Máster en Cloud Apps
Desarrollo y despliegue de aplicaciones en la nube

Curso académico 2021/2022

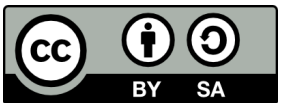
Trabajo de Fin de Máster

Planificador de carreras de montaña

Autores:
Iván Fernández Mena
Pedro A. Carrasco Ponce

Tutor:
Francisco Gortazar

Universidad
Rey Juan Carlos



Índice

Dedicatorias

A mis padres y hermana por estar siempre ahí como soporte para mi ritmo de no parar. A mis amigos, por hacerme desconectar en los momentos difíciles y a Marina por apoyarme en todo momento. A todos, gracias.

- Iván Fernández Mena

Esto no hubiese sido posible sin el apoyo de Eva en todo momento, además quiero dedicar todo este esfuerzo a mi familia.

- Pedro A. Carrasco Ponce

Agradecimientos

A Francisco Gortázar, Micael Gallego y Luis Fernández por sus enseñanzas, guía y paciencia durante todo el desarrollo del máster y a todas las personas que nos han apoyado durante todo este tiempo.

Muchas gracias

Resumen

La CI/CD es una práctica esencial en el desarrollo de software que combina la integración continua y la entrega continua. Consiste en automatizar y agilizar el proceso de desarrollo, integración, prueba y entrega de software, lo que resulta en una entrega más rápida y confiable de aplicaciones a los usuarios finales.
Cada vez más empresas, organizaciones y sectores están implementando estos sistemas en sus desarrollos de software debido a que nos permite detectar errores temprano, acelerar el tiempo de entrega, mejorar la calidad del software y fomentar la colaboración. Si a esto le unimos la capacidad de desplegar y desarrollar una aplicación en un proveedor en la nube como AWS, ofrece ventajas en términos de escalabilidad, disponibilidad, velocidad de aprovisionamiento, gestión simplificada, amplio conjunto de servicios y costos optimizados. Estas ventajas te permiten desarrollar y escalar tu aplicación de manera eficiente, liberando recursos y acelerando la entrega de valor al cliente final.

En este trabajo creamos un caso de uso real simplificado, una plataforma web de gestión de carreras de trail running como núcleo para crear todas las necesidades y propiedades que queremos desarrollar. Creando una versión inicial de una aplicación real se generarán problemas más complejos y completos. El proyecto se separa en tres bloques principales:

- Sección CI/CD en AWS donde se genera una infraestructura necesaria capaz de lanzar la cadena de tests, generar artefactos y desplegar la aplicación si todo se ha completado con éxito.
- Otra sección provee de todos los servicios de cloud en AWS para poder disponer de una aplicación con su backend y frontend capaz de ser tolerante a fallos y distribuir la carga de usuarios finales para que el cliente final tenga la mejor experiencia posible.
- Por último, el propio desarrollo funcional de la aplicación para este caso de uso.

Palabras clave:
AWS, CI/CD, IaC, Trail running, Distribuido, Tolerante a fallos, Terraform, Contenedor

Capítulo 1 Introducción

1.1. Contexto

Actualmente nos encontramos en un mundo donde los desarrollos de software necesitan generar de forma rápida valor a los clientes finales, para recibir una retroalimentación rápida del usuario y así poder ir ajustando el producto continuamente. Se genera un ciclo de retroalimentación de necesidades del cliente y desarrolladores, donde puedes ir creando un producto de forma rápida y efectiva.
Pero generar un entorno que sea capaz de aguantar el cambio continuo y tener la herramientas necesarias para que puedas sacar un proyecto de forma rápida y además hacer despliegues continuamente no es tarea fácil. Paradigmas como el desarrollo en la nube, con plataformas como AWS, Azure o GCP, están permitiendo que de forma rápida puedas obtener un MVP y validar si estás entregando valor a los usuarios. Además ofrecen la capacidad de usar servicios que se monetizan por uso, gastando lo necesario y solo con las peticiones que se realicen desde el cliente.

Con el uso de la nube eres capaz, sólo con el coste estrictamente necesario, de tener un proyecto base y empezar a obtener retroalimentación. A partir de este momento el equipo de desarrolladores consigue reportes de mejoras y de evolutivos que necesitaría hacer. El despliegue de esas mejoras o correcciones puede convertirse en una tarea compleja o muy manual, propensa a errores y tediosa para los desarrolladores, que pueden llegar a emplear muchas horas en conseguir subir su código. Por ello es importante añadir un componente más al rompecabezas, el despliegue continuo e integración continua.

Con el uso de este paradigma en un desarrollo y dentro del contexto explicado conseguimos:

- La detección temprana de errores: Permite identificar y corregir problemas rápidamente, evitando que se acumulen y se conviertan en problemas más graves.
- Mejora de la calidad del código: Al someter cada cambio a pruebas automatizadas, se reduce la posibilidad de introducir errores en el código base y se fomenta el desarrollo de un software más confiable.
- Mayor velocidad de entrega: La entrega automatizada de actualizaciones acelera el tiempo de comercialización, permitiendo que los usuarios se beneficien de nuevas características y correcciones más rápidamente.
- Reducción del riesgo: Al realizar pruebas de manera continua y automatizada, se minimiza el riesgo de errores y fallas en el software en producción.

Con todo esto un equipo es capaz de subir código a un repositorio de código y que de forma automática y segura ese cambio se vea reflejado en el producto.

Si unimos los dos paradigmas antes explicados, el desarrollo en la nube con el CI/CD, tenemos un conjunto de servicios que disponen los proveedores de la nube para estos propósitos y de forma sencilla se pueda generar el entorno necesario para que todas las

ventajas antes expuestas se hagan realidad. Un proveedor nos ofrece estos servicios ya preparados, para que con click en la consola podamos preparar los pipelines de CI/CD.

Actualmente existe un impedimento en la forma en la que se ha generado dicho ecosistema de despliegue y pruebas, para poder preparar todo lo necesario hay que ir a la consola del proveedor en la nube, añadir y configurar cada uno de los servicios uno a uno de forma manual. Esto no es problema a primera instancia ya que, aunque algo tedioso, no debería ser muy complejo. Pero si se desea hacer cambios o borrar parte de un servicio, pueden existir muchos problemas de dependencias que impidan hacer lo que se desea, además del tiempo elevado en realizar los cambios manualmente. Sin contar que si estamos haciendo pruebas, queremos en otro momento desplegar toda la infraestructura tal y como la teníamos previamente, es complicado volver a reproducir todos los pasos. Aquí es donde aparece en escena el paradigma que completa al resto, la Infraestructura como Código (Infrastructure as Code).

Utilizar infraestructura como código (IaC) en conjunto con proveedores en la nube ofrece una serie de ventajas significativas en el desarrollo y despliegue de aplicaciones.

- Automatización y consistencia: IaC permite describir la infraestructura de forma programática utilizando lenguajes de programación o herramientas específicas, lo que facilita la automatización de la creación, configuración y gestión de los recursos de la nube. Esto garantiza que la infraestructura se pueda replicar y desplegar de manera consistente, eliminando posibles errores y desviaciones causadas por configuraciones manuales.
- Control de versiones y trazabilidad: Al utilizar IaC, el código que describe la infraestructura se puede gestionar mediante sistemas de control de versiones como Git. Esto brinda la capacidad de rastrear los cambios realizados a lo largo del tiempo, revertir a versiones anteriores si es necesario y colaborar de manera más eficiente en equipo.
- Escalabilidad y flexibilidad: Los proveedores en la nube ofrecen una amplia variedad de servicios y recursos que se pueden aprovisionar mediante IaC. Esto facilita la escalabilidad de la infraestructura, ya que se pueden agregar o quitar recursos de manera rápida y eficiente según las necesidades del proyecto. Además, IaC permite ajustar fácilmente la configuración de los recursos para adaptarse a diferentes entornos, como desarrollo, pruebas o producción.
- Reproducibilidad del entorno: Al definir la infraestructura como código, se puede crear un entorno de desarrollo completamente reproducible. Esto significa que cualquier miembro del equipo puede crear rápidamente el mismo entorno de desarrollo con todos los recursos y configuraciones necesarios, evitando problemas de compatibilidad o dependencias faltantes.
- Eficiencia y ahorro de costos: Al utilizar IaC con AWS, es posible optimizar la utilización de los recursos y evitar costos innecesarios. Los recursos se pueden aprovisionar bajo demanda y desaprovisionar cuando ya no sean necesarios, lo que ayuda a reducir los gastos y a utilizar los recursos de manera más eficiente.

En conclusión, en el desarrollo de software actual es crucial generar valor rápidamente y obtener retroalimentación constante de los usuarios. Para lograr esto, el uso de paradigmas como el desarrollo en la nube junto con la integración continua y el despliegue continuo es fundamental. Estos enfoques permiten una detección temprana de errores, mejoran la calidad del código, aceleran la entrega de actualizaciones y reducen el riesgo de errores en producción.

