



UNIVERSIDAD
NEBRIJA

DESARROLLO DE UNA APLICACIÓN WEB Y
ESTUDIO PRÁCTICO 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 PRÁCTICO 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

Acrónimos	5
Glosario	6
1. Proyecto	9
1.1. Antecedentes	9
1.2. Objetivos	10
1.3. Estudio del problema	10
2. Aplicación	11
2.1. Acceso a la página	13
2.2. Registro e inicio de sesión	14
2.3. Creación de “ <i>Posts</i> ”	16
2.4. Eliminación de “ <i>Posts</i> ”	18
2.5. Cierre de sesión	18
3. Infraestructura e integración	19
3.1. Bases de datos	20
3.2. Grupos de contenedores	22
3.3. Cuenta de almacenamiento	26
4. Terraform	28
4.1. Funcionamiento	29
4.2. Implementación	33
5. Ansible	37
5.1. Funcionamiento	38
5.2. Implementación	41
5.3. Funcionamiento paralelizado	44
5.4. Implementación paralelizada	45
6. Comparación	49
6.1. Rendimiento	51
6.2. Relevancia	53
6.3. Integración de una en la otra	55
7. Conclusión	56

8. Lecciones aprendidas y líneas de continuación	57
A. Generación gráfico del grafo de dependencias	58
B. Tiempos de aprovisionado de las herramientas	58
C. Verbosidad de los proyectos y playbooks	59
Bibliografía	60

Índice de figuras

1.	Diseño original a alto nivel (O. Salvador, 2022)	12
2.	Diseño revisado (O. Salvador, 2022)	12
3.	Paso de mensajes <code>getPosts()</code> (O. Salvador, 2022)	13
4.	Proceso de registro, (O. Salvador, 2022)	14
5.	Peticiones <code>register()</code> y consecuente <code>login()</code> (O. Salvador, 2022)	15
6.	Cabeceras en la petición del navegador (O. Salvador, 2022)	15
7.	Extractos de código del backend para trabajar con Redis (O. Salvador, 2022)	16
8.	Uso de credenciales de almacenamiento, backend, <code>mutation.js</code> (O. Salvador, 2022)	17
9.	Extracto de eliminación de imágenes, backend, <code>mutation.js</code> (O. Salvador, 2022)	18
10.	Diagrama de infraestructura (O. Salvador, 2022)	19
11.	Ejemplo de uso de MongoDB en el backend (O. Salvador, 2022)	20
12.	Azure Data Explorer, colección “users” (O. Salvador, 2022)	21
13.	Consola de Redis desde el portal de Azure (O. Salvador, 2022)	21
14.	Listado de llaves con <code>redis-cli</code> en contenedor (O. Salvador, 2022)	22
15.	Tiempo necesario para aprovisionar Redis usando Terraform (O. Salvador, 2022)	22
16.	Comandos para subir la imagen del frontend (O. Salvador, 2022)	23
17.	Comandos de arranque de los servidores contenedorizados (O. Salvador, 2022)	23
18.	Azure Container Registry compartiendo capas entre imágenes (O. Salvador, 2022)	24
19.	Diagrama a alto nivel, conexiones y orden de aprovisionado (O. Salvador, 2022)	25
20.	Peticiones GraphQL con cURL (O. Salvador, 2022)	25
21.	Investigación de los detalles de la infraestructura (O. Salvador, 2022)	26
22.	Comando para subir imágenes (O. Salvador, 2022)	26
23.	Depuración de CORS (O. Salvador, 2022)	27
24.	Sintaxis de referencia, documentación de Terraform (Hashicorp, s.f.-g, 2022)	28
25.	Extracto del grafo del frontend (O. Salvador, 2022)	30
26.	Planificación de operaciones contra la plataforma (O. Salvador, 2022)	31
27.	Declaración y configuración del proveedor, <code>providers.tf</code> (O. Salvador, 2022)	32
28.	Tres proyectos de Terraform y dos imágenes de Docker (O. Salvador, 2022)	33
29.	Uso de bloques <code>data</code> para las variables de entorno de los contenedores (O. Salvador, 2022)	34
30.	Extracto del grafo del frontend (O. Salvador, 2022)	35
31.	Configuración de CORS del almacenamiento, <code>main.tf</code> (O. Salvador, 2022)	36
32.	Configuración de red limitando por IP, <code>main.tf</code> (O. Salvador, 2022)	36
33.	Configuración de sonda de estado, backend, <code>main.tf</code> (O. Salvador, 2022)	36
34.	Árbol de archivos de un playbook (O. Salvador, 2022)	37

35.	Formato de una Play (O. Salvador, 2023)	38
36.	Formato de una tarea (O. Salvador, 2023)	38
37.	Ejecución de una tarea de Ansible y su resultado (O. Salvador, 2023)	40
38.	Resumen de ejecución de un playbook, resultados de las tareas (O. Salvador, 2023) . .	40
39.	Play, con los tres primeros roles, <code>ansible/site.yml</code> (O. Salvador, 2023)	41
40.	Rol <code>infra-base</code> , variables, carga y uso (O. Salvador, 2023)	42
41.	Inclusión y uso de variables condicionales <code>/vars/main.yml</code> (O. Salvador, 2023)	42
42.	Peticiones de información en los roles <code>docker</code> y <code>backend</code> (O. Salvador, 2023)	43
43.	Formato de una tarea asíncrona (O. Salvador, 2023)	44
44.	Valor registrado de tarea asíncrona (O. Salvador, 2023)	44
45.	Tarea de espera en <code>parallel.ansible/infra-base/ main.yml</code> (O. Salvador, 2023) .	45
46.	Resultado de la espera de varias tareas asíncronas (O. Salvador, 2023)	46
47.	Espera de resultados de tres tareas asíncronas (O. Salvador, 2023)	46
48.	Procesos de tareas asíncronas de fondo (O. Salvador, 2023)	46
49.	Playbook paralelizado, etapas, dependencias y peticiones (O. Salvador, 2023)	47
50.	<code>/vars/main.yml</code> (O. Salvador, 2023)	48
51.	Tarea de eliminado saltada (O. Salvador, 2023)	48
52.	Tiempos de los proyectos en segundos (O. Salvador, 2023)	52
53.	Verbosidad de los proyectos (O. Salvador, 2023)	52
54.	Tecnologías más populares entre profesionales (Stack Overflow, 2022)	53
55.	Tendencias de búsqueda (Google Trends, 2023)	54
56.	Frecuencia de contribuciones a los proyectos (GitHub, 2023)	54
57.	Tarea de despliegue en <code>roles/terraform/task/main.yml</code> (O. Salvador, 2023)	55
58.	Comando para generar diagramas de los grafos de Terraform (O. Salvador, 2022) . . .	58

Índice de tablas

1.	Tiempos medios de aprovisionado	58
2.	Verbosidad de los proyectos de Terraform	59
3.	Verbosidad de Terraform y Ansible	59

Acrónimos

ACA Azure Container Apps. 22	HTTPS HyperText Transfer Protocol Secure. 15
ACI Azure Container Instance. 22	IaC Infrastructure as Code. 9
ACR Azure Container Registry. 24	IP Internet Protocol. 12
AGPL Affero General Public Licence. 20	JSON JavaScript Object Notation. 28
AKS Azure Kubernetes Service. 22	MPL Mozilla Public Licence. 28
API Application Programming Interface. 10	NPM Node Package Manager. 16
ARM Azure Resource Manager. 50	PR Pull Request. 9
AWS Amazon Web Services. 11	REST REpresentational State Transfer. 11
BBDD Bases de Datos. 20	S3 Simple Storage Solution. 11
CDK Cloud Development Kit. 50	SaaS Software as a Service. 11
CI/CD Continuous Integration and Continuous Deployment. 29	SSH Secure SHell. 37
CLI Command Line Interface. 29	SSPL Server-Side Public Licence. 20
CORS Cross-Origin Resource Sharing. 11	TLS Transport Layer Security. 15
CRUD Create Read Update Destroy. 11	TUI Terminal User Interface. 29
FQDN Fully Qualified Domain Name. 12	URL Uniform Resource Locator. 17
GPL General Public Licence. 37	YAML YAML Ain't Markup Language. 37
HCL Hashicorp Configuration Language. 28	

Azure	La plataforma cloud de Microsoft	11
Backend	Servidor final, con la lógica de negocio. En esta practica, intercambiable con middleware	10
Blob	Unidad de almacenamiento en los contenedores de almacenamiento de Azure	12
Bucket	Unidad de almacenamiento en AWS	12
Cliente	Proceso que se conecta a un servidor, en el caso de una página web, el código de la página que corre en navegador del usuario	10
Cloud	Un servicio o plataforma contratado como servicio, el ordenador de un tercero	9
Docker	Software de virtualización por contenedores	10
Frontend	Servidor de archivos con los que un navegador puede renderizar una página web	10
Fullstack	Conjunto de todos los sistemas, frontend, backend y bases de datos	9
GraphQL	Graph Query Language, alternativa a REST, propuesta por Facebook	10
Infraestructura	Conjunto de componentes hardware y software sobre los que se apoya una aplicación	9
Localhost	El nombre de host estándar para la misma máquina	39
Middleware	Servidor entre frontend y bases de datos	10
MongoDB	Base de datos no-SQL documental	10
On-Prem	“On-Premises”, Infraestructura en las premisas de la empresa, que no es contratada a terceros	29
Overhead	Coste añadido incurrido por procesos de gestión o supervisión	55

Pipeline	Conjunto de procesos automatizados, normalmente en un repositorio o entorno cloud, que se aplican después de ser disparados por eventos como un commit a dicho repo	9
Redis	Base de datos llave-valor que solo guarda en memoria. Rápida pero no persistente, para tokens	11
Reverse-Proxy	Proxy dentro del firewall, en particular usado por recursos dentro de la organización para acceder a otros también dentro. En mi caso, el valor es presentar una única dirección, saltándome el problema de CORS	11
Token	También conocido como identificador de sesión, es una cadena de caracteres con la que seguir a un usuario, y darle acceso sin que tenga que enviar su contraseña repetidas veces	10
Traefik	Proxy inverso de última generación, descubre servicios por su cuenta	12
URL pre-firmada	URL con credenciales de uso limitado incluidos, para distribución	17
Wildcard	Valor que representa una equivalencia a cualquier valor	35
Wrapper	Capa de abstracción sobre una herramienta, con la traducción literal “envoltorio”	50

D. Óscar Salvador Sotoca autoriza a que el presente trabajo se guarde y custodie en los repositorios de la Universidad Nebrija y además NO autoriza a su disposición en abierto.

1 Proyecto

El estado del arte para el aprovisionado de *Infraestructura* es contratarla, de manera flexible, a un proveedor *Cloud*. En particular, 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 (*IaC*), ofrece replicabilidad, reutilización, y una forma de afrontar el mantenimiento de esta con técnicas de desarrollo convencionales como el control de versión. De esta manera, se puede tener guardada la configuración de la infraestructura deseada, y volverla a desplegar en caso de pérdida (*Disaster Recovery*). Hace posible mantener un mayor control sobre los cambios a la infraestructura, haciéndolos pasar por un proceso de validación, como *PRs* o pudiendo hacerlos pasar por un *Pipeline* en el que se apliquen pruebas automatizadas. Además, los cambios quedan reflejados, junto a su autor en dicho repositorio.

Desplegar la infraestructura como código garantiza replicabilidad ya que una vez depurada no es falible como lo serían interacciones con el interfaz gráfico del proveedor. Ofrece reutilización ya que los componentes descritos en un proyecto pueden ser copiados a otro fácilmente, y en ningún momento es necesario repetir el proceso de aprovisionamiento a mano.

Este es un proyecto de desarrollo, en el que exploraré dos de las soluciones de Infraestructura como Código más populares: Terraform y Ansible (en particular Ansible Playbooks). Las usaré para contratar, en un proveedor cloud, los recursos necesarios para un sistema *Fullstack* desarrollado específicamente para este proyecto.

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

1.1. Antecedentes

Este no es un proyecto original, la aplicación que quiero implementar debería ser una maqueta representativa de los elementos y comportamientos del sistema medio actualmente en uso, intentando replicar fehacientemente las funcionalidades que se esperan de este, aún si más sencillo.

El valor añadido de este proyecto será la explicación del desarrollo de la aplicación, del desarrollo del código para su aprovisionamiento y la comparación entre herramientas comerciales para el segundo.

1.2. Objetivos

Considero la funcionalidad básica necesaria, que separa a las aplicaciones web 2.0, es el contenido de usuario, que los individuos que usen la página puedan subir comentarios y en particular imágenes. Esto requiere el primer punto. Hacerla públicamente accesible es facilitado por el segundo. El objetivo lectivo de la práctica requiere el tercero.

1. Implementar un sistema fullstack representativo de la arquitectura que se podría encontrar en una aplicación comercial
 - a) Servidor de estáticos de la página, *Frontend*.
 - b) Servidor de contenido, con imágenes que se usen, y se puedan subir y borrar desde el *Cliente*.
 - c) *Middleware* de una sola capa, *Backend*, con *API GraphQL*.
 - Base de datos para *Tokens* de sesión.
 - d) Persistencia del contenido de texto usando una base de datos *MongoDB*.
2. Implementar la infraestructura que requieran los componentes, e integrarlos en ella
3. Desarrollar el código necesario para contratar la infraestructura necesaria en Terraform y Ansible, y compararlos

1.3. Estudio del problema

He descompuesto el problema en cuatro fases de trabajo:

1. Programación de componentes lógicos frontend y backend en local, usando *Docker* para prototipar e iterar rápido.
2. Creación manual de los recursos y migración a la nube, depurando la integración para un entorno de producción.
3. Descripción del diseño de la infraestructura como código con ambas herramientas.
4. Elaboración de la memoria, explicando las tecnologías usadas y comparando las herramientas IaC.

Completé la primera fase, y la parte de la cuarta correspondiente, a tiempo de la entrega parcial. Esto dejó las dos siguientes y el grueso de la cuarta para la entrega final.

2 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 los cimientos de sus prácticas cuatro (fullstack, *CRUD* en *REST*, Salvador, 2022a) y cinco (solo frontend, GraphQL, Salvador, 2022b); y un ejercicio de clase no publicado (fullstack, GraphQL con tokens de verificación). Todos 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, de manera 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 su sesión (*logout*).

Las diferencias principales frente a los desarrollos antes mencionados 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.
- Proveedor cloud: este será el primer proyecto en el que no desarrolle solo en local

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 máquina 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 la entrega actual estos problemas están solventados.

Compuesto por un proxy inverso (*Traefik*), almacenamiento S3, servidores frontend y backend, Redis y Mongo, el sistema presentaba la siguiente forma. Cabe resaltar que el frontend está mostrado como parte del *Bucket* porque planeo almacenar su código compilado ahí, y que el propio servidor “statics” sirviese las imágenes y estos fuentes (también técnicamente estáticos). Este no es el caso en la versión actual.

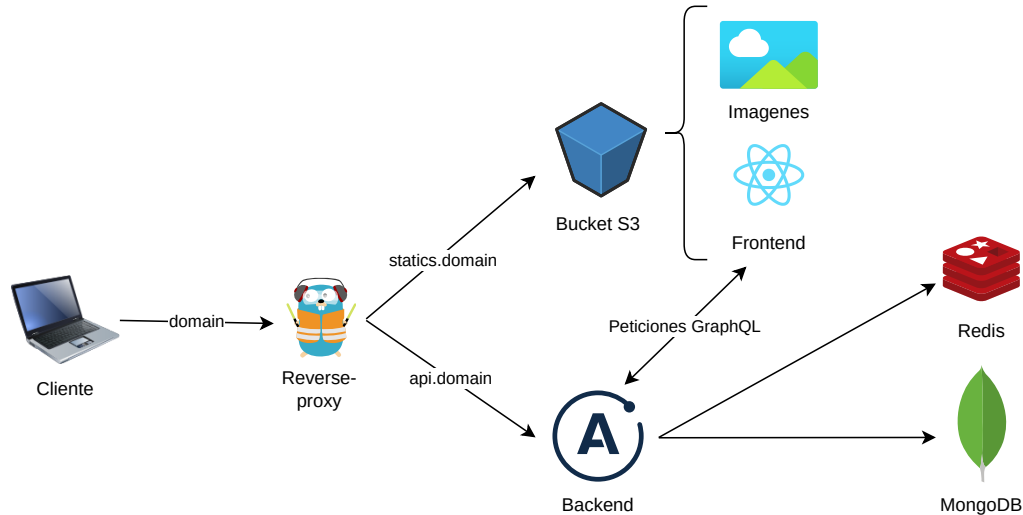


Figura 1: Diseño original a alto nivel (O. Salvador, 2022)

Las necesidades de las piezas principales de la aplicación han dictado la infraestructura que he implementado, iterando sobre él. Con la correcta configuración de CORS en el almacenamiento de *Blobs*, es posible mantener la seguridad sin necesitar un proxy inverso. Cabe resaltar que en mi modelo actual todos los componentes están expuestos a internet, teniendo todos una dirección *IP*, y el frontend un *FQDN*.

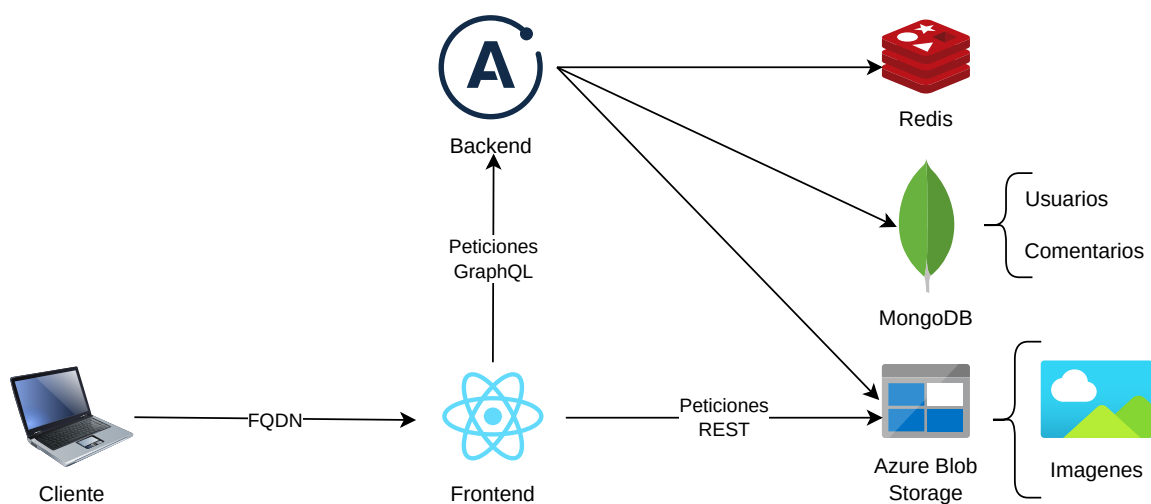


Figura 2: Diseño revisado (O. Salvador, 2022)

2.1. Acceso a la página

He escrito el frontend en TypeScript, lo que hace necesario compilar los fuentes (`.tsx`) a JavaScript. Entrare en detalle sobre cómo son servidos estos y el resto de estáticos del frontal en la sección dedicada a infraestructura.

El usuario accede al servidor del frontal y baja los archivos de la página a su navegador. Desde ahí, el cliente hace peticiones GraphQL al backend, y REST al *Azure Blob Storage*. La primera petición que hace el cliente es al backend, recuperando la lista de *posts*. Para hacerlo, el backend usa un *connection string* con la que enlaza con la base de datos Mongo. En esta petición no es necesario que el usuario este autenticado, ya que no tendría sentido pedirle los credenciales tan pronto.

La lista de *posts* contiene las direcciones de la imagen y autor de cada uno. He configurado el almacén para permitir lectura pública. El cliente renderiza cada comentario, recuperando una a una las imágenes, y si el usuario estuviese autenticado, mostrándole la opción de borrar aquellos de los que sea autor.

