Informe Seguimiento 2 Aplicación de Gestión Financiera

Rafael Gómez Pérez
Mención en Ingeniería del Software

Índice de contenidos

1.Histórico de revisiones

Fecha	Versión	Descripción	Autor
22/05/2019	1.0	Creación del informe de progreso 2	Rafa
24/05/2019	1.1	Resultados provisionales medianamente terminados	Rafa
25/05/2019	1.2	Resultados provisionales terminados	Rafa

2 Objetivos del proyecto

Dado que nos encontramos ante el segundo informe de progreso, hemos creído oportuno mostrar de una forma algo mas visual la madurez de los objetivos establecidos al principio del proyecto.

La tabla a continuación muestra los códigos de los objetivos definidos previamente junto con un porcentaje aproximado que representa cuanto trabajo representado por cada objetivo esta actualmente completado.

Objetivo	Grado de Madurez
tfg-obj-01	80%
tfg-obj-02	100%
tfg-obj-03	50%
tfg-obj-04	40%
tfg-obj-05.1	45%
tfg-obj-06	70%
tfg-obj-07	0%

3 Planificación del Proyecto

En el apartado de diseño y arquitectura cubriremos las diferentes fases del proceso de diseño que se han realizado durante el período de tiempo entre este informe y el anterior.

En cuanto a la planificación del proyecto, actualmente nos encontramos entre 2 y 5 días por detrás *9de lo previsto inicialmente.

Esto se debe a la dificultad y la falta de experiencia en tecnologías como Docker Swarm, Spring Cloud (Netflix) y Spring Webflux. La falta de experiencia nos ha llevado a tener que realizar una cantidad de investigación mucho mayor a la esperada inicialmente, lo cual nos ha llevado a ralentizar el ritmo de trabajo.

Así mismo, la falta de madurez e información online con algunas de estas tecnologías Spring Webflux y Project Reactor (ambas lanzadas a principios de 2018) nos ha complicado y ralentizado el proceso de desarrollo.

4 Metodología de trabajo

En cuanto a metodología de trabajo, seguimos utilizando la misma que se comentó durante los informes previos (Kanban con feature-oriented). Pero, a diferencia de lo que se comentó en los informes mencionados se ha decidido realizar unos cambios en la parte de feature-oriented.

Anteriormente comentamos que nuestra implementación de feature-oriented sería la siguiente:

- Dividir el proyecto en funcionalidades claramente definidas.
- Separar y estimar cada funcionalidad según su impacto en cada lado de la aplicación Frontend o Backend.
- Agregar todas las tareas al backlog de Kanban.
- Escoger una funcionalidad e implementarla en ambos lados del proyecto antes de pasar a la siguiente.

Tras ver el tiempo restante y la dificultad que nos está llevando desarrollar la parte de backend, hemos decidido que cambiar nuestra implementación de feature-oriented a la siguiente:

- Dividir el proyecto en funcionalidades claramente definidas.
- Separar y estimar cada funcionalidad según su impacto en cada lado de la aplicación Frontend o Backend.
- Agregar todas las tareas al backlog de Kanban.
- Escoger un lado del proyecto (Front o Back) y desarrollar todas sus funcionalidades (Priorizamos Backend ya que existe una dependencia desde Front que necesita tener Back hecho).

Como podemos observar, la diferencia no mucha pues seguimos implementando feature-oriented programming simplemente modificamos como escoger funcionalidades a implementar.

5 Implementación de microservicios

Como pudimos observar en el informe anterior, el proyecto consta de varios microservicios (consular anexo 2ª imagen).

Respecto a su implementación, originalmente comentamos que el plan consistía en desarrollar la parte de Backend de forma monolítica para después de implementar el Frontend, realizar una migración hacía microservicios.

Tras pensar en otras alternativas, hemos decidido que creemos que es la mejor forma de hacerlo. Dada la casi inexistente experiencia laboral en entornos con arquitecturas basadas en microservicios con la que contamos, cabe la posibilidad que no hayamos podido realizar toda la migración de los microservicios antes de la fecha final del proyecto.

