DP1 2020-2021
Documento de Detalles de las Propuestas a A+

Mineral House Spa

https://github.com/gii-is-DP1/dp1-2020-g2-07

Miembros:

- JUAN MANUEL GARCIA CRIADO
- FERNANDO MIGUEL HIDALGO AGUILAR
- MIGUEL MOLINA RUBIO
- JAVIER RAMOS ARRONDO
- FRANCISCO JAVIER RODRIGUEZ ELENA

Tutor: Irene Bedilia Estrada

GRUPO G2-07

Versión 2.0

04/02/2020

Propuesta 1 (Fernando Miguel Hidalgo Aguilar): **Ejecución programada de algún método para calcular a cierta hora del día algún dato interesante (por ejemplo,el total de ventas realizadas en ese momento para mostrarlo en alguna vista).**
En el caso de nuestro proyecto, contamos con unos reportes económicos (Balances) que se generan automáticamente el día 1 de cada mes, tomando los datos del mes anterior.

Primero, se añade la etiqueta @EnableScheduling en la clase Petclinic Application.java, lo que nos permitirá usar @Schedule en cualquier parte del proyecto, en nuestro caso, dentro de Balance Service

```java
@Slf4j
@SpringBootApplication()
@EnableScheduling
public class PetclinicApplication {

    public static void main(String[] args) {
        SpringApplication.run(PetclinicApplication.class, args);
        log.info("Application has been deployed!");
    }

}
```

```java
//Comprueba si hoy es dia 1 del mes a las 00:00
@Scheduled(cron = "0 0 0 1 */1 *")
public void checkStmDay() {
    LocalDate day = getPrimerDiaMesPrevio();
    String month = day.getMonth().toString();
    String year = getAnyo(day);

    createBalance(day,month,year);
    log.info(String.format("Income Statement of previous month has been created"));
}
```

Podemos observar que con la etiqueta no es suficiente, es necesario especificar un momento en el tiempo. Se descartaron las opciones de Fixed Delay y Fixed Rate, debido a que su funcionamiento consiste en contar el tiempo desde la ejecución previa del método, además de utilizar milisegundos como unidad de medida
Se optó por tanto por el uso de cron, una expresión regular con la estructura:

```
second, minute, hour, day of month, month, day(s) of week
```

En nuestro caso, segundos, minutos y horas se establece a 0. Dia del mes es 1.
El **“*”** es un comodín que marca cualquiera, y **“/”** es un comodín que significa sumar. Por tanto, en los meses vemos que estamos en un mes cualquiera y le sumamos 1, para que compruebe el mes que viene a partir de ahora. El día de la semana no nos es relevante, por tanto también lo marcamos como cualquiera.

Las ventajas de este acercamiento al manejo de la fecha, es simplificar por mucho el código necesario para calcular el día 1, dejándolo todo más refinado en una sencilla etiqueta.

Además, garantiza la ejecución de la comprobación en todo momento, pues la etiqueta está constantemente funcionando en segundo plano, no depende de que el usuario realice ninguna acción en concreto, como por ejemplo, visitar el listado de balances

Bibliografía:

@Scheduled Annotation Spring
Cron Expressions
Run a task on 1st of every month Spring
Configure @Schedule

Propuesta 2 (Miguel Molina Rubio): **Estudio de en qué consisten las pruebas de mutación y aplicación de las pruebas de mutación para probar alguno de los servicios implementados en el proyecto.**

Las pruebas que hemos implementado funcionan teniendo en cuenta un resultado esperado (por ejemplo, esperamos que una función nos devuelva el valor 2), pero esto no asegura realmente la calidad de un test.

Una forma de comprobar la calidad es mediante pruebas de mutación, que consisten en que para una pieza de código que queremos probar, se crean mutantes, esto es, pequeñas alteraciones en el código. Si por ejemplo tenemos una función tal que f(x) = a*b, una posible mutación de f(x) sería g(x) = a/b.

La idea es que los tests buenos deben fallar (es decir, deben "matar a los mutantes"), lo que nos permite detectar defectos en el código. Una mutación muere cuando causa un fallo en la prueba, pero sobrevive si el mutante no puede afectar al comportamiento del test. Nuestra métrica para la prueba es el porcentaje de mutantes muertos (la prueba ha sido un éxito si el 100% de mutantes mueren).

Para implementar las pruebas de mutación, hemos utilizado el plugin de Pitest para Maven (http://pitest.org/quickstart/maven/). Para ello hemos añadido lo siguiente a nuestro pom.xml:

```
            <plugin>
                <groupId>org.pitest</groupId>
                <artifactId>pitest-maven</artifactId>
                <version>1.6.2</version>
                <configuration>
                    <targetTests>

<param>org.springframework.samples.petclinic.service.*</param>
                    </targetTests>
                </configuration>
            </plugin>
```

Para ejecutar las pruebas de mutación es tan sencillo como utilizar el siguiente comando:
`mvn org.pitest:pitest-maven:mutationCoverage`

Lo que nos genera un informe en la carpeta target/pit-reports (https://github.com/gii-is-DP1/dp1-2020-g2-07/tree/master/target/pit-reports/202102040009, abrir el index.html), en nuestro caso, cuando lo hemos ejecutado para los service hemos generado el siguiente informe:

Pit Test Coverage Report

Package Summary

org.springframework.samples.petclinic.service

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
12	75%	286/381	55%	117/214	78%	117/150

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
AdminService.java	86%	12/14	33%	2/6	50%	2/4
AuthoritiesService.java	0%	0/15	0%	0/3	0%	0/0
BalanceService.java	61%	40/66	46%	12/26	86%	12/14
BonoService.java	56%	9/16	44%	4/9	100%	4/4
CircuitoService.java	94%	29/31	59%	10/17	63%	10/16
CitaService.java	100%	10/10	67%	2/3	67%	2/3
ClienteService.java	100%	40/40	64%	14/22	64%	14/22
EmailService.java	0%	0/7	0%	0/4	0%	0/0
EmployeeService.java	78%	28/36	57%	13/23	87%	13/15
HorarioService.java	81%	64/79	63%	40/63	82%	40/49
SalaService.java	100%	22/22	90%	9/10	90%	9/10
UserService.java	71%	32/45	39%	11/28	85%	11/13

Report generated by PIT 1.6.2

Bibliografía:

https://www.baeldung.com/java-mutation-testing-with-pitest
https://www.adictosaltrabajo.com/2015/11/10/mutation-testing-con-pit/
http://pitest.org/quickstart/maven/