UNIVERSIDAD
NEBRIJA

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

UNIVERSIDAD NEBRIJA

GRADO EN INGENIERÍA INFORMÁTICA

MEMORIA PRÁCTICAS EN EMPRESA

Óscar Salvador Sotoca

Enero/2023

UNIVERSIDAD
NEBRIJA

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

UNIVERSIDAD NEBRIJA

GRADO EN INGENIERÍA INFORMÁTICA

MEMORIA PRÁCTICAS EN EMPRESA

Óscar Salvador Sotoca

Enero/2023

Tutor académico: Carlos Castellanos Manzaneque

Índice

Índice de figuras

1 Experiencia en empresa

tbc

1.1. Introduccion, mis responsabilidades

1.2. Onboarding

1.3. Organigrama

2 Proyecto

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

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

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

2.1. Antecedentes

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

2.2. Estudio del problema

He descompuesto el problema en tres fases de trabajo:

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

2.3. Objetivos

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

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

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

3 Aplicación

Propongo una aplicación web para subir fotos con comentarios como caso de uso. Basada en los contenidos de la asignatura de *Programación de interfaces web*. Parte de las bases de las prácticas cuatro (full stack, CRUD en REST, **misgit1**) y cinco (solo front-end, GraphQL, **misgit2**); y un ejercicio de clase no publicado (full stack, GraphQL con tokens de verificación), todas en React. Pero compone un esfuerzo propio, estando formada por piezas nuevas, investigadas para este proyecto, y un desarrollo propio desde el principio,

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

Las diferencias principales frente a los antes mencionados anteriores son:

- Redis: el uso de una base de datos llave-valor, que solo guarda los tokens en memoria. Frente a tener esta funcionalidad en la propia base de datos persistente (MongoDB).
- Hospedaje de imágenes: almacenamiento usando una solución de externa, originalmente AWS S3, más tarde Azure Storage Container.
- Reverse Proxy: durante el desarrollo en local, para evitar problemas de CORS. Desde entonces, ha demostrado ser innecesario.
- Proveedor cloud: este será el primer proyecto en el que no desarrollo 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 maquina sobre la que correr el componente.

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

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

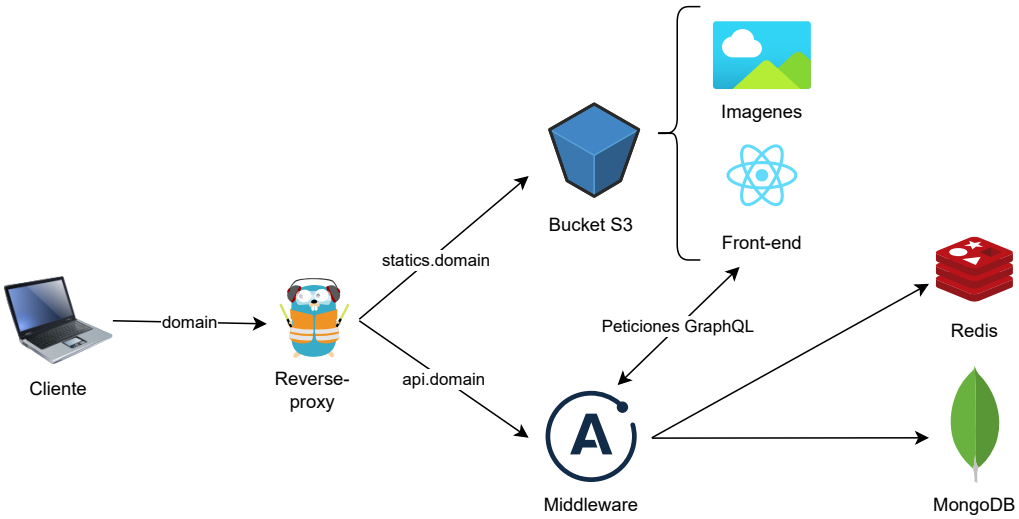


Figura 1: Diseño original a alto nivel (O. Salvador, 2022)

Las necesidades de las piezas principales de la aplicación han dictado la infraestructura que he implementado, iterando sobre el. 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 front-end un FQDN.

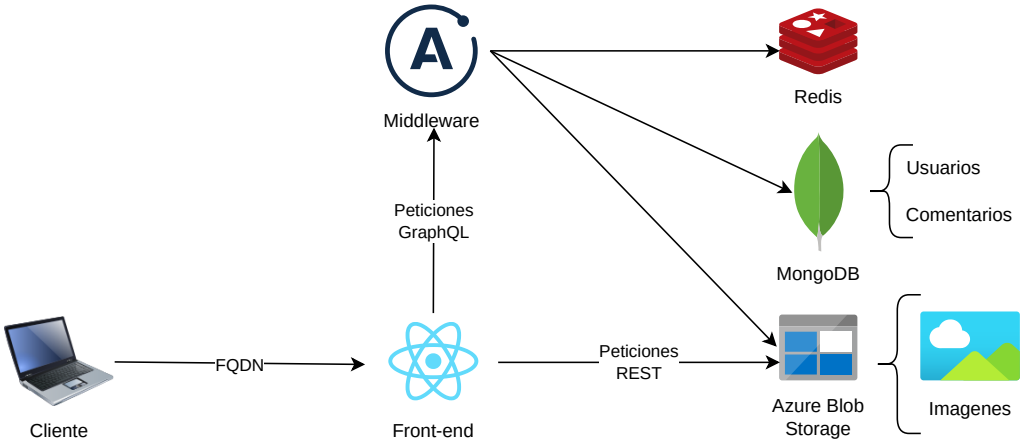


Figura 2: Diseño revisado (O. Salvador, 2022)

