Universidad Internacional de la Rioja

ESIT

Máster universitario en DevOps, Cloud Computing y Automatización de Producción de Software.

Construcción automatizada de una API Rest en AWS para pequeñas empresas

Trabajo Fin de Máster

Presentado por: Ortiz Llamas, Borja

Director: Lucas Simarro, Jose Luis

22/7/2020, Colmenar Viejo, Madrid, España

Resumen

Este proyecto trata de resolver uno de los problemas que se pueden encontrar todas las empresas pequeñas, que es encontrar un software que les resuelva sus funcionalidades y que no tenga un precio muy elevado. Para esto se usa una API Rest con la funcionalidad de guardado de clientes con los productos que han comprado, que este preparada para su evolución constante y así se consiga a lo largo plazo cumplir con todas las expectativas de las pequeñas empresas. Para conseguir esto seguiremos la cultura DevOps, para la evolución constante se han creado unos circuitos de integración y circuitos de despliegue completos y con calidad, también desplegando la aplicación con infraestructura como código. Ha sido usada Cloud pública, Amazon Web Services, con toda la infraestructura. Se usara la metodología SCRUM para la organización del proyecto. La aplicación consigue desplegarse en minutos y con capacidad completa de evolucionar.

keywords: AWS, Infraestructura como código, integración continua y despliegue continuo, DevOps.

Abstract

This proyect is about resolve one problem can have all the small companys, that is find a software who can resolve their functionalities and with no expensive price. For this will be used an API Rest with the funcionality of save clients with theirs products, also with capacity of improve easily and with that the aplication will achieve the objetives of the small companies. For this it is going to be used the DevOps culture, for the continuous evolution have been created circuites of integration and deployment, also deployment the infraestructure as code all the infraestructure. The SCRUM metodology for the proyect. This aplication has achieved all the objetives.

keywords: AWS, Infraestructure as Code, continuous integration and continuous deployment, DevOps.

Índice

Listado de figuras

1. Introducción

Este trabajo tiene el objetivo de realizar la construcción de manera automatizada una API Rest dirigida a realizar la funcionalidad de guardado de clientes para pequeñas empresas, que sea capaz de desplegarse y usarse de la manera más sencilla posible y que pueda tener un ciclo de vida también lo más largo posible. Esto para conseguir ofrecer a las pequeñas empresas la capacidad de tener su funcionalidad con una API Rest que sea capaz de cambiar velozmente adaptándose a un mercado completamente cambiante. Esto es específicamente para lo que la cultura DevOps ha sido creada, por esto este proyecto será realizado basandonos en esta cultura y intentando aplicarla en todo momento.

Para cumplir con toda la funcionalidad que necesitan la pequeñas empresas desde el primer momento se necesitaría mucho tiempo, ademas de tener la capacidad de entender muchísimos sectores, conocimiento que es muy difícil de tener. Por ello se necesita que este proyecto sea capaz de cambiar fácilmente para cuando empiecen a añadirse nuevas empresas se pueda evolucionar de la manera más sencilla posible. Además de que esta aplicación tiene que estar monitorizada para que cuando pudieran haber errores sean solucionados rápidamente. También se necesitará que los clientes prueben sus funcionalidades, antes incluso de comprar la aplicación y tener un entorno productivo. Se ha de ser capaces de realizar todas estas tareas lo más rápido y eficazmente posible.

Para conseguir estos objetivos, se ha trabajado en este proyecto con la cultura DevOps, automatizando todo lo posible, y desplegandolo en una infraestructura Cloud. Se ha usado Amazon Web Services que es una de las Cloud publicas más usadas, con lo cual nos aseguramos mantenimiento a largo plazo. Así como metodologías ágiles para que la organización funcione lo más rápido posible. Con un entorno de trabajo de los desarrolladores para la aplicación, llamado desarrollo y uno de producción dedicado a las empresas. También usando tecnologías que tengan tengan un largo ciclo de vida y de esa manera sigan siendo mantenidas con parches de seguridad, cosa muy importante para cualquier negocio. Ademas los entornos de integración tienen que ser eficaces, poniendo código en producción lo más rápido posible con pruebas, tanto unitarias (Junit) como integradas (desplegandose con DockerCompose), se harán las pruebas cada vez que se realiza la compilación. También tenemos que tener en cuenta que no solo el ciclo de desarrollo de la API deberá de tener sus entornos de integración, debemos incluir la

infraestructura como código usada para desplegar entornos en Cloud y así poder desplegar de manera muy sencilla entornos nuevos.

La motivación con este proyecto es que se pueda dar a las pequeñas empresas un buen entorno de trabajo con toda la funcionalidad necesaria para la compra venta. Tener una API Rest que sea capaz de evolucionar velozmente puede ser muy beneficioso, de esta manera las empresas podrían tener su funcionalidad en una aplicación móvil y web unificada con los mismos datos, incluso integrarla en otros aplicativos de terceros. Podrán dar un mejor servicio a los clientes y poder aumentar su beneficio de ventas.

Ademas otra motivación que tiene el proyecto es poder ofrecer de manera rápida y barata el consumo del software. Siguiendo la cultura DevOps seremos capaces de ofrecer lo más rápido posible el nuevo software. Y con la Cloud de Amazon seremos capaces de ofrecer el servicio de una manera barata, y sin preocupaciones de que en un corto plazo vayan a tener que cambiarlo por actualizaciones.

1.2 – Estructura de la memoria

Capítulo 1 – Introducción:

> Se presentará la memoria, una explicación de los objetivos de este proyecto y como serán realizados.

Capítulo 2 – Estado del arte:

> Se describirá la funcionalidad de este proyecto y los casos de uso que serán desplegados. También las tecnologías que serán usadas en este proyecto de manera teórica.

Capítulo 3 – Objetivos y metodología

> Se explicara los objetivos principales y secundarios de este proyecto, también la metodología que sera usada en el proyecto.

Capítulo 4 – Descripción de los automatismos:

Explicación de los automatismos, como serán usados y lo las tecnologías que desplegarán para el uso en el proyecto.

Capítulo 5 – Conclusiones:

Los objetivos cumplidos del proyecto y los conocimientos que se han extraído del proyecto que podrán ser usados en otros.

Capítulo 6 – Posibles mejoras:

Mejoras que podrá tener este proyecto tecnológicamente y metodológicamente.

2. Estado del arte

2.1 Descripción de la aplicación

Esta aplicación se encargara de manejar a los clientes con los productos que hayan comprado en el comercio. Esta funcionalidad estará en un API Rest desplegado en AWS. Para poder ver los métodos, cabeceras, objetos JSON y las descripciones de la API en este swagguerHub : https://app.swaggerhub.com/apis/borjaOrtizLlamas/SmallCompaniesAPI/1.0.0
También el versionado de la API podrá ser encontrado en la misma Url, y cual esta publicado.

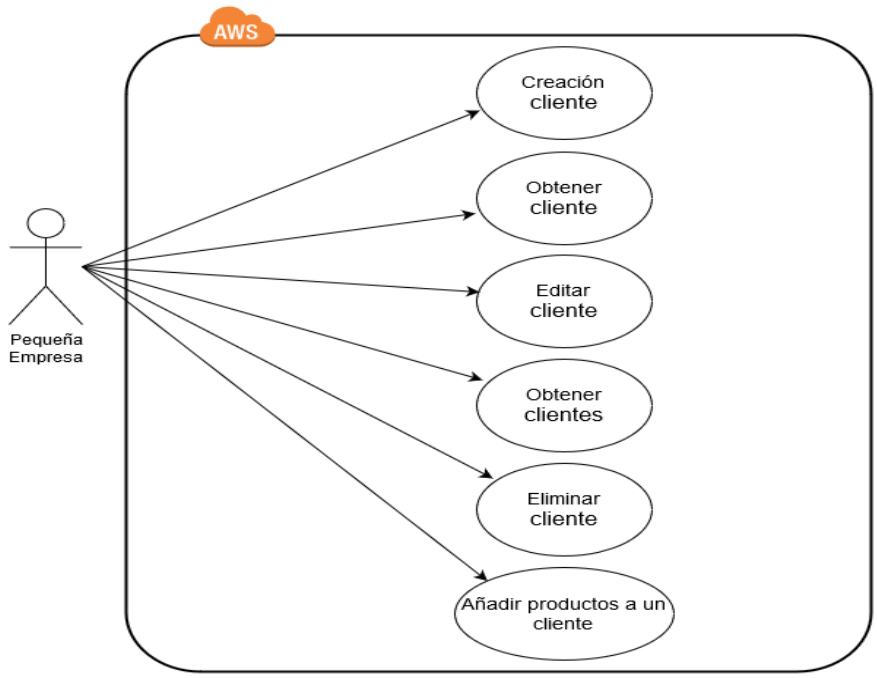

Figura 1. 2.1: Diagrama de casos de uso.