Sin embargo, configurar y mantener manualmente la infraestructura necesaria para el CI/CD puede ser complejo y propenso a errores. Aquí es donde entra en juego la infraestructura como código (IaC). IaC ofrece automatización y consistencia al describir la infraestructura de forma programática, lo que facilita la creación, configuración y gestión de los recursos en la nube. Además, permite el control de versiones, la escalabilidad, la flexibilidad y la reproducibilidad del entorno de desarrollo. También ayuda a optimizar el uso de los recursos y a reducir costes.

Al combinar IaC con proveedores en la nube como AWS, se pueden aprovechar los servicios y recursos disponibles para construir y desplegar aplicaciones de manera eficiente. Los proveedores de la nube ofrecen herramientas integradas para crear pipelines de CI/CD, lo que simplifica aún más el proceso.

1.2. Motivación

La principal motivación que nos ha llevado a decidir este proyecto ha sido experimentar. En un principio pensamos definir un punto de partida y montar un prototipo de forma rápida para ir evolucionándolo haciendo iteraciones de desarrollo siguiendo la metodología de Trunk Based Development (TBD) para hacer cambios pequeños, una sola rama de trabajo y que se pongan a producción rápidamente. Haciendo también que fuese fácil arreglar errores si los hubiese, probar nuevas características etc.

Durante el desarrollo del proyecto hemos visto que era más interesante profundizar en el tema de infraestructura, por ello hemos ‘girado’ nuestra idea principal a montar una infraestructura que nos permitiera desplegar de manera fiable y rápida el proyecto. En este momento creemos que el sistema ya estaría preparado para hacer iteraciones siguiendo la metodología TBD, pero ya se sale del alcance del proyecto porque donde hemos profundizado y a lo que hemos dedicado tiempo es a la infraestructura.

Hemos dejado una primera aproximación del proyecto funcional con un backend y un frontend pero como se verá, el trabajo real está en la infraestructura, el sistema de despliegue de código y de integración.

Aún así no sería difícil continuar con el proyecto desde el punto de partida que hemos generado, siempre y cuando se respete la calidad del código añadiendo en todo momento tests automatizados y se itere en pequeños pasos para ir añadiendo nuevas funcionalidades o mejorar las existentes.

1.3. Objetivos

Este proyecto tiene como principal objetivo desarrollar un ecosistema de integración continua y despliegue continuo apoyándose en un proveedor en la nube, en el caso del proyecto AWS (Amazon Web Services) y usando Terraform como software de Infraestructura como Código (IaC).
Nuestra propuesta es la de mostrar las ventajas del uso de IaC con Terraform y servicios de CI/CD con AWS frente a otras alternativas que se han visto a lo largo del máster en ambos casos. Como base lo suficientemente compleja y completa para poder crear la infraestructura necesaria para el proyecto, se usa el caso de uso de un gestor de carreras de montaña para poder tener este MVP mínimo y que la aplicación web pudiera tener el peso que se necesitaba.

El proyecto también trata la complejidad que requiere el correcto desarrollo de una infraestructura con todos los servicios en un aplicación real en la nube, con CI/CD y con IaC. El uso correcto de Terraform para generar la infraestructura y sin olvidar buenas prácticas en código. El trato de las redes, todos los servicios de control a fallos y balanceadores de carga, gestores de bases de datos y todos los servicios que se ven implicados en un desarrollo complejo.

Se quiere mostrar también el uso de los servicios específicos de CI/CD que nos ofrece AWS, explicar sus peculiaridades, ventajas y desventajas vistas en el proyecto. De igual manera con el uso de Terraform frente a otros servicios usados en el máster como CloudFormation propios de AWS o incluso CDK.

1.4. Estructura del documento

La memoria estará dividida en los siguiente apartados:

1. Introducción: Breve resumen del proyecto, motivaciones y objetivos de este.
2. Estado del arte: Descripción del estado actual de las tecnologías utilizadas para este proyecto.
3. Requisitos: Explicación de los servicios que el proyecto debe ofrecer para cumplir con lo esperado.
4. Arquitectura y componentes: Contiene un resumen de la arquitectura del proyecto, los componentes utilizados y la interacción entre ellos.
5. Experimentación: Muestra el resultado del proyecto, así como las pruebas realizadas.
6. Conclusiones: Conocimiento y conclusiones que hemos podido extraer de nuestro trabajo, además de posibles maneras de continuarlo o mejorarlo en un futuro.

Capítulo 2 Estado del arte

En el momento de empezar el desarrollo del proyecto, tras tener claros los requerimientos por parte de los usuarios finales, decidimos evaluar distintas opciones en cuanto a las tecnologías a utilizar en los desarrollos. Entre todos los lenguajes, frameworks y tecnologías de almacenamiento decidimos:

- Utilizar Typescript tanto en Backend como en Frontend para ello utilizamos:
    - NestJS como framework de backend.
    - VueJS como framework de frontend.
- Utilizar una base de datos SQL, en este caso nos hemos decantado por Postgresql.
- Utilizar Docker para la gestión de las imágenes de los desarrollos que se despliegan en la nube y como herramienta de desarrollo local junto con Docker Composer, de forma que todas las dependencias y las imágenes son las mismas para el entorno de producción que para el local.
- Como sistema de gestión de versiones utilizamos Git.
- Trabajamos con la rama *master* sin hacer nuevas ramas ni PR a no ser que estemos probando alguna nueva funcionalidad, queremos acabar con un entorno que nos permita TBD.

NestJS

Hemos decidido utilizar NestJS en el backend porque teníamos claro que íbamos a utilizar un framework que nos proporcionase las herramientas necesarias para la gestión de servicios mediante el patrón contenedor, patrón repositorio para el acceso a datos y con el cuál pudiésemos empezar el desarrollo de manera rápida. Nuestro objetivo es hacer el proyecto API first por lo que los endpoints devuelven JSON.

Durante el máster, en la parte de Node no hemos usado ningún framework en las prácticas, en la parte de Java se usó Sprint Framework. Esto nos ha motivado a usar NestJS para experimentar con un framework que se pareciese al que hemos estado usando en Java.

La elección de NestJS ha sido principalmente por ser un framework similar a otros que ya conocemos en otros lenguajes (Symfony en PHP, Sprint en Java) y porque ahora mismo es un proyecto con una muy buena comunidad y que se está estableciendo casi como un estándar de facto en desarrollos de API dentro del ecosistema Typescript.

VueJS

El objetivo del proyecto no es tener una aplicación lista para el cliente final, si no una prueba de concepto sobre la cual podamos evolucionar rápidamente según los requerimientos cambiantes que nos vayan proporcionando, por eso hemos hecho un prototipado funcional del frontend y para ello hemos elegido VueJS porque es un framework que conocemos y con el que podíamos empezar rápido a conectar con la API de backend.

PostgreSQL

Para la persistencia de datos, elegimos PostgreSQL porque en un principio íbamos a utilizar la extensiones espaciales (PostGIS) que proporciona, pero al final no nos han hecho falta. Aún así es un sistema de base de datos muy eficiente tanto en inserción como en lectura, es software libre y está soportado por RDS de Amazon. Además de ser sencillo de montar en el entorno de desarrollo con Docker.

Docker

Para el despliegue de las imágenes de Frontend y de Backend así como para el desarrollo hemos elegido utilizar contenedores Docker. Esto nos permite tener tanto en desarrollo como en producción sistemas similares, con las mismas versiones y los mismos componentes software instalados. Para el entorno de desarrollo hemos utilizado docker composer para levantar de manera rápida todo el sistema.