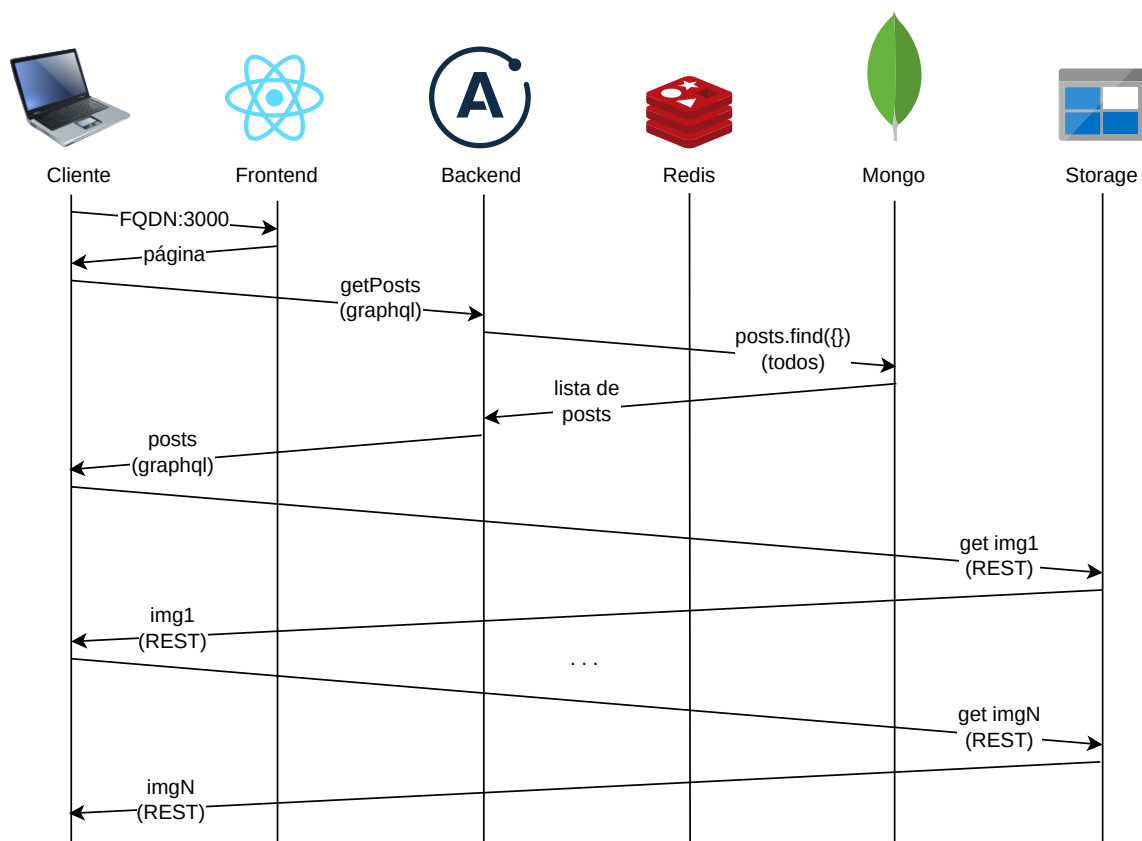


Figura 3: Paso de mensajes `getPosts()` (O. Salvador, 2022)

2.2. Registro e inicio de sesión

Con la página renderizada, el usuario puede ver una opción en la esquina superior derecha, “Login”. Hacer clic sobre el botón despliega un *modal*, un desplegable de React.

Dentro de este se le piden nombre de usuario y contraseña. La primera vez que entre, no tendrá cuenta y deberá registrarse. En el modal hay dos solapas, “Login” y “Register”. Elegir la segunda cambiara el modal por uno dedicado al registro. Este comparte los campos del anterior, y añade una segunda entrada para confirmar la contraseña. En el cliente se comprueba la contraseña: los valores de los campos de contraseñas tienen que ser iguales, una contraseña debe tener al menos cuatro caracteres, una minúscula, mayúscula, número, y carácter especial. Si valen, se envía una petición al backend para guardar el usuario. Este comprueba en Mongo si ya existe uno con ese nombre y, de hacerlo, niega la creación. En caso contrario, se crea.

```
74 let validPattern = /^(?=.*[A-Z])(?=.*[!@#$%*"])(?=.*[0-9])(?=.*[a-z]).{4,}[ ]*$/g.test(confirmPw);
```

(a) Verificación de caracteres de la contraseña, frontend, `LoginPrompt.tsx` (O. Salvador, 2022)

```
153 register: async (parent, args, ctx) =>{  
154   try{  
155     const db = ctx.db;  
156     const {userName, password} = args;  
157     const exists = await db.collection("users").findOne({userName});  
158     if (exists){  
159       throw new ApolloError("User already exists", "USER_EXISTS");  
160     }  
}
```

(b) Comprobación del usuario contra la base de datos, backend, `mutation.js` (O. Salvador, 2022)

Figura 4: Proceso de registro, (O. Salvador, 2022)

La operación de registro desata un inicio de sesión, automáticamente, al acabar. En esta, al buscar en la base de datos, con usuario y contraseña, si no hay una entrada en la que ambas encajen con la petición, se responde con error. El error es genérico, “User or password incorrect”, deliberadamente no confirmando si existe el usuario, por seguridad.

De ser un éxito, el backend a continuación genera una cadena de caracteres aleatoria y guarda este token, junto con su usuario, en la base de datos Redis. He diseñado la función de manera que los tokens caduquen automáticamente después de una hora, y se eliminan del Redis. Como he planteado

el sistema, un mismo usuario puede tener varias sesiones abiertas a la vez. Al responder al cliente le pasa este token, que después se guarda como cookie en el navegador.

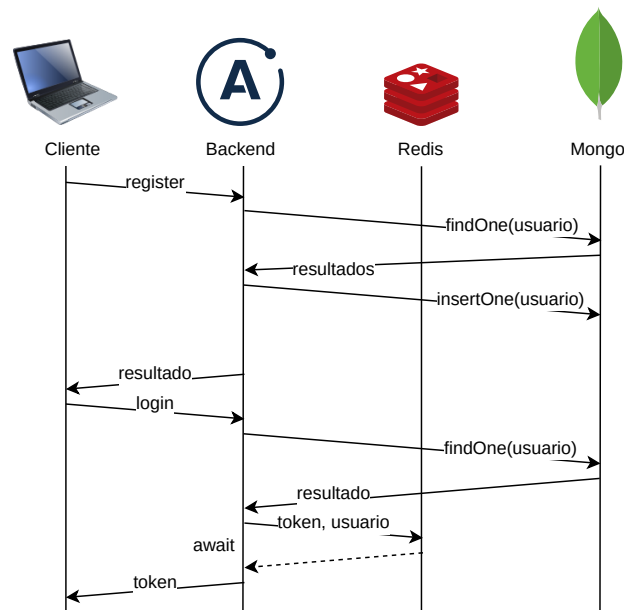


Figura 5: Peticiones `register()` y consecuente `login()` (O. Salvador, 2022)

En la entrega parcial tenía como objetivo tunelizar el tráfico del cliente al backend. Al pasar el tráfico por *TLS*, no sería necesario cifrarlo en la propia aplicación. No he sido capaz de implementar los certificados necesarios para que la conexión sea *HTTPS*. Como resultado, los credenciales son intercetales

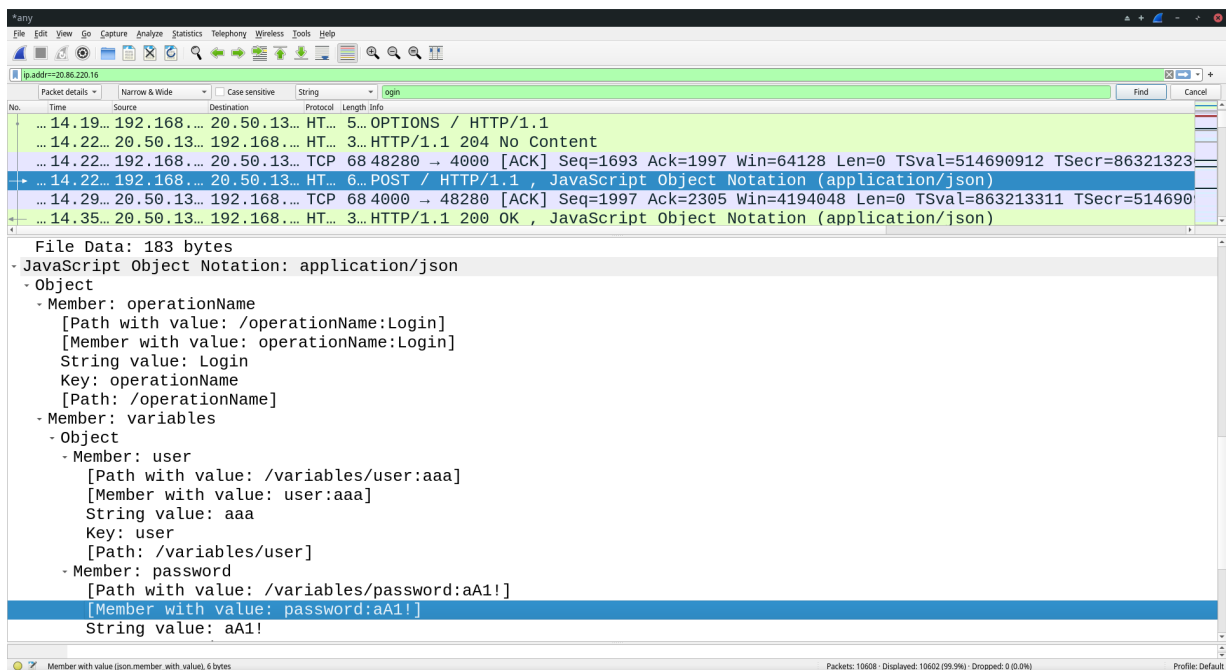


Figura 6: Cabeceras en la petición del navegador (O. Salvador, 2022)

Todas las demás peticiones requieren que el usuario esté autenticado. En todas se espera que la petición venga con un token, y en cada una de ellas se comprueba este contra la base de datos Redis. Creo un único cliente de Redis para el backend, y lo uso para resolver el token a un usuario. Paso el usuario y cliente a cada función por el contexto de ejecución de ApolloServer. Las funciones que necesitan verificación comprueban el usuario. Algunas que necesitan conectarse a Redis ahorran crear el suyo propio. Durante la integración de Redis en el backend usé el siguiente artículo: Leal, s.f.

```
const redisClient = redis.createClient({
  socket:{
    host: process.env.REDIS_HOST,
    port: process.env.REDIS_PORT,
    tls: true
  },
  password: process.env.REDIS_PASS,
});
```

(a) Declaración, `index.js`

```
redisClient.connect();
```

```
const token = req.headers.authorization || "";
```

```
const user = await redisClient.get(token);
```

```
return{db, user, token, redisClient};
```

(b) Conexión y paso por contexto, `index.js`

```
57   removePost: async (parent, args, ctx) => {
58     try {
59       const {db, user, token, redisClient} = ctx;
60       console.log(`\n\nremovepost user: ${user}`)
61       if(!user) throw new ApolloError("Unauthorized", "401")
```

(c) Uso del nombre de usuario resuelto para autorizar `mutation.js`

Figura 7: Extractos de código del backend para trabajar con Redis (O. Salvador, 2022)

2.3. Creación de “*Posts*”

Una vez el usuario se ha identificado, el botón “Login” queda sustituido por dos: “Post” y “Logout”. Al pulsar el primero, aparecerá un nuevo modal. En él hay dos botones a su vez, uno al que se puede arrastrar una imagen para adjuntarla al comentario, y otro para quitarla, para que el usuario pueda cambiar de elección. Al elegir una imagen, se muestra en el modal una vista previa de ella. Para conseguir esta funcionalidad he usado un paquete de *NPM* separado, *react-images-uploading* (Nguyen y Tran, 2022). Mi implementación se basa en su código de referencia, pero considerablemente editado, ya que mi uso es más limitado que el que demuestran en él.

En el modal hay un campo de texto, para escribir un comentario que acompañe a la imagen. Al escribir algo sobre este, aparecerá un botón a su derecha para subir el *post*. Cuando el backend recibe la petición `addPost()` del cliente, no recibe ninguna imagen, solo el autor y el comentario. Después de autenticarlo, genera una *URL pre-firmada*. Sube esta y el comentario del *post*, junto con el autor a Mongo. Por último, devuelve la *URL* al cliente, que se conecta al almacén y sube la imagen directamente.

El almacén tiene restricciones de acceso, por seguridad. Permite leer sus contenidos libremente, pero solo alguien autorizado puede subir nuevos. Pasar los credenciales de administración del almacén al cliente presenta un riesgo de seguridad. Hacerlo en el backend, donde podrían estar con menos riesgo, presenta dos problemas: además del desafío que es subir la imagen a Azure, subirla al backend primero; y el tráfico añadido, ya que ahora el backend tiene primero que bajarse la imagen y luego subirla. Mitigo completamente el problema de seguridad no dándole al cliente más que lo mínimo para que suba la imagen. Este mínimo es la URL pre-firmada.

Aunque el término viene de la implementación de AWS, en Azure se puede conseguir un resultado similar. Primero importo del entorno de ejecución del proceso las variables con los credenciales de administración del almacén, aquí no son preocupantes. Con ellas genero una llave de acceso compartido *StorageSharedKeyCredential* y con esta una firma de acceso compartido. Uso esta para generar una URL en la que se puede crear un Blob. Tiene una caducidad de una hora y solo permite crear uno. Esta llave desechable va embebida en la URL, que devuelvo como respuesta a la petición. Esta ha sido una de las partes más complicadas, y he usado extensivamente la documentación oficial, en particular: Microsoft: Jewell, Myers, Berry et al., 2022, Microsoft: Jewell, Berry, Myers, McClister et al., 2022, Microsoft: Jewell, Myers, Estabrook et al., 2022, Microsoft: Zhu et al., 2022, y Microsoft: Myers et al., 2022.

```
const key = new StorageSharedKeyCredential(  
  accountName= AZURE_ACCOUNT_NAME,  
  accountKey= AZURE_PRIMARY_KEY  
)
```

(a) Credenciales, cuenta de almacenamiento

```
const containerSAS = generateBlobSASQueryParameters({  
  containerName: AZURE_CONTAINER,  
  permissions: ContainerSASPermissions.parse("c"),  
  expiresOn: expDate  
},sharedKeyCredential=key).toString();
```

(b) Creación de la *Shared Access Signature*

Figura 8: Uso de credenciales de almacenamiento, backend, `mutation.js` (O. Salvador, 2022)

De vuelta en el cliente, subir la imagen es una simple petición REST, usando `fetch()` para hacer un PUT de la imagen que el usuario ha subido a este.

2.4. Eliminación de “*Posts*”

Cuando el usuario (autenticado) tenga uno o más *posts* a su nombre, al renderizar la página, el cliente los marcará con la opción de eliminarlos. Si el usuario hace clic sobre esta opción, el cliente alimentará el identificador del *post* a la función `removePost()`

En el backend, se comprueba si el token del cliente es válido, y si el identificador de *post* representa uno existente. Superadas estas dos condiciones, se comprueba que el nombre del usuario es el mismo que el del autor. Es importante recalcar que esta comprobación no depende solo de la implementación del cliente, se hace dos veces. Antes de borrar la entrada en la base de datos, borro la imagen del almacén. Aunque me costó más aprender a subir las imágenes, borrarlas también ha sido difícil. El artículo que más me ha ayudado ha sido Microsoft: Jewell, Berry, Myers y Estabrook, 2022, igual que los otros, de la documentación de oficial.

Igual que en la creación, recupero los credenciales para acceder al almacén de las variables de entorno y la junto en una *StorageSharedKeyCredential*. Con esta creo un cliente para el contenedor, y con este, un cliente para el blob de la imagen elegida. Si la imagen existe, la borro. Después del borrado, elimino su entrada en la base de datos Mongo.

```
104      const key = new StorageSharedKeyCredential(accountName= AZURE_ACCOUNT_NAME,  
      ↪      accountKey= AZURE_PRIMARY_KEY)  
  
107      const containerClient = new ContainerClient(  
108          `https://${AZURE_ACCOUNT_NAME}.blob.core.windows.net/${AZURE_CONTAINER}/`,  
109          key  
110      )  
  
118      const blockBlobClient = await containerClient.getBlockBlobClient(blobName)  
  
126      const result = await blockBlobClient.deleteIfExists(options)
```

Figura 9: Extracto de eliminación de imágenes, backend, `mutation.js` (O. Salvador, 2022)

2.5. Cierre de sesión

Esta operación es muy corta, el cliente manda una petición al backend para que este borre la entrada de la base de datos Redis. Al recibir la respuesta confirmando la eliminación, borra la cookie del navegador del usuario.

3 Infraestructura e integración

Uso una mezcla heterogénea de servicios de Azure para satisfacer las necesidades de los componentes mencionados a alto nivel en la Figura 2 (p. 12), dejando atrás la simplicidad de la primera fase de entrega, donde todos los componentes eran contenedores manejados con Docker Compose. Los componentes necesarios para que el sistema funcione son:

1. **Grupo de recursos** con el que contener a todos los demás, y mantenerlos organizados.
2. **Red virtual** para poder acceder a los recursos que quiero públicos. Algunos, como los grupos de contenedores la necesitan para ser contratados.
3. **Cuenta de almacenamiento y contenedor de almacenamiento** donde hospedar las imágenes con contenedores de “blobs”.
4. **Registro de contenedores** al que subir las imágenes del frontend y backend.
5. **CosmosDB**, la solución de base de datos de Azure, con compatibilidad para MongoDB.
6. **Redis** como cache para los tokens.
7. **Instancias de contenedores**, una de las soluciones de Azure para correr contenedores.

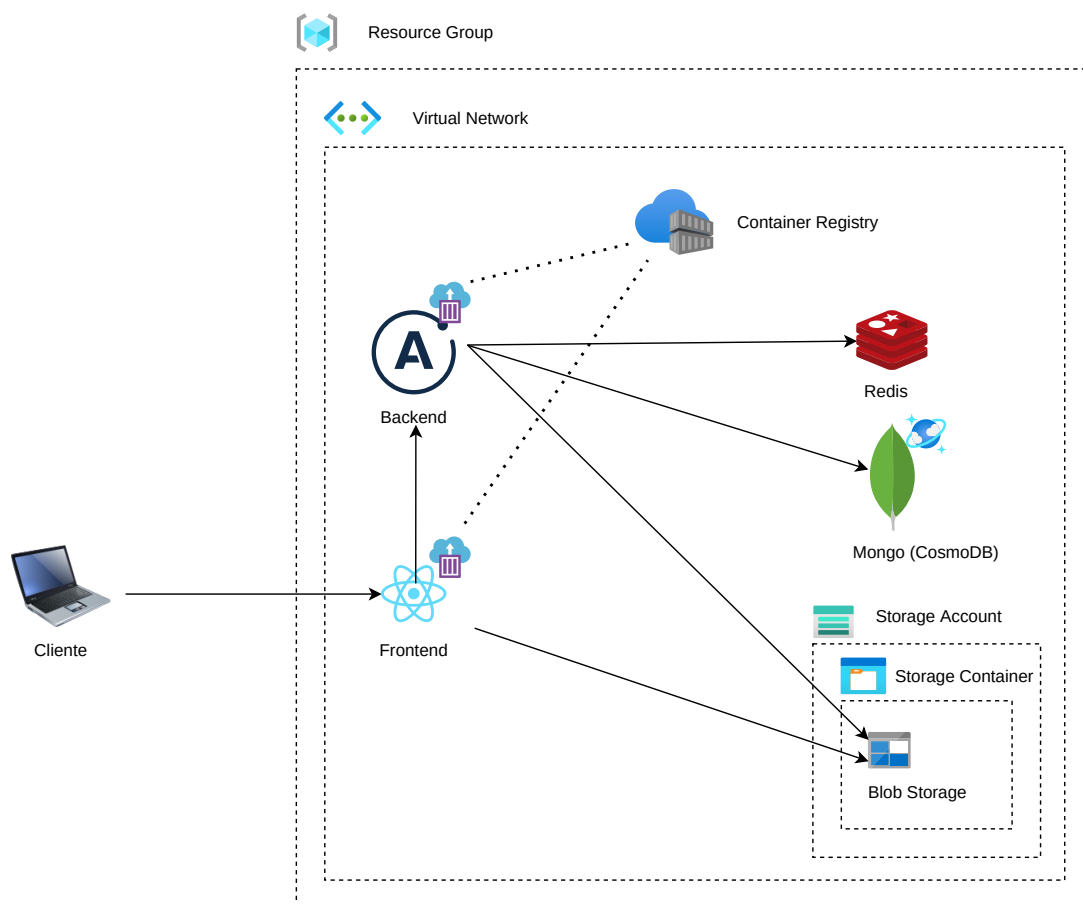


Figura 10: Diagrama de infraestructura (O. Salvador, 2022)

3.1. Bases de datos

El sistema necesita dos *BBDD*: una documental para almacenar los *posts* y usuarios de forma persistente; y una llave-valor para almacenar impermanentemente las parejas token-usuario.