Teniendo esto en cuenta, creemos que el poder presentar un proyecto implementado de forma monolítica (en el peor de los casos) o híbrida en caso de no poder terminar la migración completa, nos beneficiará mucho mas que el poder mostrar todos los microservicios implementados y una aplicación (producto final) a medio implementar o con funcionalidad reducida significativamente.

6 Exposición de resultados

Para explicar los resultados hemos de explicar previamente de manera general la arquitectura básica de un proyecto en Spring.

Spring basa su arquitectura separando el proyecto en diferentes capas, o layers. Entre estas layers encontramos Controller, Service y Repository como layers principales, aunque tambien intervienen otras como el Command o los Converters durante todo el ciclo de vida de la petición.

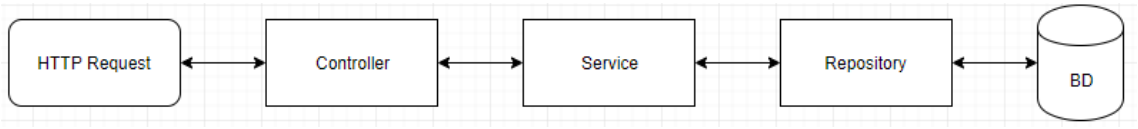
El Controller se encarga de mapear la request a un EndPoint que nuestra aplicación está exportando en la API Rest a través de su URL. Mapea y redirecciona la request y sus datos hacía el EndPoint cuya URL coincida y, si es necesario, hace comprobaciones adicionales (permisos, roles, restricción de acceso…).

El Service es la capa donde se ejecuta toda la denominada “Bussiness Logic” o Lógica de Negocio. Por esto nos referimos a todas las transformaciones de datos u operaciones específicas que se deban a los requerimientos del proyecto. En esta capa es donde se suele utilizar los Converters para pasar de un DTO (o Command) a un POJO (Plain Old Java Object) con el que trabajaremos.

El Repository es la capa del proyecto que interactua con la base de datos (DB), su principal funcion es realizar operaciones CRUD sobre la DB designada del proyecto.

En **Spring Webflux** contamos de dos formas diferentes de implementar una HTTP Request:

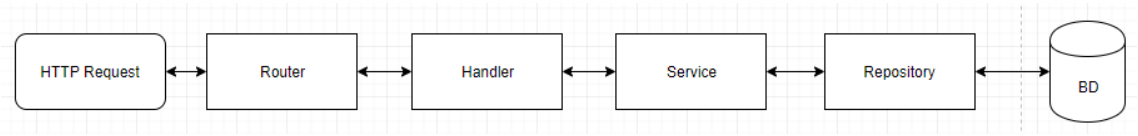
Podemos implementarla de forma similar a las anteriores versiones de Spring (con un **RestController**) de modo que la arquitectura de la Request quedaría de la siguiente forma.



Por otro lado, podemos utilizar los Routers o Handlers que introduce Spring Webflux para realizar esta misma implementación de una forma más funcional.

Spring Webflux añade una capa mas a la solución previa de Spring que consiste en dividir la funcionalidad del Controller en dos capas diferentes (Router y Handler). El Router, como su nombre indica, se encarga de redireccionar la petición HTTP hacía el EndPoint pertinente y en específico hacía el Handler pertinente. Por otro lado, el Handler es quien conoce que operaciones específicas han de realizarse para cada EndPoint.

Por lo que utilizando estas nuevas capas nos quedaría una funcionalidad como esta:



Tras todo esto uno se pregunta bueno pero que ventajas aporta el uso de incrementar el número de capas y clases a crear. A fecha de creación de este documento (2019), no existen ventajas de utilizar Routers y Handlers más allá de reducir la cantidad de código que se escribe, aunque, debido a la madurez de estas nuevas funcionalidades, perdemos muchas cosas por usarlas (Swagger, Validators) son entre otras, cosas que no podemos utilizar con esta nueva arquitectura almenos por el momento.

