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

Curso académico 2020/2021

Trabajo de Fin de Máster

Plataforma de compras con microservicios y CI/CD en cluster k8s en la nube

Autor: Álvaro Martín Martín
Tutor: Micael Gallego Carrillo


Universidad
Rey Juan Carlos

Resumen

En el presente documento se pretende plasmar el trabajo abordado durante la realización del TFM del máster, en el cual se ha querido poner en práctica las técnicas y tecnologías vistas durante el curso:

- Patrones de diseño / Arquitectura hexagonal
- DDD / CQRS
- Diferentes tecnologías de comunicación: REST, colas.
- Testing unitario, de integración, de componente, end to end.
- CI / CD
- Docker y k8s

Para ello, la memoria se ha estructurado en los siguientes apartados:

1. **Objetivos**: descripción de las metas a conseguir mediante el desarrollo del TFM.
2. **Contexto y funcionamiento del proyecto**: Indica brevemente en qué consiste el proyecto, las piezas que lo componen, los entornos disponibles y los flujos de despliegue.
3. **Desarrollo del TFM**: se indican tantos los pasos como las tecnologías usadas en detalle, para facilitar la comprensión del código.
4. **Conclusiones**: se indica los puntos que se han sacado en claro tras la finalización del máster y del TFM.
5. **Mejoras a futuro**: listado de mejoras técnicas a realizar sobre el proyecto entregado.
6. **Bibliografía**: recopilación del material usado durante la implementación del TFM.

1. Objetivos

Los objetivos que se desean cubrir en este TFM son:

- Implementar un sistema de microservicios independientes y reusables, poniendo en práctica todo lo aprendido durante el curso:
    - API de usuarios: permitirá registrarse y autenticarse a usuarios y administradores, así como obtener el detalle y añadir saldo a los usuarios.
    - API de productos: permite gestionar los productos y su stock.
    - API de compras: permite a los usuarios registrados añadir productos al carrito y realizar la compra de éstos.
- Que dicho sistema sea desplegable en cualquier cluster de k8s.
- Que disponga de una batería de tests y documentación de calidad.
- Proporcionar un mecanismo automatizado de CI/CD que asegure la calidad y el funcionamiento de los servicios de cara a desplegar en la nube, haciendo uso de github actions.
- Hacer uso de servicios de AWS.
- Trabajar con diferentes tipos de BD: SQL y NoSQL.
- Hacer uso de diferentes patrones: mensajería, SAGA, patrón estrategia, etc.

2. Contexto y funcionamiento del proyecto

Como se comenta anteriormente, para la realización del TFM se ha optado por la implementación del backend de una plataforma de compra de productos en la nube usando arquitectura de microservicios y un diseño basado en eventos.

Los micros que componen la aplicación serán:

- Servicio usuarios.
- Servicio productos.
- Servicio de compra.
- API Gateway (Único punto de entrada y enrutado para todos los micros).

La plataforma funciona de la siguiente forma:

1. Los usuarios deben registrarse y autenticarse para poder realizar compras.
2. Se permite también el alta y autenticación de administradores, para poder gestionar los productos en venta en la plataforma.
3. Un usuario sólo puede disponer de un carrito de la compra no completado de forma simultánea.
4. Los usuarios autenticados pueden ver el listado de productos disponibles y añadirlos al carrito de la compra.
5. Una vez den por finalizado el carrito, se generará un pedido, que se validará y, si todo va bien, se retira el stock, se resta el saldo al usuario y se dará por cerrado. En caso contrario se liberará el stock y no se restará el saldo al usuario (patrón SAGA).

Se dispone de dos entornos para los cuales se usan diferentes namespaces de Okteto: PRE y PRO.

- **PRE**: cada vez que se realiza un push en la (única) rama main de los repositorios de despliega en este entorno. Es el entorno de pruebas integradas donde se valida que se dispone de la funcionalidad correcta.
  En este entorno se hace uso de contenedores con volúmenes para las BBDD. Un diagrama arquitectónico de este entorno puede verse en la Figura 1.
- **PRO**: una vez hecha la validación en PRE, se etiqueta con una release, que desencadena el despliegue automático en PRO. En este entorno las BBDD son recursos proporcionados por AWS (Ver Figura 2).

El flujo de CI/CD usado es el siguiente (ver Figura 3):

1. Cada vez que se realiza un commit en local se lanza un githook que valida el formato del código y lanza los tests unitarios. En caso de fallar no permite realizar el commit.
2. Al realizar el push se lanza otro githook que lanzan los tests integrados. De igual forma si los tests no pasan, no se permite subir el código al repositorio.
3. Si se realizó de forma correcta el push en la rama main, se dispara el workflow de github actions que vuelve a validar el formato del código, lanza todos los tests,

genera la imagen docker con la etiqueta ‘trunk’ y despliega la imágen en el clúster de k8s (Okteto) usando el namespace de PRE.

4. En caso de considerar que se cumple la funcionalidad deseada, se añade el tag y se hace push de él para generar la release y nuevamente se dispara el workflow. Dicho workflow comprueba que el tag sea correcto, publica los artefactos, genera la imagen con la etiqueta de la release y latest, y despliega la imagen en el clúster de okteto en el namespace de PRO.

Toda la información está detallada en los README de los servicios.