Originalmente diseñe el sistema en local, ambas suplidas por contenedores, con las imágenes oficiales `mongo:latest` y `redis/redis-stack:latest`. Las dos permiten su uso en local, para desarrollo, sin autenticación. Este no es el caso en producción, en Azure ambas necesitan credenciales para poder acceder, detallaré la implementación de estos en la siguiente sección.

Desde el 16 de Octubre, 2018, MongoDB Inc. ha publicado las nuevas versiones de su software bajo la licencia *SSPL* (*Server Side Public Licence*) (MongoDB, 2018). Esta es una licencia de código libre, que permite la copia y distribución, basada en *AGPLv3*, pero con el requisito añadido de que cualquier proveedor cloud que ofrezca la funcionalidad del software con esta licencia debe publicar la totalidad de su código fuente. Esto incluye software, APIs y cualquier otro componente necesario para replicar la solución del proveedor.

Como resultado de este cambio, plataformas cloud como la ofrecida por Microsoft pasaron a ofrecer alternativas compatibles con MongoDB, pero resultado de un desarrollo independiente. En el caso de Azure, su solución de base de datos es CosmosDB, y tiene un modo de uso compatible con aplicaciones diseñadas para Mongo.

En mi experiencia durante este proyecto, los desarrolladores de esta alternativa han conseguido un éxito completo. La integración de mi aplicación con esta solución fue admirablemente trivial. Solo tuve que cambiar la URL del cliente por un *Connection String* a la nueva. En esta cadena de caracteres van incluidos los credenciales. El backend la recibe como variable de entorno, por lo que no tuve siquiera que tocar su código.

47 <code>const mongourl = process.env.MONGO_URL;</code>	57 <code>removePost: async (parent, args, ctx) => {</code>
54 <code>const client = new MongoClient(mongourl);</code>	59 <code>const {db, user, token, redisClient} = ctx;</code>
67 <code>const db = client.db("test");</code>	68 <code>found = await</code>
84 <code>return{db, user, token, redisClient};</code>	<code> ↪ db.collection("posts").findOne({_id:</code>
	<code> ↪ ObjectId(postid)});</code>

(a) index.js(b) mutation.js

Figura 11: Ejemplo de uso de MongoDB en el backend (O. Salvador, 2022)

Por su parte, integrar Redis también fue sencillo. Además de alimentarle la URL en la que encontrar al servidor, he tenido que darle el puerto y contraseña, especificando en el cliente que es tráfico tunelizado, ver Figura 7 (p. 16).

Azure ofrece interfaces a ambos, para poder depurar. En el caso de CosmosDB, un interfaz gráfico, *Data Explorer* con el que conseguir la misma funcionalidad que satisface durante el desarrollo en local con el interfaz provisto por Mongo en la imagen `mongo-express:latest`.

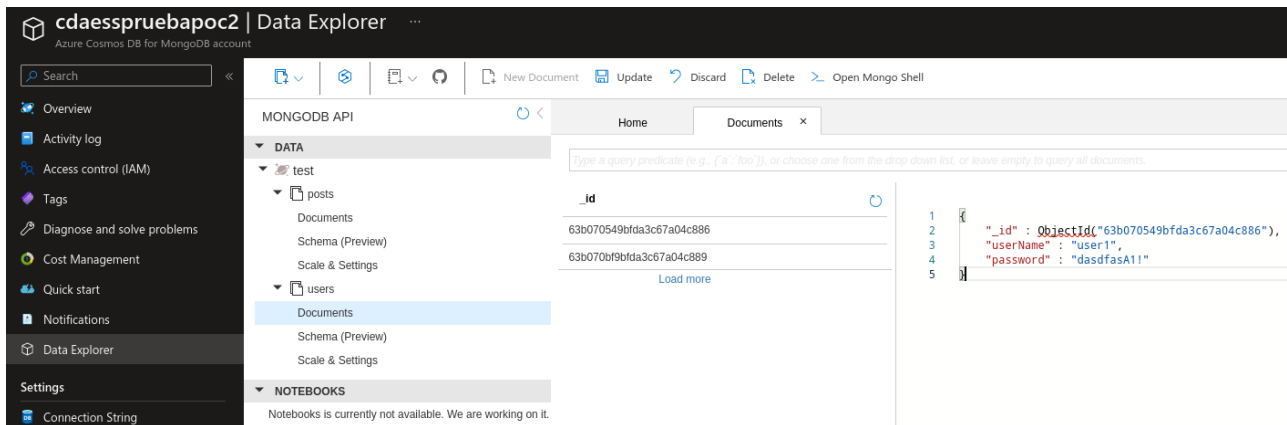


Figura 12: Azure Data Explorer, colección “users” (O. Salvador, 2022)

En el caso de Redis este es una línea de comandos, accesible desde el portal de Azure, dentro del recurso, con la opción “Console”. Permite comandos como “`keys *`” para poder ver todas las llaves que tiene guardadas en ese momento.



Figura 13: Consola de Redis desde el portal de Azure (O. Salvador, 2022)

A diferencia de los otros componentes, como el frontend, backend, o almacenamiento de blobs, tanto Mongo como Redis usan su propio protocolo. Esto implica que para poder acceder al servicio desde fuera del portal de Azure es necesario tener un cliente adecuado. En el caso de Redis, la misma imagen `redis/redis-stack:latest` incluye `redis-cli`, un cliente de Redis por línea de comandos. Cabe resaltar que en la siguiente imagen me conecté de manera insegura (puerto 6379), después de manualmente editar la configuración que despliego como código, en la que solo permito conexiones tunelizadas (puerto 6380). Es posible establecer un túnel y conectarse de manera segura, Baig, 2018, aún si no lo he hecho para esta práctica.

```
root@5f5cda6ef50a:/# redis-cli -h rdbesspruebapoc2.redis.cache.windows.net -p 6379 -a g6XGqZftgHaC0slu6kcLTDV7JJJ0a4yIIAzCaGayGzc=
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
rdbesspruebapoc2.redis.cache.windows.net:6379> info keyspace
# Keyspace
db0:keys=3,expires=3,avg_ttl=1928207
rdbesspruebapoc2.redis.cache.windows.net:6379> keys *
1) "m6vay3y0que"
2) "9rcct19x71"
3) "svf8kahovt"
```

Figura 14: Listado de llaves con `redis-cli` en contenedor (O. Salvador, 2022)

Las bases de datos son los elementos más pesados de levantar, pero curiosamente Redis tarda más que CosmosDB. Mi explicación para este fenómeno es que CosmosDB este diseñado completamente por el equipo de Microsoft, mientras que Redis se levante como aplicación de terceros en el contenedor o máquina virtual que utilicen. Aparte del motor de virtualización que use, como se ve en la Figura 13, corre sobre Windows, que siempre es más pesado que sus alternativas basadas en Linux. Como el código es cerrado, esto es solo especulación. En la todas las ejecuciones de `terraform apply` en las que despliego Redis, es el componente que más tarda, normalmente superando los veinte minutos.

```
azurerm_redis_cache.redis: Still creating... [20m40s elapsed]
azurerm_redis_cache.redis: Still creating... [20m50s elapsed]
azurerm_redis_cache.redis: Creation complete after 20m54s [id=
```

Figura 15: Tiempo necesario para aprovisionar Redis usando Terraform (O. Salvador, 2022)

3.2. Grupos de contenedores

Azure ofrece varias soluciones para hospedar aplicaciones web según el nivel de control que el cliente desee sobre la infraestructura. Sus soluciones relevantes principales son: *Azure Kubernetes Service* (AKS), *Azure Container Instance* (ACI), y *Azure Container Apps* (ACA). La primera ofrece control completo sobre el clúster en el que se ejecutan los contenedores, junto con el trabajo que implica manejarlos manualmente. La última abstrae demasiado y dificulta parte de la gestión que quería

realizar. Como resultado, he elegido utilizar grupos de contenedores, ACI para los servidores de tanto el frontend como el backend.

Para poder levantar una instancia de contenedor, Azure necesita tener su imagen disponible. Se pueden elegir imágenes de `hub.docker.com`, pero he preferido crear mi propio registro dentro del proyecto y subir ahí mis imágenes. Subirlas sigue el mismo proceso que a cualquier otro registro, después de hacer login en `azure-cli` es necesario hacerlo en el registro particular, y luego empujar la imagen.

```
$ az acr login -n <NOMBRE_DE_REGISTRO>
$ docker build -t fullstackpoc-front:1.0.0 .
$ docker tag fullstackpoc-front:1.0.0 <NOMBRE_DE_REGISTRO>.azurecr.io/fullstackpoc-front:latest
$ docker push <NOMBRE_DE_REGISTRO>.azurecr.io/fullstackpoc-front:latest
```

Figura 16: Comandos para subir la imagen del frontend (O. Salvador, 2022)

He tenido problemas para conseguir generar las imágenes, el desarrollo en local donde ambos son contenedores no es representativo de conseguir *dockerizar* las aplicaciones y prepararlas para un entorno de producción. El backend, escrito en JavaScript fue sencillo, solo necesitando una instalación de los paquetes que uso antes de poder servir los contenidos del proyecto. Por el contrario, el frontend, escrito en TypeScript fue más complicado. A parte de las dependencias de Node, son necesarias las demandadas por React, y compilar el proyecto a JavaScript. Además, no se pueden servir de la misma manera, necesita un servidor específico. Antes de usar el paquete *serve* de Node usaba una imagen de Nginx¹. Pero en ambos la página era estática, no ejecutando las peticiones GraphQL. He conseguido solucionarlo como muestro en el extracto. Compilo el proyecto inmediatamente antes de servirlo. Tropecé con esta solución por mi cuenta. No he conseguido responder el por qué así si funciona.

```
21 # CMD ["npm", "start"]
22 CMD ["node", "src/index.js"]
```

(a) Dockerfile del backend

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

(b) Dockerfile del frontend

Figura 17: Comandos de arranque de los servidores contenedorizados (O. Salvador, 2022)

¹El **Dockefile** en el que compilaba el proyecto con una imagen de Node y luego pasaba `/build` a una segunda imagen, de Nginx para servirla está disponible en la carpeta del frontend, “Dockerfile-nginx”. Me base en: Bachina, 2021, Singh, 2021 y S. Saini, 2022

Docker está orientado a capas. En mi opinión, su sistema de ficheros para estas es su valor principal, por encima del uso directo del kernel del huésped (rendimiento), frente a las máquinas virtuales tradicionales. En mis `Dockefile` uso generosamente las instrucciones “RUN” y “COPY”, cada una de ellas generando una nueva capa. Antiguamente tantas capas habrían reducido el rendimiento por sobre-abstracción, pero en las nuevas versiones no tiene tanto efecto, ya que se compactan (Docker, 2022), y mis imágenes son pequeñas. El beneficio de generar varias capas es evidente en la construcción de las imágenes y en su subida al registro (*ACR*). Docker comparte las capas entre imágenes. Esto implica que, durante la depuración, puede solo ser necesario instalar las dependencias de un proyecto una vez, todas las imágenes que compartan esa capa partiendo de su progreso. También significa que solo es necesario subir las capas que sean distintas de las que el registro ya tiene, como muestro en la figura, subiendo la imagen del frontend después de haber subido el backend.

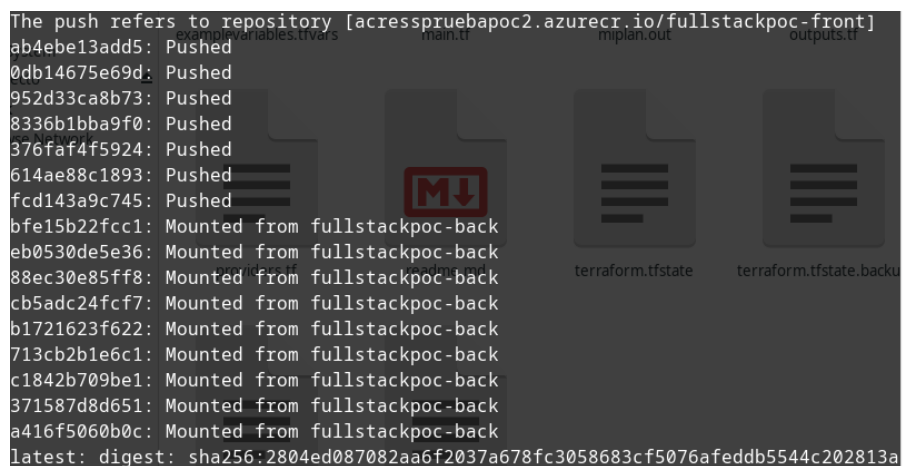


Figura 18: Azure Container Registry compartiendo capas entre imágenes (O. Salvador, 2022)

Con las imágenes disponibles, Azure puede empezar a desplegar los grupos de contenedores. Por dependencias entre los componentes, es necesario construir la infraestructura en pasos. Primero el grupo de recursos, red virtual, bases de datos, y almacenamiento de blobs. Segundo el backend. Tercero y final, el frontend.

El backend necesita tener acceso a las bases de datos y almacenamiento, al arrancar lo primero que hace es intentar montar un cliente con Redis y otro con Mongo. Si no tiene sus direcciones y credenciales como variables de entorno al arrancar, fracasara su ejecución, y Azure reiniciara el contenedor. Esto pasará en bucle. El frontend tiene la misma dependencia con el backend, e implícitamente con los demás componentes a través de el. No se pueden exportar variables en el Dockerfile, Azure no las sobrescribe. En la siguiente figura muestro, simbólicamente, las direcciones de los componentes, y como se conectan entre sí, además del orden en el que hay que aprovisionarlos.

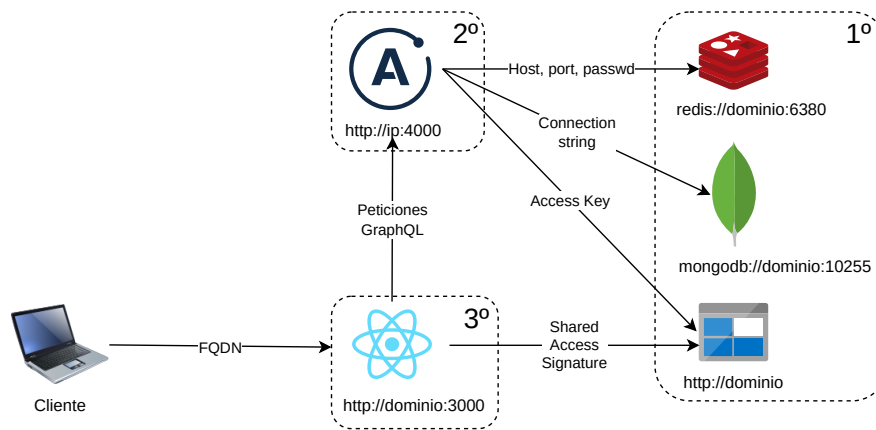


Figura 19: Diagrama a alto nivel, conexiones y orden de aprovisionado (O. Salvador, 2022)

Durante el desarrollo, ApolloServer, la librería que uso para el servidor de GraphQL en el backend, publicaba un interfaz gráfico desde el que hacer peticiones para la depuración. Por defecto este se deshabilita en producción. Se puede especificar que siga disponible, pero es mala praxis mantenerlo cuando la aplicación está publicada. Para poder seguir depurando en producción, he vuelto a usar cURL. GraphQL sigue siendo REST, aunque parezca una tecnología distinta a primera vista. A continuación, dos ejemplos de peticiones al backend: `getPosts()` (solo había un `post`) y `login()`, con el usuario con el que me identifico en el Data Explorer de CosmosDB de fondo.

```

[user@torre ~]$ curl 20.238.159.190:4000
GET query missing.[user@torre ~]$
[user@torre ~]$ curl -X POST http://20.238.159.190:4000 -d '{"query":"query{getPosts{_id, name, comment}}"}' -H 'Content-Type: application/json'
{"data":{"getPosts":[{"_id":"63b070e99bfda3c67a04c88a","name":"sdasdaasdfsdaasfasd","comment":"otro usuario"}]}}

```

```

$ curl -v -X POST http://<IP>:4000 -d '{"query":"query{getPosts{_id, name, comment}}"}' -H
'Content-Type: application/json'

```

(a) Petición `getPosts()`

```

[user@torre ~]$ curl -X POST http://20.238.159.190:4000 -d '{"query":"mutation {login(userName: \"sdasda\", password: \"dasdfasA1!\")}}"}' -H 'Content-Type: application/json'
{"data":{"login":"m6vay3y0que"}}
[user@torre ~]$

```

```

1 {
2   "_id" : ObjectId("63b070549bfda3c67a04c886"),
3   "userName" : "sdasda",
4   "password" : "dasdfasA1!"
5 }

```

```

$ curl -v -X POST http://<IP>:4000 -d '{"query":"mutation {login(userName: \" <USERNAME> \", password: \" <PASSWORD> \")}}"}' -H 'Content-Type: application/json'

```

(b) Mutación `login()`

Figura 20: Peticiones GraphQL con cURL (O. Salvador, 2022)

Todos los componentes son públicamente accesibles, aun si todos salvo los contenedores están protegidos con credenciales. Todos tienen una IP, y salvo el backend, todos tienen un FQDN. Esto significa que a todos se les pueden hacer peticiones, y descubrir información sobre ellos, como su posición. He contratado los elementos del proyecto en la región de Azure “westeurope”, que resulta ser Ámsterdam.

```
[user@virtualbox fullstack-verificacion]$ geoiplookup 20.31.29.192
GeoIP Country Edition: NL, Netherlands
```