Debido a esto, hemos optado por utilizar Routers y Handlers solo en los servicios más simples y hemos dejado Controllers en los servicios más complejos. Como entendemos que todos estos conceptos pueden ser un poco sobreacogedores al principio, haremos un ejemplo paso a paso de cada una de las arquitecturas del proyecto.

6.1 Resultados Arquitectura de Controller

En este apartado, seguiremos el ciclo de vida de una petición HTTP en una implementación con Controller explicando en cada paso que estamos haciendo.

```java
@RestController
@EnableAutoConfiguration
public class GroupController {

    private final GroupService groupService;

    @Autowired
    public GroupController(GroupService groupService) { this.groupService = groupService; }

    @PostMapping(value = "groups/{groupId}/user/add", headers ="Accept=application/json")
    public Mono<ResponseEntity<Group>> addUserToGroup(@PathVariable("groupId") String groupId,
                                                      @RequestBody GroupUser groupUser) {
        return groupService.addUserToGroup(groupId, groupUser);
    }
```

Como hemos comentado en otros informes, Spring se basa en anotaciones (@RestController) para ayudarnos a reducir enormemente la cantidad de código que hemos de poner nosotros como desarrolladores. En este snippet podemos observar unas cuantas como son @RestController (le indica a Spring que esta clase es un Controller y por tanto es donde deben llegar las request), @Autowired es una de las mas comunes y se encarga de inyectarnos la dependencia en nuestra clase. @PostMapping se encarga de exponer este EndPoint en la URL especificada con los headers especificados y solo si el método es POST.

```java
public interface GroupService {

    Mono<Group> createGroup(Group command);
    Mono<Group> findById(String id);
    Flux<Group> findAll();
    void deleteById(String id);
    Mono<ResponseEntity<Group>> addUserToGroup(String groupId, GroupUser groupUser);
    Mono<ResponseEntity<Void>> deleteGroup(String groupdId);
    Mono<ResponseEntity<Group>> deleteUserFromGroup(String groupId, ObjectId userId);
    Mono<ResponseEntity<Group>> addExpensesToGroup( GroupExpense groupExpenses, String groupId);
    Mono<ResponseEntity<Group>> deleteExpensesFromGroup(ObjectId groupExpensesId,  String groupId);
    Mono<ResponseEntity<Group>> updateExpensesFromGroup(GroupExpense groupExpenses, String groupId);
    Mono<ResponseEntity<Group>> addAdminToGroup(GroupUser admin, String groupId);
    Mono<ResponseEntity<Group>> removeAdminFromGroup(ObjectId admin, String groupId);
}
```

Algo que no hemos comentado anteriormente y que hace referencia a los objetivos **tfg-obj-03** y **tfg-obj-04** es el uso de Interfaces como esta para declarar todos los métodos que nuestra capa de Service contiene.

Spring, por su cuenta, se encarga de buscar en el mismo package todas las clases que correspondan con el nombre GroupServiceImpl para realizar un @Autowire automático de todas sus dependencias.

```java
@Service
public class GroupServiceImpl implements GroupService {

    private final GroupRepository groupRepository;
    private final UserService userService;

    @Autowired
    public GroupServiceImpl(GroupRepository groupRepository, UserService userService) {
        this.groupRepository = groupRepository;
        this.userService = userService;
    }

    @Override
    public Mono<Group> createGroup(Group group) {
        return groupRepository.insert(group);
    }
```

Como hemos descrito previamente, la capa del Service se encarga de realizar toda la lógica del servicio, pero, teniendo en cuenta, que este caso se trata de un insert y el objeto ya viene completo no hemos de realizar nada mas que la llamada al repositorio.

```java
@Repository
public interface GroupRepository extends ReactiveMongoRepository<Group, String> {

}
```

Una de las ventajas de Spring es el completar funcionalidades automáticamente en tiempo de ejecución, en este caso nosotros no hemos de realizar nada mas que declarar nuestra interfaz del Repository y al extender de la clase ReactiveMongoRepository ya tenemos “Out of the box” las siguientes funcionalidades (entre otras).