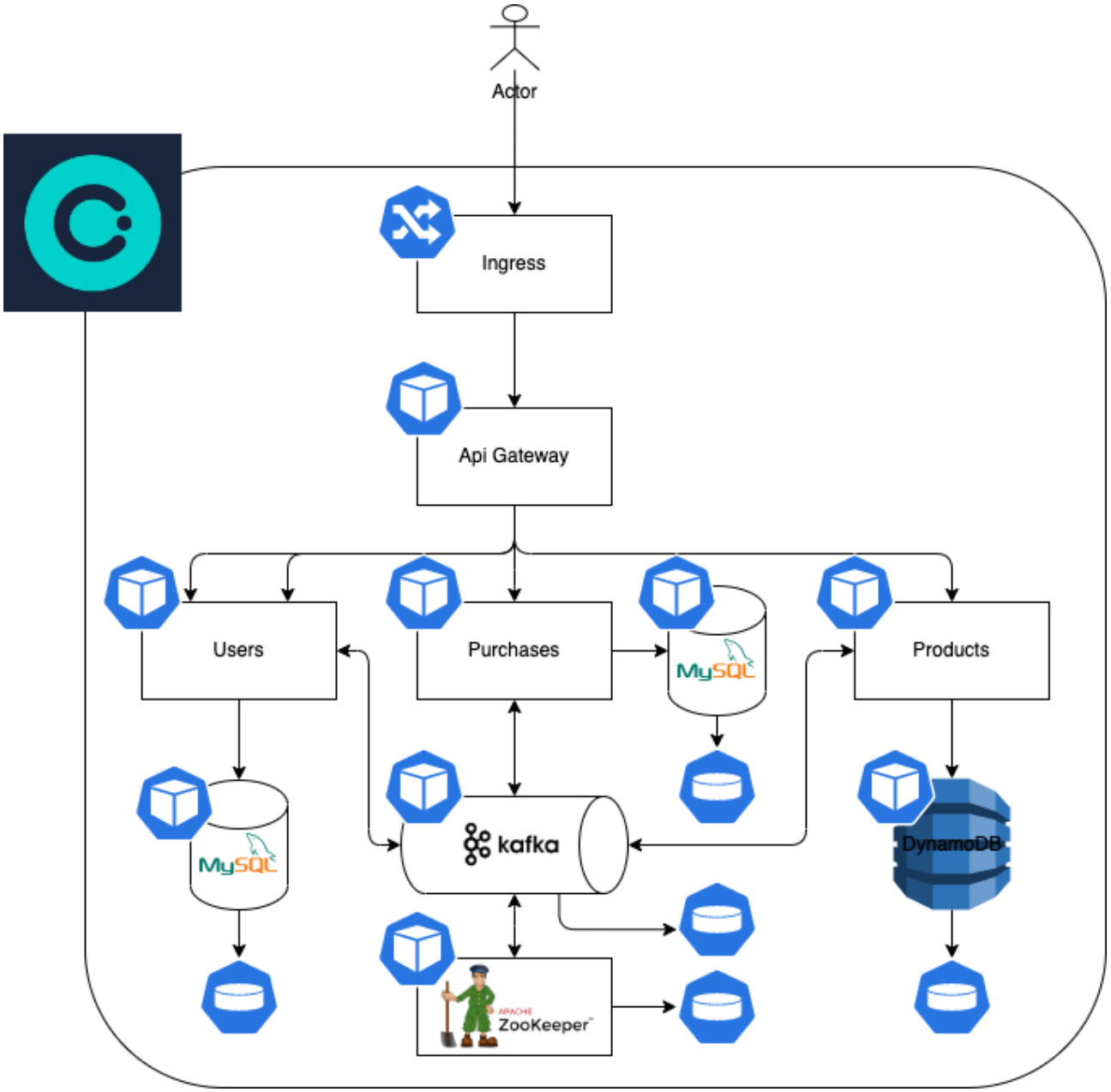

Figura 1: Arquitectura del entorno de PRE

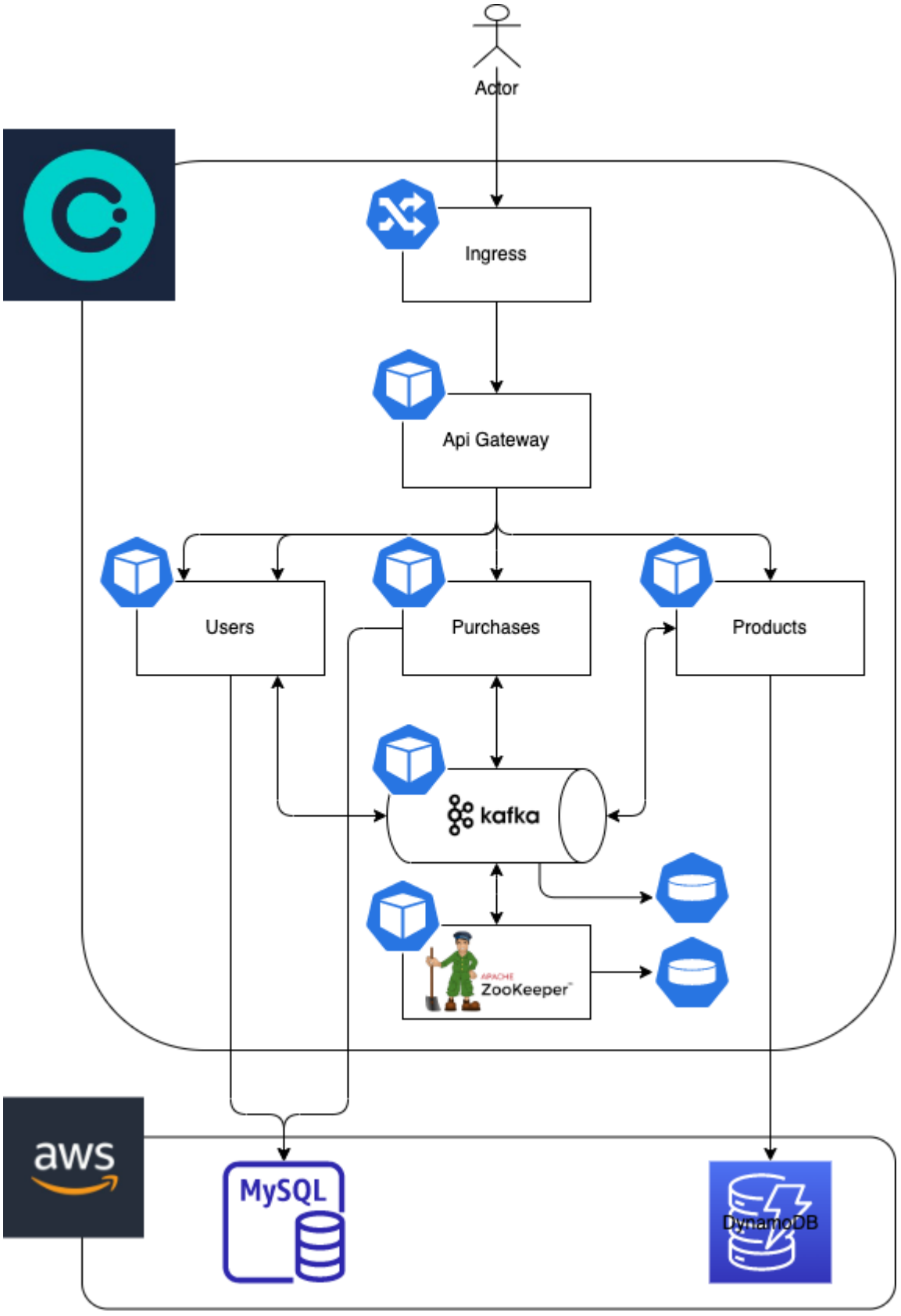

Figura 2: Arquitectura del entorno de PRO

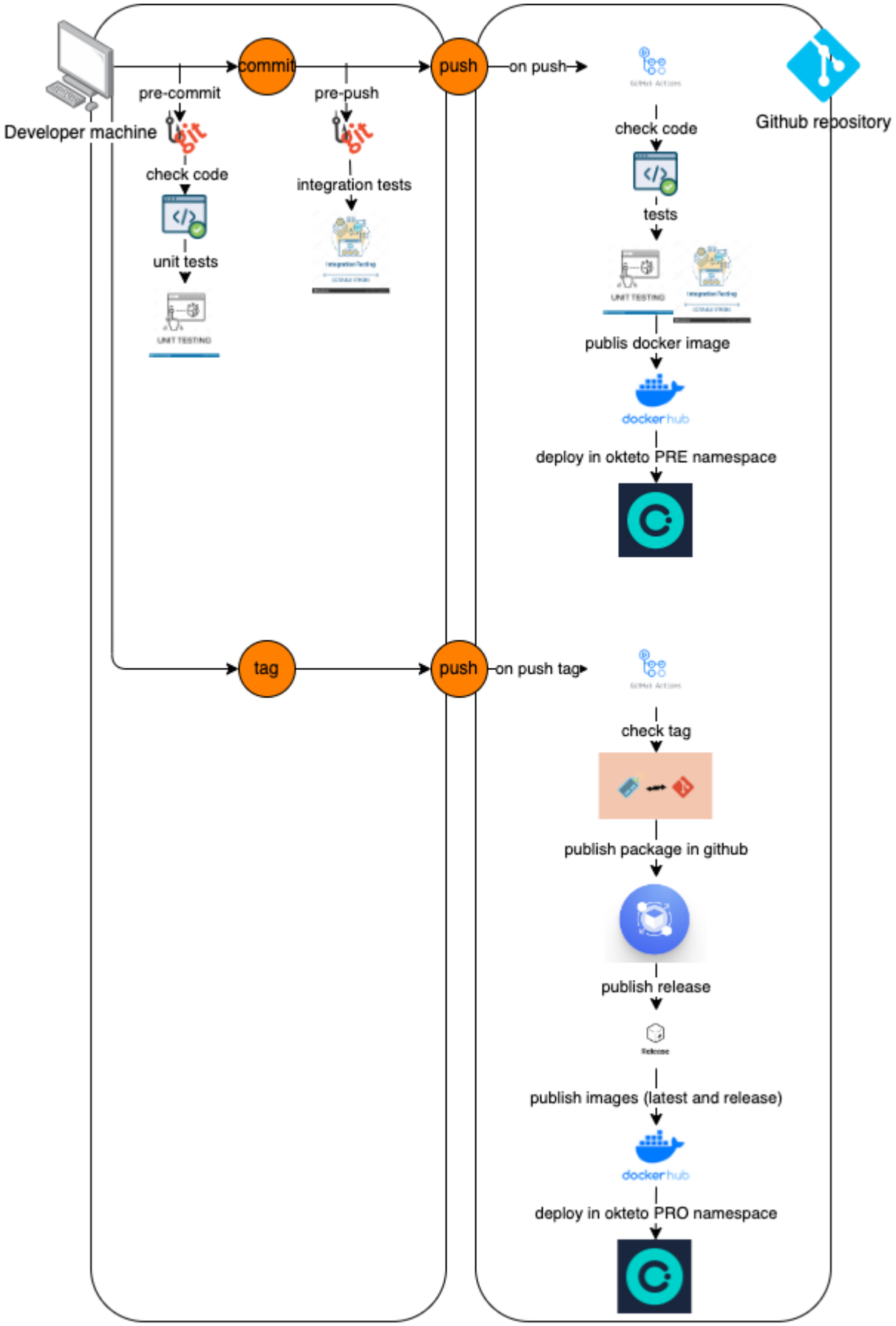

Figura 3: Flujo de CI/CD

3. Desarrollo del TFM

Se ha llevado a cabo el desarrollo de los micros de la siguiente forma:

- Servicio usuarios:
  - Endpoints API Rest:
    - Usuarios (USER):
      - Registro usuario.
      - Autenticación (Token JWT).
      - Obtener información del usuario.
      - Añadir saldo al usuario.
    - Adminstradores (ADMIN):
      - Registro administrador.
      - Autenticación (Token JWT).
  - Mensajería
    - Consumo de eventos de tópicos de Kafka para la validación y retención del saldo.
    - Producción de eventos en tópicos de Kafka para confirmar el flujo de compra.
  - Arq por capas
  - Desarrollo en node y express.
  - BD Amazon MySQL.
- Servicio productos:
  - Endpoint API Rest:
    - CRUD productos (todos autenticado como ADMIN, lectura también como USER)
  - Mensajería
    - Consumo de eventos de tópicos de Kafka para la reserva/bloqueo de stock de productos.
    - Producción de eventos en tópicos de Kafka para confirmar el flujo de compra.
  - Arq por capas.
  - Desarrollo node y express
  - BD Amazon DynamoDB.
- Servicio de compra:
  - Endpoints API REST (autenticado como USER):
    - Crear carro.
    - Asignar cantidad de productos al carro (añadir/quitar).
    - Eliminar producto del carro.
    - Eliminar carro.
    - Comprar (completar carro y generar pedido).
  - Cuando se finalice un carrito de la compra, se hará uso del patrón SAGA, con envío de eventos a los servicios de productos y usuarios para bloquear

stock y retener dinero. En caso de error se deberá deshacer las operaciones correspondientes.
    - Arq hexagonal.
    - CQRS:
        - Se implementan comandos para este servicio.
    - Event sourcing:
        - Se manda evento a un tópico de Kafka que el servicio recibirá y se encargará de procesar.
    - Desarrollo en Spring Boot.
    - BD MySQL
        - Modelado de campos de tipo JSON (items del carrito, errores en el pedido de compra), de esta forma se hace uso de BD multimodelo.
- API Gateway:
    - Único punto de entrada y enrutado para todos los micros.
    - Desarrollo en Spring Boot
    - CDCT.

Cada uno de esos micros tiene su propio repositorio en github. Además se dispone de otro repositorio para la infraestructura, en la que habrá un cloudFormation para el despliegue de los recursos necesarios de Amazon (BBDD).

Se han seguido las siguientes pautas comunes en cada uno de los micros:

- Se ha aplicado DDD para definir cada uno de los dominios/micros.
- Implementación siguiendo el enfoque API First haciendo uso de las buenas prácticas. Cada micro definirá primero en la carpeta API del repo del proyecto el contrato de los APIS. Se partirá de dicho contrato para el desarrollo. De esta forma, posibles clientes tienen claro el contrato a seguir.
- Se han usado varios lenguajes y frameworks para la implementación de los microservicios: nodejs, express, Java 11, Spring Boot.
- Todos los micros dispondrán de un certificado y trabajarań con https.
- Todos los repositorios de los micros cuentan con una carpeta postman que contiene una colección que permite validar los flujos de cada una de las APIs en los diferentes entornos (local, PRE y PRO). En el caso del API Gateway, dicha colección prueba de forma conjunta todos los flujos:
    - Alta, autenticación de usuario y añadir saldo.
    - Alta y autenticación de administrador.
    - Alta, modificación y consulta de productos.
    - Alta de carrito, añadir items, completar el carrito.
- Todos los micros cumplirán la pirámide de tests, con sus tests unitarios y de integración, en caso de ser necesario se usarán test containers. Las pruebas se lanzarán de forma automática durante el flujo de CI/CD.
- Se hará uso de CDCT para los tests y dependencias con otros micros (en este caso sólo api gateway)
- Las imágenes de Docker de las app Java se realizarán mediante jib.

