Consideraciones de diseño

Descripción general

La solución fue desarrollada en Spring Boot.

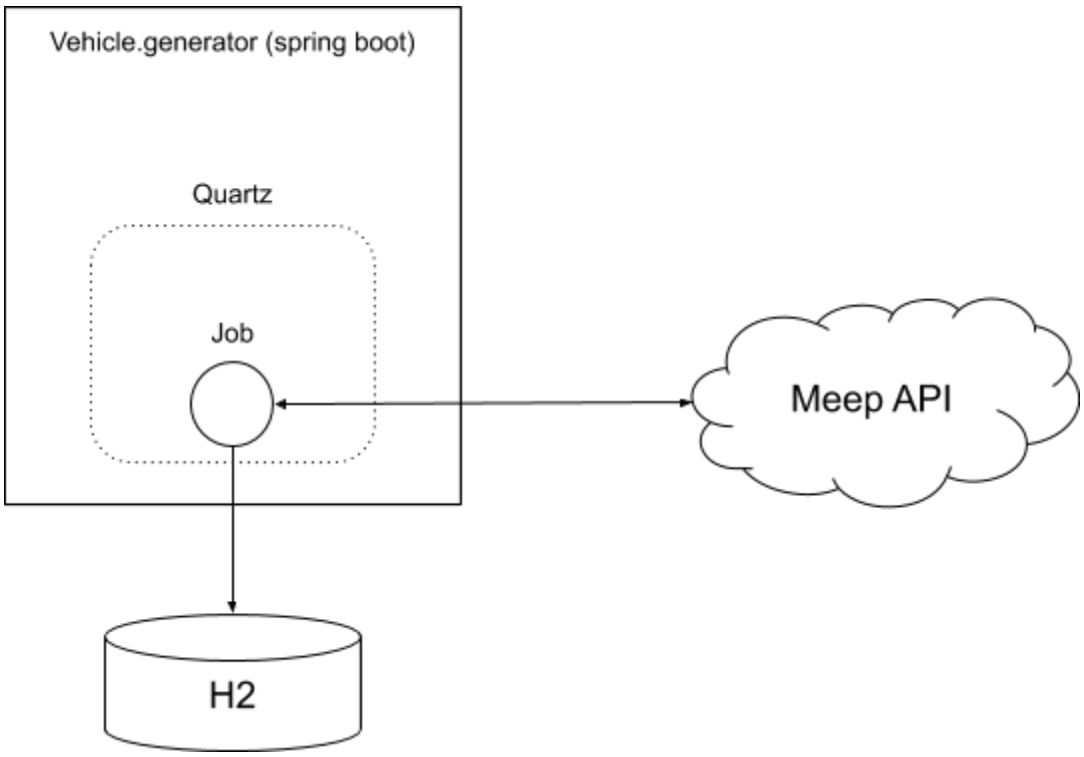
Para la planificación periódica de la lógica de actualización de vehículos se utilizó Quartz.

El endpoint de Meep para obtener los vehículos se hace con RestTemplate.

Se usa una expresión Cron para definir la frecuencia de ejecución del Job. Esta parametrización se hace en el application.properties. Lo mismo ocurre con los parámetros usados en la petición al endpoint de vehículos, se parametriza en el application.properties.

Por simplicidad se ha decidido usar H2 para la persistencia.

En el siguiente diagrama se esboza la arquitectura general.



Esbozo de arquitectura

Diagrama de clase

A continuación se presenta el diagrama de clases mostrando las clases y métodos principales.