Como vemos en la figura 1, el diagrama de casos de uso , tendremos 6 funcionalidades principales en la aplicación (También están descritos en el swaggerHub), que consistirán en un CRUD de clientes de la empresa. Usaremos las buenas practicas de Rest. Objeto cliente:

```
{
  "name": "Juan Pedro",
  "products": [
    {
      "name": "Zapato",
      "price": "2.3",
      "description": "zapatos diversos"
    }
  ]
}
```

Figura 2: 2.1 Ejemplo del objeto JSON Cliente

- Creación cliente: En ese flujo se podrá crear un cliente nuevo. Se generara el cliente nuevo con su nombre como clave primaria, no podrá haber 2 clientes con el mismo “name”, se enviara como body el objeto JSON Cliente .
- Obtención de un cliente: En este flujo se podrá obtener un cliente en especifico según su “name”. Se usara la ruta “/client/${name}” con el método GET.
- Editar cliente: En este caso se podrá editar un cliente según su “name”. Se usara la ruta de “/client/${name}” con el método PUT, se enviara como body el objeto JSON Cliente.
- Obtener clientes: Se podrán obtener todos los clientes. Se usara la ruta de “/client/” con el método GET.
- Eliminar cliente: Se podrá eliminar un cliente en especifico en la aplicación. Se usara la ruta de “/client/${name}” con el método DELETE.
- Eliminar cliente: Se podrá añadir productos un cliente en especifico en la aplicación. Se usara la ruta de “/client/${name}” con el método POST y enviando un objeto de tipo producto.

2.2 Descripción de las tecnologías

Tecnologías de la familia de la Cloud de AWS [1]*:*

ECS (Kubernetes) [1]:

AWS ECS (Amazon Elastic Container Service) es un servicio de orquestación de contenedores de Amazon, tiene el mismo funcionamiento que Kubernetes pero en AWS. Tiene capacidad de funcionamiento con AWS Fargate y con que es un motor informático sin servidor para desplegar contenedores. Aquí lo que se va a usar en este proyecto:

- Task: Es capaz de lanzar de lanzar contenedores en el mismo contexto. Capaces de tener volúmenes, limites de CPU y memoria. Tiene el mismo funcionamiento que un Pod en kubernetes.
- Service: Son capaces de lanzar diversas task en el mismo servicio, tendría el mismo funcionamiento que un replicaset en Kubernetes. Tiene capacidad de desplegar pods en distintas subzonas y distintas redes de manera autogestionada para que tenga capacidad de recuperación si tenemos errores en zonas.
- Cluster: El cluster de ECS tiene la capacidad de lanzar servicios.

---

[1] https://aws.amazon.com/es/ecs/

Registry ECR (Registry)[1]:

Amazon Elastic Container Registry es el registro de contenedores de AWS. Se situá en una infraestructura escalable de alta disponibilidad y esta integrado con IAM.

Servidor de ejecución en EC2 (Maquina Virtual) [2]:

Amazon Elastic Compute Cloud es un servicio que proporciona maquinas virtuales con Ubuntu server.

Infraestructura especifica para redes:

- *Aplication load balancer [3]:* AWS dispone de un balanceador de carga clásico. Puede funcionar como un balanceador de carga sobre distintos grupos de instancias, como puede ser grupos de nodos. Proporciona direccionamiento de solicitudes dirigidos a la arquitecturas modernas como una de microservicios. Tiene la capacidad de tener una URL propia y redireccionar peticiones a una red interna.
- *Route53 [4]*: Es el servicio de DNS de AWS. Esta preparado para usarse con todos los demás servicios de AWS como de EC2 y ECS.

Packer [5] (Generador de AMIS):

Packer es un automatizador para generar imágenes, puede usarse en diversas plataformas y no hay problema de su uso con AWS. Puede ser aprovisionado por Ansible.

Ansible [6] (Administración de configuraciones) :

Ansible es un software que sirve para la automatización de configuración y administración de servidores. Utiliza YALM para describir las configuraciones que se ejecutan en los servidores, son llamados playbooks.

---

[1] https://aws.amazon.com/es/ecr/

[2] https://aws.amazon.com/es/ec2/

[3] https://aws.amazon.com/es/elasticloadbalancing/

[4] https://aws.amazon.com/es/route53/

[5] https://www.packer.io/docs

[6] https://docs.ansible.com/

Terraform [1] (Infraestructura como código) :

Terraform es un software que sirve para ejecutar código que despliegue infraestructura, este proceso llamado “Infraestructura como código” sirve para automatizar despliegues como realizar modificaciones en infraestructuras ya desplegadas. Terraform tiene capacidad de uso con AWS, Google Cloud, Azure y la gran mayoría de Clouds tanto publicas como privadas. Terraform guarda el estado de la infraestructura y a la realización de cambios se modifica lo que esta desplegado en la nube.

Docker [2]:

Docker es un software que sirve para crear contenedores de de software. Los contenedores de software es un sistema de visualización en el que un único sistema operativo provee un kernel a distintos contenedores de software (aislados unos de otros). Los contenedores de software sirven para desplegar aplicaciones de manera de manera aislada y monolítica, son rápidos de desplegar y los contenedores pueden compartir steps y disminuir su tamaño.

Elastic Stack [3] (Monitorización):

Generado por la empresa Elastic, este conjunto de software sirve para poder monitorizar tu aplicación. El diminutivo de esta tecnología es ELK. Tiene distintas aplicaciones con cada una un objetivo:

- Kibana: Provee de la visualización de los logs en un dashboard. Tiene diversos plugins para poder mostrar datos.
- Elasticsearch: Motor de búsqueda y analítica, base de datos para logs.
- LogStash: Pipeline de procesamiento de datos, recibe datos, los transforma y los enviá a elasticSearch. Tiene capacidad de modificar los datos que ha recibido para distintos outputs.
- Beats: Plataforma de agentes ligeros que envían datos de maquinas. Tienen diversos agentes, como puede ser filebeats que lee ficheros y envía estos logs donde se indique.

El ciclo de alimentación de logs este:

---

[1] https://www.terraform.io/docs/

[2] https://docs.docker.com/

[3] https://www.elastic.co/es/elastic-stack

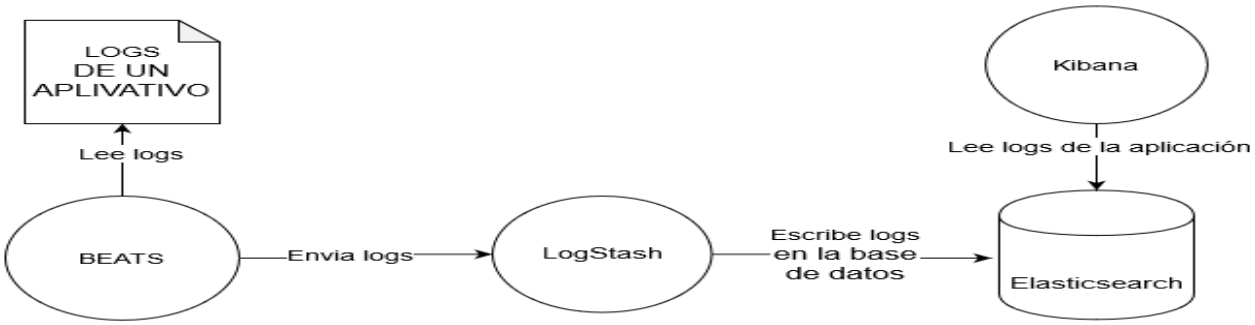

Figura 3: 2.2 Alimentación de logs

MongoDB [1] (Base de datos):

Base de datos no relacional, guarda los datos en estructuras BSON. No tienen esquemas prefijados y permite que los modelos de datos sean muy dinámicos.

Jenkins [2] (Automatización de despliegues) :

Jenkins es una aplicación que sirve para automatizar el despliegue de aplicaciones. Con diversos plugins es capaz de controlar la integración continua y el despliegue continuo de las aplicaciones de una organización.

Java con el framework Spring boot [3] (Aplicativo) :

Este framework tiene la ventaja de que cuando desplegamos la aplicación que tiene un tomcat embebido, que permitirá recibir peticiones. También aprovisiona de librerías de Spring core para una configuración sencilla. Trabaja con peticiones de tipo REST y tenemos la capacidad de usar los metedos de este tipo. Así tendremos la capacidad de generar un API REST con este Sofware. Ademas, al ser ejecutado en Java, tenemos drivers muy usados por la comunidad para distintas bases de datos como puede ser MongoDB o Oracle.

Maven [4] (Empaquetador de aplicación):

Esta herramienta sirve para la construcción y empaquetado de proyectos Java. Provee de librerías y usa un XML para su configuración, llamado “pom.xml”.

---

[1] https://docs.mongodb.com/

[2] https://www.jenkins.io/doc/book/

[3] https://spring.io/projects/spring-boot

[4] https://maven.apache.org/guides/index.html

Git [1]:

Es un Software de control de versiones. Guarda el código en repositorios. Git trabaja con ramas y commits, las ramas son un conjunto de commits. Git dispone de diversos flujos de trabajo, Gitflow es el más usado utiliza distintos prefijos de ramas para diferenciar el trabajo entre equipos de manera más sencilla,. Gitflow sigue esta nomenclatura para las ramas de apoyo para el momento de trabajo:

1. Feature: Estas ramas con este prefijo sirven para nuevas tareas que se han de realizar. Por ejemplo: feature/update-flow.
2. Bugfix: Estas ramas con este prefijo sirven para solucionar errores durante el desarrollo y cuando el código no ha salido a producción.
3. Hotfix: Estas ramas con este prefijo sirven para cuando hayan errores en producción que tiene que ser rápidamente arreglado.

Las otras ramas estarían basadas en entornos, por ejemplo master correspondería con el entorno de producción.

Cada versión es un “commit” dentro de este software, disponen de un hash propio y los podremos manejar con facilidad.

