ha publicado, un comentario, y una imagen. La lista de *posts* puede verse sin iniciar sesión (*login*). Un usuario por identificar puede iniciar sesión o registrarse. La segunda crea un usuario y después dispara el inicio de sesión automáticamente, transparente al usuario. Una vez identificado puede: Hacer *posts*, lo que implica subir una imagen y un comentario; borrar los comentarios de los que sea autor; y cerrar sesión (*logout*).

Las diferencias principales frente a los antes mencionados anteriores son:

- Redis: el uso de una base de datos llave-valor, que solo guarda los tokens en memoria. Frente a tener esta funcionalidad en la propia base de datos persistente (MongoDB).
- Hospedaje de imágenes: almacenamiento usando una solución de externa, originalmente AWS S3, más tarde Azure Storage Container.
- Reverse Proxy: durante el desarrollo en local, para evitar problemas de CORS. Desde entonces, ha demostrado ser innecesario.

En la primera fase entregué un repositorio que disponía de las partes mencionadas antes, como contenedores Docker desplegados con `docker-compose`. Algunos de los componentes que para ese hito implementé como contenedores se pueden, según el proveedor, contratar como *Software as a Service*, pagando por el uso en lugar de la maquina sobre la que correr el componente.

Desde la primera entrega he cambiado mi objetivo de proveedor, de Amazon Web Services a Azure por razones no técnicas (accesibilidad a una cuenta y fondos). En esa versión me apoyaba sobre MinIO, una solución de almacenamiento local compatible con el API de S3. Aunque conseguí la subida y acceso a imágenes, no tuve tiempo para implementar controles de seguridad. En entrega actual estos problemas están solventados.

Compuesto por un proxy inverso (Traefik), almacenamiento S3, servidores front-end y back-end, Redis y Mongo, el sistema presentaba la siguiente forma. Cabe resaltar que el front-end está mostrado como parte del *bucket* porque planee almacenar su código compilado ahí, y que el propio servidor de "statics" sirviese las imagenes y estos fuentes (también técnicamente estáticos). Este no es el caso en la versión actual.

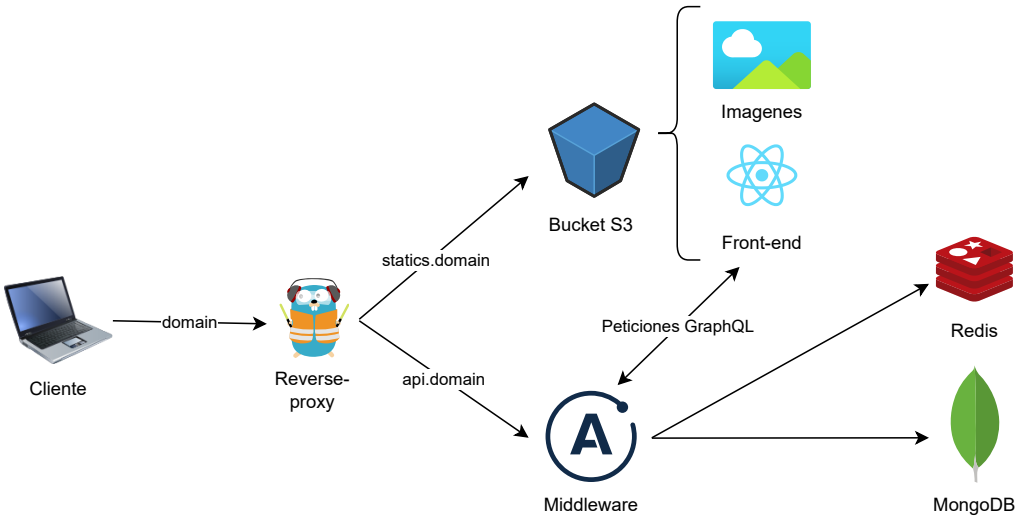


Figura 1: Diseño original a alto nivel (O. Salvador, 2022)

Las necesidades de estas piezas han dictado la infraestructura que he implementado, su diseño habiendo sido iterativo.