3.1. Acceso a la página

He escrito el front-end en TypeScript, lo que hace necesario compilar los fuentes (`.tsx`) a JavaScript. Entrare en detalle sobre como 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 ahi, 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* para enlazar con la base de datos Mongo. En esta petición no es necesario que el usuario este autenticado, no tendría sentido pedirle los credenciales tan pronto.

La lista de *posts* contiene la direcciones de la imágen y autor de cada uno. He configurado el almacen para permitir lectura publica. El cliente renderiza cada comentario, recuperando una a una las imagenes, y, si el usuario estuviese autenticado, mostrandole la opción de borrar aquellos de los que sea autor.

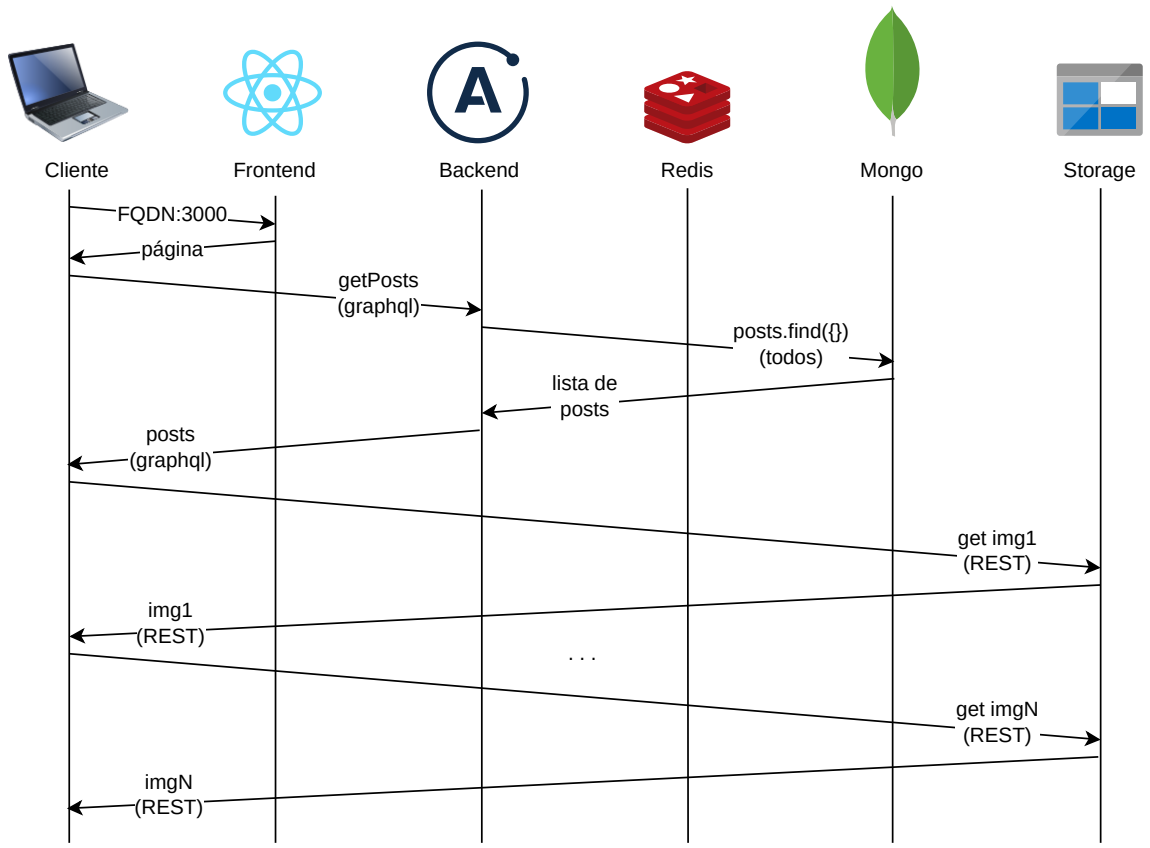


Figura 3: Paso de mensajes `getPosts()` (O. Salvador, 2022)

3.2. Registro e inicio de sesión

Con la pagina renderizada el usuario puede ver una opción en la esquina superior derecha, “Login”. Hacer click 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, 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 carácteres 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 eliminen 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 a continuación se guarda como cookie en el navegador.