- Los Dockerfile se realizarán haciendo uso de las buenas prácticas y usando Dockerfile linterns.
- Las aplicaciones implementarán espera de dependencias de otro servicio evitando el uso de scripts como wait-for-it y otras estrategias.
- Todos los micros cuentan con un script para facilitar el desarrollo en local (*dockerize.sh*), que permite dockerizar y levantar la aplicación como un contenedor docker con sus dependencias necesarias (docker-compose).
- Se seguirán las buenas prácticas y se hará uso de herramientas[1] para asegurar que el cluster de k8s está securizado.
- Se hace uso de charts de helm para reutilizarlos en el despliegue de los micros en los diferentes clusters de cada entorno.
- Se hará uso de Github como repositorio.
- Se han desarrollado siguiendo el modelo de TBD (Trunk-based development), teniendo en cuenta buenas prácticas para mantener el estilo y la calidad del código. Por ello se usan hooks de git para obligar a ejecutar lintern y los tests antes de hacer subir los cambios a la rama y asegurarnos de la calidad y funcionamiento del código.
- Los commits se realizan siguiendo las buenas prácticas.
- CI/CD: se realizará mediante el uso de Github actions.
- Se dispondrá de dos entornos: PRE (pruebas integradas) y PRO (entorno productivo). Para ello se usa okteto como cluster de k8s en la nube con diferentes namespaces. Las BBDD para el entorno de PRE serán contenedores con persistent volumes. En PRO se usan instancias en AWS.
- Se usa Amazon como proveedor de recursos en la nube.
- Se usa Kafka como broker de mensajería.
- Todos los servicios contienen la documentación necesaria en su repositorio (README) que explica en detalle su estructura y uso.

Los pasos seguidos para el desarrollo del TFM han sido los siguientes:

1. Repaso de la materia vista.
2. Ideación de la propuesta.
3. Repositorio de infraestructura
4. Desarrollo del primer micro: users-service.
5. Desarrollo del API Gateway.
6. Desarrollo del API de productos.
7. Microservicio de compras.

3.1 Repaso de la materia

Con el fin de poder realizar una propuesta, realicé un repaso del temario del máster. Conforme avanzaba la lectura del material, iba apuntando aquello que consideraba vital reflejar en el TFM, así como las posibles ideas que me iban surgiendo. Esta tarea fue una de las que más tiempo dediqué, era importante tener claro los puntos a cubrir y cómo encajarlos entre sí.

[1] https://github.com/aquasecurity/kube-bench y https://github.com/aquasecurity/kube-hunter

3.2 Ideación de la propuesta

Con todo lo recopilado en el punto anterior, llegó el momento de sentarse y acabar de encajar las piezas para dar forma a la propuesta. Sintiendo no ser muy original, acabé optando por la realización de una aplicación de compra de productos y entrega. Este tipo de aplicaciones me permitía poner en práctica varios de los puntos vistos durante el curso: DDD (cada micro representa un dominio), SAGAS, CQRS, Event sourcing, llamadas entre micros mediante API Rest (API Gateway), mensajería, etc. El resultado final fue la propuesta mencionada anteriormente en el resumen.

3.3 Repositorio de infraestructura

De forma inicial tenía sentido crear un repositorio[2] en el que almacenar toda la gestión posible de la infraestructura. En un primer paso se ha creado una estructura /infrastructure/cloud/aws/cloudFormation en la que se ha guardado el archivo para definir el stack de cloud formation de Amazon, donde definiremos todos los recursos de Amazon a usar en el TFM. Evidentemente este stack o el posible contenido del repositorio se irá modificando a la par que corresponda con el desarrollo de los micros.

3.4 Desarrollo del primer micro

La idea inicial que tuve era la de llevar a cabo un primer micro en el que poner en práctica todos los puntos a tener en cuenta en el TFM. De esta forma consideré que sería más fácil desarrollar de forma inicial un único micro, ir solucionando los problemas encontrados y refinarlo, para después llevarlo al resto. Definición del API, modularización del proyecto, desarrollo del código, validación de código y estilo, tests unitarios y de integración, dockerización, kubernetización y flujos de integración y despliegue continuos van a darse de forma muy similar en todos los micros.

El micro elegido como punto de partida fue el de usuarios[3]. Varios fueron los motivos que me llevaron a tomar la decisión:

- La tecnología usada. El micro va a estar desarrollado en nodejs, tecnología en la que no soy experto. Prefería de forma inicial comenzar con aquello que pudiese ser más complejo y dilatarse en el tiempo por los posibles problemas encontrados, dejando para el final aquello que me resultase más sencillo de desarrollar (micros en Java).
- Dependencia del resto de micros con él. Dado que este micro es el que permite dar de alta a los usuarios y realizar la autenticación para llamar al resto de endpoints, era evidentemente un buen punto de partida.
- Sencillez. La complejidad de la lógica de este micro era pequeña.

Una vez elegido el punto de partida, los pasos seguidos fueron los siguientes:

API First

Comenzamos con la definición del API Rest con Open API[4]. Dicha definición, nos permite tener claro el cuerpo de las peticiones y las posibles respuestas. Además, de esta forma obtuve varias ventajas:

---

[2] https://github.com/mca-tfm/infrastructure
[3] https://github.com/mca-tfm/users
[4] https://github.com/mca-tfm/users/blob/main/apis/rest/openapi-v1.yml

- Tener un contrato para poder realizar desarrollos en paralelo, aunque no haya sido así.
- Documentar nuestro API.
- Desde el contrato se podría haber generado el código de la aplicación node. Esta era la idea inicial, pero tras probar y ver que el código generado no se aproximaba demasiado a lo que tenía en mente, opté por hacer el desarrollo de forma manual, obviando esta parte. De cara a poder realizar esto, una de las opciones usadas fue la de usar la opción ***Generate Server*** del editor online de swagger:

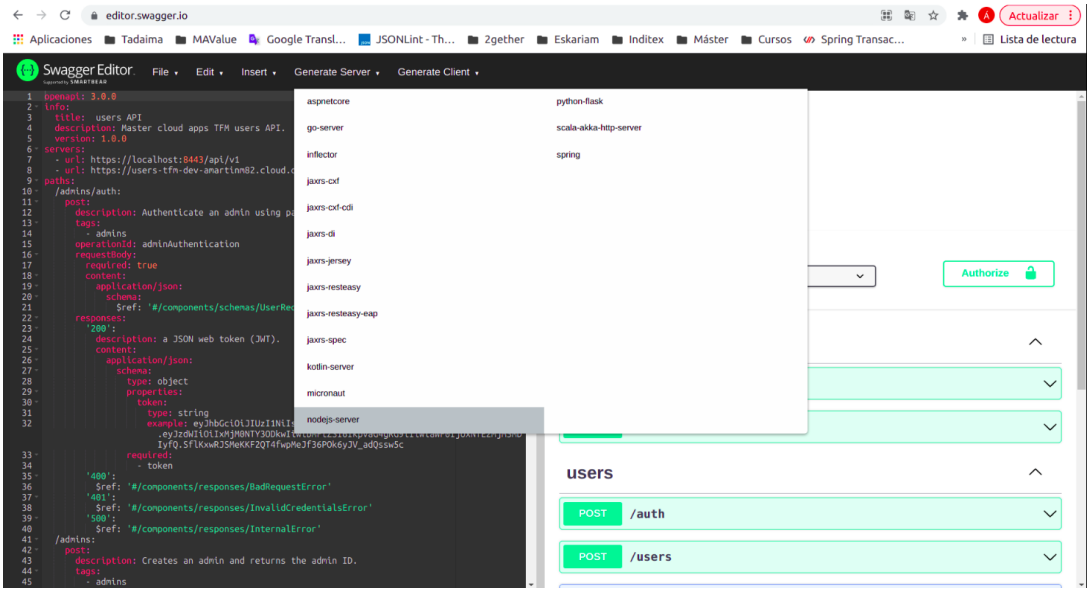

Figura 4: Vista de Generate server en el editor online de swagger

- Nos permite realizar peticiones de prueba, además de ser un punto de partida interesante para generar la colección de Postman necesaria para poder realizar pruebas.

Para todo lo anterior fue útil la página del editor online de swagger[5].

Desarrollo inicial del micro

A continuación se llevó a cabo el desarrollo del servidor que implementa la especificación definida. Para ello se optó por una arquitectura por capas, donde los router (controladores) hacen uso de los servicios, que a su vez hacen uso de los repositorios. Para llevar a cabo la comunicación entre capas y abstraernos de la tecnología de BD usada y sus modelos (en este caso Sequelize) se ha hecho uso de DTOs. Los router reciben requests que transforman en DTOs y que pasan a los servicios. De igual manera los servicios transforman la respuesta obtenida por los repositorios en DTOs de respuesta que devuelven a los controladores.

Con esto se pretende desacoplar la lógica del modelo, y si se opta por cambiar la tecnología de la BD usada, que esto sea lo más sencillo posible, simplemente cambiando modelos y repositorios.

[5] https://editor.swagger.io/

Además inicialmente se ha hecho uso de certificados autofirmados para que el servidor use HTTPS.

En esta parte del desarrollo se abordaron de forma inicial los endpoints de creación y autenticación, pudiendo tener una base sobre la que trabajar, y completando el resto de endpoints a futuro.

Tests

Para este micro se optó por una metodología TLD (Test Last Development), realizando los tests tras la implementación del código. Sin embargo, con la experiencia adquirida en el desarrollo de este proyecto, se pretende que el resto de desarrollos usen un enfoque TDD, debido a que a partir de este punto ya tenemos noción de la realización de tests en este marco tecnológico.

En primer lugar se llevaron a cabo los tests unitarios, en los que se ha puesto más detalle tratando de tener la mayor cobertura posible.