3.1. Inicio de sesión

El cliente front-end estará preparado en el S3. El cliente entra al dominio, el proxy dirige la petición inicial hacia los estáticos, y le sirve el JavaScript del front-end.

Desde el cliente se piden los "posts". Estos comentarios están guardados en mongo. Cada post contiene una dirección a su imagen correspondiente. Esta esta guardada en el S3, y solo referenciada en el post. Una vez el cliente tiene la lista de posts, para cada uno, pide su archivo correspondiente. Esta operación no necesita login, es pública. El bucket tendrá permisos de lectura públicos, pero de escritura restringidos.

Con el cliente renderizado, el usuario puede hacer clic en el único botón, login. Desde aquí, puede iniciar sesión si ya se tiene usuario, o crear uno nuevo. Ninguna de estas dos peticiones necesita autorización.

Crear usuario pide un nombre y contraseña. la última se tiene que escribir dos veces y debe tener

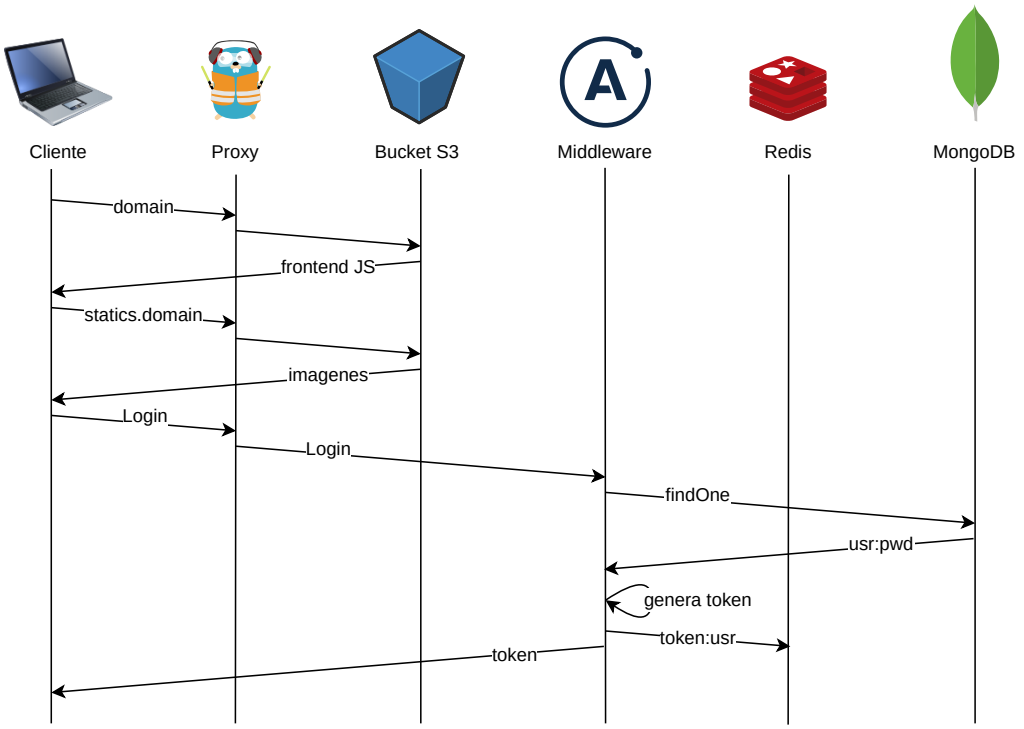


Figura 2: Paso de mensajes durante login (O. Salvador, 2022)

caracteres variados, por seguridad. Si ya hay un usuario con ese nombre, el back-end rechazara la petición. En el caso contrario, la operación se completa y de vuelta en el cliente se dispara un login con las credenciales del usuario recién creado.