(a) Ubicación de los servidores

```
[user@torre ~]$ traceroute acifesspruebapoc2.westeurope.azurecontainer.io
traceroute to acifesspruebapoc2.westeurope.azurecontainer.io (20.31.29.192), 30 hops max, 60 byte packets
 1  _gateway (192.168.1.1)  0.256 ms  0.294 ms  0.335 ms
 2  192.168.0.4 (192.168.0.4)  1.112 ms  1.106 ms  1.264 ms
 3  * * *
 4  * 13.red-81-46-66.customer.static.ccgg.telefonica.net (81.46.66.13)  4.832 ms  4.872 ms
 5  * 10.red-81-46-66.customer.static.ccgg.telefonica.net (81.46.66.10)  5.321 ms  18.red-81-46-66.customer
 6  * 205.red-81-46-0.customer.static.ccgg.telefonica.net (81.46.0.205)  5.675 ms  5.611 ms
 7  be33-400-grtmadix2.net.telefonicaglobalsolutions.com (216.184.113.180)  6.046 ms  216.184.113.248 (216
 8  microsoft-be10-grtmadix2.net.telefonicaglobalsolutions.com (213.140.51.165)  4.859 ms  microsoft-be18-g
 9  microsoft-be18-grtmadix2.net.telefonicaglobalsolutions.com (81.173.106.25)  4.865 ms  ae20-0.icr04.par3
10  ae20-0.icr03.par30.ntwk.msn.net (104.44.231.228)  30.443 ms  20.226 ms  be-104-0.ibr01.par30.ntwk.msn.n
11  be-104-0.ibr01.par30.ntwk.msn.net (104.44.23.100)  26.718 ms  be-5-0.ibr01.lon22.ntwk.msn.net (104.44.
12  be-15-0.ibr02.ams30.ntwk.msn.net (104.44.31.3)  28.819 ms  27.858 ms  27.262 ms
13  be-15-0.ibr02.ams30.ntwk.msn.net (104.44.31.3)  27.392 ms  27.524 ms  28.188 ms
14  * * *
15  * * *
```

(b) traceroute contra el FQDN del frontend

Figura 21: Investigación de los detalles de la infraestructura (O. Salvador, 2022)

3.3. Cuenta de almacenamiento

Este componente fue el primero que integré, en noviembre. Originalmente había planeado usar un S3 en AWS, y tuve que rehacer las peticiones para adaptarlo. No hay un contenedor local con el que simular un Blob Storage como MinIO lo es para S3. En ambas plataformas subir y borrar imágenes de sus almacenamientos ha sido la parte más difícil del desarrollo, pero AWS tenía mejor soporte y fue más cómodo y rápido. Como expliqué en la sección *Aplicación*, genero una URL con credenciales perecederos a la que subir la imagen. Cualquiera con esa URL puede subir una imagen. Después de que se suba una, los credenciales no tienen permiso para sobrescribirla. Durante el desarrollo tomaba las URL que genera el backend (no rellenaba a mano la URL que muestro) y probaba a subir imágenes manualmente, usando una vez más, cURL.

```
$ curl -v -X PUT "https://<CUENTA_DE_almacénAMIENTO>.blob.core.windows.net/
<CONTENEDOR_DE_almacénAMIENTO>/<NOMBRE_DE_IMAGEN>.png?<SHARED_ACCESS_SIGNATURE>" --data-binary
@<IMAGEN_A_SUBIR>.png -H "x-ms-blob-type: BlockBlob"
```

Figura 22: Comando para subir imágenes (O. Salvador, 2022)

La depuración de la configuración de CORS (*Cross-Origin Resource Sharing*) ha sido la parte más difícil del proyecto pese a su simpleza. Al principio no configuraba *Allowed* y *Exposed headers*, que resultó ser la solución (Microsoft: et al., 2021). Cuando no lo hacía y no funcionaba, saqué la conclusión errónea que se ve en las siguientes imágenes², que se podía usar el FQDN sin protocolo ni puerto. También muestran la dificultad de depurar, cuando todo parece estar en orden.

Allowed origins	Allowed methods	
acifesspruebapoc2.westeurope.azurecontainer.io	GET,HEAD,POST,OPTIONS,PUT	
Allowed headers	Exposed headers	Max age
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://${data.azurearm_container_group.aci_back.ip_address}:4000"
35   REACT_APP_CORS_ORIGIN_TO_ALLOW = "http://${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':
82   ↪ 'application/x-www-form-urlencoded',
83   ↪ "Access-Control-Allow-Origin": `${process.env.
84   ↪ REACT_APP_CORS_ORIGIN_TO_ALLOW}`,
85   "x-ms-blob-type": "BlockBlob",
86   "Vary" : "Origin"
87 })
88 const req = await fetch(url, {
89   method: "PUT",
90   body: imageList.at(0)!.file,
91   headers: myHeaders
92 }).then((res) => {
93   console.log(res)
94 })

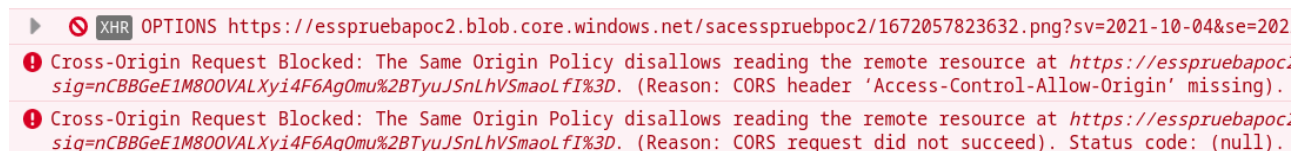
```

▼ 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

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

(d) Cabeceras en la petición del navegador



(e) Fracaso de la petición PUT a la cuenta de almacenamiento desde el cliente

Figura 23: Depuración de CORS (O. Salvador, 2022)

²Todas las imágenes están disponibles en el repositorio, en `documentacion/capturas`, para poderlas leer mejor. En particular, el apartado e), que he cortado para poderlo leer, es la imagen `azure_cors_firefox_error.png`

4 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 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 24: Sintaxis de referencia, documentación de Terraform (Hashicorp, s.f.-g, 2022)

La extensión de la mayoría de los archivos HCL en un proyecto de Terraform es **.tf**. En JSON los elementos siguen la misma estructura que en HCL, pero ligeramente adaptada y con el prefijo de terraform antes de la extensión (ej. **.tf.json**). Los nombres de los archivos son importantes, 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 último no hay bloques, solo parejas de identificadores y expresiones (sus valores). Se puede usar para poblar las declaraciones de variables.

4.1. Funcionamiento

Hashicorp ha hecho admirablemente sencillo la “instalación” de Terraform. Distribuyen un binario listo para usar, y trivialmente portátil. 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 última 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, s.f.-f). 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. Contiene información comprometedor, como credenciales. Es importante manejarlo de forma segura. Adicionalmente, el estado se puede usar 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 de la plataforma (Hashicorp, s.f.-d).

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 describen una configuración válida. Si lo es, resuelve las variables. Recorre `variables.tf` y las puebla con `variables.tfvars`. Se puede especificar un fichero de variables con “`-var-file variables.tfvars`”. Si hay más 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 sobrescribir las variables del fichero, para una ejecución, de dos maneras. Puede incluir la opción “`-var nombre='valor'`”, o exportar variables de sistema con el nombre de esta y el prefijo `TF_VAR`. Por ejemplo, para la variable “`rg_name`”, exportar en la misma consola la variable “`TF_VAR_rg_name`”.

Lo siguiente que hace es recuperar el estado actual de cualquier objeto que ya este creado en la plataforma (Hashicorp, s.f.-b, Hashicorp, s.f.-i). 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 recorriéndose los archivos fuente, 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 el campo `depends_on` al que mencionen (Hashicorp, s.f.-e). Después de añadir los nodos al grafo con las dependencias adecuadas, etiqueta a cada uno con metadatos 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 más sencillo, el frontend (detalles en la siguiente sección), es solo una parte para no dedicarle una página³.

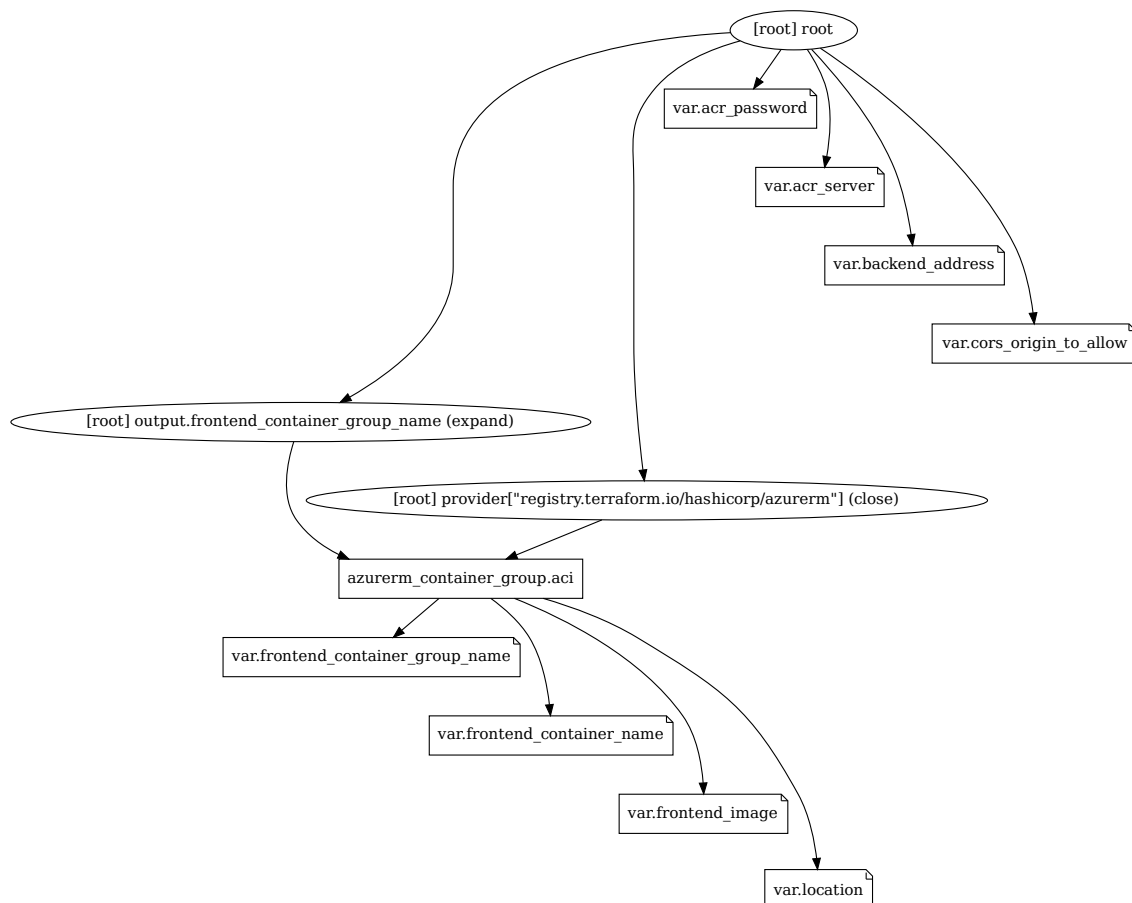


Figura 25: Extracto del grafo del frontend (O. Salvador, 2022)

³He generado los diagramas completos para todos los proyectos, en el Anexo A detallo como generarlos. Están en la carpeta `documentacion/terraform-graph`

Finalmente, Terraform calcula y muestra la serie de pasos que seguirá para alcanzar el estado deseado, si hay diferencias. Los cambios a la infraestructura propuestos pueden ser: crear nuevos recursos, destruir recursos existentes, y actualizar un recurso existente, que a veces requiere la destrucción y sustitución (*creación in-place*) de este.

```

+ container {
+   commands = (known)
+   cpu      = 0.5
+   environment_variables = {
+     "MONGO_URL" = "mongodb://c

```

(a) Creación

```

- container {
-   commands = []
-   cpu      = 0.5
-   environment_variables = {
-     "MONGO_URL" = "mongodb://

```

(b) Destrucción

```

~ container {
~   commands = [] -> (known after apply)
~   environment_variables = { # forces replacement
+     "MONGO_URL" = "https://cdaesspruebapoc2.documents.azure.com:443/"
# (1 unchanged element hidden)
}
name = "acicesspruebapoc2"
- secure_environment_variables = (sensitive value)
# (3 unchanged attributes hidden)

```

(c) Reemplazo

Figura 26: 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, s.f.-b).

El penúltimo comando mencionado es **terraform apply**. Implícitamente desata una planificación, con los pasos antes descritos, incluida la importación del estado actual en la plataforma.

Alternativamente se le puede alimentar un plan ya calculado incluyendo su nombre (ej. **terraform apply FICHERO_DESTINO.out** para el plan anterior), al haber resuelto las variables para este, no es necesario volverlo a hacer. Terraform utiliza el grafo de nodos para crear los recursos. Lo usa para encontrar nodos sin dependencias, y poder paralelizar sus operaciones. Por defecto busca un paralelismo de diez recursos, se puede cambiar con “`-parallelism=<N>`”.

Para quitar los recursos descritos en los fuentes de configuración de la plataforma, y dejarla yerma, Terraform ofrece dos opciones. La primera es el último comando que presenté, **terraform destroy**. La otra opción es realizar un nuevo **terraform apply**, añadiendo la opción “`-destroy`”. En ambos el resultado es el mismo, Terraform recorrerá el grafo de dependencias eliminando todos los nodos. Si es posible, paralelizará las operaciones.

Durante todos los comandos y sus pasos, Terraform necesita interactuar con la plataforma objetivo. Lo consigue a través de proveedores. Estos son *plug-ins*, componentes adicionales que añaden la funcionalidad específica para su plataforma particular. Esta no tiene que ser en la nube, hay proveedores para aprovisionado en plataformas *on-prem*: Hashicorp, s.f.-h, terraform-provider-openstack, s.f., Cloudstack, s.f. Permiten separar el desarrollo de proyectos Terraform con configuraciones a alto nivel, uniforme entre plataformas, abstrayendo los detalles específicos de la implementación de su API correspondiente.

Según Hashicorp, “Las principales responsabilidades de los Plugins Proveedores son:

- Inicialización de cualquier librería incluida que se utilice para realizar llamadas a la API.
- Autenticación con el proveedor de infraestructura
- Definir los recursos que se asignan a servicios específicos

” (Hashicorp, s.f.-c § Terraform Plugins)

El binario principal, *Terraform Core*, no incluye proveedores. En su lugar, cuando se ejecuta `terraform init` busca en los archivos de configuraciones del proyecto y descarga del *Terraform Registry* los proveedores que encuentre declarados. No son necesarios más pasos.

```
1 terraform {  
2   required_providers {  
3     azurearm = {  
4       source = "hashicorp/azurearm"  
5       version = "3.0.0"  
6     }  
7   }  
8 }
```

(a) Declaración

```
10 provider "azurearm" {  
11   features {}  
12  
13   subscription_id = var.subscription_id  
14   tenant_id       = var.tenant_id  
15 }
```

(b) Configuración

Figura 27: Declaración y configuración del proveedor, `providers.tf` (O. Salvador, 2022)

En el *Registry* hay más de mil proveedores hechos por Hashicorp y por la comunidad, señalados como tal. También contiene *módulos*, pequeños proyectos, configuraciones que permiten trabajar con conjuntos de recursos como uno solo. Un usuario de Terraform puede utilizarlos, ya que son públicamente accesibles, o crear los suyos propios. Crear módulos propios fomenta el código reutilizable, y en un proyecto mayor o para una organización, tener estas plantillas facilita el desarrollo. Para las necesidades de este proyecto he considerado que superaban su ámbito.

4.2. Implementación

He diseñado la infraestructura al mismo tiempo que aprendía como montarla en Terraform, por el beneficio de poderla eliminar y montar a menudo, reduciéndome el coste. Lo reflejaré en esta sección, explicando varios pasos en falso que tomé. He usado la documentación de Terraform para el proveedor de Azure extensivamente durante el desarrollo, buscando la entrada para cada recurso. Con tal de no desbordar las referencias solo la referiré aquí: Hashicorp, s.f.-a.

Como mencioné en la sección *Infraestructura e integración*, el frontend necesita al backend antes de crearse, y este a los demás componentes. Para solucionar esto, he dividido la infraestructura en tres proyectos de Terraform. Un primer proyecto `/terraform` monta todo salvo los grupos de contenedores. Después, es necesario entrar a las carpetas `/backend/terraform` y `/frontend/terraform` y aplicarlos, en ese orden. Estos dos necesitan encontrar las imágenes Docker con su código en el ACR. He conseguido una implementación en la que los proyectos sucesivos recuperan por su cuenta los valores que necesitan de los recursos generados en sus predecesores.

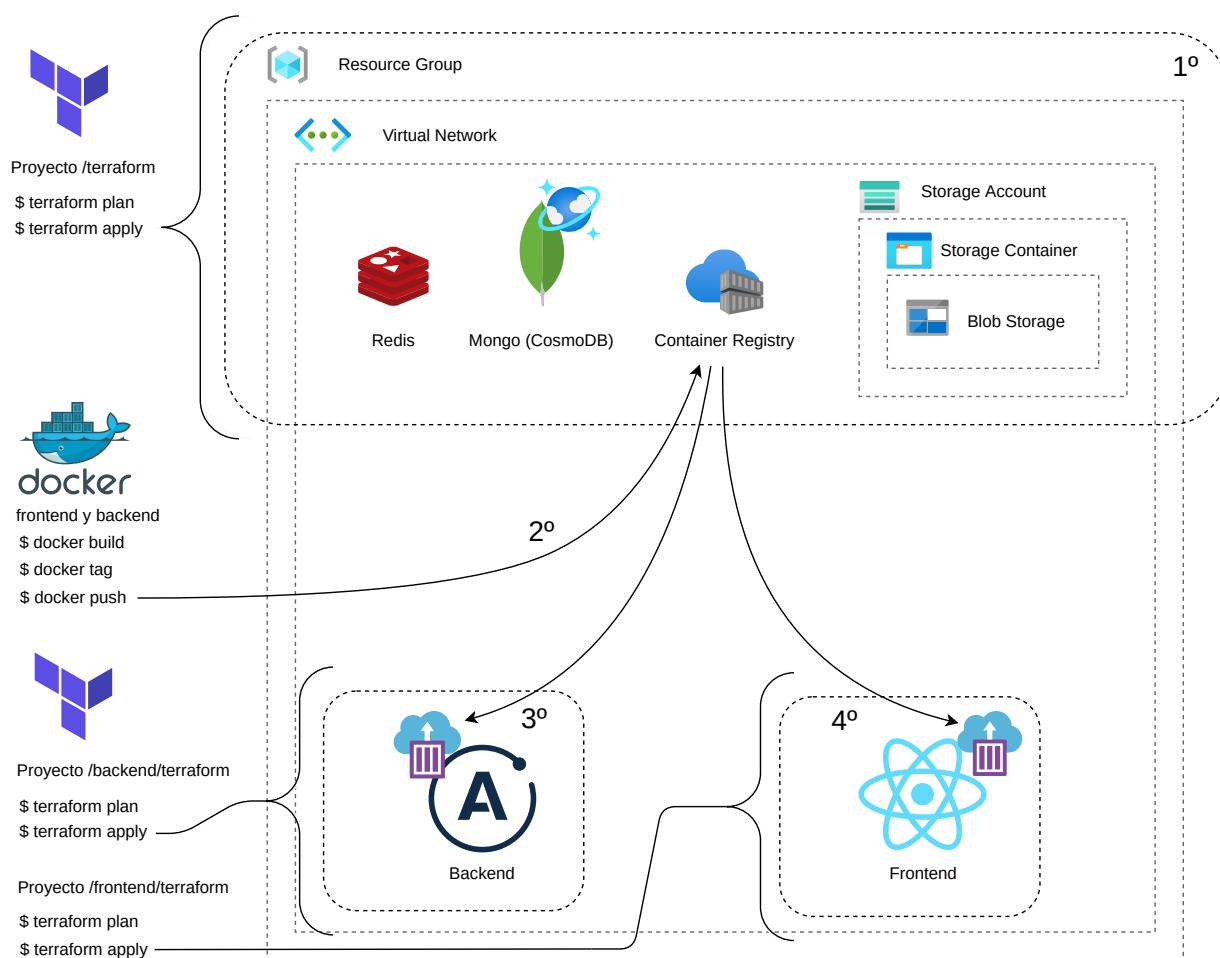


Figura 28: Tres proyectos de Terraform y dos imágenes de Docker (O. Salvador, 2022)

Terraform es capaz de encontrar los recursos porque tiene sus nombres y porque uso bloques *data*. Para que tenga los nombres es necesario declararlos en el `variables.tfvars` de cada proyecto. Podría usar un único archivo para los tres, pero como expliqué en la subsección anterior, causaría avisos, ya que Terraform vería que hay variables en el archivo que no son usadas por el proyecto. Dado que solo es necesario escribir los nombres de los recursos que buscar, se pueden llenar los tres `.tfvars` a priori.

En el siguiente extracto utilizo la variable con el nombre y grupo de recursos de la cuenta de almacenamiento para identificarla. Con esos dos datos, Terraform entiende que tiene que buscar el recurso y resolver sus detalles al llamarlo por su pareja de etiquetas. Las dos etiquetas de un bloque detrás su tipo son el de recurso dentro del proveedor, y el nombre por el que identificar el recurso durante el `plan` y `apply` (en el grafo de dependencias) del proyecto. Al juntar estas dos, se identifica al recurso inequívocamente y se puede acceder a sus propiedades. En el caso de la cuenta de almacenamiento, el backend necesita ser alimentado: su nombre, el nombre del contenedor de almacenamiento que he creado en el primer proyecto, y la llave de administración. Las dos primeras se saben antes de montar la infraestructura, pero la llave se genera aleatoriamente con cada despliegue del proyecto. Usar el bloque *data* y referirme a sus contenidos como muestro ahorra al usuario actualizarlo su valor de manera manual.