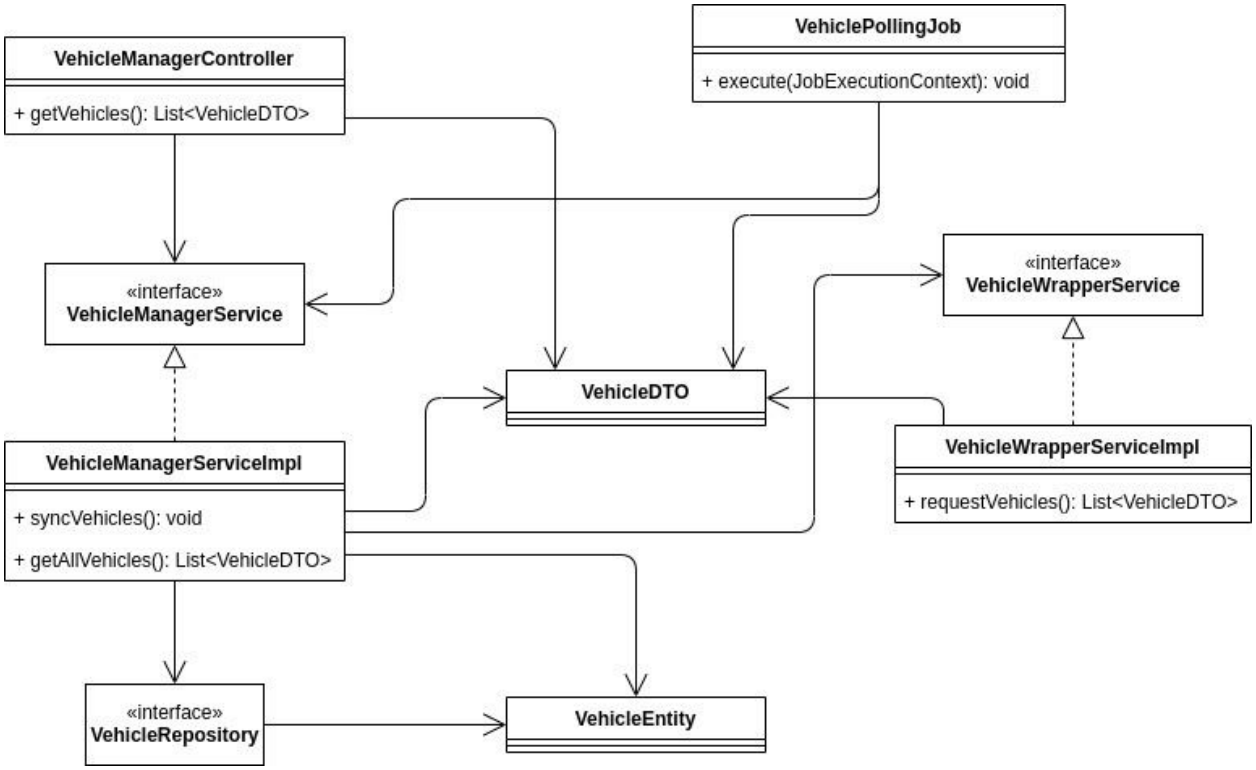


Diagrama de clases

La clase registrada en Quartz para ejecución es *VehiclePollingJob*. Luego, la clase que ejecuta la lógica principal de actualización es *VehicleManagerServiceImpl*. Solo se tiene un repositorio con el que se interactúa contra la base de datos, *VehicleRepository*. Por conveniencia aunque no es requisito se ha expuesto un endpoint en el controlador *VehicleController* con el cual se pueden consultar los vehículos que se tienen hasta el momento.

Diagrama de secuencia

Se presenta el diagrama de secuencia asociado al caso de uso de actualización de la información de vehículos.