---

[1] https://git-scm.com/

3.Objetivos y metodología

3.1 Objetivos principales

Los objetivos principales de la aplicación define el alcance del proyecto, hasta donde seremos capaces de realizar con el mismo. Son estos:

- Poder crear clientes desde una API Rest pública de manera sencilla. Definir la interfaz de clientes.
- Capacidad de integración continua de infraestructura como código y su despligue continuo con la herramienta de Terraform. Siendo esta desplegada por entornos.
- Capacidad de de integración continua de los contenedores usados en la API
- Despliegue de los contenedores con Kubernetes. ECS es la tecnología que se encarga de esta funcionalidad en Amazon. Es la que usaremos con un cluster en desarrollo y otro en producción.
- Capacidad de creación de AMI's que serán usadas en la infraestructura aprovisionadas con Ansible.

3.2 Objetivos secundarios

Objetivos que no son necesarios para que el proyecto se lleve a cabo pero sí una considerable mejora y que se deberían de llevar a cabo para que el proyecto sea realizado en una verdadera cultura **DevOps**, son estos:

- Monitorizar la aplicación con Kibana.
- Pruebas integradas de la API Rest. Se programarán las pruebas desde otro proyecto para que puedan codificarse de manera separada. Esto dará la posibilidad de añadir un equipo como QA.

3.3 Metodología

Este proyecto al estar planteado en una cultura **DevOps** sin ninguna duda, será realizado con una metodología de tipo ágil. Usaremos dentro de las metodologías ágiles, SCRUM. Tiene periodos de

tiempo con unos determinados ritos llamados Sprint que se van repitiendo, en estos periodos deberemos de llevar acabo un determinado numero de tareas, llamadas Historias o Features. Los ritos de SCRUM son 4 y se realizaran con el equipo:

- Dailys: que se producen todos los días para explicar los bloqueos y el trabajo a entregado.
- Sprint planings: Se producen cada comienzo de Sprint para ver las historias que hacer.
- Sprint reviews: Esta reunión trata de mostrar lo realizado en el Sprint.
- Sprint retrospective: Este rito trata de intentar con el equipo de ver en que se puede mejorar.

Una de sus grandes ventajas de la metodología SCRUM es que tenemos la posibilidad de variarlo.

En el caso de este proyecto, que será realizado por un desarrollador tendremos Sprints de un mes, las dailys no serán necesarias al ser un solo desarrollador, las retrospectivas tampoco serán necesarias al ser un solo desarrollador, las Sprint reviews se realizarán con el director del proyecto una vez al mes y finalmente con la presentación del proyecto.

Deberá de ser incluido un Sprint 0, que consiste en hacer las tareas para crear la proposición del proyecto y su gestión. También definir el alcance del mismo y hasta que punto queremos llegar. Cosa que hemos resueltos en el punto 3.1 Objetivos principales y 3.2 Objetivos secundarios de esta memoria. Estos puntos fueron definidos en la propuesta del TFM.

Para este proyecto sera usado un tablero Kanban, y sera abierto un tablero para este proyecto en la aplicación Trello[1]. Las tareas podrán estar en 5 estados en nuestro Kanban:

1. Backlog: Se encuentran redactadas y se indica el objetivo que se quiere alcanzar con esta feature.
2. To do: en este estado se encontraran las tareas que seran realizas en este ciclo (Spring) por el desarrollador.
3. Working: Se encuentra desarollandose en este momento.
4. Blocked: Una tarea que se encuentra bloqueda por alguna razón. Por ejemplo una tarea puede ser parada por falta de información o falta de tiempo.
5. Done: La tarea se encuentra realizada

---

[1] https://trello.com/

El tablero que usaremos de trello es público, este es su enlace:https://trello.com/b/dnpShqZu/tfm.

Aquí un ejemplo en la figura 4 de las tareas del sprint 1, en el que se realizo las imágenes de IAM de la instancia de MongoDB, Jenkins y Elastic stack:

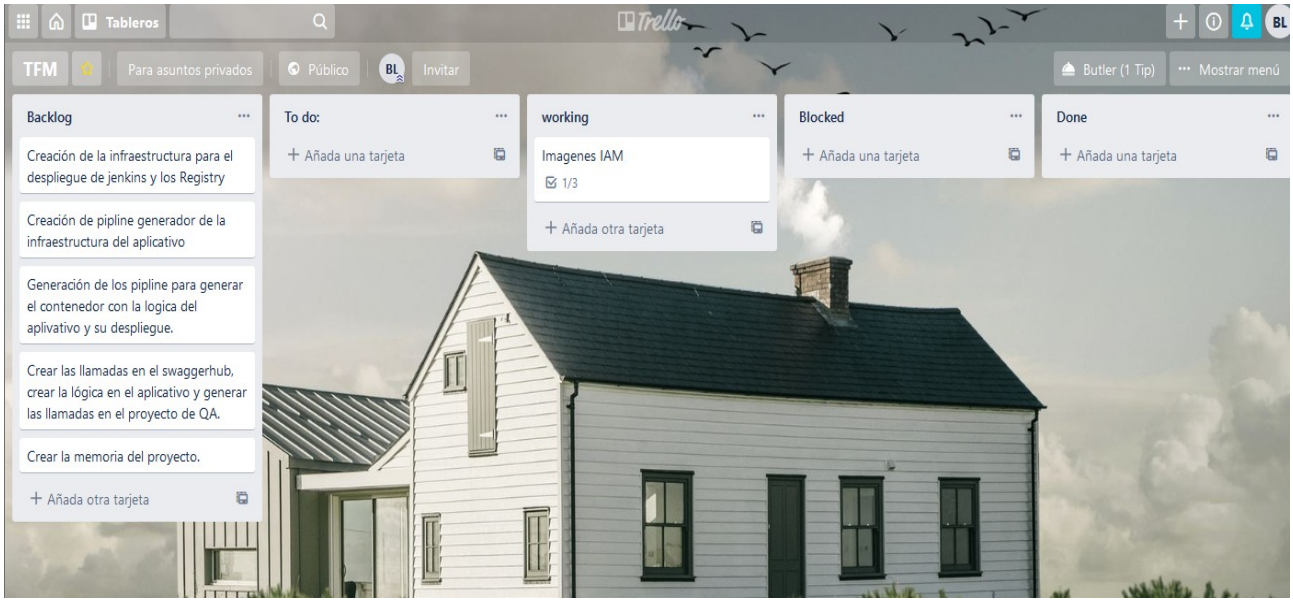

Figura 4: 3.3 Sprint 1

En la figura 5 se observan las tareas del Sprint 2.

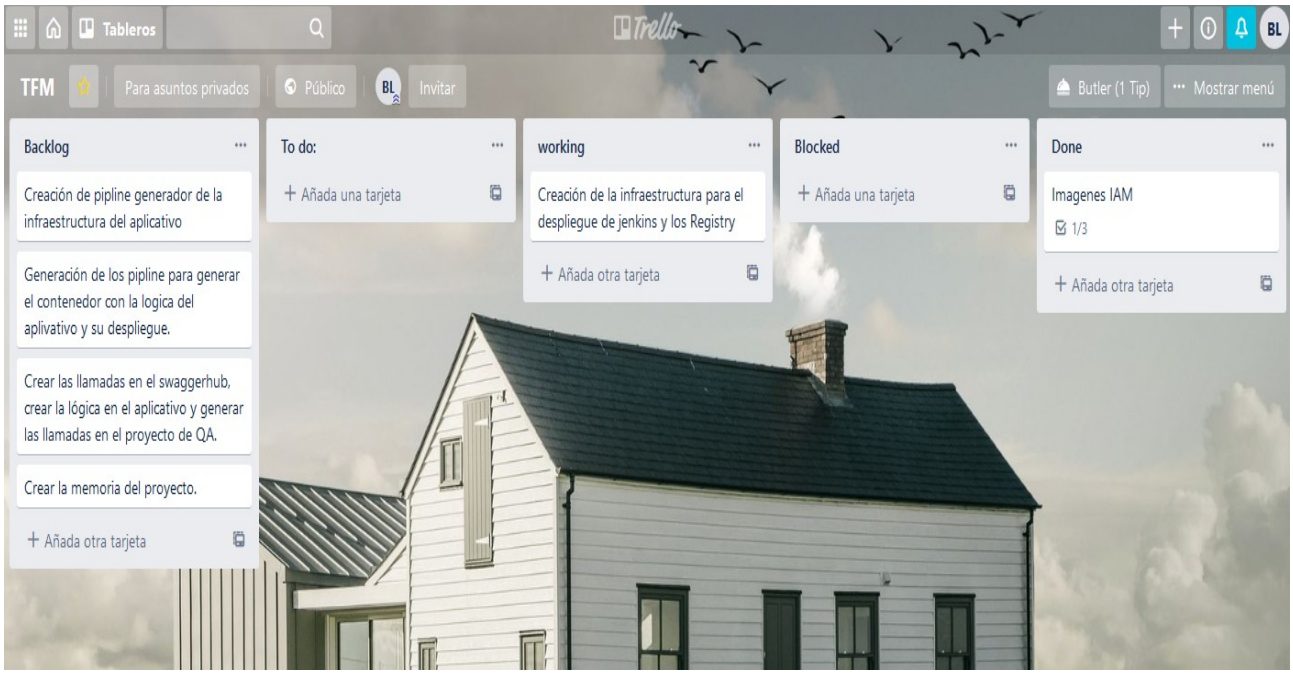

Figura 5: 3.3 Sprint 2

Aquí en la figura 6, vemos que nos encontramos en un sprint acabando la memoria en el sprint 6, con las tareas realizadas a excepción de la creación de la memoria.