A continuación se llevó a cabo el desarrollo de los tests de integración. He aquí donde encontramos el primer escollo en el camino:
Para el desarrollo de los tests de integración, se hace uso de test containers, para levantar contenedores con la imagen de la BD correspondiente MySQL. Sin embargo, al tratar de levantar la aplicación, Sequelize al cargar el modelo necesita tener la conexión establecida a la BD, y al lanzar los tests no esperaba que se crease el contenedor.
Esto llevó a dedicar bastante tiempo para encontrar la solución a este problema. Finalmente, se solventó haciendo uso de la dependencia de la librería jest-testcontainers[6], que hace que los tests esperen a que el contenedor se cree y la BD esté levantada de forma previa a lanzar los tests.

De la implementación de los tests se obtuvo también la implementación de la lógica de reintentos para conectar a la BD.

Por último se proporcionaron los comandos pertinentes para poder ejecutar todos los tests, sólo los tests unitarios, o sólo los tests de integración.

ESLint

Para asegurar la calidad del código, se añadió ESLint al proyecto. Se configuró el proyecto para que cada vez que se lancen los tests, de forma previa ejecutase la validación de código, de tal forma que si nuestro código tiene algún error de estilo, no permite ejecutarlos.

Ya que carecía de experiencia en el uso de este tipo de librerías, tuve que dedicar un tiempo a la lectura y entendimiento de material relacionado. Los recursos usados se mencionan en el apartado ESLint de Bibliografía[7].

Como nota señalar que se optó por el uso de la configuración de Airbnb para las reglas de ESLint, debido a que es una de las más aceptadas y usadas por la comunidad.

CI (Git Hooks)

Dado que se hace uso de la estrategia de Trunk Based Development, todo lo que se encuentre subido en la rama debe estar preparado para ser puesto en producción en cualquier momento. Por tanto tiene sentido asegurar que aquello que sube cumple los estándares y es completamente funcional.

---

[6] https://github.com/Trendyol/jest-testcontainers
[7] https://docs.google.com/document/d/1rzfNC5GhiKooNIXE9CLif8khE_ZH1lKt/edit#heading=h.nmkz6txcyjno

Durante la lectura sobre el uso de ESLint encontré algo que se adecuaba perfectamente a la naturaleza del proyecto, los hooks de git. Descubrí que era posible hacer uso de ellos para controlar que sólo se pudiese hacer commit o push de aquel código que cumpliese los requisitos deseados: estar bien formado y pasar los tests.

Para ello opté por tomar la siguiente estrategia:

- Al realizar un commit, el ESLint y los tests unitarios se lanzan de forma previa. Si la ejecución de estos falla, no se permite realizar commit de los cambios.
- Al hacer un push, los tests de integración se lanzan, si el resultado de estos no es OK, entonces no se permite hacer push sobre la rama remota.

De esta forma estamos realizando left shift programming, intentando detectar y prevenir los errores del software lo antes posible, y asegurando así la robustez de lo que exista en la rama.

Para la creación de los hooks, se hizo uso de la librería husky[8] que simplificaba mucho el proceso. Los scripts de los hooks que se ejecutan se encuentran dentro de la carpeta .husky[9] del proyecto.

Dockerización

Para los siguientes pasos era necesario “Dockerizar” la aplicación. Para ello se dió de alta el Dockerfile, validado mediante hadolint[10]. A continuación se crearon los siguientes ficheros:

- **Docker-compose-dev[11]:** permite levantar los recursos necesarios para el desarrollo de pruebas en local (BD MySQL).
- **Docker-compose[12]:** levanta los recursos necesarios para ejecutar la aplicación usando una imagen de docker local, además de la BD. La imagen a usar se recupera de la variable de entorno *DOCKER_LOCAL_IMAGE* definida en el archivo .env.
- **Dockerize.sh[13]:** se encarga de generar la imagen docker en local con el código actual, sin subirla a Dockerhub[14]. Para ello usa como nombre de imagen la variable de entorno *DOCKER_LOCAL_IMAGE* (que además añade escribe en el fichero .env del que el docker-compose recupera las variables de entorno a usar). A continuación levanta la app ejecutando el fichero docker-compose anterior.

Por último, se añadió un script en el *package.json* del proyecto que invoca al script de dockerización. En él asigna el valor a la variable de entorno DOCKER_LOCAL_IMAGE, usando el nombre del usuario y la versión del proyecto indicada en el package.json (*${USER}/tfm-users:$npm_package_version-dev*).

Esto es útil para gente con dependencia del proyecto que esté realizado desarrollos. Por ejemplo, si alguien está desarrollando un frontal que requiere de la app, podría descargarlo en local y lanzar el script para ejecutar la aplicación, y poder hacer pruebas invocándola desde su frontal.

---

[8] https://www.npmjs.com/package/husky
[9] https://github.com/mca-tfm/users/tree/main/.husky
[10] https://hadolint.github.io/hadolint/
[11] https://github.com/mca-tfm/users/blob/main/docker-compose-dev.yml
[12] https://github.com/mca-tfm/users/blob/main/docker-compose.yml
[13] https://github.com/mca-tfm/users/blob/main/dockerize.sh
[14] https://hub.docker.com/

CI (Github actions)

Con todo lo anterior ya era posible comenzar con el flujo de integración remoto. Para ello se ha hecho uso de Github actions[15].
Se crea el fichero .github/workflows/ci-cd.yml[16] (inicialmente llamado github-action.yml, y renombrado posteriormente), en el que se define una serie de jobs encadenados, cada uno dependiente del anterior, por lo que si uno de ellos falla no se continúa con el flujo.

Las actions se definen para ser ejecutadas cuando se realiza un push o una PR sobre la única rama del repositorio: main. Por tanto, al realizar el push sobre la rama remota, se ejecutan los siguientes jobs:

- eslint: Analiza el código en la rama buscando errores de estilo.
- tests: Ejecuta los tests unitarios y de integración.
- publish-image: Genera y publica la imagen Docker ***tfm-users*** con la etiqueta ***trunk*** en Dockerhub. Esa imagen es la usada para el entorno de desarrollo, por lo que cada vez que se publican cambios en la rama, se sobrescribe dicha imagen con los últimos cambios.

Kubernetización (manifiestos k8s/Helm)

De cara a poder comenzar con los despliegues de la aplicación se comenzó a kubernetizar ésta. Inicialmente se generan los manifiestos de k8s[17] en la carpeta de dicho nombre del proyecto.

Una vez comprobado que los manifiestos funcionaban, se generaron los templates de Helm en la carpeta helm/charts[18]. La idea es que se pueda utilizar el chart para hacer despliegue en los diferentes entornos (namespaces) con diferentes configuraciones:

- namespace: indicará el entorno sobre el que se realiza el pase (tfm-dev-amartinm82 para el entorno de PRE, tfm-amartinm82 para el entorno de PRO)
- Hacer uso de una instancia de BD en Amazon o por el contrario usar un contenedor con un persistent volume permanente (mediante uso de anotación ***"helm.sh/resource-policy": keep*** para evitar su borrado).
- Imagen a usar para crear los contenedores (usar la imagen de desarrollo, con tag trunk, o la última release generada para PRO).
- Número de réplicas, recursos de los contenedores, etc.

Para la creación de los charts de helm se han tenido en cuenta el uso de buenas prácticas[19].

Para las pruebas de despliegue, inicialmente se hizo uso de minikube[20], y una vez visto que funcionaba correctamente, se optó por usar Okteto[21], de esta forma nuestra app está disponible en la web.

---

[15] https://docs.github.com/en/actions
[16] https://github.com/mca-tfm/users/blob/main/.github/workflows/ci-cd.yml
[17] https://github.com/mca-tfm/users/tree/main/k8s
[18] https://github.com/mca-tfm/users/tree/main/helm/charts
[19] https://codersociety.com/blog/articles/helm-best-practices
[20] https://minikube.sigs.k8s.io/docs/start/
[21] https://www.okteto.com/

Inicialmente la app se define como sólo accesible desde el cluster (ClusterIp), pero poder acceder y hacer pruebas, y mientras se realiza el resto de desarrollos de forma temporal se añadió un ingress de Okteto que permite acceder de forma externa a los endpoints de la aplicación.

A futuro los endpoints serán sólo accesibles a través del api gateway.

CD

Con el chart de helm disponible, modificamos el workflow de github actions para que, tras los pasos previos, desplegar la aplicación en el cluster de k8s. Para ello se hizo uso de la action del marketplace de Github actions Helm Deploy[22].

Completar el micro

Con todo lo anterior, se optó por completar la funcionalidad restante del micro, ya que disponíamos de un marco de trabajo con CI/CD robusto y testado. Por tanto los siguientes pasos fueron:

- Añadir endpoints de autenticación para obtener los tokens JWT.
- Añadir middlewares para verificar que los tokens pasados son válidos o que un usuario tiene permisos para acceder a un recurso.
- Añadir el resto de endpoints necesarios.

Postman

Finalmente, aprovechando la especificación de open api realizada, se generó y modificó una colección de Postman, que nos permite ejecutar de forma rápida una batería de tests que nos permite saber si el sistema está desplegado y las funcionalidades básicas funcionan (smoke tests).