El usuario puede hacer login en el cliente, enviando su usuario y contraseña, por ahora en claro (mi objetivo es pasarlo por TLS, lo hare cuando la esté bien y tenga certificados). La petición es reconocida en el proxy y reenviada al middleware. Por la escala de la práctica y su longevidad, he hecho un middleware unificado, en lugar de separarlo en front (más expuesto, y sin estado) y back (con sesión).

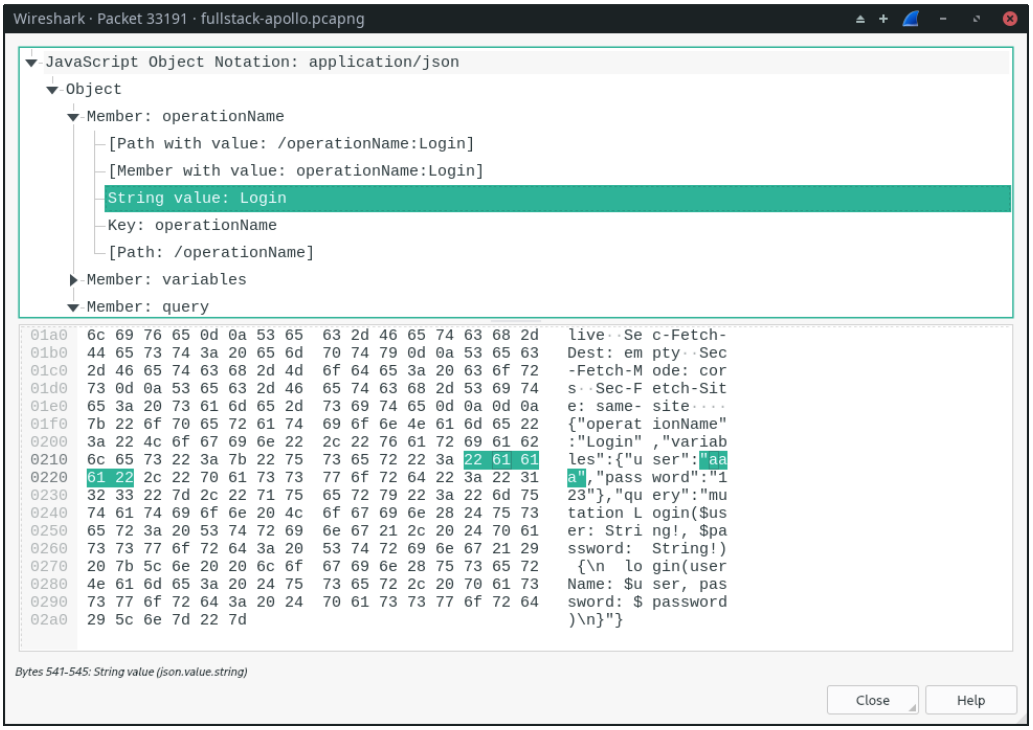


Figura 3: Inspección de los mensajes durante login usando Wireshark (O. Salvador, 2022)

Desde el middleware, se hace una petición a Mongo y comprueban las credenciales en la colección “users”. Si son correctos, se genera un token, y se añade este y el nombre de usuario como entrada en Redis. finalmente, se devuelve al cliente el token del usuario, donde es guardado como una cookie el token tiene una caducidad de una hora.

Inmediatamente después de hacer login, se vuelve a preguntar al middleware los posts, enviando el token. Ahí, este es resuelto al nombre de usuario, y comparado con el de los posts, y se levanta un flag en todos los posts que sean del usuario haciendo la petición.

El resto de las peticiones al middleware requieren del cliente el uso de la cabecera .Authorization”, con un token valido, que se comprueba con cada una. La lista es recibida por el cliente y se vuelve a pintar. Los posts del usuario reciben un botón para borrarlos. Si estos son pulsados, se envía una petición con el _id en la colección de mongo, con el que se identifican en el back y borrados de la colección.