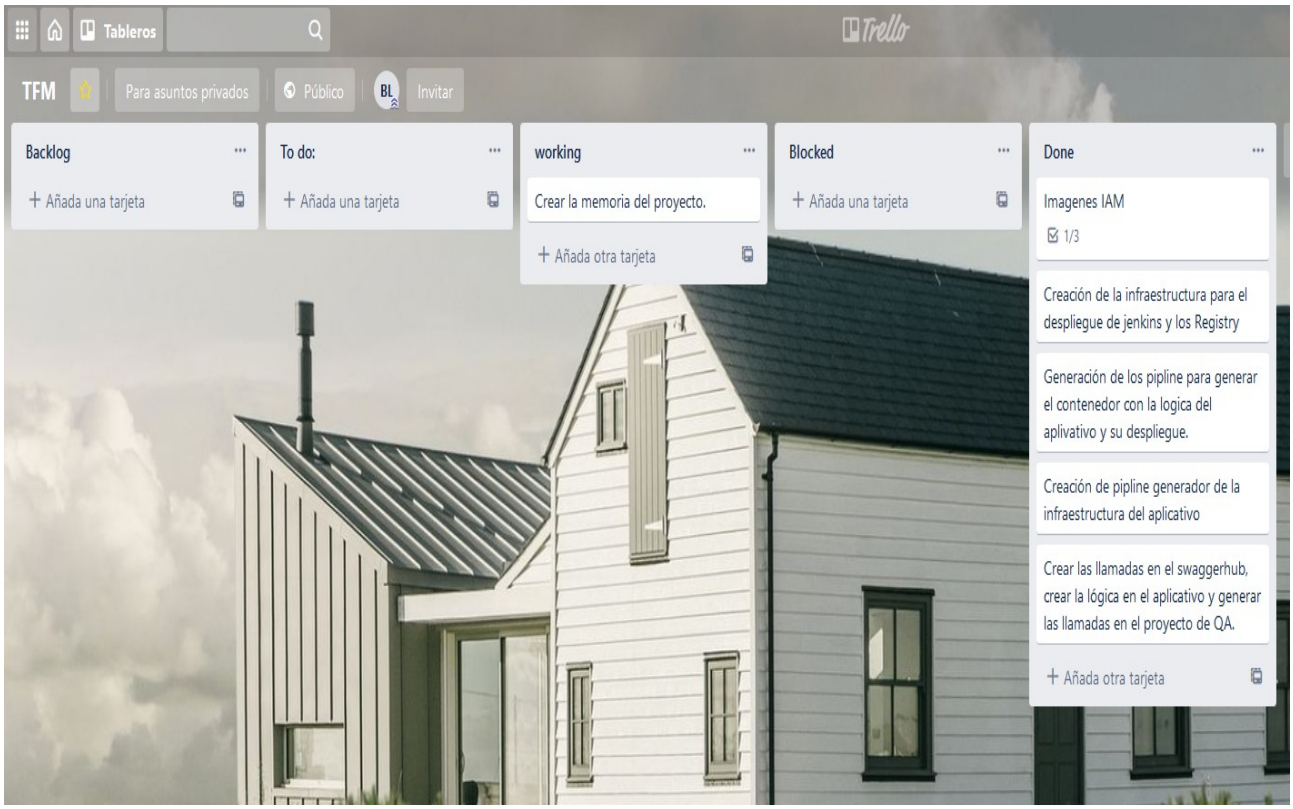

Figura 6: 3.3 Sprint 6

De esta manera iremos indicando lo que se realizado en cada sprint y cuando habremos acabado el proyecto.

4. Descripción de los automatismos

4.1 CI y CD de Infraestructura:

Como se ha de tener en este proyecto **dos entornos** desplegados en la Cloud de AWS, uno de desarrollo y otro de producción, se deberá de construir también un circuito de integración continua que genere toda la infraestructura y que mantenga los entornos. Se realizara con infraestructura como código, sera realizado con la aplicación de Terraform. La aplicación Terraform sera ejecutada a través de un Pipeline en Jenkins.

Antes de explicar el Pipeline de infraestructura de la aplicación, **la infraestructura** en la que se ejecutara **Jenkins también sera generada por infraestructura como código**, ejecutado por la aplicación de Terraform, y con todo lo necesario para desplegarse, como los **usuarios IAM** y los **Registry para contenedores de Docker**. La AMI de Jenkins estará ya generada, el repositorio es este Jenkins: https://github.com/borjaOrtizLlamas/jenkinsAMI. Jenkins se encontrara en una red VPC (Red privada virtual) aparte de la aplicación, aunque daremos posibilidad de que Jenkins sea accesible desde una IP publica. Con esto Jenkins podrá ser usado por los desarrolladores sin ningún problema. El proyecto que se encargara de desplegar la infraestructura para desplegar el proyecto esta en este repositorio: https://github.com/borjaOrtizLlamas/tfm-static-infraestructure. No debemos olvidar que se tendrá que disponer de un usuario IAM capaz de desplegar infraestructura como código y incluir sus claves en las variables de Terraform para que sea capaz de usarlo.

Ahondando más en el Pipeline de despliegue de la infraestructura, lo primero es que se debe de explicar es lo que se a de desplegar en cada entorno, encontrándose todo ello en la **misma VPC, distinta a la de jenkins**:

- **EC2**: Se deberá de desplegar dos instancias basadas en AMIS, una con la base de datos y otra con todo el ELK (Elastic Stack) para la monitorizar el aplicativo. Estarán en una subred dedicada a instancias. Aquí se explica lo que tiene cada una de las instancias:
    - Para la instancia con el ELK, es necesario instalar un ElasticSearch para guardar la configuración de Kibana y también para poder guardar y leer los logs. Logstash sera la aplicación que alimente de logs la base de datos de ElasticSearch, esta aplicación (Logstash) podrá ser llamada por FileBeats y imprimir los logs para el uso de Kibana. Necesitara estos puertos abiertos para poder acceder: 9200 para visualizar Kibana, 5044 para que se alimente a través de logstash. También tendrá

un record con la URL kibana.tfm.com para más facilidad de su acceso. No tendrá IP publica, dado que tendremos que usalas lo mínimo necesario posible, AWS tiene un limite de dar IP publicas. Estará basado en un sistema operativo Ubuntu 18.04.

- La instancia con la base de datos de nuestro aplicativo sera del tipo MongoDB, esta base de datos podrá ser accedida a través de su puerto 27017. También tendrá unrecord con la url mongo.tfm.com para más facilidad de su acceso. No tendrá IP publica. Estará basado en un sistema operativo Ubuntu 18.04.

Estas dos **AMIS ya estarán generadas por Packer y aprovisionadas por Ansible**, no tendrán su propio circuito de integración. Para generar las AMIS, se podrá encontrar los repositorios con el código de Packer y el Playbook de Ansible que se han usado para generarlas aquí:

- ELK: https://github.com/borjaOrtizLlamas/kibana_AMI
- MongoDB: https://github.com/borjaOrtizLlamas/mongoAMI

- **ECS**: Se tendrá de crear la definición de una tarea, que correspondería a la misma definición que un Pod en kubernetes. Esta Task, llamada “APIRestSmallCompany” y el sufijo según el entorno, tendrá 2 contenedores, un contenedor con la aplicación de Spring boot (https://github.com/borjaOrtizLlamas/small_comerce_api_rest_container con el dockerfile), que contiene la lógica del aplicativo, y otro contenedor con filebeat (https://github.com/borjaOrtizLlamas/beats_sidecar/ con el dockerfile), que enviara los logs generados por la aplicación de Spring boot a Logstash. Los Registry de estos contenedores serán estos:

  - Contenedor del aplicativo (Spring boot): 005269061637.dkr.ecr.eu-west-1.amazonaws.com/small_comerce_api_rest
  - Contenedor de Filebeats: http://005269061637.dkr.ecr.eu-west-1.amazonaws.com/filebeats_unir

  Se tendrá que indicar desde que puerto se podrá acceder a la aplicación, que sera 8080. También se ha de indicar el lugar en el que se dejaran los logs, para esto se compartirá un volumen entre los contenedores para que los contenedores puedan comunicarse.

  Se deberá de desplegar 2 veces esta misma Task, para que no tengamos problemas de rendimiento. Para esto sera un usado un Service, llamado “serviceApiRest” y el sufijo

según el entorno, que el encargado de ejecutar las Task. Dispondrá de 2 subredes desplegadas en 2 subzonas para que no haya problemas de ejecución si alguna subzona se cae por algún tipo de mantenimiento. Todas las Task que serán lanzadas en este Service pertenecerán a un mismo Target group, que servirá para que puedan recibir peticiones por parte del balanceador que será explicado más adelante. El Service necesitara un usuario IAM para ejecutarse. Y finalmente, este Service, se desplegara en un cluster de ECS, que tendrá el nombre de “api_rest_cluster” y el sufijo según el entorno.