```
6  data "azurerm_storage_account" "sa" {
7    name = var.storage_account_name
8    resource_group_name = var.resource_group_name
9  }

21 resource "azurerm_container_group" "aci" {

45   container {

51     environment_variables = {

56       STORAGE_ACCOUNT_NAME = data.azurerm_storage_account.sa.name
57       STORAGE_CONTAINER_NAME = var.storage_container_name
58       STORAGE_ACCOUNT_KEY = data.azurerm_storage_account.sa.primary_access_key
59     }
  }
```

Figura 29: Uso de bloques *data* para las variables de entorno de los contenedores (O. Salvador, 2022)

Terraform usa el proveedor para recuperar la información de recursos que se han creado en proyectos anteriores (y que por tanto no están disponibles por parejas de etiquetas) usando el API de la plataforma. Luego alimento los detalles que necesito de estos recursos a los contenedores, y el código

en ellos se los encuentra como variables de entorno, completamente abstraído de cómo han llegado hasta ahí. Es de esta manera que puedo pasar las variables de entorno que piden en su código a los contenedores del backend (connection string de Mongo; dirección y credenciales de Redis; y credenciales de la cuenta de almacenamiento) y frontend (dirección del backend), actualizándolas automáticamente.

Otro ejemplo de los bloques de datos recuperando detalles de la plataforma son los credenciales del ACR, que está protegido con usuario y contraseña. Con un bloque similar al de la cuenta de almacenamiento, Terraform lo encuentra y los recupera, para poderlos usar en aprovisionado del ACI.

La etiqueta de tipo de recurso del proveedor normalmente le sirve a este para inferir las dependencias con los demás recursos. Sin embargo, durante mi desarrollo tuve una ocasión en la que falló. Al aprovisionar la cuenta de almacenamiento configuro sus contenedores de almacenamiento. Esta configuración tiene que ir en el bloque de la cuenta, pero afecta al contenedor. El contenedor se debería crear después de la cuenta, ya que cuelga de ella en sus propiedades. Aun así, encontraba el siguiente error.

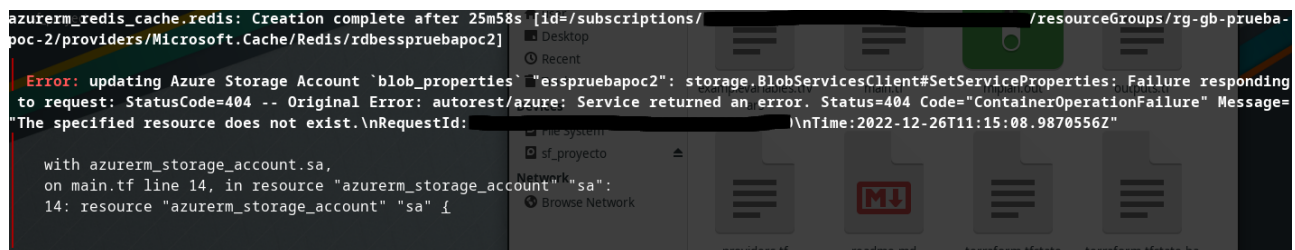


Figura 30: Extracto del grafo del frontend (O. Salvador, 2022)

Encontré dos *issues* cerrados (Bernadim, 2019, realsnick, 2016) y uno abierto (Wellington, 2021) con problemas similares. En mi caso la solución fue añadir una declaración dependencia explícita a la cuenta en el contenedor, con la instrucción `depends_on`. Esta no se puede poner en la configuración de la cuenta, y no tiene sentido asignársela al bloque principal de ella. Teorizo que sea un error del proveedor de Azure interactuando con el grafo de dependencias.

La configuración por la que tuve estos problemas es importante, precisamente las propiedades de CORS del recurso, como muestro en la captura del portal de Azure en la Figura 23 (p. 27). En esta especifico las cabeceras y métodos permitidos y expuestos, y en particular el origen del que se permiten referencias, el frontend. Antes de averiguar cómo implementar la seguridad de CORS usaba *Wildcards*, que aceptaban cualquier valor, bueno para depuración, pero deshacen el valor de la funcionalidad.

```

14 resource "azurerm_storage_account" "sa" {
26
27     blob_properties {
34         cors_rule {
35             allowed_origins =
36                 ↪ ["http://${var.frontend_container_group_name}.${var.location}.azurecontainer.io:3000"]
37             allowed_methods = ["GET", "HEAD", "POST", "OPTIONS", "PUT"]
38             allowed_headers = ["x-ms-blob-type", "access-control-allow-origin", "vary"]
39             exposed_headers = ["x-ms-blob-type", "access-control-allow-origin", "vary"]
40             max_age_in_seconds = 86400
41         }
42     }

```

Figura 31: Configuración de CORS del almacenamiento, `main.tf` (O. Salvador, 2022)

También en este componente, pero independiente a los problemas de CORS, durante el desarrollo usaba una configuración de red que negaba cualquier acceso salvo aquel desde direcciones IP permitidas, la mía. Hacía esto para que un tercero no usara mi almacenamiento sin permiso, pero inhabilita la aplicación para producción.

Finalmente, y separado del almacenamiento, también me gustaría mencionar la configuración de sondas de estado de los contenedores en las ACI. Si tienen menos tiempo, en particular el frontend, que hace la compilación de los fuentes en Azure, se marcan como muertos y son reiniciados, en bucle.

```

21 # network_rules {
22 #     default_action = "Deny"
23 #     ip_rules = var.whitelisted_ip
24 # }

```

Figura 32: Configuración de red limitando por IP, `main.tf` (O. Salvador, 2022)

```

67 liveness_probe {
68     http_get {
69         path = "/"
70         port = 4000
71         scheme = "Http"
72     }
73     period_seconds = 60
74     failure_threshold = 20
75     timeout_seconds = 50
76     initial_delay_seconds = 240
77 }

```

Figura 33: Configuración de sonda de estado, backend, `main.tf` (O. Salvador, 2022)

5 Ansible

Ansible es, ante todo, una herramienta de automatización. Es capaz de abordar la mayoría de las responsabilidades de la administración de sistemas. Originalmente fue desarrollado por Michael DeHaan, y después adquirido por Red Hat. Ha sido adoptado por una comunidad y empresas que han añadido muchas funcionalidades (Ansible, 2022d). Como resultado, hay dos distribuciones: *Community Ansible*, publicado bajo la licencia *GPLv3*; y *Red Hat Ansible Automation Platform*, varios proyectos distribuidos como un único producto. Durante esta práctica sólo uso la primera.

Entre las responsabilidades que automatiza están la gestión de configuraciones, despliegues, ejecución de tareas, y más recientemente, el aprovisionado de infraestructura como código. Está escrito principalmente en Python (Ansible, 2023), pero es extensible, y los *módulos* pueden estar escritos en cualquier lenguaje.

Como resultado de las necesidades del resto de servicios que ofrece, Ansible utiliza un *nodo de control* y uno o más *nodos gestionados*, a los que también se refieren como *hosts*. Es una herramienta *agentless*, es decir, que no necesita tener un cliente en las maquinas que gestiona, solo estar instalado en el nodo de gestión. Lo único que deben tener los nodos gestionados es *SSH* instalado, con lo que se conecta a ellos y ejecuta los comandos necesarios.

Utiliza archivos con grupos de *tasks* que realizar, descritas de forma declarativa en formato *YAML*. En estas tareas se especifican los detalles del resultado deseado, sin información de como alcanzarlo. Los *YAML* con tareas se pueden ejecutar solos, o agrupar en *roles*, que a su vez son parte de un *playbook*.

Un *playbook* puede tener tantos roles como sea necesario, que son ordenados y asignados a un nodo gestionado en un *play*. Este es otro archivo *YAML*, que se sienta a nivel raíz del *playbook*. En él se utilizan nodos gestionados declarados en el fichero *hosts* dentro del inventario, asignando a cada uno el rol que le corresponda. En esta práctica solo he usado Ansible con *playbooks*, en lugar de definiendo las tareas en *YAMLs* sueltos y desorganizados. Los roles y tareas dentro de un *playbook* pueden ser ejecutados independientemente, usando etiquetas para elegir cuales se van a aplicar en tiempo de ejecución.

```
playbook
  inventory
  hosts
  roles
    role-1
    ...
    role-N
      files
      handlers
      meta
      tasks
        main.yml
      templates
      vars
  play
```

Figura 34: Árbol de archivos de un *playbook* (O. Salvador, 2022)

Para que las tareas realicen las funciones que se declaran, Ansible usa módulos. Según su documentación, “El código o los binarios que Ansible copia y ejecuta en cada nodo gestionado (cuando es necesario) para llevar a cabo la acción definida en cada tarea” (Ansible, 2022b). Son piezas sueltas, cada una consiguiendo un fin, solo una siendo usada en cada tarea, pero varias pudiendo ser usadas en el conjunto de tareas de un playbook.

Ansible usa *colecciones* para agrupar y distribuir más fácilmente estos módulos. En este proyecto, que solo trabajaba con Azure, la única que he instalado es `azure.azurecollection`. Se pueden instalar con *Ansible Galaxy*, un gestor por línea de comandos, similar a un gestor de paquetes. Cabe resaltar que la instalación no es sencilla, y las dependencias, al menos en el caso de Azure fueron un obstáculo. Tardé más tiempo en conseguir una instalación utilizable que en aprender y hacer mi primer playbook.

5.1. Funcionamiento

Con los componentes básicos de Ansible explicados, se puede entender el formato de las plays y tareas. Las plays relacionan un nodo gestionado particular con unos roles específicos, con una conexión y opcionalmente una *estrategia* para ejecutarlos. Las tareas invocan un módulo y le alimentan unos parámetros para que sepa los detalles de su ejecución.

```
- name: <Nombre del play>
  host: <Nodo gestionado destino>
  connection: <Tipo de conexion>
  gather_facts: <yes/no>
  strategy: <Tipo>
  roles:
    - role1: <Nombre de la carpeta>
      tags: ['']
    - role2: <Nombre de la carpeta>
      tags: ['']
```

Figura 35: Formato de una Play
(O. Salvador, 2023)

```
- name: <Nombre de la tarea>
  modulo:
    campo1.del_modulo: <valor>
    campo2.del_modulo: <valor>
  register: <Nombre para los resultados>
```

Figura 36: Formato de una tarea
(O. Salvador, 2023)

Un usuario del playbook puede invocarlo, dentro de su carpeta raíz, con el comando `ansible-playbook play`, donde “play” es el archivo YAML con las relaciones de nodos a roles.

En este punto, el usuario también puede añadir las etiquetas de roles que se quieren ejecutar y evitar, y cualquier variable que se quiera sobrescribir o añadir a la ejecución.

Cuando se lance, Ansible leerá el archivo play y empezará a ejecutar cada entrada. Una a una, buscará su “host” en el inventario, la lista de nodos gestionados. Si lo encuentra, intentará conectarse a él como se le ha especificado en la entrada. Para infraestructura, no es necesario conectarse a un host remoto. Pero Ansible está diseñado para realizar sus tareas después de hacerlo, por lo que es necesario realizar una conexión, aun si es a la misma máquina desde la que se lanza, *localhost*.

Si se especifica filtrado por etiquetas, con las opciones “`--tags 'A'`” o “`--skip-tags 'B'`”, ejecutará solo los roles que tengan las etiquetas mencionadas, y dentro de ellos, saltará los que no se desean. Las etiquetas se pueden declarar en los roles, y serán heredadas por las tareas, o por cada una, y se respetarán al entrar al rol, ignorando las que no cumplan la selección. En el host, y por cada rol, se realizan las sustituciones de las variables usadas en las tareas por sus valores en los archivos de variables que se incluyan, que se encuentren en el host, que se hayan heredado del rol, y de la ejecución del playbook con la opción “`-e 'nombre=valor'`”. Tarea a tarea, Ansible descargará al host el módulo que esta use, y tomará los pasos que él vea necesarios para conseguir el estado final que se ha declarado en ella.

Aunque es declarativo, Ansible ejecuta las tareas de manera secuencial, en el orden que las encuentra en el YAML. Esto lo hace, en cierto modo, procedimental, solo que a alto nivel. Si se ha usado el campo “register”, se guardará el resultado de la ejecución bajo el nombre que se le haya dado, y este quedará disponible como variable para el resto de las tareas que vengan detrás. Se pueden imprimir por pantalla con la tarea “`- debug:`” a la misma altura que las demás, que se puede declarar en cualquier punto del YAML (salvo dentro de otra tarea), y muestra esas o cualquier otra variable a la que tenga acceso. Las variables se guardan en formato JSON, por cuyos atributos se puede navegar sencillamente con punto y nombre.

Las variables que se hayan generado en un rol pueden ser usadas en roles que lo sigan. Esto se puede conseguir exportándolas como variables del sistema al host, y recuperándolas como *facts* del host en el rol que las consuma con *hostvars* (Ansible, 2022h; Ansible Core Team, 2022; Ansible Core Team y DeHaan, 2022; K. Saini, 2021). He optado por no hacerlo en el proyecto ya que me parece una implementación menos limpia que la alternativa, consultar esos valores a Azure con otras tareas y hacerlas disponibles dentro del rol en el que se usan; y porque aumenta la dependencia entre roles, obligando a su ejecución secuencial, que he usado raras veces durante el desarrollo, y puede no ser el único caso de uso en producción.

En la siguiente figura muestro los resultados de dos tareas. La primera crea un grupo de recursos en Azure, y registra su resultado. La segunda imprime el valor del resultado de la primera, que tiene formato JSON. En la figura se puede ver además el estado del resultado de las tareas.

```
TASK [infra-base : Create a resource group in azure] *****
changed: [localhost]

TASK [infra-base : debug] *****
ok: [localhost] => {
  "rg": {
    "changed": true,
    "contains_resources": false,
    "failed": false,
    "state": {
      "id": "/subscriptions/                               /resourceGroups/rg-gb-prueba-poc-3",
      "location": "westeurope",
      "name": "rg-gb-prueba-poc-3",
      "provisioning_state": "Succeeded",
      "tags": null
    }
  }
}
```

Figura 37: Ejecución de una tarea de Ansible y su resultado (O. Salvador, 2023)

Ansible imprime los estados de cada tarea después de realizarla, y un resumen al acabar todas las del playbook que se hayan seleccionado. Si una tarea falla durante la ejecución, a no ser que se haya indicado lo contrario, se parará la ejecución. Se pare o no, el número de tareas fallidas se mostrará en rojo, igual que las exitosas en verde y los cambios en naranja. Aparte de estos resultados, en el ámbito de esta práctica, puede darse el resultado *skipped*, que explicaré en la siguiente sección.

```
PLAY RECAP *****
localhost           : ok=19   changed=10   unreachable=0   failed=0   skipped=0   rescued=0   ignored=0
```

Figura 38: Resumen de ejecución de un playbook, resultados de las tareas (O. Salvador, 2023)

Es importante señalar que las tareas son procesos inconexos para Ansible, no tiene un modelo general o entendimiento de ellas, y solo la inteligencia que los desarrolladores del playbook y de las colecciones que use hayan embebido en estos. La colección `azure.azcollection` es idempotente, es decir, es capaz de detectar cuando el componente que se le ha mandado crear en una tarea ya existe, y no crearlo de nuevo. Sin embargo, no entra a comparar los detalles. Si existe un recurso en su grupo con el mismo nombre, pero una configuración distinta, no lo reemplazara por la nueva. Tampoco le cambiará la configuración si el usuario no lo ha programado. Salvo que haya sido contemplado, será necesario borrar el recurso y volver a ejecutar el playbook.

5.2. Implementación

He explicado el funcionamiento general de Ansible para aprovisionar infraestructura, ahora detallaré como lo he usado yo para generar la infraestructura necesaria para este proyecto. He desarrollado los playbooks de Ansible después de tener un diseño claro de la infraestructura, y los componentes integrados. Esto y haber desarrollado antes los proyectos de Terraform me ha permitido declararla muy rápidamente. Este tiempo es también un testamento a lo fácil que Ansible ha hecho el desarrollo.

Igual que en la sección equivalente de Terraform, solo referiré la documentación de los módulos una vez, aunque la haya usado para crear todas las que uso: Microsoft, 2022.

En esta práctica no he usado las carpetas “files”, “handlers” y “meta” –que menciono en la Figura 34 (p. 37)– en ninguno de mis playbooks, ya que sus funcionalidades superan mis necesidades en el ámbito de esta. Las que añaden son: copiar ficheros al host, hacer que eventos esperen a disparadores en este, y metadatos de las dependencias respectivamente (Ansible, 2022g).

Como ya he mencionado en secciones anteriores, las dependencias entre los recursos y procesos para crear los recursos dicta el orden en el que se tienen que crear. En el playbook `ansible/ansible` (todos los playbooks están en la carpeta “ansible”) he creado cuatro roles. Tres para la infraestructura que en Terraform uso proyectos, y otro para Docker, aprovechando la versatilidad de Ansible fuera de sólo IaC. Con un solo comando, “`ansible-playbook site.yml`”, dentro del playbook, se puede desplegar el proyecto entero. La play sigue el formato de la Figura 35 (p. 38).

```
1  - name: Provision the project's resources and build docker images
2    hosts: localhost
3    connection: local
4    gather_facts: no
5    roles:
6      - role: infra-base
7        tags: ['base']
8
9      - role: docker
10       tags: ['docker', 'backend']
11       vars:
12         targetimage: backend
13
14      - role: backend
15       tags: ['backend']
```

Figura 39: Play, con los tres primeros roles, `ansible/site.yml` (O. Salvador, 2023)

Dentro de cada rol, las tareas se ejecutan y usan los módulos como he descrito en la sección anterior. El fichero de variables que incluyo está vacío, es solo un ejemplo, por seguridad.

```
1 - include_vars:
2   file: '../vars/main.yml'
```

```
14 - name: Create a resource group in azure
15   azure_rm_resourcegroup:
16     name: '{{ resource_group_name }}'
17     location: '{{ location }}'
18     auth_source: cli
19     register: rg
20
21 - debug:
22   var: rg
```

(a) Creacion del grupo, que resulta en la
Figura 37 (p. 40), /tasks/main.yml

```
1 location:
2 subscription:
3 tenant:
4
5 resource_group_name:
6 virtual_network_name:
7 container_registry_name:
8
9 # storage
10 storage_account_name:
11 storage_container_name:
12 frontend_container_group_name:
```

(b) Extracto del fichero de variables,
/vars/main.yml, que carga el rol

Figura 40: Rol **infra-base**, variables, carga y uso (O. Salvador, 2023)

Solo he tenido que desarrollar un rol de Docker porque sus variables están parametrizadas, y las resuelvo según la variable “targetimage” que le paso desde el play. Con un condicional, importo el fichero de variables de los roles “backend” o “frontend”, y uso su nombre para resolver los valores que apuntan a carpetas, ya que tienen el mismo.