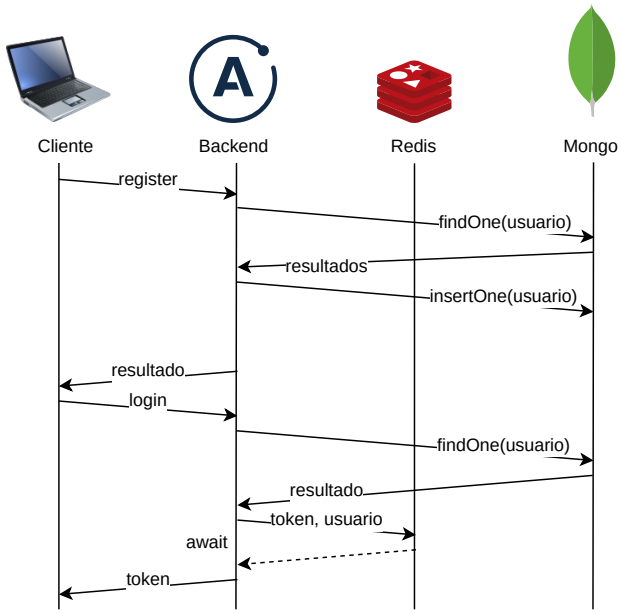


Figura 5: Peticiones `register()` y consecuente `login()` (O. Salvador, 2022)

En la entrega parcial tenía como objetivo tunelizar el trafico del cliente al backend. Al pasar el trafico 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 interceptables

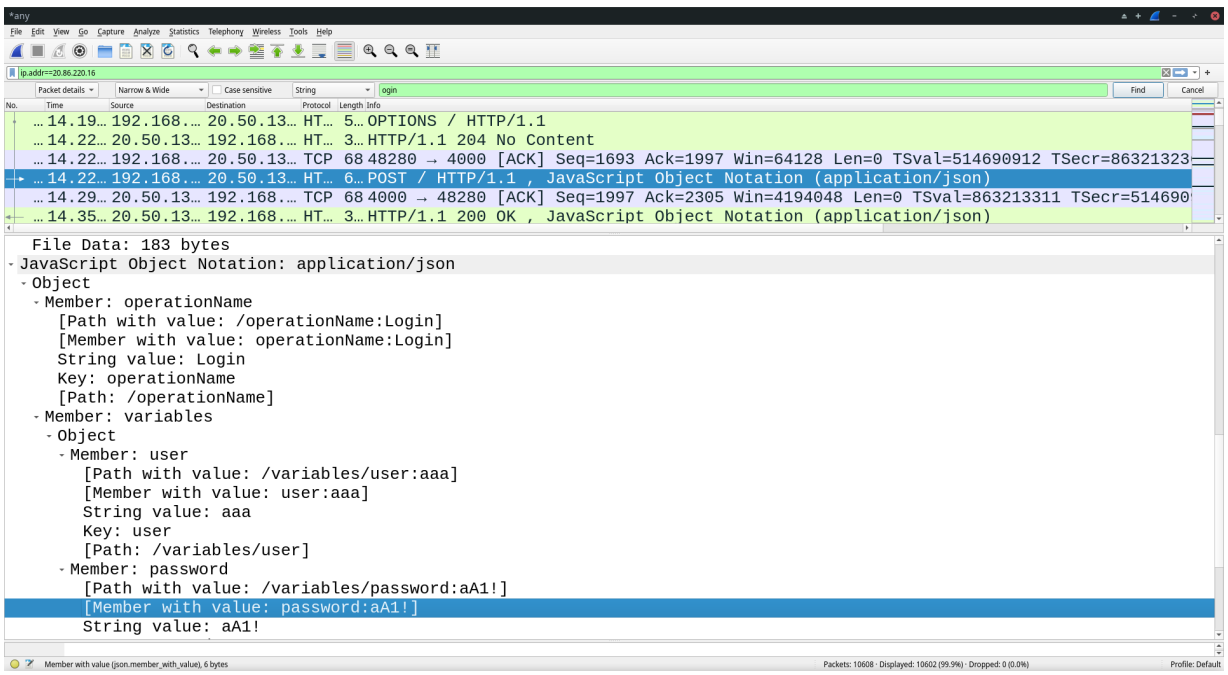


Figura 6: Cabeceras en la petición del navegador (O. Salvador, 2022)

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 su propio cliente. Durante la integración de Redis en el backend usé el siguiente articulo de Humberto Leal **rediscode**.

```
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)

3.3. 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 el 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* (**react_imgs**). Mi implementación se basa en su código de referencia, pero considerablemente editado, ya que mi uso es mas limitado que el que demuestran en el.

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 frontend, que se conecta al almacen y sube la imagen directamente.

El almacen 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 frontend presenta un riesgo de seguridad. Hacerlo en el backend, donde podrían estar con menos riesgo, presenta dos: además del desafío que es subir la imagen a Azure, subirla al backend primero; y el trafico añadido, ya que ahora el backend tiene primero que bajarse la imagen y luego subirla. Mitigo completamente el problema de seguridad no dándole al frontend mas 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 de Mirosoft, en particular: **ms_add_blob1**, **ms_add_blob2**, **ms_add_blob3**, **ms_add_blob4**, y **ms_add_blob5**.

```
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 frontend, subir la imagen es una simple petición REST, usando `fetch()` para hacer un `PUT` de la imagen que el usuario ha subido al cliente a ella.