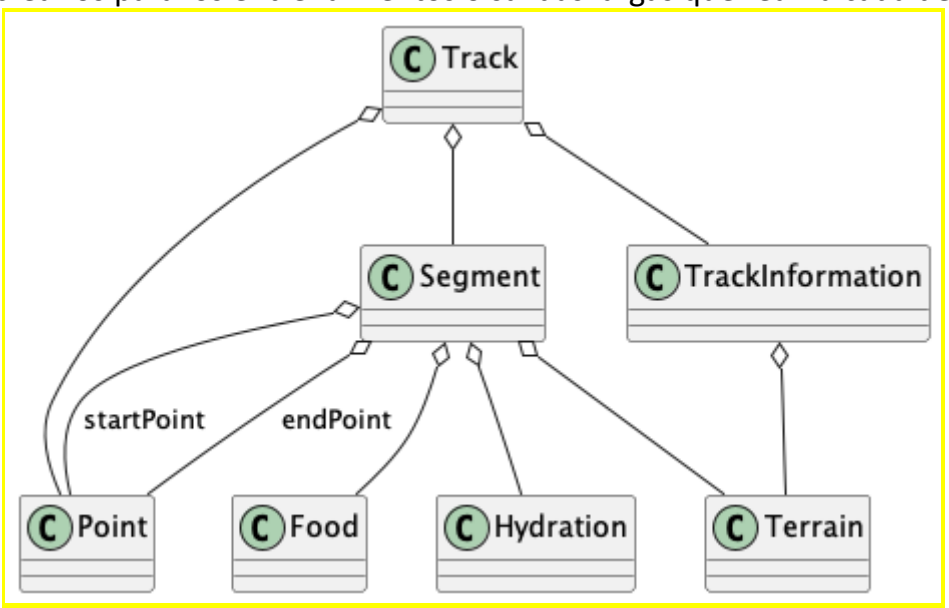
Capítulo 3 - Experimentación

En este capítulo recorreremos cada sección del proyecto para demostrar como se ha usado cada uno de los servicios de AWS para disponibilizar la aplicación web usando AWS CI/CD y como esto se puede apoyar de forma sencilla en IaC con Terraform. Se comparan las tecnologías del proyecto con las usadas en máster y plasmarán las diferencias observadas.

3.1. Aplicación web solida

Antes de mostrar cada una de las secciones a destacar de la infraestructura o tecnologías cloud. Es importante destacar la aplicación web como un componente igual de valioso que el resto de secciones del proyecto. Se trata del núcleo y de la base del desarrollo, es necesario disponer de una aplicación web compleja, para que el resto de secciones tengan un valor real y el esfuerzo en la creación de todos los servicios del proyecto sirvan para dar una mejor experiencia al usuario final y facilitar el trabajo al desarrollador.

Se ha buscado un caso de uso real, que no sólo se quedará en un ámbito académico y con cierto potencial a seguir creciendo para ser una plataforma de uso para los usuarios reales. La aplicación web está centrada en un planificador de carreras de montaña, con la intención de ofrecer al usuario una plataforma sencilla donde pueda subir sus tracks de próximas carreras o entrenamientos, disponga de la información de ese recorrido y pueda añadir la información específica de la alimentación en carrera, avituallamientos, descansos o ritmos que quiere tener. En la mayoría de ocasiones, las carreras disponen de estos mapas informativos, pero no permiten ser configurados de forma personal a cada corredor, no permite crearlos para los entrenamientos o salidas largas que realiza cada deportista.



Para poder desarrollar este caso de uso se ha dispuesto de un backend en NestJS con el que se han generado todos los endpoint del CRUD necesarios para los archivos de tipo GPX. Este archivo establece una forma estándar para el intercambio y almacenamiento de información de mapas en dispositivos GPS, teléfonos inteligentes y ordenadores. Este archivo contiene la información del recorrido con puntos en el mapa (latitud y longitud) ,

altura, metadatos… Para almacenar toda la información de este archivo se ha creado una base de datos relacional de tipo postgreSQL. Como el enfoque del proyecto es usar servicios gestionados de proveedores cloud, en particular de AWS, se ha usado su servicio principal de bases de datos llamado RDS. Destacar la creación de librerías de lectura de estos archivos y la problemática de su guardado para después su lectura de forma eficiente y correcta.

El frontend ha sido desarrollado en VueJS con Typescript para la generación de una interfaz simple e intuitiva para el usuario. El objetivo en el proyecto era crear un frontend sencillo como apoyo al resto de componentes y servicios, por lo que se ha desarrollado lo justo para poder tratar la información del backend y también poder probar lo que sería desarrollar otro sistema de despliegue CI/CD para un frontend. El despliegue de la otra mitad de lo que sería una aplicación web.

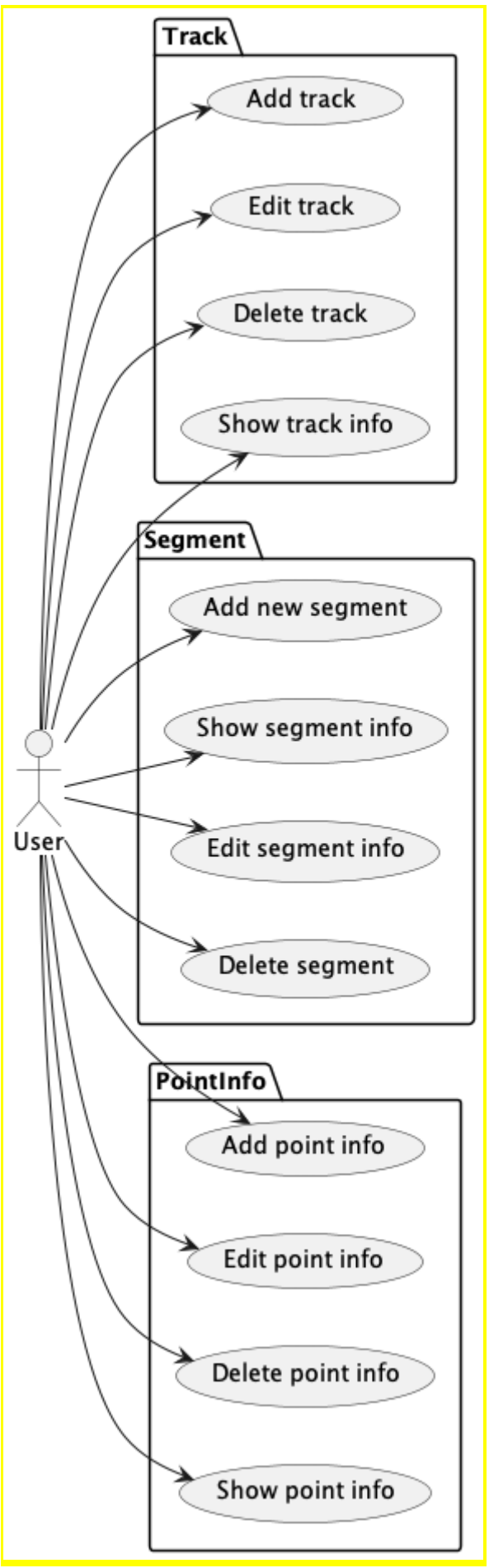
Casos de uso

Como casos de uso iniciales hemos partido de los siguientes:

- Como usuario quiero poder añadir un track al sistema y que se almacene la información de los puntos así como descripción y tipo de terreno.
- Como usuario quiero poder editar la información del track (no los puntos) así como tener la posibilidad de borrar el track.
- Como usuario quiero poder ver la información del track junto con su recorrido.

Como casos de uso futuro tenemos los siguientes:

- Como usuario quiero poder dividir el track en segmentos configurables de forma que:
    - Un segmento nunca puede ser más grande que el track
    - Los segmentos no se pueden solapar
    - El punto de inicio siempre será anterior al de final en el sentido del recorrido del track
    - Los segmentos deben utilizar todo el track, en caso de que la totalidad de los segmentos no cubran el track, los espacios vacíos se generarán como segmentos automáticamente.
    - El orden de los segmentos será determinado por su posición en el track y el sentido de este.
- Como usuario quiero obtener la siguiente información de los segmentos:
    - Distancia del segmento calculada automáticamente
    - Desnivel positivo y negativo calculado automáticamente
    - Tipo de terreno: Lo añade el usuario
    - Comentario del segmento: Lo añade el usuario
- Como usuario quiero poder añadir en el track distintos puntos con información libre, estos puntos se pueden utilizar para determinar avituallamientos, puntos de descanso, etc.
- Como usuario quiero poder editar la información de los segmentos y borrar segmentos. Se deben seguir cumpliendo las condiciones de los segmentos.


Track
Add track
Edit track
Delete track
Show track info
Segment
Add new segment
Show segment info
Edit segment info
Delete segment
PointInfo
Add point info
Edit point info
Delete point info
Show point info
User

Escalado y disponibilidad

Nuestro proyecto está basado en contenedores, de esta forma se puede probar en local de forma fácil y sencilla, sin tener que realizar un despliegue en la nube continuamente para probar el software. Además, los contenedores, son una forma eficiente y confiable de empaquetar, distribuir y ejecutar aplicaciones web. Nos ayudan a mejorar la portabilidad, la escalabilidad, la flexibilidad y la seguridad de las aplicaciones, y fomentan un desarrollo ágil y consistente en entornos de equipo.

Uno de los pilares iniciales del proyecto es aprovechar los servicios de AWS para que la aplicación web pudiera utilizar las ventajas que nos daba la computación en la nube. Con esto en mente, para el despliegue de contenedores se ha usado Amazon Elastic Container Service (ECS), un servicio de administración de contenedores completamente administrado por AWS que ayuda en la implementación y ejecución de aplicaciones basadas en contenedores. ECS ayuda en una aplicación web proporcionando una plataforma escalable, administrada y altamente disponible para la implementación y ejecución de contenedores. Al usar AWS Fargate en el proyecto se encargará de todo el escalado y la administración de la infraestructura necesaria para ejecutar sus contenedores.