```
1 - include_vars:
2   # file: "../vars/main.yml"
3   file: '{{ "../backend/vars/main.yml" if targetimage == "backend" else
↪  "../frontend/vars/main.yml" }}'
```

```
26 - name: Build image and with build args
27   community.docker.docker_image:
28     # name: '{{ acr.registries[0].login_server }}/{{ backend_image }}'
29     name: '{{ backend_image if targetimage == "backend" else frontend_image }}'
30     tag: 1.0.0
31     build:
32       # dockerfile: '../{{ targetimage }}/Dockerfile'
33       path: '../{{ targetimage }}'
34     source: build
35     register: build
```

Figura 41: Inclusión y uso de variables condicionales /vars/main.yml (O. Salvador, 2023)

Los módulos de la colección de Azure son capaces de preguntar por detalles de recursos además de crearlos y configurarlos, y esta funcionalidad está disponible al desarrollador. Es valiosa para los recursos que consumen variables que cambian en cada despliegue de la infraestructura, igual que los bloques *data* lo eran en Terraform.

Continuando con el rol “docker”, y mostrando otro ejemplo con el rol “backend”, los resultados de las peticiones pueden ser accedidas por el JSON de la respuesta registrada. Ambas tareas usan un módulo distinto al que se usa para crear los recursos. Estos son específicos para peticiones, y tienen el sufijo “_info”. En los dos utilizo campos añadidos para indicar los valores quiero recuperar: los credenciales y servidor del ACR; y las llaves y dirección de Redis. Estos valores quedan disponibles para el resto de las tareas que estén después en el YAML de sus roles respectivos.

```
16 - name: Get instance of Registry
17   azure_rm_containerregistry_info:
18     resource_group: '{{ resource_group_name }}'
19     name: '{{ acr_name }}'
20     retrieve_credentials: true
21     register: acr
22
23 - debug:
24   msg: '{{ acr.registries[0].credentials.username }} {{
↵   acr.registries[0].credentials.password }} {{ acr.registries[0].login_server }}'
```

(a) Detalles y credenciales al ACR, `docker/main.yml`

```
14 - name: Get Azure Cache for Redis with access keys by name
15   azure_rm_redis_info:
16     resource_group: '{{ resource_group_name }}'
17     name: '{{ redis_db_name }}'
18     return_access_keys: true
19     register: rdb
20
21 - debug:
22   msg: '{{ rdb.redisconfigs[0].host_name }} {{ rdb.redisconfigs[0].access_keys.primary }}'
```

(b) Detalles y llaves a Redis, `backend/main.yml`

Figura 42: Peticiones de información en los roles `docker` y `backend` (O. Salvador, 2023)

5.3. Funcionamiento paralelizado

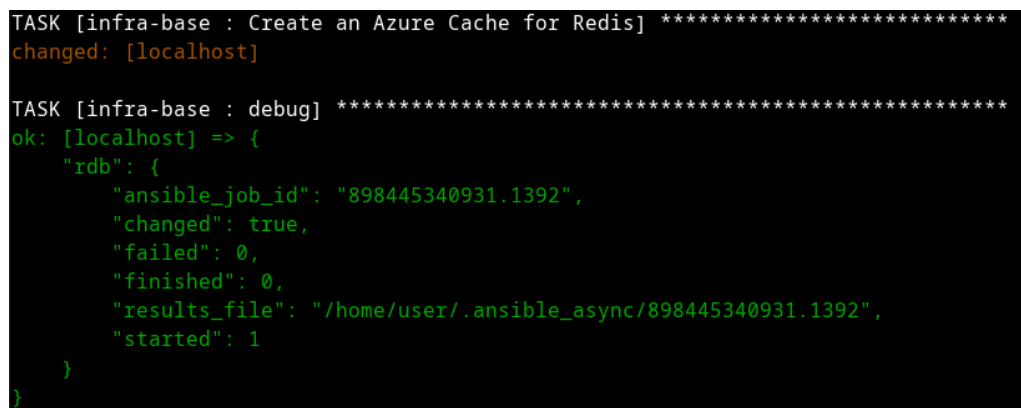
Ansible es capaz de ejecutar dos o más roles al mismo tiempo, y dentro de cada uno, también varias tareas a la vez. Para dar una mejor representación de las capacidades de la herramienta, he vuelto a implementar el playbook anterior aprovechando estas funcionalidades.

Las tareas se pueden paralelizar haciéndolas asíncronas, de manera que las siguientes no tienen que esperar a que acabe su ejecución para empezar. Ansible continúa siguiendo el orden en el que aparecen para lanzarlas, pero así, una más corta puede lanzarse y acabar durante la ejecución de una más larga, potencialmente reduciendo el tiempo de ejecución del rol a tan solo el de la tarea más larga. Para hacer que una tarea se ejecute de forma asíncrona necesita recibir dos campos a la altura del nombre en su declaración: `async` y `poll`. Estos tienen que estar acompañados de valores numéricos: el tiempo máximo de la tarea antes de cancelarla y el tiempo entre comprobaciones de su estado (Ansible, 2022c). Asignar a la segunda el valor '0' hace que no se pregunte durante la ejecución, conocido como *fire and forget*

```
- name: <Nombre de la tarea>
  modulo:
    campo1_del_modulo: <valor>
    campo2_del_modulo: <valor>
  register: <Nombre para la ejecución de la tarea>
  async: <Segundos antes de abortar la tarea>
  poll: <Segundos entre peticiones de estado>
```

Figura 43: Formato de una tarea asíncrona (O. Salvador, 2023)

Cuando una tarea es asíncrona, en la variable de resultados que se registre no aparece el JSON que devuelve la ejecución de la tarea, sino otro con información sobre la ejecución de esta.



```
TASK [infra-base : Create an Azure Cache for Redis] *****
changed: [localhost]

TASK [infra-base : debug] *****
ok: [localhost] => {
  "rdb": {
    "ansible_job_id": "898445340931.1392",
    "changed": true,
    "failed": 0,
    "finished": 0,
    "results_file": "/home/user/.ansible_async/898445340931.1392",
    "started": 1
  }
}
```

Figura 44: Valor registrado de tarea asíncrona (O. Salvador, 2023)

Igual que las tareas, los roles se pueden ejecutar de manera concurrente. Se puede conseguir usando la estrategia **free**. Esta se especifica en el campo opcional **strategy**, en la invocación al rol, como se ve en la Figura 35 (p. 38). Si no se obvia el campo, el rol se ejecuta de manera secuencial, como si se hubiese declarado con el valor **serial**. La primera le indica a Ansible que ejecute las tareas del rol sin esperar a los demás hosts (Ansible, 2022a). Como con las tareas, esto permite a roles rápidos no tener que esperar a que se complete uno lento, o que se haya quedado atascado.

Ansible crea nuevos procesos, *forks* para manejar el paralelismo que se le pida. Por defecto, tiene un límite de cinco forks, pero este se puede cambiar por N añadiendo “**forks = <N>**” al fichero **ansible.cfg** en la raíz del playbook (Ansible, 2022f).

5.4. Implementación paralelizada

Habiendo ya explicado la implementación del playbook sobre el que está basado este, y extractos de su código, solo explicaré las diferencias en el código y el comportamiento a alto nivel. Usando los mismos roles con tareas asíncronas, he implementado dos plays, una en la que los roles se llaman de manera secuencial, y otra en la que los roles de Docker que invocan con la estrategia “free”.

En la explicación del funcionamiento he mencionado que el valor registrado de una variable, al hacerse asíncrona, queda sobrescrito por información sobre su ejecución. Para conseguir los mismos valores que en el playbook convencional, es necesario, como se puede entender, esperar a que la tarea acabe su ejecución. Esto requiere la creación de una tarea nueva, con la lista de atributos **ansible_job_id** de las tareas a las que esperar. Hasta que se resuelvan todas, o se agoten los **retries**, entre los cuales pasarán los segundos que se asignen al campo **delay**, la tarea bloqueará el rol. En su artículo, Wenzin, 2018, me ayudó a entender la implementación de este comportamiento.

```
163 - name: Wait for Registry, CosmosDB
    ↪ and Redis
164   async_status:
165     jid: '{{ item }}'
166   register: wait
167   until: wait.finished
168   retries: 100
169   delay: 30 # delay between retries
170   with_items:
171     - '{{ acr.ansible_job_id }}'
172     - '{{ db.ansible_job_id }}'
173     - '{{ rdb.ansible_job_id }}'
```

Figura 45: Tarea de espera en **parallel_ansible/infra-base/main.yml** (O. Salvador, 2023)

En la siguiente figura se pueden ver varias tareas asíncronas, como se lanzan todas sin esperar al resultado de la anterior, y luego se espera a que acaben una a una. Estas son las peticiones de

detalles, que en la implementación anterior habrían tardado cuatro veces más.

```
TASK [backend : Get instance of Database Account] *****
changed: [localhost]

TASK [backend : Get facts for one account] *****
changed: [localhost]

TASK [backend : Get instance of Registry] *****
changed: [localhost]

TASK [backend : Wait for gets] *****
FAILED - RETRYING: [localhost]: Wait for gets (10 retries left).
ok: [localhost] => (item=940930007664.6150)
ok: [localhost] => (item=986867555115.6171)
ok: [localhost] => (item=101067438053.6191)
ok: [localhost] => (item=207770785108.6211)
```

Figura 46: Resultado de la espera de varias tareas asíncronas (O. Salvador, 2023)

Esta muestra las trazas que imprime el anterior extracto de código. Imprime el estado de cada tarea al acabar (en la imagen, “changed”), y una cuenta atrás de los reintentos restantes. El aprovisionado de Redis es lento, y así se contiene el tiempo de ejecución del resto de tareas en el suyo.

```
TASK [infra-base : Wait for Registry, CosmosDB and Redis] *****
changed: [localhost] => (item=23053290133.3637)
FAILED - RETRYING: [localhost]: Wait for Registry, CosmosDB and Redis (100 retries left).
FAILED - RETRYING: [localhost]: Wait for Registry, CosmosDB and Redis (99 retries left).
FAILED - RETRYING: [localhost]: Wait for Registry, CosmosDB and Redis (98 retries left).
FAILED - RETRYING: [localhost]: Wait for Registry, CosmosDB and Redis (97 retries left).
FAILED - RETRYING: [localhost]: Wait for Registry, CosmosDB and Redis (96 retries left).
FAILED - RETRYING: [localhost]: Wait for Registry, CosmosDB and Redis (95 retries left).
FAILED - RETRYING: [localhost]: Wait for Registry, CosmosDB and Redis (94 retries left).
changed: [localhost] => (item=605755824045.3659)
FAILED - RETRYING: [localhost]: Wait for Registry, CosmosDB and Redis (100 retries left).
FAILED - RETRYING: [localhost]: Wait for Registry, CosmosDB and Redis (99 retries left).
FAILED - RETRYING: [localhost]: Wait for Registry, CosmosDB and Redis (98 retries left).
FAILED - RETRYING: [localhost]: Wait for Registry, CosmosDB and Redis (97 retries left).
```

Figura 47: Espera de resultados de tres tareas asíncronas (O. Salvador, 2023)

La última figura ejemplifica el resultado la ejecución del mismo extracto que la anterior, pero sin una tarea de espera. Ansible devuelve la línea de comando, pero no ha acabado las tareas. Durante el desarrollo, este comportamiento me sorprendió, y es la razón por la que añadiese esta tarea de espera en particular.

```
[user@virtualbox parallel_ansible]$ ps -ef | grep ansible
user      2046      1  0 10:51 ?        00:00:00 /usr/bin/python /home/user/.ansible/tmp/ansible-tmp-16729
98710.3613777-2030-56361038365362/async_wrapper.py 714560051259 1000 /home/user/.ansible/tmp/ansible-tmp-1672
998710.3613777-2030-56361038365362/AnsiballZ_azure_rm_rediscache.py _
user      2047     2046  0 10:51 ?        00:00:00 /usr/bin/python /home/user/.ansible/tmp/ansible-tmp-16729
98710.3613777-2030-56361038365362/async_wrapper.py 714560051259 1000 /home/user/.ansible/tmp/ansible-tmp-1672
998710.3613777-2030-56361038365362/AnsiballZ_azure_rm_rediscache.py _
user      2048     2047  0 10:51 ?        00:00:00 /usr/bin/python /home/user/.ansible/tmp/ansible-tmp-16729
98710.3613777-2030-56361038365362/AnsiballZ_azure_rm_rediscache.py _
user      2153    1183  0 10:59 pts/0    00:00:00 grep --colour=auto ansible
```

Figura 48: Procesos de tareas asíncronas de fondo (O. Salvador, 2023)

Lanzar varias tareas y esperar a sus resultados antes de usarlos es la forma en la que he paralelizado los roles. El primer play que hice, `site.yml`, se limita a lanzar los roles de manera secuencial, mientras que el segundo además aprovecha la estrategia “free”. Esta se aplica a todos los roles que cuelguen de un host en el play, por lo que he creado tres conexiones a localhost, la primera y última crean la infraestructura solo con el paralelismo que tengan sus roles; y la segunda contiene los roles de construcción y subida de las imágenes Docker al mismo tiempo. Ambos plays tienen los mismos componentes, con las mismas dependencias. En el siguiente diagrama, los números señalan fases secuenciales, y los subíndices pasos dentro de estas. Están repetidos en los pasos que he podido implementar como concurrentes. Las flechas representan movimiento de imágenes, y las líneas peticiones de información que resuelvo con tareas “_info”.

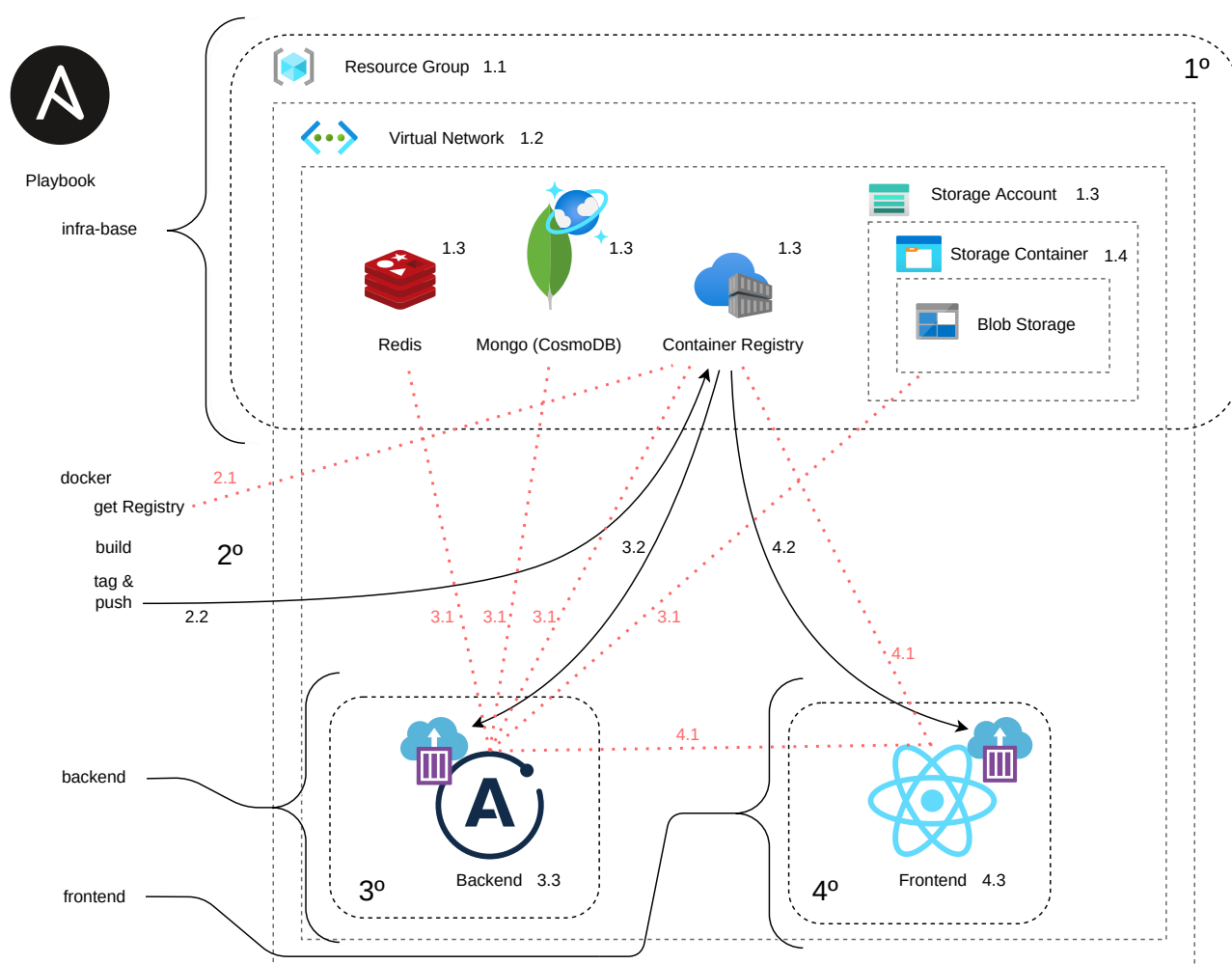


Figura 49: Playbook paralelo, etapas, dependencias y peticiones (O. Salvador, 2023)

El segundo play sin embargo, precisamente por hacer la construcción y subida de las imágenes a la vez, tiene peor rendimiento que el primero (Anexo B). La razón es que, por realizarse en paralelo, ninguno de los procesos se beneficia de haber realizado la ejecución del otro. Como las dos imágenes, el backend y frontend, usan Node, en secuencia se encontrarían las capas de la primera en los dos

procesos de la segunda. Durante la construcción, Docker tiene que bajarse la imagen base que comparten para cada una, ya que ningún rol la encuentra al empezar. Una vez construidas, tampoco se benefician en la subida, ya que el ACR no tiene las capas, y necesita que los dos las suban todas.

Como objetivo añadido durante el desarrollo, este playbook (`parallel_azure`), tiene un rol más de los necesarios para desplegar el proyecto. La finalidad de este es borrar la infraestructura del proyecto. En Ansible, esta funcionalidad y otras, como modificar la configuración de un recurso existente, tienen que ser implementadas por el desarrollador. Solo se diferencia de la tarea de creación del grupo de recursos en un campo a nivel de la tarea, y otros dos dentro del módulo. El primero es un condicional, que depende de una variable. Los otros, `state` y `force_delete_nonempty` le indican a Ansible que borre el grupo y todos los recursos que contiene.

```
24 - name: Destroy a resource group and all it's contents in azure
25   azure_rm_resourcegroup:
26     name: '{{ resource_group_name }}'
27     location: '{{ location }}'
28     force_delete_nonempty: yes
29     state: absent
30     auth_source: cli
31     # when: destroy_all is defined
32     when: destroy_all == "true"
33     register: rg
```

Figura 50: `/vars/main.yml` (O. Salvador, 2023)

En busca de no borrar accidentalmente la infraestructura justo después de crearla, he colocado este rol el primero en los plays. Para no lanzarlo accidentalmente, aunque se ejecute `ansible-playbook` sin excluir la etiqueta “destroy”, si no se incluye la variable a “true”, toma el valor opuesto. Se puede sobrescribir con la opción “`-e 'destroy_all=true'`”. No hacerlo resultará en la tarea siendo saltada como se ve en la figura.