- **Load Balancer**: El balanceador de AWS servirá para poder gestionar peticiones y balancear la carga entre un Target Group, que en este caso serán las Task lanzadas en el Service de ECS. También un Load Balancer puede funcionar como proxy, por lo cual el load balancer también se encargara del acceso a Kibana, dado que este Kibana no tiene una ip publica, sería imposible acceder a el dashboard con de logs, en cambio usándose de proxy no habrá ningún problema. No olvidar que este Load Balancer deberá de tener disponibilidad de acceso a todas las subredes.
- **Route53**: Es el servicio DNS de AWS. Crearemos una zona para nuestra VPC y daremos record a nuestras instancias de EC2.
- **Variables**: Se deberá de **parametrizar** campos para que sean capaces de cambiar fácilmente. Como el rango de ipds de la VPC y subredes, la zona y la subzona que usara esta VPC y subredes. El usuario IAM que sera usado para que Terraform ejecute todo el código. Claves SSH para el acceso a instancias de EC2. Estas variables estarán en un repositorio privado aparte, para darle seguridad a las claves privadas y al usuario IAM con acceso a nuestra cuenta de AWS.

Todo este proyecto de ejecutado con Terraform esta en esta https://github.com/borjaOrtizLlamas/tfm_infraestucture_generator. El Pipeline se encuentra en el mismo repositorio.

Para desplegar esto con dos entornos sé dispondrán de **dos ramas**, una llamada “develop” y otra llamada “master”. La rama “develop” tendrá la infraestructura como código de el entorno de desarrollo y la rama “master” tendrá la infraestructura de producción. El **gitflow** de este proyecto es simple, se realizara todos los commits de trabajo directamente sobre la rama de develop y cuando se haya comprobado que los cambios realizados funcionan en el entorno, se tendrá la capacidad de poder subirlos a la rama master y desplegarlos en producción

Para ejecutar Terraform por ramas se deberá de tener un proyecto en Jenkins de tipo “Multibranch

Pipeline“, con el nombre de “INFRAESTUCTURE_GENERATOR”. Este tipo de Pipeline genera un Pipeline por cada rama de un repositorio. Este Pipeline ejecutara un Jenkinsfile, este Jenkinsfile, a su vez, ejecutara Terraform y ya desplegaremos la infraestructura como código.

No olvidar que este Jenkinsfile también descarga el **repositorio con las variables**. Tendremos variables para cada entorno, las variables para desarrollo tendrán el nombre de “variables_develop.tfvars” para el entorno de desarrollo y para producción tendrá el nombre de “variables_master.tfvars”. Aquí en las variables se deberá de **parametrizar** el rango de ips de la VPC de la infraestructura, también de las subredes. También deberemos de **parametrizar** la zona en el que se desplegaran toda la infraestructura y las subzonas para las subredes. Y no olvidar las claves SSH de las maquinas para cuando queramos acceder para debugearlas.

Aquí por pasos (Steps) lo que ejecutara estos Pipeline de infraestructura:

1. “Download proyect and variables – dev”: Descargara las variables para terraform.
2. “Approve build” Se ejecutara “terraform plan” para mostrar en los logs de la ejecución del Pipeline lo que se va a desplegar si se ejecuta “terraform apply”.Ademas se dispondrá de input en el que se podrá parar la ejecución si el “terraform plan” si no ha devuelto el valor deseado. Los logs se mostraran como la figura 7 y el input como la figura 8:

```
Terraform will perform the following actions:

[1m  # aws_ecs_cluster.api_rest_cluster[0m will be created[0m[0m
[0m  [32m+[0m[0m resource "aws_ecs_cluster" "api_rest_cluster" {
      [32m+[0m [0m[1m[0marn[0m[0m  = (known after apply)
      [32m+[0m [0m[1m[0mid[0m[0m   = (known after apply)
      [32m+[0m [0m[1m[0mname[0m[0m = "api_rest_cluster-DEV"

      [32m+[0m [0msetting {
          [32m+[0m [0m[1m[0mname[0m[0m  = (known after apply)
          [32m+[0m [0m[1m[0mvalue[0m[0m = (known after apply)
        }
    }

[1m  # aws_ecs_service.service_for_api[0m will be created[0m[0m
[0m  [32m+[0m[0m resource "aws_ecs_service" "service_for_api" {
      [32m+[0m [0m[1m[0mcluster[0m[0m                          = (known after apply)
```

Figura 7: 4.1 Logs de terraform plan

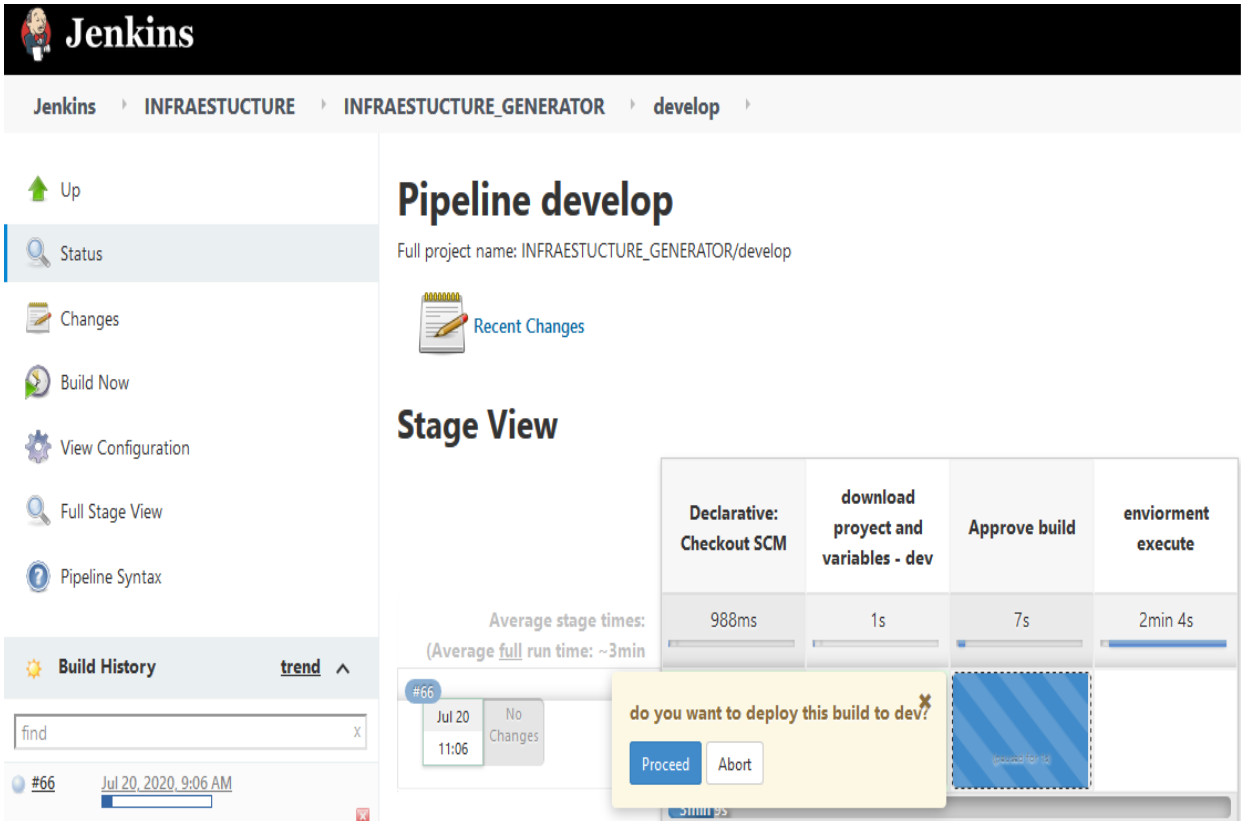

Figura 8: 4.1 Pregunta de creación de infraestructura

3. Se ejecuta “terraform apply” si aceptamos el input y se despliega el cambio.

En la figura 9 se muestra una figura con todos los steps pasados:

Figura 9: 4.1 Steps del Pipeline INFRAESTUCTURE_GENERATOR

En la figura 10 se muestra una figura de la infraestructura generada a través de este Pipeline:

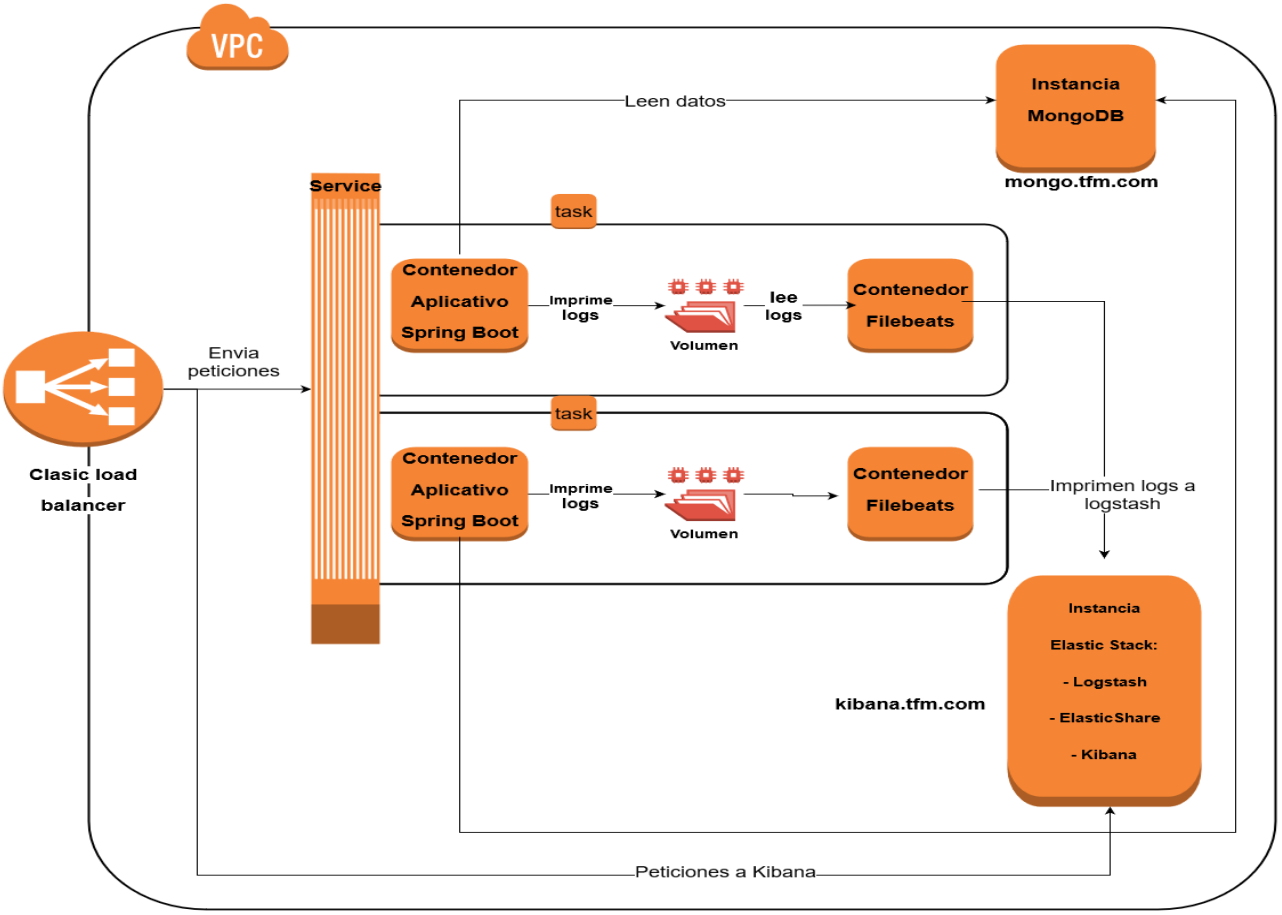

Figura 10: 4.1 Infraestructura generada por el Pipeline INFRAESTUCTURE_GENERATOR

4.2 CI del contenedor con el aplicativo:

En este proyecto, se deberá de tener un contenedor desplegado con la lógica del aplicativo. Este contenedor tiene ejecutándose un proyecto de Spring boot, que nos provee de un servidor Tomcat que despliega la aplicación.

El proyecto de Spring boot esta situado en este repositorio: https://github.com/borjaOrtizLlamas/small_commerce_api_rest. En este proyecto se seguirá un **Gitflow. En** el que las historias se realizaran en una rama con el prefijo feature, seguidas de una barra "/" y el nombre de la historia asignada el el tello, estas ramas nacerán de la rama de desarrollo. Cuando la rama con el prefijo feature este con el trabajo completo podrá mergearse con la rama develop en el que se comprobara que funciona con el trabajo de los demás compañeros, en definitiva, sera una rama el la que se integrara trabajo. En este punto, si con una tarea ya entregada surge un error, se abrirá una rama con el prefijo bugfix , seguidas de una barra "/" y el nombre de la historia asignada en

tello, en el que se volverá a abrir en la historia que se haya fallado. En este punto, con ya la rama develop funcionando en el entorno de desarrollo, sé podría pasar el código a master y ya poner este código a funcionar en el entorno de producción. Si surge un error en este punto, en producción, se debería de crear una rama con el prefijo hotfix, que nacerá de master en este caso, y ya solucionado este problema, la rama se mergeara en el mismo master, no se deberá olvidar después de mergear estos cambios en la rama de desarrollo.

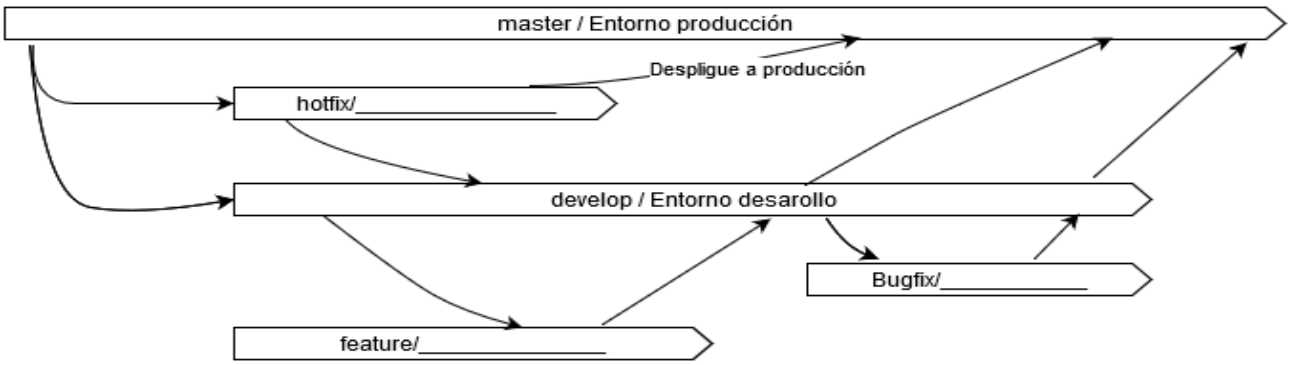

Figura 11: 4.2 Gitflow del aplicativo

El CI y CD de la aplicación dispondrá de un un proyecto en Jenkins de tipo “Multibranch Pipeline“ llamado “SMALL_COMMERCE_API_REST_BUILD” y otro pipeline, que se llamara “SMALL_COMMERCE_API_REST_DEPLOY”. Usaremos Jenkinsfile[1] para la lógica de los de los pipeline, estos usan grooby en su ejecución.

“SMALL_ COMMERCE_API_REST_BUILD” al ser un Pipeline de tipo Multibranch se tendrá la capacidad de que se vayan añadiendo ramas al repositorio en el que se encuentra el proyecto de Spring boot y que se vayan también reflejando en este pipeline, añadiendo Pipelines relacionados con cada rama. Quedaría de esta manera con una rama master, develop, y feaure/update-flow:

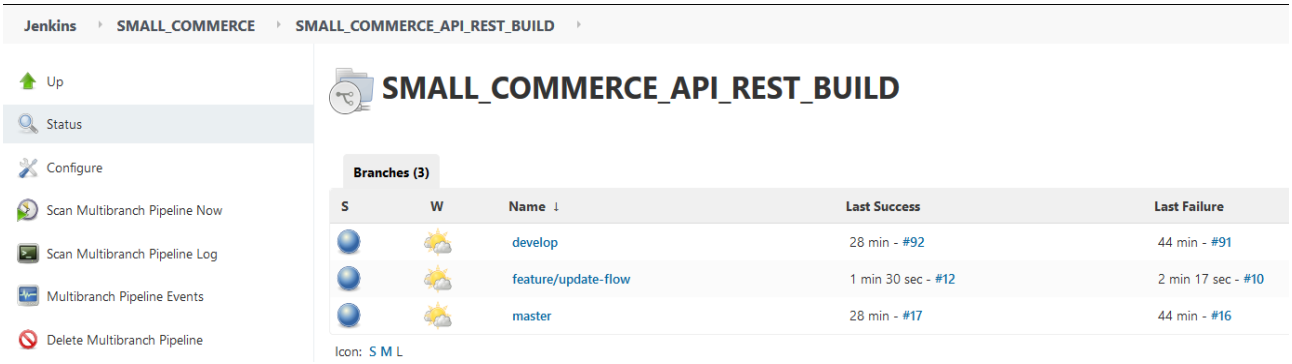

Figura 12: 4.2 Ramas del proyecto multipipline SMALL_COMMERCE_API_REST_BUILD

---

[1] https://www.jenkins.io/doc/book/pipeline/jenkinsfile/

De esta manera se podrá ejecutar cada rama de manera simple. El Jenkinsfile de estos Pipeline se encuentran en la rama del repositorio del aplicativo. Los steps de este Pipeline que se ha de realizar son estos:

1. "*Build proyect with Junit*": Con maven se realizá una instalación del proyecto en el que se ejecutarán **test unitarios**.

2. "*Docker – build*": Con Git se descargara el repositorio que tiene el contenedor, el repositorio tiene este enlace: https://github.com/borjaOrtizLlamas/small_comerce_api_rest_container, siempre se ejecutara en la rama de master. Después, ya descargado el Dockerfile, se procederá a crear el contenedor, que necesitara el .jar generado por Spring boot. Este contenedor sera tageado con el numero de la build del Pipeline en el que se ha ejecutado, un guión y la rama, si la rama tiene una barra de tipo "/" la convertira en guion. Por ejemplo: "1-feature-tarea" o "2-develop". Con excepción de la rama "master" que tageara el contenedor con el numero de la build del job, el guión y el prefijo pro, además que lo añadirá con el ultimo latest. Por ejemplo: "1-pro". Todavía en este step del job, no será realizado el comando de "docker push" al Registry, sera realizado después de las pruebas funcionales. También tendremos que tener en cuenta que a producción solo deberán pasar versiones productivas y no de pruebas. Por ello, solo los contenedores tageados con la rama master deberán de ser capaces de llegar al entorno de producción.