- findAll → Encuentra todos los elementos en un Documento (MongoDB)
- findById → Encuentra el elemento con el id como parámetro
- deleteById → Elimina el elemento con el id como parámetro
- save → Busca si el objeto pasado existe si es así lo modifica, si no lo inserta.
- saveAll → Guarda/Modifica todos los objetos pasados en la DB
- insert → Inserta en la DB el objeto pasado como parámetro
- insertAll → Inserta en la DB los objetos pasados como parámetros
- existsById → Comprueba si existe algo registrado con ese ID en la DB

6.2 Resultados Arquitectura con Routers y Handlers

En este apartado, analizaremos el ciclo de vida de una petición HTTP implementando la arquitectura de Spring Webflux con Routers and Handlers.

```
@Configuration
public class UserRouter {

    @Bean
    public RouterFunction<ServerResponse> userRoute(UserHandler userHandler) {
        return RouterFunctions
                .route(POST( pattern: "/user/create").and(accept(MediaType.APPLICATION_JSON)),
                        userHandler::createUser)
                .andRoute(GET( pattern: "/user").and(accept(MediaType.APPLICATION_JSON)),
                        userHandler::getAllUsers)
                .andRoute(GET( pattern: "/user/{userId}").and(accept(MediaType.APPLICATION_JSON)),
                        userHandler::getUserById)
                .andRoute(POST( pattern: "/user/update").and(accept(MediaType.APPLICATION_JSON)),
                        userHandler::updateUser)
                .andRoute(DELETE( pattern: "/user/delete/{userId}").and(accept(MediaType.APPLICATION_JSON)),
                        userHandler::deleteUser)
                .andRoute(POST( pattern: "/user/{userId}/image/add").and(accept(MediaType.MULTIPART_FORM_DATA)),
                        userHandler::uploadProfileImage)
                .andRoute(GET( pattern: "/user/{userId}/image/{imageId}").and(accept(MediaType.APPLICATION_JSON)),
                        userHandler::getProfileImage);
    }
}
```

Como podemos observar, toda la funcionalidad encargada de redireccionar la petición hacía el EndPoint pertinente se encuentra en una sola función. Cada método route de nuestro Router se encarga de redireccionar la peticion del tipo que indica (POST/GET/PUT/DELETE) que contenga los headers establecidos hacía el método del handler correspondiente.

Como podemos observar en el handler, primero recuperamos la variable que hemos pasado por la URL (Id del usuario) y procedemos a recuperar el objeto User que corresponde con ese ID.

```java
@Component
public class UserHandler {

    private UserService userService;

    static Mono<ServerResponse> notFound = ServerResponse.notFound().build();

    @Autowired
    public UserHandler(UserService userService) { this.userService = userService; }

    public Mono<ServerResponse> getUserById(ServerRequest serverRequest) {
        String id = serverRequest.pathVariable( name: "userId");
        return userService.findById(id)
                .flatMap(user -> ServerResponse.ok()
                        .contentType(MediaType.APPLICATION_JSON)
                        .body(fromObject(user)))
                .switchIfEmpty(notFound);
    }
```

Una vez hemos recuperado ese User, mediante un flatMap creamos una ServerResponse donde le añadimos el resultado de nuestra consulta a BD como body. Recordamos que estamos trabajando con Spring Webflux que se basa en Reactive Programming por lo que es Asíncrono. Esto significa que, mientras se ejecuta el userService ya estamos comenzando a realizar el flatMap.

```java
@Override
public Mono<User> findById(String id) {
    return userRepository.findById(id);
}
```