```
TASK [destroy : Check if destroy is enabled] *****
skipping: [localhost]

TASK [destroy : debug] *****
ok: [localhost] => {
  "msg": {
    "changed": false,
    "skip_reason": "Conditional result was False",
    "skipped": true
  }
}
```

Figura 51: Tarea de eliminado saltada (O. Salvador, 2023)

6 Comparación

Esto no ha sido un uso exhaustivo de las herramientas, ambas tienen funcionalidades a las que no he encontrado el uso en este proyecto, o que superaban su ámbito, como filtros de datos en Ansible, las soluciones de plantillas o los test de ambos. Pero sí que han sido unas aproximaciones suficientemente detalladas como para sacar conclusiones.

Antes de comparar sus diferencias recorreré sus puntos en común. El primero es un mito sobre la infraestructura como código, que no cumplen. He oído mencionar que la infraestructura descrita con estas herramientas se puede migrar de una plataforma a otra. Esto no es así, las dos utilizan identificadores y nombres para los recursos que hacen la implementación específica a la plataforma para la que se haga.

Los propios recursos no siempre tienen una traducción directa entre servicios cloud. Por ejemplo, el almacenamiento de objetos en AWS y Azure es similar en propósito, pero su funcionamiento, opciones y por tanto declaraciones IaC son muy diferentes. Dos diferencias son: el contenedor de almacenamiento de Azure necesitando pertenecer a una cuenta, y el S3 pudiendo ser usado como servidor para estáticos de una página web.

Es difícil que estas dos herramientas, u otras similares, ofrezcan una adaptación automática del diseño. Este trabajo tiene que ser cuidadosamente considerado y realizado a mano. Sin embargo, el beneficio de las dos es su abstracción, aprender a usar Ansible o Terraform implica conocer estas, en lugar de una propia a cada plataforma. Cuando quieras cambiar de esta, o empezar un proyecto nuevo, ya tienes práctica usando la herramienta, su funcionamiento y su documentación.

La documentación está estandarizada para todas las colecciones de Ansible y proveedores de Terraform, lo que hará rápido encontrar el recurso equivalente y volver a desarrollar su código. Gracias a la abstracción y naturaleza declarativa, los componentes de distintas plataformas compartirán muchos campos, pero estos tienen que ser revisados uno a uno, no copiados ciegamente.

Como resultado de esta facilidad, asumo, se ha popularizado la idea de que se haga *casi* automáticamente entre los que conocen su existencia, pero no han usado las herramientas. Otra causa posible es confusión alrededor de lo que un proyecto *multicloud* significa. Ansible y Terraform pueden desplegar recursos de distintas plataformas en un mismo proyecto, sin dificultades adicionales a uno con componentes en solo una. Una vez más, esto no significa una compatibilidad automática de las configuraciones entre ellas.

Las similitudes principales que han quedado evidentes durante este proyecto y expuestas en la memoria, hasta ahora son:

- Usan *Wrappers*, abstracciones para hacer peticiones al API de Azure, igual que su cliente CLI, sus plantillas *ARM* o el *CDK* de AWS para este. En Terraform se distribuyen como un único binario que se descarga solo al inicializar el proyecto. Por el contrario, Ansible distribuye dos por cada tipo de recurso, uno con el que trabajar contra él, y otro con el que recuperar su información, y los junta todos en colecciones. La instalación de estas puede ser complicada. Como he mencionado, tuve problemas con las dependencias de la colección de Azure.
- La sintaxis es muy similar, la pareja siendo declarativas. Indicas lo que quieres, no el cómo conseguirlo, y las herramientas encuentran la forma de hacerlo. Los bloques y tareas tienen campos y etiquetas, para declarar: el wrapper y tipo de recurso en su plataforma; el nombre del recurso; y sus propiedades. También tienen un tipo de acción dedicada a recuperar información de la plataforma, y a guardar variables con los resultados de sus operaciones, “output” en Terraform; “register” y “debug” en Ansible.

Los wrappers de ambos antes o después se han quedado cortos en cosas poco importantes, Terraform no es capaz de recuperar el *connection string* de CosmosDB, y Ansible no devuelve las llaves de Redis sueltas. En las dos he podido solucionarlo, creando el string juntando las llaves y dirección en Terraform, y cortando el de Redis, que era lo único que me devolvía Ansible, para extraer las llaves.

- Ambos soportan reutilización de código entre proyectos, aun si no lo he demostrado en mi uso. Terraform usa módulos, que incluso se ofrecen ya hechos en su registro, y Ansible soporta plantillas de roles con Jinja, con parámetros que se resuelven en tiempo de ejecución.
- Ninguna de las herramientas es propietaria, aun si la filosofía detrás de sus licencias es muy diferente (código abierto frente a código libre). En las dos, cualquiera puede inspeccionarlo y contribuir. También llevan esta mecánica a sus plug-ins.

Aunque Hashicorp ha desarrollado la mayoría de los proveedores de Terraform (incluido el de Azure), su registro es hogar a proveedores de terceros. Los desarrolladores de Ansible han descargado casi la totalidad de este trabajo a las empresas interesadas, incluida la colección de Azure, producida por Microsoft directamente.

Por el contrario, son resultado de planteamientos e historias de desarrollo muy distintos. Terraform es una herramienta para aprovisionar recursos cloud. Sólo hace eso, pero sobresale en ello. Ansible nació como una herramienta de automatización, su extensibilidad y popularidad resultaron en su adaptación a herramienta para IaC. Se repercute en diferencias marcadas:

- Los proyectos de Terraform son un todo. Son evaluados, tratados y usados como tal. Su contrario trata cada acción como una entidad separada, que se puede agrupar con otras en una estructura que puede crecer para realizar más que un solo despliegue en cada playbook. Mi solución en el primero ha sido simular polyrepos, repositorios pequeños y separados para cada proyecto, con carpetas desperdigadas, mientras que Ansible, salvo porque he creado varios playbooks, solo tendría una.
- Ansible es un orquestador de tareas, permite organizarlas y ejecutarlas de muchas maneras, según la necesidad. Pero no tiene una perspectiva global de lo que está haciendo. Terraform construye su grafo de dependencias, y relaciona los componentes, creando los recursos según las dependencias que descubra, sin que el orden de sus declaraciones importe. Por tanto, Ansible no es realmente declarativo, no a nivel de rol ni de playbook, solo realmente al de tarea.
- Terraform abstrae al desarrollador de la implementación de funcionalidades como destruir los componentes; actualizarlos si existen, pero su configuración es distinta a la actualmente declarada; y la paralelización de estas operaciones y la creación. Ansible depende de los desarrolladores, los de los módulos y el final para ofrecer funcionalidades como la idempotencia, que según la colección puede no estar garantizada.

6.1. Rendimiento

He contabilizado varias métricas sobre los proyectos y playbooks para añadir contexto. En los Anexos B y C muestro las cifras y explico como las he conseguido.

Primero, he medido los tiempos de ejecución de cada etapa: cada proyecto de Terraform, y construcción de imágenes; y roles de los playbooks de Ansible; por separado. Las cifras que ofrezco son las medias de cinco ejecuciones de cada uno. Con tan pocas medidas no son necesariamente representativas, y había valores anómalos que no he rechazado.

No he realizado un muestreo más extenso por el coste y tiempo necesarios. Tampoco he calculado estadísticos detallados porque considero que superan el ámbito de la memoria.

Como se puede ver en la siguiente figura, el playbook de Ansible con tareas asíncronas consigue los mejores tiempos. Terraform, sin el esfuerzo añadido que requiere el paralelismo en la otra, le sigue muy cerca. Varios minutos por detrás está, como es de esperar, la solución sin concurrencia. Este es el primer playbook que hice, completamente secuencial. Aunque no lo he añadido en el gráfico, también he medido el play con roles concurrentes, que tiene tiempos mucho peores por las razones que expliqué en *Ansible § Implementación paralelizada*.

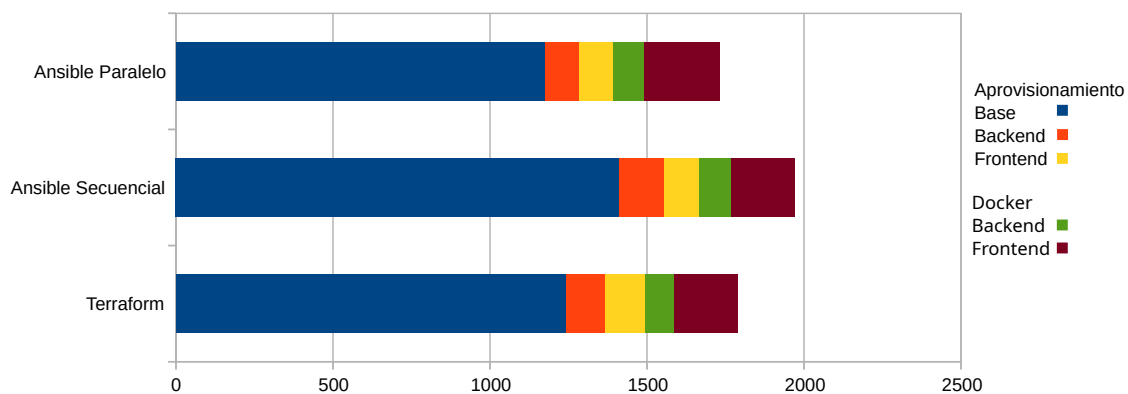


Figura 52: Tiempos de los proyectos en segundos (O. Salvador, 2023)

Después, he medido las líneas de código (excluyendo vacías y comentadas) y caracteres totales de los proyectos y playbooks. He excluido los roles de Docker, ya que Terraform no tiene equivalente y sería injusto. También he excluido el rol “destroy”, que solo está en el playbook paralelo por la misma razón. Cabe destacar que un tercio de las líneas y quinto de los caracteres de Terraform se dedica a los archivos `variables.tfvars`, cuya complejidad es nula en comparación con la que añade implementar paralelismo en Ansible.

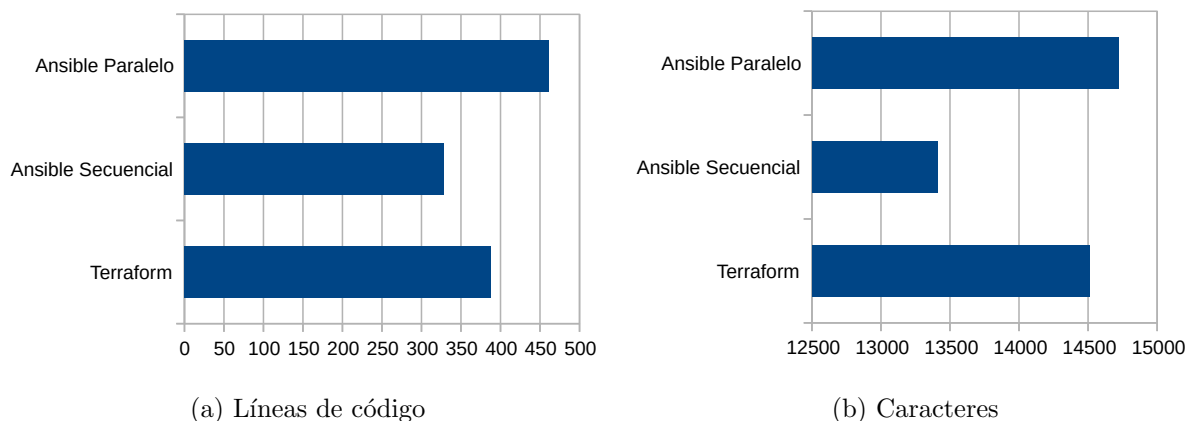


Figura 53: Verbosidad de los proyectos (O. Salvador, 2023)

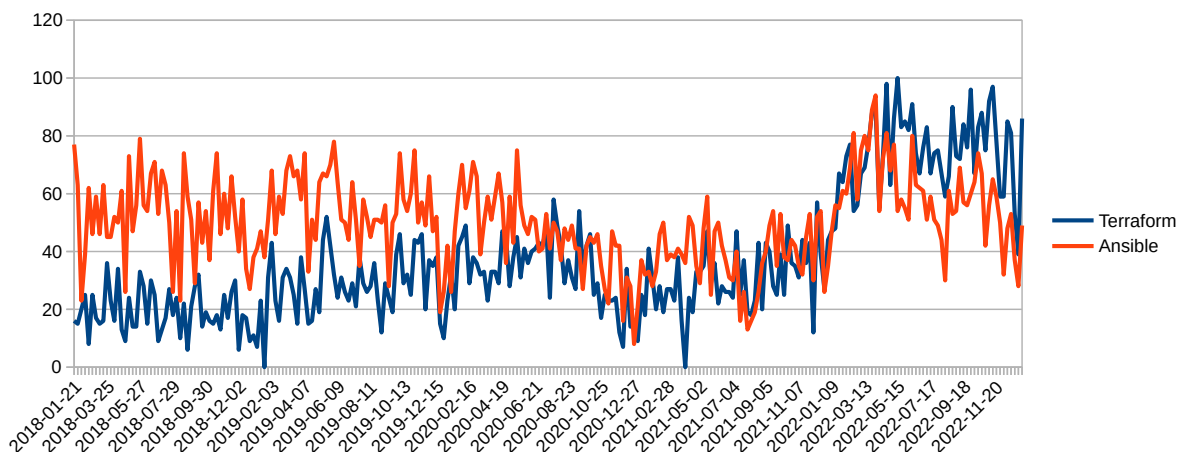
6.2. Relevancia

Apelar al número de usuarios para dictar la superioridad de una herramienta sobre la otra sería construir un argumento falaz. Sin embargo, si que es un punto de contexto relevante, ya que una herramienta más popular probablemente tenga más documentación y artículos, que ayudarán en el desarrollo. En 2022, Terraform y Ansible fueron las sexta y séptima tecnologías mas populares en Stack Overflow, de 46.432 respuestas de desarrolladores profesionales (Stack Overflow, 2022), ocupando un 12,3 % y 9,64 % respectivamente.

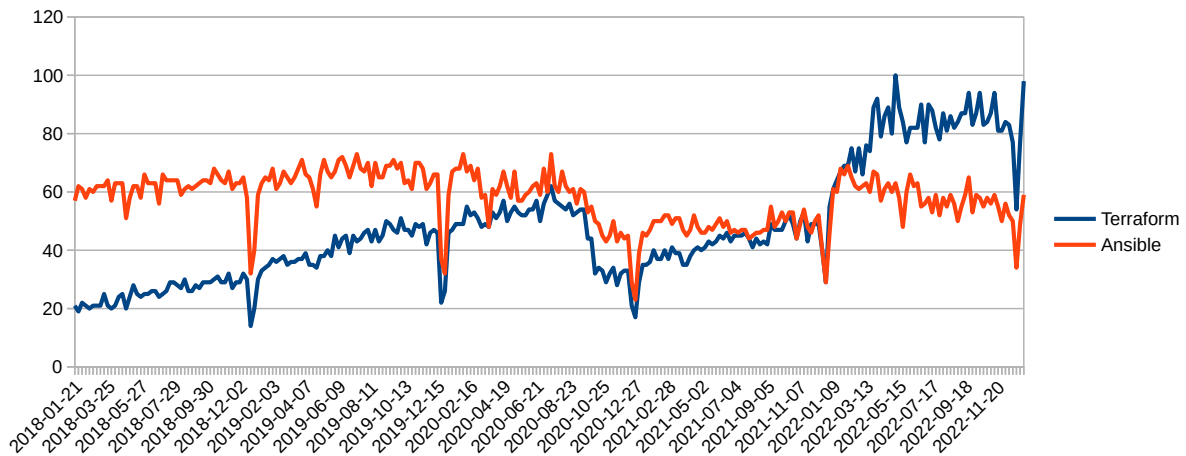


Figura 54: Tecnologías más populares entre profesionales (Stack Overflow, 2022)

Para añadir contexto a estas posiciones, también incluyo las tendencias de búsquedas de ambas (Google Trends, 2022) en los últimos cinco años. En el último, Ansible parece haber estagnado, ligeramente a nivel nacional, y de manera marcada globalmente.



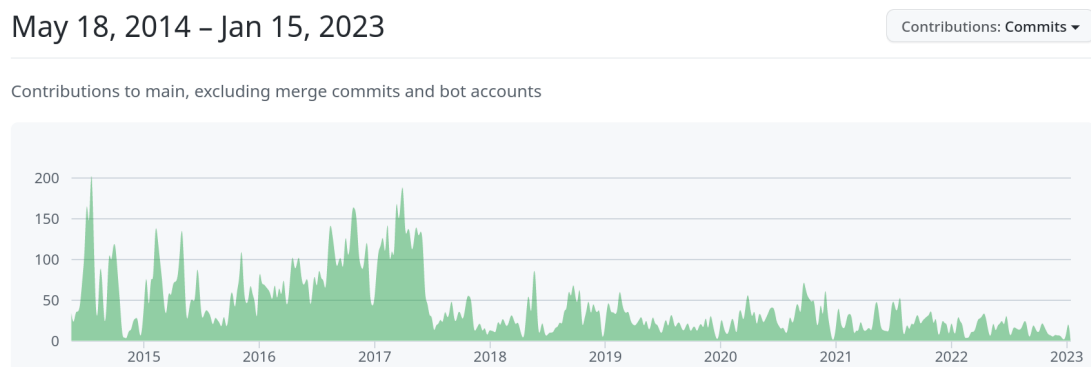
(a) Popularidad en España



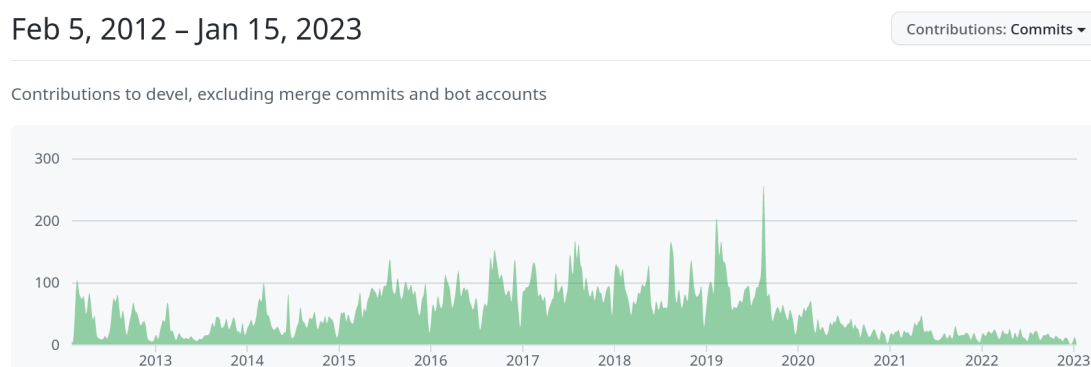
(b) Popularidad mundial

Figura 55: Tendencias de búsqueda (Google Trends, 2023)

Finalmente, gracias a la transparencia de los desarrollos de las dos, podemos ver las frecuencias de commits a sus repositorios (Terraform - GitHub, 2023, Ansible - GitHub, 2023). Ambas han superado su fase de desarrollo inicial, y llevan años con un número de contribuciones estabilizado.



(a) Commits a hashicorp/terraform



(b) Commits a ansible/ansible

Figura 56: Frecuencia de contribuciones a los proyectos (GitHub, 2023)

6.3. Integración de una en la otra

Estas herramientas no son mutuamente excluyentes. Como último ejercicio, me propuse implementar un playbook en el que se use el módulo de Terraform (Ansible, 2022e) y mis proyectos de este para hacer el aprovisionado. Me apoyé sobre el siguiente artículo de Medium para aprender a usarlo y sus detalles: Jain, 2021. La implementación del playbook `terraform_terraform` es muy sencilla, ya que solo se necesita una tarea para aplicar un proyecto, y el rol que la contiene pudiéndose reusar para los tres que tengo en Terraform.

Este playbook ha conseguido los mejores tiempos (Anexo B) para el despliegue de la infraestructura base, mejores incluso que los de Terraform por su cuenta. Esto es sorprendente, ya que debería tener al menos ese tiempo, y encima el añadido del *Overhead* de Ansible. Como posible explicación se me ocurrió que el módulo de Ansible reimplementase partes de Terraform, de manera más optimizada. Aprovechando que estos proyectos son abiertos, he revisado el código del módulo (Fontein et al., 2023). En el he descubierto que su implementación, en Python,

tiene sencillamente una función en la que lanza el comando CLI después de convertir los campos de la tarea a opciones de este. Sí que realiza un plan. Habiendo explorado esta posibilidad y sabiendo que había seguido el mismo método para todas las mediciones, la posibilidad más probable es que no controlase suficientes variables durante el estudio. Aunque hice cinco medidas, realicé las de este playbook días después de los otros, y Azure podría haber tenido menos demanda durante este.

La verbosidad de esta solución es solo ligeramente superior a la de Terraform suelto, ya que deja poco que implementar. El módulo de Terraform incluye todas sus funcionalidades, incluida la actualización de recursos.

También hay módulos de Terraform para trabajar con Ansible (como Cloudposse, 2022), pero no son oficiales ni populares. Aparte, se puede conseguir ejecutando el comando `ansible-playbook` en un recurso aprovisionado (DigitalOcean: Savic, 2021). Ninguna de estas opciones integra Ansible de manera comparable con la inversa. Por tanto, no las he perseguido.