3.2. CRUD de contenido de usuarios

Partiendo de un usuario autenticado, cuando el usuario selecciona el botón “Send” en el Modal post, dispara la petición GraphQL “addPost”, que recibe el comentario escrito. El middleware, soluciona el token del usuario a su nombre. Además, con los credenciales del bucket, que nunca salen del middleware, se genera una URL firmada en la que el cliente puede subir una imagen. Esta es añadida junto a los otros detalles como una nueva entrada a la colección de Mongo. La URL es devuelta al cliente, y la imagen se sube desde este.

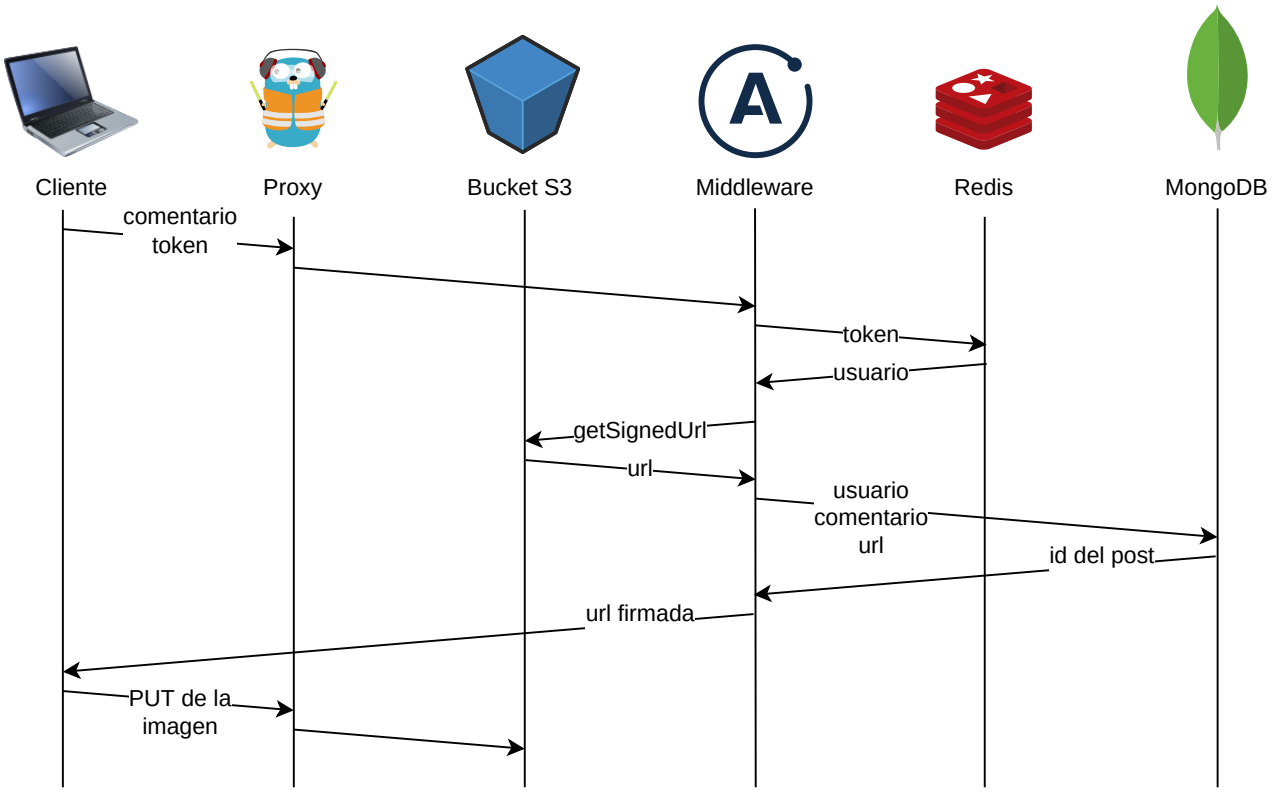


Figura 4: Paso de mensajes para añadir post (O. Salvador, 2022)

Este método permite mantener los credenciales seguros, no dejándolos pasar por el cliente en ningún momento. Al mismo tiempo, permite ahorrar ancho de banda, subiendo la imagen directamente al bucket, en lugar de hacerla pasar primero al middleware.