3.4. Eliminación de "*Posts*"

Cuando el usuario (autenticado) tenga uno o mas *posts* a su nombre, al renderizar la página, el cliente los marcará con la opción de eliminarlos. Si el usuario hace click 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 frontend, se hace dos veces. Antes de borrar la entrada en la base de datos, borro la imagen del almacen. Aunque me costó más aprender a subir las imágenes, borrarlas también ha sido dificil. El articulo que mas me ha ayudado ha sido **ms_rm_blob**, igual que los otros, de la documentación de Microsoft.

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

3.5. 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.

4 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. 7), 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 lo necesitan.
3. **Cuenta de almacenamiento** y **contenedor de almacenamiento** donde hospedar las imágenes con contenedores de "blobs".
4. **Registro de contenedores** al que subir las imagenes del frontend y backend.
5. **CosmoDB**, 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.

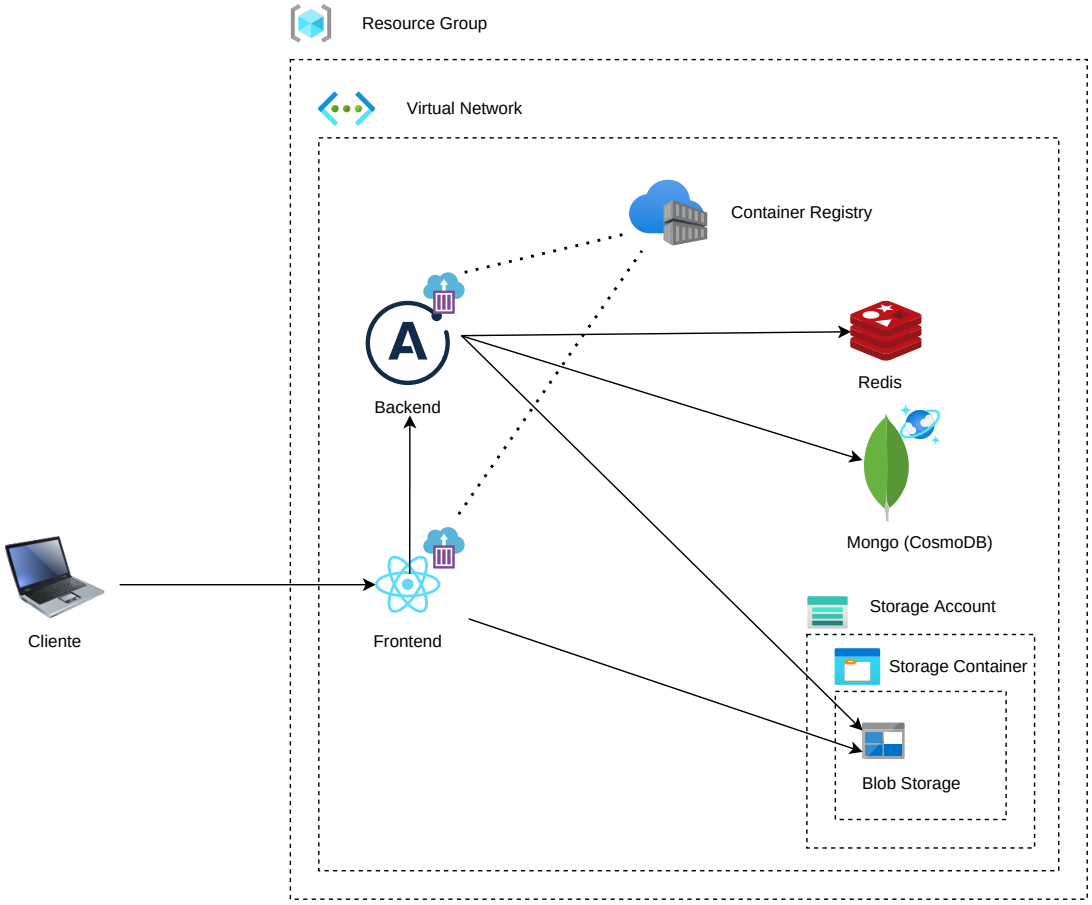


Figura 10: Diagrama de infraestructura (O. Salvador, 2022)

4.1. 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, con ambas suplidas por contenedores, con las imagenes 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, detallare 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*) (**mongo_sspl**). 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 enteridad 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 CosmoDB, 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 de mi 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  const mongourl = process.env.MONGO_URL;

54    const client = new MongoClient(mongourl);

67      const db = client.db("test");

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

(a) `index.js`

```
57  removePost: async (parent, args, ctx) => {

59    const {db, user, token, redisClient} = ctx;

68    found = await
      ↪  db.collection("posts").findOne({_id:
      ↪  ObjectId(postid)});
```

(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 trafico tunelizado, ver Figura 7 (p. 11).

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 satifací durante el desarrollo en local con el interfaz provisto por Mongo en la imagen `mongo-express:latest`.