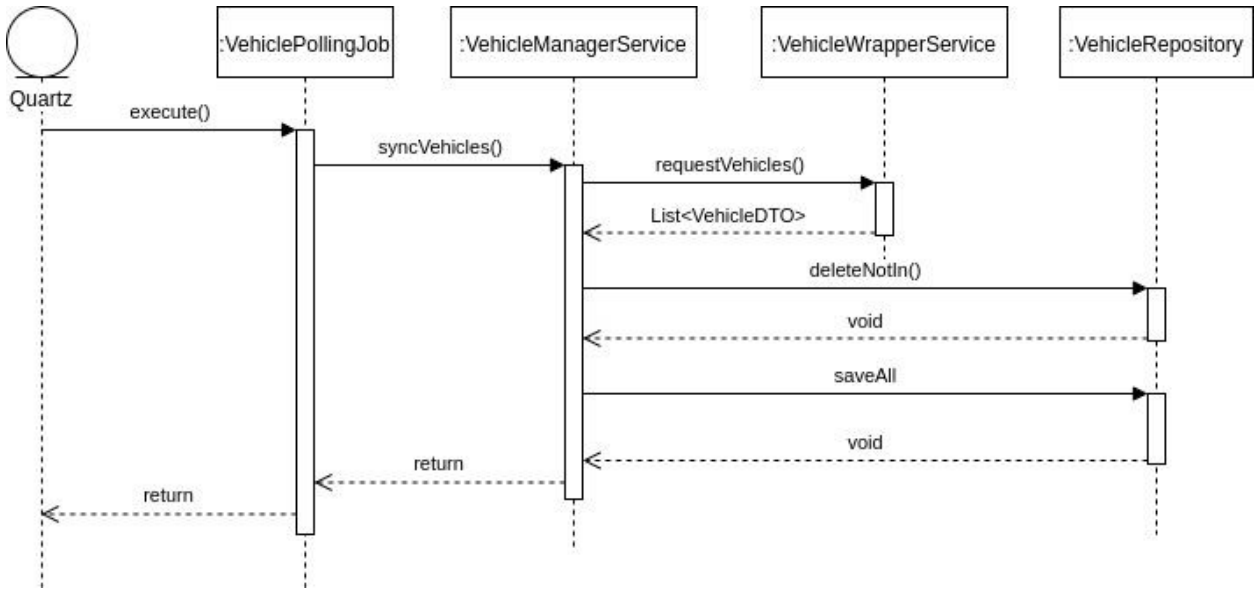


Diagrama de secuencia para algoritmo principal

El caso de la actualización de vehículos es tratado como si lo obtenido por el endpoint de Meep fuera un conjunto y lo que se tiene en la base de datos fuera otro conjunto. Luego lo que se busca es la intersección entre ambos. Esta es la definición que se le ha dado a el requisito de “detección de cambios,... nuevos vehículos disponibles y vehículos que antes estaban disponibles y ahora dejan de estar disponibles”.

Posibles problemas y mejoras

- Hacer llamadas paralelas, subdividiéndo el conjunto de vehículos obtenidos del endpoint. Esto elevaría la escalabilidad de la solución principalmente en caso de que el *boundingbox* sea más grande (más vehículos al tener una área más grande o también si se incluyen más proveedores).
- Usar una base de datos relacional o no relacional en lugar de H2
- Usar Kubernetes para tener más control en la orquestación de los contenedores, especialmente en el *vehicle.updater.* ¿quizás controlar las réplicas del MS?
- Es posible que otros proveedores tengan columnas que no están en el modelo relacional
    - Podría implicar cambiar el modelo relacional. ¿Tablas por proveedor?
    - Usar una BD no relacional ya que el esquema no es rígido. ¿Vale la pena?
- El algoritmo idealmente debería tardar menos que el intervalo de ejecución del Job.
- Sería conveniente si estuviera páginado el *endpoint* de Meep de forma que se pueda leer por partes si son muchos datos. Cada página se podría procesar de forma concurrente (como se describe al comienzo).
- Solo se ha aplicado PU a al método que contiene la lógica principal. Sería conveniente aplicar PUs al resto del código.

Despliegue de la solución

Despliegue regular

- Compilar con **`mvn clean install`**
- Ejecutar con **`java -jar`**

Despliegue con Docker

Por simplicidad no se ha creado Dockerfile sino que en su lugar se usa el plugin *jib-maven-plugin:dockerBuild* para compilar y construir la imagen en un paso. El comando sería:

```
./mvnw com.google.cloud.tools:jib-maven-plugin:dockerBuild
-Dimage=meep/vehicle-updater
```

Luego, iniciar el contenedor con:

```
docker run -p "8080:8080" meep/vehicle-updater
```

Se decide exponer el puerto 8080 en caso de querer consultar el endpoint creado para obtener los vehículos disponibles o si se desea hacer consultas a la BD de H2 a través de la consola.