3. "*Docker – test*": Se añadirán también **pruebas funcionales**. Se **desplegará el contenedor anteriormente generado** dentro de docker-compose. En Docker-compose [1] también se ha de desplegar un contenedor con MongoDB para que las pruebas realizadas sean completamente reales, recuérdese que con Docker-compose el naming que se le añada al contenedor podrá ser usado también como DNS. Este archivo se encuentra ubicado en el mismo repositorio que el contenedor Docker .Una vez desplegado con Docker-compose el proyecto, se deberá ejecutar el proyecto que prueba esta aplicación, se encuentra en un repositorio aparte, que es este: https://github.com/borjaOrtizLlamas/small_commerce_api_rest_test, si la rama ejecutada es master este proyecto se ejecutara en rama master, en cambio si ejecutar otra rama siempre se ejecutara este proyecto en la rama de develop. Se basa en lanzar peticiones con un main de java. Este proyecto ejecutara lo que se indique en el swagger hub (indicado en el punto 2.1). Si funcionara este test, será que los test han pasado por parte del equipo de QA.

---

[1] https://docs.docker.com/compose/

4. "*Login*" : Logeara con AWS en Registry para que podamos hacer push con Docker. Usa aws-cli para este objetivo.

5. "Push to repository": Terminara de hacer push al Registry de AWS. Recordamos que tiene el tag latest, que dejara de tener el ultimo lo tuviera, también se puseara el tag con el numero de la build y el nombre de la rama. Finalmente imprime el tag que usaremos para desplegar el contenedor en la infraestructura, para desplegarlo usaremos el job con nombre "SMALL_COMMERCE_API_REST_DEPLOY", que pide como input el tag del contenedor que va a desplegar.

Aquí se muestra la figura 12 de los steps del Pipeline:

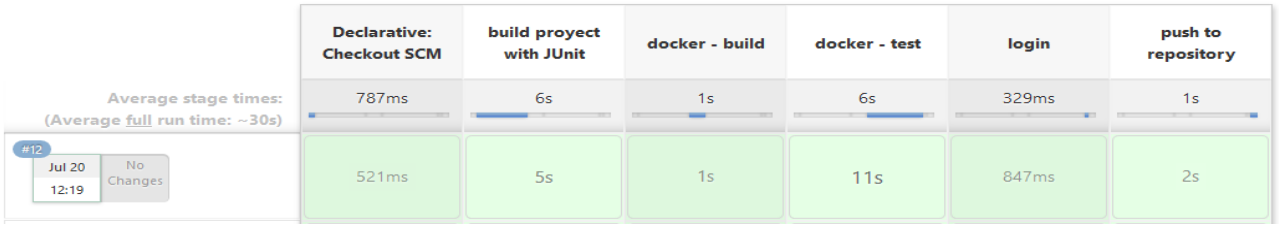

Figura 13: 4.2 Steps de un Pipeline de SMALL_COMMERCE_API_REST_BUILD

Ahora para realizar el deploy de este contenedor en la infraestructura, tendremos otro Pipeline anteriormente nombrado, llamado "SMALL_COMMERCE_API_REST_DEPLOY". Este job se encargara del despliegue del contenedor del aplicativo, en el entorno de desarrollo y producción, en el orden que han sido escritos. Este Pipeline se encuentra en este repositorio:https://github.com/borjaOrtizLlamas/deploy_ECS y siempre se ejecutara en la rama master. Aquí los steps del Pipeline:

1. Se introducirá como variable de entrada el tag queremos desplegar por los entornos:

Figura 14: 4.2 Indicación de tag que se desplegara

2. "Dowload variables": Se descargaran variables para que se pueda ejecutar Terraform. Con las credenciales del usuario IAM que tenga permisos de ejecutar infraestructura como código.

3. “Build task”: Se deberá de construir el nuevo Task de ECS para desplegarlo con el nuevo contenedor, lo haremos con la aplicación de terraform, construyendo la Task con infraestructura como código.

4. “Deploy to dev”: Este step se encargara de realizar el “rolling update”[1] del la Task en el entorno de desarrollo, primero tendrá que ser preguntado si el desarrollador verdaderamente quiere desplegar el contenedor indicado. Esto se realizara con el comando de aws-cli, que se encarga de ejecutar acciones sobre la Cloud de AWS. En el comando, que deberá de incluir los nombres del clúster, del servicio que se va a modificar y de la Task que queremos que pase a su ultima versión. El nombre del clúster cambia por entorno con su sufijo así como las Task y los Service, nos basaremos en eso para solo cambiar en el entorno de desarrollo. Este sera el comando que se ejecuta en este step el rolling Update:

   “aws ecs update-service --cluster api_rest_cluster-DEV --service serviceApiRest-DEV --task-definition APIRestSmallCompany-DEV”

5. “Deploy to pro”: Realiza lo mismo que el paso anterior pero en el entorno de producción. También se deberá de incluir la excepción de que solamente llegaran al entorno de producción los contenedores con el sufijo -pro, de esta manera el final de git-flow acabara en la rama master. El comando que será lanzado para el entorno de producción será este, con los namigns de producción:

   “aws ecs update-service --cluster api_rest_cluster-PRO --service serviceApiRest-PRO --task-definition APIRestSmallCompany-PRO”

Aquí una figura con los Steps realizados por este Pipeline:

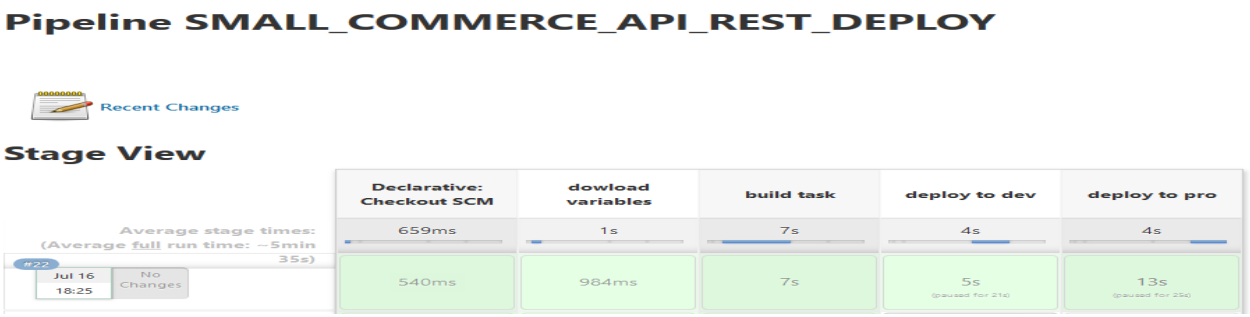

Figura 15: 4.2 Steps del Pipeline SMALL_COMMERCE_API_REST_DEPLOY

---

[1] https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/

Finalmente, esta figura muestra el que será el UML con todo el flujo que seguirá esta integración continua para la lógica del aplicativo:

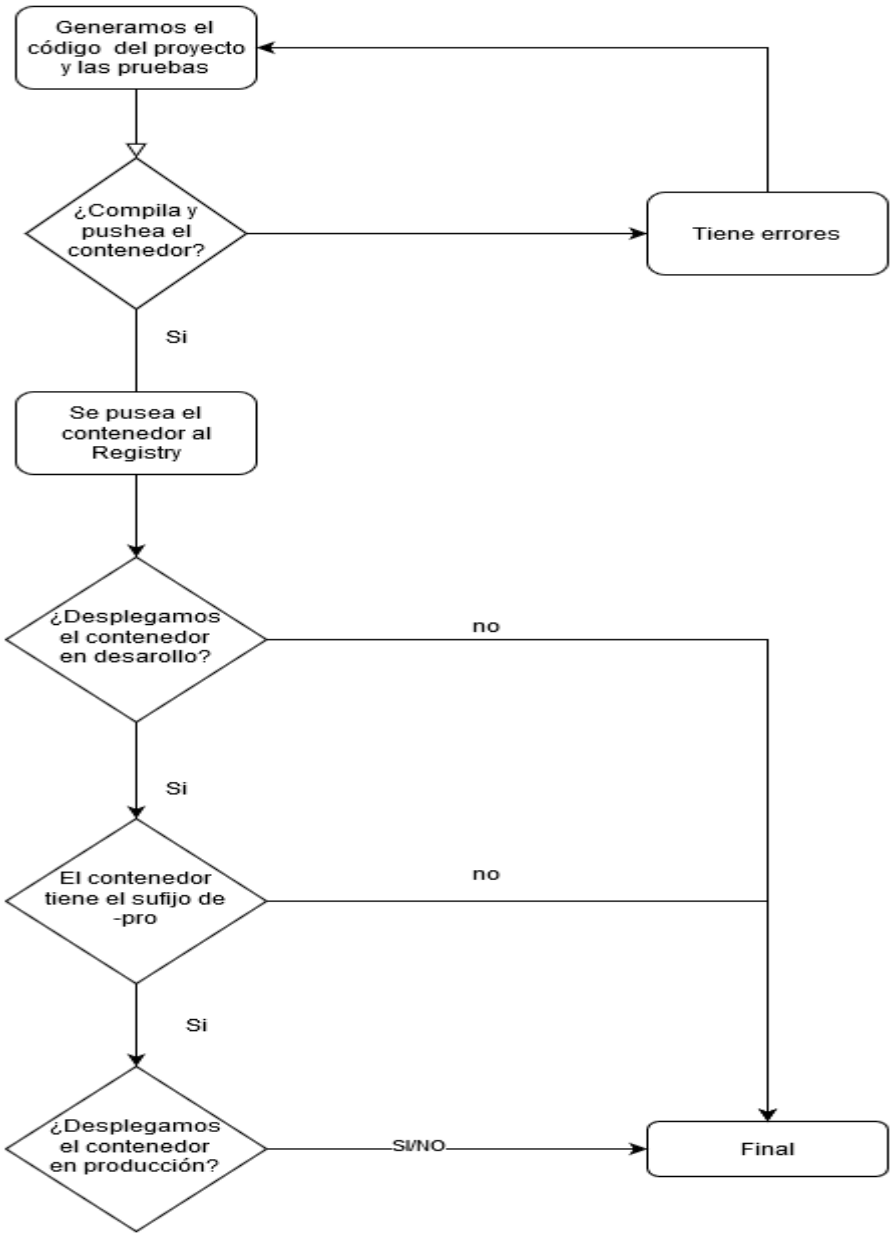

Figura 16: 4.2 UML de despliegue del aplicativo

4.3 CI de Filebeats

Para realizar esta integración continua se deberá de generar un Pipeline, en el que se tendrá la

capacidad de generar y pusear el contenedor de filebeats. EL nombre de este Pipeline será este step “BEATS_SIDECAR_LOGS_BUILD”. El Jenkinsfile se encuentra en el mismo lugar que el Dockerfile, este es el enlace: https://github.com/borjaOrtizLlamas/beats_sidecar.