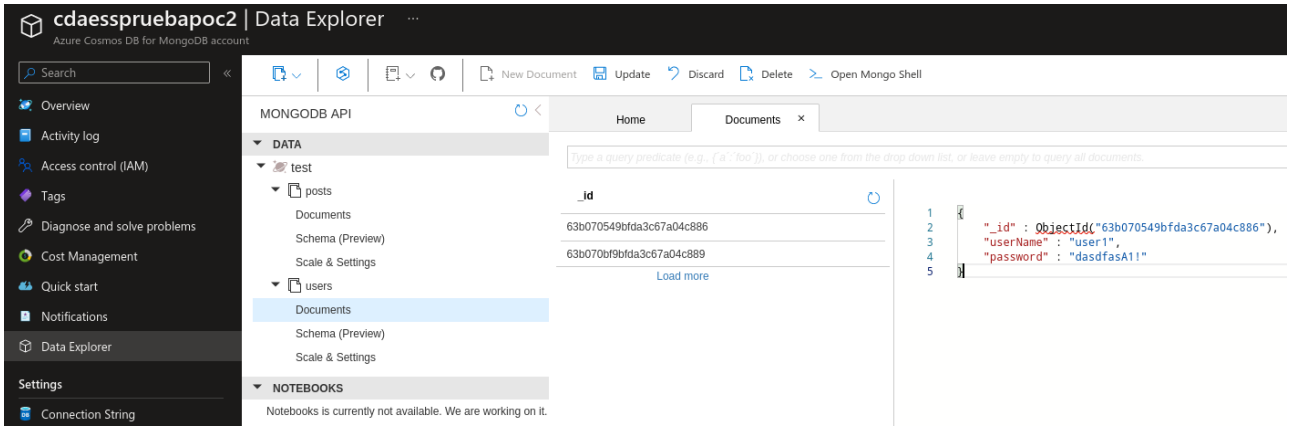


Figura 12: Azure Data Explorer, colección “users” (O. Salvador, 2022)

En el caso de Redis es una linea de comandos, accesible desde el portal de Azure, dentro del recurso, con la opción “Console”. 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 linea 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 tunel y conectarse de manera segura, **redis_connect**, aún si no lo he hecho para esta práctica.

```
root@5f5cda6ef50a:/# redis-cli -h rdbesspruebapoc2.redis.cache.windows.net -p 6379 -a g6XGqZftgHaCOSlu6kcLtDV7JJJOa4yIIAzCaGayGzc=
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) "m6vay3yOque"
2) "9rcctl9x7l"
3) "svf8kahovt"
```

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

Las bases de datos son los elementos mas pesados de levantar, pero curiosamente Redis tarda mas que CosmoDB. 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 maquina virtual que utilicen. Aparte del motor de virutalización que usen, 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, casi siempre 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)

4.2. 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 (APA). 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 imagenes de `hub.docker.com`, pero he preferido crear mi propio registro dentro del proyecto y subir ahi mis imagenes. 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
```

Comandos para subir la imágen del frontend

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 Nginx[1]. 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 podido responder el por que 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)

[1] El `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: **docker_nginx1**, **docker_nginxi2** y **docker_nginx3**

Docker esta orientado a capas. En mi opinión, su sistema de ficheros (AuFS) orientado a capas es su valor principal, por encima del uso directo del kernel del huesped, frente a las máquinas virtuales tradicionales. En mis `Dockefile` uso generosamente las instrucciones "`RUN`" y "`COPY`", cadad 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_layers**), y mis imagines 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. Tambien 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 imágen del frontend después de haber subido el backend.

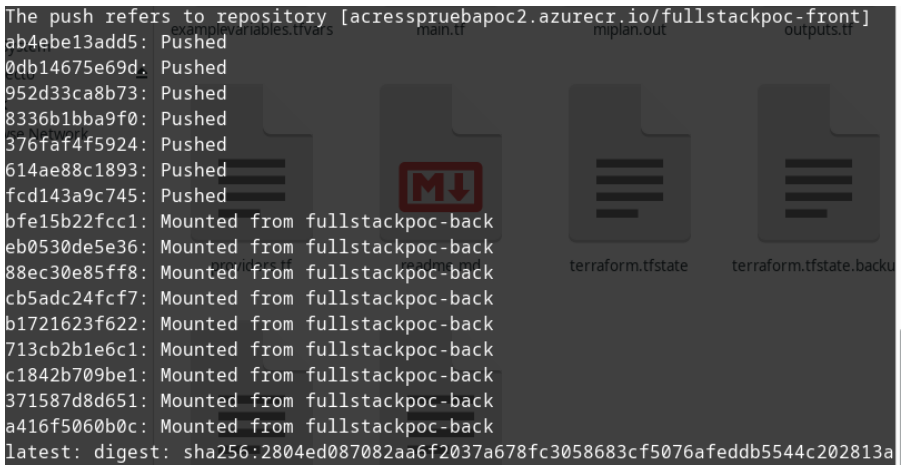


Figura 18: Azure Container Registry compartiendo capas entre imágenes (O. Salvador, 2022)

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; Y tercero el frontend. No se pueden exportar variables en el Dockerfile, Azure no las sobreescribe.

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. 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.

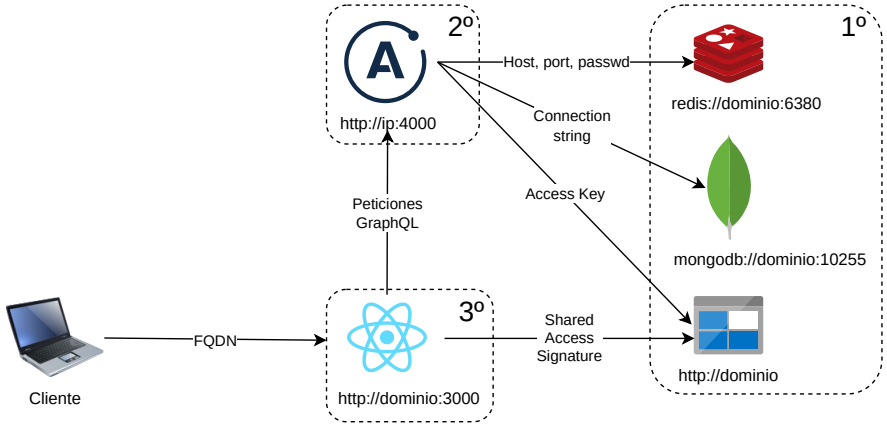


Figura 19: Diagrama a alto nivel, conexiones y orden de aprovisionado (O. Salvador, 2022)

Durante el desarrollo, ApolloServer, la librería que uso para el servidor de GraphQL en el backend, publica 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 ~]$
[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":"sdasdaasdfsdafdasfasd","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":"m6vay3yOque"}}
[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 publicamente 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 Amsterdam.