Esta parte del Service es la que se esta ejecutando en un thread, mientras el Handler va ejecutando el resto del código en otro. En el momento en que el Service obtiene resultado, éste es enviado al thread del Handler.

```java
@Repository
public interface UserRepository extends ReactiveMongoRepository<User, String> {

    Mono<User> findByUsername(String username);
}
```

Como podemos observar a simple vista, este Repository tiene una diferencia respecto al anterior, y ésta no se debe a las diferentes arquitecturas que estamos implementando. En todas las interfaces de los Repository, si declaramos un método con los "keywords" permitidos seguido con un atributo del objeto en la base de datos Spring se encargará de implementar este método en tiempo de ejecución.

6.3 Resultados tangibles del proyecto

Ahora que hemos visto (un poco por encima) como funciona la arquitectura de Spring Webflux, podemos entender algo mejor los resultados del proyecto hasta la fecha.

Actualmente el proyecto se divide en dos partes, Frontend y Backend. El apartado de Frontend aun no ha sido empezado, por otro lado, el apartado de Backend se encuentra completamente desarrollado y desplegado.

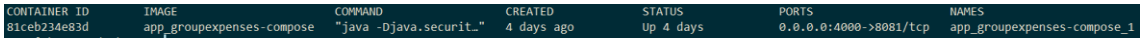
De cara a desplegar el proyecto en el cloud, hemos decidido utilizar DigitalOcean como Cloud Provider. DigitalOcean ofrece muchas funcionalidades "Out of the Box" como creacion de clusters, auto instalación de servicios a gusto del consumidor y, lo mas importante, es muy económico.

Para desplegar el proyecto, hemos utilizado Docker, en específico docker-compose.

```yaml
version: '2.0'
services:
  groupexpenses-compose:
    build: .
    ports:
      - 4000:8081
```

En nuestro caso, debido a que no queremos complicar el proyecto más de lo necesario, hemos creado un docker-compose bastante sencillo.

Una vez la imagen ha sido creada, se sube a DockerHub (repositorio de imágenes de Docker). Para posteriormente, descargar la imagen en el servidor y ejecutar el contenedor de Docker.



Como podemos observar, el contenedor del proyecto esta ejecutándose por lo que ya tenemos desplegado

7 Conclusiones provisionales

Como conclusiones del proyecto (por el momento), podemos observar que la dificultad del proyecto y las tecnologías que están siendo utilizadas pueden haber sido algo mayores de lo que nos parecía inicialmente antes de ponernos a trabajar con ellas.

Creemos que la dificultad del proyecto no solo radica en las tecnologías con las que será implementado, sino en su propia lógica de negocio a pesar de que pueda parecer fácil a simple vista. Hasta el punto de haber tenido que incrementar la dificultad de este apartado para conseguir la aprobación del proyecto, a sabiendas, de que sería imposible implementarlo.

Resumiendo, creemos que hemos podido pecar de ser algo ambiciosos con querer trabajar con tecnologías muy punteras en el sector del desarrollo del Software, pese a no tener experiencia con la mayoría de ellas y eso nos está volviendo en forma de problemas, bugs y retraso en la planificación inicial.