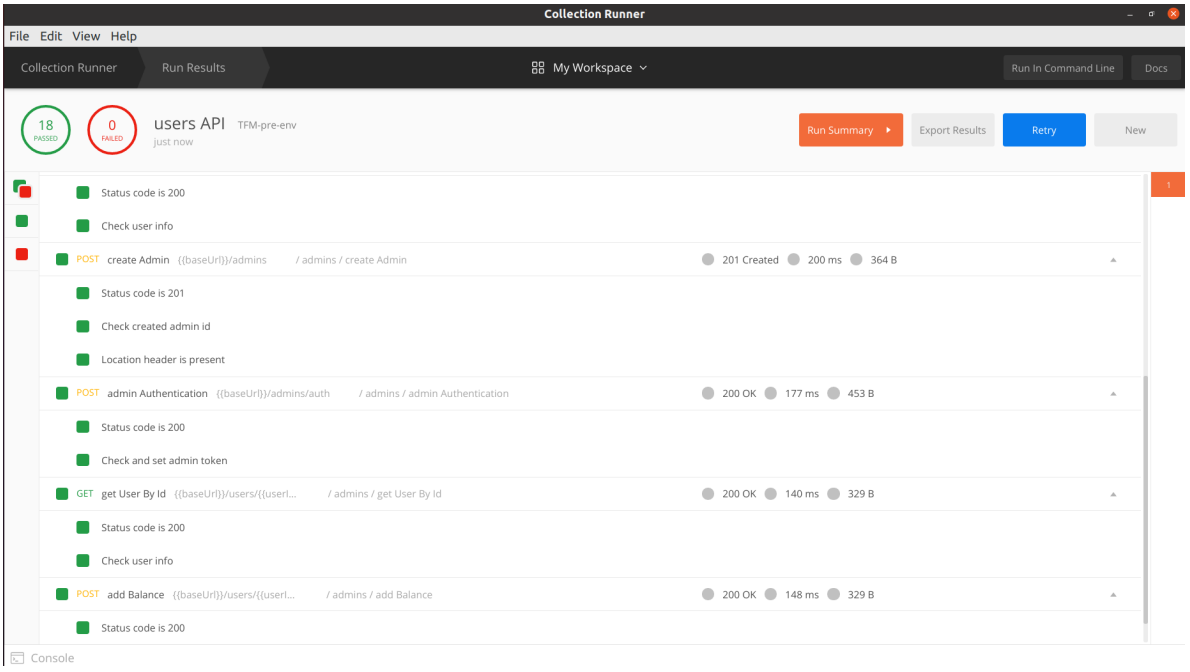

Figura 5: Colección de Postman

Generación de release

[22] https://github.com/marketplace/actions/helm-deploy

Para generar releases, dado que se usa TBD, la rama está preparada en cualquier momento para que ese código se ponga en PRODUCCIÓN. Por tanto, basta con etiquetar la rama en el punto en el que se va a librar ese código en el entorno.

Para ello se controla desde el hook de pre-push que la etiqueta usada corresponda con la versión del package.json. En caso de no ser así no se permite realizar el push de la rama.

Para facilitar la tarea se ha creado un script release que se encarga de generar el tag con la versión actual del package.json y publicarla en el repositorio remoto:

```
npm run release
```

A continuación, se añade un nuevo workflow de github actions llamado release[23], que se encargará de realizar las acciones pertinentes tras hacer el push de la etiqueta:

- check-tag: comprueba que la etiqueta corresponda con la versión del package.json.
- publish-package: publica el paquete npm en github.
- publish-release: publica la release en github usando usando el changelog como descripción. Para simplificar se utiliza un action existente[24].
- publish-image: publica dos imágenes, latest y la versión que corresponda con el package.json, en dockerhub. La imagen latest se sobrescribirá con la última imagen generada cada vez que se genere una release.
- deploy: publica la imagen usando el chart de helm definido y basándose en el action oficial de Helm[25].
- bump-version: incrementa la versión del package.json haciendo uso del mismo action de github usado en la publicación de la release.

3.5 Desarrollo del API Gateway

Dado que el proyecto va estar compuesto de diferentes APIs, tiene sentido añadir un API Gateway para poder centralizar un único punto de acceso para los diferentes APIs, así como poder exponer endpoints que sean combinación de varios, o incluso definir un comportamiento común de circuit breaking, rate limit, etc.

Por tanto, ya que todos los APIs van a ser expuestos mediante esta pieza, era lógico que éste fuese el siguiente paso a dar. Desarrollar e integrar con el primer API desarrollado previamente, ayudaría a detectar todos los posibles problemas de forma inicial para evitarlos a futuro al integrar el resto de APIs.

Para ello se ha optado por implementar el API Gateway[26] usando Spring Cloud Apigateway[27], dada todas las features que ofrece y su sencilla configuración.

Para definir el enrutado del API inicial opté por la aproximación funcional, en lugar de la configuración mediante el fichero de propiedades.

---

[23] https://github.com/mca-tfm/users/blob/main/.github/workflows/release.yml
[24] https://github.com/phips28/gh-action-bump-version
[25] https://github.com/marketplace/actions/helm-deploy
[26] https://github.com/mca-tfm/api-gateway
[27] https://spring.io/projects/spring-cloud-gateway

La hoja de ruta de implementación del API Gateway fue muy similar a la seguida en el desarrollo del micro inicial:

1. Definición de la especificación del API[28] (referencia al API anterior).
2. Desarrollo del código/enrutado.
3. Definición de hooks para el flujo de integración continua
4. Definición de chequeo del código acorde a un estándar de estilo.
5. Tests
6. Modificación de los hooks:
    a. pre-commit: ejecuta la validación del código con checkstyle y los tests unitarios.
    b. pre-push: lanza los tests CDCT, así como los de integración.
7. Dockerización de la aplicación
8. Añadir workflows de CI a github actions.
9. Kubernetización de la aplicación.
10. Modificación del workflow de ci-cd para desplegar en el cluster de k8s del entorno de preproducción
11. Añadir el flujo de generación de release.
12. Completar la documentación.

A continuación se detalla los pasos más relevantes, dado que otros son muy similares a los explicados con anterioridad.

Definición de hooks para el flujo de integración continua

De forma similar a lo realizado en el primer API, se hace uso de los hooks de git. Investigando descubrí que existía un plugin no oficial de Maven[29] que de forma fácil nos ofrecía la posibilidad de instalar los hooks de forma local.

Checkstyle

Si para el proyecto anterior de nodejs elegimos ESLint para asegurar la calidad del código y amoldarse a un estándar, para este proyecto Java la alternativa ha sido checkstyle[30].

Se eligió como estándar para el proyecto el de Google[31] y se configuró el hook de pre-commit para que antes de realizar un commit se ejecutase la validación de código, de tal forma que si nuestro código tiene algún error de estilo, no se permita realizarlo.

Tests

Respecto a los tests en este proyecto cabe destacar el uso de CDCT mediante el uso de Pact[32]. Se han realizado tests a nivel de consumidor, que además generan un contrato local, y tests a nivel de proveedor que consumen esos contratos generados previamente. De esta forma nos aseguramos que el comportamiento del API de usuarios que servimos desde el API Gateway es el esperado.

---

[28] https://github.com/mca-tfm/api-gateway/blob/main/api/openapi.yml
[29] https://github.com/phillipuniverse/githook-maven-plugin
[30] https://checkstyle.sourceforge.io/
[31] https://github.com/mca-tfm/api-gateway/blob/main/checkstyle/intellij-java-google-style.xml
[32] https://docs.pact.io/

Como herramientas complementarias se ha usado JUnit 5[33], y testcontainers[34], que levantan la imagen de los APIs usados (inicialmente Users API, y el resto a posteriori) para poder lanzar los tests de proveedor. Para ello se apoya en un fichero docker compose[35], tal y como se explica en la documentación del proyecto. Además, para los tests unitarios, se ha hecho uso también de Wiremock[36], para mockear las peticiones a los APIs a los que se enruta (Users API, etc).

Cabe destacar que este punto tuvo complicación debido al uso de certificados en el API, por lo que en los tests se deshabilita la validación de SSL.

Por último indicar que los tests se han desarrollado haciendo uso de JUnit 5 Suites[37], permitiendo lanzar los tests por categoría (unitarios, de integración y CDCT).

Dockerización

Para generar la imagen Docker de la app se ha hecho uso de Jib[38]. Esto nos permite no tener instalado Docker en la máquina, y evitar tener que generar nosotros nuestro propio Dockerfile para generar imágenes optimizadas. Para generar la imagen se configuró el proyecto usando el plugin de Maven existente[39].

De manera similar al API de usuarios, se ha creado un script ***dockerize.sh***[40] que se comporta parecido lanzando el comando maven correspondiente (Ver Dockerización en la sección anterior).

CI (Github actions)

Con todo lo anterior ya era posible comenzar con el flujo de integración remoto. Por tanto se creó el fichero .github/workflows/ci-cd.yml[41], en el que se definen los jobs, dependientes entre sí.

Las actions se definen para ser ejecutadas cuando se realiza un push o una PR sobre la única rama del repositorio: main. Por tanto, al realizar el push sobre la rama remota, se ejecutan los siguientes jobs:

- checkstyle: Analiza el código en la rama buscando errores de estilo.
- tests: Ejecuta los tests unitarios, de integración, y CDCT, teniendo cuidado de ejecutar primero los tests de consumidor para generar los contratos que usan los test de proveedor posteriormente.
- publish-image: Genera y publica la imagen Docker ***tfm-apigw*** con la etiqueta ***trunk*** en Dockerhub. Esa imagen es la usada para el entorno de desarrollo, por lo que cada vez que se publican cambios en la rama, se sobrescribe dicha imagen con los últimos cambios.

Kubernetización (manifiestos k8s/Helm)

---

[33] https://junit.org/junit5/docs/current/user-guide/
[34] https://www.testcontainers.org/
[35] https://www.testcontainers.org/modules/docker_compose/
[36] https://wiremock.org/docs/junit-jupiter/
[37] https://junit.org/junit5/docs/current/user-guide/#junit-platform-suite-engine
[38] https://github.com/GoogleContainerTools/jib
[39] https://github.com/GoogleContainerTools/jib/tree/master/jib-maven-plugin
[40] https://github.com/mca-tfm/api-gateway/blob/main/docker/dockerize.sh
[41] https://github.com/mca-tfm/api-gateway/blob/main/.github/workflows/ci-cd.yml

Una vez somos capaces de generar la imagen Docker, procedemos a kubernetizarla generando los manifiestos en la carpeta k8s[42], y posteriormente se generaron los templates de Helm en la carpeta helm/charts[43]. La idea es usar el chart para hacer despliegue en los diferentes entornos (namespaces) con las diferentes configuraciones que correspondan.

Cabe señalar que dado que el API Gateway depende de otros recursos creados antes en el cluster de k8s (API users), es necesario pasar ciertos valores que indiquen por ejemplo el nombre del servicio y el puerto de éstos, para poder comunicarnos con ellos.

Al igual que antes, la app se define como sólo accesible desde el cluster (ClusterIp), se añadió un ingress de Okteto que permite acceder de forma externa.

CD

Con el chart de helm disponible, modificamos el workflow de github actions para que, tras los pasos previos, desplegar la aplicación en el cluster de k8s.

Debido a esto ya se pudo modificar la configuración de despliegue del API de usuarios, que antes estaba configurada para acceder desde un ingress, para no ser visibles desde fuera, y acceder únicamente a través del api gateway. Por tanto se modificaron sus manifiestos y el chart de helm, eliminando el recurso ingress, y actualizando la doc para hacer referencia al acceso vía API Gateway.