Gracias al denominado Task Definition (Definición de tarea) de Amazon ECS hemos podido especificar el contenedor o los contenedores necesarios para dicha tarea, así como la imagen y el repositorio de Docker, los requisitos de memoria y CPU, los volúmenes de datos compartidos y la forma en que los contenedores están vinculados entre sí. Gracias a todas las posibilidades que te brinda ECS de configuración, se puede lanzar tantas tareas como desee a partir de un archivo único de definición de tareas. Los archivos de definición de tareas también nos han dado un control de versiones, por lo que Amazon ECS recuperará automáticamente los contenedores con estado incorrecto para asegurarse de que disponga de la cantidad adecuada de contenedores para su aplicación y se disponga de una versión estable en caso de fallo en la aplicación web.

Sistema distribuido, tolerancia a fallos y balanceadores de carga

La aplicación web aunque siendo una primera versión o MVP, debía ser capaz de tolerar el mayor número de peticiones posibles y afectar lo menos posible a la experiencia de usuario, aunque la carga en el sistema fuera alta. Por lo tanto, acompañando a ECS se dispone el servicio de Amazon Elastic Load Balancer (ELB), en particular un Application Load Balancer, que permite distribuir el tráfico entre los contenedores de ECS que tenemos desplegados en dos zonas de disponibilidad diferentes. Esto garantiza una distribución equilibrada de la carga, mejora el rendimiento y la capacidad de respuesta de la aplicación web, incluso durante períodos de alta demanda.

Con el uso del ALB conseguimos también redirigir automáticamente el tráfico a instancias saludables en otras zonas de disponibilidad. Esto asegura que la aplicación web permanezca disponible incluso en caso de fallos de los servidores backend. Aprovechando así junto a Amazon ECS una escalabilidad automática según la demanda de la aplicación. A

medida que la carga aumenta, los balanceadores de carga pueden escalar horizontalmente agregando automáticamente instancias de backend para manejar el tráfico adicional. Esto garantiza que la aplicación web pueda manejar aumentos repentinos de la demanda sin interrupciones.

Por último, los balanceadores de carga de AWS admiten la terminación SSL/TLS, lo que te permite configurar y administrar los certificados SSL/TLS para tus dominios personalizados. Esto permite que el balanceador de carga maneje el tráfico HTTPS y cifre la comunicación entre los usuarios y la aplicación web, mejorando la seguridad y la confidencialidad de los datos transmitidos.

Sistema de caché en frontend

En el caso del frontend se ha disponibilizado el artefacto final, y por tanto, el código html final en un bucket de AWS S3, especificado este bucket como una web estática Single Page Application (SPA).
Dicha opción es muy usada en la nube ya que disponemos de una web estática con las ventajas que nos da s3 de replicabilidad de los archivos, disponibilidad y escalabilidad. Unido a esto y para mejorar la velocidad de acceso de los usuarios finales se ha añadido como puerta de entrada a este bucket de s3 un Amazon CloudFront. Se trata de un servicio de entrega de contenido (CDN) globalmente distribuido proporcionado por AWS.

CloudFront mejora el rendimiento, la escalabilidad y la seguridad de una aplicación web al proporcionar una red de entrega de contenido globalmente distribuida. Al distribuir el contenido desde ubicaciones cercanas a los usuarios finales, reduce la latencia y mejora la velocidad de carga.

Además , dentro de las propiedades se puede especificar con detalle qué endpoint se guardan en las ubicaciones cercanas del usuario finales, invitando así la sobrecarga del sistema con peticiones similares o iguales entre ellas.

Este apartado aunque introducido en el código de despliegue de la infraestructura, se ha dejado comentado y por lo tanto, fuera del despliegue inicial. Esta decisión ha sido tomada por la naturaleza del frontend en el estado actual del proyecto. Al ser una web estática alojada en un bucket de s3 es necesario refrescar cuando se hace una petición a la base de datos, si se desea obtener la última respuesta el almacenaje en caché no tiene un sentido real.

Redes y conectividad entre servicios

Al tratarse de un sistema completo realizado desde cero y con conectividad a internet, es necesario generar todos los componentes de interconexión interna y hacia internet para que el usuario pueda realizar peticiones a la aplicación web. Ha sido necesario disponer de una VPC para toda la aplicación y cuatro VPC SUBNETS, dos privadas y dos públicas, separadas en parejas en dos zonas de disponibilidad diferentes dentro de una misma región y así poder desplegar de forma paralela en ambas. Las subnets privadas contienen la

base de datos AWS RDS, y por tanto, sin acceso externo. Nada más que se permite el acceso por un security group que se asigna al contenedor de ECS.

Como puerta de entrada al backend se dispone de un Internet Gateway, servicio que proporciona conectividad entre una VPC y el Internet público. Permite que los recursos dentro de la VPC accedan a Internet, se comuniquen con servicios y recursos externos. El Internet Gateway es una parte fundamental de la configuración de red en AWS y facilita la comunicación segura y bidireccional entre la VPC y el mundo exterior.

Destacar la importancia de la configuración de los Grupos de seguridad para controlar puertos y conectividad entre servicios, todas las políticas y roles de IAM siguiendo los estándares de aws sobre privilegio mínimo y el esfuerzo en configuración del Application Load Balancer y CloudFront.

Aunque tratándose de un sistema de red sencillo dentro de las posibilidades en AWS, adquiere un gran importancia la configuración de la red en AWS, en que es esencial para el correcto funcionamiento, el rendimiento de las aplicaciones y servicios en la nube. Una configuración de red adecuada permite la conectividad segura y confiable entre los recursos usados en el proyecto, garantiza una distribución de carga eficiente y mejora la latencia y la disponibilidad de la aplicación. Además, una red bien diseñada puede mejorar la seguridad al permitir una segmentación adecuada de los recursos y la aplicación de políticas de seguridad.

Monitorización y alertado

Gracias a la monitorización en AWS como parte integrada dentro de los servicios de AWS, conseguimos tener una visibilidad completa y garantizar un rendimiento óptimo de una aplicación web en la nube. La monitorización te permite recopilar y analizar datos sobre la infraestructura, los servicios y la aplicación en sí, lo que te ayuda a identificar problemas, optimizar recursos y tomar medidas correctivas de manera proactiva.

En particular en servicios como ECS con el backend desplegado nos permite visualizar en tiempo real los logs de la propia aplicación y si surgiera cualquier error en ejecución aparecería aquí plasmado. Además, otras monitorizaciones que se han usado de forma frecuente en el desarrollo han sido la propia del load balancer para monitorizar cómo se reparte la carga o si su funcionamiento es correcto.

2.2. Terraform vs CloudFormation

Tanto AWS CloudFormation como Terraform son herramientas populares para el aprovisionamiento y gestión de infraestructura en la nube, específicamente en el entorno AWS. Ambas herramientas permiten describir y gestionar la infraestructura como código, lo que significa que puedes definir y automatizar tu infraestructura utilizando archivos de configuración.

Con el uso de Terraform a lo largo del proyecto han aparecido un conjunto de conclusiones en su uso y en cómo esto se podría hacer con CloudFormation. Con las diferentes interacciones que se han hecho en la infraestructura se ha ido consolidado el conocimiento en el uso de Terraform. Se han estudiado las peculiaridades que tiene respecto a otros lenguajes de configuración para generar infraestructura.

A continuación expresaremos en diferentes valores o agrupaciones aquellas secciones que hemos visto destacables en el uso de Terraform versus el lenguaje que ya conociamos y habíamos visto en el máster.

Sintaxis

En el caso de CloudFormation se utiliza JSON o YAML para definir las pilas de recursos. La sintaxis se basa en la estructura de objetos y propiedades. Para Terraform se utiliza su propio lenguaje de configuración llamado HashiCorp Configuration Language (HCL), que es más legible, permite definir recursos y sus interacciones de manera declarativa.
A lo largo del proyecto hemos notado como HCL es un lenguaje muy intuitivo y fácil de tratar, donde se encontraba de forma sencilla cada uno de los servicios que se quería desplegar. Al tratarse de un lenguaje propio y limpio específico para este uso, hace que la lectura y la creación de un código limpio y mantenible sea más fácil que en Cloudformation con JSON o YAML.

HCL proporciona características específicas de Terraform, como la interpolación de variables y referencias a recursos, que facilitan la definición de infraestructuras complejas y la gestión de dependencias. Permite una descripción declarativa de los recursos y sus relaciones en lugar de tener una estructura basada en pares clave-valor. Esto puede hacer que la configuración sea más intuitiva y fácil de entender.