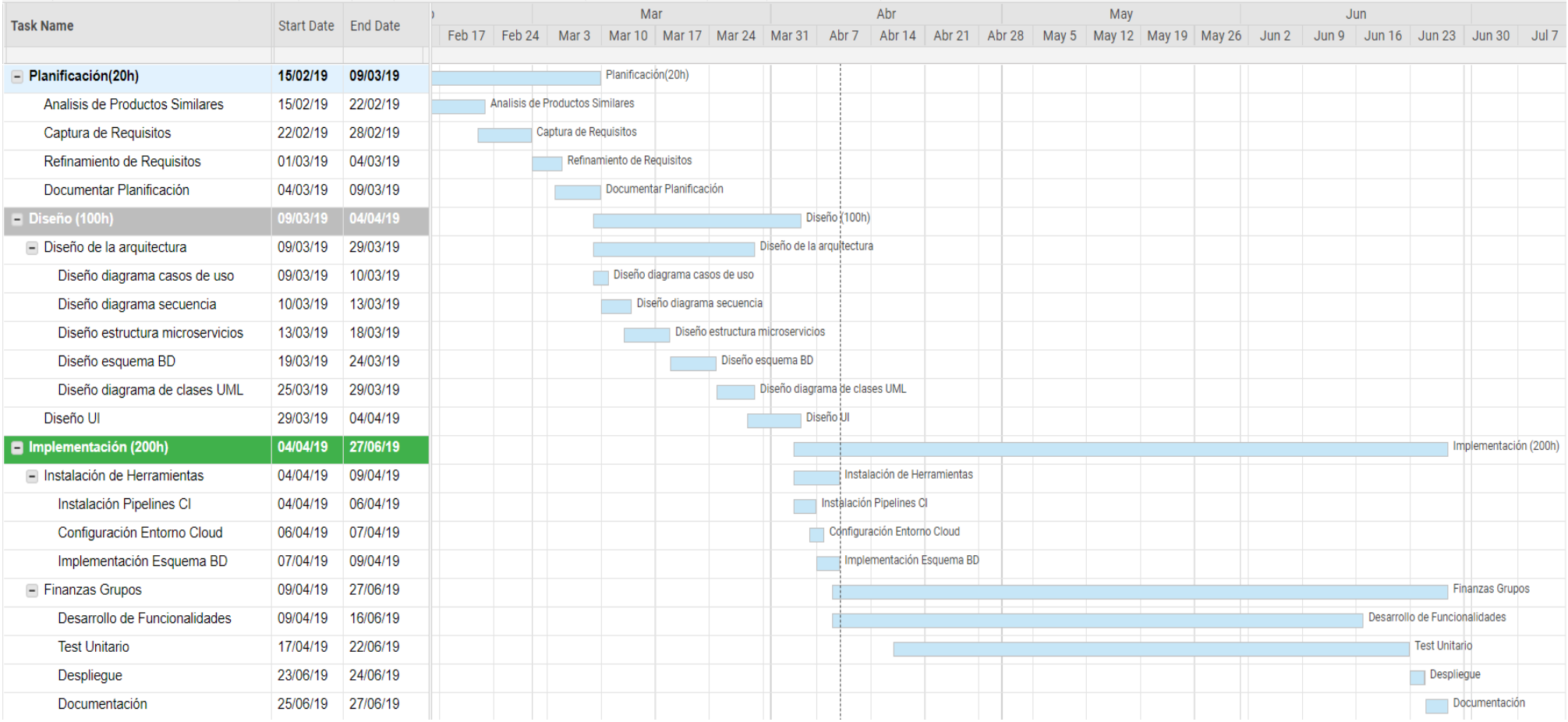
8-Listado de Acrónimos

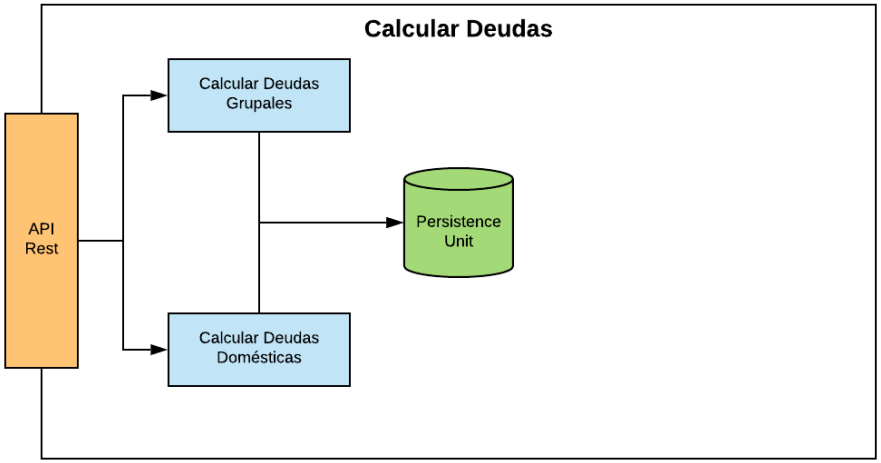
Abreviatura	Significado
DTO	Data Transfer Object
POJO	Plain Old Java Object
URL	Uniform Resource Locator
CRUD	Create, Read, Update, Delete operations
DB	Database
API	Application Programming Interface
HTTP	Hypertext Transfer Protocol