Generación de release

Por último, para la generación de release, se configuró el proyecto para usar Maven-release-plugin[44].

Generar una release es tan simple como ejecutar el comando de maven que se indica en la documentación del proyecto. Generará una release usando como tag la versión del proyecto, sin -SNAPSHOT, y desencadenando todo el flujo definido en el workflow de release de la carpeta .github/workflows.

El flujo es prácticamente idéntico al explicado en la sección anterior (Generación de release), con la diferencia del bump de la versión, ya que es el plugin de Maven quien se encarga de hacerlo automáticamente.

3.6 Desarrollo del API de productos

La realización del micro de productos[45] cambia respecto a los anteriores en cuanto que en éste se aplica desarrollo usando enfoque TDD.

Los pasos seguidos fueron:

1. Definir el API[46]
2. Añadir la configuración inicial del proyecto con eslint, tests y git hooks. En este punto cabe señalar que no nos permitirá hacer commit de ningún cambio que no

[42] https://github.com/mca-tfm/api-gateway/tree/main/k8s
[43] https://github.com/mca-tfm/api-gateway/tree/main/helm/charts
[44] https://maven.apache.org/maven-release/maven-release-plugin/index.html
[45] https://github.com/mca-tfm/products
[46] https://github.com/mca-tfm/products/blob/main/apis/rest/openapi-v1.yml

pase los tests unitarios, por tanto para el TDD, los commits se verán a nivel de test y código funcionando.
3. Implementar tests y el correspondiente código.
   NOTA: Para la generación de los ids con dynamodb se hizo uso de la librería dynamodb-atomic-counter[47].
4. Dockerización[48] y helm chart[49].
5. Postman
6. CI despliegue en PRE[50]
7. Añadir la configuración al apigateway para poder probar despliegue en PRE.
8. Modificar el stack de cloud formation para crear las tablas de dynamodb.
9. Crear un usuario en AWS con grupo con sólo permisos de acceso a dynamodb.
10. Añadir workflow de release[51], modificar pre-push hook, añadir script de release.
11. Generar release de products y desplegarla.
12. Generar release del api gateway y desplegarla.

3.7 Microservicio de compras

Se sigue el flujo de igual forma que los anteriores.

1. Definir el API[52].
2. Añadir estructura de proyecto
3. Definir ci: stilo, prehooks, etc.
4. Añadir estructura hexagonal
5. Desarrollo de los endpoints.
6. Añadir CI/CD PRE[53]
7. Añadir enrutado API Gateway.
8. Añadir consumo de mensajes en products y users service.
9. Generar y desplegar la release[54].
10. Publicar release de products y users.
11. Generar release del api gateway y desplegarla.
12. Completar la documentación.

Dado que los pasos son exactamente iguales a los anteriores, sólo destacar:

- Se ha aplicado CQRS y Event Sourcing en el desarrollo de este micro.
- Para la lectura y producción de mensajes se ha hecho uso de Spring-Kafka[55].
- Tanto zookeeper[56] como kafka[57] usados en la aplicación se despliegan mediante manifiestos ya existentes encontrados en la web.
- Se ha hecho uso de campos de tipo JSON en la BD, para almacenar la lista de items dentro de un carro de la compra. Esto es así dado que esa información no se

[47] https://www.npmjs.com/package/dynamodb-atomic-counter
[48] https://github.com/mca-tfm/products/blob/main/dockerize.sh
[49] https://github.com/mca-tfm/products/tree/main/helm/charts
[50] https://github.com/mca-tfm/products/blob/main/.github/workflows/ci-cd.yml
[51] https://github.com/mca-tfm/products/blob/main/.github/workflows/release.yml
[52] https://github.com/mca-tfm/purchases/blob/main/apis/rest/openapi-v1.yml
[53] https://github.com/mca-tfm/purchases/blob/main/.github/workflows/ci-cd.yml
[54] https://github.com/mca-tfm/purchases/blob/main/.github/workflows/release.yml
[55] https://www.baeldung.com/spring-kafka
[56] https://kubernetes.io/docs/tutorials/stateful-application/zookeeper/
[57] https://kow3ns.github.io/kubernetes-kafka/manifests/

puede relacionar directamente con otras entidades dentro de la BD, ya que pertenece a otro sistema.
- Se ha hecho uso del patrón estrategia y patrón SAGA para el flujo de compra. Dicho flujo se detalla en el siguiente apartado.

Flujo de compra

El método update del caso de uso de orders[58] usa un patrón estrategia. En función del estado al que se actualiza el pedido, recupera el servicio con la lógica a ejecutar adicional a actualizar el estado del pedido.

A continuación se describen los estados y sus acciones:

- Validating items[59]: llama al método de validación de items del repositorio. Esto implica publicar un evento en la cola de validación de items, que el servicio products lee. Con la información de dicho evento valida cada uno de los items del pedido:
    - Si los items son correctos, se retiene el stock correspondiente y se manda un evento al tópico de cambio de estado del pedido, solicitando actualizar el estado al siguiente estado indicado en el evento (Validating balance).
    - Si no lo son, se manda un evento al tópico de cambio de estado del pedido, indicando que el pedido debe pasar al estado de fallo indicado en el evento (rechazado).
    - El servicio de compras lee el tópico de actualización de estado y ejecuta el método update previamente comentado del caso de uso de orders.
- Validating balance[60]: Si la validación de los items fue correcta, se llama al método de validación de saldo del usuario, por tanto, se envía un evento al tópico de validación de saldo, que le servicio users está escuchando.
    - Si el usuario tiene suficiente saldo, se resta el importe del pedido y se manda un evento al tópico de cambio de estado del pedido, solicitando actualizar el estado al siguiente estado indicado en el evento (Hecho).
    - Si no dispone de suficiente dinero, se manda un evento al tópico de cambio de estado del pedido, indicando que el pedido debe pasar al estado de fallo indicado en el evento (rechazado).
    - El servicio de compras lee el tópico de actualización de estado y ejecuta de nuevo el método update del caso de uso de orders.
- Rejected[61]: Si el estado pasa a ser rechazado, se revisa el estado previo que llevó a rechazar el pedido, y en función de éste se ejecutan las acciones pertinentes (patrón SAGA).

---

58 https://github.com/mca-tfm/purchases/blob/main/src/main/java/es/codeurjc/mca/tfm/purchases/domain/usecases/OrderUseCaseImpl.java#L69

59 https://github.com/mca-tfm/purchases/blob/main/src/main/java/es/codeurjc/mca/tfm/purchases/domain/services/impl/ValidatingItemsOrderStateServiceImpl.java

60 https://github.com/mca-tfm/purchases/blob/main/src/main/java/es/codeurjc/mca/tfm/purchases/domain/services/impl/ValidatingBalanceOrderStateServiceImpl.java

61 https://github.com/mca-tfm/purchases/blob/main/src/main/java/es/codeurjc/mca/tfm/purchases/domain/services/impl/RejectedStateServiceImpl.java

    - Si el pedido se rechaza en el paso de la validación de items, no se ha realizado ninguna acción previa en el sistema, por lo que no es necesario hacer nada
    - Si el pedido se rechazó debido a que el usuario no contaba con suficiente crédito, es necesario volver a liberar la reserva de stock. Para ello se llama al método de liberación de stock, que publica un evento que el servicio de products escucha. Con dicho evento incremente de nuevo la cantidad de stock retenida previamente.
    - Se llama al método de finalización de pedido.
- Done[62]: no hay acciones que realizar, salvo llamar al método de finalización de pedido.
- Tanto los estado Rejected como Done son estados finales, que acaban desembocando en la llamada al método de finalización del pedido (actualmente no realiza ninguna acción). Por ello ambos extienden de una clase abstracta con el código común[63].

Para un mejor entendimiento del flujo de un pedido se adjuntan los siguientes diagramas:

- Pedido validado y realizado correctamente (ver Figura 6).
- Pedido rechazado en la validación de items (ver Figura 7).
- Pedido rechazado en la validación de saldo del usuario (ver Figura 8).

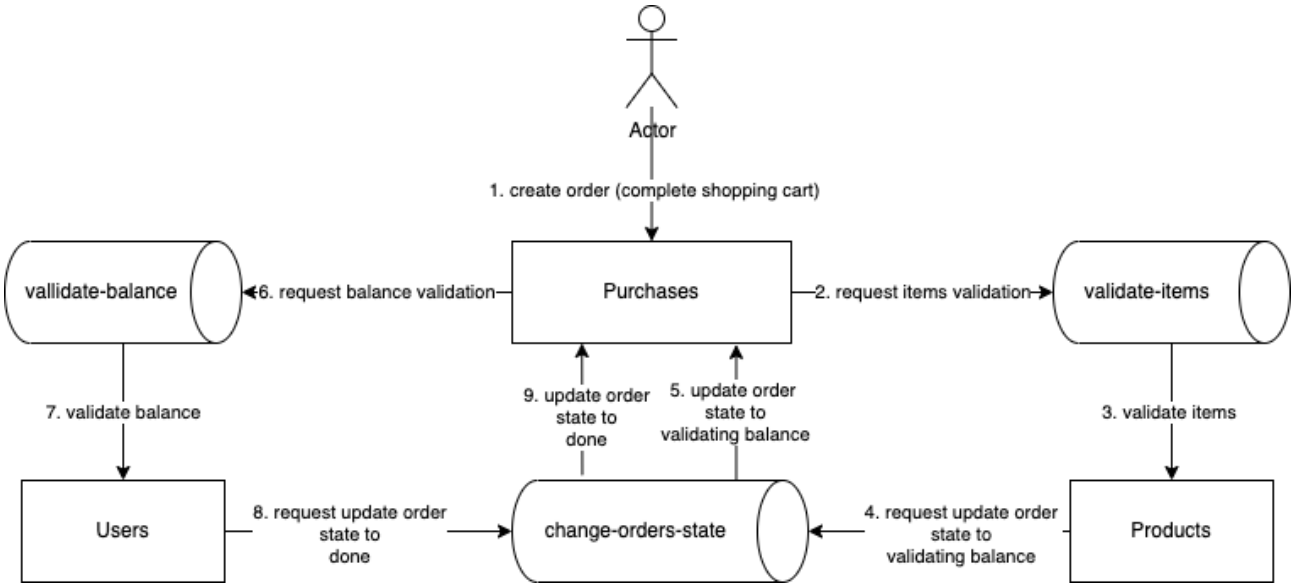

Figura 6: Diagrama de un flujo de compra de un pedido correcto.