4 Infraestructura

tbc

4.1. Grupos de contenedores

```
36  # CMD ["serve", "-d", "-s", "build"]
37  CMD ["sh", "-c", "npm run build && serve -d -s build"]
```

Figura 5: Comando de arranque del servidor, `Dockerfile` (O. Salvador, 2022)

4.2. Cuenta de almacenamiento

Allowed origins	Allowed methods	Allowed headers	Exposed headers	Max age
acifesspruebapoc2.westeurope.azurecontainer.io	GET,HEAD,POST,OPTIONS,PUT	x-ms-blob-type,access-control-allow-origin,vary	x-ms-blob-type,access-control-allow-origin,vary	86400

(a) Configuración de CORS de la cuenta de almacenamiento

```
33    environment_variables = {
34      REACT_APP_API_URL = "http://${data.azurerm_container_group.aci_back.ip_address}:4000"
35      REACT_APP_CORS_ORIGIN_TO_ALLOW = "http://${var.frontend_container_group_name}.${var.location}.azurecontainer.io:3000"
36    }
```

(b) Carga de variables en Terraform, `main.tf`

```
80    const myHeaders = new Headers({
81      'content-type':
      ↪ 'application/x-www-form-urlencoded',
82      "Access-Control-Allow-Origin":
      ↪ `${process.env.REACT_APP_CORS_ORIGIN_TO_ALLOW}`,
83      "x-ms-blob-type": "BlockBlob",
84      "Vary" : "Origin"
85    })
86    const req = await fetch(url, {
87      method: "PUT",
88      body: imageList.at(0)!.file,
89      headers: myHeaders
90    }).then((res) => {
91      console.log(res)
```

(c) Cabeceras del frontend, `PostPromt.tsx`

Request Headers (801 B)

Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.5
access-control-allow-origin: acifesspruebapoc2.westeurope.azurecontainer.io
Connection: keep-alive
Content-Length: 1848270
content-type: application/x-www-form-urlencoded
DNT: 1
Host: esspruebapoc2.blob.core.windows.net
Origin: *http://acifesspruebapoc2.westeurope.azurecontainer.io:3000*
Referer: *http://acifesspruebapoc2.westeurope.azurecontainer.io:3000/*
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: cross-site
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:105.0) Gecko/20100101 Firefox/105.0
vary: Origin
x-ms-blob-type: BlockBlob

(d) Cabeceras en la petición del navegador

```
XHR OPTIONS https://esspruebapoc2.blob.core.windows.net/sacesspruebpoc2/1672057823632.png?sv=2021-10-04&se=2022-12-26T13:30:23Z&sr=c&sp=c&sig=nCBBGeE1M8OOVALXyi4F6AgOmu+TyuJSnLhVSmaoLfI=  CORS Missing Allow Origin
Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at https://esspruebapoc2.blob.core.windows.net/sacesspruebpoc2/1672057823632.png?sv=2021-10-04&se=2022-12-26T13%3A30%3A23Z&sr=c&sp=c&sig=nCBBGeE1M8OOVALXyi4F6AgOmu%2BTyuJSnLhVSmaoLfI%3D. (Reason: CORS header 'Access-Control-Allow-Origin' missing). Status code: 403. [Learn More]
Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at https://esspruebapoc2.blob.core.windows.net/sacesspruebpoc2/1672057823632.png?sv=2021-10-04&se=2022-12-26T13%3A30%3A23Z&sr=c&sp=c&sig=nCBBGeE1M8OOVALXyi4F6AgOmu%2BTyuJSnLhVSmaoLfI%3D. (Reason: CORS request did not succeed). Status code: (null). [Learn More]
```

(e) Fracaso de la petición PUT a la cuenta de almacenamiento

Figura 6: Depuración de CORS (O. Salvador, 2022)

4.3. Contexto

otras herramientas, imperativas

5 Terraform

Terraform es una herramienta puramente de aprovisionado de infraestructura. Permite contratarla y cambiarla, manteniendo control de versión. Desarrollado y mantenido por Hashicorp, fue publicado en 2014, bajo la licencia de código abierto Mozilla (MPLv2.0).