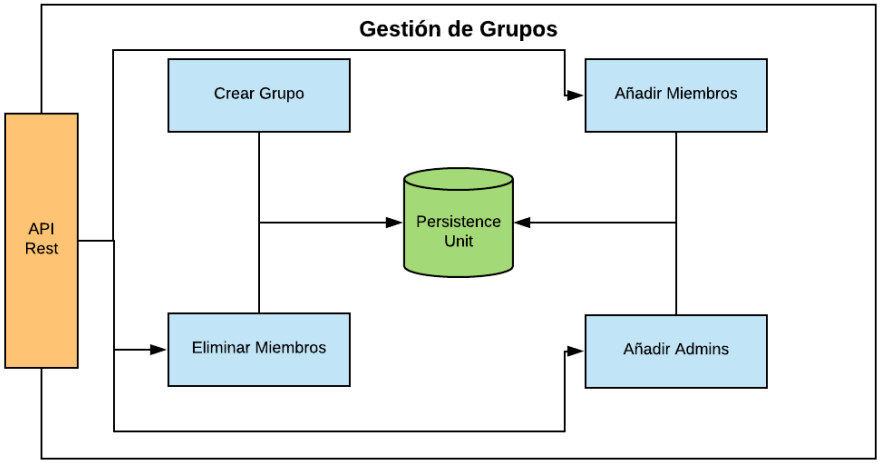
9-Referencias y Bibliografía

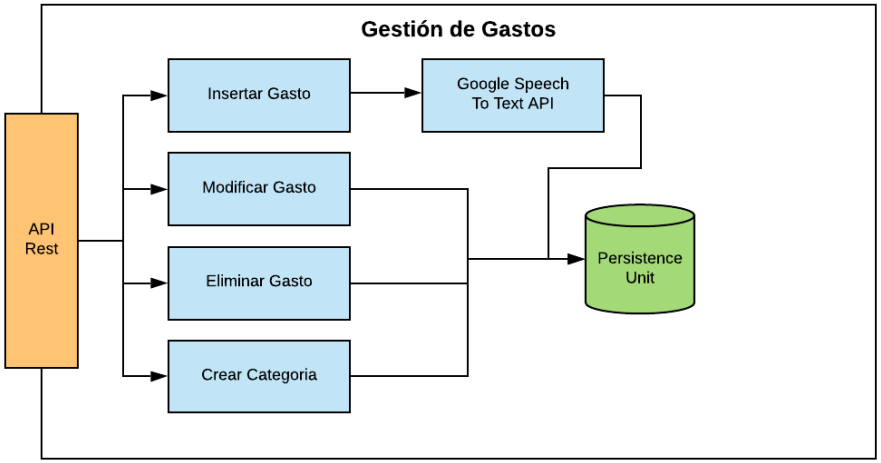
[1] https://docs.docker.com/engine/swarm/, Mayo 2019
[2] https://spring.io/projects/spring-cloud, Mayo 2019
[3] https://www.baeldung.com/spring-webflux, Mayo 2019
[4] https://projectreactor.io/, Mayo 2019
[5] https://www.idento.es/blog/desarrollo-web/que-es-una-api-rest/, Mayo 2019
[6] http://www.robertocrespo.net/kaizen/implementar-microservicios-spring-boot-iv-documentar-apis-rest-swagger/, Mayo 2019
[7] https://www.digitalocean.com/community/tutorials/, Mayo 2019
[8] https://www.techopedia.com/definition/133/cloud-provider, Mayo 2019
[9] https://hub.docker.com/, Mayo 2019