62

https://github.com/mca-tfm/purchases/blob/main/src/main/java/es/codeurjc/mca/tfm/purchases/domain/services/impl/DoneOrderStateServiceImpl.java

63

https://github.com/mca-tfm/purchases/blob/main/src/main/java/es/codeurjc/mca/tfm/purchases/domain/services/impl/FinalAbstractBaseOrderState.java

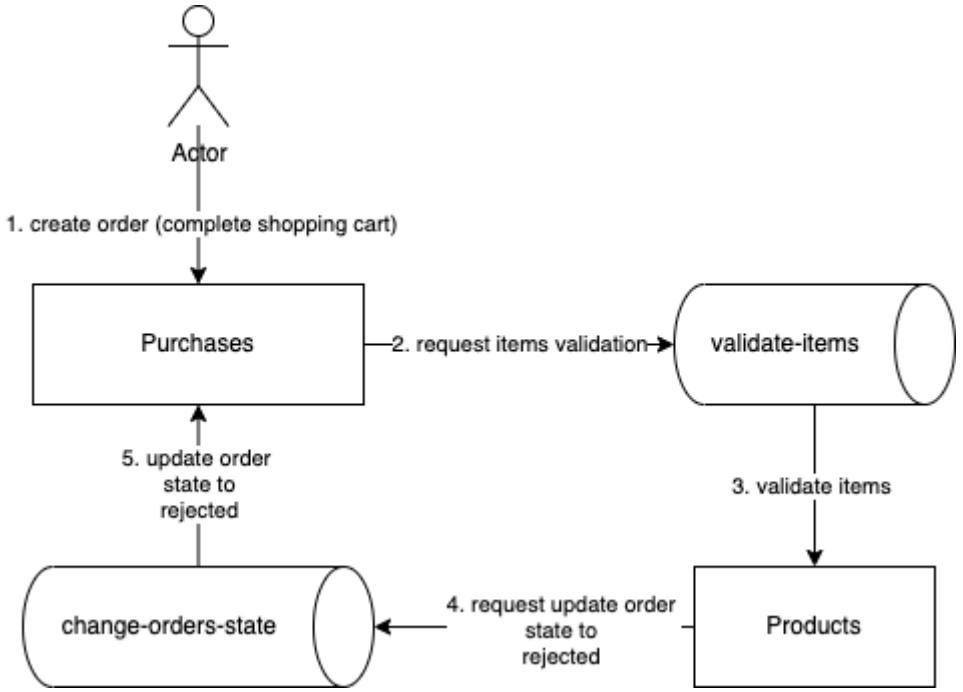

Figura 7: Diagrama de un flujo de compra rechazado en la validación de items.

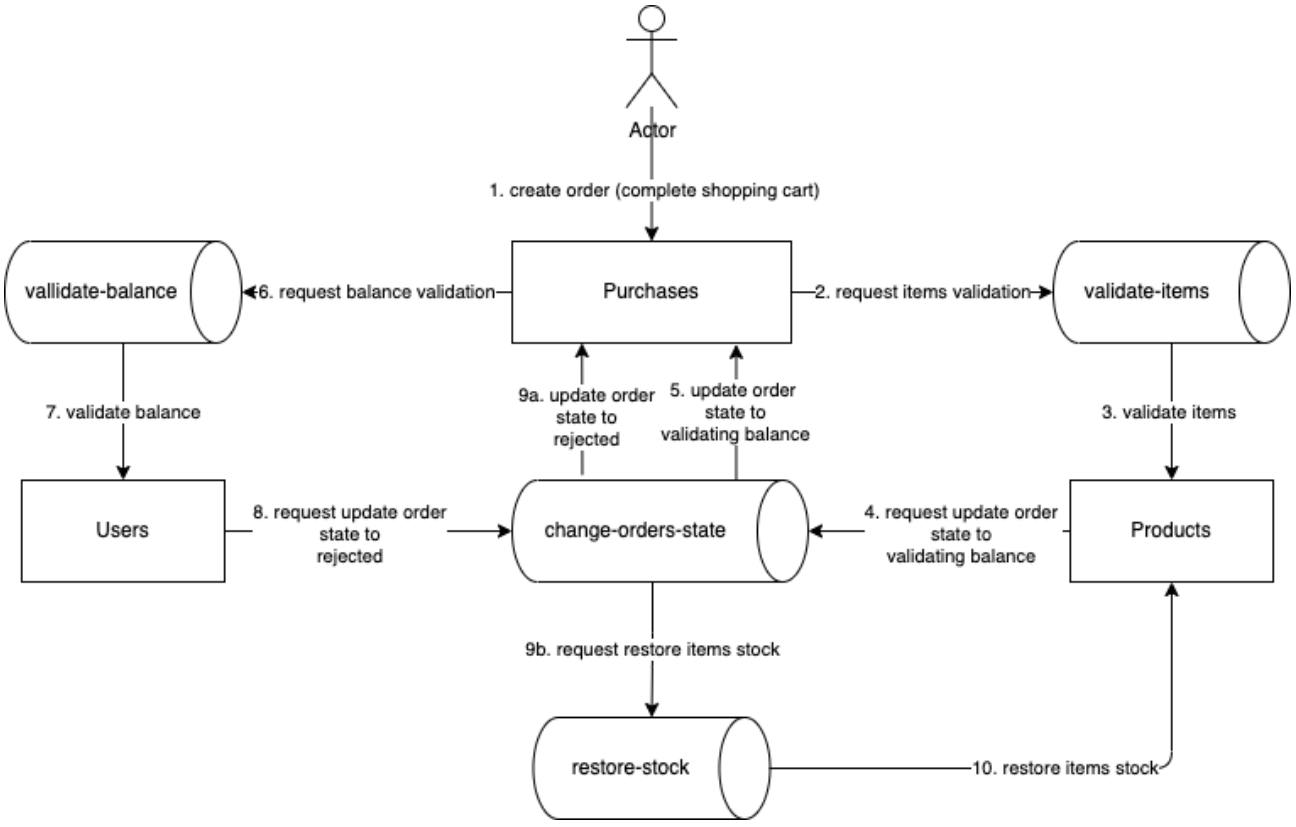

Figura 8: Diagrama de un flujo de compra rechazado en la validación del saldo del usuario.

4. Conclusiones

Sentarse y dedicar varios días a pensar, idear y diseñar el proyecto ha sido vital. Invertir tiempo en un buen diseño suele dotar de potencia y robustez a los sistemas, evitando posibles problemas o refactorizaciones futuras. De esta forma ha sido fácil hacer el desarrollo teniendo claro cuál era el objetivo y los pasos a dar.

El diseño del proyecto inicialmente era mucho más complejo y ambicioso, pero se tuvo que simplificar por falta de tiempo, aunque seguiré intentando iterar sobre el proyecto para plasmar toda la idea inicial (trabajos futuros).

Se ha aplicado DDD y arquitectura hexagonal en el proyecto. En este caso, al ser un proyecto tan pequeño, los dominios estaban claramente identificados, así como la interacción entre ellos. Sin embargo, desarrollar haciendo uso de arquitectura hexagonal me ha resultado bastante complejo. Considero que es un modelo de desarrollo que pese a sus bondades, complejiza mucho los desarrollos. Quizás sea la costumbre de trabajar con arquitecturas por capas, pero hacer el cambio no ha resultado todo lo fácil que se hubiese deseado.

Comenzar el desarrollo de los servicios dotándolos de forma inicial de los mecanismos de control de formato y calidad del código (linterns), batería de tests y flujos de CI/CD ha sido fundamental por varios motivos:

- El código va a cumplir con el estándar de formato y calidad definido desde el minuto 1, evitando tener que reformatearlo a posteriori, y asegurando que cualquier persona que quiera contribuir al proyecto va a seguir manteniendo ese estándar.
- Con la ejecución de los tests nos aseguramos de que en cada iteración de desarrollo sobre cada uno de los micros, la funcionalidad previa sigue funcionando (regresión) y lo añadido funciona como se espera. De esta forma verificamos que el código está preparado en todo momento para ser puesto en producción.
- Desplegar los microservicios es un proceso automatizado y sencillo, que sólo preocupa durante su desarrollo y primera ejecución, luego es algo que resulta “transparente”.

Me hubiese gustado probar el modelo TBD con varias personas implicadas en el proyecto. Veo las bondades del modelo, que obligan a implantar las herramientas necesarias para asegurar que el código en la rama main está preparado en todo momento para su puesta en producción. He de añadir que no estaba acostumbrado a trabajar directamente sobre la rama main, y esto me generaba cierto respeto. En muchas ocasiones, para realizar pruebas o desarrollos que eran totalmente nuevos para mí, veía la necesidad de crear ramas donde llevarlos a cabo. En lugar de eso acababa realizando las pruebas en otro repositorio aparte para evitar llenar la rama de commits de prueba y error.

Me hubiese gustado aplicar TDD durante todo el desarrollo del TFM, pero me ha resultado difícil aplicarlo por momentos. Durante el desarrollo en arquitectura hexagonal, era difícil visualizar la implementación de los tests metiendo lógica en capas que no correspondían para posteriormente moverlas en la fase de refactor. Supongo que es cuestión de trabajar sobre ello disponiendo de tiempo.