Se basa en archivos con configuraciones para saber que desplegar. Pueden estar escritos en el "Hashicorp Configuration Language" (HCL) o en formato JSON. Estos contienen el estado *deseado* de la infraestructura, los elementos que se quieren presentes (e implícitamente, los que no), y que propiedades deberían tener. Para ofrecer esta presentación uniforme de las configuraciones entre distintas plataformas, Terraform utiliza *proveedores* que se encargan de los detalles del proceso de aprovisionado, abstrayéndolos del usuario.

Los elementos, *bloques*, de un proyecto pueden tener uno de varios tipos, algunos de los principales son: declaraciones de variables, de proveedores, recursos, datos de recursos, y salidas. Para declarar variables (y opcionalmente darles un valor predeterminado) que usar en los demás bloques se usa `variable`. Con `required_providers` y `provider` se especifica el proveedor y sus propiedades en el proyecto. El bloque principal es `resource`, donde se especifica un objeto que crear en en la plataforma destino. Los bloques `data` y `output` permiten recuperar información de la plataforma, detalles de objetos que ya estén creados durante el proceso (para usar en otros bloques) y después de la ejecución respectivamente (para usar en otros proyectos). Los bloques HCL deben seguir el siguiente formato:

```
<BLOCK TYPE> '<BLOCK LABEL>' '<BLOCK LABEL>' {
      # Block body
      <IDENTIFIER> = <EXPRESSION> # Argument
}
```

Figura 7: Sintaxis de referencia, documentación de Hashicorp **hashicorp_lang1**

La extensión de la mayoría de archivos HCL en un proyecto de Terraform es `.tf`. En JSON los elementos siguen la misma estructura que en HCL, pero ligeramente adaptada y el prefijo de terraform antes de la extension (eg. `.tf.json`). Los nombres de los archivos es importante, en particular `variables.tf`, `providers.tf`, `main.tf`, y `outputs.tf`. Estos contienen los bloques antes mencionados, aunque, según la escala del proyecto pueden ir todos en el principal. Aparte de estos, hay otros archivos para el funcionamiento de Terraform que mencionaré en la siguiente sección, y `variables.tfvars`. En este no hay bloques, solo parejas de identificadores y expresiones (sus valores). Se puede usar para poblar las declaraciones de variables.

5.1. Funcionamiento

Hashicorp ha hecho admirablemente sencillo la "instalación" de Terraform. Distribuyen un binario listo para usar, y trivialmente portable. En linux, con colocarlo en `/usr/local/bin` queda reconocido por la consola. Ofrece una miríada de opciones en su CLI. Los comandos principales son: `init`, `validate`, `plan`, `apply`, y `destroy`.

El primer comando, `terraform init`: descarga los proveedores que se hayan indicado, crea algunos archivos para su operación y carga un estado remoto si se le indica. En Terraform el estado es una captura en local de los detalles de la implementación de la infraestructura como estaban en la plataforma objetivo la ultima vez que se actualizó.

Este fichero es necesario para que Terraform funcione. "El propósito principal del estado de Terraform es almacenar los enlaces entre los objetos en un sistema remoto y las instancias de recursos declarados en su configuración" (**hashicorp_state**). Se guardan los identificadores y propiedades de los recursos en el archivo `terraform.tfstate`, en formato JSON. Se pueden ver sus contenidos en cualquier momento con el comando `terraform show` Adicionalmente se genera una copia de seguridad, `terraform.tfstate.backup` automáticamente. A menudo contiene información comprometedora, como credenciales, es importante guardarlo de forma segura.