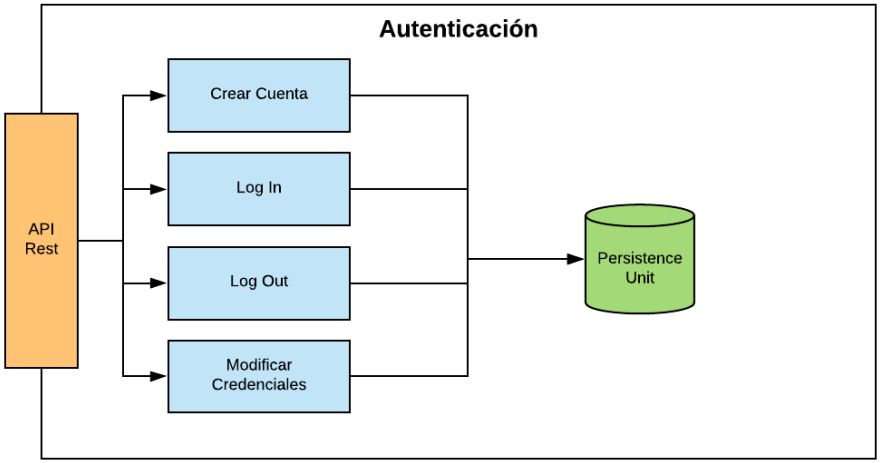
Anexo


Task Name
Start Date
End Date
Mar
Abr
May
Jun
Feb 17
Feb 24
Mar 3
Mar 10
Mar 17
Mar 24
Mar 31
Abr 7
Abr 14
Abr 21
Abr 28
May 5
May 12
May 19
May 26
Jun 2
Jun 9
Jun 16
Jun 23
Jun 30
Jul 7
Planificación(20h) 15/02/19 09/03/19
Analisis de Productos Similares 15/02/19 22/02/19
Captura de Requisitos 22/02/19 28/02/19
Refinamiento de Requisitos 01/03/19 04/03/19
Documentar Planificación 04/03/19 09/03/19
Diseño (100h) 09/03/19 04/04/19
Diseño de la arquitectura 09/03/19 29/03/19
Diseño diagrama casos de uso 09/03/19 10/03/19
Diseño diagrama secuencia 10/03/19 13/03/19
Diseño estructura microservicios 13/03/19 18/03/19
Diseño esquema BD 19/03/19 24/03/19
Diseño diagrama de clases UML 25/03/19 29/03/19
Diseño UI 29/03/19 04/04/19
Implementación (200h) 04/04/19 27/06/19
Instalación de Herramientas 04/04/19 09/04/19
Instalación Pipelines CI 04/04/19 06/04/19
Configuración Entorno Cloud 06/04/19 07/04/19
Implementación Esquema BD 07/04/19 09/04/19
Finanzas Grupos 09/04/19 27/06/19
Desarrollo de Funcionalidades 09/04/19 16/06/19
Test Unitario 17/04/19 22/06/19
Despliegue 23/06/19 24/06/19
Documentación 25/06/19 27/06/19


Calcular Deudas
API Rest
Calcular Deudas Grupales
Calcular Deudas Domésticas
Persistence Unit


Gestión de Grupos
API Rest
Crear Grupo
Añadir Miembros
Persistence Unit
Eliminar Miembros
Añadir Admins


Gestión de Gastos
API Rest
Insertar Gasto
Google Speech To Text API
Modificar Gasto
Eliminar Gasto
Crear Categoria
Persistence Unit


Autenticación
API Rest
Crear Cuenta
Log In
Log Out
Modificar Credenciales
Persistence Unit