```
25 - name: Deployment of base project
26   community.general.terraform:
27     project_path: '../../{{ targetproject }}'
28     variables_files: 'variables.tfvars'
29     state: present
30     force_init: true
31     register: tfoutput
32
33 - debug:
34   var: tfoutput
```

Figura 57: Tarea de despliegue en `roles/terraform/task/main.yml` (O. Salvador, 2023)

7 Conclusión

Terraform es una mejor herramienta de aprovisionamiento. Es más fácil de hacer funcionar, aprender, rápido desplegando y de desarrollar para las funcionalidades que proporciona. Ansible no tiene, de serie, los lujos que ofrece su alternativa, e implementarlos incurre un coste. Para que un playbook compita con todo lo que trae Terraform, tendrá que ser más verboso y complicado de leer.

Sin embargo, si se me pidiese recomendar una, no necesariamente excluiría a Ansible. Para proyectos multietapa, fuertemente secuenciales, los beneficios de Terraform se pierden. Si un proyecto necesita pasos adicionales a la contratación de recursos, solo Ansible podrá abordarlos sin necesitar apoyarse en herramientas externas. En estos casos, para automatizar los despliegues con pipelines, Terraform estará fuertemente acoplado a la plataforma, siendo un paso en ella, mientras que su contrincante es el pipeline, y se puede mover fácilmente a otra solución de CI/CD.

El bagaje técnico de su origen sobrecomplica Ansible, pero puede conseguir lo mismo que Terraform, en el mismo tiempo, y ambas herramientas se pueden aprender en una semana. En una organización grande, un equipo dedicado podría encargarse de la complejidad que Ansible necesita para conseguir el rendimiento de Terraform, además implementando el paralelismo solo donde el beneficio lo justifique. La misma flexibilidad que permite a Ansible paralelizar donde sea necesario, e integrarse con otras herramientas, le da la opción de incorporar un proyecto de Terraform si la tarea lo supera, imposible a la inversa.

Cuanto mayor sea la infraestructura por aprovisionar, más evidentes serán los beneficios de Terraform, que resolverá las dependencias y optimizará el despliegue automáticamente. Estos son beneficios que, a través de muchos proyectos, y muchos despliegues de cada uno, son favorables a Terraform. Aún así, con el tamaño crecerá la probabilidad de que haya etapas del despliegue que no sea capaz de asumir por su cuenta.

En resumen, la flexibilidad de Ansible lo hace una mejor herramienta para navegar los requisitos de despliegue de una aplicación, sus necesidades y su infraestructura, mientras que Terraform es preferible para proyectos que no escapen de su ámbito, el aprovisionamiento.

8 Lecciones aprendidas y líneas de continuación

Esta práctica ha sido el proyecto mas grande que he realizado en la carrera, hasta ahora. Como tal, ha sufrido problemas que ejercicios más pequeños y guiados no ofrecen la oportunidad de observar.

- **Scope Creep** es más que un riesgo teórico que estudiar en *Gestión de proyectos tecnológicos*, es un problema muy real que se produce naturalmente, y tiene que ser controlado deliberadamente. Un mal estudio de los requisitos de esta memoria me hizo sobreestimar el esfuerzo que se pedía, y añadir funcionalidades para justificar el desarrollo que he realizado. En mi primer planteamiento del problema, no incluía el uso de Redis ni almacenamiento de imágenes, mucho menos las operaciones con ellas desde el cliente. Esta última ha consumido un tercio del tiempo total de la práctica. Gracias a este proyecto, tengo un poco más de experiencia, que usaré para hacer un mejor estudio y sobre todo, estimación de tiempos en los siguientes que encuentre.
- **Riesgo Técnico** es el nombre con el que describo las incógnitas que causa proponer un proyecto con herramientas que no conoces. Estas lagunas sin conocer pueden resultar fáciles de abordar, como Redis que fue trivial de implementar, o prohibitivas, como el uso de imágenes. Cuando se empiezan a acumular las complicaciones, se vuelve más probable que el proyecto acabe en fracaso. En diciembre, este era un prospecto muy real para mí, la integración de los componentes estaba siendo costosa, y no confiaba en que fuese a poder entregar un solo playbook de Ansible.
- **DevOps** no es solo una salida profesional interesante para mí, suple una necesidad muy real de proyectos grandes. Haber afrontado este proyecto de manera iterativa, con las fases de desarrollo que describí en la primera sección, y no desplegar en Azure hasta diciembre ha añadido mucho trabajo innecesario. De haber automatizado los despliegues, y trabajado con Azure desde el principio, el tiempo de integración de las piezas, que ha sido el segundo más prolongado, podría haber sido muy reducido. Esta práctica se beneficiaría, para completar su propósito como maqueta de un sistema real, de un ciclo de vida. Esta sería una de las principales adiciones que añadir a futuro, despliegues automatizados con pipelines de la infraestructura e imágenes.
- **Cifrar** el tráfico tiene que contemplarse desde el principio, y no es fácil. Aunque no era parte de mis objetivos, el tráfico cifrado del cliente al backend me ha superado. Son las únicas comunicaciones que no están hasta estándar, y lo primero que mejoraría en una revisión.

A Generación gráfico del grafo de dependencias

En su documentación, Hashicorp, s.f.-e, explica como imprimir el grafo de dependencias como texto, y usar `dot` para convertirlo a un gráfico. Como el diagrama que sale por defecto es muy plano, ya que el objetivo de Terraform es encontrar las dependencias y maximizar las operaciones que pueda paralelizar, el grafo es muy plano. A su comando, que solo usa `dot`, le he añadido otra herramienta, `unflatten`, que intenta hacer los grafos más verticales. Las opciones “-l” y “-f” del comando `unflatten` indican la cantidad de niveles o filas, y la reorganización de las flechas respectivamente. La opción “-Tpdf” de `dot` es el formato de salida, PDF.

```
terraform graph | unflatten -l 45 -f | dot -Tpdf >base-graph.pdf
```

Figura 58: Comando para generar diagramas de los grafos de Terraform (O. Salvador, 2022)

B Tiempos de aprovisionado de las herramientas

Estos tiempos son la media de cinco ejecuciones por cada herramienta y escenario. He abreviado “Aprovisionado” y “Docker”. “Docker” implica la construcción de la imagen, etiquetado, y subida al registro de Azure. En cada herramienta he construido las imágenes después de borrar todas las que tenía el sistema, para que empezaran desde cero. En el caso de Terraform, la construcción a la que se refiere es manual, haciéndola con los comandos que muestro en la Figura 16 (p. 23). Dicho esto, entre el frontend y backend no las he borrado, y se refleja en el frontend heredando capas y tardando menos. En un caso de uso real, no se borrarían las imágenes entre pasos. Terraform necesita generar un plan antes de aplicarlo, y he optado por generarlo antes de aplicarlo, en lugar de dejar a Terraform hacerlo durante el `apply`. De ahí que sus tiempos estén desglosados. Tardaría menos como un único paso, pero sería mala praxis. Los tiempos de Docker en Ansible Terraform son los mismos que en secuencial.

Herramienta	A. base	A. Backend	A. Frontend	D. backend	D. frontend
Terraform	15s + 20m 28s	17s + 1m 48s	16s + 1m 50s	1m 33s	3m 21s
Ansible Secuencial	23m 31s	2m 23s	1m 52s	1m 41s	3m 42s
Ansible Paralelo	19m 36s	1m 47s	1m 50s	1m 38s	3m 59s
y docker paralelo	-	-	-	6m 14s	
Ansible Terraform	19m 12s	2m 11s	2m 13s	-	-

Tabla 1: Tiempos medios de aprovisionado

C Verbosidad de los proyectos y playbooks

He conseguido estas cifras usando tres herramientas de CLI: `cloc`, `wc` y `find`. La primera cuenta las líneas de código, excluyendo comentarios y líneas en blanco. La segunda, con la opción “-c”, cuenta el número de caracteres en un fichero. Con la tercera elijo los ficheros que alimentar a la anterior.

Fichero(s)	Métrica	base	backend	frontend
variables.tf	Líneas de código	36	36	33
	Caracteres	1324	1400	1007
Resto	Líneas de código	112	92	79
	Caracteres	3906	3977	2898
Total	Líneas de código	148	128	112
	Caracteres	5230	5377	3905

Tabla 2: Verbosidad de los proyectos de Terraform

En el playbook `parallel.ansible` uso dos plays, que no he contado en el total. He contado la play serial, que tiene 21 líneas de código y 505 caracteres. La que usa la estrategia free tiene 46 y 745. Tampoco he incluido en ninguno el rol de Docker, al que Terraform no tiene equivalente. De la misma manera, excluyo de los resultados del rol “destroy” del playbook paralelo, ya que solo está ahí. Una tercera parte de las líneas y quinta de los caracteres de Terraform son los ficheros `variables.tfvars`

Herramienta/Playbook	Líneas de código	Caracteres
Terraform	388	14512
Ansible Secuencial	329	13411
Ansible Paralelo	461	14719
Terraform en Ansible	44+388	3511+14512

Tabla 3: Verbosidad de Terraform y Ansible

Bibliografía

- Ansible. (2022a). *free – Executes tasks without waiting for all hosts*. <https://docs.ansible.com/ansible/2.9/plugins/strategy/free.html#free-strategy>
- Ansible. (2022b). *Ansible concepts*. https://docs.ansible.com/ansible/latest/getting_started/basic_concepts.html
- Ansible. (2022c). *Asynchronous actions and polling*. https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_async.html
- Ansible. (2022d). *Collection Index*. <https://docs.ansible.com/ansible/latest/collections/index.html>
- Ansible. (2022e). *community.general.terraform module – Manages a Terraform deployment (and plans)*. https://docs.ansible.com/ansible/latest/collections/community/general/terraform_module.html
- Ansible. (2022f). *Controlling playbook execution: strategies and more § Setting the number of forks*. https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_strategies.html#setting-the-number-of-forks
- Ansible. (2022g). *Roles*. https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_reuse_roles.html
- Ansible. (2022h). *Using Variables*. https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_variables.html#accessing-information-about-other-hosts-with-magic-variables
- Ansible. (2023). *GitHub - ansible/ansible: Ansible is a radically simple IT automation platform*. <https://github.com/ansible/ansible>
- Ansible Core Team. (2022). *ansible.builtin.gather_facts module – Gathers facts about remote hosts*. https://docs.ansible.com/ansible/latest/collections/ansible/builtin/gather_facts_module.html
- Ansible Core Team & DeHaan, M. (2022). *ansible.builtin.setup module – Gathers facts about remote hosts*. https://docs.ansible.com/ansible/latest/collections/ansible/builtin/setup_module.html#ansible-collections-ansible-builtin-setup-module
- Ansible - GitHub. (2023). *Contributors to ansible/ansible*. <https://github.com/ansible/ansible/graphs/contributors>
- Bachina, B. (2021). *Dockerizing React App With NodeJS Backend — Typescript Version*. <https://medium.com/bb-tutorials-and-thoughts/dockerizing-react-app-with-nodejs-backend-typescript-version-55a40389b0ac>
- Baig, Z. (2018). *How to Fix Redis CLI Error Connection Reset by Peer*. <https://datanextsolutions.com/blog/how-to-fix-redis-cli-error-connection-reset-by-peer/>
- Bernadim, M. (2019). *terraform resource doesn't exist when using modules and data resource 7014*. <https://github.com/hashicorp/terraform-provider-aws/issues/7014>

- Cloudposse. (2022). *GitHub - cloudposse/terraform-null-ansible: Terraform Module to run ansible playbooks*. <https://github.com/ansible-collections/community.general/blob/main/plugins/modules/terraform.py>
- Cloudstack. (s.f.). *CloudStack Provider*. <https://registry.terraform.io/providers/cloudstack/cloudstack/latest/docs>
- DigitalOcean: Savic. (2021). *How To Use Ansible with Terraform for Configuration Management*. <https://www.digitalocean.com/community/tutorials/how-to-use-ansible-with-terraform-for-configuration-management>
- Docker. (2022). *Best practices for writing Dockerfiles*. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
- Fontein, F., Coca, B., & Podobnik, T. J. (2023). *community.general/terraform.py at main · ansible-collections/community.general*. <https://github.com/ansible-collections/community.general/blob/main/plugins/modules/terraform.py>
- Google Trends. (2022). *Comparación de Terraform y Ansible en los últimos 5 años*. <https://trends.google.es/trends/explore?date=today%205-y&q=Terraform,Ansible>
- Hashicorp. (s.f.-a). *Azure Provider*. <https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs>
- Hashicorp. (s.f.-b). *Command: plan*. [https://developer.hashicorp.com/terraform/cli/commands/plan\(v1.3.x\)](https://developer.hashicorp.com/terraform/cli/commands/plan(v1.3.x))
- Hashicorp. (s.f.-c). *How Terraform Works With Plugins*. <https://developer.hashicorp.com/terraform/plugin/how-terraform-works> (accedido: 2022-12-30)
- Hashicorp. (s.f.-d). *Import*. [https://developer.hashicorp.com/terraform/cli/import\(v1.3.x\)](https://developer.hashicorp.com/terraform/cli/import(v1.3.x))
- Hashicorp. (s.f.-e). *Resource Graph*. [https://developer.hashicorp.com/terraform/internals/graph\(v1.3.x\)](https://developer.hashicorp.com/terraform/internals/graph(v1.3.x))
- Hashicorp. (s.f.-f). *State*. [https://developer.hashicorp.com/terraform/language/state\(v1.3.x\)](https://developer.hashicorp.com/terraform/language/state(v1.3.x))
- Hashicorp. (s.f.-g). *Terraform Language Documentation*. [https://developer.hashicorp.com/terraform/language\(v1.3.x,commit2022-10-20\)](https://developer.hashicorp.com/terraform/language(v1.3.x,commit2022-10-20))
- Hashicorp. (s.f.-h). *Terraform Provider for VMware vSphere*. <https://registry.terraform.io/providers/hashicorp/vsphere/latest/docs>
- Hashicorp. (s.f.-i). *Use Refresh-Only Mode to Sync Terraform State*. <https://developer.hashicorp.com/terraform/tutorials/state/refresh> (accedido: 2022-12-29)
- Jain, I. (2021). <https://medium.com/geekculture/the-most-simplified-integration-of-ansible-and-terraform-49f130b9fc8>. <https://medium.com/geekculture/the-most-simplified-integration-of-ansible-and-terraform-49f130b9fc8>
- Leal, H. (s.f.). *Connecting to a Redis database using Node.js — Northflank*. <https://northflank.com/guides/connecting-to-a-redis-database-using-node-js> (accedido: 2022-12-28)

- Microsoft. (2022). *Azure.Azcollection*. <https://docs.ansible.com/ansible/latest/collections/azure/azcollection/index.html>
- Microsoft: Jewell, P., Berry, D., Myers, T., & Estabrook, N. (2022). *Delete and restore a blob in your Azure Storage account using the JavaScript client library*. <https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blob-delete-javascript#delete-a-blob>
- Microsoft: Jewell, P., Berry, D., Myers, T., McClister, C., Estabrook, N., Buck, A., & Barnett, J. (2022). *Get started with Azure Blob Storage and JavaScript*. <https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blob-javascript-get-started?tabs=azure-ad>
- Microsoft: Jewell, P., Myers, T., Berry, D., & Estabrook, N. (2022). *Create and use account SAS tokens with Azure Blob Storage and JavaScript*. <https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blob-account-delegation-sas-create-javascript?tabs=blob-service-client>
- Microsoft: Jewell, P., Myers, T., Estabrook, N., & Berry, D. (2022). *Get URL for container or blob in Azure Storage using the JavaScript client library*. <https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blob-get-url-javascript>
- Microsoft: Myers, T., Howell, J., & Shahan, R. (2021). *Cross-Origin Resource Sharing (CORS) support for Azure Storage*. <https://learn.microsoft.com/en-us/rest/api/storageservices/cross-origin-resource-sharing--cors--support-for-the-azure-storage-services>
- Microsoft: Myers, T., Mohan, R., Wells, J., McCullough, S., Jia, J., & yullims. (2022). *Create an account SAS*. <https://learn.microsoft.com/en-us/rest/api/storageservices/create-account-sas>
- Microsoft: Zhu, E., Jurek, D., Zhu, S., & Beddall, S. (2022). *Azure Storage Blob client library for JavaScript - version 12.12.0*. <https://learn.microsoft.com/en-us/javascript/api/overview/azure/storage-blob-readme?view=azure-node-latest>
- MongoDB. (2018). *Server Side Public Licence*. <https://www.mongodb.com/licensing/server-side-public-license>
- Nguyen, D., & Tran, T. (2022). *npm: react-images-uploading*. <https://www.npmjs.com/package/react-images-uploading>
- realsnick. (2016). *creating azure rm storage container fails after creating azure rm storage 7005*. <https://github.com/hashicorp/terraform/issues/7005>
- Saini, K. (2021). *An introduction to Ansible facts*. <https://www.redhat.com/sysadmin/playing-ansible-facts>
- Saini, S. (2022). *Dockerfile for React and Typescript*. <https://www.saasbase.dev/dockerfile-for-react-and-typescript/>
- Salvador, Ó. (2022a). *Programación de Interfaces Web, práctica 4*. <https://github.com/oscarsalvador/NEB-frontend-p4>
- Salvador, Ó. (2022b). *Programación de Interfaces Web, práctica 5*. <https://github.com/oscarsalvador/NEB-frontend-p5>

- Singh, K. P. (2021). *Dockerize your React app*. <https://dev.to/karanpratapsingh/dockerize-your-react-app-4j2e>
- Stack Overflow. (2022). *Developer Survey 2022*. <https://survey.stackoverflow.co/2022/#most-popular-technologies-tools-tech-prof>
- Terraform - GitHub. (2023). *Contributors to hashicorp/terraform*. <https://github.com/hashicorp/terraform/graphs/contributors>
- terraform-provider-openstack. (s.f.). *OpenStack Provider*. <https://registry.terraform.io/providers/terraform-provider-openstack/openstack/latest/docs>
- Wellington, R. (2021). *StorageAccountNotFound while creating Storage Account 12052*. <https://github.com/hashicorp/terraform-provider-azurerm/issues/12052>
- Wenzin, M. (2018). *How to run Ansible tasks in parallel*. <https://blog.crisp.se/2018/01/27/maxwenzin/how-to-run-ansible-tasks-in-parallel>