Gracias a los IDEs como pyCharm que disponen ya de plugins específicos de HCL, resulta más fácil el desarrollo en Terraform de la infraestructura. Por otra parte es posible que se encuentre documentación, librerías y herramientas para JSON y YAML, al ser formatos de datos comunes, de manera más fácil que para HCL.

Características y uso en AWS

CloudFormation está específicamente diseñado para AWS, lo que significa que ofrece soporte nativo para todos los servicios de AWS. Puede crear y gestionar recursos de forma integrada, admite la actualización y eliminación de pilas de recursos completas. Terraform está pensada como una herramienta de infraestructura multiplataforma, lo que significa que también es compatible con otros proveedores de la nube, como Azure y Google Cloud Platform. Esto permite definir y gestionar infraestructuras complejas y multicloud.

Vemos una gran ventaja en el uso de Terraform para casos en los que necesitas usar dos nubes en el mismo proyecto o en proyectos diferentes, usando Terraform el equipo de desarrolladores solo tiene que aprender HCL para pasar de un proveedor en la nube a otro.

Por contra, sin haberlo probado directamente en el proyecto, hemos experimentado que Terraform usa el provider específico de aws por lo que muchos de los servicios y recursos generados solo están en aws. No se puede hacer el paso de una nube a otra de forma tan trivial como se querría. Si se quisiera llevar a otros proveedores en la nube como Azure o GCP habría que realizar un número elevado de cambios y ajustar todos los recursos específicos de la nube de AWS.

Además, es importante destacar que en muchas fases en el desarrollo del proyecto ha sido necesario acceder a CloudFormation dentro de la consola de AWS para visualizar cómo es la plantilla de Cloudformation, cómo ha realizado Terraform el despliegue internamente en AWS o si el error de forma externa es insuficiente y necesitas más información de lo ocurrido en un despliegue.

Comunidad y ecosistema

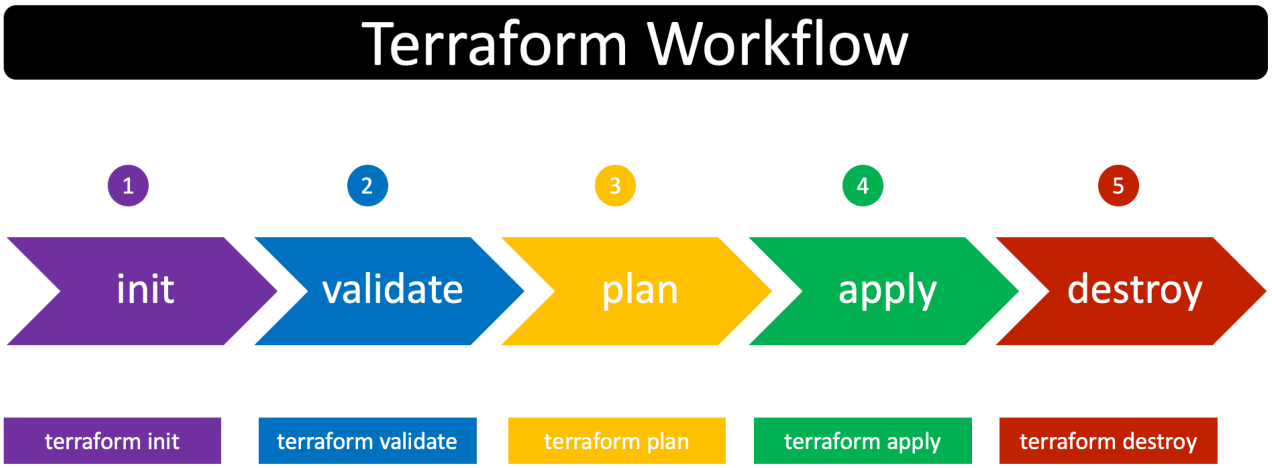
CloudFormation como parte del ecosistema de AWS, tiene una comunidad grande y activa. AWS proporciona una amplia documentación y ejemplos de plantillas, y también hay muchos recursos comunitarios disponibles. Terraform también tiene una comunidad fuerte y activa, pero al ser una herramienta multiplataforma, su comunidad es aún más amplia. HashiCorp, la empresa detrás de Terraform, proporciona documentación completa y ejemplos de uso, y también hay una gran cantidad de módulos y proveedores de la comunidad disponibles.

En ambos lenguajes hay una buena documentación. Es necesario en muchas ocasiones leer ambas para entender cómo usar un recurso y ajustar las propiedades para el despliegue buscado. En Terraform nos hemos encontrado con un avance muy rápido de su ecosistema y como de un mes a otro una función que usábamos perfectamente se ponía en desuso y era necesario buscar una forma alternativa de realizar dicha acción. Esto puede provocar que aunque el servicio no cambie se tenga que ir manteniendo continuamente tu infraestructura al usar Terraform.

Gestión de los cambios

En el caso de CloudFormation se permite realizar cambios en la infraestructura existente utilizando actualizaciones en lugar de crear una nueva pila de recursos. Las actualizaciones pueden ser automáticas o controladas manualmente. Con Terraform añadimos una capa más de control, podemos tener un plan de ejecución antes de aplicar los cambios, lo que permite visualizar y revisar los cambios propuestos. Terraform puede aplicar los cambios de manera controlada y proporciona opciones para deshacer cambios no deseados.

Como hemos visto en el proyecto y se visualiza en la imagen a continuación, gracias a Terraform podemos tener toda una cadena de comandos para tener un control muy completo ante los cambios antes de subirlos, de esta forma estamos seguros que no genera conflicto con algo que ya esté desplegado y pueda perjudicial al uso normal de la aplicación.



A continuación, se explicarán estos comandos y alguno complementario usado en el proyecto:

- *terraform init*: Este comando se utiliza para inicializar un nuevo directorio de trabajo de Terraform. Descarga y configura los proveedores de infraestructura necesarios y establece el backend de Terraform.
- *terraform plan*: Este comando se utiliza para generar un plan de ejecución. Terraform examina los archivos de configuración y muestra una lista de cambios que se realizarán en la infraestructura. Esto incluye la creación, modificación o eliminación de recursos.
- *terraform apply*: Este comando se utiliza para aplicar los cambios definidos en el plan. Terraform realiza los cambios en la infraestructura según lo definido en los archivos de configuración. Antes de aplicar los cambios, se muestra un resumen y se solicita una confirmación.
- *terraform destroy*: Este comando se utiliza para destruir todos los recursos creados por Terraform. Advierte al usuario sobre los recursos que se eliminarán y solicita confirmación antes de realizar la eliminación. Es importante tener cuidado al utilizar este comando, ya que puede borrar la infraestructura existente.
- *terraform validate*: Este comando se utiliza para validar la sintaxis y la coherencia de los archivos de configuración de Terraform. Comprueba si hay errores o advertencias en los archivos y asegura que sean válidos antes de iniciar un plan o una aplicación.
- *terraform state*: Este comando se utiliza para administrar el estado de Terraform, que es un archivo que almacena información sobre los recursos creados y administrados por Terraform. Puedes usar comandos como terraform state list, terraform state show y terraform state rm para ver o modificar el estado de los recursos.
- *terraform workspace*: Este comando se utiliza para administrar espacios de trabajo (workspaces) en Terraform. Los espacios de trabajo permiten gestionar entornos separados y trabajar con diferentes configuraciones. Puedes crear, seleccionar y eliminar espacios de trabajo utilizando comandos como terraform workspace new, terraform workspace select y terraform workspace delete.

2.3. AWS CDK

AWS CDK (Cloud Development Kit) es un marco de desarrollo de software de código abierto que permite a los desarrolladores definir, aprovisionar y gestionar recursos de AWS utilizando lenguajes de programación populares como Python, TypeScript y Java. En lugar de utilizar plantillas estáticas como en AWS CloudFormation, AWS CDK permite escribir infraestructura como código utilizando un conjunto de librerías y constructos de alto nivel.

Una vez que se ha definido la infraestructura utilizando AWS CDK, se puede generar un código de CloudFormation correspondiente. El código de CloudFormation generado se puede desplegar y gestionar utilizando las capacidades y herramientas existentes de AWS CloudFormation. AWS CDK proporciona una abstracción más cercana al código y permite a los desarrolladores utilizar los beneficios de la programación, como la modularidad, la reutilización de código y las mejores prácticas de desarrollo de software.

En el proyecto se ha usado Terraform en vez de AWS CDK, primero con una decisión educativa, siendo una tecnología nueva dentro del ámbito del máster y no usada por los integrantes del TFM. En el caso de CDK sí se ha usado fuera del ámbito del máster. Este conocimiento previo ha generado que se obtengan unas conclusiones con el uso de Terraform en vez de CDK.