Pese a que se ha hablado mucho de él durante el curso, nunca había hecho uso de Okteto como clúster de k8s. Me parece un proyecto muy interesante y potente para la formación y pruebas en cluster de k8s para aquellos que lo deseen. La capa gratuita permite hacer usos de varios namespaces, y en cada uno de ellos se puede desplegar hasta 10 pods, con 5Gb de almacenamiento y dotar a los pods de 1gb de ram y 1 cpu como máximo.
Sin embargo hay algunas peculiaridades que lo diferencian con un clúster normal, como por ejemplo que dota de un ingress de acceso abierto par cada uno de los pods que se desplieguen, y hay que desactivarlo si se quiere restringir el acceso a ellos:

```
metadata:
 annotations:
   dev.okteto.com/auto-ingress: 'false'
```

Además cabe añadir que pasado un tiempo de inactividad, el cluster se hiberna, y si ese tiempo excede un límite, se elimina. Por ello, para evitar este tipo de problemas tuve que:

- Realizar peticiones al cluster de PRE cada cierto tiempo para evitar su borrado.
- Implementar el flujo de despliegues manual de releases, para poder desplegar en PRO aquellos pods que se hubiesen eliminado. Esto último acaba sirviendo también como un posible mecanismo de rollback en caso de querer dar marcha atrás y volver a una release anterior.

El cluster de PRO estaba encendido sólo en momentos puntuales, ya que se hacía uso de la capa gratuita de Amazon, y no se quería incurrir en gastos de facturación. Para ello, se realizaba la ejecución del cloudFormation para proporcionar de los recursos necesarios a la aplicación, y posteriormente se hacía el despliegue/levantaba el cluster de PRO.

El dotar al servicio de usuarios y productos del sistema de mensajería no es algo que hubiese hecho en un sistema real. Era más partidario de que fuese el servicio de compras quien se encargase de hacer llamadas REST o gRPC a los servicios para validar los productos y hacer la reserva de stock, o validar y retener el saldo del usuario. Por simplificar (evitar realizar llamadas REST teniendo que pasar un token a los endpoints y tener que gestionar la caducidad de éste, o tener que implementar un servicio gRPC, o incluso por gestionar problemas en la llamada por posibles cortes de red y saber si se ha llamado varias veces para retener el saldo o el stock) se ha optado por el mecanismo de las colas.
Para solucionarlo en parte, se ha dotado de una propiedad que permite activar o desactivar el consumo de mensajes, en caso de que se quisieran usar los micros en otro sistema.

Durante todo el curso lectivo, conforme se iban impartiendo las diferentes asignaturas, tenía muy claro que quería poder realizar un TFM en el que poder abordar la mayor parte del temario. No sólo para poder profundizar en puntos en los que quizás se pasó de forma superflua durante las clases y prácticas, si no para poder repasar todo lo visto y que estos nuevos conocimientos calasen en mí, aunque implicase realizar un TFM más complejo de lo necesario.

Por último señalar que la realización del máster me ha servido para:

- Asentar y mejorar conocimientos ya adquiridos a lo largo de mi experiencia laboral. Me parecía importante perfeccionar la parte de despliegues en la nube mediante

cluster de k8s, así como el uso de AWS, pese a haber realizado proyectos con estas tecnologías.

- Adquirir nuevos conocimientos que me ayudasen en las tareas y responsabilidades de mi día a día en el trabajo. Si algo tenemos claro en el mundo del desarrollo software, es que debemos de estar en continua formación para poder aportar valor. Es renovarse o morir.
- Convertirme, a nivel profesional, en un perfil más valioso para mi empresa y más atractivo para el mundo laboral.

5. Mejoras a futuro

Pese a que se han aplicado gran parte de las tecnologías y metodologías vistas en el máster, se podría seguir avanzando en esta línea. En concreto, se han identificado los siguientes puntos de mejora:

- Añadir a los micros los endpoints de acceso a openapi/swagger.
- Añadir javadoc a los microservicios en nodejs.
- Incrementar las versiones de node y Java a la última versión.
- Usar certificados no autofirmados.
- Sacar los certificados de la imagen del proyecto y el secreto e inyectarlos en el proceso de CD.
- Comprobar que los micros se han desplegado automáticamente en el workflow (lanzar los tests de Postman automáticamente).
- Configurar apigateway para que no confie en todos los certificados.
- añadir np ingress/egress para limitar la visibilidad a los pods sólo desde donde corresponda.
- Añadir paginación y ordenación al endpoint get all de products.
- Al crear el rol de AWS para acceder a dynamodb, darle sólo permiso para nuestras tablas.
- Añadir endpoint get all para el carrito de la compra.
- Hacer proyecto purchases multimodulo en lugar de usar tres directorios diferentes para la arquitectura hexagonal.
- Mejorar la validación del token en purchases.
- Usar operador de kafka en lugar de statefulsets.
- Añadir la funcionalidad pensada inicialmente y recortada por falta de tiempo:
    - Servicio notificaciones
        - Lambda de AWS implementado con node escuchando en una cola SQS.
        - Envío de mails vía Amazon SES.
        - Cada micro tendrá Feature toggles para poder activar/desactivar las notificaciones.
    - Servicio usuarios:
        - Cuando el usuario se registra, si las notificaciones están activas, se le mandará un mail indicando el registro:
            - Envío de evento a la cola SQS de notificaciones.
    - Servicio de compra
        - Si las notificaciones están activadas para el micro, se manda evento a SQS la cola de notificaciones.
        - Añadir dirección de entrega al pedido.
    - Servicio de entrega
        - Funcionalidad:
            - Recibe eventos desde compra.
            - Lo almacena en BD en estado PENDING.
            - En función de la provincia de entrega indicada en el proceso de compra simula un tiempo de preparación, al finalizar dicho tiempo, actualiza el estado a PACKED.
            - En función de la provincia de entrega indicada en el proceso de compra simula un tiempo de envío, al finalizar dicho tiempo, actualiza el estado a SENT.

                - Cada cambio de estado del pedido se actualizará a tiempo real mediante websockets en la web de seguimiento de pedidos del usuario. Por tanto se proporcionará un endpoint API Rest que devuelva el estado de los pedidos en curso (no enviados).
                - Cada cambio de estado del pedido, si las notificaciones están activadas para el micro, se enviará un mensaje a la cola SQS para notificar los cambios de estado al usuario vía mail.
            - Arq hexagonal
            - Desarrollo en Spring Boot
            - BD DynamoDB con reactividad.
        - Frontal Web
            - Usando los endpoints suministrados mediante el API Gateway anterior permite:
                - Acceder al usuario para realizar compras
                - Listar productos en stock.
                - Añadir productos al carro.
                - Comprar los productos.
                - Hacer seguimiento en tiempo real (websockets) del estado de sus pedidos.
- Pruebas de carga con la herramienta Artillery.
- Incorporar patrones de observabilidad:
    - Métricas de la aplicación
        - Prometheus & Grafana
    - Agregación de logs
        - ELK/Grafana+Loki
    - Trazas distribuidas
        - Zipkin
    - Investigar OPEN TELEMETRY.
- Añadir Sonar al flujo de CI/CD.

6. Bibliografía

Open API

https://swagger.io/resources/articles/adopting-an-api-first-approach/
https://editor.swagger.io/

Testcontainers

https://github.com/Trendyol/jest-testcontainers
https://www.testcontainers.org/
https://www.testcontainers.org/modules/docker_compose/
https://github.com/testcontainers/testcontainers-java/blob/master/modules/junit-jupiter/src/test/java/org/testcontainers/junit/jupiter/ComposeContainerTests.java

ESLint

https://lenguajejs.com/javascript/caracteristicas/eslint/
https://medium.com/the-node-js-collection/why-and-how-to-use-eslint-in-your-project-742d0bc61ed7
https://ichi.pro/es/como-agregar-codigo-linting-para-un-proyecto-de-node-js-195785420807381

Git hooks

https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks
https://www.npmjs.com/package/husky
https://github.com/phillipuniverse/githook-maven-plugin

Docker

https://hadolint.github.io/hadolint/
https://hub.docker.com/

Github actions

https://docs.github.com/en/actions
https://github.com/marketplace/actions/build-and-push-docker-images
https://github.com/marketplace/actions/helm-deploy
https://github.com/marketplace/actions/publish-release
https://github.com/marketplace/actions/release-changelog-builder
https://github.com/phips28/gh-action-bump-version
https://docs.github.com/en/actions/publishing-packages/publishing-java-packages-with-maven

Kubernetes

https://minikube.sigs.k8s.io/docs/start/
https://okteto.com/

Helm

https://helm.sh/
https://codersociety.com/blog/articles/helm-best-practices

Packages

https://docs.github.com/en/packages/quickstart

Spring Cloud Gateway

https://spring.io/projects/spring-cloud-gateway#overview
https://spring.io/guides/gs/gateway/

Checkstyle

https://checkstyle.sourceforge.io/
https://www.baeldung.com/checkstyle-java

Pact

https://docs.pact.io/
https://docs.pact.io/implementation_guides/jvm/consumer/junit5
https://hmh.engineering/how-to-write-and-validate-pact-contracts-using-junit5-and-restassured-72b578e7dd65
https://levelup.gitconnected.com/consumer-tests-with-pact-junit5-springboot-b9dff34aa302

Wiremock

https://wiremock.org/docs/junit-jupiter/

Jib

https://github.com/GoogleContainerTools/jib
https://github.com/GoogleContainerTools/jib/tree/master/jib-maven-plugin

Maven

https://github.com/phillipuniverse/githook-maven-plugin
https://github.com/GoogleContainerTools/jib/tree/master/jib-maven-plugin
https://maven.apache.org/maven-release/maven-release-plugin/examples/prepare-release.html

Dynamodb

https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GettingStarted.NodeJs.html
https://www.npmjs.com/package/dynamodb-atomic-counter

Kafka

https://www.baeldung.com/spring-kafka
https://developer.confluent.io/quickstart/kafka-on-confluent-cloud/?utm_medium=sem&utm_source=google&utm_campaign=ch.sem_br.nonbrand_tp.prs_tgt.kafka_mt.xct_rgn.emea_lng.eng_dv.all_con.kafka-general&utm_term=apache+kafka&placement=&device=c&creative=&gclid=CjwKCAjw9-KTBhBcEiwAr19ig6LPZ2uCmKN-zlCLkUXtf0ffnFmxmtK_kuDLt2JexWfiR31ULoqKrBoChUwQAvD_BwE
https://kafka.js.org/docs/consumer-example

Campos JSON en MySQL

https://prateek-ashtikar512.medium.com/how-to-handle-json-in-mysql-4adaeeb1d42f
https://medium.com/javarevisited/spring-boot-testing-testcontainers-and-flyway-df4a71376db4