Adicionalmente, el estado se puede guardar remotamente, para facilitar el desarrollo colaborativo y el uso en pipelines de CI/CD. Con el comando `terraform import` se puede crear o actualizar el estado del proyecto en la plataforma *cloud* u *on-prem*. Si el proyecto ya tiene los fuentes con los nombres de los recursos, también es capaz de poblarlos con su configuración en la plataforma (**hashicorp_import**).

En el tercero, `terraform plan`, se convierten los archivos de configuraciones a un conjunto de pasos que Terraform puede seguir para alcanzar dicho estado deseado. Ejecutarlo automáticamente lanzará el segundo, `terraform validate`, que comprueba si los fuentes tienen una configuración valida. Si lo es, resuelve las variables. Recorre `variables.tf` y las puebla con `variables.tfvars`. Si hay mas en el `.tfvars` de las que se usan, Terraform avisará de ello. Si hay menos, y no tienen valor predeterminado son pedidas al usuario por TUI. El usuario puede especificar valores en el comando, que sobreescriben el valor de `variables.tfvars`

Lo siguiente que hace es recuperar el estado actual de cualquier objeto que ya este creado en la plataforma (**hashicorp_plan**, **hashicorp_plan_refresh**). Con la configuración previa

actualizada, calcula las diferencias frente a la propuesta. El último paso para generar un plan es calcular un grafo de dependencias. Terraform hace esto recorriendose los fuentes, viendo los recursos mencionados, y añadiendo un nodo por cada uno. Las parejas de "`<BLOCK LABEL>`" indican dependencia del segundo al primero, los bloques que estén definidos dentro de otros a su padre, y los bloques que usen la etiqueta `depends_on` al que mencionen (**hashicorp_graph**). Después de añadir los nodos al grafo con las dependencias adecuadas, etiqueta a cada uno con meta-datos basándose en las diferencias entre el estado y la configuración, para saber que operaciones tomar con cada nodo.

El grafo se puede generar independientemente con `terraform graph` en cualquier momento, solo para visualizar. El resultado del comando es el texto del grafo, en formato DOT. A continuación muestro un extracto del proyecto mas sencillo, el frontend (detalles en la siguiente sección), es solo una parte para no dedicarle una página[1].