AWS CDK puede requerir una curva de aprendizaje más pronunciada en comparación con Terraform. Esto se debe a que los desarrolladores deben familiarizarse con los conceptos de CDK, así como con los lenguajes de programación admitidos (como Python, TypeScript o Java). Terraform, por otro lado, utiliza un lenguaje específico para su configuración, mucho más simplificado, para que en pocas líneas y propiedades de un recurso se realice lo que se desea.

La implementación de un proyecto basado en AWS CDK puede generar un código de CloudFormation mucho más grande en comparación con la implementación de la misma infraestructura con Terraform. Esto se debe a que AWS CDK genera código de CloudFormation detallado y específico para cada recurso definido, lo que puede resultar en plantillas de CloudFormation más grandes y complejas.

Por último, consideramos que el nivel de madurez y estabilidad es mayor en Terraform, aunque AWS CDK ha ganado popularidad rápidamente, sigue siendo una herramienta relativamente nueva en comparación con Terraform, que ha estado en uso y desarrollo durante más tiempo. Esto implica que Terraform puede tener una mayor madurez y estabilidad en ciertos aspectos, como la cantidad de proveedores de la nube compatibles y las funcionalidades avanzadas. Implica también que CDK es más propenso a continuos cambios y con posibilidad de que no sean retrocompatibles con versiones pasadas.

2.3. CI/CD en AWS

El uso de los servicios de CI/CD que nos ofrece AWS para poder acelerar los ciclos de desarrollo y entregar rápido los cambios realizados en la aplicación han tenido un gran peso en el proyecto. Dentro del proyecto se han usado los tres servicios principales de AWS para poder preparar un flujo de despliegue continuo e integración continua. A continuación, proporcionamos una descripción de algunos de los servicios, junto con sus ventajas y desventajas que hemos visto en su uso en el proyecto.

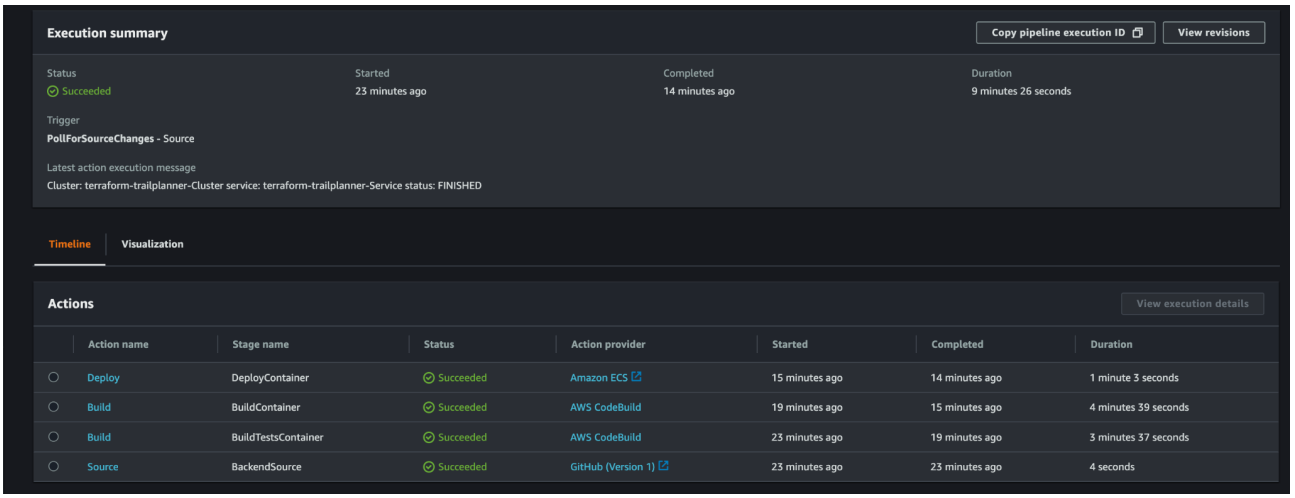
AWS Codepipeline

Es un servicio de CI/CD completamente administrado que permite automatizar el flujo de trabajo de entrega continua de software. Proporciona una forma sencilla de crear, probar y desplegar aplicaciones en AWS. A lo largo del proyecto nos ha permitido configurar fácilmente diferentes etapas, como construcción, prueba, empaquetado y despliegue, para los dos pipelines que tenemos, el despliegue del frontend y backend, con las peculiaridades que tiene cada uno.

CodePipeline se integra de manera nativa con otros servicios de AWS y admite una amplia gama de casos de uso. Puedes integrar herramientas y servicios de terceros, así como escribir scripts personalizados para satisfacer tus requisitos específicos. Por ello ha sido perfecto para el despliegue en una variedad de entornos que teníamos en la aplicación, como AWS Fargate, migraciones en AWS RDS y disponer del frontend en un bucket de s3.

Al disponer de una interfaz de usuario, se visualiza claramente el estado de cada etapa del flujo de trabajo de CI/CD. Puedes monitorear y controlar el progreso de tu proceso de entrega de principio a fin, lo que facilita la detección y solución de problemas. Incluso para una fase de pruebas es muy útil ya que puedes lanzar una release manualmente con el mismo código y así visualizar cómo se comporta el pipeline ante algún cambio.

A continuación se muestran dos ejemplos de la interfaz de AWS Codepipeline en ejecuciones reales del proyecto. El primero muestra de tiempos de ejecución del backend y el segundo la visión de estado general del despliegue del frontend.



Por otra parte, hemos notado que la curva de aprendizaje inicial, cómo configurar y familiarizarse con CodePipeline puede requerir cierto tiempo y conocimiento previo de los conceptos de CI/CD y los servicios de AWS.
Hemos invertido un número superior de horas a las estimadas en aprender a utilizar todas las características y funcionalidades del servicio. En particular destacar la falta de documentación o ejemplos con versiones más nuevas que hay en Terraform para Codepipeline. En la mayoría de las ocasiones nos hemos encontrado que se trataban de versiones muy antiguas, ejemplos hechos desde la consola o demasiado simples para cubrir las dudas que nos surgían en su preparación.

No es el servicio usado en el proyecto que ha generado más costes, pero sí hemos tenido que controlar su ejecución. Tiene un modelo de precios basado en el número de canalizaciones activas y las acciones ejecutadas. A medida que el tamaño y la complejidad del flujo en el proyecto iba en aumento, nos podíamos encontrar costos adicionales al esperado por el uso de CodePipeline.

AWS Codebuild

AWS CodeBuild es un servicio completamente administrado de compilación y creación de artefactos de software. Proporciona un entorno escalable y seguro para compilar, probar y generar paquetes de aplicaciones de manera automatizada.

En el proyecto nos ha permitido mucha libertad para, con los archivos de configuración, generar los artefactos que se necesitaban entre etapas del code pipeline y finalmente el despliegue final. En el caso del despliegue del backend se han usado dos Codebuild, el primero lanza la cadena de test unitarios, para antes del despliegue final asegurarnos que la aplicación sigue estable con los últimos cambios realizados. Además, en este primer Codebuild, siempre que los test hayan ido bien, se genera la migración de la base de datos en AWS RDS, para que en el caso de no existir las tablas o estar en una versión antigua, deja estable la base de datos antes de su despliegue final. El segundo Codebuild del backend es el encargado en conectarse a AWS ECR y realizar con comandos de docker el build y push al repositorio de AWS. En ambos Codebuild se asignan todos los tags de lastest y de su versión.

En el caso de frontend solo disponemos de un Codebuild que se encarga de lanzar la cadena de tests unitarios si dispone de ellos, realiza el comando de build y genera como artefacto el directorio: *dist*. Estos ficheros contienen el frontal de la aplicación web como web estática SPA. De esta forma se puede disponer como web estática en un bucket de s3.

En el desarrollo del proyecto hemos observado varios puntos a destacar de AWS CodeBuild. Al ser un servicio totalmente administrado y con escalado automático, no nos hemos tenido que preocupar por qué máquina o propiedades eran necesarios para lanzar nuestra compilación, sólo nos hemos preocupado en generar los *buildspec*. Se ha integrado muy fácilmente con otros servicios de AWS como Codepipeline y Codedeploy, facilitando mucho la generación de artefactos y donde se disponibiliza. Como hemos expresado antes en el backend, CodeBuild nos ha permitido personalizar el entorno de compilación según nuestras necesidades específicas. Puedes configurar fácilmente las fases de compilación, las tareas de compilación y las pruebas unitarias utilizando scripts personalizados o utilizando las opciones predefinidas disponibles.

Por otro lado, AWS CodeBuild no proporciona un despliegue directo. CodeBuild se enfoca en la construcción y generación de artefactos de software, pero no en el despliegue directo de aplicaciones. Para lograr una entrega continua completa, es necesario combinar CodeBuild con otros servicios, como AWS CodePipeline o AWS CodeDeploy. Aunque el despliegue no es la tarea principal de Codebuild, en el proyecto hemos visto cómo habría sido útil donde no se dispone de un Codedeploy específico, por ejemplo en las subidas a AWS ECR o actualizaciones en AWS RDS. Para no tener que usar la flexibilidad que nos ofrece Codebuild y hacerlo directamente como un paso más.

