# Corp – Payment Manager

## Objectivo:

La aplicación entiendo que simula la función de un API Gateway Pattern.  
El servicio atiende a un consumidor y por detrás consume otros recursos, que pueden estar en distintas configuraciones, protocolos, etc.  
  
El PM intenta procesar los request y enviar al cliente la información esencial lo mas rápido posible, para esto guarda las acciones en su tabla la cual usa como de cola de espera de las operaciones que luego van ocurrir.

En este caso es procesar las transacciones y cambiar el estado del balance.  
  
  
Implementacion

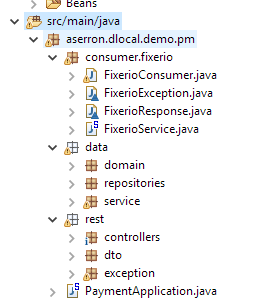
### Base de datos:

Dos tablas que están relacionadas, pero esta relación no se usa, ya que “merchant” esta en otra app, que puede no tener la misma base de datos.  
  
El id de Sales, es Binary(16) guardando transparentemente ids del UUID generadas por hibernate, el id se genera en la plataforma con el generador uui2 de hibernate, que utiliza el nombre y el clock para generar el hash.  
También usan el tipo UUID de forma automática.

## Applicaciones

### PM

La estructura

La clase principal es el contexto de spring.

Este conecta un ExceptionHandler general al inicio antes de pasar la creación de los controladores.

El RestController, es el contexto donde al final van a estar los objetos principales.

Este inicia el servicio REST y crea la instancia de los servicios.

Estos son TaskScheduler, SalesService y MerchantService.

Es algo común a las dos aplicaciones que el controlador solo tenga la lógica relativa a la creación y destrucción de los servicios y como vincular los servicios con la api.

La lógica de negocios esta en los servicios.

La capa de datos esta compuesta por Repositorios y Entidades, la mayor parte de la funcionalidad se logra configurando hibernate y spring data.

De esta forma se puede lograr por ej pedir un Sale por merchant y transaction id, sin escribir la query usando un patrón semántico y definiendo el método en la interface.

Se ocupa el tiempo preparando el dato de forma óptima para que se pueda maximizar el uso automático de hibernate y spring data.

De esta forma los servicios son los que operan los repositorios y de esta forma se separa los datos de la capa del controlador, en esta se usan POJOS y la mismas entidades como validadores y como forma de asegurar un contrado en las respuestas.  
  
Por separación de responsabilidad SaleCreationRequest q es un POJO anotado con Jackson para serializar y validar la entrada está en el paquete de rest y no en el paquete domain.

Se viola por conveniencia esta separación cuando se usan las entidades como validador y como formato de serializado en el controlador. Por conveniencia, aunque no tiene por qué ser así ya que la mayoría de la validación está disponible por Jackson.

#### SaleService

Sale es un clásico CrudRepo, que implementa hibernate y JPA.

SaleService > SaleRepository > Sale seria como se van moviendo las responsabilidades.

La entidad es donde esta la mayor configuración ya que usando anotaciones se puede lograr que hibernate nos de le resultado que esperamos.

Tenemos generación de uuid nivel 4, validación de los tipos, con esta se puede validar la entrada del post.  
  
El generar el id local, sumado al uso de UUID hace que ni siquiera sea necesario que se haya hecho una persistencia, ya q el binomio va a dar resultado al existir una entidad con los datos merchant\_id y transaction\_id.

La excepciones se usan en el controlador y en el servicio para marcar las condiciones que se piden, como la existencia de merchant.

### Fixer.io

El task scheduler usa el FixerioService para lograr que se pueda consumir el servicio REST remoto.

El servicio enmascara la funcionalidad de la llamada, las clases de soporte como FixerioException están todas ahí en el mismo paquete ya que son una funcionalidad modular.

Aparte de la llamada al REST, el servicio ofrece la conversión de la moneda origen a usd.

Se usa BigDecimal para guardar las monedad y asegurar las operaciones como el cambio y la conversión que son sensibles a la precisión, se usa 3 puntos para asegurar que tengamos los 2 lugares necesarios.

Se guarda el tiempo y se lanza una excepción si el servicio no puede ser llamado.

### MerchantService

Implementa el consumo del REST local y la acción “check”

Si bien están implementadas las entidades necesarias para generar una estructura local de los

Merchant, esta no es necesaria. Esta para generar un display rico en la lista de sale, actualmente está desactivado para apegarse a la letra.