Este contenedor solo sera necesario pusearlo en el Registry una vez para que cuando se ejecute el Pipeline de infraestructura (“INFRAESTUCTURE_GENERATOR”) tenga el contenedor disponible.

Los steps que se tendrán serán estos:

1. “*Login*”: Logeara con AWS en Registry para que podamos hacer push con Docker.
2. “Build” Generamos el contenedor, dispondrá de la configuración de filebeats, que enviara logs a logstash.
3. “Push to repository”: Hace push contra el repositorio realizando el tageo o de latest y la ultima versión de la build.

Aquí una figura con los Steps realizados por este Pipeline:

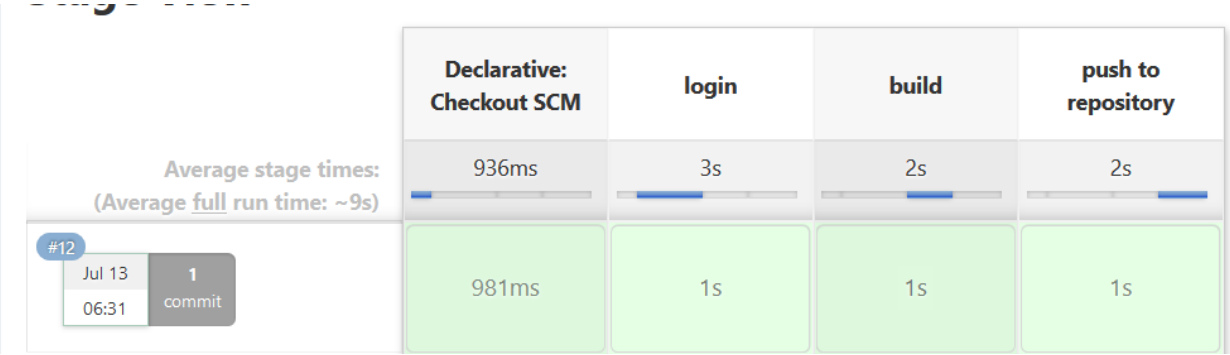

Figura 17: 4.3 Steps del Pipeline BEATS_SIDECAR_LOGS_BUILD

5. Conclusiones

En este proyecto se ha conseguido tener una aplicación capaz de proveer a las pequeñas empresas de la funcionalidad de añadir clientes con productos que hayan comprado en un API Rest con infraestructura Cloud, con circuitos de integración continua y despliegue continuo. Los circuitos de integración además con pruebas funcionales capaces de probar la aplicación antes de su despliegue. Con una infraestructura monitorizada, en Cloud, creada por infraestructura como código lo que permite un despliegue rápido de los entornos con la aplicación , la infraestructura como código permite tener la capacidad de mejora con rapidez y tendremos mucha calidad en su despliegue. La monitorización ofrecida por ELK además es capaz monitorizar la aplicación con mucha fiabilidad.

Con este trabajo realizado se ha conseguido una aplicación robusta, con un entornos de pruebas (desarrollo) y otro de producción, capaz de evolucionar rápidamente, con capacidad de salir a producción con la calidad necesaria para que los clientes estén tranquilos con este proyecto. Esto es lo más importante de este proyecto, al estar realizado en una cultura **DevOps** se conseguirá tener toda la automatización posible y por lo tanto que el proyecto sea capaz de llegar a tener toda la funcionalidad que pida el cliente lo más rápido posible y con una alta calidad.

Ademas, como se ha seguido la cultura **DevOps**, se dispondrá de un circuito de integración en el que el equipo encargado de la infraestructura Cloud del proyecto y otro circuito para el desarrollo, de esa manera se podrá trabajar con la mayor independencia un equipo del otro.

También al ser usada una metodología de trabajo SCUM, los equipos se organizaran de la manera más rápida. Por no contar que al ser esta metodología conocida en muchísimas empresas, cuando se añada personal en el equipo se adaptaran a la manera de trabajo de manera.

Haciendo una comparación con un proyecto ejecutado sin cultura **DevOps**, sin tener infraestructura como código implementada con Terraform, sin tener circuitos de integración, se podría llegar a poner en producción este proyecto de una manera más rápida, pero en cambio se perdería la capacidad de evolucionarla con calidad. También sin infraestructura como código no seriamos capaces de desplegar toda la infraestructura en minutos y ademas con replicas. A corto plazo es mas largo adaptarse a esta tecnología, pero de esta forma se tendrá el ciclo de vida muchísimo más largo en la aplicación.

La conclusión final del proyecto es que seremos capaces de tener un largo ciclo de vida, siendo capaces de desplegar cambios velozmente y con calidad, clave de la cultura **DevOps**.

6. Posibles mejoras

Un proyecto DevOps siempre tiene que estar capacitado para futuras mejoras y tener un largo ciclo de vida y tener claro que en el mundo de la informática siempre hay avances disponibles. Por esto sera detallado en este punto una parte de las mejoras que se pueden realizar en este momento al proyecto:

1. Circuito de integración para AMIs:

   Se deberá de tener en este proyecto un circuito de integración y despliegue para las tres Amis del proyecto, la de ELK, Jenkins y MongoDB.

2. Plugin de despliegue:

   En vez de usar el Pipeline “SMALL_COMMERCE_API_REST_DEPLOY” se deberá de usar el plugin llamado “promoted builds plugin”[1] que consigue una mayor claridad para los despliegues al ser más visual en lo que se encuentra desplegado y en lo que no.

3. Mejorar los flujos de la aplicación:

   Deberían de haber más flujos en la aplicación como posibilidad de un CRUD en los productos, seguridad de los usuarios con Spring Security, etc.

4. Añadir Cucumber[2] para QA:

   Tiene mejor funcionamiento que el proyecto realizado para pruebas y es especifico para QA.

5. Mejorar la Monitorización:

   Podríamos mejorar la monitorización como de que países llegan los clientes, cuanto tiempo pasan en nuestra página, capacidad de hacer estadísticas con nuestros datos,etc.

6. Añadir Sonar[3]:

   Podríamos añadir Sonar para revisar la calidad del código, tanto el de la aplicación, como el código QA.

---

[1]: https://plugins.jenkins.io/promoted-builds/

[2]: https://cucumber.io/

[3]: https://www.sonarqube.org/

7. Añadir una URL publica para la API.

   Podríamos añadir una url pública para nuestro clúster, que sea fija y publica en los DNS, tiene un precio alrededor de unos 14$ y servirá para añadir llamar de manera sencilla a la API Rest desde cualquier aplicación.

8. Añadir un circuito con el que se fuera capaz de añadir más APIs Rest:

   Podríamos tener un proyecto en el que podamos añadir más Apis indicando el repositorio que usara para la compilación, con la misma tecnología que la usada en el Pipeline de “SMALL_COMMERCE_API_REST_BUILD”. Con ello el proyecto sera capaces de tener más ontenedores con funcionalidades distintas y así el proyecto podrá de parecerse lo más osible a un ERP.

9. Disponer Artifactory[1]:

   Deberíamos de ser capaces de tener Artifactory para el guardado de los componentes del API Rest y tener todas las versiones guardadas. De esta manera el proyecto sera capaz de desplegar mucho más velozmente en la lógica y los nuevos proyectos que se vayan a generar.

10. Que los usuarios de Jenkins estén integrados con IAM de AWS y así cada nuevo desarrollador pueda disponer de su usuario con sus roles específicos.

11. Autoescalado de Pods:

    Deberíamos de ser capaces de tener autoescalado de pods de manera automática para mejorar el rendimiento en puntos altos de demanda y menor gasto cuando no haya muchas peticiones.

La capacidad de mejora continua en cualquier proyecto **DevOps** es constante y se podrían poner muchísimas mejoras pero creemos que con estas el proyecto debería de ser capaz de tener una amplia capacidad.

---

[1] https://jfrog.com/artifactory/

7. Anexos

Manual GITFlow

https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow

Manual Ubuntu 18.04

https://help.ubuntu.com/

Jenkins

https://www.jenkins.io/doc/book/pipeline/jenkinsfile/

Creación de Figuras

https://app.diagrams.net/

Terraform

https://www.terraform.io/

aws-cli

https://aws.amazon.com/es/cli/

AWS Cloud

https://docs.aws.amazon.com/