Hemos notado que a medida que se hacía más grande la aplicación, los costos en Codebuild aumentaban debido a que están basados en el tiempo de compilación. CodeBuild se factura en función de la duración de las compilaciones y el uso de los recursos

de cómputo. Si tienes proyectos de gran escala con tiempos de compilación prolongados, es posible que debas considerar los costes asociados con el uso de CodeBuild.

Por último, en el proyecto se ha notado que Codebuild y en particular con los scripts de buildspec, configurar CodeBuild puede requerir cierto tiempo y conocimientos previos sobre las herramientas y tecnologías utilizadas en tu proyecto. Es posible que debas definir y configurar adecuadamente los entornos de compilación, las fases de compilación y las pruebas para adaptar CodeBuild a la aplicación. En el proyecto hemos tenido que dar muchas iteraciones hasta dar con la configuración y etapas correctas para el despliegue final.

AWS Codedeploy

AWS CodeDeploy es un servicio de implementación automatizada que permite desplegar aplicaciones rápidamente en una variedad de entornos. Proporciona opciones flexibles de despliegue, integración con servicios de AWS y herramientas de desarrollo, actualizaciones sin tiempo de inactividad, rollbacks automáticos y capacidades de monitoreo.
Con CodeDeploy hemos automatizado y simplificado dos procesos. El despliegue en ECS del contenedor a partir de todos los artefactos creados en Codebuild y el subida del artefacto generado en el codebuild para el frontend y su despliegue en un bucket de S3. Ese bucket de S3 está configurado como web estática, por lo que una vez se despliegue el código se actualiza de forma automática en la aplicación web.

AWS CI/CD vs Github Action

AWS CI/CD y GitHub Actions son dos enfoques diferentes pero complementarios para lograr la integración continua y la despliegue continua (CI/CD) en el desarrollo de software. En el proyecto se ha usado la solución de AWS para poder probar los servicios de forma profunda y para que la solución fuera totalmente integrada dentro del servicio de AWS junto con el resto de la aplicación web. Como en el máster hemos usado Github Actions, a continuación, se resumen las características y diferencias entre ambos:

AWS CI/CD:

- Como el resto de la aplicación está en AWS obtenemos una gestión centralizada. En AWS CI/CD, a través de servicios como AWS CodePipeline y AWS CodeDeploy obtenemos una solución de CI/CD completamente gestionada dentro del entorno de AWS. Puedes configurar y orquestar fácilmente pipelines de entrega continua utilizando servicios nativos de AWS. Además, por el mismo motivo, AWS CI/CD se integra estrechamente con otros servicios de AWS, como AWS CodeCommit, AWS CodeBuild, Amazon EC2, AWS Lambda, etc. Esto facilita la construcción, prueba y despliegue de aplicaciones en un entorno de nube coherente.
- AWS CI/CD permite la automatización y orquestación de pipelines de CI/CD utilizando una interfaz gráfica o infraestructura como código (Terraform como en el

proyecto). Puedes definir etapas, acciones y desencadenadores personalizados para configurar y controlar todo el proceso de CI/CD.
- AWS CI/CD ofrece flexibilidad y escalabilidad al permitir la integración con herramientas y servicios de terceros. Por lo que no es necesario utilizar todos sus servicios en el flujo de despliegue, como por ejemplo, en el proyecto se ha usado github como repositorio de código en vez de AWS commit. Además, puedes personalizar y extender el flujo de trabajo de CI/CD utilizando scripts, complementos y servicios adicionales.

GitHub Actions:

- GitHub Actions es una funcionalidad nativa de GitHub. Te permite definir y ejecutar flujos de trabajo de CI/CD directamente desde tu repositorio de GitHub. Al tener en el proyecto el repositorio de código en Github se podría haber tenido un control de esta funcionalidad y desplegar los artefactos desde la CLI de AWS.
- GitHub Actions cuenta con una amplia comunidad de desarrolladores, un ecosistema de acciones predefinidas y personalizadas que se pueden utilizar en los flujos de trabajo. Esto facilita la reutilización y el intercambio de flujos de trabajo entre proyectos.
- Integración con servicios externos: GitHub Actions se integra con varios servicios externos, como servicios de nube, herramientas de prueba, plataformas de implementación, etc. Esto te permite utilizar tus herramientas favoritas en tus flujos de trabajo de CI/CD. En el caso del proyecto con los propios proveedores como AWS.

En resumen, AWS CI/CD y GitHub Actions ofrecen soluciones para la integración continua y la implementación continua, pero con enfoques ligeramente diferentes. AWS CI/CD está más enfocado en proporcionar una solución nativa y gestionada dentro del entorno de AWS, mientras que GitHub Actions está estrechamente integrado con la plataforma de desarrollo colaborativo de GitHub. Por lo tanto, en el caso del proyecto como estábamos enfocados en usar solo servicios disponibles de AWS, tiene más sentido usar ya los propios servicios de CI/CD de AWS con su integración mucho más directa con sus propios servicios.

Hemos notado tras usar de forma más completa ambos servicios que AWS tiene un coste algo más elevado que Github. El pipeline en el caso de Github es gratuito y en AWS se cobra por cada uno. Y aunque el coste por trabajo no es muy diferente si es algo más elevado en AWS que en Github Actions.

Destacar que si se produjera una migración a otro repositorio de código, como por ejemplo Bitbucket, el pipeline de AWS CI/CD no se verá afectado y los cambios en la infraestructura serán mínimos.

En el proyecto hemos notado que la curva de aprendizaje, sobre todo en fases iniciales, en el caso de AWS es alta. En el caso de Github Actions su configuración inicial y en el desarrollo es más sencilla e intuitiva en fases iniciales. En el momento que el proyecto está maduro y el flujo de CI/CD es estable, la complejidad en su uso es similar.

Por último, sin tener un impacto en el proyecto debido a que solo disponemos de una rama master, en el caso de Github Actions hay soporte en el pipeline de CI/CD con varias ramas de forma nativa. En el caso de AWS no es así, es necesario realizar cambios y estrategias en infraestructuras más complejas. No es algo decisivo en nuestro proyecto, pero sí hay que tenerlo en cuenta si se requiere tener varios entornos, como Desarrollo y Producción.

Capítulo 4

4.1. Conclusión

A la vista de los resultados obtenidos y del desarrollo del proyecto, podemos determinar varias conclusiones.
La primera es la dificultad de iniciar un desarrollo de una aplicación web o MVP que sea segura, escalable y con CI/CD, todo ello en la nube y usando IaC. De forma aislada el uso de estos servicios en la nube no requiere mucho esfuerzo, pero sí crear la interconexión de todos los servicios y que su funcionamiento sea el básico para poder tener la aplicación web deseada. Seguramente con los conocimientos que disponemos al terminar tanto de Terraform como con lo aprendido sobre los servicios de AWS, este proceso ahora sería mucho más rápido.
El proyecto se ha ido desarrollando en local con la flexibilidad que nos dan los contenedores en docker, mientras que la infraestructura se completaba. Debido a este desarrollo paralelo no se ha podido aprovechar en la mayoría del desarrollo el flujo de CI/CD en AWS.

Una vez generada toda la aplicación web y el flujo de CI/CD, comprobamos en los últimos cambios como de forma rápida somos capaces de dar valor al usuario final, en pocos minutos desde que se realizaba la subida de código al repositorio está desplegado. Al poder desplegar de forma automática somos capaces de obtener una retroalimentación rápida del usuario y poder iterar con nuevos cambios.

Además, gracias a este flujo de CI/CD somos capaces de proteger el despliegue ante posibles errores de código con la cadena de test unitarios antes de que llegue al usuario final. Existe una parte básica en esta ventaja y es que es necesario una cadena de test completa y robusta, si son test insuficientes o no están correctamente hechos, dicha protección no es aprovechada al máximo. Hay que tener en cuenta también el desarrollo continuo de los test acompañando a las nuevas funcionalidades.