### TransactionJobService

Se podría proponer que el TaskSch debería estar en el contexto de app, pero como el mismo usa el servicio de Sale, para evitar problemas lo puse en ese contexto donde puede acceder al servicio de sale sin crear una instancia nueva y sin usar interceptors.

Es un scheduler que al instanciarse en el controller rest es conectado el Task Runner y ejecutado cada 30 segundo.

Este enmascara las operaciones de la acción “balance”

También usa BigDecimal para la moneda.

### Error Handling

ControllerAdivice es implementado y los errores del controlador REST son atapados en esta clase, el paquete rest.exception se dedica a estas tareas.

En resumen, el controlador REST debe tomar los llamados, lanzar excepciones si los parámetros, la llamada o alguna condición no se respetan, cambiando códigos como BAD\_REQUEST, más allá del clásico 201 o 404.

Los servicios, los repos, pueden también lanzar excepciones una vez que el contexto se mueva hacia ellos.

Si bien hay un error handler la idea es que las excepciones fluyan en la dirección del contexto de ejecución, de forma que se pueda aprovechar por ej. el error de validación del repo y traducirlo en el extremo a un “bad request”.

## Aplicación Merchant

Más simple de explicar que la anterior, sigue el mismo patrón, dominio, excepciones, rest están en dominios separados.

El controlador REST es el contexto de ejecución base y se implementa como antes configurando hibernate, aquí hay mucho código que no tiene una influencia real porque es donde estaba armando un control unificado para el REST que se pudiera usar en muchos casos y en ambas aplicaciones.

Se destaca aquí la intención de tener un control sobre como vemos las excepciones de forma unificada y coherente.

Esta la implementación que permite ver la cola de errores aparte de la información propia de la excepción.

Todo en formato JSON para poder dar soporte.

## Testing.

Estudie e incluso hay un test que esta desactivado que logra hacer un request y testear la respuesta.

Si bien se puede hacer test unit sobre los servicios creo que lo que precisaban era el teste de integración con Mockito.

La idea es inicializar la aplicación en un contexto falso simulando que esta online, este contexto envuelve la aplicación permitiendo llamar a los servicios a usando requests simuladas.

Esto se puede usar para chequear las restricciones y condiciones, ej. llamar al servicio fixer más rápido que 30 segundos, esto requiere de más configuración porque hay que simular el contacto con el servidor para luego obtener un resultado del controlador.

En el peor de los casos la tarea es crear el contexto de ejecución con todos los elementos para que la aplicación ejecute la acción sin tener ningún error y al mismo tiempo que la acción se ejecute con realismo.

## Más allá de lo que se pide.

La aplicación podría implementar una cola en memoria para controlar como se está llamando al REST, se podría hacer llamado asincrónico en la llamada a recursos externos.

Se podrían implementar dos error handler, uno específico para la capa de servicios y otro para la capa REST, de esta forma se podría capturar un error de conexión a la basa de datos que generalmente es crítico y termina en la aplicación no pudiendo arrancar.

### Nota: DB

Hibernate 5.x y MySQL 8 ofrecen el soporte necesario para el manejo de la columna con id UUID.

El configurar hibernate para que opere transparente con el tipo UUID de java 8 hace que no sea necesario crear traductores e implementar los métodos de testeo de identidad, la UUID de nivel 4 que se usa toma el address y el clock com parte del hash haciendo imposible que se creen dos entidades, aunque sea con mínimo instante, habría que probar que sucede con tiempos muy chicos, pero sería un test sintético.

Se llegó a implementar uso de UUID en el lugar donde se necesitaba para la impotencia.

Se entiende que merchant y transaction podrían tener estos ids.

Y se puede pensar que también por ej. en el caso de merchant, tal vez de podría trabajar sin saber de esos ids, solo con el nombre.

### Nota: Código sin uso

Hay código que estaba a prueba, especialmente en el área de los error handlers y las clases asociadas a estos, que no tienen mucha influencia en la lógica de la app.

La idea era generar una estructura concreta para el manejo de REST, esto sería implementar como mínimo el RestControllerAdvice que tiene una lista linda de métodos.

Otro es el EntityResponseControllerAdvice, este tiene enfoque al tipo de respuesta que se tiene cuando un validador de Jackson es capturado en el controller.

De lograr esto se pude lograr una respuesta controlada y que nos permite trabajar más rápido al evitar que la aplicación se caiga por excepciones a veces obvias.

También podemos como se logró en merchant tener una vista rica en el REST del error y de cola de errores previos.