```
[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 (104.44.1
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)

4.3. Cuenta de almacenamiento

Este componente fue el primero que integré, en Noviemrbe. 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 imagenes de sus almacenamientos ha sido la parte mas difícil del desarrollo, pero AWS tenía mejor soporte y fue mas 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 sobreescribirla. Durante el desarrollo tomaba las URL que genera el backend (no rellenaba a mano la URL que muestro) y probaba a subir imagenes manualmente, usando una vez más, cURL.

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

Comando para subir imagenes

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 (**cors__almacen**). Cuando no lo hacía y no funcionaba saque la conclusión erronea que se ve en las siguientes imágenes[2], que se podía usar el FQDN sin protocolo ni puerto. Tambien 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

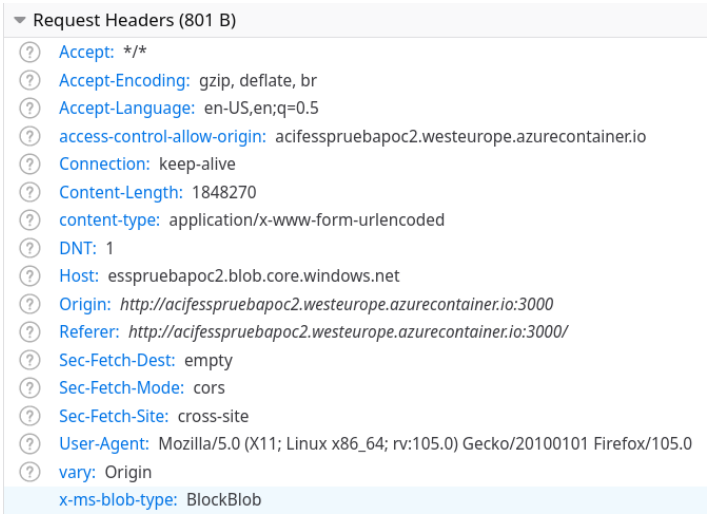
(a) Configuración de CORS de la cuenta de almacenamiento

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

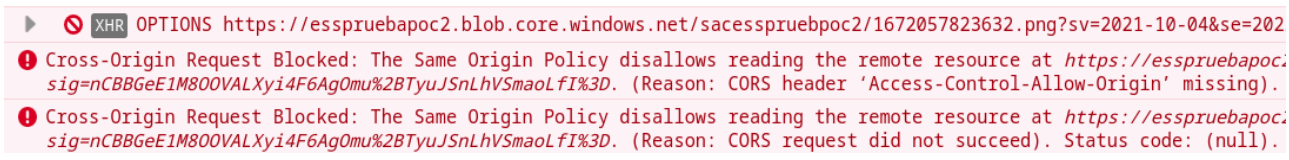
(b) Carga de variables en Terraform, `main.tf`

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

(c) Cabeceras, 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)

[2]Todas 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`

5 Terraform

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

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

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

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

Figura 24: Sintaxis de referencia, documentación de Hashicorp (**hashicorp__lang1**, 2022)