En el proyecto hemos tenido que trabajar con muchos productos diferentes que ofrece AWS, desde servicios de bases de datos como RDS a servicios de balanceo de carga como ALB. AWS consigue que muchas acciones que son necesarias en una aplicación web y que tendría que gestionar los desarrolladores en una máquina, se conviertan en tareas más sencillas gracias a todos sus servicios gestionados en la nube.
Como desarrolladores nos olvidamos de tener que actualizar versiones de la base de datos o de gestionar las máquinas donde se despliega el backend, esto consigue que se gaste menos esfuerzo en la infraestructura y se esté más centrado en el producto a desarrollar. Aunque esta ayuda es real, sigue siendo necesario un conocimiento en los servicios de AWS elevados para realizar un desarrollo mínimamente complejo. Se tratan y conectan muchos servicios diferentes que si no conoces cómo funcionan, cómo se configuran y sus bases sólidas, es decir, qué está sucediendo internamente, hacen que pueda ser imposible avanzar o arreglar problemas que te encuentras en el desarrollo. Debido a todo lo explicado anteriormente, nos hemos encontrado que la curva de aprendizaje en el uso de

la nube junto con el uso de Terraform ha ido aumentando a medida que se insertan nuevas funcionalidades y la aplicación web se vuelve más compleja.

Conociendo que el proyecto iba ir creciendo en funcionalidades, Terraform nos ha ofrecido una gran flexibilidad para ir probando los cambios en la infraestructura. Aunque con una curva de aprendizaje inicial algo alta, con los conocimientos básicos adquiridos hemos sido capaces de ir generando toda la infraestructura siguiendo los estándares que marca AWS en sus pilares arquitectónicos, generando un código escalable y mantenible en el tiempo. Una tecnología de IaC ha sido perfecta en el ámbito del proyecto, poder borrar todos los servicios tras cada sesión de trabajo nos permite no incurrir en gastos innecesarios cuando no se trabajaba en el proyecto. Así poder recuperar y desplegar el último estado de la infraestructura cuando se continuaba con el desarrollo.

Hemos visto en el proyecto como Terraform o CDK es mucho más intuitivo y fácil de usar que el IaC nativo de AWS, Cloudformation. Gracias al uso de módulos de Terraform y el formato que tiene con su lenguaje propio HCL, hace que su trabajo y visualización de recursos sea más rápida que en YAML o JSON.
La gestión de errores o problemas que te encuentras en el desarrollo es más clara y sencilla en Terraform en la mayoría de casos y los errores de sintaxis han sido más rápidos de localizar antes de tener que realizar el despliegue en la nube. El problema que más ha afectado al proyecto ha sido la documentación o uso de ejemplos. Los IaC que despliegan en AWS al tener que traducir a Cloudformation para su despliegue hacen que la documentación del lenguaje nativo sea la más actualizada y completa. En el caso de Terraform en algunas ocasiones han faltado ejemplos actualizados o claros para lo que se buscaba teniendo que ir a la documentación de Cloudformation e investigar como se traduce a Terraform. Por último, en el caso de Terraform no creemos que realmente se pueda usar un mismo código en diferentes proveedores en la nube, debido al uso de funciones muy específicas de cada proveedor cloud en el que despliegas, pero sí es útil tener un único lenguaje para diferentes proyectos y poder tener diferentes nubes en un mismo proyecto. Todo esto no ha sido validado en el proyecto.

Dentro de los objetivos del proyecto no estaba contemplado dar prioridad a uno de los pilares de AWS en lo que refiere a la gestión de costes de una arquitectura en la nube. El proyecto en un inicio se ha contemplado para ser desarrollado dentro del free tier de AWS, que proporciona muchos servicios de forma gratuita por un año con limitaciones que dependen de qué servicio. Según iba creciendo el proyecto se han integrado servicios que no están dentro de esta capa gratuita y algunos servicios que para el correcto funcionamiento de la aplicación web superan esos limitantes básicos que ofrece AWS. Hemos visto que aunque AWS ofrece grandes ventajas respecto a servicios on-premise y poder ejecutar bajo demanda tu aplicación web, en el caso de una idea de negocio que se lleva a cabo o la creación de un aplicación web con cierta dificultad es importante controlar los costes para no llevarse ninguna sorpresa. Se debe dar importancia al coste versus ingresos por usuario de nuestra aplicación web.

En conclusión, el proyecto ha servido para mostrar el uso de las tecnologías y servicios de CI/CD en la nube para disponer de una aplicación web con cierta complejidad y con un caso

de uso real. Muestra la necesidad de conocer los servicios del proveedor de la nube que se usa y su uso en profundidad, añadiendo una capa más al proyecto desplegando la infraestructura con Terraform como IaC. Como el uso de Terraform ayuda en el trabajo con la infraestructura y su comparación con otros servicios conocidos de IaC. La unión de un IaC, con un proveedor en la nube como es AWS y un sistema de CI/CD, hacen que una aplicación web del proyecto pueda crecer lo que se desee.

4.2. Trabajo futuro

La plataforma ha sido desarrollada con la flexibilidad y la usabilidad en mente para que, a través de la publicación de la misma como software libre, cualquier institución, empresa o particular pueda aportar a esta. La idea de un planificador de carreras de trail running tiene una posibilidad real en el mercado y puede ir creciendo hasta que se volviese una aplicación web usada por muchos usuarios. En el proyecto se ha realizado la base para añadir tracks al sistema, pero hay muchas funcionalidades que no son complejas a corto plazo y aportan valor al usuario final, como por ejemplo secciones dentro de un track o la propia gestión de usuarios.

En la aplicación web se tiene automatizado el código del backend y frontend para que se pueda desplegar de forma automática y segura cuando se reciba un commit. Pero en el caso de la infraestructura en Terraform se realiza el despliegue de forma manual con los comandos de Terraform, sería interesante añadir otro AWS Codepipeline que apunte al repositorio de infraestructura y así ningún despliegue tendría que realizarse de forma manual por los desarrolladores, incluso si se ve necesario añadir cadenas de test y validaciones antes del despliegue.

Por otra parte, hemos encontrado que los tiempos en el despliegue del backend en AWS ECS son elevados, debido a que tiene que hacer la construcción del artefacto en dos ocasiones, para lanzar los test y después para crear el artefacto final, algo que no es limitante a nivel funcional pero hace que la entrega de un cambio al usuario final pueda ser más lenta y si se desea hacer varios cambios de forma continua o a la vez, puedan generar un retardo elevado. Para ello sería interesante investigar a qué es debido y cómo optimizar dicho despliegue.
Se podrían optimizar los comandos de los AWS Codebuilds, configurar las máquinas de despliegue de forma diferente o incluso plantear otro servicio que no sea ECS. Esta última solución podría ser interesante para tratar también la mejor gestión de los costes. Como otro punto de mejora sería interesante investigar con detalle cuáles son los servicios que más cuestan como es el caso de ECS y balanceadores, para estudiar alternativas como AWS Batch o investigar configuraciones de estos servicios optimizando los costes.

En su comienzo la intención del trabajo era el desarrollo de un software usando TBD, debido a la complejidad del desarrollo de la infraestructura y la propia aplicación web, nos centramos en otros apartados, pero sería interesante ahora que la aplicación web y el pipeline de CI/CD es estable y es completo, añadir las nuevas funcionalidades con una estrategia de desarrollo TBD.

Capítulo 5 - Bibliografia

1. Documentación oficial de servicios aws: https://docs.aws.amazon.com/?nc2=h_ql_doc_do
2. Terraform plan: https://terraformguru.com/terraform-certification-using-azure-cloud/03-Terraform-Command-Basics/
3. Documentación oficial Terraform. Aws provider. https://registry.terraform.io/providers/hashicorp/aws/latest/docs
4. Tutoriales diferentes de Terraform: https://galvarado.com.mx/post/tutorial-infraestructura-como-c%C3%B3digo-con-terraform/
5. Acloudguru. Cursos de AWS y Terraform. https://learn.acloud.guru/dashboard
6. Ejemplo aws-samples ecs ci-cd con Terraform: https://github.com/aws-samples/aws-ecs-cicd-terraform
7. Ejemplo aws-samples codepipeline con Terraform: https://github.com/aws-samples/aws-codepipeline-terraform-cicd-samples
8. Modulo de RDS Aurora para Terraform: https://github.com/terraform-aws-modules/terraform-aws-rds-aurora
9. Framework NestJs: https://nestjs.com/
10. ORM utilizado en el desarrollo: https://typeorm.io/
11. VueJs: https://vuejs.org/
12. Blog de Wanago que nos ha sido de gran ayuda para configurar y TypeORM en Nestjs: https://wanago.io/2020/05/18/api-nestjs-postgresql-typeorm/
13. Entrada del blog de Wanago que hemos utilizado para configurar las migraciones automáticas en NestJs: https://wanago.io/2022/07/25/api-nestjs-database-migrations-typeorm/
14. Especificación del formato de fichero GPX: https://www.topografix.com/GPX/1/1/

Capítulo 6 - Anexo

A continuación se dispondrán los enlaces de github donde está el código del proyecto y la documentación técnica de cada uno:

- Infraestructura en Terraform: https://github.com/MasterCloudApps-Projects/trail-planner
- Código backend: https://github.com/MasterCloudApps-Projects/trail-planner-backend
- Código frontend: https://github.com/MasterCloudApps-Projects/trail-planner-frontend