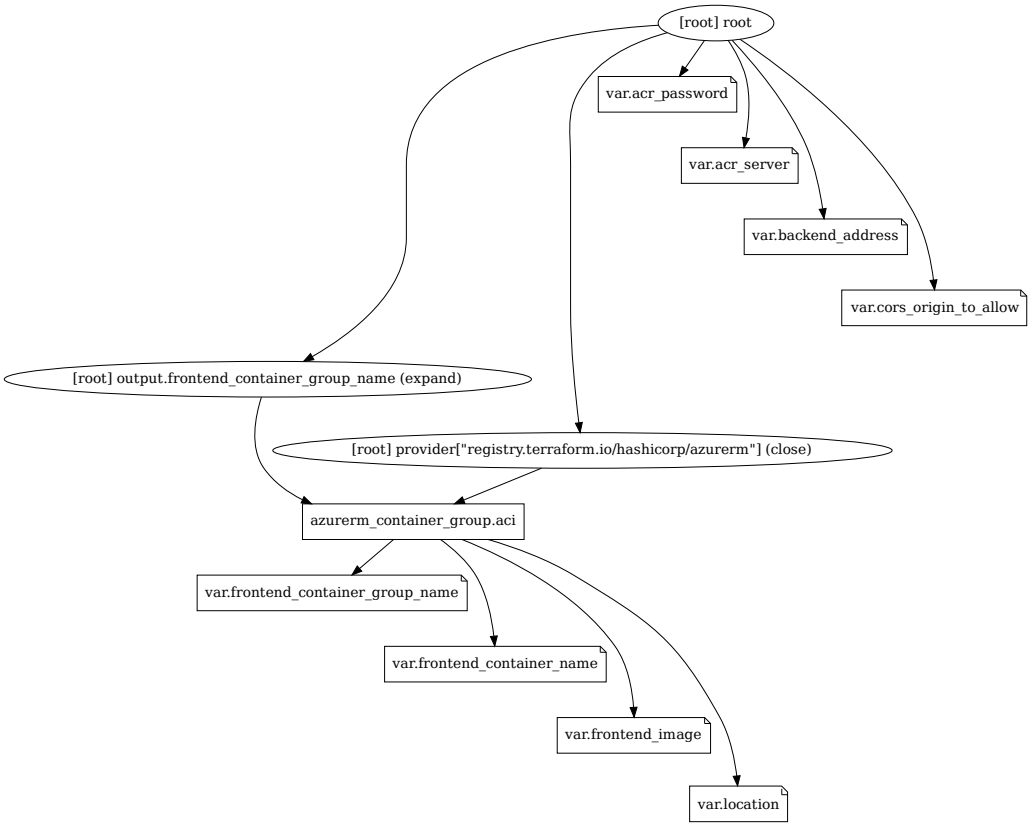


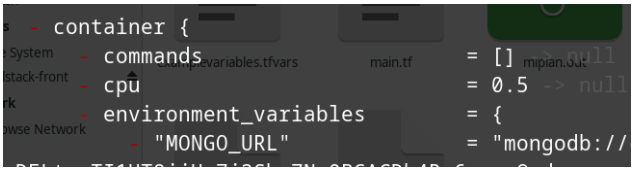
Figura 8: Extracto del grafo del frontend (O. Salvador, 2022)

Finalmente, Terraform calcula y muestra la serie de pasos que Terraform seguirá para alcanzar el estado deseado, si hay diferencias.

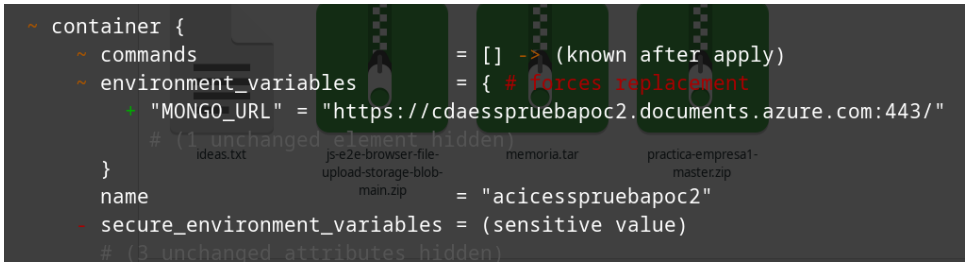
[1]He generado los diagramas completos para todos los proyectos, y un `readme.md` con detalles de como generarlos. Están en la carpeta `documentacion/terraform-graph`

Los cambios a la infraestructura propuestos por el plan pueden ser: crear nuevos recursos, destruir recursos existentes, y actualizar un recurso existente, que a veces requiere la destrucción y sustitución *creación in-place* del recurso.

(a) Creación



(b) Destrucción



(c) Reemplazo

Figura 9: Planificación de operaciones contra la plataforma (O. Salvador, 2022)

La ejecución de la planificación sin más opciones genera un *plan especulativo*, solo en memoria y sin intención de ser aplicado. Este puede ser usado para comprobar si los efectos son los deseados. Para guardarlo es necesario añadir la opción `-out=FICHERO_DESTINO.out` (**hashicorp_plan**).

El penúltimo comando mencionado es `terraform apply`. Implícitamente desata una planificación, con los pasos antes descritos. Alternativamente se le puede alimentar un plan ya calculado incluyendo su nombre (eg. `terraform apply FICHERO_DESTINO.out` para el plan anterior)
Organización en carpetas, modulos

5.2. Implementación

6 Ansible

tbc

7 Comparación

tbc dsaa**rediscode**aa**knuth-fa**

8 Conclusión

tbc

9 Lecciones aprendidas

a