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

5.1. Funcionamiento

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

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

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

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

En el tercero, `terraform plan`, se convierten los archivos de configuraciones a un conjunto de pasos que Terraform puede seguir para alcanzar dicho estado deseado. Ejecutarlo automáticamente lanzará el segundo, `terraform validate`, que comprueba si los fuentes tienen una configuración valida. Si lo es, resuelve las variables. Recorre `variables.tf` y las puebla con `variables.tfvars`. Se puede especificar un fichero de variables con `-var-file variables.tfvars` Si hay mas en el `.tfvars` de las que se usan, Terraform avisará de ello. Si hay menos, y no tienen valor predeterminado son pedidas al usuario por TUI. El usuario puede sobreescribir las variables del fichero para una ejecucion de dos maneras. Puede incluir la opcion `-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_plan**, **hashicorp_plan_refresh**). Con la configuración previa actualizada, calcula las diferencias frente a la propuesta. El último paso para generar un plan es calcular un grafo de dependencias. Terraform hace esto recorriendose los fuentes, viendo los recursos mencionados, y añadiendo un nodo por cada uno. Las parejas de "`<BLOCK LABEL>`" indican dependencia del segundo al primero, los bloques que estén definidos dentro de otros a su padre, y los bloques que usen la etiqueta `depends_on` al que mencionen (**hashicorp_graph**). Después de añadir los nodos al grafo con las dependencias adecuadas, etiqueta a cada uno con meta-datos basándose en las diferencias entre el estado y la configuración, para saber que operaciones tomar con cada nodo.

El grafo se puede generar independientemente con `terraform graph` en cualquier momento, solo para visualizar. El resultado del comando es el texto del grafo, en formato DOT. A continuación muestro un extracto del proyecto mas sencillo, el frontend (detalles en la siguiente sección), es solo una parte para no dedicarle una página[3].

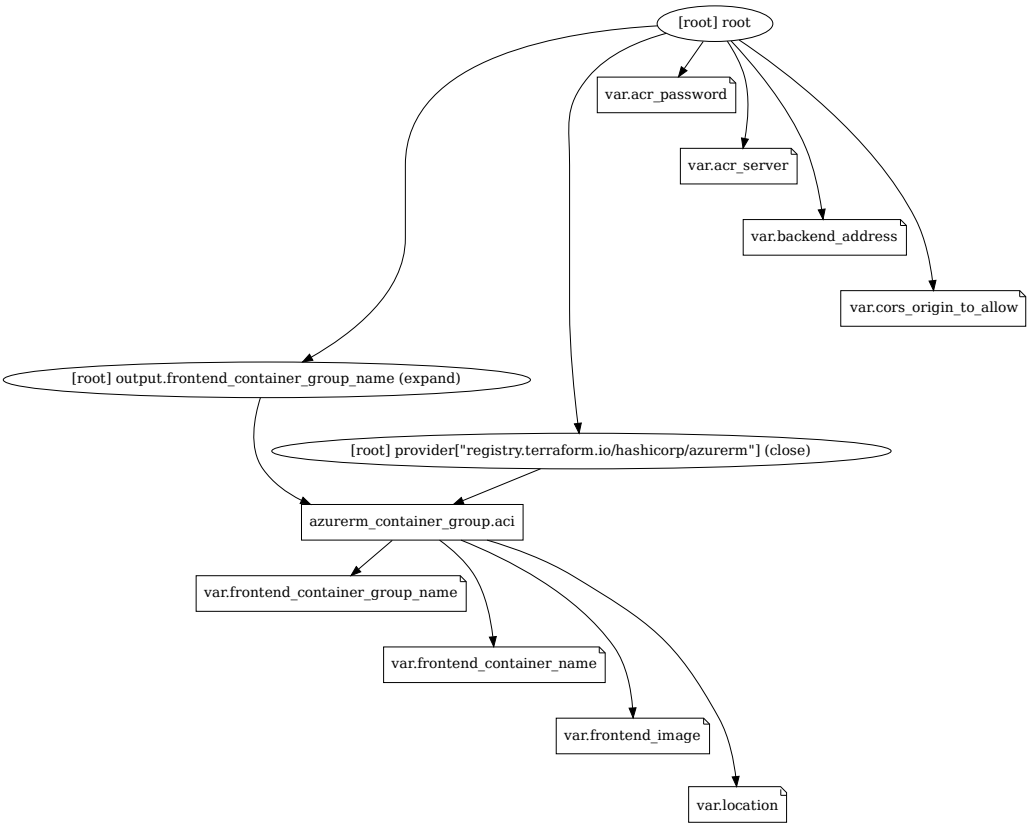


Figura 25: Extracto del grafo del frontend (O. Salvador, 2022)

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

Finalmente, Terraform calcula y muestra la serie de pasos que Terraform 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*) del recurso.

(a) Creación

(b) Destrucción

(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_plan**).

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 (eg. `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.

Para quitar los recursos propuestos en los fuentes de configuración de la plataforma, y dejarla yerma Terraform ofrece dos opciones. El último comando que he presentado es `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 especifica para su plataforma particular. Esta no tiene que ser en la nube, hay proveedores para aprovisionado en plataformas *on-prem*: **onprem_provider1**, **onprem_provider2**, **onprem_provider3**. Permiten separar el desarrollo de proyectos Terraform con configuraciones a alto nivel, uniforme entre plataformas, y los detalles específicos de la implementación de su API correspondiente.

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

- Inicialización de cualquier biblioteca 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_plugins** § 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 mas pasos.

```
terraform {
  required_providers {
    azurerm = {
      source = "hashicorp/azurerm"
      version = "3.0.0"
    }
  }
}
```

(a) Declaración

```
provider "azurerm" {
  features {}

  subscription_id = var.subscription_id
  tenant_id       = var.tenant_id
}
```

(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 usarlos, ya que son publicamente 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.

5.2. Implementación

He explicado el funcionamiento general de Terraform, ahora detallaré como lo he usado yo para generar la infraestructura necesaria para este proyecto. He diseñado la infraestructura al mismo tiempo que aprendía como montarla en Terraform, por el beneficio de poderla eliminar y montar a menudo, reduciendome el coste. Lo reflejaré en esta sección, explicando varios pasos en falso que tomé.

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 la 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.

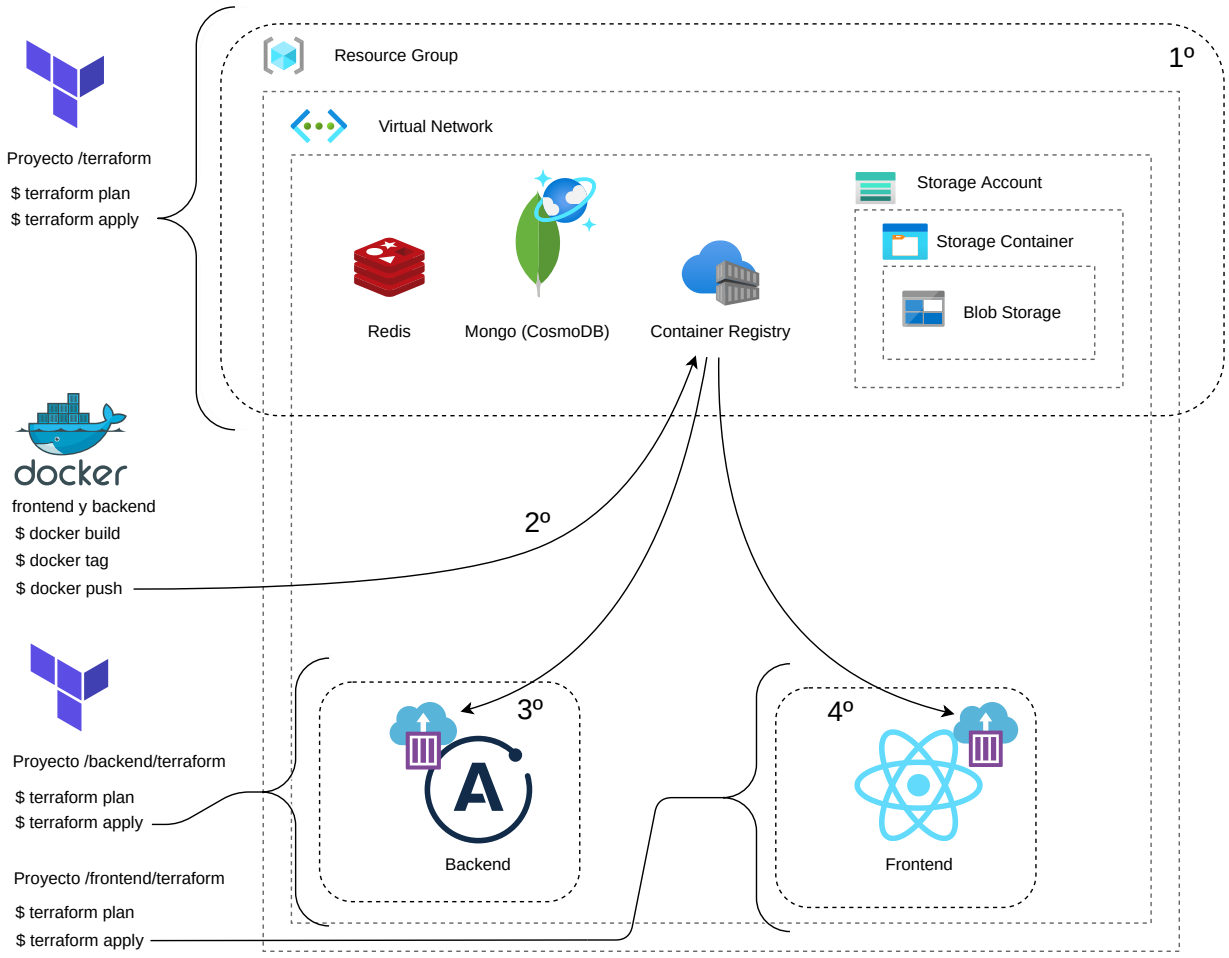


Figura 28: Tres proyectos de Terraform y dos imagenes de Docker (O. Salvador, 2022)

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 del tipo de bloque, son el tipo 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 inequivocamente y se pueden 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 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 estan disponibles por parejas de etiquetas) usando el API de la plataforma. Luego alimento los detalles que necesito de estos recursos a los contendores, y el código

en ellos se los encuentra como variables de entorno, completamente abstraído de como ha 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), actualizandolas automáticamente.

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.
He encontrado dos *issues* cerrados (**tf_depends3**, **tf_depends2**) y uno abierto (**tf_depends1**) con problemas similares. En mi caso lo soluciono depends on

6 Ansible

tbc

7 Comparación

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

8 Conclusión

tbc

9 Lecciones aprendidas

a